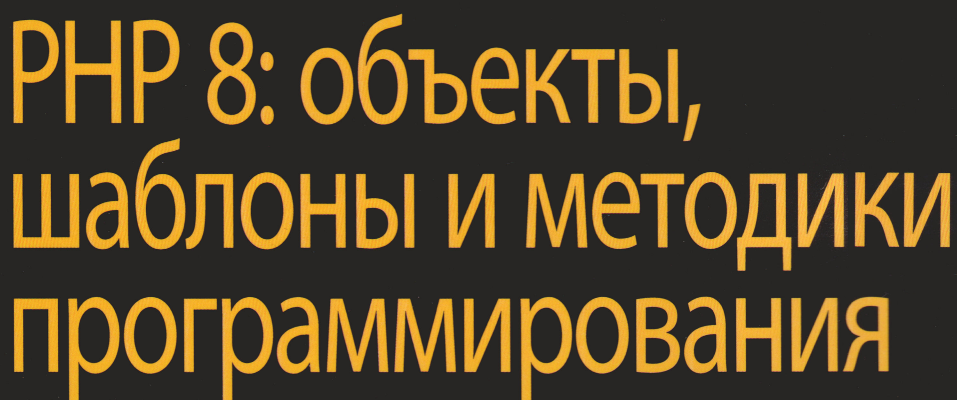


PHP 8: объекты, шаблоны и методики программирования

Изучение объектно-ориентированных средств PHP, проектных шаблонов и основных инструментальных средств разработки

Шестое издание

Мэтт Зандстра

Apress®
www.apress.com

PHP 8: объекты, шаблоны и методики программирования

Изучение объектно-ориентированных средств PHP, проектных шаблонов и основных инструментальных средств разработки

Шестое издание

PHP 8 Objects, Patterns, and Practice

Mastering OO Enhancements, Design Patterns, and Essential Development Tools

Sixth Edition

Matt Zandstra

Apress®

PHP 8: объекты, шаблоны и методики программирования

Изучение объектно-ориентированных средств PHP, проектных шаблонов и основных инструментальных средств разработки

Шестое издание

Мэтт Зандстра

Київ
Комп'ютерне видавництво
"ДІАЛЕКТИКА"
2021

УДК 004.738.5
З-27

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

Зандстра, М.

З-27 PHP 8: объекты, шаблоны и методики программирования, 6-е изд./Мэтт Зандстра; пер. с англ. И.В. Красикова. — Киев. : "Диалектика", 2021. — 866 с. : ил. — Парал. тит. англ.

ISBN 978-617-7987-35-1 (укр.)
ISBN 978-1-4842-6790-5 (англ.)

В книге рассматриваются методики объектно-ориентированного программирования на PHP и применение главных принципов проектирования программного обеспечения на основе классических проектных шаблонов, а также описываются инструментальные средства и нормы практики разработки, тестирования, непрерывной интеграции и развертывания надежного прикладного кода. Настоящее, шестое, издание книги полностью обновлено в соответствии с версией 8 языка PHP. В этой книге подробно описаны новые возможности PHP, такие как атрибуты и многочисленные усовершенствования в области объявления типов.

УДК 004.738.5

Все права защищены.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства APress, Berkeley, CA.

Authorized Russian translation of the English edition of *PHP 8 Objects, Patterns, and Practice*, 6th edition (ISBN 978-1-4842-6790-5), published by APress, Inc., Copyright © 2021 by Matt Zandstra.

This translation is published and sold by permission of APress, Inc., which owns or controls all rights to sell the same.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher. Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

ISBN 978-617-7987-35-1 (укр.)
ISBN 978-1-4842-6790-5 (англ.)

© "Диалектика", перевод, 2021
© 2021 by Matt Zandstra

Оглавление

Введение 21

Часть I. Объекты 23

Глава 1. Проектирование и сопровождение приложений на PHP 25
Глава 2. PHP и объекты 39
Глава 3. Азы объектов 49
Глава 4. Расширенные возможности 105
Глава 5. Средства для работы с объектами 181
Глава 6. Объекты и проектирование 239

Часть II. Проектные шаблоны 271

Глава 7. Назначение и применение проектных шаблонов 273
Глава 8. Некоторые принципы проектных шаблонов 287
Глава 9. Генерация объектов 307
Глава 10. Шаблоны для программирования гибких объектов 363
Глава 11. Выполнение задач и представление результатов 395
Глава 12. Шаблоны корпоративных приложений 451
Глава 13. Шаблоны баз данных 525

Часть III. Практика 591

Глава 14. Практика — хорошая (и плохая) 593
Глава 15. Стандарты PHP 607
Глава 16. Создание и использование компонентов PHP средствами Composer 631
Глава 17. Контроль версий средствами Git 649
Глава 18. Тестирование средствами PHPUnit 683
Глава 19. Автоматическое построение средствами Phing 727
Глава 20. Виртуальная машина Vagrant 759
Глава 21. Непрерывная интеграция 775
Глава 22. Объекты, проектные шаблоны и практика 809
Приложение А. Дополнительные источники информации 827
Приложение Б. Простой синтаксический анализатор 833
Предметный указатель 861

Содержание

Об авторе 19
О техническом рецензенте 19
Благодарности 20

Введение 21
Ждем ваших отзывов! 22

Часть I. Объекты 23

Глава 1. Проектирование и сопровождение приложений на PHP 25
Проблема 25
PHP и другие языки 28
Об этой книге 32
Объекты 32
Проектные шаблоны 33
Практика 33
Что нового в шестом издании книги 36
Резюме 36

Глава 2. PHP и объекты 39
Неожиданный успех объектов в PHP 39
Вначале был PHP/FI 39
Синтаксические удобства в версии PHP 3 40
Версия PHP 4 и незаметная революция 40
Изменения приняты: PHP 5 43
Заполнение пробела: PHP 7 45
PHP 8: продолжение консолидации 46
Дебаты сторонников и противников объектов 46
Резюме 47

Глава 3. Азы объектов 49
Классы и объекты 49
Первый класс 49
Несколько первых объектов 50
Установка свойств в классе 52
Работа с методами 55

Создание метода конструктора 58
Объявление свойств в конструкторе 60
Аргументы по умолчанию и именованные аргументы 61
Аргументы и типы 63
Примитивные типы данных 63
Дополнительные функции проверки типов 68
Объявления типов: объектные типы 69
Объявления типов: примитивные типы 72
Смешанные типы 75
Объединения 77
Типы, принимающие значение `null` 79
Объявление возвращаемого типа 79
Наследование 80
Проблема наследования 81
Использование наследования 87
Управление доступом к классам: модификаторы `public`, `private` и `protected` 94
Типизированные свойства 98
Резюме 103

Глава 4. Расширенные возможности 105

Статические методы и свойства 105
Константные свойства 111
Абстрактные классы 112
Интерфейсы 115
Трейты 118
Проблема, которую позволяют решить трейты 119
Определение и применение трейтов 120
Применение нескольких трейтов 121
Сочетание трейтов с интерфейсами 122
Устранение конфликтов имен методов с помощью ключевого слова `insteadof` 123
Назначение псевдонимов переопределенным методам трейта 125
Применение статических методов в трейтах 126
Доступ к свойствам класса-хоста 127
Определение абстрактных методов в трейтах 128
Изменение прав доступа к методам трейта 129
Позднее статическое связывание: ключевое слово `static` 130
Обработка ошибок 135
Исключения 137

Завершенные классы и методы 149
Внутренний класс `Error` 151
Работа с методами-перехватчиками 152
Определение методов-деструкторов 161
Копирование объектов с помощью метода `__clone()` 163
Определение строковых значений для объектов 167
Функции обратного вызова, анонимные функции и замыкания 169
Анонимные классы 177
Резюме 179

Глава 5. Средства для работы с объектами 181

PHP и пакеты 181
Пакеты и пространства имен в PHP 182
Автозагрузка 195
Функции для исследования классов и объектов 201
Поиск классов 203
Получение сведений об объекте или классе 204
Получение полностью квалифицированной строковой ссылки на класс 206
Получение информации о методах 207
Получение информации о свойствах 210
Получение сведений о наследовании 210
Вызов методов 211
Reflection API 214
Краткое введение в Reflection API 214
Время засучить рукава 216
Исследование класса 218
Исследование методов 221
Исследование аргументов методов 223
Использование интерфейса Reflection API 226
Атрибуты 232
Резюме 237

Глава 6. Объекты и проектирование 239

Определение программного проекта 239
Объектно-ориентированное и процедурное программирование 240
Ответственность 246
Связность 247
Тесная связь 247
Ортогональность 248

Выбор классов 248
Полиморфизм 250
Инкапсуляция 253
Забудьте, как это делается 254
Четыре явных признака недоброкачественного кода 256
Дублирование кода 256
Класс, который слишком много знал 256
На все руки мастер 257
Условные инструкции 257
Язык UML 258
Диаграммы классов 258
Диаграмма последовательностей 267
Резюме 270

Часть II. Проектные шаблоны 271

Глава 7. Назначение и применение проектных шаблонов 273

Что такое проектные шаблоны 274
Краткий обзор проектных шаблонов 277
Название 277
Постановка задачи 278
Решение 278
Следствия 279
Формат “Банды четырех” 279
Причины для применения проектных шаблонов 281
Шаблоны определяют задачи 281
Шаблоны определяют решения 281
Шаблоны не зависят от языка программирования 281
Шаблоны определяют словарь 282
Шаблоны проверяются и тестируются 283
Шаблоны предназначены для совместной работы 283
Шаблоны способствуют удачным проектам 284
Шаблоны применяются в распространенных каркасах 284
PHP и проектные шаблоны 284
Резюме 285

Глава 8. Некоторые принципы проектных шаблонов 287

Открытие шаблонов 287
Композиция и наследование 288
Проблема 289
Применение композиции 292

Развязка 296
Проблема 296
Ослабление связанности 298
Программируйте на основе интерфейса, а не его реализации 301
Меняющаяся концепция 303
Проблемы применения шаблонов 304
Шаблоны 305
Шаблоны для формирования объектов 305
Шаблоны для организации объектов и классов 305
Шаблоны, ориентированные на задачи 305
Промышленные шаблоны 305
Шаблоны баз данных 306
Резюме 306

Глава 9. Генерация объектов 307

Формирование объектов: задачи и решения 307
Шаблон Singleton 313
Проблема 314
Реализация 315
Следствия 318
Шаблон Factory Method 319
Проблема 319
Реализация 323
Следствия 326
Шаблон Abstract Factory 326
Проблема 326
Реализация 328
Следствия 331
Шаблон Prototype 333
Проблема 333
Реализация 335
Доведение до крайности: шаблон Service Locator 339
Блестящее одиночество: шаблон Dependency Injection 342
Проблема 342
Реализация 343
Dependency Injection и атрибуты 350
Следствия 360
Резюме 361

Глава 10. Шаблоны для программирования гибких объектов 363
Структурирование классов для повышения гибкости объектов 363
Шаблон Composite 364
Проблема 365
Реализация 368
Следствия 374
Резюме 379
Шаблон Decorator 380
Проблема 380
Реализация 383
Следствия 388
Шаблон Facade 389
Проблема 389
Реализация 392
Следствия 393
Резюме 394

Глава 11. Выполнение задач и представление результатов 395
Шаблон Interpreter 395
Проблема 396
Реализация 397
Трудности реализации шаблона Interpreter 408
Шаблон Strategy 409
Проблема 410
Реализация 411
Шаблон Observer 415
Реализация 418
Шаблон Visitor 426
Проблема 427
Реализация 429
Трудности реализации шаблона Visitor 435
Шаблон Command 436
Проблема 436
Реализация 437
Шаблон Null Object 443
Проблема 444
Реализация 447
Резюме 449

Глава 12. Шаблоны корпоративных приложений 451
Краткий обзор архитектуры 452
Шаблоны 452
Приложения и уровни 453
Нарушение правил с самого начала 457
Шаблон Registry 457
Реализация 460
Уровень представления данных 465
Шаблон Front Controller 466
Шаблон Application Controller 481
Шаблон Page Controller 499
Шаблоны Template View и View Helper 507
Уровень логики приложения 512
Шаблон Transaction Script 512
Шаблон Domain Model 518
Резюме 523

Глава 13. Шаблоны баз данных 525
Уровень хранения данных 525
Шаблон Data Mapper 526
Проблема 527
Реализация 527
Следствия 546
Шаблон Identity Map 548
Проблема 548
Реализация 550
Следствия 553
Шаблон Unit of Work 554
Проблема 554
Реализация 555
Следствия 560
Шаблон Lazy Load 561
Проблема 561
Реализация 562
Следствия 564
Шаблон Domain Object Factory 564
Проблема 565
Реализация 565
Следствия 567

Шаблон Identity Object 569
Проблема 570
Реализация 570
Следствия 577
Шаблоны Selection Factory и Update Factory 578
Проблема 578
Реализация 579
Следствия 584
Что теперь осталось от шаблона Data Mapper 585
Резюме 588

Часть III. Практика 591

Глава 14. Практика — хорошая (и плохая) 593
Не кодом единым 594
Снова изобретаем колесо 594
Ведите себя хорошо 598
Дайте коду крылья 599
Стандарты 601
Vagrant 602
Тестирование 603
Непрерывная интеграция 604
Резюме 605

Глава 15. Стандарты PHP 607
Зачем нужны стандарты 607
Рекомендованные стандарты PHP 609
Особенности рекомендованных стандартов PSR 610
На кого рассчитаны рекомендации стандартов PSR 611
Программирование в избранном стиле 612
Основные рекомендации стандарта PSR-1 по стилю программирования 613
Рекомендации стандарта PSR-12 по стилю программирования 616
Проверка и исправление исходного кода 623
Рекомендации стандарта PSR-4 по автозагрузке 626
Самые важные правила 627
Резюме 630

Глава 16. Создание и использование компонентов PHP средствами Composer 631

Что такое Composer 632
Установка Composer 632
Установка пакетов 633
Установка пакетов из командной строки 634
Версии пакетов 635
Поле `require-dev` 637
Composer и автозагрузка 638
Создание собственного пакета 639
Добавление сведений о пакете 640
Пакеты для конкретной платформы 641
Распространение пакетов через сайт Packagist 642
Работа с закрытыми пакетами 645
Резюме 647

Глава 17. Контроль версий средствами Git 649

Зачем нужен контроль версий 650
Установка Git 652
Использование онлайн-хранилища Git 652
Конфигурирование сервера Git 655
Создание удаленного хранилища 656
Подготовка хранилища для локальных пользователей 656
Предоставление доступа пользователям 657
Закрытие доступа к системной оболочке для пользователя `git` 658
Начало проекта 659
Клонирование хранилища 663
Обновление и фиксация изменений 664
Добавление и удаление файлов и каталогов 668
Добавление файла 669
Удаление файла 669
Добавление каталога 670
Удаление каталогов 671
Метка о выпуске 671
Ветвление проекта 672
Резюме 681

Глава 18. Тестирование средствами PHPUnit 683

Функциональные и модульные тесты 684
Тестирование вручную 685

Общее представление о PHPUnit 688
Создание контрольного примера 689
Методы утверждений 692
Тестирование исключений 693
Выполнение наборов тестов 695
Ограничения 696
Имитации и заглушки 699
Тесты достигают своей цели, когда завершаются неудачно 704
Написание веб-тестов 708
Рефакторинг кода веб-приложения для тестирования 709
Простое веб-тестирование 712
Введение в Selenium 715
Предупреждения относительно тестирования 723
Резюме 726

Глава 19. Автоматическое построение средствами Phing 727

Назначение Phing 728
Получение и установка Phing 729
Создание документа построения 730
Целевые задания 732
Свойства 735
Типы 745
Задания 752
Резюме 758

Глава 20. Виртуальная машина Vagrant 759

Задача 759
Простейшая установка 761
Выбор и установка образа операционной системы в Vagrant 761
Монтирование локальных каталогов на виртуальной машине Vagrant 764
Подготовка 766
Настройка веб-сервера 768
Настройка сервера баз данных MariaDB 769
Настройка имени хоста 771
Краткие итоги 773
Резюме 774

Глава 21. Непрерывная интеграция 775

Что такое непрерывная интеграция 775
Подготовка проекта к непрерывной интеграции 779

Установка сервера Jenkins 791
Установка модулей, подключаемых к серверу Jenkins 793
Установка открытого ключа доступа к Git 794
Установка проекта 796
Первое построение проекта 800
Настройка отчетов 801
Автоматический запуск процессов построения проектов 805
Резюме 807

Глава 22. Объекты, проектные шаблоны и практика 809

Объекты 809
Выбор 810
Инкапсуляция и делегирование 811
Развязка 811
Повторное использование кода 812
Эстетика 813
Проектные шаблоны 814
Преимущества проектных шаблонов 815
Шаблоны и принципы проектирования 816
Практика 819
Тестирование 820
Стандарты 820
Контроль версий 821
Автоматическое построение 821
Непрерывная интеграция 822
Что упущено из виду 822
Резюме 826

Приложение А. Дополнительные источники информации 827

Литература 827
Статьи 828
Сайты 829

Приложение Б. Простой синтаксический анализатор 833

Сканер 833
Синтаксический анализатор 843

Предметный указатель 861

Луизе, которая для меня важнее всего.

Об авторе

Мэтт Зандстра почти 20 лет проработал веб-программистом, консультантом по PHP и составителем технической документации. Он — автор книги *SAMS Teach Yourself PHP in 24 Hours*, 3-е издание которой вышло под названием *Освой самостоятельно PHP за 24 часа* в русском переводе (ИД "Вильямс", 2007), а также был одним из авторов книги *DHTML Unleashed* (SAMS Publishing, 1997). Кроме того, он писал статьи для *Linux Magazine*, *Zend.com*, *IBM DeveloperWorks* и *php|architect Magazine*.

Мэтт был старшим разработчиком и техническим руководителем в Yahoo! и LoveCrafts. Он работает консультантом в компании, предоставляющей консультационные услуги в области системной архитектуры и управления, и разработчиком на языках PHP и Java. Он также пишет фантастические рассказы.

О техническом рецензенте

Пол Трегоинг (Paul Tregoing) имеет почти 20-летний опыт разработки и сопровождения программного обеспечения в самых разных условиях эксплуатации. В течение пяти лет он работал старшим разработчиком в команде, ответственной за главную страницу веб-сайта компании Yahoo!, где и сформировал свой первый сценарий на PHP с помощью языка Perl. Ему довелось также работать в компаниях Bloomberg, Schlumberger и даже в Британском управлении по антарктическим исследованиям (British Antarctic Survey), где он имел удовольствие непосредственно общаться с тысячами пингвинов.

В настоящее время Пол Трегоинг работает внештатным инженером и выполняет мелкие и крупные проекты по созданию многоуровневых веб-приложений для различных клиентов на основе PHP, JavaScript и многих других технологий. Пол — ненасытный читатель научно-фантастической и фэнтезийной литературы и не скрывает своих честолюбивых намерений: написать в этих жанрах что-нибудь свое в ближайшем будущем. Он проживает в Кембридже (Великобритания) с женой и детьми.

Благодарности

Как всегда, хочу выразить признательность всем, кто работал над этим изданием книги. Но я должен упомянуть и тех, кто работал над всеми предыдущими изданиями.

Некоторые из основополагающих концепций этой книги были опробованы мною на конференции в Брайтоне, где мы все собрались, чтобы познакомиться с замечательными возможностями версии PHP 5. Выражаю благодарность организатору конференции Энди Бадду (Andy Budd) и всему активному сообществу разработчиков Брайтона. Благодарю также Джесси Вайт-Синис (Jessey White-Cinis), которая познакомила меня на конференции с Мартином Штрайхером (Martin Streicher) из издательства Apress.

Как и прежде, сотрудники издательства Apress оказывали мне неоценимую поддержку, высказывали ценные замечания и обеспечивали всяческое содействие. Мне очень посчастливилось работать с ними и использовать их профессиональный опыт и знания.

Мне также очень повезло, что у меня есть такой товарищ и коллега, как Пол Трегоинг, который взялся рецензировать настоящее издание. Тот факт, что сам язык PHP постоянно активно развивается (в том числе во время написания этой книги), требует постоянной бдительности и внимания. Примеры кода, которые были совершенно корректными в ранних версиях, становятся неверными в силу быстрой эволюции языка. Настоящее издание выгодно отличается в лучшую сторону благодаря знаниям, опыту, проницательности и вниманию Пола к деталям, за что я искренне ему признателен!

Выражаю благодарность и безграничную любовь своей жене Луизе, а поскольку написание этой книги совпало с тремя пандемическими локдаунами — то и моим детям, Холли и Джейку, за то, что они периодически отвлекали меня от писательской деятельности и вносили необходимую разрядку в мою работу в моем офисе (на углу кухонного стола).

Благодарю Стивена Мецкера (Steven Metsker) за любезное разрешение реализовать на PHP упрощенную версию API синтаксического анализатора, который он представил в своей книге *Building Parsers with Java* (Addison-Wesley Professional, 2001).

Я обычно работаю под музыку и в предыдущих изданиях этой книги упомянул о великом диджее Джоне Пиле (John Peel), поборнике всего подпольного и эклектичного в авангардной музыке. А в работе над этим изданием я пользовался фонограммой из программы современной музыки Late Junction (Поздний переход), которую постоянно крутили на радиостанции BBC Radio 3. Благодарю их за нестандартный подход к подбору музыки.

Введение

Когда меня впервые посетила мысль написать эту книгу, объектно-ориентированными средствами разработки в PHP пользовались лишь избранные программисты. Но со временем мы стали свидетелями не только неуклонного роста популярности объектно-ориентированных средств языка PHP, но и развития каркасов приложений. Безусловно, каркасы чрезвычайно полезны. Они составляют основу многих (а ныне, вероятно, большинства) веб-приложений. Более того, каркасы зачастую в точности демонстрируют основные подходы к проектированию программного обеспечения, рассматриваемые в этой книге.

Но именно в этом кроется для разработчиков определенная опасность, как, впрочем, и в применении любого полезного интерфейса API. Опасность состоит в том, что пользовательские приложения могут стать зависимыми от некоего стороннего гуру, который никак не удосужится внести исправления в созданный им каркас или по собственной прихоти внес в него существенные коррективы. На самом деле подобная точка зрения часто приводит к тому, что разработчики программного обеспечения рассматривают внутреннюю структуру каркаса как некий "черный ящик", а свою часть работы считают не более чем небольшой надстройкой, поставленной на вершину огромной непознанной инфраструктуры.

Несмотря на то что я отношусь к закоренелым изобретателям велосипедов, суть моих доводов состоит вовсе не в том, что я призываю вас полностью отказаться от своих любимых каркасов и начать создавать приложения на основе проектного шаблона MVC "с нуля", по крайней мере изредка. Напротив, разработчики должны ясно понимать задачи, которые призван решать конкретный каркас, а также разбираться в методиках, применяемых для их решения. Разработчики должны оценивать каждый каркас не только с точки зрения предлагаемых им функциональных возможностей, но и с точки зрения проектных решений, которые были приняты их создателями, а также судить, насколько качественно они реализованы. И когда позволяют обстоятельства, разработчики должны двигаться дальше, создавая сначала скромные, но конкретные приложения, а со временем — собственные библиотеки повторно используемого кода.

Надеюсь, что эта книга поможет разработчикам приложений на PHP выработать навыки проектно-ориентированного мышления и реализо-

вать их на собственных программных платформах и библиотеках. В ней описаны также некоторые концептуальные средства, которые пригодятся вам, читатель, когда наступит время действовать в одиночку и брать всю ответственность на себя.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`
WWW: `http://www.dialektika.com`

Часть I

Объекты

ГЛАВА 1

Проектирование и сопровождение приложений на PHP

Версия PHP 5.0 была выпущена в июле 2004 года. В ней был внедрен целый ряд радикальных усовершенствований, и самым главным среди них, вероятно, стала коренным образом улучшенная поддержка объектно-ориентированного программирования (ООП). Именно это обстоятельство и вызвало повышенный интерес к объектам и методике ООП в среде разработчиков приложений на PHP. На самом деле это был новый этап развития процесса, начавшегося благодаря выпуску версии 4 языка PHP, в которой ООП впервые стало реальностью.

В этой главе рассмотрен ряд задач, для решения которых требуется методика ООП. В ней вкратце изложена эволюция проектных шаблонов и связанных с ними норм практики программирования. Кроме того, здесь в общих чертах будут обозначены следующие темы, освещаемые в данной книге.

- *Эволюция катастрофы*. Проект не удался.
- *Проектирование и PHP*. Каким образом методика ООП укоренилась в сообществе разработчиков приложений на PHP.
- *Эта книга*. Объекты, шаблоны, практика программирования.

Проблема

Проблема в том, что PHP — очень простой язык. Он искушает вас попробовать реализовать свои идеи и радует хорошими результатами. Большую часть написанного кода на PHP можно встроить непосредственно в разметку веб-страниц, потому что для этого в PHP предусмотрена соответствующая поддержка. Вводя дополнительные функции (например, код доступа к базе данных) в файлы, которые можно перенести с одной страницы на другую, вы не успеете оглянуться, как получите рабочее веб-приложение.

И, тем не менее, вы уже стоите на пути к краху. Конечно, вы этого не осознаете, потому что ваш сайт выглядит потрясающе. Он прекрасно работает, заказчики довольны, а пользователи тратят деньги.

Трудности возникают, когда вы возвращаетесь к исходному коду, чтобы начать новый этап разработки. Теперь ваша команда увеличилась, пользователей стало больше да и бюджет вырос. Но, если не принять меры, все пойдет прахом. Образно говоря, все выглядит так, как будто ваш проект был отравлен.

Ваш новый программист изо всех сил старается понять код, который вам кажется простым и естественным, хотя, возможно, местами слегка запутанным. И этому новому программисту требуется больше времени, чем вы рассчитывали, чтобы войти в курс дела и стать полноправным членом команды. На простое изменение, которое вы рассчитывали сделать за один день, уходит три дня, потому что вы обнаруживаете, что в результате нужно обновить порядка двадцати веб-страниц.

Один из программистов сохраняет свою версию файла поверх тех серьезных изменений, которые вы внесли в тот же код немного раньше. Потеря обнаруживается только через три дня, к тому времени как вы изменили собственную локальную копию файла. Еще один день уходит на то, чтобы разобраться в этом беспорядке, а между тем без дела сидит третий программист, который также работал с этим файлом.

Ваше веб-приложение пользуется популярностью, и поэтому вам нужно перенести его исходный код на новый сервер. Устанавливать на нем свой проект приходится вручную, и в этот момент вы обнаруживаете, что пути к файлам, имена баз данных и пароли жестко закодированы во многих исходных файлах. В результате вы останавливаете работу своего веб-сайта на время переноса исходного кода, потому что не хотите затереть изменения в конфигурации, которых требует такой перенос. Работа, которую планировалось выполнить за два часа, в итоге растягивается на восемь часов, поскольку обнаружилось, что кто-то умный задействовал модуль `ModRewrite` веб-сервера Apache, и теперь для нормальной работы веб-приложения требуется, чтобы этот модуль функционировал на сервере и был правильно настроен.

В конечном счете вы успешно преодолеваете второй этап разработки. И полтора дня все идет хорошо. Первое сообщение об ошибке приходит в тот момент, когда вы собираетесь уходить с работы домой. Еще через минуту звонит заказчик с жалобой. Его сообщение об ошибке напоминает предыдущее, но в результате более тщательного анализа обнаруживается,

что это другая ошибка, которая вызывает схожее поведение. Вы вспоминаете о простом изменении в начале данного этапа, которое потребовало серьезно модифицировать остальную часть проекта.

И тогда вы понимаете, что модифицировано было не все. Это произошло либо потому, что некоторые моменты были упущены в самом начале, либо потому, что изменения в проблемных файлах были затерты в процессе объединения. В страшной спешке вы вносите изменения, необходимые для исправления ошибок. Вы слишком спешите и не тестируете внесенные изменения. Ведь это же простые операции копирования и вставки, что тут может случиться?

Придя на работу на следующее утро, вы выясняете, что модуль корзины для покупок не работал всю ночь. Оказалось, что, внося вчера изменения в последнюю минуту, вы пропустили открывающие кавычки, в результате чего код стал неработоспособным. И пока вы спали, потенциальные клиенты из других часовых поясов бодрствовали и были готовы потратить деньги в вашем интернет-магазине. Вы исправляете ошибку, успокаиваете заказчика и мобилизуете команду на “пожарные” работы в течение еще одного дня.

Подобная история о ежедневных буднях программистов может показаться преувеличением, но все это мне случалось наблюдать неоднократно. Многие проекты на PHP начинались с малого, а затем превращались в настоящих монстров!

В данном проекте логика приложения содержится также на уровне представления данных, и поэтому дублирование происходит уже в коде запросов к базе данных, проверке аутентификации, обработке форм, причем этот код копируется с одной страницы на другую. Всякий раз, когда требуется внести коррективы в один из таких блоков кода, это приходится делать везде, где присутствует такой код, иначе неминуемо возникнет ошибка.

Отсутствие документации затрудняет понимание исходного кода, а недостаточность тестирования оставляет скрытые дефекты необнаруженными вплоть до момента развертывания приложения. Изменение основного направления деятельности заказчика часто означает, что в результате модификации прикладной код меняет свое первоначальное назначение и в конце концов начинает выполнять задачи, для которых он изначально вообще не был предназначен. А поскольку такой код, как правило, разрабатывался и развивался в качестве единой гремучей смеси, в которой было много чего намешано, то очень трудно, или даже невозможно, перестро-

иться и переписать отдельные его фрагменты, чтобы он соответствовал новой цели.

Но все это не так уж и плохо, если вы — независимый консультант по PHP. Анализ и исправление подобных систем позволят вам покупать дорогие напитки и наборы DVD в течение полугода или даже дольше. А если серьезно, то проблемы подобного рода как раз и отличают удачную коммерческую деятельность от неудачной.

PHP и другие языки

Феноменальная популярность языка PHP означает, что он был основательно протестирован во всех сферах своего применения еще на начальном этапе своего развития. Как поясняется в следующей главе, язык PHP начинался как набор макрокоманд для управления персональными веб-страницами. С появлением версии PHP 3, а в значительной степени — PHP 4, этот язык быстро стал популярным и мощным инструментом для создания веб-сайтов крупных коммерческих компаний. Но во многих случаях область его применения ограничивалась разработкой сценариев и средств управления проектами. В некоторых кругах специалистов PHP заслужил несправедливую репутацию языка для любителей, который лучше всего приспособлен для задач представления данных.

Примерно в то же время, когда наступило новое тысячелетие, в других сообществах программистов стали распространяться новые идеи. Интерес к объектно-ориентированному проектированию всколыхнул сообщество программирующих на Java. Вам такой интерес может показаться излишним, поскольку Java и так является объектно-ориентированным языком. Но ведь Java задает лишь рациональное зерно, которым нужно еще научиться правильно пользоваться, поскольку применение в программах классов и объектов само по себе не определяет конкретный подход к проектированию.

Понятие проектного шаблона как способа описания задачи вместе с сутью ее решения впервые обсуждалось еще в 1970-х годах. Пожалуй, можно считать удачей, что первоначально эта идея возникла в области строительства и архитектуры, а не в вычислительной технике, появившись в работе Кристофера Александера (Christopher Alexander) *A Pattern Language* в 1977 году. В начале 1990-х годов приверженцы ООП стали применять аналогичные методики для определения и описания задач

разработки программного обеспечения. Основополагающая книга по проектным шаблонам, *Design Patterns: Elements of Reusable Object-Oriented Software*[1], опубликованная в 1995 году под авторством "Банды четырех", незаменима и по сей день. Шаблоны, которые в ней описаны, обязательны для каждого, кто делает первые шаги в данной области. Именно поэтому большинство шаблонов, описываемых в данной книге, взято из этого фундаментального труда.

В интерфейсах API языка Java изначально применяются многие основные шаблоны, но до конца 1990-х годов проектные шаблоны так и не были полностью осмыслены сообществом программистов. Книги по проектным шаблонам быстро заполонили отделы компьютерной литературы в книжных магазинах, и на форумах программистов появились первые признаки горячей полемики со словами похвалы или неодобрения.

Возможно, вы думаете, что шаблоны — это эффективное средство передачи специальных знаний предметной области или, наоборот, "мыльный пузырь" (какой точки зрения придерживаюсь лично я, видно из названия данной книги). Каким бы ни было ваше мнение, трудно отрицать, что подход к разработке программного обеспечения, который поощряется в шаблонах, полезен сам по себе.

Не менее популярными для обсуждения стали и родственные темы. К их числу относится методика экстремального программирования (Extreme Programming — XP), одним из авторов которой является Кент Бек (Kent Beck)[2]. Экстремальное программирование — это подход к проектированию, который направлен на гибкое, проектно-ориентированное, очень тщательное планирование и выполнение. Один из главных его принципов настаивает на том, что ключевым фактором для успешного выполнения проекта является тестирование. Тесты должны быть автоматическими и выполняться часто; и желательно, чтобы они были разработаны до написания самого кода.

Еще одно требование методики экстремального программирования состоит в том, что проекты должны быть разбиты на небольшие (попросту — мелкие) повторяющиеся стадии. А код и требования к нему должны постоянно анализироваться. Архитектура и проект должны быть предме-

---

[1] Гамма, Э., Хелм, Р., Джонсон, Р., Влиссидес, Д. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* : Пер. с англ. — Изд-во "Питер", 2007.

[2] Бек, К. *Экстремальное программирование* : Пер. с англ. — Изд-во "Питер", 2007.

том постоянного совместного обсуждения, что ведет к частому пересмотру разрабатываемого кода.

Методика экстремального программирования стала воинствующим крылом в движении сторонников проектирования программного обеспечения. Что касается умеренной тенденции, то она представлена в одной из лучших книг по программированию, которые я когда-либо читал: *The Pragmatic Programmer* (Andrew Hunt, David Thomas, издательство Addison-Wesley Professional, 1999)[3].

Некоторые считают, что методика экстремального программирования была в какой-то степени культовым явлением, но за два десятилетия практики объектно-ориентированного программирования она достигла высочайшего уровня, а ее принципы широко использовались и заимствовались. В качестве мощного дополнения к шаблонам был использован процесс реорганизации кода, называемый *рефакторингом*. Рефакторинг развивался с 1980-х годов, но только в 1999 году он был кодифицирован в каталоге шаблонов рефакторинга, который был опубликован Мартином Фаулером (Martin Fowler) в книге *Refactoring: Improving the Design of Existing Code*[4] и определил данную область проектирования программного обеспечения.

С развитием и ростом популярности методики экстремального программирования и проектных шаблонов тестирование также стало злободневным вопросом. Особое значение автоматических тестов еще более усилилось с появлением мощной тестовой платформы JUnit, которая стала главным инструментом в арсенале средств программистов на Java. Основополагающая статья *Test Infected: Programmers Love Writing Tests* (Kent Beck, Erich Gamma; `http://junit.sourceforge.net/doc/testinfected/testing.htm`) стала превосходным введением в этот предмет и до сих пор не потеряла своей актуальности.

Примерно в то же время вышла более эффективная версия PHP 4 с усиленной поддержкой объектов. Благодаря этим усовершенствованиям появилась возможность создавать полностью объектно-ориентированные приложения. Программисты воспользовались этой возможностью, к некоторому удивлению создателей ядра Zend Зеэва Сураски (Zeev Suraski) и Энди Гутманса (Andi Gutmans), которые присоединились к Расмусу Лер-

---

[3] Хант, Э., Томас, Д. *Программист-прагматик: 2-е юбилейное изд.*,: Пер. с англ. — Изд-во “Диалектика”, 2020.

[4] Фаулер, М., Бек, К., Брант, Д., Опдайк, У., Робертс, Д. *Рефакторинг: улучшение проекта существующего кода* : Пер. с англ. — Изд-во “Диалектика”, 2017.

дорфу (Rasmus Lerdorf) для разработки PHP. Как поясняется в следующей главе, поддержка объектов в PHP оказалась далеко не идеальной, но при строгой дисциплине и аккуратном применении синтаксиса на PHP можно было все же писать объектно-ориентированные программы.

Тем не менее неприятности, подобные описанной в начале этой главы, случались часто. Культура проектирования была еще недостаточно развита, и в книгах по PHP о ней едва ли упоминалось. Но в Интернете интерес к этому предмету был очевиден. Леон Аткинсон (Leon Atkinson) написал статью о PHP и проектных шаблонах для Zend в 2001 году, а Гарри Фойкс (Harry Fuecks) завел в 2002 году журнал по адресу `www.phppatterns.com` (он уже давно прекратил свое существование, хотя сайт по указанному адресу по-прежнему действует). В конечном итоге стали появляться проекты на основе шаблонов (например, BinaryCloud), а также инструментальные средства для автоматического тестирования и документирования разрабатываемых приложений.

Выход первой бета-версии PHP 5 в 2003 году обеспечил будущее PHP в качестве языка для объектно-ориентированного программирования. Процессор Zend Engine 2 обеспечил существенно улучшенную поддержку объектов. И, что не менее важно, его появление стало сигналом, что объекты в частности и методика объектно-ориентированного программирования вообще могут служить основанием для любого проекта на PHP.

С годами версия PHP 5 продолжала развиваться и совершенствоваться. В ней появились такие важные средства, как поддержка пространств имен и механизма замыканий. За это время версия PHP 5 завоевала репутацию надежного и самого лучшего средства для разработки серверных веб-приложений.

Эта тенденция была продолжена в версии PHP 7, выпущенной в декабре 2015 года. В частности, в этой версии поддерживаются объявления типов параметров и возвращаемых типов — возможности, внедрения которых многие годы настоятельно требовали многие разработчики, о чем упоминалось в том числе на страницах предыдущих изданий данной книги. Есть много других возможностей и улучшений, включая анонимные классы, улучшенное использование памяти и повышение скорости. С годами, с точки зрения объектно-ориентированного разработчика, язык постоянно становится все более надежным, ясным и приятным в работе.

В декабре 2020 года, почти через пять лет после выпуска PHP 7, был подготовлен к выпуску PHP 8. Хотя некоторые детали реализации могут измениться (и изменились за время написания этой книги), основные его

функциональные возможности уже доступны и подробно описаны в этой книге. Они включают улучшения объявления типов, оптимизированное назначение свойств и многие другие новые функции.

Об этой книге

Эта книга не является попыткой открыть что-то новое в области объектно-ориентированного программирования, и в этом отношении она просто "стоит на плечах гигантов". Ее цель — исследовать в контексте PHP некоторые устоявшиеся принципы проектирования и основные шаблоны проектирования (особенно те, которые упоминаются в книге *Design Patterns* — классическом труде "Банды четырех"). В конце книги будет предпринята попытка выйти за пределы строгих ограничений прикладного кода, чтобы рассмотреть инструментальные средства и методики, которые могут помочь в работе над проектом. За исключением этого введения и краткого заключения, данная книга разделена на три основные части: "Объекты", "Проектные шаблоны" и "Практика".

Объекты

Часть I начинается с краткой истории развития PHP и объектов — с их превращения из дополнительной возможности в версии PHP 3 в основную начиная с версии PHP 5. Вы можете быть опытным программистом и свободно писать программы на PHP, но при этом очень мало знать или почти ничего не знать об объектах. Именно по этой причине данная книга начинается с основных принципов, — чтобы объяснить, что такое объекты, классы и наследование. Даже на этой начальной стадии в данной части книги рассматриваются некоторые усовершенствования объектов, появившиеся в версиях PHP 5, PHP 7 и PHP 8.

После основ объекты будут рассмотрены более углубленно в ходе исследования более развитых объектно-ориентированных возможностей PHP. Отдельная глава в этой части книги посвящена инструментальным средствам, которые предусмотрены в PHP для работы с объектами и классами.

Но знать, как объявить класс и как его использовать для создания экземпляра объекта, еще недостаточно. Сначала необходимо правильно выбрать участников для разрабатываемой системы и определить оптимальные способы их взаимодействия. Описать и усвоить эти варианты выбора намного

труднее, чем простые факты об инструментальные средствах и синтаксисе объектов. Данная часть завершается введением в объектно-ориентированное проектирование на PHP.

Проектные шаблоны

Шаблон описывает задачу, поставленную в проекте программного обеспечения, предоставляя саму суть решения. "Решение" здесь не является частью кода, которую можно найти в справочнике, вырезать и вставить (хотя на самом деле справочники — превосходные ресурсы для программистов). В проектном шаблоне описывается подход, который можно использовать для решения поставленной задачи. К этому может прилагаться пример реализации, но он менее важен, чем концепция, которую он призван иллюстрировать.

Часть II начинается с определения проектных шаблонов и описания их структуры. В ней рассматриваются также некоторые причины их широкой распространенности.

При создании шаблонов обычно выдвигаются некоторые основополагающие принципы проектирования, которых следует придерживаться в процессе разработки всего приложения. Уяснение этого факта поможет лучше понять причины создания шаблона, который затем можно применять при создании любой программы. Некоторые из этих принципов обсуждаются в данной части книги, где представлен также унифицированный язык моделирования (Unified Modeling Language — UML) — платформенно-независимый способ описания классов и их взаимодействия.

Несмотря на то что данная книга не является справочником по проектным шаблонам, в ней все же рассматриваются некоторые из самых известных и полезных шаблонов. Сначала описывается задача или проблема, для решения которой предназначен каждый шаблон, а затем анализируется ее решение и демонстрируется пример его реализации на PHP.

Практика

Даже самое гармоничное архитектурное сооружение разрушится, если не обращаться с ним должным образом. В части III представлены инструментальные средства, с помощью которых можно создать структуру, способную обеспечить удачное завершение проекта. Если в остальных частях этой книги речь идет о практике проектирования и программирования, то

в этой части — о практике сопровождения прикладного кода. Рассматриваемые здесь инструментальные средства позволяют создавать поддерживающую инфраструктуру проекта, помогая обнаруживать ошибки по мере их проявления, способствуя сотрудничеству программистов и обеспечивая простоту установки и ясность прикладного кода.

Ранее вкратце упоминалось об эффективности автоматических тестов. Данная часть начинается со вступительной главы, в которой представлен краткий обзор насущных задач и решений в данной области разработки программного обеспечения.

Многие программисты поддаются искушению делать все самостоятельно. А диспетчер зависимостей Composer совместно с главным хранилищем пакетов Packagist обеспечивает доступ к тысячам пакетов с управляемыми зависимостями, из которых легко составляются целые проекты. В этой части обсуждаются достоинства и недостатки самостоятельной реализации в сравнении с развертыванием пакетов средствами Composer. Здесь, в частности, описывается применяемый в Composer механизм установки, упрощающий процесс развертывания пакетов вплоть до выполнения единственной команды.

Код — это продукт коллективного труда. С одной стороны, этот факт приносит внутреннее удовлетворение, а с другой — может превратить все в сущий кошмар. Система контроля версий Git предоставляет группе программистов возможность работать совместно над одним и тем же кодом, не рискуя затереть результаты работы друг друга. Она позволяет получать моментальные снимки проекта на любой стадии его разработки, видеть, кто и какие изменения вносил, а также разбивать проект на независимые ветки, которые затем можно объединить. В системе Git сохраняется ежедневное состояние проекта.

Когда разработчики совместно работают над приложениями или библиотеками, они нередко вносят разные условные обозначения и стили оформления кода в общий проект. И хотя в таком явлении нет ничего дурного, оно подрывает основы организации взаимодействия. От понятий *совместимости* и *соответствия* нередко бросает в дрожь, но непреложным является тот факт, что творческая инициатива в Интернете опирается все же на стандарты. Соблюдая определенные соглашения, можно свободно экспериментировать в невообразимо обширной "песочнице". Поэтому в новой главе этой части исследуются стандарты PHP, польза от них для разработчиков и причины, по которым они должны строго соблюдать *соответствие* стандартам.

Существуют две неизбежные проблемы. Во-первых, ошибки часто повторяются в одном и том же фрагменте кода, из-за чего некоторые рабочие дни проходят с ощущением повторения уже однажды пройденного. И во-вторых, улучшения часто портят столько же или даже больше, чем исправляют. Автоматическое тестирование помогает решить обе эти проблемы, обеспечивая систему раннего предупреждения об ошибках в прикладном коде. В этой части книги представлена PHPUnit — эффективная реализация так называемой тестовой платформы xUnit, первоначально предназначавшейся для Smalltalk, а теперь приспособленной для многих языков, особенно для Java. Здесь рассматриваются функции PHPUnit в частности и преимущества тестирования вообще, а также цена, которую приходится за него платить.

Для приложений характерен беспорядок. Им могут понадобиться файлы, которые необходимо скопировать в нестандартные места, базы данных или изменение конфигурации сервера. Короче говоря, установка приложений требует *немалых* хлопот. Утилита Phing представляет собой точную переносимую версию утилиты Ant, применяемой в проектах на Java. Обе утилиты, Phing и Ant, интерпретируют файл построения и обрабатывают исходные файлы любым заданным вами способом. Чаще всего их нужно скопировать из исходного каталога в различные системные каталоги. Но если ваши потребности возрастут, то утилита Phing сможет легко их удовлетворить.

В некоторых компаниях строго придерживаются вполне определенных платформ разработки, но зачастую команды разработчиков работают в самых разных операционных системах. Например, подрядчики могут быть оснащены портативными персональными компьютерами (как у Пола Трегоинга, технического рецензента пятого и настоящего изданий книги), а некоторые члены команд разработчиков бесконечно пропагандируют (это дело религии) свой излюбленный дистрибутив Linux (как, например, я — версию Fedora). Многие из них мечтают приобрести новый и привлекательный MacBook Air, чтобы произвести впечатление на окружающих во время деловых встреч за чашкой кофе или производственных совещаний (хотя это совсем не делает их последователями моды современной молодежи). И на всех этих платформах может запускаться комплект серверного программного обеспечения LAMP с разной степенью трудности. Но в идеальном случае разработчики должны отлаживать свой код в той среде, которая очень похожа на конечную производственную систему. Поэтому здесь рассматривается инструментальное средство Vagrant, в котором при-

меняется виртуализация для того, чтобы члены команды разработчиков могли работать на своих излюбленных платформах разработки, но выполнять прикладной код общего проекта в системе, похожей на производственную.

Тестировать и создавать проект — это, конечно, хорошо, но для того чтобы извлечь из этого пользу, необходимо установить и непрерывно выполнять наборы тестов. Если не автоматизировать процесс создания и тестирования проекта, все очень быстро пойдет прахом. Поэтому здесь рассматривается ряд средств и методик, которые совместно называются *непрерывной интеграцией* и призваны помочь справиться с подобной задачей.

Что нового в шестом издании книги

PHP — это живой язык, и все его средства постоянно обновляются, изменяются и дополняются. Поэтому новое издание книги было тщательно пересмотрено и основательно обновлено, чтобы описать все новые изменения и возможности PHP.

В настоящем издании описываются новые возможности PHP, такие как атрибуты и многочисленные усовершенствования в области объявления типов. В соответствующих примерах используются средства PHP 8 (там, где это уместно), поэтому имейте в виду, что вам часто придется выполнять исходный код в интерпретаторе PHP 8. В противном случае будьте готовы к тому, чтобы изменить код и сделать его совместимым с предыдущей версией PHP.

Резюме

Эта книга посвящена объектно-ориентированному проектированию и программированию. В ней описаны средства для работы с кодом PHP, начиная с приложений для совместной работы программистов и заканчивая утилитами для развертывания кода.

Эти две темы раскрывают одну и ту же проблему под различными, но дополняющими друг друга углами зрения. Основная цель состоит в том, чтобы создавать системы, отвечающие заданным требованиям и обеспечивающие возможности совместной работы над проектами.

Второстепенная цель состоит в достижении эстетичности систем программного обеспечения. Программисты создают продукты, облекая их в

определенную форму и приводя их в нужное действие. Они тратят немало рабочего времени, воплощая эти формы в жизнь. Программисты создают нужные им средства — отдельные классы и объекты, компоненты программного обеспечения или окончательные продукты, — чтобы соединить их в единое изящное целое. Процесс контроля версий, тестирования, документирования и построения представляет собой нечто большее, чем достижение этой цели. Это часть формы, которой требуется достичь. Необходимо иметь не только ясный и понятный код, но и кодовую базу, предназначенную как для разработчиков, так и для пользователей. А механизм совместного распространения, чтения и развертывания проекта должен иметь такое же значение, как и сам прикладной код.

ГЛАВА 2

PHP и объекты

Объекты не всегда были основной частью PHP-проекта. Более того, идея реализовать объекты пришла в голову разработчикам PHP в виде запоздалой мысли.

Но впоследствии эта идея доказала свою жизнеспособность. В этой главе дается общее представление об объектах и описывается процесс разработки и внедрения объектно-ориентированных средств в PHP. В частности, здесь рассматривается следующие вопросы.

- *PHP/FI 2.0.* Это язык PHP, но не такой, каким мы его знаем.
- *PHP 3.* Первое появление объектов.
- *PHP 4.* Развитие объектно-ориентированных средств.
- *PHP 5.* Объекты в основе языка.
- *PHP 7.* Восполнение пробела.
- *PHP 8.* Продолжение консолидации.

Неожиданный успех объектов в PHP

При таком количестве существующих объектно-ориентированных библиотек и приложений на PHP, не говоря уже об улучшенной поддержке объектов, развитие последних в PHP может показаться кульминацией естественного и неизбежного процесса. На самом же деле все это очень далеко от истины.

Вначале был PHP/FI

Своим происхождением язык PHP, каким мы его знаем сегодня, обязан двум инструментальным средствам, которые разработал Расмус Лердорф на языке Perl. Сокращение "PHP" обозначало "Personal Home Page Tools" (Средства для персональной начальной страницы), а "FI" — "Form Interpreter" (Интерпретатор форм). Вместе они составляли набор макроко-

манд для отправки SQL-запросов в базу данных, обработки форм и управления процессом обмена данными.

Затем эти средства были переписаны на языке C и объединены под названием "PHP/FI 2.0". На этой стадии синтаксис языка PHP отличался от известного ныне, хотя и *не* значительно. В нем поддерживались переменные, ассоциативные массивы и функции, но объектов не было и в помине.

Синтаксические удобства в версии PHP 3

На самом деле объекты не стояли на повестке дня даже на этапе планирования версии PHP 3. Главными архитекторами версии PHP 3 были Зеэв Сураски и Энди Гутманс. Версия PHP 3 представляла собой полностью переписанную первоначальную версию PHP/FI 2.0, но тогда объекты еще не считались необходимой частью нового синтаксиса.

По словам Зеэва Сураски, поддержка классов была добавлена почти в самом конце, а точнее — 27 августа 1997 года. Классы и объекты на самом деле представляли собой просто другой способ определения ассоциативных массивов и доступа к ним.

Разумеется, добавление методов и наследования существенно расширило возможности классов по сравнению с легендарными ассоциативными массивами. Но на операции с классами все еще накладывались жесткие ограничения. В частности, нельзя было получить доступ к переопределяемым методам родительского класса (если вы еще не знаете, что это такое, не отчаивайтесь — это понятие поясняется далее). Еще одним недостатком, о котором речь пойдет в следующем разделе, был не самый оптимальный способ передачи объектов в сценариях на PHP.

В то время объекты считались второстепенным вопросом. Об этом свидетельствовало и то обстоятельство, что объектам придавалось мало значения в официальной документации. В руководстве по PHP можно было найти лишь одно предложение о них и единственный пример кода. Более того, в данном примере не демонстрировалось применение наследования или свойств.

Версия PHP 4 и незаметная революция

Версия PHP 4 стала еще одним революционным шагом в развитии языка, хотя большинство основных изменений в ней произошло незаметно для пользователя. Для расширения возможностей языка PHP был заново

написан процессор Zend, название которого происходит от имен **Ze**ev и **And**i. Процессор Zend стал одним из основных компонентов, положенных в основу работы PHP. Любая вызываемая в PHP функция на самом деле относится к ярусу высокоуровневых расширений. Такие функции выполняют все возложенные на них обязанности, например взаимодействие с интерфейсом API системы управления базами данных или манипулирование символьными строками. А на нижнем уровне процессор Zend управляет памятью, передает управление другим компонентам и преобразует синтаксис PHP, с которым приходится иметь дело каждый день, в выполняемый байт-код. Именно процессору Zend мы обязаны поддержкой таких основных языковых средств, как классы.

С точки зрения рассматриваемых нами *объектов* большим преимуществом версии PHP 4 стала возможность переопределять родительские методы и получать доступ к ним из дочерних классов. Но оставался и крупный недостаток. Присвоение объекта переменной, передача его функции или его возвращение из метода приводило к появлению копий этого объекта. Поэтому присвоение, подобное следующему:

```
$my_obj = new User('bob');
$other = $my_obj;
```

приводило к появлению двух копий объекта типа `User`, а не двух ссылок на один и тот же объект данного типа. В большинстве объектно-ориентированных языков программирования используется более естественное присваивание по ссылке, а не по значению, как здесь. Это означает передачу и присваивание указателей на объекты, а не копирование самих объектов. Устанавливаемый по умолчанию режим передачи по значению приводил к появлению в сценариях множества скрытых ошибок, когда программисты модифицировали объект в одной части сценария и ожидали, что эти изменения отразятся на всех его копиях. В данной книге представлено немало примеров, в которых сохраняется несколько ссылок на один и тот же объект.

К счастью, в программе на PHP можно было принудительно осуществить передачу объекта по ссылке, но для этого нужно было использовать довольно неуклюжую конструкцию. Ниже показано, каким образом осуществляется присваивание по ссылке:

```
$other =& $my_obj;
// Переменные $other и $my_obj указывают
// на один и тот же объект
```

Передача объекта по ссылке происходит следующим образом:

```
function setSchool(& $school) {
    // Параметр $school теперь ссылается на сам объект,
    // а не на его копию
}
```

А возвращается объект по ссылке следующим образом:

```
function & getSchool( ) {
    // Возврат ссылки на объект, а не его копии
    return $this->school;
}
```

Несмотря на то что такая конструкция действовала исправно, программисты часто забывали добавить символ амперсанда, и вероятность появления ошибок в объектно-ориентированном коде была очень высока. Обнаружить такие ошибки было очень трудно, потому что они редко приводили к сообщениям об ошибках, а правдоподобное поведение прикладного кода на самом деле было неверным.

Описание синтаксиса в целом и объектов в частности было расширено в руководстве по PHP, и ООП стало превращаться в базовую методику. Объекты в PHP были приняты сообществом программистов небесспорно, и сообщения, подобные "Зачем мне нужны эти объекты?", часто вызывали горячую полемику на форумах и в списках рассылки. На сайте Zend наряду со статьями, в которых звучали предостережения, размещались статьи, которые поощряли объектно-ориентированное программирование на PHP. Но, несмотря на недостатки механизма передачи объектов по ссылке и горячую полемику, многие программировавшие на PHP просто приняли новые возможности, приправив свой код символами амперсандов в соответствующих местах. Популярность объектно-ориентированной версии языка PHP стала расти. Вот что Зеэв Сураски написал по этому поводу в своей статье на сайте DevX.com (`http://www.devx.com/webdev/Article/10007/0/page/1`).

Одним из крутых и неожиданных поворотов в истории развития PHP стало зарождение и постепенное превращение объектно-ориентированного программирования в самую популярную парадигму для растущего числа стандартных приложений на PHP, несмотря на очень ограниченные функциональные возможности, многие недостатки и ограничения. Эта, вообще говоря, неожиданная тенденция застала PHP в неблагоприятной ситуации. Стало очевидным, что

объекты ведут себя как [ассоциативные] массивы, а не так, как объекты в других объектно-ориентированных языках.

Как отмечалось в предыдущей главе, интерес к объектно-ориентированному проектированию стал очевиден из растущего числа публикаций статей на сайтах и в форумах в Интернете. В официальном хранилище PEAR программного обеспечения PHP также была принята методика объектно-ориентированного программирования.

Оглядываясь в прошлое, можно подумать, что введение в PHP поддержки средств объектно-ориентированного программирования стало результатом вынужденной капитуляции перед лицом неизбежности. Но не следует забывать, что методика объектно-ориентированного программирования получила широкое распространение только в середине 1990-х годов, хотя и существует еще с 1960-х годов. Язык Java, весьма способствовавший популяризации методики объектно-ориентированного программирования, был выпущен только в 1995 году, а язык C+ появился как расширение процедурного языка C в 1979 году. После длительного периода эволюции он совершил большой скачок только в 1990-х годах. В 1994 году вышла версия 5 языка Perl. Это была еще одна революция для бывшего процедурного языка, что дало возможность пользователям перейти к понятию объектов (хотя некоторые утверждают, что поддержка объектно-ориентированных возможностей в Perl напоминает дополнения, сделанные задним числом). Для небольшого процедурного языка поддержка объектов в PHP была разработана на удивление быстро, что продемонстрировало оперативный отклик на требования пользователей.

Изменения приняты: PHP 5

В версии PHP 5 уже была явно выражена поддержка объектов и объектно-ориентированного программирования. Но это еще не означало, что объекты стали единственным средством для программирования на PHP (кстати, и в данной книге это совсем не утверждается). Тем не менее объекты были признаны эффективными и важными средствами разработки корпоративных приложений, а в PHP предусмотрена их полная и всесторонняя поддержка.

Безусловно, одним из самых важных следствий усовершенствований в версии PHP 5 явилось принятие этого языка в качестве основного в крупных интернет-компаниях. Например, такие компании, как Yahoo! и

Facebook, стали интенсивно пользоваться PHP для создания своих программных платформ. С появлением версии 5 PHP стал одним из стандартных языков для разработки корпоративных веб-приложений в Интернете.

Объекты из дополнительной возможности превратились в основу языка. И вероятно, самым важным изменением стал новый механизм передачи объектов по ссылке, а не по значению. Но это было только начало. На протяжении всей данной книги, и особенно в этой ее части, описывается немало других изменений, расширяющих и усиливающих поддержку объектов в PHP, включая закрытые и защищенные методы и свойства, ключевое слово `static`, пространства имен, уточнения типов, называемые теперь объявлениями типов, а также исключения. Версия PHP 5 существует около 12 лет, и новые важные средства внедрялись в ней постепенно.

На заметку Следует отметить, что PHP, строго говоря, не перешел к передаче по ссылке с введением PHP 5, и эта ситуация не изменилась. По умолчанию, когда объект присваивается, передается методу или возвращается из него, идентификатор этого объекта *копируется*. Пока вы не переходите принудительно к передаче по ссылке с использованием символа `&`, вы все еще выполняете операцию копирования. Однако на практике между этим видом копирования и передачей по ссылке разница незначительна, поскольку вы с вашим скопированным идентификатором ссылаетесь на тот же целевой объект, что и с использованием оригинала.

Например, в версии PHP 5.3 были введены пространства имен. Это позволило создавать именованные области видимости для классов и функций, в результате чего снизилась вероятность дублирования имен при включении компонентов в библиотеки и расширении системы. Кроме того, пространства имен избавляют от неприятной необходимости соблюдать неудобные соглашения об именах, как в приведенном ниже примере.

```
class megaquiz_util_Conf
{
}
```

Подобные имена классов служат для предотвращения конфликтов между пакетами, но, соблюдая соответствующие соглашения, можно только еще больше запутать код. В данной версии появилась также поддержка замыканий, генераторов, трейтов и позднего статического связывания.

Заполнение пробела: PHP 7

Программисты всегда хотят большего. И многим поклонникам проектных шаблонов по-прежнему не хватало в PHP двух основных средств: объявлений скалярных типов аргументов и уточнений типов возвращаемых значений. В версии PHP 5 можно было строго ограничить тип аргумента, передаваемого функции или методу, при условии, что это был объект, массив, а в дальнейшем и вызываемый код. Но ограничить тип скалярных величин, таких как целые числа, символьные строки и числа с плавающей точкой, было нельзя. Более того, если требовалось объявить тип, возвращаемый методом или функцией, сделать это вообще никак не удавалось.

Как поясняется далее в книге, объявление метода применяется в объектно-ориентированном проектировании в качестве своего рода контракта. Методу требуются определенные входные данные, а он, в свою очередь, обязуется возвратить данные определенного типа. В версии 5 программистам на PHP приходилось задействовать комментарии, условные обозначения и ручной контроль типов, чтобы обеспечить во многих случаях контракты подобного рода. Ниже приведена выдержка из четвертого издания данной книги.

> *...среди разработчиков новой версии PHP так и не было достигнуто соглашения по поводу поддержки уточнений для возвращаемых типов данных. Это позволило бы определять в методе или объявлении функции тип возвращаемого объекта. Данное средство должно быть со временем реализовано в движке PHP. Уточнение типов возвращаемых объектов позволило бы еще больше усовершенствовать в PHP поддержку шаблонных принципов программирования наподобие программирования на основе интерфейса, а не его реализации. Как только эта возможность будет реализована, я надеюсь, что включу ее описание в новое издание книги.*

И вот этот час настал. В версии PHP 7 были введены объявления скалярных типов данных (scalar type declarations), называвшиеся ранее уточнениями типов (type hints), а также объявления возвращаемых типов данных. Более того, в PHP 7.4 безопасность типов оказалась только усиленной путем внесения свойств типов, которые также будет рассмотрены в данном издании. В версии PHP 7 появились и другие полезные средства, в том числе анонимные классы и некоторые улучшения, касающиеся пространств имен.

PHP 8: продолжение консолидации

PHP всегда был сорокой, которая не могла устоять перед тем, чтобы утащить красивые (и доказанно работающие) "блестяшки" из других языков программирования. PHP 8 вводит много новых функций, включая атрибуты, часто известные в других языках как *аннотации*. Эти удобные дескрипторы могут использоваться для обеспечения дополнительной контекстуальной информации о классах, методах, свойствах и константах в системе. Кроме того, PHP 8 продолжил расширение поддержки объявлений типов. Особенно интересным в этой области является объявление типа объединения. Оно позволяет вам указать, что тип свойства или параметра должен быть ограничен одним из нескольких указанных типов. Вы можете блокировать свои типы одновременно с использованием гибкости типов PHP.

Дебаты сторонников и противников объектов

Объекты и объектно-ориентированное проектирование, похоже, разжигают страсти среди программистов. Многие профессиональные программисты годами писали отличные программы, не пользуясь объектами, и PHP продолжает оставаться великолепной платформой для процедурного веб-программирования.

В данной книге повсюду демонстрируется пристрастие автора к объектно-ориентированному программированию, поскольку оно отражает его мировоззрение, отличающееся привязанностью к объектам. А поскольку данная книга посвящена объектам и является введением в объектно-ориентированное проектирование, нет ничего удивительного в том, что основное внимание в ней уделяется объектно-ориентированным методикам. Но в то же время в книге нигде не утверждается, что использование объектов — это единственно правильный путь к успеху в программировании на PHP.

Выбор PHP в качестве объектно-ориентированного языка является делом личных предпочтений каждого разработчика. В какой-то степени справедливо, что вполне приемлемые для эксплуатации системы можно по-прежнему создавать с помощью функций и глобального кода. Некоторые замечательные инструментальные средства (например, WordPress)

по-прежнему строятся на основе процедурной архитектуры, хотя в настоящее время в них могут обширно применяться объекты. Но в то же время программировать на PHP, пренебрегая поддержкой объектов в этом языке, становится все труднее. И не в последнюю очередь это объясняется тем, что сторонние библиотеки, используемые в проектах, чаще всего оказываются объектно-ориентированными.

Читая данную книгу, все же стоит помнить знаменитый девиз Perl: "Любую задачу можно решить несколькими способами". Это особенно верно для небольших программ, когда важнее быстро получить рабочий код и запустить его, чем создавать структуру, которая сможет эффективно и безболезненно вырасти в крупную систему (такого рода наспех разрабатываемые проекты называются *костылями (spikes)*).

Код — это гибкая среда. Самое главное — понять, когда быстрое испытание идеи станет основанием для дальнейшего развития, и вовремя остановиться, прежде чем проектные решения будут вам диктовать громадный объем кода. И если вы решили выбрать проектно-ориентированный подход при работе над своим проектом, то можно надеяться, что эта книга станет для вас хорошей отправной точкой и поможет приступить к созданию объектно-ориентированных архитектур.

Резюме

В этой короткой главе объекты рассмотрены в контексте языка PHP. Будущее PHP во многом связано с объектно-ориентированным проектированием. В ряде последующих глав представлено текущее состояние поддержки объектов в PHP и рассмотрены некоторые вопросы проектирования.

ГЛАВА 3

Азы объектов

В данной книге основное внимание уделяется объектам и классам, поскольку с появлением версии 5 более десяти лет назад они стали основными элементами языка PHP. В этой главе заложен прочный фундамент для дальнейшей работы с объектами и описаны подходы к проектированию на основе исследования объектно-ориентированных языковых средств PHP. Если объектно-ориентированное программирование — новая для вас область, постарайтесь очень внимательно прочитать эту главу.

В этой главе рассматриваются следующие вопросы.

- *Классы и объекты.* Объявление классов и создание экземпляров объектов.
- *Методы-конструкторы.* Автоматизация установки начальных значений объектов.
- *Элементарные типы и классы.* Почему тип имеет большое значение.
- *Наследование.* Зачем нужно наследование и как его использовать.
- *Видимость.* Упрощение интерфейсов объектов и защита методов и свойств от вмешательства извне.

Классы и объекты

Первым препятствием для понимания объектно-ориентированного программирования служит странная и удивительная связь между классом и объектом. Для многих людей именно эта связь становится первым моментом откровения, первой искрой интереса к объектно-ориентированному программированию. Поэтому давайте уделим должное внимание самим основам.

Первый класс

Классы часто описывают с помощью объектов. И это забавно, потому что объекты часто описывают с помощью классов. Такое хождение по кру-

гу может сильно затруднить первые шаги в объектно-ориентированном программировании. Именно классы определяют объекты, и поэтому начать следует с определения классов.

Короче говоря, *класс* — это шаблон кода, применяемый для создания объектов. Класс объявляется с помощью ключевого слова `class` и произвольного имени класса. В именах классов может использоваться любое сочетание букв, цифр и символа подчеркивания, но они не должны начинаться с цифры. Код, связанный с классом, должен быть заключен в фигурные скобки. Объединим эти элементы вместе, чтобы создать класс следующим образом:

```
// Листинг 3.1

class ShopProduct
{
   // Тело класса
}
```

Класс `ShopProduct` из данного примера является полноправным классом, хотя пока что и не слишком полезным. Тем не менее мы сделали нечто очень важное, определив тип (или категорию данных), чтобы использовать его в своих сценариях. Важность такого шага станет для вас очевидной по ходу дальнейшего чтения этой главы.

Несколько первых объектов

Если класс — это шаблон для создания объектов, то объект — это данные, которые структурируются в соответствии с шаблоном, определенным в классе. И в этом случае говорят, что объект — это экземпляр класса. Его тип определяется классом.

Воспользуемся классом `ShopProduct` как шаблоном для создания объектов типа `ShopProduct`. Для этого нам потребуется оператор `new`, за которым указывается имя класса, как показано ниже:

```
// Листинг 3.2

$product1 = new ShopProduct();
$product2 = new ShopProduct();
```

После оператора `new` указывается имя класса в качестве его единственного операнда. В итоге создается экземпляр этого класса; в данном примере — объект типа `ShopProduct`.

Мы воспользовались классом `ShopProduct` как шаблоном для создания двух объектов типа `ShopProduct`. И хотя функционально объекты `$product1` и `$product2` идентичны (т.е. пусты), это два разных объекта одного типа, созданные с помощью одного и того же класса.

Если вам все еще непонятно, обратимся к аналогии. Представьте, что класс — это форма для отливки, с помощью которой изготавливают пластмассовых уток, а объекты — отливаемые утки. Тип создаваемых объектов определяется формой отливки. Утки выглядят одинаковыми во всех отношениях, но все-таки это разные предметы. Иными словами, это разные экземпляры одного и того же типа. У отдельных уток могут быть даже разные серийные номера, подтверждающие их индивидуальность. Каждому объекту, создаваемому в сценарии PHP, присваивается идентификатор, однозначно идентифицирующий этот объект в течение всего времени его существования. Идентификаторы объектов в PHP используются повторно, даже в пределах одного и того же процесса. Эту особенность можно продемонстрировать, выведя объекты `$product1` и `$product2` на печать следующим образом:

```
// Листинг 3.3

var_dump($product1);
var_dump($product2);
```

В результате показанных вызовов на экран будет выведена следующая информация:

```
object(popp\ch03\batch01\ShopProduct)#235 (0) {
}
object(popp\ch03\batch01\ShopProduct)#234 (0) {
}
```

На заметку В версиях PHP 4 и PHP 5 (до версии 5.1 включительно) объекты можно выводить на печать непосредственно. В итоге объект будет приведен к символьной строке, содержащей его идентификатор. Но, начиная с версии PHP 5.2, такая возможность больше не поддерживается, и любая попытка интерпретировать объект как символьную строку приведет к ошибке, если только в классе этого объекта не будет определен метод `__toString()`[1]. Методы будут рассмотрены далее в этой главе, а метод `__toString()` — в главе 4.

---

[1] Обратите внимание на то, что имя метода начинается с двух знаков подчеркивания. — *Примеч. ред.*

Передав объект функции `var_dump()`, можно получить о нем полезные сведения, включая внутренний идентификатор каждого объекта, указываемый после символа `'#'`. Чтобы сделать рассматриваемые здесь объекты более интересными, можно немного изменить определение класса `ShopProduct`, добавив в него специальные поля данных, именуемые *свойствами.*

Установка свойств в классе

В классах можно определять специальные переменные, которые называются *свойствами*. Свойство, называемое также *переменной-членом*, содержит данные, которые могут варьироваться от объекта к объекту. Так, для объектов типа `ShopProduct` требуется возможность изменять поля, содержащие название товара и его цену.

Определение свойства в классе похоже на определение обычной переменной, за исключением того, что в операторе объявления перед именем свойства следует указать одно из ключевых слов, характеризующих область кода, для которой это свойство может быть доступно: `public`, `protected` или `private`.

Мы еще вернемся к этим ключевым словам далее в этой главе. А пока что определим некоторые свойства с использованием ключевого слова `public`:

```
// Листинг 3.4

class ShopProduct
{
   public $title             = "Стандартный товар";
   public $producerMainName  = "Фамилия автора";
   public $producerFirstName = "Имя автора";
   public $price             = 0;
}
```

Как можно видеть, мы определили четыре свойства, присвоив каждому из них стандартное значение. Теперь любым объектам, экземпляры которых получаются с помощью класса `ShopProduct`, будут присваиваться стандартные данные. А ключевое слово `public`, присутствующее в объявлении каждого свойства, обеспечивает доступ к этому свойству за пределами контекста его объекта.

К переменным свойств можно обращаться с помощью оператора объекта, обозначаемого как '->', указав имя объектной переменной и имя свойства, как показано ниже:

```
// Листинг 3.5

$product1 = new ShopProduct();
print $product1->title;
```

В результате выполнения приведенных выше строк кода будет выведено следующее:

```
Стандартный товар
```

Свойства объектов были определены как `public`, поэтому их значения можно прочитать и можно присвоить им новые значения, заменив тем самым набор значений по умолчанию, устанавливаемых в классе:

```
// Листинг 3.6

$product1 = new ShopProduct();
$product2 = new ShopProduct();

$product1->title = "Собачье сердце";
$product2->title = "Ревизор";
```

Объявляя и устанавливая свойство `$title` в классе `ShopProduct`, мы гарантируем, что при создании любого объекта типа `ShopProduct` это свойство будет присутствовать, а его значение будет заранее определено. Это означает, что в том коде, в котором используется данный класс, можно работать с любыми объектами типа `ShopProduct`. Но поскольку свойство `$title` можно легко переопределить, его значение может меняться в зависимости от конкретного объекта.

На заметку Код, в котором используется класс, функция или метод, обычно называется клиентом класса, функции или метода или просто *клиентским кодом*. Этот термин будет часто встречаться в последующих главах.

На самом деле в PHP необязательно объявлять все свойства в классе. Свойства можно добавлять в объект динамически, следующим образом:

```
// Листинг 3.7

$product1->arbitraryAddition = "Дополнительный параметр";
```

Следует, однако, иметь в виду, что такой способ назначения свойств для объектов считается неудачной нормой практики в объектно-ориентированном программировании и почти никогда не применяется. Почему такая практика считается неудачной? Дело в том, что при создании класса вы определяете конкретный тип данных. Тем самым вы сообщаете всем, что ваш класс (и любой объект, который является его экземпляром) содержит определенный набор полей и функций. Если в классе ShopProduct имеется свойство $title, то в любом коде, манипулирующем объектами типа ShopProduct, предполагается, что свойство $title определено. Но подобной гарантии в отношении свойств, устанавливаемых динамически, дать нельзя.

Созданные нами объекты пока еще производят довольно тягостное впечатление. Когда нам потребуется манипулировать свойствами объекта, это придется делать за пределами данного объекта. Следовательно, нужно каким-то образом устанавливать и получать значения свойств объекта. Установка нескольких свойств в целом ряде объектов очень быстро становится довольно хлопотным делом, как показано ниже:

```
// Листинг 3.8

$product1->title             = "Собачье сердце";
$product1->producerMainName  = "Булгаков";
$product1->producerFirstName = "Михаил";
$product1->price             = 5.99;
```

Здесь снова используется класс ShopProduct, и все стандартные значения его свойств переопределяются одно за другим до тех пор, пока не будут указаны все сведения о товаре. Теперь, когда у нас имеются некоторые данные, к ним можно обратиться следующим образом:

```
// Листинг 3.9

print "Автор: {$product1->producerFirstName} "
    . "{$product1->producerMainName}\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
```

У такого подхода к определению значений свойств имеется ряд недостатков. В языке PHP допускается определять свойства динамически, так что вы не получите предупреждения, если забудете имя свойства или сделаете в нем опечатку. Например, пусть я хочу ввести строки кода

```
// Листинг 3.10

$product1->producerFirstName = "Михаил";
$product1->producerMainName  = "Булгаков";
```

но случайно ввожу код

```
// Листинг 3.11

$product1->producerFirstName  = "Михаил";
$product1->producerSecondName = "Булгаков";
```

С точки зрения интерпретатора PHP этот код совершенно корректен, поэтому никакого предупреждения об ошибке мы не получим. Но когда понадобится распечатать имя автора, мы получим неожиданные результаты.

Еще одно затруднение состоит в том, что мы слишком "нестрого" определили свой класс, в котором совсем не обязательно нужно указывать название книги, цену или фамилию автора. Клиентский код может быть уверен, что эти свойства существуют, но зачастую их исходно устанавливаемые стандартные значения вряд ли будут его устраивать. В идеале следовало бы побуждать всякого, кто создает экземпляры объекта типа `ShopProduct`, задавать вполне осмысленные значения его свойств.

И, наконец, нам придется приложить немало усилий, чтобы сделать то, что, вероятнее всего, придется делать очень часто. Как было показано выше, распечатать полное имя автора — дело весьма хлопотное. И было бы неплохо, если бы объект делал это вместо нас. Все эти затруднения можно разрешить, если снабдить объект типа `ShopProduct` собственным набором функций, чтобы пользоваться ими для манипулирования данными свойств в контексте самого объекта.

Работа с методами

Если свойства позволяют объектам сохранять данные, то методы позволяют выполнять конкретные задачи. *Методы* — это специальные функции, объявляемые в классе. Как и следовало ожидать, объявление метода напоминает объявление функции. После ключевого слова `function` указывается имя метода, а вслед за ним — необязательный список переменных-аргументов в круглых скобках. Тело метода заключается в фигурные скобки, как показано ниже:

```
// Листинг 3.12

public function myMethod($argument, $another)
{
  // ...
}
```

В отличие от функций, методы следует объявлять в теле класса. При этом можно указывать также ряд спецификаторов, в том числе ключевое слово, определяющее видимость метода. Как и свойства, методы можно определять как `public`, `protected` или `private`. Объявляя метод как `public`, мы тем самым обеспечиваем возможность его вызова за пределами текущего объекта. Если в определении метода опустить ключевое слово, определяющее область его видимости, то метод будет объявлен как `public` неявно. К модификаторам методов мы еще вернемся далее в этой главе.

На заметку В главе 15 рассматриваются советы, как достичь наилучшего качества кода. Стандарт стиля кодирования PSR-12 требует, чтобы видимость была объявлена для каждого метода.

```
// Листинг 3.13

class ShopProduct
{
    public $title              = "Стандартный товар";
    public $producerMainName   = "Фамилия автора";
    public $producerFirstName = "Имя автора";
    public $price              = 0;

    public function getProducer()
    {
        return $this->producerFirstName . " "
             . $this->producerMainName;
    }
}
```

В большинстве случаев метод вызывается с помощью объектной переменной, за которой следуют оператор '->' и имя метода. При вызове метода необходимо указывать круглые скобки, как и при вызове функции (даже если аргументы методу не передаются):

```
// Листинг 3.14

$product1 = new ShopProduct();
```

```
$product1->title             = "Собачье сердце";
$product1->producerMainName  = "Булгаков";
$product1->producerFirstName = "Михаил";
$product1->price             = 5.99;

print "Автор: {$product1->getProducer()}\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
```

Итак, мы ввели в класс `ShopProduct` метод `getProducer()`. Обратите внимание на то, что этот метод был определен как открытый (`public`), а следовательно, его можно вызывать за пределами класса `ShopProduct`.

В теле метода `getProducer()` мы воспользовались новым средством — псевдопеременной `$this`. С ее помощью реализуется механизм доступа к экземпляру объекта из кода класса. Чтобы было проще понять принцип действия такого механизма, попробуйте заменить выражение `$this` "текущим экземпляром объекта". Например, инструкция

```
$this->producerFirstName
```

означает: *свойство `$producerFirstName` текущего экземпляра объекта.*

Таким образом, метод `getProducer()` объединяет и возвращает значения свойств `$producerFirstName` и `$producerMainName`, избавляя нас от лишних хлопот всякий раз, когда требуется распечатать полное имя автора.

Хотя нам удалось немного усовершенствовать наш класс, ему по-прежнему присуща излишняя гибкость. Мы все еще рассчитываем на то, что программист будет изменять стандартные значения свойств объекта типа `ShopProduct`. Но это затруднительно в двух отношениях. Во-первых, потребуется пять строк кода, чтобы должным образом инициализировать объект типа `ShopProduct`, и ни один программист не поблагодарит нас за это. И во-вторых, мы никак не можем гарантировать, что какое-нибудь свойство будет определено при инициализации объекта типа `ShopProduct`. Поэтому нам потребуется метод, который будет вызываться автоматически при создании экземпляра объекта из его класса.

Создание метода конструктора

Метод конструктора вызывается при создании объекта. Он служит для настройки экземпляра объекта, установки определенных значений его свойств и выполнения всей подготовительной работы к применению объекта.

На заметку До версии PHP 5 имя метода-конструктора совпадало с именем класса, к которому оно относилось. Так, в качестве конструктора класса `ShopProduct` можно было использовать метод `ShopProduct()`. Такой синтаксис считается устаревшим, начиная с версии PHP 7, и вообще не работает в версии PHP 8. Метод-конструктор следует именовать `__construct()`.

Обратите внимание на то, что имя метода-конструктора начинается с двух символов подчеркивания. Это правило именования распространяется и на многие другие специальные методы в классах PHP. Теперь определим конструктор для класса `ShopProduct`:

На заметку Встроенные методы, имена которых начинаются таким образом, известны как *магические методы*, потому что в определенных обстоятельствах они вызываются автоматически. Вы можете прочитать о них подробнее в руководстве по PHP по адресу `www.php.net/manual/en/language.oop5.magic.php`. Хотя это и не требуется в обязательном порядке, лучше избегать двойных подчеркиваний в именовании собственных методов.

```
// Листинг 3.15

class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price = 0;
    public function __construct($title, $firstName,
                                $mainName, $price)
    {
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
    }
    public function getProducer()
    {
```

```
        return $this->producerFirstName . " "
              . $this->producerMainName;
    }
}
```

И снова мы добавляем в класс новые функциональные возможности, стараясь сэкономить время и силы программиста и избавить его от необходимости дублировать код, работающий с классом ShopProduct. Метод __construct() автоматически вызывается при создании объекта с помощью оператора new, как показано ниже:

```
// Листинг 3.16

$product1 = new ShopProduct(
                "Собачье сердце",
                "Михаил",
                "Булгаков",
                 5.99
           );

print "Автор: {$product1->getProducer()}\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
```

Значения всех перечисленных аргументов передаются конструктору. В данном примере конструктору передаются название книги, фамилия и имя автора и цена. Для присвоения значений соответствующим свойствам объекта в методе-конструкторе применяется псевдопеременная $this.

На заметку Теперь получать экземпляры и пользоваться объектом типа ShopProduct стало безопаснее и проще, ведь получение экземпляров и установка значений свойств выполняются в одном операторе. При написании любого кода, в котором используется объект типа ShopProduct, можно быть уверенным, что все свойства этого объекта будут инициализированы.

Вы можете оставить свойства неинициализированными — и это не приведет к ошибке, но для программы любая попытка использовать неинициализированное свойство будет фатальной.

Объявление свойств в конструкторе

Хотя мы и сделали класс ShopProduct безопаснее и удобнее с точки зрения клиента, при этом мы ввели большое количество шаблонов. Взгляните на класс. Чтобы создать объект с четырьмя свойствами, нам в общей сложности требуется три набора обращений к данным. Прежде всего мы объявляем свойства, затем мы предоставляем аргументы для конструктора, а после соединяем все вместе, присваивая аргументы метода свойствам. PHP 8 предоставляет функциональную возможность *объявления свойств в конструкторе*, которая позволяет существенно сократить ввод. Добавляя ключевое слово для указания видимости аргумента конструктора, можно скомбинировать аргументы с объявлением свойств *и* одновременно присвоить им значения. Вот как выглядит новая версия ShopProduct:

```
// Листинг 3.17

class ShopProduct
{
    public function __construct(
            public $title,
            public $producerFirstName,
            public $producerMainName,
            public $price
    ) {
    }
    public function getProducer()
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

И объявления, и присваивания свойствам в сигнатуре метода конструктора обрабатываются неявно. Кроме уменьшения дублирования кода, такой синтаксис уменьшает вероятность появления ошибок в коде. Делая класс более компактным, вы облегчаете чтение и понимание исходного кода, позволяя читателю сосредоточиться на его логике.

На заметку Объявление свойств в конструкторе было введено в PHP 8. Если ваш проект все еще работает с PHP 7, это преимущество нового синтаксиса вам недоступно.

Очень важным аспектом объектно-ориентированного программирования является предсказуемость. Вы должны разрабатывать классы таким образом, чтобы пользователи объектов были уверены в их возможностях. Одни из способов сделать объект безопасным — воспроизвести предсказуемые типы данных, хранящиеся в его свойствах. При этом можно гарантировать, что в свойстве `$name`, например, всегда будут находиться только символьные данные. Но как этого добиться, если данные для инициализации свойств поступают в виде параметров извне класса? В следующем разделе мы изучим механизм, который можно использовать для явного определения типов объектов при объявлении методов.

Аргументы по умолчанию и именованные аргументы

Со временем списки аргументов метода могут расти и становиться все более длинными и громоздкими. Это может существенно усложнить работу с классом, так как становится трудно отслеживать аргументы, требующиеся его методам. Мы можем облегчить жизнь программистов — клиентов вашего класса, предоставляя в определениях методов значения по умолчанию. Пусть, например, для объекта `ShopProduct` мы обязаны передать название, но для имени и фамилии автора вполне приемлемы пустые строковые значения, а для цены — нулевое значение. В нынешнем состоянии класса мы должны передавать конструктору `ShopProduct` все эти данные:

```
// Листинг 3.18

$product1 = new ShopProduct("Каталог книг", "", "", 0);
```

Мы можем упростить этот код, предоставляя значения по умолчанию для аргументов:

```
// Листинг 3.19

class ShopProduct
{
    public function __construct(
            public $title,
            public $producerFirstName = "",
            public $producerMainName = "",
```

```
            public $price = 0
    ) {
    }
    // ...
}
```

Эти присваивания активируются только в том случае, если вызывающий код не предоставляет значения при вызове конструктора. Теперь при вызове конструктора совершенно необходимо указать только одно значение — название:

```
// Листинг 3.20

$product1 = new ShopProduct("Каталог книг");
```

Значения аргументов по умолчанию могут сделать работу с методами более удобной, но, как это часто случается, они также могут вызвать непреднамеренные осложнения. Что будет с моим красивым компактным конструктором, если я захочу предоставить цену, но оставить значения по умолчанию для имени и фамилии? До PHP 8 я просто ничего не смог бы сделать. Чтобы указать цену, я должен был бы предоставить пустые строки для имени и фамилии, т.е. не получив никакой выгоды от значений по умолчанию; более того — мне пришлось бы поработать, чтобы выяснить, какие значения конструктор ожидает в качестве значений по умолчанию для имени и фамилии. Должен ли я передать пустые строки? Или некоторые иные значения? Ничуть не экономя мои усилия, значения по умолчанию вполне могут просто вызвать путаницу.

К счастью, PHP 8 предоставляет *именованные аргументы*. Теперь при вызове метода я могу указать имя аргумента со значением, которое я хочу передать. Затем PHP связывает значение с корректным аргументом в сигнатуре метода независимо от порядка передачи аргументов в вызове:

```
// Листинг 3.21

$product1 = new ShopProduct(
        price: 0.7,
        title: "Каталог книг"
);
```

Обратите внимание на использованный здесь синтаксис: я сообщаю PHP, что хочу установить аргумент `$price` равным `0.7`, сначала указывая имя аргумента `price`, затем — двоеточие и значение, которое я хочу предоставить в качестве значения аргумента. Поскольку я использую име-

нованные аргументы, их порядок при вызове больше не имеет значения, и мне больше не нужно предоставлять пустые строки в качестве значения имени и фамилии автора.

Аргументы и типы

Типы определяют порядок работы с данными в сценариях. Например, строковый тип используется для хранения и отображения символьных данных, а также для выполнения операций над такими данными с помощью строковых функций. Целые числа применяются в математических выражениях, булевы значения — в логических выражениях и т.д. Эти категории называются *элементарными* или *примитивными* типами данных (primitive types). Класс также определяет тип данных, но более высокого уровня. Поэтому объект класса `ShopProduct` принадлежит как примитивному типу `object`, так и типу класса `ShopProduct`. В этом разделе мы рассмотрим обе разновидности типов данных по отношению к методам класса.

При определении методов и функций не требуется, чтобы аргумент относился к конкретному типу. Но в этом одновременно заключается и преимущество, и недостаток. То, что аргумент может быть любого типа, приносит немало удобств. Благодаря этому можно создавать методы, которые будут гибко реагировать на данные различных типов и приспосабливать свои функциональные возможности к меняющимся обстоятельствам. Но, с другой стороны, такая гибкость может стать причиной неприятностей в коде, когда в теле метода ожидается аргумент одного типа, но на самом деле передается аргумент другого типа.

Примитивные типы данных

PHP является слабо типизированным языком, а это означает, что объявлять тип данных, который должен храниться в переменной, не нужно. Так, в пределах одной и той же области видимости переменная `$number` может содержать как числовое значение `2`, так и символьную строку `"two"` (“два”). В таких строго типизированных языках программирования, как С или Java, вы обязаны определить тип переменной до присваивания ей значения, и, конечно, это значение должно быть указанного типа.

Но это совсем не означает, что в PHP отсутствует понятие типа. Каждое значение, которое можно присвоить переменной, имеет свой тип данных.

В PHP тип значения переменной можно определить с помощью одной из функций для проверки типов. В табл. 3.1 перечислены примитивные типы данных, имеющиеся в PHP, а также соответствующие им функции. Каждой функции передается переменная или значение, а она возвращает значение `true` ("истина"), если аргумент относится к соответствующему типу данных. Проверка типа переменной особенно важна при обработке аргументов в методе или функции.

Таблица 3.1. *Примитивные типы данных в PHP и их проверочные функции*

Проверочная функция	Тип	Описание
`is_bool()`	`Boolean`	Одно из двух логических значений: `true` (истина) или `false` (ложь)
`is_integer()`	`Integer`	Целое число; является псевдонимом функций `is_int()` и `is_long()`
`is_float()`	`Double`	Число с плавающей (десятичной) точкой; является псевдонимом функции `is_double()`
`is_string()`	`String`	Символьные данные
`is_object()`	`Object`	Объект
`is_resource()`	`Resource`	Дескриптор, используемый для идентификации и работы с такими внешними ресурсами, как базы данных или файлы
`is_array()`	`Array`	Массив
`is_null()`	`Null`	Неопределенное значение

Пример примитивных типов данных

Помните, что вы должны внимательно следить за типами данных в своем коде. Рассмотрим пример одного из многих затруднений, с которыми вы можете столкнуться, используя типы данных.

Допустим, требуется извлечь параметры конфигурации из XML-файла. Элемент `<resolvedomains></resolvedomains>` разметки XML-документа сообщает приложению, следует ли пытаться преобразовывать IP-адреса в доменные имена. Необходимо заметить, что такое преобразование полезно, хотя и является относительно медленной операцией. Ниже приведен фрагмент кода XML для данного примера:

```
// Листинг 3.22

<settings>
    <resolvedomains>false</resolvedomains>
</settings>
```

Приложение извлекает символьную строку "false" и передает ее в качестве параметра методу outputAddresses(), который выводит информацию об IP-адресах. Ниже приведено определение метода output Addresses():

```
// Листинг 3.23

class AddressManager
{
  private $addresses = ["209.131.36.159", "216.58.213.174"];
  public function outputAddresses($resolve)
  {
    foreach ($this->addresses as $address) {
      print $address;
      if ($resolve) {
        print " (".gethostbyaddr($address).")";
      }
      print "\n";
    }
  }
}
```

Разумеется, в класс AddressManager можно внести ряд улучшений. В частности, жестко кодировать IP-адреса в виде массива непосредственно в классе не очень удобно. Тем не менее метод outputAddresses() циклически обходит массив IP-адресов и выводит его элементы по очереди. Если значение аргумента $resolve равно true, то данный метод выводит не только IP-адреса, но и соответствующие им доменные имена.

Ниже приведен один из возможных вариантов применения класса AddressManager вместе с дескриптором settings в элементе XML-разметки файла конфигурации. Попробуйте обнаружить ошибку в приведенном ниже фрагменте кода:

```
// Листинг 3.24

$settings = simplexml_load_file(__DIR__ . "/resolve.xml");
$manager = new AddressManager();
$manager->outputAddresses((string)$settings->resolvedomains);
```

В этом фрагменте кода для получения значения из элемента разметки resolvedomains применяется интерфейс SimpleXML API. В рассматриваемом здесь примере нам известно, что это значение представляет собой текстовый элемент "false", и поэтому оно приводится к строковому типу, поскольку именно так рекомендуется поступать в документации по SimpleXML API.

Но анализируемый здесь код ведет себя совсем не так, как мы того ожидали. Передавая символьную строку "false" методу outputAddresses(), мы не знаем, какой именно тип аргумента используется в этом методе по умолчанию. Данный метод ожидает получить логическое значение аргумента (true или false). При выполнении условного оператора символьная строка "false" на самом деле преобразуется в логическое значение true. Дело в том, что при выполнении проверки интерпретатор PHP с готовностью преобразует непустое строковое значение в логическое значение true. Поэтому фрагмент кода

```
if ( "false" ) {
  // ...
}
```

равнозначен фрагменту кода

```
if ( true ) {
  // ...
}
```

Исправить этот недостаток можно по-разному. Во-первых, можно сделать метод outputAddresses() менее требовательным, чтобы он распознавал символьную строку и применял некоторые основные правила для ее преобразования в логический эквивалент:

```
// Листинг 3.25

public function outputAddresses($resolve)
{
  if (is_string($resolve)) {
    $resolve = (preg_match("/^(false|no|off)$/i", $resolve))
                                 ? false : true;
  }
  // ...
}
```

Но с точки зрения проектирования существуют достаточно веские основания избегать приведенного выше решения. Вообще говоря, при проектировании методов или функций лучше предусматривать для них строгий интерфейс, исключающий двусмысленность, вместо неясного и нетребовательного интерфейса. Подобные решения так или иначе вызывают путаницу и порождают ошибки.

Во-вторых, можно оставить метод `outputAddresses()` без изменений, дополнив его комментариями с четкими инструкциями, что аргумент `$resolve` должен содержать логическое значение. Такой подход позволяет предупредить программиста, что нужно внимательно читать инструкции, а иначе — пенять на себя.

```
// Листинг 3.26

/**
  * Вывести список адресов.
  * Если переменная $resolve содержит значение true,
  * то адрес преобразуется в эквивалентное имя хоста.
  * @param $resolve Boolean Преобразовать адрес?
  */
function outputAddresses( $resolve )
{
   // ...
}
```

Это вполне разумное решение, если вы уверены в том, что программисты, пользующиеся вашим классом, добросовестно прочтут документацию к нему. Наконец, можно сделать метод `outputAddresses()` строгим в отношении типа данных аргумента `$resolve`. Для таких простых типов данных, как логические значения, до выпуска версии PHP 7 это можно было сделать лишь одним способом: написать код для проверки входных данных и предпринять соответствующее действие, если они не отвечают требуемому типу:

```
// Листинг 3.27

function outputAddresses($resolve)
{
  if (! is_bool($resolve)) {
     // Принять решительные меры
  }
  //...
}
```

Такой подход вынуждает клиентский код предоставить корректный тип данных для аргумента `$resolve`.

На заметку В приведенном ниже разделе "Объявления типов: объектные типы" описан намного более совершенный способ наложения ограничений на типы аргументов, передаваемых методам и функциям.

Преобразование строкового аргумента внутри метода — более дружественный подход с точки зрения клиента, но, вероятно, он может вызвать другие проблемы. Обеспечивая механизм преобразования в методе, мы предугадываем контекст его использования и намерение клиента. С другой стороны, соблюдая логический тип данных, мы ставим клиента перед необходимостью принять решение о том, следует ли преобразовывать символьные строки в логические значения, и если да, то определить, какие слова должны соответствовать логическим значениям `true` и `false`. А между тем метод `outputAddresses()` разрабатывался с целью решить конкретную задачу. Такой акцент на выполнении конкретной задачи при *намеренном игнорировании более широкого контекста* является важным принципом объектно-ориентированного программирования, к которому мы еще не раз будем возвращаться в данной книге.

На самом деле стратегии обращения с типами аргументов зависят от степени серьезности возможных ошибок. Большинство значений примитивных типов данных в PHP автоматически приводятся к нужному типу в зависимости от конкретного контекста. Так, если числа, присутствующие в символьных строках, используются, например, в математических выражениях, они автоматически преобразуются в эквивалентные целые значения или же значения с плавающей точкой. В итоге прикладной код может быть нетребовательным к ошибкам несоответствия типов данных.

В общем случае, однако, когда дело касается как объектов, так и примитивных типов, лучше оказаться слишком строгим. К счастью, PHP 8 предоставляет для обеспечения безопасности типов больше инструментов, чем когда-либо ранее.

Дополнительные функции проверки типов

Мы уже видели функции, которые проверяют переменные на принадлежность примитивным типам. Однако следует упомянуть несколько функций, которые идут дальше простой проверки примитивных типов значений и обеспечивают более общую информацию о способах

использования данных, содержащихся в переменных. Они перечислены в табл. 3.2.

Таблица 3.2. *Дополнительные функции проверки типов*

Проверочная функция	Описание
`is_countable()`	Массив объектов, который может быть передан функции `count()`
`is_iterable()`	Структура данных, которая может быть обойдена с помощью цикла `foreach`
`is_callable()`	Код, который может быть вызван, — зачастую это анонимная функция или функция, имеющая имя
`is_numeric()`	`int`, `long` или строка, которая может быть преобразована в число

Функции, описанные в табл. 3.2, проверяют не принадлежность определенным типам, а возможность тем или иным образом рассматривать тестируемые значения. Например, если `is_callable()` возвращает `true` для переменной, то вы знаете, что можете рассматривать ее как функцию или метод и вызывать ее. Аналогично вы можете обходить содержимое переменной, которая успешно проходит тест `is_iterable()`, хотя это может быть не массив, а особый вид объекта.

Объявления типов: объектные типы

Как упоминалось ранее, переменная аргумента может иметь любой примитивный тип данных, но по умолчанию ее тип не оговаривается, и поэтому она может содержать объект любого типа. С одной стороны, такая гибкость удобна, а с другой — она может стать причиной осложнений при определении метода.

Рассмотрим в качестве примера метод, предназначенный для работы с объектом типа `ShopProduct`:

```
// Листинг 3.28

class ShopProductWriter
{
  public function write($shopProduct)
  {
    $str = $shopProduct->title . ": "
```

```
        . $shopProduct->getProducer()
        . " (" . $shopProduct->price . ")\n";
    print $str;
  }
}
```

Мы можем протестировать работу этого класса следующим образом:

```
// Листинг 3.29

$product1 = new ShopProduct("Собачье сердце",
                            "Михаил", "Булгаков", 5.99);
$writer = new ShopProductWriter();
$writer->write($product1);
```

В итоге получим следующий результат:

```
Собачье сердце: Михаил Булгаков (5.99)
```

Класс `ShopProductWriter` содержит единственный метод — `write()`. Методу `write()` передается объект типа `ShopProduct`. Свойства и методы последнего используются в нем для создания и вывода результирующей строки с описанием товара. Мы используем имя переменной аргумента `$shopProduct` как напоминание программисту, что методу `$write()` следует передать объект типа `ShopProduct`, хотя соблюдать это требование необязательно. Это означает, что программист может передать методу `$write()` некорректный объект или вообще данные примитивного типа и ничего не узнать об этом до момента обращения к аргументу `$shopProduct`. К тому времени в нашем коде уже могут быть выполнены какие-либо действия, поскольку предполагалось, что методу `write()` был передан настоящий объект типа `ShopProduct`.

На заметку В связи с изложенным выше у вас может возникнуть вопрос: почему мы не ввели метод `write()` непосредственно в класс `ShopProduct`? Все дело в ответственности. Класс `ShopProduct` отвечает за хранение данных о товаре, а класс `ShopProductWriter` — за вывод этих данных. По мере чтения этой главы вы начнете постепенно понимать, в чем польза такого разделения ответственности.

Для решения упомянутой выше проблемы в PHP 5 были добавлены объявления типов классов, называвшиеся уточнениями типов аргументов. Чтобы добавить объявление типа класса к аргументу метода, достаточно указать перед ним имя класса. Поэтому в метод `write()` можно внести следующие коррективы:

```
// Листинг 3.30

public function write(ShopProduct $shopProduct)
{
  // ...
}
```

Теперь методу `write()` можно передавать аргумент `$shopProduct`, содержащий только объект типа `ShopProduct`. В следующем фрагменте кода предпринимается попытка “перехитрить” метод `write()`, передав ему объект другого типа:

```
// Листинг 3.31

class Wrong
{
}
```

Вот фрагмент кода, который пытается вызвать `write()` для объекта `Wrong`:

```
// Листинг 3.32

$writer = new ShopProductWriter();
$writer->write(new Wrong());
```

Метод `write()` содержит объявление типа класса, и поэтому передача ему объекта типа `Wrong` приведет к фатальной ошибке, как показано ниже:

```
TypeError: popp\ch03\batch08\ShopProductWriter::write():
Argument #1 ($shopProduct) must be of type
popp\ch03\batch04\ShopProduct, popp\ch03\batch08\Wrong
given, called in /var/popp/src/ch03/batch08/Runner.php on ...
```

На заметку В выводе примера `TypeError` вы, возможно, заметили, что ссылки на классы включают много дополнительной информации. Например, класс `Wrong` упоминается как `popp\ch03\batch08\Wrong`. Это примеры пространств имен, с которыми вы встретитесь в главе 4, “Расширенные возможности”.

Это позволяет, вызывая метод `write()`, не проверять тип передаваемого ему аргумента и делает сигнатуру метода намного более понятной для программиста клиентского кода. Он сразу же увидит требования, предъявляемые к вызову метода `write()`. Ему не нужно будет беспокоиться по поводу скрытых ошибок, возникающих в результате несоответствия типа аргумента, поскольку объявление типа класса соблюдается строго.

Несмотря на то что автоматическая проверка типов данных служит превосходным средством для предотвращения ошибок, важно понимать, что объявления типов классов проверяются во время выполнения программы. Это означает, что объявление класса сообщит об ошибке только тогда, когда методу будет передан нежелательный объект. И если вызов метода write() глубоко скрыт в условном операторе, который выполняется только однажды, на Рождество, то вам, возможно, придется поработать на праздники, если вы тщательно не проверите свой код.

Объявления типов: примитивные типы

До выхода PHP 7 можно было накладывать ограничения только на объекты и пару других типов (вызываемых и массивов). Наконец, PHP 7 предоставил возможность объявления скалярных типов. Это позволяет вам обеспечить передачу в списке аргументов логическими, строковыми, целочисленными типами и типами с плавающей точкой.

Имея в своем распоряжении объявления скалярных типов, можно наложить ограничения на класс ShopProduct следующим образом:

```
// Листинг 3.33

class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price = 0;
    public function __construct(string $title, string $firstName,
                                string $mainName, float $price)
    {
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
    }
    // ...
}
```

Изменив таким образом метод-конструктор, можно гарантировать, что аргументы $title, $firstName и $mainName всегда будут содержать строковые данные, тогда как аргумент $price — числовое значение с плавающей точкой. В этом можно убедиться, попытавшись получить экземпляр класса ShopProduct с неверными данными:

```
// Листинг 3.34

// Не сработает!
$product = new ShopProduct("Название", "Имя", "Фамилия", []);
```

В данном случае предпринимается попытка получить экземпляр класса ShopProduct. Конструктору этого класса передаются три символьные строки и пустой массив вместо требующегося числового значения с плавающей точкой, что в конечном итоге приведет к неудачному исходу. Благодаря объявлениям типов данных интерпретатор PHP не допустит такого, выдав следующее сообщение об ошибке:

```
TypeError: popp\ch03\batch09\ShopProduct:: construct(): Argument #4
($price) must be of type float, array given, called in...
```

По умолчанию интерпретатор PHP там, где это возможно, выполнит неявное приведение значений аргументов к требующимся типам данных. И это характерный пример упоминавшегося ранее противоречия между безопасностью и гибкостью прикладного кода. Так, в новой реализации класса ShopProduct символьная строка будет автоматически преобразована в числовое значение с плавающей точкой, и поэтому следующая попытка получить экземпляр данного класса, где символьная строка "4.22" преобразуется внутренним образом в числовое значение 4.22, не вызовет проблем:

```
// Листинг 3.35

$product = new ShopProduct("Название", "Имя", "Фамилия", "4.22");
```

Все это, конечно, замечательно, но вернемся к затруднению, возникшему в связи с применением класса AddressManager, где символьная строка "false" негласно преобразовывалась в логическое значение. По умолчанию это преобразование будет происходить по-прежнему, даже если применить объявление логического типа данных в методе AddressManager::outputAddresses() следующим образом:

```
// Листинг 3.36

public function outputAddresses(bool $resolve)
{
  // ...
}
```

А теперь рассмотрим следующий вызов, где методу `outputAddresses()` передается символьная строка:

```
// Листинг 3.37

$manager->outputAddresses("false");
```

Вследствие неявного приведения типов этот вызов функционально равнозначен вызову данного метода с логическим значением `true`.

Объявления скалярных типов данных можно сделать строгими, но только на уровне отдельных исходных файлов. В следующем примере кода устанавливаются строгие объявления типов данных, а метод `outputAddresses()` снова вызывается с символьной строкой:

```
// Листинг 3.38

declare(strict_types=1);
$manager->outputAddresses("false");
```

Этот вызов приведет к появлению следующей ошибки типа `TypeError` из-за строгих объявлений типов данных:

```
TypeError: popp\ch03\batch09\AddressManager::outputAddresses():
Argument #1 ($resolve) must be of type bool, string given,
called in...
```

На заметку Объявление `strict_types` применяется к тому исходному файлу, из которого делается вызов, а не к исходному файлу, в котором реализована функция или метод. Поэтому соблюдение строгости такого объявления возлагается на клиентский код.

Иногда требуется сделать аргумент необязательным и при этом наложить ограничение на его тип, если он все же указывается. Для этого достаточно указать стандартное значение аргумента, устанавливаемое по умолчанию, как показано ниже:

```
// Листинг 3.39

class ConfReader
{
  public function getValues(array $default = null)
  {
    $values = [];

    // Выполнить действия для получения новых значений
```

```
        // Добавить переданные значения
        // (результат всегда является массивом)

        $values = array_merge($default, $values);
        return $values;
    }
}
```

Смешанные типы

Объявление смешанного типа, введенное в PHP 8.0, может рассматриваться как пример "синтаксического сахара", который сам по себе мало что дает. Нет никакой *функциональной* разницы между

```
// Листинг 3.40

class Storage
{
    public function add(string $key, $value)
    {
        // Действия с $key и $value
    }
}
```

и

```
// Листинг 3.41

class Storage
{
    public function add(string $key, mixed $value)
    {
        // Действия с $key и $value
    }
}
```

Во второй версии я объявил, что аргумент `$value` в `add()` должен принимать тип `mixed`, иными словами — любой из типов `array`, `bool`, `callable`, `int`, `float`, `null`, `object`, `resource` и `string`. Таким образом, объявление `mixed $value` представляет собой то же, что и `$value` без объявления типа в списке аргументов. Тогда зачем вообще беспокоиться об объявлении `mixed`? По сути, вы объявляете, что аргумент *преднамеренно* сделан таким, что может принимать любое значение. Сам по себе аргумент может быть предназначен для передачи любого значения, но, возможно, он остался без объявления типа потому, что автор кода был ленивым. Ключе-

вое слово mixed удаляет сомнения и неопределенность, и по этой причине оно весьма полезно.

Подводя итоги, в табл. 3.3 перечислены объявления типов, поддерживаемые PHP.

Таблица 3.3. *Объявления типов*

Объявление типа	Начиная с версии	Описание
array	5.1	Массив. По умолчанию может быть null-значением или массивом
int	7.0	Целое значение. По умолчанию может быть null или целым значением
float	7.0	Числовое значение с плавающей (десятичной) точкой. Допускается целое значение, даже если включен строгий режим. По умолчанию может быть null, целым или числовым значением с плавающей точкой
callable	5.4	Вызываемый код (например, анонимная функция). По умолчанию может быть null
bool	7.0	Логическое значение. По умолчанию может быть null или логическим значением
string	5.0	Символьные данные. По умолчанию может быть null или строковым значением
self	5.0	Ссылка на содержащий класс
[тип класса]	5.0	Тип класса или интерфейса. По умолчанию может быть null
iterable	7.1	Может быть обойден с использованием цикла foreach (необязательно массив)
object	7.2	Объект
mixed	8.0	Явное указание, что значение может быть любого типа

Объединения

Существует довольно большой промежуток между объявлением mixed и относительной строгостью объявлений типов. Что же делать, если нужно ограничить аргумент двумя, тремя или несколькими именованными типами? До PHP 8 единственным способом достичь этого была проверка типа в теле метода. Вернемся к классу Storage с новым требованием. Метод add() должен принимать только строку или логическое значение в качестве $value. Вот реализация, которая выполняет проверку типа в теле метода:

```
// Листинг 3.42

class Storage
{
  public function add(string $key, $value)
  {
    if (! is_bool($value) && ! is_string($value))
    {
      error_log("Требуется тип string или bool, а не " .
                gettype($value));
      return false;
    }
    // Действия с $key и $value
  }
}
```

На заметку Вместо возврата false лучше генерировать исключение. Об исключениях мы поговорим в главе 4, "Расширенные возможности".

Хотя такая проверка вручную работоспособна, она громоздка и неудобочитаема. К счастью, PHP 8 предоставляет новую функциональную возможность: объединения, которые позволяют объединить два или более типов, разделенных символом "|", создавая объявление составного типа.

Вот новая версия класса Storage:

```
// Листинг 3.43

class Storage
{
  public function add(string $key, string|bool $value)
  {
    // Действия с $key и $value
  }
}
```

Если теперь попытаться установить в качестве `$value` что-либо, кроме строки или булева значения, будет получено знакомое сообщение об ошибке `TypeError`.

Вот как можно сделать `add()` разрешающим немного больше, например добавим разрешение принимать значение `null`:

```
// Листинг 3.44

class Storage
{
  public function add(string $key, string|bool|null $value)
  {
    // Действия с $key и $value
  }
}
```

Объявления объединений могут работать и с объявлениями типов объектов. Вот пример для принятия типа `ShopProduct` или значения `null`:

```
// Листинг 3.45
public function setShopProduct(ShopProduct|null $product)
{
    // Действия с $product
}
```

Поскольку многие методы принимают или возвращают `false` в качестве альтернативного значения, PHP 8 поддерживает псевдотип `false` в контексте объединений. В этом примере методу можно передавать объект типа `ShopProduct` или `false`:

```
// Листинг 3.46
public function setShopProduct(ShopProduct|false $product)
{
    // Действия с $product
}
```

Это более полезно, чем объединение `ShopProduct|bool`, потому что я хочу запретить возможность передавать методу значение `true`.

На заметку Объединения были добавлены в PHP 8.

Типы, принимающие значение `null`

Если объединение принимает `null` как один из двух вариантов, можно использовать другую запись — тип, принимающий значение `null`, который состоит из объявления типа, которому предшествует вопросительный знак. Таким образом, показанная далее версия `Storage` будет принимать либо строку, либо значение `null`:

```
// Листинг 3.47

class Storage
{
  public function add(string $key, ?string $value)
  {
    // Действия с $key и $value
  }
}
```

При описании объявлений типов классов здесь подразумевается, что типы и классы являются синонимами, хотя у них имеются существенные различия. При определении класса определяется также тип, но он может описывать целое семейство классов. Механизм, посредством которого различные классы можно группировать под одним типом, называется *наследованием*. О наследовании речь пойдет в следующем разделе.

Объявление возвращаемого типа

Так же, как мы можем объявить тип аргумента, мы можем использовать и объявления возвращаемого типа, чтобы ограничить типы, которые могут возвращать наши методы. Объявление возвращаемого типа размещается непосредственно после закрывающей скобки метода или функции и состоит из двоеточия, за которым следует тип. Как при объявлении типов аргументов, так и при объявлении возвращаемых типов поддерживается один и тот же набор типов. Вот как я могу ограничить возвращаемый тип метода `getPlayLength()`:

```
// Листинг 3.48

public function getPlayLength(): int
{
    return $this->playLength;
}
```

Если этот метод попытается вернуть не целочисленное значение при вызове, PHP сгенерирует сообщение об ошибке:

```
TypeError: popp\ch03\batch15\CDProduct::getPlayLength():
Return value must be of type int, none returned
```

Поскольку таким образом обеспечивается возврат значения указанного типа, любой код, который вызывает этот метод, может уверенно обрабатывать возвращаемое значение как целое число.

Объявления возвращаемого типа поддерживают объединения и типы, принимающие значения `null`. Вот как можно использовать объединение:

```
// Листинг 3.49

public function getPrice(): int|float
{
    return ($this->price - $this->discount);
}
```

Начиная с PHP 8, имеется один тип, который поддерживается объявлениями возвращаемого типа, но не типов аргументов. Вы можете объявить, что метод никогда не вернет никакого значения, используя псевдотип `void`. Так, например, поскольку метод `setDiscount()` предназначен только для установки значения, но не для его возврата, я использую объявление возвращаемого типа `void`:

```
// Листинг 3.50

public function setDiscount(int|float $num): void
{
    $this->discount = $num;
}
```

Наследование

Наследование — это механизм, посредством которого один или несколько классов можно получить из некоторого базового класса. Класс, унаследованный от другого класса, называется его *подклассом*. Эта связь обычно описывается с помощью терминов *родительский* и *дочерний*. В частности, дочерний класс происходит от родительского и наследует его характеристики, состоящие из свойств и методов. Обычно функциональные возможности родительского класса, который называется также *суперклассом*, дополняются в дочернем классе новыми функциональными возможностями.

Поэтому говорят, что дочерний класс расширяет родительский. Прежде чем приступить к исследованию синтаксиса наследования, рассмотрим проблемы, которые оно поможет нам решить.

Проблема наследования

Вернемся к классу ShopProduct, который в настоящий момент является достаточно обобщенным. Он может обрабатывать самые разные товары, как показано ниже:

```
// Листинг 3.51

$product1 = new ShopProduct( "Собачье сердце",
                             "Михаил", "Булгаков", 5.99 );

$product2 = new ShopProduct( "Классическая музыка. Лучшее",
                             "Антонио", "Вивальди", 10.99 );

print "Автор: "      . $product1->getProducer() . "\n";
print "Композитор: " . $product2->getProducer() . "\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
Композитор: Антонио Вивальди
```

Как видите, разделение имени автора на две части очень хорошо подходит при работе как с книгами, так и с компакт-дисками. В этом случае мы можем отсортировать товары по фамилии автора (т.е. по полю, содержащему строковые значения "Булгаков" и "Вивальди"), а не по имени, в котором содержатся менее определенные строковые значения "Михаил" и "Антонио". Лень — это отличная стратегия проектирования, поэтому на данной стадии разработки не стоит особенно беспокоиться об употреблении класса ShopProduct для описания более чем одного вида товара.

Но если добавить в рассматриваемый здесь пример несколько новых требований, то дело сразу же усложнится. Допустим, требуется отобразить данные, характерные только для книг или только для компакт-дисков. Скажем, для компакт-дисков желательно вывести общую продолжительность звучания, а для книг — количество страниц. Безусловно, могут быть и другие различия, но даже эти наглядно показывают суть возникшей проблемы.

Как же расширить данный пример, чтобы учесть все эти изменения? На ум сразу приходят два варианта. Во-первых, можно расположить все данные в классе ShopProduct, а во-вторых, можно разделить класс

ShopProduct на два отдельных класса. Рассмотрим сначала первый вариант. Ниже показано, как объединить все данные о книгах и компакт-дисках в одном классе:

```
// Листинг 3.52

class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(string $title, string $firstName,
                                string $mainName, float $price,
                                int $numPages = 0,
                                int $playLength = 0)
    {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
             . $this->producerMainName;
    }
}
```

Чтобы продемонстрировать проявляющиеся противоречия, в данном примере были использованы методы доступа к свойствам $numPages и $playLength. В итоге объект, экземпляр которого создается с помощью приведенного выше класса, будет всегда содержать избыточные методы. Кроме того, экземпляр объекта, описывающего компакт-диск, приходит-

ся создавать с помощью лишнего аргумента конструктора. Таким образом, для компакт-диска будут сохраняться данные и функциональные возможности класса, относящиеся к книгам (количество страниц), а для книг — данные о продолжительности звучания компакт-диска. С этим пока еще можно как-то мириться. Но что если добавить больше видов товаров, причем каждый из них с собственными методами обработки, а затем ввести дополнительные методы для каждого вида товара? В конечном счете класс окажется слишком сложным и трудным для применения.

Поэтому принудительное объединение в одном классе полей, относящихся к разным товарам, приведет к созданию слишком громоздких объектов с лишними свойствами и методами.

Но этим рассматриваемая здесь проблема не ограничивается. Функциональные возможности данного класса также вызывают серьезные трудности. Представьте метод, выводящий краткие сведения о товаре. Допустим, что сведения о товаре требуются отделу продаж для указания в виде одной итоговой строки в счете-фактуре. Они хотят, чтобы мы включили в нее время звучания компакт-диска и количество страниц книги. Таким образом, нам придется реализовать этот метод отдельно для каждого вида товара. Чтобы отслеживать формат объекта, можно воспользоваться специальным признаком, как демонстрируется в следующем примере:

```
// Листинг 3.53

public function getSummaryLine(): string
{
    $base  = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    if ($this->type == 'book')
    {
        $base .= ": {$this->numPages} стр.";
    } elseif ($this->type == 'cd')
    {
        $base .= ": Время звучания - {$this->playLength}";
    }
    return $base;
}
```

Чтобы верно установить значение свойства `$type`, можно протестировать аргумент конструктора `$numPages`. Если он содержит число больше нуля, то это книга, иначе — компакт-диск. В итоге класс `ShopProduct` стал еще более сложным, чем нужно. По мере добавления дополнительных отличий в форматы или ввода новых форматов нам будет все труднее

справляться с реализацией функциональных возможностей данного класса. Поэтому для решения рассматриваемой здесь проблемы, по-видимому, придется выбрать другой подход.

В связи с тем, что класс ShopProduct начинает напоминать "два класса в одном", нам придется признать этот факт и создать два типа вместо одного. Ниже показано, как это можно сделать:

```
// Листинг 3.54

class CDProduct
{
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(string $title, string $firstName,
                                string $mainName, float $price,
                                int $playLength)
    {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->playLength        = $playLength;
    }

    public function getPlayLength(): int
    {
        return $this->playLength;
    }

    public function getSummaryLine(): string
    {
        $base = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": Время звучания - {$this->playLength}";
        return $base;
    }

    public function getProducer(): string
    {
        return $this->producerFirstName . " "
        . $this->producerMainName;
    }
}
```

```
// Листинг 3.55

class BookProduct
{
    public $numPages;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(string $title, string $firstName,
                                string $mainName, float $price,
                                int $numPages)
    {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": {$this->numPages} стр.";
        return $base;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
}
```

Мы постарались справиться с упомянутой выше сложностью, хотя для этого пришлось кое-чем пожертвовать. Теперь метод `getSummaryLine()` можно создать для каждого вида товара, даже не проверяя значение специального признака. И ни один из приведенных выше классов больше не содержит полей или методов, не имеющих к нему никакого отношения.

Жертва состоит в дублировании кода. Методы `getProducer()` абсолютно одинаковы в обоих классах. Конструктор каждого из этих классов одинаково устанавливает ряд сходных свойств. И это еще один признак дурного тона, которого следует всячески избегать в программировании.

Если требуется, чтобы методы getProducer() действовали одинаково в каждом классе, любые изменения, внесенные в одну реализацию, должны быть внесены и в другую. Но так мы очень скоро нарушим согласованность обоих классов.

Даже если мы уверены, что сможем справиться с дублированием, на этом наши хлопоты не закончатся. Ведь у нас теперь имеются два типа, а не один!

Вспомните класс ShopProductWriter. Его метод write() предназначен для работы с одним типом ShopProduct. Как же исправить это положение, чтобы все работало, как прежде? Мы можем удалить объявление типа класса из сигнатуры метода, но тогда нам остается лишь надеяться, что методу write() будет передан объект правильного типа. Для проверки типа мы можем ввести в тело метода собственный код, как показано ниже:

```
// Листинг 3.56

class ShopProductWriter
{
    public function write($shopProduct): void
    {
        if (
            !($shopProduct instanceof CDProduct) &&
            !($shopProduct instanceof BookProduct)
        )
        {
            die("Передан неверный тип данных ");
        }
        $str = "{$shopProduct->title}: "
            .     $shopProduct->getProducer()
            . " ({$shopProduct->price})\n";
        print $str;
    }
}
```

Обратите внимание, что в данном примере используется выражение с участием оператора instanceof. Вместо него подставляется логическое значение true, если объект, расположенный слева от оператора instanceof, относится к типу, указанному справа. И снова мы были вынуждены ввести новый уровень сложности. Ведь нам нужно не только проверять аргумент $shopProduct на соответствие двум типам в методе write(), но и надеяться, что в каждом типе будут поддерживаться те же

поля и методы, что и в другом. Согласитесь, что иметь дело только с одним типом было бы намного лучше. Ведь тогда мы могли бы воспользоваться объявлением типа класса в аргументе метода write() и были бы уверены, что в классе ShopProduct поддерживается конкретный интерфейс.

Свойства класса ShopProduct, связанные с книгами и компакт-дисками, не используются одновременно, но они, по-видимому, не могут существовать и по отдельности. Нам нужно оперировать книгами и компакт-дисками как одним типом данных, но в то же время обеспечить отдельную реализацию метода для каждого формата вывода. Следовательно, нам нужно предоставить общие функциональные возможности в одном месте, чтобы избежать дублирования, но в то же время сделать так, чтобы при вызове метода, выводящего краткие сведения о товаре, учитывались особенности этого товара. Одним словом, нам придется воспользоваться наследованием.

Использование наследования

Первый шаг в построении дерева наследования состоит в том, чтобы найти элементы базового класса, которые не соответствуют друг другу или требуют разного обращения с ними. Во-первых, нам известно, что методы getPlayLength() и getNumberOfPages() не могут находиться в одном классе. И во-вторых, нам известно, что потребуются разные реализации метода getSummaryLine(). Мы воспользуемся этими различиями в качестве основания для создания двух производных классов:

```
// Листинг 3.57

class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;
    public function __construct(string $title, string $firstName,
                                string $mainName, float $price,
                                int $numPages = 0,
                                int $playLength = 0)
    {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
```

```
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
            . $this->producerMainName;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}

// Листинг 3.58

class CDProduct extends ShopProduct
{
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": Время звучания - {$this->playLength}";
        return $base;
    }
}

// Листинг 3.59

class BookProduct extends ShopProduct
{
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
```

```
        $base .= "{$this->producerFirstName} )";
        $base .= ": {$this->numPages} стр.";
        return $base;
    }
}
```

Чтобы создать производный (дочерний) класс, достаточно указать в его объявлении ключевое слово extends. В данном примере мы создали два новых класса — BookProduct и CDProduct. Оба они расширяют класс ShopProduct.

Поскольку в производных классах конструкторы не определяются, при получении экземпляров этих классов будет автоматически вызываться конструктор родительского класса. Дочерние классы наследуют доступ ко всем методам типа public и protected из родительского класса, но не к методам и свойствам типа private. Это означает, что метод getProducer() можно вызвать для созданного экземпляра класса CDProduct, как показано ниже, несмотря на то, что данный метод определен в классе ShopProduct:

```
// Листинг 3.60

$product2 = new CDProduct(
                        "Классическая музыка. Лучшее",
                        "Антонио",
                        "Вивальди",
                        10.99,
                        0,
                        60.33
                         );

print "Композитор: {$product2->getProducer()}\n";
```

Таким образом, оба дочерних класса наследуют поведение общего родительского класса. И мы можем обращаться с объектом типа CDProduct так, как будто это объект типа ShopProduct. Мы можем также передать объект типа BookProduct или CDProduct методу write() из класса ShopProductWriter, и все будет работать как следует.

Обратите внимание на то, что для обеспечения собственной реализации в классах CDProduct и BookProduct переопределяется метод getSummaryLine(). Производные классы могут расширять и изменять функциональные возможности родительских классов. И в то же время каждый класс наследует свойства родительского класса.

Реализация метода getSummaryLine() в суперклассе может показаться избыточной, поскольку этот метод переопределяется в обоих дочерних классах. Тем не менее мы предоставляем базовый набор функциональных возможностей, который можно будет использовать в любом новом дочернем классе. Наличие этого метода в суперклассе гарантирует также для клиентского кода, что во всех объектах типа ShopProduct будет присутствовать метод getSummaryLine(). Далее будет показано, как выполнить это обязательство в базовом классе, вообще не предоставляя никакой реализации. Каждый объект дочернего класса ShopProduct наследует все свойства своего родительского класса. В собственных реализациях метода getSummaryLine() из классов CDProduct и BookProduct обеспечивается доступ к свойству $title.

Усвоить понятие наследования сразу не так-то просто. Объявляя один класс, расширяющий другой, мы гарантируем, что экземпляр его объекта определяется характеристиками сначала дочернего, а затем — родительского класса. Чтобы лучше понять наследование, его удобнее рассматривать с точки зрения поиска. Так, если сделать вызов $product2->getProducer(), интерпретатор PHP не сможет найти указанный метод в классе CDProduct. Поиск завершится неудачно, и поэтому будет использована реализация данного метода в классе ShopProduct. С другой стороны, когда делается вызов $product2->getSummaryLine(), интерпретатор PHP находит реализацию метода getSummaryLine() в классе CDProduct и вызывает его.

То же самое относится и к доступу к свойствам. При обращении к свойству $title в методе getSummaryLine() из класса BookProduct интерпретатор PHP не находит определение этого свойства в классе BookProduct и поэтому использует определение данного свойства, заданное в родительском классе ShopProduct. А поскольку свойство $title используется в обоих подклассах, оно должно определяться в суперклассе.

Даже беглого взгляда на конструктор класса ShopProduct достаточно, чтобы понять, что в базовом классе по-прежнему выполняется обработка тех данных, которыми должен оперировать дочерний класс. Так, конструктору класса BookProduct должен передаваться аргумент $numPages, значение которого устанавливается в одноименном свойстве, а конструктор класса CDProduct должен обрабатывать аргумент $playLength и одноименное свойство. Чтобы добиться этого, определим методы-конструкторы в каждом дочернем классе.

Конструкторы и наследование

Определяя конструктор в дочернем классе, вы берете на себя ответственность за передачу требующихся аргументов родительскому классу. Если же вы этого не сделаете, то получите частично сконструированный объект.

Чтобы вызвать нужный метод из родительского класса, придется обратиться непосредственно к этому классу через дескриптор. Для этой цели в PHP предусмотрено ключевое слово `parent`.

Чтобы обратиться к методу в контексте класса, а не объекта, следует использовать символы "`::`", а не "`->`".

На заметку Оператор разрешения области видимости "`::`" будет подробно рассмотрен в главе 4, "Расширенные возможности".

Синтаксическая конструкция `parent::__construct()` означает следующее: "Вызвать метод `__construct()` из родительского класса". Изменим рассматриваемый здесь пример таким образом, чтобы каждый класс оперировал только теми данными, которые имеют к нему непосредственное отношение:

```
// Листинг 3.61

class ShopProduct
{
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    public function __construct($title, $firstName,
                                $mainName, $price)
    {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
    }

    public function getProducer(): string
    {
        return $this->producerFirstName . " "
        . $this->producerMainName;
    }
```

```
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}


// Листинг 3.62

class BookProduct extends ShopProduct
{
    public $numPages;
    public function __construct(string $title, string $firstName,
                                string $mainName, float $price,
                                int $numPages)
    {
        parent::__construct($title,$firstName,$mainName,$price);
        $this->numPages = $numPages;
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base = "{$this->title} ( $this->producerMainName, ";
        $base .= "$this->producerFirstName )";
        $base .= ": {$this->numPages} стр.";
        return $base;
    }
}


// Листинг 3.63

class CDProduct extends ShopProduct
{
    public $playLength;
    public function __construct(string $title, string $firstName,
                                string $mainName, float $price,
                                int $playLength)
    {
        parent::__construct($title,$firstName,$mainName,$price);
        $this->playLength = $playLength;
    }
```

```
    public function getPlayLength(): int
    {
        return $this->playLength;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": Время звучания - {$this->playLength}";
        return $base;
    }
}
```

В каждом дочернем классе перед определением его собственных свойств вызывается конструктор родительского класса. Теперь в базовом классе известны только его собственные данные. Дочерние классы обычно являются специализированными вариантами родительских классов. Как правило, не следует допускать, чтобы в родительских классах было известно что-нибудь о дочерних классах.

На заметку До появления PHP 5 имя конструктора совпадало с именем того класса, к которому он относился. В новом варианте обозначения конструкторов для единообразия используется имя `__construct()`. При использовании старого синтаксиса вызов конструктора из родительского класса был привязан к имени конкретного класса, например `parent::ShopProduct();`. А в версии PHP 7 старый синтаксис вызова конструкторов признан устаревшим и полностью удален в PHP 8.

Вызов перекрытого метода

Ключевое слово `parent` можно использовать в любом методе, перекрывающем свой эквивалент из родительского класса. Когда метод перекрывается, вероятнее всего, требуется расширить, а не отменить функциональные возможности родительского класса. Достичь этого можно, вызвав метод из родительского класса в контексте текущего объекта. Если снова проанализировать реализации метода `getSummaryLine()`, то можно заметить, что значительная часть кода в них дублируется. И лучше воспользоваться этим обстоятельством, как показано ниже, чем повторять функциональные возможности, уже имеющиеся в классе `ShopProduct`:

```
// Листинг 3.64

// Класс ShopProduct...

public function getSummaryLine(): string
{
    $base  = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    return $base;
}

// Листинг 3.65

// Класс BookProduct...

public function getSummaryLine(): string
{
    $base  = parent::getSummaryLine();
    $base .= ": {$this->numPages} стр.";
    return $base;
}
```

Итак, мы определили основные функции для метода `getSummaryLine()` в базовом классе `ShopProduct`.

Вместо того чтобы повторять их в подклассах `CDProduct` и `BookProduct`, мы просто вызовем родительский метод, прежде чем выводить дополнительные данные в итоговую строку.

А теперь, после изложения основ наследования, можно, наконец, рассмотреть видимость свойств и методов, чтобы получить полную картину происходящего.

Управление доступом к классам: модификаторы `public`, `private` и `protected`

До сих пор мы явно или неявно объявляли все свойства как открытые (`public`). Такой тип доступа задан по умолчанию для всех методов и свойств, объявляемых с помощью устаревшего ключевого слова `var`.

На заметку Ключевое слово `var` начиная с PHP 5 является устаревшим и, вероятно, в будущем будет полностью удалено из языка.

Элементы класса можно объявить как `public` (открытые), `private` (закрытые) или `protected` (защищенные). Это означает следующее.

- К открытым свойствам и методам можно получать доступ из любого контекста.
- К закрытому свойству и методу можно получить доступ только из того класса, в котором они объявлены. Даже подклассы данного класса не имеют доступа к таким свойствам и методам.
- К защищенным свойствам и методам можно получить доступ либо из содержащего их класса, либо из его подкласса. Никакому внешнему коду такой доступ не предоставляется.

Чем же это может быть нам полезным? Ключевые слова, определяющие область видимости, позволяют раскрыть только те элементы класса, которые требуются клиенту. Это дает возможность задать ясный и понятный интерфейс для объекта.

Управление доступом, позволяющее запретить клиенту обращаться к некоторым свойствам и методам класса, помогает также избежать ошибок в прикладном коде. Допустим, требуется организовать поддержку скидок в объектах типа `ShopProduct`. С этой целью можно ввести свойство `$discount` и метод `setDiscount()` в данный класс, как показано ниже:

```
// Листинг 3.66

// Класс ShopProduct...

    public $discount = 0;

//...

    public function setDiscount(int $num): void
    {
        $this->discount = $num;
    }
```

Имея на вооружении механизм определения скидки, мы можем создать метод `getPrice()`, в котором во внимание принимается установленная скидка:

```
// Листинг 3.67

public function getPrice()
{
    return ($this->price - $this->discount);
}
```

Но здесь возникает затруднение. Нам требуется раскрыть только скорректированную цену, но клиент может легко обойти метод `getPrice()` и получить доступ к свойству `$price` следующим образом:

```
print "Цена товара - {$product1->price}\n";
```

В итоге будет выведена исходная цена, а не цена со скидкой, которую нам требуется представить. Чтобы избежать этого, достаточно сделать свойство `$price` закрытым (`private`), тем самым запретив клиентам прямой доступ к нему и вынуждая их вызывать метод `getPrice()`. Любая попытка получить доступ к свойству `$price` вне класса `ShopProduct` завершится неудачно. Это свойство перестанет существовать для внешнего мира.

Но объявление свойства как `private` может оказаться излишне строгой мерой, ведь тогда получить доступ к закрытым свойствам своего родительского класса не сможет дочерний класс. А теперь представьте следующие бизнес-правила: скидка не распространяется только на книги. В таком случае можно переопределить метод `getPrice()`, чтобы он возвращал свойство `$price` без учета скидки:

```
// Листинг 3.68

// Класс BookProduct...

public function getPrice(): int|float
{
    return $this->price;
}
```

Свойство `$price` объявлено в классе `ShopProduct`, а не в `BookProduct`, и поэтому попытка в приведенном выше фрагменте кода получить доступ к этому свойству завершится неудачно. Чтобы разрешить это затруднение, следует объявить свойство `$price` защищенным (`protected`) и тем самым предоставить доступ к нему дочерним классам. Напомним, что к защищенным свойствам или методам нельзя получить доступ за пределами иерархии того класса, в котором они были объявлены. Доступ к ним можно получить только из исходного класса или его дочерних классов.

Как правило, предпочтение следует отдавать конфиденциальности. Сначала сделайте свойства закрытыми или защищенными, а затем по мере надобности ослабляйте ограничения, накладываемые на доступ к ним. Многие (если не все) методы в ваших классах будут открытыми, но, опять же, если у вас есть сомнения, ограничьте доступ. Метод, предоставляющий

локальные функциональные возможности другим методам в классе, вообще не нужен пользователям класса. Поэтому сделайте его закрытым или защищенным.

Методы доступа

Даже если в клиентской программе потребуется обрабатывать значения, хранящиеся в экземпляре вашего класса, как правило, стоит запретить прямой доступ к свойствам данного объекта. Вместо этого создайте методы, которые возвращают или устанавливают нужные значения. Такие методы называют *методами доступа* или методами *получения* и *установки*.

Ранее на примере метода `getPrice()` уже демонстрировалось одно преимущество, которое дают методы доступа: с их помощью можно фильтровать значения свойств в зависимости от обстоятельств. Метод установки можно также использовать для соблюдения типа свойства. Если объявления типов классов накладывают определенные ограничения на аргументы методов, то свойства могут содержать данные любого типа. Вспомните определение класса `ShopProductWriter`, в котором для вывода списка данных используется объект типа `ShopProduct`. Попробуем пойти дальше и сделать так, чтобы в классе `ShopProductWriter` можно было одновременно выводить данные из любого количества объектов типа `ShopProduct`:

```
// Листинг 3.69

class ShopProductWriter
{
    public $products = [];
    public function addProduct(ShopProduct $shopProduct): void
    {
        $this->products[] = $shopProduct;
    }
    public function write(): void
    {
        $str = "";
        foreach ($this->products as $shopProduct)
        {
            $str .= "{$shopProduct->title}: ";
            $str .= $shopProduct->getProducer();
            $str .= " ({$shopProduct->getPrice()})\n";
        }
        print $str;
    }
}
```

Теперь класс ShopProductWriter стал намного более полезным. Он может содержать много объектов типа ShopProduct и сразу выводить информацию обо всех этих объектах. Но мы все еще должны полагаться на то, что программисты клиентского кода будут строго придерживаться правил работы с классом. И хотя мы предоставили метод addProduct(), мы не запретили программистам непосредственно манипулировать свойством $products. В итоге можно не только добавить объект неверного типа в массив свойств $products, но и затереть весь массив и заменить его значением простого типа. Чтобы не допустить этого, нужно сделать свойство $products закрытым:

```
// Листинг 3.70

class ShopProductWriter
{
    private $products = [];
    //...
```

Теперь внешний код не сможет повредить массив свойств $products. Весь доступ к нему должен осуществляться через метод addProduct(), а объявление типа класса, которое используется в объявлении этого метода, гарантирует, что в массив свойств могут быть добавлены только объекты типа ShopProduct.

Типизированные свойства

Таким образом, путем объединения объявлений типов в сигнатурах методов с объявлениями видимости свойств вы можете управлять свойствами типов в своих классах. Вот еще один пример: класс Point, в котором я использую объявления типа и видимость свойства для управления типами свойств:

```
// Листинг 3.71

class Point
{
    private $x = 0;
    private $y = 0;
    public function setVals(int $x, int $y)
    {
        $this->x = $x;
        $this->y = $y;
    }
```

```
    public function getX(): int
    {
        return $this->x;
    }
    public function getY(): int
    {
        return $this->y;
    }
}
```

Поскольку свойства `$x` и `$y` закрытые, они могут быть установлены только через метод `setVals()`, а поскольку `setVals()` принимает только целочисленные значения, вы можете быть уверены, что `$x` и `$y` всегда содержат целые числа.

Конечно, поскольку эти свойства определены как `private`, единственное средство, с помощью которого они могут быть доступны, — это методы доступа.

До PHP 7.4, в котором введены типизированные свойства, мы бы имели проблемы. Но сейчас мы можем объявлять типы наших свойств:

```
// Листинг 3.72

class Point
{
    public int $x = 0;
    public int $y = 0;
}
```

Я сделал свойства `$x` и `$y` открытыми и использовал объявление типа для того, чтобы ограничить их типы. В результате при желании я могу избавиться от метода `setVals()`, не жертвуя возможностями управления. Мне также больше не нужны методы `getX()` и `getY()`. Теперь `Point` — исключительно простой класс, но даже при наличии открытых свойств он предоставляет гарантии о хранимых данных.

Давайте попробуем указать строку в качестве значения одного из этих свойств:

```
// Листинг 3.73

$point = new Point();
$point->x = "a";
```

PHP не позволит этого, выдавая следующее сообщение об ошибке:

```
TypeError: Cannot assign string to property
popp\ch03\batch11\Point::$x of type int
```

На заметку В объявлении свойств типов можно использовать типы-объединения.

Семейство классов ShopProduct

В заключение этой главы внесем коррективы в класс ShopProduct и его дочерние классы таким образом, чтобы ограничить доступ к их элементам и воспользоваться некоторыми из описанных выше возможностей:

```
// Листинг 3.74

class ShopProduct
{
    private int | float $discount = 0;
    public function __construct(private string $title,
                                private string $producerFirstName,
                                private string $producerMainName,
                                protected int | float $price)
    {
    }
    public function getProducerFirstName(): string
    {
        return $this->producerFirstName;
    }
    public function getProducerMainName(): string
    {
        return $this->producerMainName;
    }
    public function setDiscount(int | float $num): void
    {
        $this->discount = $num;
    }
    public function getDiscount(): int
    {
        return $this->discount;
    }
    public function getTitle(): string
    {
        return $this->title;
    }
```

```
    public function getPrice(): int | float
    {
        return ($this->price - $this->discount);
    }
    public function getProducer(): string
    {
        return $this->producerFirstName . " "
        . $this->producerMainName;
    }
    public function getSummaryLine(): string
    {
        $base = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}
```

В дополнение к закрытию доступа к большинству свойств путем установки их видимости `private` (или `protected` в случае `$discount`) я воспользовался возможностью объявления свойств в конструкторе, объединив свои объявления свойств с сигнатурой конструктора. Я также воспользовался объявлением типа свойства для `$discount`, демонстрирующим новую возможность PHP 8 — объединение типов, и ограничил `$discount` так, чтобы это свойство могло принимать значения *либо* типа `int`, либо типа `float`. Это ограничение может показаться избыточным, поскольку свойство `$discount` объявлено как `private`, а объявление типа в методе `setDiscount()`, которое представляет собой еще одно объединение, обеспечивает выполнение того же условии. Тем не менее это хорошая практика — объявлять типы своих свойств, потому что частично это действует как своего рода документация, а частично — препятствует возникновению случайных ошибок при дальнейшем развитии класса `ShopProduct`:

```
// Листинг 3.75

class CDProduct extends ShopProduct
{
    public function __construct(string $title, string $firstName,
                                string $mainName, int | float $price,
                                private int $playLength)
    {
        parent:: __construct($title,$firstName,$mainName,$price);
    }
    public function getPlayLength(): int
    {
```

```
        return $this->playLength;
    }
    public function getSummaryLine(): string
    {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": Время звучания - {$this->playLength}";
        return $base;
    }
}
```

Я вновь использую возможность объявления свойств в сигнатуре конструктора, на этот раз только для одного аргумента: `$playLength`. Поскольку я передаю остальные аргументы конструктора в родительский класс, я не устанавливаю их видимость. Вместо этого я использую их в теле конструктора:

```
// Листинг 3.76

class BookProduct extends ShopProduct
{
    public function __construct(string $title, string $firstName,
                                string $mainName, int | float $price,
                                private int $numPages)
    {
        parent:: construct($title,$firstName,$mainName,$price);
    }
    public function getNumberOfPages(): int
    {
        return $this->numPages;
    }
    public function getSummaryLine(): string
    {
        $base  = parent::getSummaryLine();
        $base .= ": - {$this->numPages} стр.";
        return $base;
    }
    public function getPrice(): int | float
    {
        return $this->price;
    }
}
```

Итак, в этой версии семейства `ShopProduct` все свойства являются либо `private`, либо `protected`. Я также добавил в классы ряд методов доступа.

Резюме

В этой главе мы подробно рассмотрели основы объектно-ориентированного программирования на PHP, превратив первоначально пустой класс в полностью функциональную иерархию наследования. Мы разобрались в некоторых вопросах проектирования, особенно касающихся типов данных и наследования. Из этой главы вы также узнали о поддержке видимости элементов кода в PHP и ознакомились с некоторыми примерами ее применения. В следующей главе будут представлены другие объектно-ориентированные возможности PHP.

ГЛАВА 4

Расширенные возможности

В предыдущей главе вы ознакомились с объявлениями типов классов и управлением доступом к свойствам и методам. Все это позволяет очень гибко управлять интерфейсом класса. В настоящей главе мы подробно исследуем более развитые объектно-ориентированные возможности языка PHP.

В этой главе рассматриваются следующие вопросы.

- *Статические методы и свойства.* Доступ к данным и функциям с помощью классов, а не объектов.
- *Абстрактные классы и интерфейсы.* Отделение проектного решения от его реализации.
- *Трейты.* Совместное использование реализации в иерархиях классов.
- *Обработка ошибок.* Общее представление об исключениях.
- *Завершенные классы и методы.* Ограниченное наследование.
- *Методы-перехватчики.* Автоматическая передача полномочий.
- *Методы-деструкторы.* Освобождение ресурсов после использования объекта.
- *Клонирование объектов.* Создание копий объектов.
- *Преобразование объектов в символьные строки.* Создание сводного метода.
- *Функции обратного вызова.* Расширение функциональных возможностей компонентов с помощью анонимных функций и классов.

Статические методы и свойства

Во всех примерах из предыдущей главы мы работали с объектами. Классы были охарактеризованы как шаблоны, с помощью которых создаются объекты, а объекты — как активные компоненты, методы которых можно вызывать и доступ к свойствам которых можно получать. Из этого был

сделан вывод, что вся реальная работа в объектно-ориентированном программировании выполняется в экземплярах классов. А классы в конечном счете служат шаблонами для создания объектов.

Но на самом деле не все так просто. Доступ к методам и свойствам можно получать в контексте класса, а не объекта. Такие методы и свойства являются статическими и должны быть объявлены с помощью ключевого слова `static`, как показано ниже:

```
// Листинг 4.1

class StaticExample
{
    public static int $aNum = 0;
    public static function sayHello(): void
    {
        print "Здравствуй, Мир!";
    }
}
```

Статические методы — это функции, применяемые в контексте класса. Они не могут сами получать доступ к обычным свойствам класса, потому что такие свойства относятся к объектам. Но из статических методов можно обращаться к *статическим свойствам*. Если изменить статическое свойство, то все экземпляры данного класса смогут получить доступ к новому значению этого свойства.

Доступ к статическому элементу осуществляется через класс, а не через экземпляр объекта, и поэтому переменная для ссылки на объект не требуется. Вместо этого используются имя класса и два знака двоеточия `"::"`, как в следующем примере кода:

```
// Листинг 4.2

print StaticExample::$aNum;
StaticExample::sayHello();
```

Этот синтаксис был представлен в предыдущей главе. Мы использовали синтаксическую конструкцию `"::"` вместе с ключевым словом `parent`, чтобы получить доступ к переопределенному методу родительского класса. Но теперь мы будем обращаться к классу, а не к данным, содержащимся в объекте. Ключевое слово `parent` можно использовать в коде класса, чтобы получить доступ к суперклассу, не употребляя имя класса. Чтобы получить доступ к статическому методу или свойству из того же самого

класса (а не из дочернего класса), мы пользуемся ключевым словом `self`. Ключевое слово `self` служит для обращения к текущему классу подобно тому, как псевдопеременная `$this` — к текущему объекту. Поэтому обращаться к свойству `$aNum` за пределами класса `StaticExample` следует по имени его класса, как показано ниже:

```
StaticExample::$aNum;
```

А в самом классе `StaticExample` для этой цели можно воспользоваться ключевым словом `self` следующим образом:

```
// Листинг 4.3

class StaticExample2
{
    public static int $aNum = 0;
    public static function sayHello(): void
    {
        self::$aNum++;
        print "Привет! (" . self::$aNum . ")\n";
    }
}
```

На заметку Вызов метода с помощью ключевого слова `parent` — это единственный случай, когда следует использовать статическую ссылку на нестатический метод.

Если только вы не обращаетесь к перекрытому методу родительского класса, синтаксической конструкцией `"::"` следует пользоваться только для доступа к методам или свойствам, явно объявленным статическими.

Но в документации часто можно увидеть примеры применения синтаксической конструкции `"::"` для ссылок на методы или свойства. Это совсем не означает, что рассматриваемый элемент кода непременно является статическим, — просто он относится к указанному классу. Например, ссылку на метод `write()` из класса `ShopProductWriter` можно сделать следующим образом: `ShopProductWriter::write()`, несмотря на то, что метод `write()` не является статическим. Мы будем пользоваться синтаксической конструкцией `"::"` в тех примерах из данной книги, в которых это уместно.

По определению обращение к статическим методам и свойствам происходит в контексте класса, а не объекта. Именно по этой причине статические свойства и методы часто называют переменными и свойствами клас-

са. Вследствие такой ориентации на класс пользоваться псевдопеременной `$this` в теле статического метода для доступа к элементам текущего объекта нельзя.

А зачем тогда вообще нужны статические методы или свойства? Статические элементы имеют ряд полезных характеристик. Во-первых, они доступны из любого места в сценарии, при условии, что имеется доступ к классу. Это означает, что такие функции можно вызывать, не передавая экземпляр класса от одного объекта другому или, что еще хуже, сохраняя ссылку на экземпляр объекта в глобальной переменной. Во-вторых, статическое свойство доступно каждому экземпляру объекта данного класса. Поэтому так можно определить значения, которые должны быть доступны всем объектам данного типа. И, наконец, благодаря тому, что отпадает необходимость в наличии экземпляра класса для доступа к его статическому свойству или методу, можно избежать создания экземпляров объектов исключительно ради вызова простой функции.

Чтобы продемонстрировать все сказанное выше на конкретном примере, создадим в классе `ShopProduct` статический метод для автоматического инстанцирования объектов типа `ShopProduct`. Используя SQLite, определим следующую таблицу `products`:

```
// Листинг 4.4
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    type       TEXT,
    firstname  TEXT,
    mainname   TEXT,
    title      TEXT,
    price      float,
    numpages   int,
    playlength int,
    discount   int )
```

Теперь создадим метод `getInstance()`, которому передаются идентификатор строки и объект типа `PDO`. Они будут использоваться для извлечения строки из таблицы базы данных, на основании которой затем формируется объект типа `ShopProduct`, возвращаемый в вызывающую часть программы. Мы можем ввести приведенные ниже методы в класс `ShopProduct`, который был создан в предыдущей главе. Как вы, вероятно, знаете, сокращение PDO означает *PHP Data Object* (Объект данных PHP). А класс `PDO` обеспечивает универсальный интерфейс для различных приложений баз данных.

```
// Листинг 4.5

// ShopProduct class...
private int $id = 0;
// ...
public function setID(int $id): void
{
    $this->id = $id;
}
// ...
public static function getInstance(int $id, \PDO $pdo): ShopProduct
{
    $stmt = $pdo->prepare("select * from products where id=?");
    $result = $stmt->execute([$id]);
    $row = $stmt->fetch();

    if (empty($row))
    {
        return null;
    }

    if ($row['type'] == "book")
    {
        $product = new BookProduct(
            $row['title'],
            $row['firstname'],
            $row['mainname'],
            (float) $row['price'],
            (int) $row['numpages']
        );
    }

    elseif($row['type'] == "cd")
    {
        $product = new CDProduct(
            $row['title'],
            $row['firstname'],
            $row['mainname'],
            (float) $row['price'],
            (int) $row['playlength']
        );
    }
    else
    {
        $firstname = (is_null($row['firstname'])) ? "" :
        $row['firstname'];
        $product = new ShopProduct(
```

```
            $row['title'],
            $firstname,
            $row['mainname'],
            (float) $row['price']
        );
    }

    $product->setId((int) $row['id']);
    $product->setDiscount((int) $row['discount']);
    return $product;
}
```

Как видите, метод `getInstance()` возвращает объект типа `ShopProduct`, причем он достаточно развит логически, чтобы на основании значения поля `type` создать объект с нужными характеристиками. В данном примере специально опущена обработка ошибок, чтобы сделать его как можно более лаконичным. Так, в реально работающей версии такого кода нельзя слишком полагаться на то, что переданный объект типа `PDO` был правильно инициализирован для обращения к нужной базе данных. На самом деле объект типа `PDO`, вероятно, следует заключить в оболочку класса, гарантирующего надлежащее поведение. Подробнее об ООП применительно к базам данных речь пойдет в главе 13, "Шаблоны баз данных".

Метод `getInstance()` оказывается более пригодным в контексте класса, чем в контексте объекта. Он позволяет легко преобразовать информацию из базы данных в объект, не требуя отдельного объекта типа `ShopProduct`. В этом методе вообще не используются методы или свойства экземпляра, и поэтому ничто не мешает объявить его статическим. И тогда, имея в своем распоряжении достоверный объект типа `PDO`, можно вызвать данный метод из любого места в приложении, как показано ниже:

```
// Листинг 4.6
$dsn = "sqlite:/tmp/products.sqlite3";
$pdo = new \PDO($dsn, null, null);
$pdo->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
$obj = ShopProduct::getInstance(1, $pdo);
```

Подобные методы работают, как "фабрики", поскольку берут "сырье" (например, информацию, полученную из строки таблицы базы данных или файла конфигурации) и используют его для создания объектов. Термин *фабрика* относится к коду, предназначенному для создания экземпляров объектов. Примеры подобных "фабрик" еще не раз будут встречаться в последующих главах.

Безусловно, рассмотренный выше пример выявляет не меньше проблем, чем демонстрирует решений. И хотя метод ShopProduct::getInstance() был сделан доступным из любой части программы, не прибегая к получению экземпляра класса ShopProduct, при его определении потребовалось также, чтобы объект типа PDO был передан из клиентского кода. А откуда его взять? И насколько допустимой считается норма практики, когда в родительском классе известно все до мельчайших подробностей о его дочерних классах? Такая практика, конечно, недопустима. Проблемы наподобие “Где взять набор ключевых объектов программы и их значений?” и “Насколько одни классы должны быть осведомлены о других классах?” широко распространены в объектно-ориентированном программировании. Различные подходы к формированию объектов будут рассмотрены в главе 9, “Генерация объектов”.

Константные свойства

Некоторые свойства объектов не должны изменяться. Например, такие элементы, как коды ошибок или коды состояния программы, обычно задаются в классах вручную. И хотя они должны быть общедоступными и статическими, клиентский код не должен иметь возможности их изменять.

В языке PHP постоянные свойства можно определить в самом классе. Как и глобальные константы, константы класса нельзя изменить после их определения. Постоянное свойство объявляется с помощью ключевого слова const. В отличие от обычных свойств перед именем константного свойства не ставится знак доллара. Для них зачастую принято выбирать имена, состоящие только из прописных букв, как в следующем примере:

```
// Листинг 4.7

class ShopProduct
{
    const AVAILABLE    = 0;
    const OUT_OF_STOCK = 1;
```

Константные свойства могут содержать только значения, относящиеся к примитивному типу. Константе нельзя присвоить объект. Как и к статическим свойствам, доступ к константным свойствам осуществляется через класс, а не через экземпляр объекта. А поскольку константа определяется без знака доллара, при обращении к ней не нужно указывать никакого специального знака перед ее именем:

```
// Листинг 4.8
print ShopProduct::AVAILABLE;
```

На заметку Поддержка модификаторов видимости для констант введена в PHP 7.1. Эти модификаторы работают так же, как и модификаторы видимости для свойств.

Любая попытка присвоить константе значение после ее объявления приведет к ошибке на стадии синтаксического анализа.

Константы следует использовать в тех случаях, когда свойство должно быть доступным для всех экземпляров класса и когда значение свойства должно быть фиксированным и неизменным.

Абстрактные классы

Абстрактный класс отличается тем, что невозможно создать его экземпляр. Вместо этого в нем определяется (и, возможно, частично реализуется) интерфейс для любого класса, который может его расширить.

Абстрактный класс определяется с помощью ключевого слова `abstract`. Переопределим класс `ShopProductWriter`, который мы создали в предыдущей главе, как абстрактный:

```
// Листинг 4.9

abstract class ShopProductWriter
{
    protected array $products = [];
    public function addProduct(ShopProduct $shopProduct): void
    {
        $this->products[] = $shopProduct;
    }
}
```

В абстрактном классе можно создавать методы и свойства, как обычно, но любая попытка получить его экземпляр приведет к ошибке. Например, выполнение строки кода

```
// Листинг 4.10

$writer = new ShopProductWriter();
```

приведет к выводу сообщения об ошибке:

```
Error: Cannot instantiate abstract class
popp\ch04\batch03\ShopProductWriter
```

Как правило, в абстрактном классе должен содержаться по меньшей мере один абстрактный метод. Как и абстрактный класс, он объявляется с помощью ключевого слова `abstract`. Абстрактный метод не может быть реализован в абстрактном классе. Он объявляется как обычный метод, но объявление завершается точкой с запятой, а не телом метода. Добавим абстрактный метод `write()` в класс `ShopProductWriter` следующим образом:

```
// Листинг 4.11

abstract class ShopProductWriter
{
    protected array $products = [];
    public function addProduct(ShopProduct $shopProduct): void
    {
        $this->products[] = $shopProduct;
    }
    abstract public function write(): void;
}
```

Создавая абстрактный метод, вы гарантируете, что его реализация будет доступной во всех конкретных дочерних классах, но подробности этой реализации остаются неопределенными.

Если попытаться создать производный от `ShopProductWriter` класс, в котором метод `write()` не реализован:

```
// Листинг 4.12
class ErroredWriter extends ShopProductWriter
{
}
```

то появится такое сообщение об ошибке:

```
Fatal error: Class popp\ch04\batch03\ErroredWriter contains 1
abstract method and must therefore be declared abstract or
implement the remaining methods
(popp\ch04\batch03\ShopProductWriter::write) in...
```

Таким образом, в любом классе, расширяющем абстрактный класс, должны быть реализованы все абстрактные методы или же сам этот класс также должен быть объявлен абстрактным. Расширяющий класс отвечает не только за реализацию всех абстрактных методов, но и за соответствие их сигнатур. Это означает, что уровень доступа в реализующем методе не

может быть более строгим, чем в абстрактном методе. Кроме того, реализующему методу должно передаваться такое же количество аргументов, как и абстрактному методу, и в нем должны совпадать все объявления типов классов.

Ниже приведены две реализации абстрактного класса ShopProductWriter:

```
// Листинг 4.13
class XmlProductWriter extends ShopProductWriter
{
    public function write(): void
    {
        $writer = new \XMLWriter();
        $writer->openMemory();
        $writer->startDocument('1.0', 'UTF-8');
        $writer->startElement("Товары");
        foreach ($this->products as $shopProduct)
        {
            $writer->startElement("Товар");
            $writer->writeAttribute("Наименование",
                                    $shopProduct->getTitle());
            $writer->startElement("Резюме");
            $writer->text($shopProduct->getSummaryLine());
            $writer->endElement();
            $writer->endElement();
        }
        $writer->endElement();
        $writer->endDocument();
        print $writer->flush();
    }
}
```

Вот более простой класс TextProductWriter:

```
// Листинг 4.14
class TextProductWriter extends ShopProductWriter
{
    public function write(): void
    {
        $str = "ТОВАРЫ:\n";
        foreach ($this->products as $shopProduct)
        {
            $str .= $shopProduct->getSummaryLine() . "\n";
        }
        print $str;
    }
}
```

Здесь созданы два класса, причем каждый с собственной реализацией метода `write()`. Первый класс служит для вывода сведений о товаре в формате XML, а второй — в текстовом виде. Теперь в методе, которому требуется передать объект типа `ShopProductWriter`, не имеет особого значения, объект какого из этих двух классов он получит, но в то же время имеется полная уверенность, что в обоих классах реализован метод `write()`. Обратите внимание на то, что тип свойства `$products` не проверяется, прежде чем использовать его как массив. Дело в том, что это свойство объявлено как массив и инициализировано в классе `ShopProductWriter`.

Интерфейсы

Как известно, в абстрактном классе допускается реализация некоторых методов, не объявленных абстрактными. В отличие от них, *интерфейсы* — это чистые шаблоны. С помощью интерфейса можно только определить функциональность, но не реализовать ее. Для объявления интерфейса используется ключевое слово `interface`. В интерфейсе могут находиться только объявления, но не тела методов.

Ниже приведен пример объявления интерфейса:

```
// Листинг 4.15
interface Chargeable
{
    public function getPrice(): float;
}
```

Как видите, интерфейс очень похож на абстрактный класс. В любом классе, поддерживающем этот интерфейс, необходимо реализовать все определенные в нем методы. В противном случае класс должен быть объявлен как абстрактный.

При реализации интерфейса в классе имя интерфейса указывается в объявлении этого класса после ключевого слова `implements`. После этого процесс реализации интерфейса станет точно таким же, как и расширение абстрактного класса, который содержит только абстрактные методы. А теперь сделаем так, чтобы в классе `ShopProduct` был реализован интерфейс `Chargeable`:

```
// Листинг 4.16
class ShopProduct implements Chargeable
{
    // ...
```

```
    protected float $price;
    // ...
    public function getPrice(): float
    {
        return $this->price;
    }
    // ...
}
```

В классе ShopProduct уже имеется метод getPrice(), так что же может быть полезного в реализации интерфейса Chargeable? И в этом случае ответ следует искать в типах данных. Дело в том, что реализующий класс принимает тип класса и интерфейса, который он расширяет. Это означает, что класс CDProduct теперь будет относиться к следующим типам:

```
CDProduct
ShopProduct
Chargeable
```

Этой особенностью можно воспользоваться в клиентском коде. Как известно, тип объекта определяет его функциональные возможности. Поэтому в методе

```
// Листинг 4.17
public function cdInfo(CDProduct $prod): int
{
    // Мы знаем, что можем вызвать getPlayLength()
    $length = $prod->getPlayLength();
    // ...
}
```

известно, что у объекта $prod имеется метод getPlayLength(), а также все остальные методы, определенные в классе ShopProduct и интерфейсе Chargeable.

Если передать тот же объект, но с более общими требованиями — ShopProduct, а не CDProduct, — то известно только, что предоставленный объект содержит методы ShopProduct:

```
// Листинг 4.18
public function addProduct(ShopProduct $prod)
{
    // Даже если $prod является объектом CDProduct, мы этого
    // не знаем и не можем использовать getPlayLength()
    // ...
}
```

Без дополнительной проверки в данном методе ничего не будет известно о методе getPlayLength().

Но если передать тот же самый объект типа CDProduct методу, требующему Chargeable,

```
// Листинг 4.19
public function addChargeableItem(Chargeable $item)
{
    // Все, что мы знаем о $item, — что это объект
    // Chargeable. Что он является также объектом
    // CDProduct, значения не имеет.
    // Мы можем быть уверены только в наличии getPrice()
    //
    //...
}
```

то ему ничего не будет известно обо всех методах, определенных в классе ShopProduct или CDProduct. Но данный метод интересует лишь, поддерживается ли в аргументе $item метод getPrice().

Интерфейс можно реализовать в любом классе (на самом деле в классе можно реализовать любое количество интерфейсов), поэтому с помощью интерфейсов можно эффективно объединять типы данных, не связанных никакими другими отношениями. В итоге мы можем определить совершенно новый класс, в котором реализован интерфейс Chargeable:

```
// Листинг 4.20
class Shipping implements Chargeable
{
    public function __construct(private float $price)
    {
    }
    public function getPrice(): float
    {
        return $this->price;
    }
}
```

Теперь объект типа Shipping можно передать методу addChargeableItem() таким же образом, как и объект типа ShopProduct. Для клиента, оперирующего объектом типа Chargeable, очень важно, что он может вызвать метод getPrice(). Любые другие имеющиеся методы связаны с иными типами, будь то через собственный класс объекта, суперкласс или другой интерфейс. Но они не имеют никакого отношения к клиенту.

В классе можно не только расширить суперкласс, но и реализовать любое количество интерфейсов. При этом ключевое слово `extends` должно предшествовать ключевому слову `implements`:

```
// Листинг 4.21
class Consultancy extends TimedService implements Bookable, Chargeable
{
    // ...
}
```

Обратите внимание на то, что в классе `Consultancy` реализованы два интерфейса, а не один. После ключевого слова `implements` можно перечислить через запятую несколько интерфейсов.

В языке PHP поддерживается наследование только от одного родителя (так называемое *одиночное*, или *простое*, наследование), поэтому после ключевого слова `extends` можно указать только одно имя базового класса.

Трейты

В отличие от языка C++, в PHP, как и в языке Java, не поддерживается множественное наследование. Но эту проблему можно частично решить с помощью интерфейсов, как было показано в предыдущем разделе. Иными словами, для каждого класса в PHP может существовать только один родительский класс. Тем не менее в каждом классе можно реализовать произвольное количество интерфейсов. При этом данный класс будет принимать типы всех тех интерфейсов, которые в нем реализованы.

Как видите, с помощью интерфейсов создаются новые типы объектов без их реализации. Но что делать, если требуется реализовать ряд общих методов для всей иерархии наследования классов? Для этой цели в версии PHP 5.4 было введено понятие *трейтов*[1].

По существу, трейты напоминают классы, экземпляры которых нельзя получить, но можно включить в другие классы. Поэтому любое свойство (или метод), определенное в трейте, становится частью того класса, в который включен этот трейт. При этом трейт изменяет структуру данного класса, но не меняет его тип. Трейты можно считать своего рода включени-

[1] От англ. *trait* (особенность, характерная черта), но в литературе это понятие обозначается термином *типаж* или просто *трейт*. Здесь и далее употребляется термин *трейт*, тем более что к моменту издания данной книги он уже был официально принят в русскоязычном сообществе разработчиков приложений на PHP. — *Примеч. перев.*

ями в классы. Рассмотрим на конкретных примерах, насколько полезными могут быть трейты.

Проблема, которую позволяют решить трейты

Ниже приведена версия класса ShopProduct, в который был добавлен метод calculateTax():

```
// Листинг 4.22
class ShopProduct
{
    private int $taxrate = 20;
    // ...
    public function calculateTax(float $price): float
    {
        return (($this->taxrate / 100) * $price);
    }
}
```

Методу calculateTax() в качестве параметра $price передается цена товара, и он вычисляет налог с продаж на основании ставки, величина которой хранится во внутреннем свойстве $taxrate.

Безусловно, метод calculateTax() будет доступен всем подклассам данного класса. Но что если дело дойдет до совершенно других иерархий классов? Представьте класс UtilityService, унаследованный от другого класса Service. Если в классе UtilityService потребуется определить величину налога по точно такой же формуле, то нам не останется ничего другого, как полностью скопировать тело метода calculateTax(). Это класс Service:

```
// Листинг 4.23
abstract class Service
{
    // Некоторый код...
}
```

А вот как выглядит UtilityService:

```
// Листинг 4.24
class UtilityService extends Service
{
    private int $taxrate = 20;
    public function calculateTax(float $price): float
    {
```

```
        return (($this->taxrate / 100) * $price);
    }
}
```

Поскольку `UtilityService` и `ShopProduct` не имеют никаких общих базовых классов, они не могут легко использовать общую реализацию `CalculateTax()`. Поэтому мы вынуждены копировать и вставлять нашу реализацию из одного класса в другой.

Определение и применение трейтов

Одной из целей объектно-ориентированного проектирования, которая красной нитью проходит через всю эту книгу, является устранение проблемы дублирования кода. Как будет показано в главе 11, "Выполнение задач и представление результатов", одним из возможных путей решения этой проблемы является вынесение общих фрагментов кода в отдельные повторно используемые глобальные классы. Трейты также позволяют решить данную проблему — хотя и менее изящно, но, без сомнения, эффективно.

В приведенном ниже примере кода сначала объявляется простой трейт, содержащий метод `calculateTax()`, а затем этот трейт включается в оба класса — и `ShopProduct`, и `UtilityService`:

```
// Листинг 4.25
trait PriceUtilities
{
    private $taxrate = 20;
    public function calculateTax(float $price): float
    {
        return (($this->taxrate / 100) * $price);
    }
    // Другие служебные методы
}
```

Как видите, трейт `PriceUtilities` объявляется с помощью ключевого слова `trait`. Тело трейта весьма напоминает тело обычного класса, в котором в фигурных скобках просто указывается ряд методов (или свойств, как будет показано ниже). Как только трейт `PriceUtilities` объявлен, им можно пользоваться в собственных классах. Для этой цели служит ключевое слово `use`, после которого указывается имя трейта. Итак, объявив и реализовав в одном месте метод `calculateTax()`, им можно далее воспользоваться в классе `ShopProduct`:

```
// Листинг 4.26
use popp\ch04\batch06_1\PriceUtilities;
class ShopProduct
{
    use PriceUtilities;
}
```

Конечно же, его можно добавить и в класс `UtilityService`:

```
// Листинг 4.27
class UtilityService extends Service
{
    use PriceUtilities;
}
```

Теперь, когда я вызываю эти классы, я знаю, что они без дублирования разделяют реализацию `PriceUtilities`. Если бы в `PriceUtilities` обнаружилась ошибка, ее можно было бы исправить в единственном месте:

```
// Листинг 4.28
$p = new ShopProduct();
print $p->calculateTax(100) . "\n";
$u = new UtilityService();
print $u->calculateTax(100) . "\n";
```

Применение нескольких трейтов

В класс можно включить несколько трейтов, перечислив их через запятую после ключевого слова `use`. В приведенном ниже примере кода сначала определяется, а затем используется новый трейт `IdentityTrait` в классе `ShopProduct` наряду с трейтом `PriceUtilities`:

```
// Листинг 4.29
trait IdentityTrait
{
    public function generateId(): string
    {
        return uniqid();
    }
}
```

Применяя оба трейта — `PriceUtilities` и `IdentityTrait` — с помощью ключевого слова `use`, я делаю методы `calculateTax()` и `generateId()` доступными для класса `ShopProduct`. Это означает, что эти методы становятся членами класса `ShopProduct`:

```
// Листинг 4.30
class ShopProduct
{
    use PriceUtilities;
    use IdentityTrait;
}
```

На заметку В трейте `IdentityTrait` реализован метод `generateId()`. В действительности однозначные значения идентификаторов объектов часто извлекаются из базы данных, но в целях тестирования иногда требуется использовать их локальные реализации. Более подробно об объектах, базах данных и однозначных идентификаторах речь пойдет в главе 13, "Шаблоны баз данных", где описывается шаблон Identity Map, а о тестировании и имитации функциональных возможностей объектов — в главе 18, "Тестирование средствами PHPUnit".

Теперь в классе `ShopProduct` я могу вызывать как метод `generateId()`, так и метод `CalculateTax()`:

```
// Листинг 4.31
$p = new ShopProduct();
print $p->calculateTax(100) . "\n";
print $p->generateId() . "\n";
```

Сочетание трейтов с интерфейсами

Несмотря на то что польза от трейтов не вызывает особых сомнений, они не позволяют изменить тип класса, в который они включены. Так, если трейт `IdentityTrait` используется сразу в нескольких классах, у них все равно не будет общего типа, который можно было бы указать в сигнатурах методов.

К счастью, трейты можно удачно сочетать с интерфейсами. В частности, сначала можно определить интерфейс с сигнатурой метода `generateId()`, а затем объявить, что в классе `ShopProduct` реализуются методы этого интерфейса:

```
// Листинг 4.32
interface IdentityObject
{
    public function generateId(): string;
}
```

Если я хочу, чтобы ShopProduct соответствовал типу IdentityObject, я должен сделать его реализующим интерфейс IdentityObject:

```
// Листинг 4.33
class ShopProduct implements IdentityObject
{
    use PriceUtilities;
    use IdentityTrait;
}
```

Здесь, как и в предыдущем примере, в классе ShopProduct используется трейт IdentityTrait. Но импортируемый с его помощью метод generateId() теперь удовлетворяет также требованиям интерфейса IdentityObject. А это означает, что объекты типа ShopProduct можно передавать тем методам и функциям, в описании аргументов которых указывается тип интерфейса IdentityObject:

```
// Листинг 4.34
public static function storeIdentityObject(IdentityObject $idobj)
{
    // Сделать что-нибудь с использованием IdentityObject
}
```

Устранение конфликтов имен методов с помощью ключевого слова insteadof

Безусловно, сочетать трейты с интерфейсами — это замечательно, но рано или поздно может возникнуть конфликт имен. Что, например, произойдет, если в обоих включаемых трейтах будет реализован метод calculateTax(), как показано ниже:

```
// Листинг 4.35
trait TaxTools
{
    public function calculateTax(float $price): float
    {
        return 222;
    }
}
```

В связи с тем, что в один класс были включены два трейта, содержащие методы calculateTax(), интерпретатор PHP не сможет продолжить работу, поскольку он не в состоянии определить, какой из методов дол-

жен перекрыть другой метод. В итоге выводится следующее сообщение об ошибке:

```
Fatal error: Trait method popp\ch04\batch06_3\TaxTools::calculateTax
has not been applied as
popp\ch04\batch06_3\UtilityService::calculateTax,
 because of collision with
popp\ch04\batch06_3\PriceUtilities::calculateTax in...
```

Для устранения подобного конфликта служит ключевое слово `insteadof`:

```
// Листинг 4.36
class UtilityService extends Service
{
    use PriceUtilities;
    use TaxTools
    {
        TaxTools::calculateTax insteadof PriceUtilities;
    }
}
```

Чтобы применять директивы с инструкцией `use`, необходимо дополнить их блоком кода в фигурных скобках, в котором употребляется ключевое слово `insteadof`. Слева от этого ключевого слова указывается полностью определенное имя метода, состоящее из имени трейта и метода. Оба имени разделяются двумя двоеточиями, играющими в данном случае роль оператора определения области видимости. Справа от ключевого слова `insteadof` указывается имя трейта, метод которого с аналогичным именем должен быть заменен. Таким образом, конструкция

```
TaxTools::calculateTax insteadof PriceUtilities;
```

означает, что метод `calculateTax()` из трейта `TaxTools` следует использовать вместо одноименного метода из трейта `PriceUtilities`.

Поэтому при выполнении кода

```
// Листинг 4.37
$u = new UtilityService();
print $u->calculateTax(100) . "\n";
```

будет выведено число `222`, жестко закодированное в теле метода `TaxTools::calculateTax()`.

Назначение псевдонимов переопределенным методам трейта

Как было показано выше, с помощью ключевого слова `insteadof` можно устранить конфликт имен методов, принадлежащих разным трейтам. Но что если понадобится вызвать переопределяемый метод трейта? Для этой цели служит ключевое слово `as`, которое позволяет назначить вызываемому методу псевдоним. Как и в конструкции с ключевым словом `insteadof`, слева от ключевого слова `as` следует указать полностью определенное имя метода, а справа — псевдоним имени метода. В приведенном ниже примере кода метод `calculateTax()` из трейта `PriceUtilities` восстановлен под новым именем — `basicTax()`:

```
// Листинг 4.38
class UtilityService extends Service
{
    use PriceUtilities;
    use TaxTools
    {
        TaxTools::calculateTax insteadof PriceUtilities;
        PriceUtilities::calculateTax as basicTax;
    }
}
```

Теперь класс `UtilityService` приобрел два метода: версию `TaxTools` метода `CalculateTax()` и версию из `PriceUtilities` с псевдонимом `basicTax()`. Давайте вызовем эти методы:

```
// Листинг 4.39
$u = new UtilityService();
print $u->calculateTax(100) . "\n";
print $u->basicTax(100) . "\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
222
17
```

Таким образом, метод из трейта `PriceUtilities::calculateTax()` стал частью класса `UtilityService` под именем `basicTax()`.

На заметку При возникновении конфликта имен между методами разных трейтов недостаточно просто назначить одному из методов псевдоним в блоке use. Сначала вы должны решить, какой из методов должен быть замещен, и указать это с помощью ключевого слова insteadof. Затем замещенному методу можно назначить псевдоним с помощью ключевого слова as и восстановить его в классе под новым именем.

Здесь уместно отметить, что псевдонимы имен методов можно использовать даже в отсутствие конфликта имен. Так, с помощью метода трейта можно реализовать абстрактный метод с сигнатурой, объявленной в родительском классе или интерфейсе.

Применение статических методов в трейтах

В большинстве рассмотренных до сих пор примеров применялись статические методы, поскольку для их вызова не требуются экземпляры класса. Ничто не мешает ввести статический метод и в трейт. В приведенном ниже примере кода объявление свойства PriceUtilities::$taxrate и метода PriceUtilities::calculateTax() было изменено таким образом, чтобы они стали статическими:

```
// Листинг 4.40
trait PriceUtilities
{
    private static int $taxrate = 20;
    public static function calculateTax(float $price): float
    {
        return ((self::$taxrate / 100) * $price);
    }
    // Другие служебные методы
}
```

Вот UtilityService в его минимальной форме:

```
// Листинг 4.41
class UtilityService extends Service
{
    use PriceUtilities;
}
```

Все, что делается, — ключевое слово use применяется для свойства PriceUtilities. Однако имеется ключевое отличие при вызове метода CalculateTax():

```
// Листинг 4.42
print UtilityService::calculateTax(100) . "\n";
```

Теперь я должен вызвать метод класса, а не объекта. Как и следовало ожидать, этот скрипт выводит следующее:

```
20
```

Таким образом, статические методы, объявленные в трейтах, оказываются доступными обычным образом из базового класса.

Доступ к свойствам класса-хоста

После рассмотрения приведенных выше примеров у вас могло сложиться впечатление, что для работы с трейтами подходят только статические методы. И даже те методы трейта, которые не описаны как статические, по существу являются статическими, не так ли? На самом деле нет, поскольку доступными оказываются также свойства и методы класса-хоста:

```
// Листинг 4.43
trait PriceUtilities
{
    public function calculateTax(float $price): float
    {
        // Насколько хорош такой проект?
        return (($this->taxrate / 100) * $price);
    }
    // Другие служебные методы
}
```

В приведенном фрагменте кода трейт `PriceUtilities` был откорректирован таким образом, чтобы из него можно было обращаться к свойству класса-хоста. Вот как выглядит исправленный класс-хост:

```
// Листинг 4.44
class UtilityService extends Service
{
    use PriceUtilities;
    public $taxrate = 20;
}
```

И если такое проектное решение покажется вам неудачным, то вы будете совершенно правы. Скажем больше: оно крайне неудачно! Несмотря на то что обращение из трейтов к данным, находящимся в базовом классе, является обычной практикой, объявлять свойство `$taxrate` в классе

UtilityService не было никаких веских оснований. Ведь не стоит забывать, что трейты могут использоваться во многих совершенно разных классах. И кто может дать гарантию (или даже обещание!), что в каждом базовом классе будет объявлено свойство $taxrate?

С другой стороны, было бы неплохо заключить следующий контракт: "Если вы пользуетесь данным трейтом, то обязаны предоставить в его распоряжение определенные ресурсы".

На самом деле достичь такого же результата можно благодаря тому, что в трейтах поддерживаются абстрактные методы.

Определение абстрактных методов в трейтах

Абстрактные методы можно объявлять в трейтах так же, как и в обычных классах. Чтобы воспользоваться таким трейтом в классе, в нем следует реализовать все объявленные в трейте абстрактные методы.

На заметку До PHP 8 сигнатуры абстрактных методов, определенных в трейтах, не всегда были полностью применимы. Это означало, что в некоторых обстоятельствах типы аргументов и возвращаемые типы могли отличаться в классе реализации от тех, которые установлены в объявлении абстрактного метода. Сейчас эта лазейка закрыта.

Вооруженные этим знанием, мы можем переписать предыдущий пример таким образом, чтобы трейт вынуждал любой использующий его класс предоставлять сведения о ставке налога:

```
// Листинг 4.45
trait PriceUtilities
{
    public function calculateTax(float $price): float
    {
        // Лучшее решение — мы знаем, что getTaxRate() реализован
        return (($this->getTaxRate() / 100) * $price);
    }
    abstract public function getTaxRate(): float;
    // Другие служебные методы
}
```

Объявив абстрактный метод getTaxRate() в трейте PriceUtilities, мы вынуждаем программиста клиентского кода обеспечить его реализацию в классе UtilityService:

```
// Листинг 4.46
class UtilityService extends Service
{
    use PriceUtilities;
    public function getTaxRate(): float
    {
        return 20;
    }
}
```

Если бы я не предоставил метод getTaxRate(), то благодаря абстрактному объявлению в трейте я получил бы сообщение об ошибке.

Изменение прав доступа к методам трейта

Безусловно, ничто не может помешать вам объявить методы трейта открытыми (public), защищенными (private) или закрытыми (protected). Но вы можете изменить эти модификаторы доступа к методам и непосредственно в том классе, в котором используется трейт. Как было показано выше, методу можно назначить псевдоним с помощью ключевого слова as. Если справа от этого ключевого слова указать новый модификатор доступа, то вместо назначения псевдонима будет изменен уровень доступа к методу.

Допустим, что метод calculateTax() требуется использовать только в самом классе UtilityService и требуется запретить вызов этого метода из клиентского кода. Ниже показано, какие изменения следует внести для этого в инструкцию use:

```
// Листинг 4.47
class UtilityService extends Service
{
    use PriceUtilities
    {
        PriceUtilities::calculateTax as private;
    }
    public function __construct(private float $price)
    {
    }
    public function getTaxRate(): float
    {
        return 20;
    }
```

```
    public function getFinalPrice(): float
    {
        return ($this->price + $this->calculateTax($this->price));
    }
}
```

Чтобы запретить доступ к методу `calculateTax()` за пределами класса `UtilityService`, после ключевого слова `as` в инструкции `use` был указан модификатор `private`. В итоге доступ к этому методу стал возможен только из метода `getFinalPrice()`. Если теперь попытаться вызвать метод `calculateTax()` за пределами его класса, например как

```
// Листинг 4.48
$u = new UtilityService(100);
print $u->calculateTax() . "\n";
```

то, как и ожидалось, будет выведено следующее сообщение об ошибке:

```
Error: Call to private method popp\ch04\batch06_9\
UtilityService::calculateTax() from context ...
```

Позднее статическое связывание: ключевое слово `static`

А теперь, после того как были рассмотрены абстрактные классы, трейты и интерфейсы, самое время снова ненадолго вернуться к статическим методам. Как вам, должно быть, уже известно, статический метод можно использовать в качестве фабричного для создания экземпляров того класса, в котором содержится данный метод. Если читатель такой же ленивый программист, как и автор данной книги, то его должно раздражать дублирование кода, аналогичное приведенному ниже:

```
// Листинг 4.49
abstract class DomainObject
{
}

// Листинг 4.50
class User extends DomainObject
{
    public static function create(): User
    {
        return new User();
    }
}
```

```
// Листинг 4.51
class Document extends DomainObject
{
    public static function create(): Document
    {
        return new Document();
    }
}
```

Сначала в данном примере кода был создан суперкласс под именем `DomainObject`. Само собой разумеется, что в реальном проекте в нем будут доступны функциональные возможности, общие для всех дочерних классов. После этого были созданы два дочерних класса, `User` и `Document`, и было бы желательно, чтобы в каждом из них имелся статический метод `create()`.

На заметку Почему для создания конкретного объекта был использован статический фабричный метод, а не оператор `new` и конструктор объекта? В главе 13, "Шаблоны баз данных", приведено описание проектного шаблона Identity Map. Компонент Identity Map создает и инициализирует новый объект только в том случае, если объект с аналогичными отличительными особенностями еще не создан. Если же такой объект существует, то возвращается ссылка на него. Статический фабричный метод наподобие `create()` является отличным кандидатом для реализации функциональности такого рода.

Приведенный выше код вполне работоспособен, но ему присущ досадный недостаток — дублирование. Вряд ли кому-то понравится повторять такой стереотипный код для создания каждый раз объекта класса, производного от класса `DomainObject`. Вместо этого можно попытаться переместить метод `create()` в суперкласс:

```
// Листинг 4.52
abstract class DomainObject
{
    public static function create(): DomainObject
    {
        return new self();
    }
}
```

Так *выглядит* лучше! Теперь весь общий код сосредоточен в одном месте, а для обращения к текущему классу использовано ключевое слово `self`. Однако насчет ключевого слова `self` было сделано неверное предположение, что оно *должно* работать. Но на самом деле оно *не* работает для

классов так же, как псевдопеременная `$this` — для объектов. С помощью ключевого слова `self` *нельзя* обратиться к контексту вызванного класса. Оно используется только для ссылок на класс, в контексте которого вызывается метод. Поэтому при попытке запуска приведенного выше примера получим следующее сообщение об ошибке:

```
Error: Cannot instantiate abstract class
popp\ch04\batch06\DomainObject
```

Таким образом, ссылка `self` преобразуется в ссылку на класс `DomainObject`, в котором определен метод `create()`, а не на класс `Document`, для которого этот метод должен быть вызван. До версии 5.3 это было серьезным ограничением языка PHP, которое породило массу неуклюжих обходных решений. Но в версии PHP 5.3 было впервые введено понятие *позднего статического связывания*. Самым заметным его проявлением является введение нового (в данном контексте) ключевого слова `static`. Оно аналогично ключевому слову `self`, за исключением того, что оно относится к *вызывающему*, а не к *содержащему* классу. В данном случае это означает, что в результате вызова метода `Document::create()` возвращается новый объект типа `Document` и не будет предпринята безуспешная попытка создать объект типа `DomainObject`.

Итак, воспользуемся всеми преимуществами наследования в статическом контексте следующим образом:

```
// Листинг 4.53
abstract class DomainObject
{
    public static function create(): DomainObject
    {
        return new static();
    }
}

// Листинг 4.54
class User extends DomainObject
{
}

// Листинг 4.55
class Document extends DomainObject
{
}
```

Теперь, если мы вызовем `create()` для одного из дочерних классов, мы не должны получить сообщение об ошибке. Метод должен вернуть объект, связанный с классом, который мы *вызвали*, а не с классом, в котором находится `create()`:

```
// Листинг 4.56
print_r(Document::create());
```

В результате выполнения приведенного выше кода будет выведено следующее:

```
popp\ch04\batch07\Document Object
(
)
```

Ключевое слово `static` можно использовать не только для создания объектов. Подобно ключевым словам `self` и `parent`, его можно использовать в качестве идентификатора для вызова статических методов даже из нестатического контекста. Допустим, требуется реализовать идею группирования классов типа `DomainObject`. По умолчанию в моей новой классификации все эти классы подпадают в категорию `"default"`, но я предпочитаю иметь возможность перекрытия этой группировки в некоторых ветвях иерархии наследования этих классов. Ниже показано, как этого добиться:

```
// Листинг 4.57
abstract class DomainObject
{
    private string $group;
    public function __construct()
    {
        $this->group = static::getGroup();
    }
    public static function create(): DomainObject
    {
        return new static();
    }
    public static function getGroup(): string
    {
        return "default";
    }
}

// Листинг 4.58
class User extends DomainObject
{
}
```

```
// Листинг 4.59
class Document extends DomainObject
{
    public static function getGroup(): string
    {
        return "document";
    }
}

// Листинг 4.60
class SpreadSheet extends Document
{
}

// Листинг 4.61
print_r(User::create());
print_r(SpreadSheet::create());
```

Здесь в класс `DomainObject` был добавлен конструктор, в котором ключевое слово `static` используется для вызова метода `getGroup()`. Его стандартная реализация выполняется в классе `DomainObject`, но переопределяется в классе `Document`. Кроме того, в данном примере кода был создан новый класс `SpreadSheet`, расширяющий класс `Document`. Выполнение данного фрагмента кода приведет к следующему результату:

```
popp\ch04\batch07\User Object (
    [group:popp\ch04\batch07\DomainObject:private] => default
)
popp\ch04\batch07\SpreadSheet Object (
    [group:popp\ch04\batch07\DomainObject:private] => document
)
```

Все происходящее с классом `User` не настолько очевидно и поэтому требует некоторых пояснений. В конструкторе класса `DomainObject` вызывается метод `getGroup()`, который интерпретатор PHP находит в текущем классе. Но что касается класса `SpreadSheet`, то поиск метода `getGroup()` начинается не с класса `DomainObject`, а с самого вызываемого класса `SpreadSheet`. А поскольку в классе `SpreadSheet` реализация метода `getGroup()` не предусмотрена, интерпретатор PHP вызывает аналогичный метод из класса `Document`, следуя вверх по иерархии классов. До внедрения позднего статического связывания в версии PHP 5.3 здесь возникало затруднение в связи с применением ключевого слова `self`, по которому поиск метода `getGroup()` производился только в классе `DomainObject`.

Обработка ошибок

Иногда все идет не так, как надо. Файлы где-то потерялись, объекты для связи с серверами баз данных остались не инициализированными, URL изменились, XML-файлы были повреждены, права доступа установлены неправильно, лимиты на выделенное дисковое пространство превышены... Этот список можно продолжать до бесконечности. Если пытаться предусмотреть любое осложнение в простом методе, он может просто утонуть под тяжестью собственного кода обработки ошибок.

Ниже приведено определение простого класса `Conf`, в котором сохраняются, извлекаются и устанавливаются данные в XML-файле конфигурации:

```
// Листинг 4.62
class Conf
{
    private \SimpleXMLElement $xml;
    private \SimpleXMLElement $lastmatch;
    public function __construct(private string $file)
    {
        $this->xml = simplexml_load_file($file);
    }
    public function write(): void
    {
        file_put_contents($this->file, $this->xml->asXML());
    }
    public function get(string $str): ? string
    {
        $matches = $this->xml->xpath("/conf/item[@name=\"$str\"]");

        if (count($matches))
        {
            $this->lastmatch = $matches[0];
            return (string)$matches[0];
        }

        return null;
    }
    public function set(string $key, string $value): void
    {
        if (! is_null($this->get($key)))
        {
            $this->lastmatch[0] = $value;
            return;
        }
```

```
        $conf = $this->xml->conf;
        $this->xml->addChild('item', $value)
             ->addAttribute('name', $key);
    }
}
```

Для доступа к парам “имя–значение” в классе Conf используется расширение SimpleXml. Ниже приведен фрагмент из файла конфигурации в формате XML, который обрабатывается в классе Conf:

```
<?xml version="1.0"?>
<conf>
    <item name="user">bob</item>
    <item name="pass">newpass</item>
    <item name="host">localhost</item>
</conf>
```

Конструктору класса Conf передается имя файла конфигурации, которое далее передается функции simplexml_load_file(). Возвращаемый из этой функции объект типа SimpleXmlElement сохраняется в свойстве $xml. Для обнаружения элемента item с заданным атрибутом name метод get() использует XPath. Значение найденного элемента возвращается в вызывающий код. Метод set() изменяет значение существующего элемента или создает новый элемент. И, наконец, метод write() сохраняет данные о новой конфигурации в файле на диске.

Как и код из многих приведенных выше примеров, код класса Conf сильно упрощен. В частности, в нем не предусмотрена обработка ситуаций, когда файла конфигурации не существует или в него нельзя записать данные. Этот код также слишком “оптимистичен”. В нем предполагается, что XML-документ правильно отформатирован и содержит ожидаемые элементы.

Организовать проверку подобных ошибок совсем нетрудно, но нам нужно решить, как реагировать на них, если они возникнут. В целом у нас имеются для этого следующие возможности.

Во-первых, завершить выполнение программы. Это простой, но радикальный выход из положения. В итоге вся вина за неудачное завершение программы падет на скромный класс Conf. Несмотря на то что такие методы, как __construct() и write(), удачно расположены в исходном коде класса Conf для обнаружения ошибок, им недостает информации, чтобы решить, как обрабатывать эти ошибки.

Во-вторых, вместо обработки ошибки в классе Conf можно возвратить признак ошибки в том или ином виде. Это может быть логическое или це-

лое значение, например `0` или `-1`. В некоторых классах можно также сформировать текстовое сообщение об ошибке или набор специальных признаков, чтобы клиентский код мог запросить больше информации в случае неудачного завершения программы.

Оба упомянутых выше подхода удачно сочетаются во многих пакетах PEAR, в которых возвращается объект ошибки (экземпляр класса `PEAR_Error`). Наличие этого объекта свидетельствует о том, что произошла ошибка, а подробные сведения о ней содержатся в самом объекте. И хотя применять такой способ обработки ошибок в настоящее время не рекомендуется, многие классы до сих пор так и не были обновлены из-за того, что клиентский код зачастую полагается на старые стандарты поведения при возникновении ошибок.

Трудность состоит еще и в том, что возвращаемое значение может быть искажено. И тогда приходится полагаться на то, что клиентский код будет проверять тип возвращаемого объекта после каждого вызова метода, подверженного ошибкам. А это довольно рискованно. Доверять нельзя никому!

Когда вызывающему коду из метода возвращается значение ошибки, нет никакой гарантии, что клиентский код будет оснащен лучше данного метода и сможет решить, как обрабатывать эту ошибку. А если он не сможет это сделать, то осложнения будут появляться снова и снова. Поэтому в клиентском методе следует определить, как реагировать на ошибочную ситуацию, а возможно, даже реализовать иную стратегию извещения об ошибке.

Исключения

В версии PHP 5 было введено понятие исключения (exception), представляющее собой совершенно новый способ обработки ошибок. Имеется в виду, что он совершенно новый для языка PHP, но если у вас есть опыт программирования на Java или C++, то понятие исключения вам знакомо. Применение исключений позволяет разрешить все описанные выше трудности, возникающие в связи с обработкой ошибок.

Исключение — это специальный объект, который является экземпляром встроенного класса `Exception` или производного от него класса. Объекты типа `Exception` предназначены для хранения информации об ошибках и выдачи сообщений о них. Конструктору класса `Exception` передаются два необязательных аргумента: строка сообщения и код ошибки. В этом клас-

се существуют также некоторые полезные методы для анализа ошибочной ситуации, перечисленные в табл. 4.1.

Таблица 4.1. *Открытые методы класса* `Exception`

Метод	Назначение
`getMessage()`	Получить строку сообщения, переданную конструктору
`getCode()`	Получить целочисленный код ошибки, который был передан конструктору
`getFile()`	Получить имя файла, в котором было сгенерировано исключение
`getLine()`	Получить номер строки, в которой было сгенерировано исключение
`getPrevious()`	Получить вложенный объект `Exception`
`getTrace()`	Получить многомерный массив, отслеживающий вызовы метода, которые привели к исключению, в том числе имя метода, класса, файла и значение аргумента
`getTraceAsString()`	Получить строковый вариант данных, возвращаемых методом `getTrace()`
`__toString()`	Вызывается автоматически, когда объект типа `Exception` используется в контексте символьной строки. Возвращает строку, подробно описывающую возникшее исключение

Классом `Exception` очень удобно пользоваться для извещения об ошибках и предоставления отладочной информации (в этом отношении особенно полезны методы `getTrace()` и `getTraceAsString()`). На самом деле класс `Exception` почти идентичен обсуждавшемуся ранее классу `PEAR_Error`, но исключение — это гораздо больше, чем просто хранящаяся в нем информация.

Генерация исключений

Вместе с объектом типа `Exception` используется инструкция с ключевым словом `throw`. Эта инструкция останавливает выполнение текущего метода и передает ответственность за обработку ошибок обратно вызывающему коду. Внесем приведенные ниже коррективы в метод-конструктор `__construct()`, чтобы воспользоваться инструкцией `throw`:

```
// Листинг 4.63
public function __construct(private string $file)
{
    if (! file_exists($file))
    {
        throw new \Exception("Файл '{$file}' не существует");
    }

    $this->xml = simplexml_load_file($file);
}
```

Аналогичной конструкцией можно воспользоваться и в методе `write()`:

```
// Листинг 4.64
public function write(): void
{
    if (! is_writeable($this->file))
    {
      throw new \Exception("Нельзя писать в файл '{$this->file}'");
    }

    print "{$this->file} доступен для записи\n";
    file_put_contents($this->file, $this->xml->asXML());
}

// Листинг 4.65
try
{
    $conf = new Conf("/tmp/conf01.xml");
    //$conf = new Conf( "/root/unwriteable.xml" );
    //$conf = new Conf( "nonexistent/not_there.xml" );
    print "user: " . $conf->get('user') . "\n";
    print "host: " . $conf->get('host') . "\n";
    $conf->set("pass", "newpass");
    $conf->write();
}
catch (\Exception $e)
{
    // Обработка ошибки тем или иным способом
}
```

Как видите, блок `catch` на первый взгляд напоминает объявление метода. Когда генерируется исключение, управление передается блоку `catch` в области видимости вызова. Объект `Exception` автоматически передается в качестве аргумента.

Выполнение сгенерировавшего исключение метода прекращается, и управление передается непосредственно блоку catch. Здесь вы можете выполнять любые доступные для вас действия по восстановлению после ошибки. Если это возможно, избегайте инструкции die. Так вы усложняете тестирование и можете помешать другому коду вашей системы выполнить необходимые операции очистки. Если вы не в состоянии восстановить работу после ошибки, вы можете сгенерировать новое исключение:

```
// Листинг 4.66
}
catch (\Exception $e)
{
    // Обработка ошибки тем или иным способом
    // или
    throw new \Exception("Ошибка конфигурации " . $e->getMessage());
}
```

Вы можете также сгенерировать обрабатываемое исключение повторно:

```
// Листинг 4.67
try
{
    $conf = new Conf("nonexistent/not_there.xml");
}
catch (\Exception $e)
{
    // Обработка ошибки тем или иным способом
    // или повторная генерация того же исключения
    throw $e;
}
```

Если у вас нет необходимости в самом объекте Exception при обработке ошибки, вы можете, начиная с PHP 8, просто опустить аргумент и ограничиться указанием типа:

```
// Листинг 4.68
try
{
    $conf = new Conf("nonexistent/not_there.xml");
}
catch (\Exception)
{
    // Обработка ошибки без применения объекта Exception
}
```

Создание подклассов класса `Exception`

Классы, расширяющие класс `Exception`, можно создавать таким же образом, как и любой другой класс, определенный пользователем. Имеются две причины, по которым может возникнуть потребность в такой дополнительной классификации исключений. Во-первых, можно расширить функциональные возможности класса `Exception`. И во-вторых, производный от него класс определяет новый тип класса, который может оказать помощь в самой обработке ошибок.

Дело в том, что для одной инструкции `try` можно определить столько блоков `catch`, сколько потребуется. Вызов конкретного блока `catch` будет зависеть от типа сгенерированного исключения и объявления типа класса, указанного в списке аргументов. Давайте определим ряд простых классов, расширяющих класс `Exception`:

```
// Листинг 4.69

class XmlException extends \Exception
{
    public function construct(private \LibXmlError $error)
    {
        $shortfile = basename($error->file);
        $msg = "[{$shortfile}, line {$error->line}," .
               "col {$error->column}] {$error->message}";
        $this->error = $error;
        parent:: __construct($msg, $error->code);
    }
    public function getLibXmlError(): \LibXmlError
    {
        return $this->error;
    }
}

// Листинг 4.70
class FileException extends \Exception
{
}

// Листинг 4.71
class ConfException extends \Exception
{
}
```

Объект класса LibXmlError создается автоматически, когда SimpleXml обнаруживает поврежденный XML-файл. У объекта есть свойства message и code, и он напоминает объект класса Exception. Благодаря этому сходству объект типа LibXmlError удачно используется в классе XmlException. У классов FileException и ConfException функциональных возможностей не больше, чем у любого другого подкласса, производного от класса Exception. Воспользуемся теперь этими классами в следующем примере кода, чтобы немного откорректировать методы __construct() и write() в классе Conf:

```
// Листинг 4.72

// Conf class...
public function __construct(private string $file)
{
    if (!file_exists($file))
    {
        throw new FileException("Файл '$file' не существует");
    }
    $this->xml = simplexml_load_file($file, null, LIBXML_NOERROR);
    if (!is_object($this->xml))
    {
        throw new XmlException(libxml_get_last_error());
    }
    $matches = $this->xml->xpath("/conf");
    if (!count($matches))
    {
        throw new ConfException("Не найден корневой элемент");
    }
}
public function write(): void
{
    if (!is_writeable($this->file))
    {
        throw new FileException("Нельзя писать в '{$this->file}'");
    }
    file_put_contents($this->file, $this->xml->asXML());
}
```

Метод-конструктор __construct() генерирует исключения типа XmlException, FileException или ConfException в зависимости от вида возникшей ошибки. Обратите внимание на то, что методу simplexml_load_file() передается флаг LIBXML_NOERROR, блокирующий выдачу предупреждений и дающий свободу действий для последующей обработки

этих предупреждений средствами класса XmlException. Так, если обнаружится поврежденный XML-файл, метод simplexml_load_file() уже не возвратит объект типа SimpleXMLElement. Получить доступ к ошибке можно будет с помощью метода libxml_get_last_error().

Метод write() генерирует исключение типа FileException, если свойство $file указывает на файл, недоступный для записи.

Итак, мы установили, что метод-конструктор __construct() может генерировать одно из трех возможных исключений. Как же этим воспользоваться? Ниже приведен пример кода, в котором создается экземпляр объекта Conf:

```
// Листинг 4.73
class Runner
{
    public static function init()
    {
        try
        {
            $conf = new Conf(__DIR__ . "/conf.broken.xml");
            print "user: " . $conf->get('user') . "\n";
            print "host: " . $conf->get('host') . "\n";
            $conf->set("pass", "newpass");
            $conf->write();
        }
        catch (FileException $e)
        {
            // Файл не существует или недоступен
        }
        catch (XmlException $e)
        {
            // Поврежденный xml
        }
        catch (ConfException $e)
        {
            // Неверный формат XML-файла
        }
        catch (\Exception $e)
        {
            // Ловушка: этот код не должен вызываться
        }
    }
}
```

В данном примере мы предусмотрели блок catch для каждого типа класса ошибки. Вызов конкретного блока catch будет зависеть от типа сгенери-

рованного исключения. При этом будет выполнен первый подходящий блок. Поэтому не следует забывать, что самый общий тип перехватываемой ошибки необходимо указывать в конце последовательности блоков `catch`, а самый специализированный — в ее начале. Так, если разместить блок `catch` для обработки исключения типа `Exception` перед аналогичным операторами и для обработки исключений типа `XmlException` и `ConfException`, ни один из них так и не будет вызван. Дело в том, что оба специальных исключения принадлежат более общему типу `Exception`, и поэтому они будут вполне соответствовать условию перехвата в первом блоке `catch`.

Первый блок `catch (FileException)` вызывается в том случае, если возникают проблемы при обращении к файлу конфигурации (он не существует или в него запрещена запись). Второй блок `catch (XmlException)` вызывается в том случае, если происходит ошибка при синтаксическом анализе XML-файла (например, не закрыт какой-нибудь элемент разметки). Третий блок `catch (ConfException)` вызывается в том случае, если XML-файл верного формата не содержит ожидаемого корневого элемента разметки `conf`. Наконец, последний блок `catch (Exception)` вообще не должен вызываться, потому что рассматриваемые здесь методы генерируют только три исключения, которые обрабатываются явным образом. В общем случае неплохо иметь такой блок-ловушку на тот случай, если в процессе разработки понадобится ввести новые исключения в прикладной код.

На заметку Если опустить блок-ловушку, то необходимо гарантировать действия при возникновении большинства исключений — незаметные сбои без уведомлений могут стать причиной ошибок, которые с трудом поддаются диагностике.

Преимущество этих блоков `catch` с подробно составленными условиями перехвата исключений состоит в том, что они позволяют применять к разным ошибкам различные механизмы восстановления или неудачного завершения. Например, можно прекратить выполнение программы, записать в журнал регистрации сведения об ошибке и продолжить выполнение или просто повторно сгенерировать то же самое исключение.

Еще один вариант, которым можно воспользоваться, — сгенерировать новое исключение, которое будет перекрывать текущее. Это позволяет привлечь внимание к ошибке, добавить собственную контекстную информацию и в то же время сохранить данные, зафиксированные в исключении, которое обработала программа. Дополнительную информацию об этой методике вы найдете в главе 15, “Стандарты PHP”.

Но что произойдет, если исключение не будет обработано в клиентском коде? Оно будет автоматически передано вызывающему коду, который сможет его обработать. Этот процесс будет продолжаться до тех пор, пока исключение не будет обработано или его уже нельзя будет никуда передать, — тогда произойдет неустранимая ошибка. Вот что произойдет, если в рассматриваемом здесь примере кода не будет обработано ни одно исключение:

```
PHP Fatal error: Uncaught exception 'FileException' with message
'file 'nonexistent/not_there.xml' does not exist' in ...
```

Таким образом, генерируя исключение, вы вынуждаете клиентский код брать на себя ответственность за его обработку. Но это не отказ от ответственности. Исключение должно генерироваться, когда метод обнаруживает ошибку, но не имеет контекстной информации, чтобы правильно ее обработать. В данном примере методу `write()` известно, когда и по какой причине попытка сделать запись завершается неудачно и почему, но в то же время неизвестно, что с этим делать. Именно так и должно быть. Если бы мы сделали класс `Conf` более сведущим в отношении ошибок, чем он есть в настоящий момент, он потерял бы свое конкретное назначение и стал бы менее пригодным для повторного использования.

Очистка после блоков `try/catch` с помощью оператора `finally`

Неожиданные осложнения могут возникнуть из-за того, что в ход выполнения программы могут вмешаться внешние факторы, проявляющиеся в виде исключений. Так, после возникновения исключения в блоке `try` может не выполниться код очистки или любые другие важные действия. Как пояснялось выше, при возникновении исключительной ситуации в блоке `try` управление передается первому подходящему блоку `catch`. В итоге может не выполниться код, в котором закрываются соединение с базой данных или файл, обновляется текущая информация о состоянии и т.п.

Допустим, в методе `Runner::init()` регистрируются все выполняемые действия. При этом в системном журнале регистрируется момент начала процесса инициализации, записываются все ошибки, возникающие при работе приложения, а в самом конце — момент окончания процесса инициализации. Ниже приведен типичный упрощенный фрагмент кода, выполняющего эти действия:

```
// Листинг 4.74
public static function init(): void
{
    try
    {
        $fh = fopen("/tmp/log.txt", "a");
        fputs($fh, "Начало\n");
        $conf = new Conf(dirname(FILE) . "/conf.broken.xml");
        print "user: " . $conf->get('user') . "\n";
        print "host: " . $conf->get('host') . "\n";
        $conf->set("pass", "newpass");
        $conf->write();
        fputs($fh, "Конец\n");
        fclose($fh);
    }
    catch (FileException $e)
    {
        // Файл не существует или недоступен
        fputs($fh, "Проблема с файлом\n");
        throw $e;
    }
    catch (XmlException $e)
    {
        fputs($fh, "Проблема с xml\n");
        // Поврежденный XML-файл
    }
    catch (ConfException $e)
    {
        fputs($fh, "Проблема конфигурации\n");
        // Неверный формат XML-файла
    }
    catch (\Exception $e)
    {
        fputs($fh, "Непредвиденные проблемы\n");
        // Ловушка: этот код не должен вызываться
    }
}
```

В этом примере кода сначала открывается файл регистрации `log.txt`, затем в него записываются данные, а далее вызывается код конфигурирования. Если на данном этапе возникнет ошибка, сведения об этом записываются в файл регистрации в блоке `catch`. Блок `try` завершается записью в файл регистрации и закрытием этого файла. Разумеется, если возникнет исключительная ситуация, эти действия выполнены не будут, поскольку управление будет передано соответствующему блоку `catch`, а следовательно, не будет выполнена и оставшаяся часть кода в блоке `try`. Ниже приве-

ден результат вывода в файл регистрации (системный журнал) при исключительной ситуации, возникающей при обращении к файлу:

```
Начало
Проблема с файлом
```

Как видите, в системном журнале зафиксированы момент начала процесса инициализации приложения, а также исключение, возникающее при обращении к файлу. А поскольку фрагмент кода, в котором сведения об окончании процесса инициализации выводятся в системный журнал, не выполнен, то и в файл регистрации ничего не записано.

Казалось бы, код последней стадии записи в системный журнал следует вынести за пределы блока `try/catch`, но такое решение нельзя назвать надежным. Дело в том, что при обработке исключительной ситуации в блоке `catch` может быть принято решение о возобновлении выполнения программы. Следовательно, управление может быть передано в произвольное место программы, которое находится далеко за пределами блока `try/catch`. Кроме того, в блоке `catch` может быть сгенерировано повторное исключение или полностью завершено выполнение сценария.

Чтобы помочь программистам справиться с описанной выше ситуацией, в версии PHP 5.5 была введена новая инструкция `finally`. Если вы знакомы с языком Java, то, скорее всего, уже сталкивались с ней. Несмотря на то что блок `catch` вызывается только при возникновении исключительной ситуации определенного типа, блок `finally` вызывается *всегда*, независимо от того, возникло исключение при выполнении блока `try` или нет.

Таким образом, для разрешения описанного выше затруднения достаточно переместить код последней стадии записи в системный журнал и закрытия файла в блок `finally`:

```
// Листинг 4.75
public static function init(): void
{
    try
    {
        $fh = fopen("/tmp/log.txt", "a");
        fputs($fh, "Начало\n");
        $conf = new Conf(dirname(FILE) . "/conf.broken.xml");
        print "user: " . $conf->get('user') . "\n";
        print "host: " . $conf->get('host') . "\n";
        $conf->set("pass", "newpass");
        $conf->write();
    }
```

```
    catch (FileException $e)
    {
        // Файл не существует или недоступен
        fputs($fh, "Проблема с файлом\n");
        throw $e;
    }
    catch (XmlException $e)
    {
        fputs($fh, "Проблема с xml\n");
        // Поврежденный XML-файл
    }
    catch (ConfException $e)
    {
        fputs($fh, "Проблема конфигурации\n");
        // Неверный формат XML-файла
    }
    catch (\Exception $e)
    {
        fputs($fh, "Непредвиденные проблемы\n");
        // Ловушка: этот код не должен вызываться
    }
    finally
    {
        fputs($fh, "Конец\n");
        fclose($fh);
    }
}
```

Последняя запись в системный журнал и вызов функции `fclose()` помещены в блок `finally`, и поэтому они будут выполняться всегда, даже при возникновении исключения типа `FileException` и повторной его генерации в блоке `catch`. Ниже приведен вывод в системный журнал при возникновении исключения типа `FileException`:

```
Начало
Проблема с файлом
Конец
```

На заметку Блок `finally` выполняется, если в блоке `catch` повторно генерируется исключение или выполняется оператор `return`, возвращающий значение вызывающему коду. Если же в блоке `try` или `catch` для завершения сценария вызывается функция `die()` или `exit()`, то блок `finally` не выполняется.

Завершенные классы и методы

Наследование открывает большие возможности для широкого поля действий в пределах иерархии класса. Класс или метод можно переопределить таким образом, чтобы вызов в клиентском методе приводил к совершенно разным результатам в зависимости от типа объекта, переданного этому методу в качестве аргумента. Но иногда код класса или метода нужно зафиксировать, если предполагается, что в дальнейшем он не должен изменяться. Если вы создали необходимый уровень функциональности для класса и считаете, что его переопределение может только повредить идеальной работе программы, воспользуйтесь ключевым словом `final`.

Ключевое слово `final` позволяет положить конец наследованию. Для завершенного класса нельзя создать подкласс, а завершенный метод нельзя перекрыть. Ниже показано, каким образом объявляется завершенный класс:

```
// Листинг 4.76
final class Checkout
{
    // ...
}
```

А вот как выглядит попытка создать подкласс, производный от класса `Checkout`:

```
// Листинг 4.77
class IllegalCheckout extends Checkout
{
    // ...
}
```

Это приводит к следующему сообщению об ошибке:

```
Fatal error: Class popp\ch04\batch13\IllegalCheckout may not inherit
from final class (popp\ch04\batch13\Checkout) in ...
```

Эту ситуацию можно немного облегчить, объявив завершенным не весь класс, а только метод в классе `Checkout`, как показано ниже. В объявлении завершенного метода ключевое слово `final` должно быть указано перед любыми другими модификаторами доступа наподобие `protected` или `static`:

```
// Листинг 4.78
class Checkout
{
    final public function totalize(): void
    {
        // Расчет расходов
    }
}
```

Теперь можно создать подкласс, производный от класса `Checkout`:

```
// Листинг 4.79
class IllegalCheckout extends Checkout
{
    final public function totalize(): void
    {
        // Измененный код расчета
    }
}
```

Но любая попытка переопределить метод `totalize()` приведет к неустранимой ошибке:

```
Fatal error: Cannot override final method popp\ch04\batch14\
Checkout::totalize() in
/var/popp/src/ch04/batch14/IllegalCheckout.php on line 9
```

В удачно написанном объектно-ориентированном коде обычно во главу угла ставится вполне определенный интерфейс. Но за этим интерфейсом могут скрываться разные реализации. Разные классы или их сочетания могут соответствовать общим интерфейсам, но при этом по-разному вести себя в различных ситуациях. Объявляя класс или метод завершенным, вы тем самым ограничиваете подобную гибкость. Иногда это желательно, и мы рассмотрим такие случаи далее в книге. Но прежде чем объявлять что-либо завершенным, следует серьезно подумать, действительно ли не существует ситуаций, в которых переопределение было бы полезным? Конечно, всегда можно передумать, но внести изменения впоследствии, возможно, будет нелегко (например, в том случае, если это распространяемая библиотека для совместного использования). Поэтому будьте внимательны, употребляя ключевое слово `final`.

Внутренний класс `Error`

Когда исключения только внедрялись, механизмы `try-catch` применялись главным образом в коде сценариев, но не в самом ядре PHP, где возникавшие ошибки обрабатывались собственной логикой. Это могло привести к серьезным осложнениям, если требовалось обрабатывать ошибки, возникающие в ядре PHP, таким же образом, как и исключения в прикладном коде. В качестве первоначальной меры борьбы с подобными осложнениями в версии PHP 7 был создан внутренний класс `Error`, в котором реализован тот же встроенный интерфейс `Throwable`, что и в классе `Exception`. Поэтому с классом `Error` можно обращаться так же, как и с классом `Exception`. Это означает, что в классе `Error` поддерживаются методы, перечисленные в табл. 4.1. Для обработки отдельных ошибок из класса `Error` можно создать соответствующий подкласс. Ниже показано, как можно перехватить ошибку синтаксического анализа, возникающую при выполнении оператора `eval`:

```
// Листинг 4.80
try
{
    eval("Некорректный код");
}
catch (\Error $e)
{
    print get_class($e) . "\n";
    print $e->getMessage();
}
catch (\Exception $e)
{
    // Обработка исключения Exception
}
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
ParseError
syntax error, unexpected identifier "Некорректный"
```

Таким образом, в блоке `catch` можно перехватывать некоторые типы внутренних ошибок, указав суперкласс `Error` или более конкретный его подкласс. Доступные в настоящее время подклассы, производные от внутреннего класса `Error`, перечислены в табл. 4.2.

Таблица 4.2. Подклассы, производные от внутреннего класса `Error`, появившегося в PHP 7

Тип ошибки	Описание
`ArithmeticError`	Генерируется при возникновении ошибок в математических операциях, особенно — в поразрядных арифметических операциях
`AssertionError`	Генерируется при неудачном выполнении языковой конструкции `assert()`, применяемой при отладке прикладного кода
`DivisionByZeroError`	Генерируется при попытке деления числа на нуль
`ParseError`	Генерируется при неудачной попытке выполнить синтаксический анализ кода PHP, например, с помощью оператора `eval`
`TypeError`	Генерируется при передаче методу аргумента неверного типа, возврате из метода значения неверного типа или передаче неверного количества аргументов методу

Работа с методами-перехватчиками

В языке PHP предусмотрены встроенные методы-перехватчики, которые могут перехватывать сообщения, посланные неопределенным (т.е. несуществующим) методам или свойствам. Такой механизм называется также *перегрузкой (overloading)*, но поскольку этот термин в Java и C++ означает нечто совершенно иное, то будет лучше употреблять термин *перехват (interception)*.

В языке PHP поддерживаются несколько встроенных методов-перехватчиков ("магических" методов). Аналогично конструктору `__construct()`, вызов этих методов происходит неявно, при наличии соответствующих условий. Некоторые их этих методов перечислены в табл. 4.3.

На заметку Дополнительную информацию о методах-перехватчиках можно почерпнуть из руководства по PHP по адресу `http://www.php.net/manual/en/language.oop5.magic.php`.

Таблица 4.3. Методы-перехватчики

Метод	Описание
`__get($property)`	Вызывается при обращении к неопределенному свойству
`__set($property, $value)`	Вызывается, когда присваивается значение неопределенному свойству
`__isset($property)`	Вызывается, когда функция `isset()` вызывается для неопределенного свойства
`__unset($property)`	Вызывается, когда функция `unset()` вызывается для неопределенного свойства
`__call($method, $arg_array)`	Вызывается при обращении к неопределенному нестатическому методу
`__callStatic($method, $arg_array)`	Вызывается при обращении к неопределенному статическому методу

Методы `__get()` и `__set()` предназначены для работы со свойствами, которые не были объявлены в классе (и в родительских классах). Метод `__get()` вызывается, когда клиентский код пытается прочитать необъявленное свойство. Он вызывается автоматически с одним строковым аргументом, содержащим имя свойства, к которому клиентский код пытается получить доступ. Все, что возвратит метод `__get()`, будет отправлено обратно клиенту, как если бы целевое свойство существовало с этим значением. Ниже приведен короткий пример:

```
// Листинг 4.81
class Person
{
    public function __get(string $property): mixed
    {
        $method = "get{$property}";

        if (method_exists($this, $method))
        {
            return $this->$method();
        }
    }
    public function getName(): string
    {
        return "Иван";
    }
```

```
    public function getAge(): int
    {
        return 44;
    }
}
```

Когда клиентский код пытается получить доступ к неопределенному свойству, вызывается метод `__get()`. Я реализовал его таким образом, чтобы он получал имя переданного ему свойства и добавлял к нему впереди строку `"get"`. Затем полученная строка, содержащая новое имя метода, передается функции `method_exists()`. Этой функции передается также ссылка на текущий объект, для которого проверяется существование метода. Если метод существует, он вызывается, а клиентскому коду передается значение, возвращаемое из этого метода. Следовательно, если в клиентском коде запрашивается свойство `$name`:

```
// Листинг 4.82
$p = new Person();
print $p->name;
```

то метод `getName()` вызывается неявно и выводится строка

```
Иван
```

Если же метода не существует, то не происходит ничего. Свойству, к которому пользователь пытается обратиться, присваивается значение `null`.

Метод `__isset()` действует аналогично методу `__get()`. Он вызывается после того, как в клиентском коде вызывается функция `isset()` для неопределенного свойства. В качестве примера ниже показано, как можно расширить класс `Person`:

```
// Листинг 4.83
public function __isset(string $property): bool
{
    $method = "get{$property}";
    return (method_exists($this, $method));
}
```

Теперь предусмотрительный пользователь может проверить свойство, прежде чем работать с ним:

```
// Листинг 4.84
$p = new Person();
```

```
if (isset($p->name))
{
    print $p->name;
}
```

Метод `__set()` вызывается, когда в клиентском коде предпринимается попытка присвоить значение неопределенному свойству. При этом ему передаются два аргумента: имя свойства и значение, которое требуется присвоить. Вы можете решить, как следует обращаться с этими аргументами. Продолжим расширение класса `Person` следующим образом:

```
// Листинг 4.85

class Person
{
    private ? string $myname;
    private ? int $myage;
    public function __set(string $property, mixed $value): void
    {
        $method = "set{$property}";

        if (method_exists($this, $method))
        {
            $this->$method($value);
        }
    }
    public function setName( ? string $name): void
    {
        $this->myname = $name;

        if (! is_null($name))
        {
            $this->myname = strtoupper($this->myname);
        }
    }
    public function setAge( ? int $age): void
    {
        $this->myage = $age;
    }
}
```

В этом примере применяются методы доступа для установки значения, а не для его получения. Если пользователь попытается присвоить значение неопределенному свойству, то методу `__set()` будут переданы имя этого свойства и присваиваемое ему значение. В методе `__set()` проверяется, существует ли указанный метод, и, если существует, то он вызывается. В итоге можно отфильтровать значение, присваиваемое свойству.

На заметку Не забывайте, что при описании имен методов и свойств в документации PHP зачастую употребляется статический синтаксис, чтобы было ясно, в каком классе они содержатся. Поэтому вы можете встретить ссылку на свойство `Person::$name`, даже если оно не объявлено как статическое (`static`), и к нему на самом деле можно получить доступ через объект, имя которого отличается от `Person`.

Таким образом, если мы создаем объект типа `Person`, а затем пытаемся установить свойство `Person::$name`, то вызывается метод `__set()`, потому что в классе `Person` свойство `$name` не определено. Этому методу передаются две строки, содержащие имя свойства и значение, которое требуется установить. Как именно это значение используется далее, зависит от конкретной реализации метода `__set()`. В данном примере новое имя метода образуется путем добавления строки `"set"` перед именем свойства, передаваемого в качестве аргумента. В итоге интерпретатор PHP обнаруживает метод `setName()` и вызывает его должным образом. В этом методе входное строковое значение приводится к верхнему регистру букв и сохраняется в действительно существующем свойстве, как показано ниже:

```
// Листинг 4.86
$p = new Person();
$p->name = "Иван";
// Свойству $myname присваивается строка "Иван"
```

Как и следовало ожидать из названия, метод `__unset()` является зеркальным отражением метода `__set()`. Он вызывается в том случае, если функции `unset()` передается имя неопределенного свойства. Имя этого свойства передается далее методу `__unset()`. Полученную информацию о свойстве можно обрабатывать как угодно. Так, в приведенном ниже примере кода пустое значение `null` передается найденному в итоге методу тем же самым способом, что и выше в методе `__set()`:

```
// Листинг 4.87
public function __unset(string $property): void
{
    $method = "set{$property}";

    if (method_exists($this, $method))
    {
        $this->$method(null);
    }
}
```

Метод `__call()`, вероятно, самый полезный из всех методов-перехватчиков. Он вызывается в том случае, если клиентский код обращается к неопределенному методу. При этом методу `__call()` передаются имя несуществующего метода и массив, в котором содержатся все аргументы, переданные клиентом. Значение, возвращаемое методом `__call()`, передается обратно клиенту так, как будто оно было возвращено вызванным несуществующим методом.

Метод `__call()` можно применять для целей делегирования. *Делегирование* — это механизм, с помощью которого один объект может вызвать метод другого объекта. Это чем-то напоминает наследование, когда дочерний класс вызывает метод, реализованный в родительском классе. Но при наследовании взаимосвязь между родительским и дочерним классами фиксирована. Поэтому возможность изменить объект-получатель во время выполнения программы означает, что делегирование является более гибким механизмом, чем наследование. Чтобы механизм делегирования стал понятнее, обратимся к конкретному примеру. Рассмотрим простой класс, предназначенный для форматирования информации, полученной из класса `Person`:

```
// Листинг 4.88
class PersonWriter
{
    public function writeName(Person $p): void
    {
        print $p->getName() . "\n";
    }
    public function writeAge(Person $p): void
    {
        print $p->getAge() . "\n";
    }
}
```

Безусловно, можно было бы создать подкласс, производный от этого класса, чтобы выводить различными способами сведения, получаемые из класса `Person`. В качестве примера одного из таких способов ниже приведена реализация класса `Person`, в которой используются объект типа `PersonWriter` и метод `__call()`:

```
// Листинг 4.89
class Person
{
    public function __construct(private PersonWriter $writer)
    {
    }
```

```
    public function __call(string $method, array $args): mixed
    {
        if (method_exists($this->writer, $method))
        {
            return $this->writer->$method($this);
        }
    }
    public function getName(): string
    {
        return "Bob";
    }
    public function getAge(): int
    {
        return 44;
    }
}
```

Здесь конструктору класса `Person` в качестве аргумента передается объект типа `PersonWriter`, ссылка на который сохраняется в переменной свойства `$writer`. В методе `__call()` используется значение аргумента `$method` и проверяется наличие метода с таким же именем в объекте `PersonWriter`, ссылка на который была сохранена в конструкторе. Если такой метод найден, его вызов делегируется объекту `PersonWriter`. При этом вызываемому методу передается ссылка на текущий экземпляр объекта типа `Person`, которая хранится в псевдопеременной `$this`. Так, если клиент вызовет несуществующий в классе `Person` метод следующим образом:

```
// Листинг 4.90
$person = new Person(new PersonWriter());
$person->writeName();
```

в итоге будет вызван метод `__call()`, в котором проверяется, существует ли в объекте типа `PersonWriter` метод с именем `writeName()`. Поскольку этот метод действительно существует, он вызывается. Это позволяет избежать вызова делегируемого метода вручную, как показано ниже:

```
// Листинг 4.91
public function writeName(): void
{
    $this->writer->writeName($this);
}
```

Используя методы-перехватчики, класс Person, как по волшебству, получил два новых метода из класса PersonWriter. Но хотя автоматическое делегирование избавляет от рутинной работы по стереотипному программированию вызовов методов, сам код становится сложным для понимания. И если в вашей программе активно используется делегирование, то для внешнего мира создается динамический интерфейс, который не поддается рефлексии (исследованию состава класса во время выполнения программы) и не всегда с первого взгляда понятен программисту клиентского кода. Дело в том, что логика, лежащая в основе взаимодействия делегирующего класса и целевого объекта, может быть непонятной. Ведь она скрыта в таких методах, как __call(), а не явно задана отношениями наследования или указаниями типов аргументов в методах. Методы-перехватчики находят свое применение, но пользоваться ими следует аккуратно. И в тех классах, в которых применяются эти методы, такое применение следует тщательно документировать.

К вопросам делегирования и рефлексии мы еще вернемся в этой книге.

Методы-перехватчики __get() и __set() можно также применять для поддержки составных свойств. Они могут создать определенные удобства для программиста, разрабатывающего клиентский код. Допустим, что в классе Address сохраняется информация о номере дома и названии улицы. В конечном итоге информация из этого класса будет сохранена в соответствующих полях базы данных, и поэтому такое разделение адреса на улицу и номер дома имеет определенный смысл. Но если часто требуется неразделенная информация об адресе, содержащая в одной строке номер дома и название улицы, то необходимо позаботиться и о пользователях данного класса. Ниже приведен пример класса, в котором поддерживается составное свойство Address::$streetaddress:

```
// Листинг 4.92
class Address
{
    private string $number;
    private string $street;
    public function __construct(string $maybenumber,
                                string $maybestreet = null)
    {
        if (is_null($maybestreet))
        {
            $this->streetaddress = $maybenumber;
        }
```

```
        else
        {
            $this->number = $maybenumber;
            $this->street = $maybestreet;
        }
    }
    public function __set(string $property, mixed $value): void
    {
        if ($property === "streetaddress")
        {
            if (preg_match("/^(\d+.*?)[\s,]+(.+)$/",
                           $value, $matches))
            {
                $this->number = $matches[1];
                $this->street = $matches[2];
            }
            else
            {
                throw new
                  \Exception("Ошибка анализа адреса: '{$value}'");
            }
        }
    }
    public function __get(string $property): mixed
    {
        if ($property === "streetaddress")
        {
            return $this->number . " " . $this->street;
        }
    }
}
```

При попытке установить значение несуществующего свойства `Address::$streetaddress` (через конструктор класса) будет вызван метод-перехватчик `__set()`. В нем проверяется имя составного свойства `"streetaddress"`, которое необходимо поддерживать в классе. Перед тем как установить значения свойств `$number` и `$street`, следует убедиться, что переданный адрес соответствует ожидаемому шаблону и может быть корректно проанализирован. После этого нужно извлечь значения свойств из строки адреса, которую передал пользователь. Для целей данного примера было установлено следующее простое правило: адрес корректен, если он начинается с цифры, после которой следует пробел или запятая, а затем — название улицы. Благодаря применению обратных ссылок в регулярном выражении после удачного исхода синтаксического анализа строки в мас-

сиве $matches окажутся нужные данные, которые можно присвоить свойствам $number и $street. Если же строка адреса не проходит проверку, генерируется исключение. Следовательно, когда строка наподобие "221б Бейкер-стрит" присваивается свойству Address::$streetaddress, на самом деле выделенные из этой строки значения будут присвоены свойствам $number и $street. В этом можно убедиться с помощью функции print_r(), как показано ниже:

```
// Листинг 4.93
$address = new Address("221б Бейкер-стрит");
print_r($address);
```

```
popp\ch04\batch16\Address Object
(
    [number:popp\ch04\batch16\Address:private] => 221б
    [street:popp\ch04\batch16\Address:private] => Бейкер-стрит
)
```

По сравнению с методом __set() метод __get() очень простой. При попытке доступа к несуществующему свойству Address::$streetaddress будет вызван метод __get(). В рассматриваемой здесь реализации этого метода сначала проверяется имя свойства "streetaddress", и, если оно совпадает, в вызывающую часть программы возвращается строка, составленная из двух значений свойств — $number и $street.

На заметку __get(), __set() и __call() также автоматически вызываются, когда клиент пытается получить доступ к недоступному методу или свойству (т.е. к методам или свойствам, которые объявлены как private или protected и потому скрыты от вызывающего контекста).

Определение методов-деструкторов

Как было показано ранее, при создании экземпляра объекта автоматически вызывается метод-конструктор __construct(). В версии PHP 5 в язык был добавлен метод-деструктор __destruct(), который вызывается непосредственно перед тем, как объект отправляется на “свалку”, т.е. удаляется из памяти. Этим методом можно пользоваться для выполнения завершающей очистки оперативной памяти от объектов, если в этом есть необходимость.

Представим, например, что по запросу экземпляр класса сохраняется в базе данных. В таком случае можно использовать метод __destruct(), чтобы гарантированно сохранить данные экземпляра этого класса перед его удалением из памяти. В качестве примера добавим деструктор в класс Person:

```
// Листинг 4.94
class Person
{
    private int $id;
    public function __construct(protected string $name,
                                private int $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
    public function setId(int $id): void
    {
        $this->id = $id;
    }
    public function __destruct()
    {
        if (! empty($this->id))
        {
            // Сохранение данных Person
            print "Сохранение Person\n";
        }
    }
}
```

Метод __destruct() вызывается всякий раз при вызове функции unset(), которой передается ссылка на объект, или когда в процессе не остается никаких ссылок на объект. Так что если я создам и уничтожу объект Person, то будет видно, как в игру вступит деструктор __destruct():

```
// Листинг 4.95
$person = new Person("Иван", 44);
$person->setId(343);
unset($person);
```

При этом будет выведено сообщение

```
Сохранение Person
```

Хотя такие приемы выглядят очень занимательно, следует все же высказать предостережение. Такие методы, как __call(), __destruct() и им

подобные, иногда еще называют *магическими*. Если вы когда-либо читали произведения в жанре фэнтези, то, вероятно, знаете, что магия — это не всегда благо. Магия случайна и непредсказуема, она нарушает правила и влечет скрытые расходы.

Например, в случае использования вами метода `__destruct()` может случиться так, что программист клиентского кода столкнется с неприятными сюрпризами. Задумайтесь, как работает класс `Person`: он делает запись в базу данных с помощью своего метода `__destruct()`. А теперь представьте, что начинающий разработчик решает воспользоваться вашим классом `Person`, не разобравшись в механизме его действия. Не обратив внимания на метод-деструктор `__destruct()`, он собирается создать ряд экземпляров класса `Person`. Передавая значения конструктору, он дает свойству `$name` обидную кличку генерального директора и устанавливает его свойство `$age` равным `150`. Далее разработчик прогоняет тестовый сценарий несколько раз, используя красочные сочетания имени и возраста своего начальника.

На следующее утро начальник вызовет его к себе в кабинет и потребует пояснений, почему в базе данных содержатся данные, оскорбительные для служащих компании. Мораль сей басни такова: не доверяйте магии.

Копирование объектов с помощью метода `__clone()`

В версии PHP 4 копирование объекта выполнялось очень просто — достаточно было значение одной объектной переменной присвоить другой, как показано ниже:

```
// Листинг 4.96
class CopyMe
{
}

// Листинг 4.97
$first = new CopyMe();
$second = $first;
// В версии PHP 4 переменные $second
// и $first — два разных объекта.
// Начиная с версии PHP 5 переменные $second
// и $first ссылаются на один и тот же объект
```

Но такая простота нередко служила источником многих ошибок, поскольку копии объекта случайно размножались при присвоении значений переменным, вызове методов и возврате объектов. Ситуацию еще больше ухудшало то обстоятельство, что не существовало способа проверить обе переменные, чтобы понять, относятся ли они к одному и тому же объекту. С помощью операторов равенства можно было проверить, одинаковы ли значения во всех полях двух сравниваемых объектов (оператор ==) либо являются ли обе переменные объектами (оператор ===). Но в результате этих проверок нельзя было понять, указывают ли обе переменные на один и тот же объект.

В PHP переменная, которая *кажется* содержащей объект, на самом деле содержит идентификатор, который ссылается на основную структуру данных. Когда такая переменная присваивается или передается в метод, копируется идентификатор, который она содержит. Но каждая копия при этом продолжает указывать на все тот же объект. Это означает, что в моем предыдущем примере `$first` и `$second` содержат идентификаторы, указывающие на один и тот же объект, а не на две копии объекта. Хотя в общем случае это именно то, что требуется, бывают случаи, когда нужно получить копию объекта.

В языке PHP объекты всегда присваиваются и передаются по ссылке. Это означает, что если выполнить приведенный выше пример кода в версии PHP 5, то переменные `$first` и `$second` будут содержать ссылки на один и тот же объект, а не на две разные его копии. И хотя это именно то, что, как правило, нужно для манипулирования объектами, возможны ситуации, когда требуется получить копию объекта, а не ссылку на объект.

В версии PHP 5 для этой цели предусмотрено ключевое слово `clone`. Оно применяется к экземпляру объекта и создает его дополнительную копию:

```
// Листинг 4.98
$first = new CopyMe();
$second = clone $first;
// Начиная с версии PHP 5 переменные $second
// и $first ссылаются на два разных объекта
```

И здесь только начинают возникать вопросы, касающиеся копирования объектов. Рассмотрим в качестве примера класс `Person`, созданный в предыдущем разделе. Стандартная копия объекта типа `Person` содержит идентификатор (свойство `$id`), который при реализации полноценного приложения будет использоваться для нахождения нужной строки в базе данных. Если разрешить копирование этого свойства, то программист кли-

ентского кода получит в конечном итоге два разных объекта, представляющих одну и ту же сущность данных (строку базы данных), которая может не быть той, которую он ожидал, создавая копию.

К счастью, используя ключевое слово clone для копирования объектов, можно проконтролировать, что именно копируется. Для этого следует реализовать специальный метод __clone() (обратите еще раз внимание на два знака подчеркивания, с которых начинаются имена всех "магических" методов). Метод __clone() вызывается автоматически, когда для копирования объекта используется ключевое слово clone.

При реализации метода __clone() важно понимать контекст, в котором действует данный метод. Метод __clone() вызывается для *копируемого* объекта, а не для исходного. Добавим метод __clone() в очередную версию класса Person, как показано ниже:

```
// Листинг 4.99
class Person
{
    private int $id = 0;
    public function __construct(private string $name, private $ageё)
    {
    }
    public function setId(int $id): void
    {
        $this->id = $id;
    }
    public function __clone(): void
    {
        $this->id = 0;
    }
}
```

Когда операция clone выполняется для объекта типа Person, создается его новая неполная или поверхностная (shallow) копия и вызывается *ее* метод __clone(). Это означает, что все изменения значений свойств, выполняемые в методе __clone(), отразятся только на новой копии объекта. А прежние значения, полученные из исходного объекта, будут затерты. В данном случае гарантируется, что свойство $id скопированного объекта устанавливается равным нулю:

```
// Листинг 4.100
$person = new Person("Иван", 44);
$person->setId(343);
$person2 = clone $person;
```

Поверхностное копирование гарантирует, что значения элементарных свойств будут скопированы из старого объекта в новый. Для свойств объектов выполняется копирование идентификаторов, но не лежащих в их основе объектов — возможно, это не совсем то, что нужно, когда клонируется объект. Допустим, что объект `Person` дополнен свойством, ссылающимся на объект типа `Account`, как показано ниже. В этом объекте хранятся данные о состоянии счета, которые также требуется скопировать в клонированный объект, но в то же время нам не нужно сохранять в обеих копиях объекта типа `Person` ссылки на *один и тот же* счет:

```
// Листинг 4.101
class Account
{
    public function __construct(public float $balance)
    {
    }
}

// Листинг 4.102
class Person
{
    private int $id;
    public function __construct(private string $name,
                                private int $age,
                                public Account $account)
    {
    }
    public function setId(int $id): void
    {
        $this->id = $id;
    }
    public function __clone(): void
    {
        $this->id = 0;
    }
}

// Листинг 4.103
$person = new Person("Иван", 44, new Account(200));
$person->setId(343);
$person2 = clone $person;
// Добавим $person немного денег
$person->account->balance += 10;
// Это отразится и на $person2
print $person2->account->balance;
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
210
```

В объект `$person` было добавлено специальное свойство `$account`, содержащее ссылку на объект типа `Account`. Это свойство мы намеренно сделали открытым ради краткости данного примера. Как вам, должно быть, уже известно, доступ к свойству обычно ограничивается, а по мере надобности создается метод доступа к нему. Когда же создается клон, он содержит ссылку на тот же самый объект типа `Account`, что и в объектной переменной `$person`. Мы наглядно продемонстрировали это в данном примере, добавляя сначала немного денег к балансу объекта `Account` переменной `$person`, а затем выводя его значение через переменную `$person2`.

Если нежелательно, чтобы после выполнения операции клонирования в новом объекте осталась ссылка на старый объект, это свойство следует клонировать явным образом в методе `__clone()`, как показано ниже:

```
// Листинг 4.104
public function __clone(): void
{
    $this->id = 0;
    $this->account = clone $this->account;
}
```

Определение строковых значений для объектов

Еще одним языковым средством, внедренным в версии PHP 5 явно под влиянием Java, является метод `__toString()`. До выхода версии PHP 5.2 при выводе объекта он разрешался в строку:

```
// Листинг 4.105
class StringThing
{
}

// Листинг 4.106
$st = new StringThing();
print $st;
```

Начиная с версии PHP 5.2 выполнение этого фрагмента кода стало приводить к выводу следующей ошибки:

```
Object of class popp\ch04\batch22\StringThing could
not be converted to string ...
```

Реализуя метод `__toString()`, можно управлять тем, как именно объекты будут представлены при обращении в строковом контексте (или при явном приведении к строке). Метод `__toString()` должен возвращать строковое значение. Этот метод вызывается автоматически, когда объект передается функции `print()` или `echo()`, а возвращаемое им строковое значение будет выведено на экран. В качестве примера добавим свою версию метода `__toString()` в минимальную реализацию класса `Person`:

```
// Листинг 4.107
class Person
{
    public function getName(): string
    {
        return "Иван";
    }
    public function getAge(): int
    {
        return 44;
    }
    public function __toString(): string
    {
        $desc  = $this->getName() . " (возраст ";
        $desc .= $this->getAge() . " лет)";
        return $desc;
    }
}
```

Если теперь вывести объект типа `Person` на экран следующим образом:

```
// Листинг 4.108
$person = new Person();
print $person;
Bob(age 44)
```

то получится следующий результат:

```
Иван (возраст 44 лет)
```

Метод `__toString()` особенно удобен для записи информации в файлы регистрации и выдачи сообщений об ошибках, а также для примене-

ния в тех классах, основная задача которых — передавать информацию. Например, при выводе на экран объекта класса `Exception` можно выдать краткий отчет о возникших исключениях, реализовав в этом классе метод `__toString()`.

Начиная с PHP 8 любой класс, который реализует метод `__toString()`, неявно объявлен как реализующий встроенный интерфейс `Stringable`. Это означает, что вы можете использовать тип объединения для ограничения аргументов и свойств. Вот пример:

```
// Листинг 4.109
public static function printThing(string | \Stringable $str): void
{
    print $str;
}
```

Мы можем передать строку *или* объект `Person` методу `printThing()`, и он отлично сработает в обоих случаях.

Функции обратного вызова, анонимные функции и замыкания

Несмотря на то что анонимные функции не относятся, строго говоря, к средствам объектно-ориентированного программирования, они настолько полезны, что не упомянуть их здесь нельзя. Они активно используются в объектно-ориентированных приложениях наряду с функциями обратного вызова.

На заметку Функция обратного вызова (callback) — это блок выполнимого кода, который можно сохранить в переменной или передать методам и функциям для последующего вызова.

Определим сначала несколько классов следующим образом:

```
// Листинг 4.110
class Product
{
    public function __construct(public string $name,
                                public float $price)
    {
    }
}
```

```
// Листинг 4.111
class ProcessSale
{
    private array $callbacks;
    public function registerCallback(callable $callback): void
    {
        $this->callbacks[] = $callback;
    }
    public function sale(Product $product): void
    {
        print "{$product->name}: обрабатывается \n";

        foreach ($this->callbacks as $callback)
        {
            call_user_func($callback, $product);
        }
    }
}
```

Этот код предназначен для выполнения разнообразных функций обратного вызова. В нем определены два класса: `Product` и `ProcessSale`. В классе `Product` просто сохраняются значения свойств `$name` и `$price`. Ради краткости примера они объявлены открытыми. Не следует, однако, забывать, что в реальном проекте следует сделать такие свойства закрытыми или защищенными и создать для них методы доступа.

В классе `ProcessSale` определены два метода. Первый, `register Callback()`, принимает тип `callable` и добавляет его в свойство-массив `$callbacks`. Второй метод, `sale()`, принимает объект `Product`, выводит о нем сообщение, а затем циклически обходит свойство-массив `$callbacks`.

Каждый элемент этого массива вместе с объектом типа `Product` передается функции `call_user_func()`, которая, собственно, и вызывает код, написанный пользователем. Все приведенные ниже примеры будут следовать этому образцу.

Чем же так полезны функции обратного вызова? Они позволяют во время выполнения вводить в компонент новые функциональные возможности, которые первоначально не были непосредственно связаны с основной задачей, решаемой этим компонентом. Предусмотрев в компоненте поддержку функций обратного вызова, вы тем самым позволяете другим программистам расширить функциональные возможности вашего кода, причем в таких контекстах, о которых вы даже не подозревали.

Допустим, например, что через какое-то время пользователю класса ProcessSale потребуется создать журнал продаж. Если этому пользователю будет доступен исходный код данного класса, он сможет ввести код регистрации продаж непосредственно в метод sale(). Но такое решение не всегда оказывается удачным. Если этот пользователь не является владельцем пакета, в котором определен класс ProcessSale, то все его исправления будут затерты, когда выйдет обновленная версия пакета. И даже если он является владельцем пакета, то все равно вводить в метод sale() дополнительные фрагменты кода, решающие случайные задачи, неразумно. В конечном итоге это приведет к чрезмерному расширению обязанностей данного метода и способно уменьшить возможность его повторного использования в других проектах. Мы еще вернемся к этой теме в других главах.

К счастью, в классе ProcessSale из рассматриваемого здесь примера предусмотрена возможность обратного вызова. Ниже приведена функция обратного вызова, имитирующая регистрацию товара в журнале продаж:

```
// Листинг 4.112
$logger = function($product)
{
    print "    Запись ({$product->name})\n";
};
$processor = new ProcessSale();
$processor->registerCallback($logger);
$processor->sale(new Product("Туфли", 6));
print "\n";
$processor->sale(new Product("Кофе", 6));
```

Здесь я создаю анонимную функцию, т.е. использую ключевое слово function без имени функции. Обратите внимание, что, поскольку это встраиваемая инструкция, в конце блока кода требуется точка с запятой. Анонимная функция может быть сохранена в переменной и передана функциям и методам в качестве параметра. Это именно то, что я делаю, присваивая функцию переменной $logger и передавая ее методу ProcessSale::registerCallback(). Наконец, я создаю пару товаров и передаю их методу sale(). Затем происходит обработка продажи (на самом деле — просто выводится сообщение о товаре) и выполняются все обратные вызовы. Вот описанный код в действии:

```
Туфли: обрабатывается
    Запись(Туфли)
Кофе: обрабатывается
    Запись(Кофе)
```

PHP 7.4 представляет новый способ объявления анонимных функций. Функции со стрелками функционально очень схожи с анонимными функциями, с которыми вы уже столкнулись, но их синтаксис гораздо более компактен. Вместо ключевого слова `function` они определяются с помощью ключевого слова `fn`, за которым идут круглые скобки с аргументами, оператор стрелки (=>) и единственное выражение. Такая компактная форма делает функции со стрелками очень удобными для создания небольших функций обратного вызова для пользовательской сортировки и других подобных задач. Далее вместо анонимной функции я использую полностью эквивалентную функцию со стрелкой:

```
// Листинг 4.113
$logger = fn($product) => print "    Запись ({$product->name})\n";
```

Функция со стрелкой гораздо более компактна, но поскольку она состоит только из одного выражения, ее лучше использовать только для очень простых задач.

Безусловно, функции обратного вызова не обязательно должны быть анонимными. В качестве такой функции смело можно указать имя обычной функции или даже ссылку на метод какого-нибудь объекта. Ниже приведен характерный пример:

```
// Листинг 4.114
class Mailer
{
    public function doMail(Product $product): void
    {
        print "    Отправляется ({$product->name})\n";
    }
}

// Листинг 4.115
$processor = new ProcessSale();
$processor->registerCallback([new Mailer(), "doMail"]);
$processor->sale(new Product("shoes", 6));
print "\n";
$processor->sale(new Product("coffee", 6));
```

В данном примере создается новый класс `Mailer`, содержащий единственный метод `doMail()`. Этому методу передается объект типа `Product`, о котором он выводит сообщение. При вызове метода `registerCallback()` в качестве параметра ему передается массив, а не ссылка на функцию обратного вызова, как это было раньше. Первым элементом этого массива явля-

ется объект типа `Mailer`, а вторым — символьная строка, содержащая имя метода, который требуется вызвать.

Напомним, что метод `registerCallback()` использует объявление типа для того, чтобы аргумент имел тип `callable`. PHP достаточно развит, чтобы распознать массив такого рода как вызываемый. Поэтому при правильной организации обратного вызова в первом элементе такого массива должен находиться объект, содержащий вызываемый метод, а имя этого метода указывается в виде символьной строки во втором элементе массива. При удачном исходе проверки типа аргумента будет выведен следующий результат:

```
Туфли: обрабатывается
    Отправляется (Туфли)

Кофе: обрабатывается
    Отправляется (Кофе)
```

Имеется также возможность возвратить анонимную функцию из метода, как показано ниже:

```
// Листинг 4.116
class Totalizer
{
    public static function warnAmount(): callable
    {
        return function(Product $product)
        {
            if ($product->price > 5)
            {
              print "Достигнута высокая цена: {$product->price}\n";
            }
        };
    }
}

// Листинг 4.117
$processor = new ProcessSale();
$processor->registerCallback(Totalizer::warnAmount());
$processor->sale(new Product("Туфли", 6));
print "\n";
$processor->sale(new Product("Кофе", 6));
```

В данном примере нет ничего интересного, кроме того, что метод `warnAmount()` используется в качестве фабрики анонимной функции (т.е. в нем создается анонимная функция). Тем не менее подобная струк-

тура позволяет сделать нечто гораздо большее, чем просто сгенерировать анонимную функцию. Она позволяет выгодно воспользоваться преимуществами *замыканий*. Анонимная функция может обращаться к переменным, объявленным в другой анонимной функции в родительской области видимости. Это довольно сложный принцип, чтобы понять его сразу.

Вкратце его можно объяснить так, как будто анонимная функция запоминает контекст, в котором она была создана. Предположим, нам нужно сделать так, чтобы метод `Totalizer::warnAmount()` выполнял следующее. Во-первых, мы хотим, чтобы методу можно было передавать пороговое значение стоимости проданных товаров. Во-вторых, нам нужно, чтобы он подсчитывал стоимость (т.е. сумму цен) проданных товаров. И когда стоимость проданных товаров превысит установленный порог, функция должна выполнить некоторые действия (в нашем случае, как вы уже можете догадаться, она просто выведет соответствующее сообщение).

Чтобы в анонимной функции можно было отслеживать переменные, определенные в более обширной (родительской) области видимости, используется ключевое слово `use`, как показано в приведенном ниже примере:

```
// Листинг 4.118
class Totalizer2
{
    public static function warnAmount($amt): callable
    {
        $count = 0;
        return function($product) use($amt, &$count)
        {
            $count += $product->price;
            print " Сумма: $count\n";

            if ($count > $amt)
            {
                print "Достигнута сумма: {$count}\n";
            }
        };
    }
}

// Листинг 4.119
$processor = new ProcessSale();
$processor->registerCallback(Totalizer2::warnAmount(8));
$processor->sale(new Product("Туфли", 6));
print "\n";
$processor->sale(new Product("Кофе", 6));
```

В директиве use анонимной функции, которая возвращается методом Totalizer::warnAmount(), используются две переменные. Первой из них является переменная $amt, которая передается в качестве аргумента методу warnAmount(), а вторая — переменная замыкания $count. Она объявлена в теле метода warnAmount(), и начальное ее значение равно нулю. Обратите внимание на то, что перед именем переменной $count в директиве use указан символ амперсанда '&'. Это означает, что данная переменная будет передаваться анонимной функции по ссылке, а не по значению. Дело в том, что в теле анонимной функции к ее значению прибавляется цена товара, а затем новая сумма сравнивается со значением переменной $amt. Если достигнуто запланированное значение, то выводится соответствующее сообщение, как показано ниже:

```
Туфли: обрабатывается
сумма: 6

Кофе: обрабатывается
сумма: 12

Достигнута сумма: 12
```

В данном примере показано, что значение переменной $count сохраняется между вызовами функции обратного вызова. Обе переменные, $count и $amt, остаются связанными с этой функцией, поскольку они указаны в контексте ее объявления и указаны в директиве use.

Функции со стрелками также генерируют замыкания (как и анонимные функции, они разрешаются к экземпляру встроенного класса Closure). В отличие от анонимных функций, которые требуют явной связи с переменными замыкания, они автоматически получают копии всех переменных в области видимости по значению:

```
// Листинг 4.120
$markup = 3;
$counter = fn(Product $product) => print
           "($product->name) отмечена цена: " .
              ($product->price + $markup) . "\n";
$processor = new ProcessSale();
$processor->registerCallback($counter);
$processor->sale(new Product("Туфли", 6));
print "\n";
$processor->sale(new Product("Кофе", 6));
```

PHP 7.1 предоставляет новый способ управления замыканиями в контексте объекта. Метод `Closure::fromCallable()` позволяет создавать замыкание, которое предоставляет вызываемому коду доступ к классам и свойствам объекта. Вот версия `Totalizer`, использующая свойства объекта для достижения того же результата, что и в последнем примере:

```
// Листинг 4.121
class Totalizer3
{
    private float $count = 0;
    private float $amt = 0;
    public function warnAmount(int $amt): callable
    {
        $this->amt = $amt;
        return \Closure::fromCallable([$this, "processPrice"]);
    }
    private function processPrice(Product $product): void
    {
        $this->count += $product->price;
        print " Сумма: {$this->count}\n";

        if ($this->count > $this->amt)
        {
            print "Достигнута сумма: {$this->count}\n";
        }
    }
}
```

Метод `warnAmount()` здесь не является статическим. Вот потому, благодаря `Closure::fromCallable()`, я возвращаю функцию обратного вызова методу `processPrice()`, который имеет доступ к более широкому объекту. Я устанавливаю свойство `$amt` и возвращаю ссылку на метод обратного вызова. При вызове `processPrice()` увеличивает свойство `$count` и выдает предупреждение при достижении значения свойства `$amt`. Если бы `processPrice()` был общедоступным методом, я мог бы просто вернуть `[$this,"processPrice"]`. Как мы видели, PHP достаточно умен, чтобы вывести, что такой двухэлементный массив должен разрешаться в обратный вызов. Но есть две веские причины, по которым я мог бы захотеть использовать `Closure::fromCallable()`. Во-первых, я могу дать контролируемый доступ к частным или защищенным методам без необходимости раскрывать их всему миру, получая, таким образом, расширенную функциональность при управлении доступом. Во-вторых, я получаю повышение производительности, потому что имеются дополни-

тельные накладные расходы при выводе, является ли возвращаемое значение истинно обратно вызываемым.

Далее я использую `Totalizer3` с неизмененным классом `ProcessSale`:

```
// Листинг 4.122
$totalizer3 = new Totalizer3();
$processor = new ProcessSale();
$processor->registerCallback($totalizer3->warnAmount(8));

$processor->sale(new Product("Туфли", 6));
print "\n";
$processor->sale(new Product("Кофе", 6));
```

Анонимные классы

Начиная с версии PHP 7 можно объявлять анонимные классы. Анонимные классы очень удобны в тех случаях, когда требуется создать экземпляр небольшого класса, если класс является простым и характерным для локального контекста.

Вернемся к примеру класса `PersonWriter`, начав на этот раз с создания интерфейса следующим образом:

```
// Листинг 4.123
interface PersonWriter
{
    public function write(Person $person): void;
}
```

Ниже приведена новая версия класса `Person`, в котором можно воспользоваться объектом типа `PersonWriter`:

```
// Листинг 4.124
class Person
{
    public function output(PersonWriter $writer): void
    {
        $writer->write($this);
    }
    public function getName(): string
    {
        return "Иван";
    }
    public function getAge(): int
    {
```

```
        return 44;
    }
}
```

В данном примере методу `output()` в качестве аргумента передается экземпляр объекта типа `PersonWriter`, методу `write()` которого передается экземпляр текущего класса. Благодаря такому решению класс `Person` аккуратно отделяется от реализации средства вывода на экран, которая переносится в клиентский код.

Если же в клиентском коде потребуется вывести на экран имя и возраст из объекта типа `Person`, для этого достаточно обычным образом создать класс `PersonWriter`. Но в нашем случае это будет настолько простая реализация, что с таким же успехом можно было бы создать класс и одновременно передать его объекту типа `Person`, как показано ниже:

```
// Листинг 4.125
$person = new Person();
$person->output(
  new class implements PersonWriter
  {
    public function write(Person $person): void
    {
      print $person->getName() . " " . $person->getAge() . "\n";
    }
  }
);
```

Как видите, анонимный класс можно объявить с помощью ключевых слов `new class`, после которых указываются необходимые модификаторы расширения `extends` и реализации `implements`. Далее следует блок кода этого класса. Сразу после объявления анонимного класса неявно создается его экземпляр и передается в качестве аргумента параметра методу `output()`.

Замыкания в анонимных классах не поддерживаются. Иными словами, переменные, объявленные в более обширной (родительской) области видимости, в анонимном классе недоступны. Но в то же время *можно* передать их значения конструктору анонимного класса. В качестве примера создадим немного более сложный анонимный класс, в котором реализуется интерфейс `PersonWriter`:

```
// Листинг 4.126
$person = new Person();
$person->output(
```

```
    new class("/tmp/persondump") implements PersonWriter
{
    private $path;
    public function __construct(string $path)
    {
        $this->path = $path;
    }
    public function write(Person $person): void
    {
        file_put_contents($this->path, $person->getName() . " " .
                          $person->getAge() . "\n");
    }
}
);
```

В данном примере конструктору анонимного класса передается строковый аргумент, содержащий имя файла. Это значение хранится в свойстве `$path` данного класса и в конечном итоге используется в методе `write()` для записи информации из объекта в файл.

Безусловно, если анонимный класс начинает разрастаться и усложняться, то целесообразнее создать именованный класс в соответствующем файле класса. Это особенно справедливо, если анонимный класс приходится дублировать в нескольких местах[2].

Резюме

В этой главе мы попытались осветить тему более развитых объектно-ориентированных средств языка PHP. С некоторыми из них вам еще предстоит встретиться по мере чтения данной книги. В частности, мы будем часто возвращаться к абстрактным классам, исключениям и статическим методам.

В следующей главе мы немного отойдем от описания встроенных объектно-ориентированных средств языка PHP и рассмотрим классы и функции, предназначенные для облегчения работы с объектами.

---

[2] Ради удобочитаемости программы настоятельно рекомендуем отказаться от использования анонимных классов и всегда пользоваться именованными классами. Иначе через какое-то время ваша красивая и строгая объектно-ориентированная PHP-программа превратится в “лапшу” наподобие JavaScript-кода, разобраться в которой по прошествии некоторого времени не смогут даже сами разработчики. — *Примеч. перев.*

ГЛАВА 5

Средства для работы с объектами

Как пояснялось ранее, поддержка ООП в PHP осуществляется через такие языковые конструкции, как классы и методы. Кроме того, в PHP предусмотрен большой набор функций и классов, предназначенных для оказания помощи в обращении с объектами. В этой главе мы рассмотрим некоторые средства и методики, которые можно использовать для организации, тестирования объектов и классов и манипулирования ими.

В частности, в этой главе будут рассмотрены следующие вопросы.

- *Пространства имен.* Организация прикладного кода в отдельные пакетоподобные структуры.
- *Пути включения файлов.* Позволяют указать места централизованного хранения библиотечного кода.
- *Функции для исследования классов и объектов.* Предназначены для тестирования объектов, классов, свойств и методов.
- *Интерфейс Reflection API.* Эффективный набор встроенных классов, обеспечивающий беспрецедентный доступ к информации о классе во время выполнения программы.
- *Атрибуты.* Реализация *аннотаций* в PHP, которая представляет собой механизм, с помощью которого классы, методы, свойства и параметры могут быть усовершенствованы путем применения соответствующих дескрипторов в исходном коде.

PHP и пакеты

Пакет — это набор связанных классов и функций, обычно сгруппированных некоторым образом. Пакеты можно использовать для разделения частей системы на логические компоненты. В некоторых языках программирования пакеты распознаются формально, и для них создаются различные пространства имен. А в языке PHP такого понятия, как пакет, никогда

не существовало, но начиная с версии PHP 5.3 в нем поддерживаются пространства имен. Эта возможность будет рассмотрена в следующем разделе. Кроме того, следует хотя бы вкратце описать прежний способ организации классов в пакетоподобные структуры.

Пакеты и пространства имен в PHP

Несмотря на то что концепция пакетов в PHP внутренне не поддерживалась, разработчики традиционно использовали различные схемы именования и файловую систему для организации своего кода в пакетоподобные структуры.

До появления версии PHP 5.3 разработчики были вынуждены именовать свои файлы в совместно используемом контексте. Иными словами, если вы, например, определяли класс `ShoppingBasket`, он сразу же становился доступным всему вашему приложению. Это порождало две главные проблемы. Первая и самая неприятная была связана с потенциальным конфликтом имен в проекте. Такой конфликт может показаться маловероятным — ведь достаточно было помнить, что всем классам необходимо назначать уникальные имена! Но проблема в том, что в своих проектах вам все чаще приходится пользоваться библиотечным кодом. Это очень удобно, поскольку способствует повторному использованию кода. Но допустим, что в проекте используется следующая конструкция:

```
// Листинг 5.1
require_once __DIR__ . "/../useful/Outputter.php";
class Outputter
{
    // Вывод данных
}
```

Допустим также, что в проект вносится включаемый файл `useful/Outputter.php`:

```
// Листинг 5.2
class Outputter
{
    //
}
```

Нетрудно догадаться, что это приведет к следующей неустранимой ошибке:

```
PHP Fatal error: Cannot declare class Outputter because the name is
already in use in /var/popp/src/ch05/batch01/useful/Outputter.php on
line 4
```

До появления пространств имен существовал традиционный обходной прием для разрешения подобного затруднения. Он состоял в том, чтобы указать имя пакета перед именем класса и тем самым гарантировать однозначность имен классов:

```
// Листинг 5.3
// my/Outputer.php
require_once __DIR__ . "/../useful/Outputter.php";
class my_Outputter
{
    // Вывод данных
}

// Листинг 5.4
// useful/Outputter.php
class useful_Outputter
{
    //
}
```

Проблема такого решения в том, что по мере разрастания проекта имена классов становились все длиннее и длиннее. И хотя в этом не было ничего особо страшного, в конечном счете прикладной код становился неудобочитаемым, и разработчикам было все труднее держать в голове непомерно длинные имена классов. Кроме того, огромное количество времени уходило на исправление опечаток в таких именах.

Если вам придется сопровождать такой устаревший код, вы сможете обнаружить в нем подобные образцы условных обозначений имен классов. Именно по этой причине далее в этой главе мы вкратце рассмотрим старый способ организации классов в пакетоподобные структуры.

Спасительные пространства имен

Пространства имен появились в версии PHP 5.3. По существу, они представляют собой корзину, в которую можно поместить классы, функции и переменные. К ним можно обращаться в пределах одного пространства имен безо всякого уточнения. Для того же, чтобы обратиться к этим элементам за пределами пространства имен, придется импортировать целое пространство имен или воспользоваться ссылкой на него.

Непонятно? Поясним на конкретном примере. Ниже приведен предыдущий пример кода, переделанный с помощью пространств имен:

```
// Листинг 5.5
namespace my;
require_once __DIR__ . "/../useful/Outputter.php";
class Outputter
{
    // Вывод данных
}

// Листинг 5.6
namespace useful;
class Outputter
{
    //
}
```

Обратите внимание на ключевое слово `namespace`. Как вы, вероятно, догадались, оно устанавливает указанное пространство имен. При использовании пространств имен в коде их объявление должно находиться в первой инструкции исходного файла. В приведенном выше примере созданы два пространства имен: `my` и `useful`. Зачастую в прикладных программах употребляются более длинные пространства имен. Как правило, они начинаются с названия организации или проекта, а далее указывается название пакета. В языке PHP допускаются вложенные пространства имен. Для их разделения на уровни служит знак обратной косой черты, `'\'`, как показано ниже:

```
// Листинг 5.7
namespace popp\ch05\batch04\util;
class Debug
{
    public static function helloWorld(): void
    {
        print "Привет от Debug\n";
    }
}
```

Для организации информационного хранилища обычно выбирается имя, связанное с разрабатываемым программным продуктом или организацией. С этой целью можно выбрать один из доменов (например, `getinstance.com`), поскольку доменное имя однозначно определяет его владельца. Программисты на Java зачастую обозначают доменными име-

нами свои пакеты, указывая их в обратном порядке: от общего к частному. Но для примеров кода из данной книги выбрано пространство имен popp, сокращенно обозначающее название этой книги на английском языке. Выбрав имя для хранилища, можно перейти к именам пакетов. В данном случае имя пакета состоит из обозначения главы книги и номера группы. Это дает возможность организовать группы примеров кода в отдельные пакеты. Например, текущие примеры кода из этой главы находятся в пакете popp\ch05\batch04. И, наконец, исходный код можно дополнительно организовать по категориям, что я делаю с помощью слова util.

Как же теперь вызвать метод? На самом деле все зависит от того, в каком контексте это делается. Так, если метод вызывается из текущего пространства имен, то все происходит, как и прежде, т.е. метод вызывается непосредственно, как показано ниже:

```
// Листинг 5.8
Debug::helloWorld();
```

Это так называемое неквалифицированное (неуточненное) имя класса. Мы и так находимся в пространстве имен popp\ch05\batch04\util, и поэтому нам не нужно указывать перед именем класса какой-либо путь к нему. Если же требуется вызвать метод из данного класса за пределами текущего пространства имен, это придется делать следующим образом:

```
// Листинг 5.9
\popp\ch05\batch04\Debug::helloworld();
```

Как вы думаете, каким будет результат выполнения приведенного ниже фрагмента кода?

```
// Листинг 5.10
namespace main;
popp\ch05\batch04\Debug::helloworld();
```

Это непростой вопрос. На самом деле будет получено следующее сообщение об ошибке:

```
PHP Fatal error: Class 'popp\ch05\batch04\Debug' not found in...
```

Причина в том, что в данном коде использовано относительное имя пространства имен. Интерпретатор PHP попытается найти в текущем пространстве имен main имя popp\ch05\batch04\util, но не сможет его обнаружить. Чтобы устранить этот недостаток, следует указать абсолютное

имя пространства имен. Это делается таким же образом, как и при указании абсолютных путей к файлам и URL. С этой целью перед названием пространства имен следует указать разделительный знак '\', как показано ниже. Именно этот начальный знак обратной косой черты и сообщает интерпретатору PHP, что поиск классов следует начинать не с текущего, а с корневого пространства имен:

```
// Листинг 5.11
namespace main;
\popp\ch05\batch04\Debug::helloworld();
```

Но разве введение пространств имен не должно было сократить имена классов и упростить набор исходного кода? Безусловно, имя класса Debug очень краткое и понятное, но такой "многословный" вызов метода из этого класса получился лишь потому, что в данном случае было употреблено прежнее соглашение об именовании файлов. Дело можно упростить, воспользовавшись ключевым словом use. Оно позволяет снабжать псевдонимами другие пространства имен в текущем пространстве имен, как показано в следующем примере кода:

```
// Листинг 5.12
namespace main;
use popp\ch05\batch04\util;
util\Debug::helloWorld();
```

В данном примере импортируется пространство имен popp\ch05\batch04\util, которое неявно получает псевдоним util. Обратите внимание на то, что имя этого пространства имен не предваряется знаком обратной косой черты. Дело в том, что поиск пространства имен, указанного в качестве аргумента в операторе use, начинается с глобального, а не текущего пространства имен. Если же из нового пространства имен требуется только класс Debug, можно импортировать только его, а не все пространство в целом, как показано ниже:

```
// Листинг 5.13
namespace main;
use popp\ch05\batch04\util\Debug;
Debug::helloWorld();
```

Такое соглашение используется наиболее часто. Но что произойдет, если в пространстве имен main уже определен другой класс Debug? В качестве примера ниже приведен такой класс:

```
// Листинг 5.14
namespace popp\ch05\batch04;
class Debug
{
    public static function helloWorld(): void
    {
        print "Привет от popp\\ch05\\batch04\\Debug\n";
    }
}
```

А вот как выглядит вызывающий код из пространства имен `popp\ch05\batch04`, в котором происходит обращение к обоим классам `Debug`:

```
// Листинг 5.15
namespace popp\ch05\batch04;

use popp\ch05\batch04\util\Debug;
use popp\ch05\batch04\Debug;

Debug::helloWorld();
```

Как и следовало ожидать, выполнение этого фрагмента кода ведет к следующей ошибке:

```
PHP Fatal error: Cannot use popp\ch05\batch04\Debug as Debug
because the name is already in use in...
```

Похоже, что круг замкнулся, и мы снова вернулись к конфликту имен классов. Правда, из подобного затруднения нетрудно выйти, явно указав псевдоним класса, как показано ниже:

```
// Листинг 5.16
namespace popp\ch05\batch04;

use popp\ch05\batch04\util\Debug;
use popp\ch05\batch04\Debug as CoreDebug;

CoreDebug::helloWorld();
```

Воспользовавшись ключевым словом `as` в операторе `use`, мы изменили псевдоним класса `Debug` на `CoreDebug`. Если вы пишете код в текущем пространстве имен и хотите обратиться к классу, трейту или интерфейсу, находящемуся в корневом пространстве (вне пространства имен — например, такие базовые классы, как `Exception`, `Error`, `Closure`), просто укажите перед его именем знак обратной косой черты. Ниже приведен пример класса, определенного в корневом пространстве:

```
// Листинг 5.17
class TreeLister
{
    public static function helloWorld(): void
    {
        print "Привет из корневого пространства имен\n";
    }
}
```

А вот пример кода в некотором пространстве имен:

```
// Листинг 5.18
namespace popp\ch05\batch04\util;
class TreeLister
{
    public static function helloWorld(): void
    {
        print "Привет из " . __NAMESPACE__ . "\n";
    }
}
```

```
// Листинг 5.19
namespace popp\ch05\batch04;

use popp\ch05\batch04\util\TreeLister;

TreeLister::helloWorld();  // Локальный доступ
\TreeLister::helloWorld(); // Доступ к корню
```

В коде из локального пространства имен объявлен собственный класс `TreeLister`. Клиентский код использует локальную версию, указывая полный путь с помощью инструкции `use`. Имя, квалифицированное с помощью обратной косой черты, обращается к аналогично именованному классу в корневом пространстве имен.

Ниже приведен результат выполнения предыдущего фрагмента кода:

```
Привет из popp\ch05\batch04\util
Привет из глобального пространства имен
```

Это очень ценный пример, поскольку в нем наглядно показано, как пользоваться константой `__NAMESPACE__`. Она содержит имя текущего пространства имен, что может пригодиться при отладке кода.

С помощью описанного выше синтаксиса можно определить несколько пространств имен в одном файле. Можно также употребить альтернативный синтаксис, где после ключевого имени `namespace` и названия пространства имен указываются фигурные скобки, в которые заключается блок кода:

```
// Листинг 5.20
namespace com\getinstance\util
{
    class Debug
    {
        public static function helloWorld(): void
        {
            print "Привет от Debug\n";
        }
    }
}
namespace other
{
    \com\getinstance\util\Debug::helloWorld();
}
```

Для того чтобы объединить несколько пространств имен в одном файле, предпочтительнее использовать второй синтаксис (хотя в одном файле обычно рекомендуется определять только одно пространство имен).

> **На заметку** В одном файле нельзя употреблять одновременно оба упомянутых выше синтаксиса определения пространств имен. Выберите какой-нибудь один из них и придерживайтесь именно его.

Имитация пакетов с помощью файловой системы

Независимо от используемой версии PHP классы можно упорядочивать с помощью файловой системы, которая отдаленно напоминает пакетную структуру. Например, можно создать каталоги `util` и `business` и включать находящиеся в них файлы классов с помощью инструкции `require_once()` следующим образом:

```
// Листинг 5.21
require_once('business/Customer.php');
require_once('util/WebTools.php');
```

С тем же успехом можно также употребить инструкцию `include_once()`. Единственное различие инструкций `require_once()` и `include_once()` заключается в обработке ошибок. Файл, к которому происходит обращение с помощью инструкции `require_once()`, приведет к остановке всего процесса выполнения программы, если в нем возникнет ошибка. Аналогичная ошибка, обнаруженная в результате применения инструкции `include_once()`, приведет лишь к выдаче предупреждения и прекраще-

нию выполнения включаемого файла, после чего выполнение вызвавшего его кода будет продолжено. Поэтому инструкции `require()` и `require_once()` лучше выбирать для включения библиотечных файлов, а инструкции `include()` и `include_once()` лучше подходят для выполнения таких действий, как шаблонная обработка.

На заметку Синтаксические конструкции `require()` и `require_once()` на самом деле являются инструкциями, а не функциями. Это означает, что при их использовании можно опускать скобки. Лично я предпочитаю использовать скобки в любом случае. Но если вы последуете моему примеру, будьте готовы, что вам будут надоедать педанты, которые не преминут указать вам на ошибку.

На рис. 5.1 показано, как выглядят пакеты `util` и `business` в диспетчере файлов Nautilus.

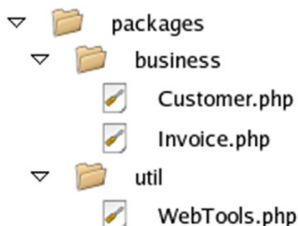


Рис. 5.1. *Пакеты PHP, организованные с использованием файловой системы*

На заметку Инструкции `require_once()` передается путь к файлу, содержащему код PHP, который будет включен в текущий сценарий и выполнен в нем. Включение указанного файла в текущий сценарий выполняется только один раз, и если этот файл был ранее включен в сценарий, то повторное его включение не происходит. Такой однократный подход особенно удобен при доступе к библиотечному коду, поскольку тем самым предотвращается случайное переопределение классов и функций. Подобная ситуация может возникнуть в том случае, если один и тот же файл включается в разные части сценария, выполняющегося в одном процессе с помощью инструкции `require()` или `include()`.

Обычно предпочтение отдается инструкциям `require()` и `require_once()`,

а не сходным с ними инструкциям `include()` и `include_once()`. Дело в том, что неустранимая ошибка, обнаруженная в файле, к которому происходит обращение с помощью инструкции `require()`, приводит к остановке всего сценария. Аналогичная ошибка, обнаруженная в файле, к которому происходит обращение с помощью инструкции `include()`, прервет выполнение включаемого файла, но в вызывающем сценарии будет сгенерировано только предупрежде-

ние. Первый вариант считается более безопасным, хотя и более радикальным.

Применение инструкции `require_once()` требует определенных издержек по сравнению с инструкцией `require()`. Если требуется сократить время выполнения сценария до минимума, то лучше воспользоваться оператором `require()`. И, как это часто бывает, приходится выбирать между эффективностью и удобством.

Что касается PHP, то в такой организации файлов нет ничего особенного. Библиотечные сценарии просто размещаются в разных каталогах, что придает проекту четкую организацию. При этом пространствами имен или соглашениями об именовании файлов и файловой системой можно пользоваться параллельно.

Именование в стиле PEAR

До внедрения пространств имен разработчики вынуждены были прибегать к определенным соглашениям об именовании файлов, чтобы исключить их конфликты. К числу наиболее употребительных среди них относится фальшивая организация пространств имен, которой придерживались разработчики PEAR.

На заметку Сокращение "PEAR" означает *PHP Extension and Application Repository* (Хранилище расширений и приложений PHP). Это официально поддерживаемый архив пакетов и средств, расширяющих функциональные возможности PHP. Основные пакеты PEAR включены в дистрибутив PHP, а другие пакеты можно добавить с помощью простой утилиты командной строки. Просмотреть список пакетов PEAR можно по адресу `http://pear.php.net`.

Для определения пакетов в PEAR используется файловая система, как пояснялось выше. До внедрения пространств имен каждый класс получал имя, исходя из пути к нему, в котором имена каталогов разделялись знаком подчеркивания.

Например, в PEAR включен пакет под именем "XML", в который вложен пакет RPC. В пакете RPC имеется исходный файл, который называется `Server.php`. Класс, определенный в исходном файле `Server.php`, называется не `Server`, как можно было ожидать. В противном случае рано или поздно произошел бы конфликт с другим классом `Server`, находящимся в каком-нибудь другом пакете PEAR или пользовательском коде. Чтобы этого не произошло, класс назван `XML_RPC_Server`. С одной стороны, имена классов, безусловно, становятся слишком громоздкими и малопривлека-

тельными. Но с другой стороны, код становится более удобочитаемым, потому что имя класса описывает его контекст.

Пути включения файлов

Для организации компонентов кода можно воспользоваться одним из двух способов. Первый из них был описан выше и состоит в размещении файлов проекта по разным каталогам файловой системы. Но ведь необходимо каким-то образом обеспечить взаимодействие компонентов. Выше мы уже коснулись путей включения файлов в проект. Включая файл, следует указать относительный (для текущего рабочего каталога) или полный путь к этому файлу в используемой файловой системе.

На заметку Важно не только понимать принцип действия путей включения файлов и разбираться в вопросах предоставления требующихся файлов, но также иметь в виду, что во многих современных системах уже не используются операторы `require` для каждого класса. Вместо этого в них применяются пространства имен в определенном сочетании с автозагрузкой классов. Об автозагрузке речь пойдет ниже, а более подробные сведения о ней, практические рекомендации и инструментальные средства будут представлены в главах 15, "Стандарты PHP", и 16, "Создание и использование компонентов PHP средствами Composer".

В приведенных до сих пор примерах порой указывалось фиксированное отношение между требующими и требуемыми файлами:

```
// Листинг 5.22
require_once __DIR__ . "/../useful/Outputter.php";
```

Такой прием вполне работоспособен, за исключением жестко закодированных отношений между файлами. В данном случае подразумевается непременное наличие каталога `useful` наряду с каталогом, содержащим вызывающий класс. Вероятно, худший способ, чем указать запутанный относительный путь наподобие приведенного ниже, трудно придумать:

```
// Листинг 5.23
require_once('../../projectlib/business/User.php');
```

Недостаток такого способа заключается в том, что путь указан не относительно того файла, в котором содержится инструкция `require_once()`, а относительно сконфигурированного вызывающего контекста, которым нередко (хотя и не всегда) оказывается текущий рабочий каталог. Такие пути к файлам служат источником недоразумений и почти всегда, как по-

казывает опыт, явно указывают на то, что система требует значительного усовершенствования и в других местах.

Безусловно, можно указать и абсолютный путь к файлу, как в следующем примере:

```
// Листинг 5.24
require_once('/home/john/projectlib/business/User.php');
```

Это будет работать для одного экземпляра, но это негибкий подход. Ведь указывая пути настолько подробно, можно “заморозить” библиотечный файл в конкретном контексте. И всякий раз, когда проект будет устанавливаться на новом сервере, придется вносить изменения во все инструкции `require` с учетом нового пути к файлу. В итоге библиотеки будет трудно переносить и будет практически невозможно делать их общими для разных проектов, не прибегая к копированию. Но в любом случае теряется смысл в размещении пакетов в дополнительных каталогах, поскольку неясно, о каком пакете идет речь: `business` или `projectlib/business`?

Если требуется включить файлы в свой код вручную, то лучше всего отвязать вызывающий код от библиотеки. Вы уже видели структуры наподобие

```
// Листинг 5.25
require_once('business/User.php');
```

В предыдущих примерах, в которых использовались пути, подобные показанному, мы неявно предполагали использование относительных путей. Другими словами, `business/User.php` был функционально идентичен `./business/User.php`. Но что если предыдущая инструкция `required` должна работать из любого каталога в системе? Вы можете сделать это с использованием пути включения. Это список каталогов, в которых PHP выполняет поиск при указании требуемого файла. Вы можете дополнить этот список, изменяя директиву `include_path`. Обычно `include_path` устанавливается в файле основной конфигурации PHP `php.ini`. Эта директива представляет собой список каталогов, разделенных двоеточиями в Unix-подобных системах, и точками с запятой — в Windows:

```
include_path = ".:/usr/local/lib/php-libraries"
```

Если используется сервер Apache, директиву `include_path` можно указать в глобальном файле конфигурации сервера, который обычно называется `httpd.conf`, или в файле конфигурации для конкретного каталога, который обычно называется `.htaccess`, с использованием следующего синтаксиса:

```
php_value include_path value .:/usr/local/lib/php-libraries
```

На заметку Файлы конфигурации `.htaccess` обычно используются веб-серверами ряда поставщиков услуг хостинга, обеспечивая очень ограниченный доступ к среде сервера.

Если при использовании таких функций файловой системы, как `fopen()` или `require()` с неабсолютным путем к файлу, этот файл не находится в иерархии текущего рабочего каталога, его поиск автоматически выполняется в каталогах, включенных в путь поиска, начиная с первого каталога в списке (чтобы активизировать такой режим поиска в функции `fopen()`, в список ее аргументов следует включить специальный флаг). Когда искомый файл найден, поиск прекращается и соответствующая функция или инструкция завершает свою работу.

Поэтому, размещая каталог с пакетами во включаемом каталоге, достаточно указать в операторах `require()` имя файла и название пакета.

Чтобы поддерживать собственный библиотечный каталог, добавьте его имя в директиву `include_path`, отредактировав файл конфигурации `php.ini`. Если вы пользуетесь серверным модулем PHP, не забудьте перезагрузить сервер Apache, чтобы изменения вступили в силу.

Если у вас нет прав доступа для редактирования файла конфигурации `php.ini`, задайте пути включения файлов непосредственно из сценария с помощью функции `set_include_path()`. Этой функции передается список каталогов (в том же виде, что и для файла конфигурации `php.ini`), который для текущего процесса заменяет список, указанный в директиве `include_path`. В файле конфигурации `php.ini`, вероятно, уже определено подходящее значение для директивы `include_path`, и вместо того чтобы его переписать, вы можете получить его с помощью функции `get_include_path()`, а затем добавить к нему свой каталог. Ниже показано, как добавить каталог в текущий путь включения файлов:

```
set_include_path( get_include_path()
    . PATH_SEPARATOR . "/home/john/phplib/");
```

Вместо константы `PATH_SEPARATOR` будет подставлен знак разделения списка каталогов, принятый в текущей операционной системе. Напомним, что в Unix — это двоеточие (`:`), а в Windows — точка с запятой (`;`). Пользоваться константой `PATH_SEPARATOR` рекомендуется из соображений переносимости кода.

Автозагрузка

Несмотря на все преимущества применения инструкции `require_once()` вместе с путями включения файлов многие разработчики избегают таких инструкций, используя вместо них автозагрузку.

На заметку В предыдущих изданиях этой книги рассматривалась встроенная функция `__autoload()`, которая предоставляла приближенную версию функциональности, рассматриваемой в данном разделе. Эта функция была объявлена устаревшей в PHP 7.2.0 и удалена в PHP 8.

Для этого классы необходимо организовать таким образом, чтобы каждый из них размещался в отдельном исходном файле. Имя каждого такого файла должно быть непосредственно связано с именем класса, который в нем содержится. Например, класс `ShopProduct` можно определить в исходном файле `ShopProduct.php` и поместить в каталог, соответствующий пространству имен данного класса.

Для автоматизации процесса включения в программу файлов классов в PHP, начиная с версии 5, предусмотрена возможность их автозагрузки. Ее стандартная поддержка очень проста, но от этого она не становится менее полезной. Для ее включения достаточно вызвать функцию `spl_autoload_register()` без параметров. После этого активизируется средство автозагрузки классов, которое автоматически вызывает встроенную функцию `spl_autoload()` при попытке получить экземпляр неизвестного класса. В качестве параметра функции `spl_autoload()` передается имя неизвестного класса, которое затем преобразуется в имя файла. Для этого имя класса преобразуется в нижний регистр букв и по очереди дополняется всеми зарегистрированными стандартными расширениями (сначала — `.inc`, а затем — `.php`). Далее функция пытается загрузить содержимое указанного файла в программу, полагая, что в нем должно содержаться определение неизвестного класса.

Ниже приведен простой пример организации автозагрузки:

```
// Листинг 5.26
spl_autoload_register();
$writer = new Writer();
```

Если в программу еще не включен файл с определением класса `Writer`, то при попытке получить экземпляр этого класса возникнет исключительная ситуация. Но поскольку ранее был установлен режим автозагрузки, интерпретатор PHP попытается включить в программу сначала

файл `writer.inc`, а затем — `writer.php`, если первый файл обнаружить не удалось. Если во включаемом файле содержится определение класса `Writer`, интерпретатор снова попытается создать его экземпляр и выполнение программы продолжится.

Стандартное поведение автозагрузки поддерживает пространства имен, заменяя имя каталога для каждого пакета:

```
// Листинг 5.27
spl_autoload_register();
$writer = new util\Writer();
```

При выполнении приведенного ниже фрагмента кода интерпретатор PHP будет искать файл `writer.php` в каталоге `util` (обратите внимание на указание имени файла в нижнем регистре).

Но что если в именах файлов классов должен учитываться регистр букв (т.е. в них могут быть как прописные, так и строчные буквы)? Так, если определение класса `Writer` будет размещено в файле `Writer.php`, то при реализации автозагрузки по умолчанию файл этого класса не будет найден.

К счастью, имеется возможность зарегистрировать пользовательскую функцию автозагрузки, чтобы учитывать различные соглашения об именовании файлов. Для этого достаточно передать функции `spl_autoload_register()` ссылку на специальную или анонимную функцию. Такой функции автозагрузки должен передаваться единственный аргумент. Тогда при попытке получить экземпляр неизвестного класса интерпретатор PHP вызовет эту специальную функцию автозагрузки, передав ей в качестве строкового параметра имя искомого класса. При этом в функции автозагрузки вам придется самостоятельно реализовать стратегию поиска и включения в программу нужных файлов классов. После вызова специальной функции автозагрузки интерпретатор PHP снова попытается получить экземпляр указанного класса.

Ниже приведен простой пример функции автозагрузки вместе с загружаемым классом:

```
// Листинг 5.28
class Blah
{
    public function wave(): void
    {
        print "saying hi from root";
    }
}
```

```
// Листинг 5.29
$basic = function(string $classname)
{
    $file = __DIR__ . "/" . "{$classname}.php";

    if (file_exists($file))
    {
        require_once($file);
    }
};
\spl_autoload_register($basic);
$blah = new Blah();
$blah->wave();
```

После первоначальной неудачной попытки создать экземпляр класса `Blah` интерпретатор PHP обнаружит, что с помощью функции `spl_autoload_register()` была зарегистрирована специальная функция автозагрузки, которой в качестве параметра было передано имя данного класса, указанное в символьной строке `"Blah"`. В данной реализации мы просто пытаемся включить в программу файл `Blah.php`. Это, конечно, получится лишь в том случае, если включаемый файл находится в том же каталоге, где и файл, в котором объявлена функция автозагрузки. На практике конфигурирование путей включения файлов придется сочетать с логикой автозагрузки. Именно такая методика применяется для реализации автозагрузки в диспетчере зависимостей Composer.

Если же требуется обеспечить поддержку устаревших соглашений об именовании файлов, процесс включения пакетов PEAR можно автоматизировать следующим образом:

```
// Листинг 5.30
class util_Blah
{
    public function wave(): void
    {
        print "Привет из файла с подчеркиваниями";
    }
}

// Листинг 5.31
$underscores = function(string $classname)
{
    $path = str_replace('_', DIRECTORY_SEPARATOR, $classname);
    $path = __DIR__ . "/$path";
```

```
    if (file_exists("{$path}.php"))
    {
        require_once("{$path}.php");
    }
};
\spl_autoload_register($underscores);
$blah = new util_Blah();
$blah->wave();
```

Как видите, функция автозагрузки преобразует знаки подчеркивания, содержащиеся в переданном ей аргументе `$classname`, в знак разделения каталогов, заданный в константе `DIRECTORY_SEPARATOR` (`'/'` для Unix и `'\'` для Windows). Затем мы пытаемся включить в программу файл класса `util/Blah.php`. Если такой файл существует и класс, который в нем содержится, назван верно, то экземпляр объекта будет получен без ошибок. Разумеется, для этого необходимо, чтобы программист соблюдал соглашение об именовании файлов, которое запрещает употребление знака подчеркивания в именах классов, за исключением тех случаев, когда этот знак служит для разделения пакетов.

А как быть с пространствами имен? Как упоминалось выше, в реализации автозагрузки по умолчанию поддерживаются пространства имен. Но если мы заменяем стандартную реализацию собственной, обязанность поддерживать пространства имен полностью возлагается на нас. Для этого достаточно проверить наличие в длинном имени класса знаков обратной косой черты и заменить их знаком разделения каталогов, принятым в используемой операционной системе:

```
// Листинг 5.32
namespace util;
class LocalPath
{
    public function wave(): void
    {
        print "Привет от " . get_class();
    }
}

// Листинг 5.33
$namespaces = function(string $path)
{
    if (preg_match('/\\\\/', $path))
    {
        $path = str_replace('\\', DIRECTORY_SEPARATOR, $path);
    }
```

```
    if (file_exists("{$path}.php"))
    {
        require_once("{$path}.php");
    }
};
\spl_autoload_register($namespaces);
$obj = new util\LocalPath();
$obj->wave();
```

Значение, передаваемое функции автозагрузки, всегда нормализовано и содержит полностью квалифицированное имя класса без начального знака обратной косой черты. Поэтому в момент получения экземпляра класса можно не беспокоиться о его псевдониме или относительной ссылке на пространство имен.

Обратите внимание, что это решение ни в коем случае не идеально. Функция `file_exists()` не учитывает путь включения, поэтому она не будет точно отражать все обстоятельства, при которых инструкция `require_once` будет работать идеально хорошо. Имеются различные решения этой проблемы. Вы можете создать собственную версию `file_exists()`, умеющую работать с путями, или пытаться затребовать файл в инструкции `try` (перехватывая в этом случае `Error`, а не `Exception`). К счастью, PHP предоставляет функцию `stream_resolve_include_path()`, которая возвращает строку, представляющую абсолютное имя файла для предоставленного пути или, что важно для наших целей, `false`, если файл в пути включения не найден:

```
// Листинг 5.34
$namespaces = function(string $path)
{
    if (preg_match('/\\\\/', $path))
    {
        $path = str_replace('\\', DIRECTORY_SEPARATOR, $path);
    }

    if (\stream_resolve_include_path("{$path}.php") !== false)
    {
        require_once("{$path}.php");
    }
};
\spl_autoload_register($namespaces);
$obj = new util\LocalPath();
$obj->wave();
```

Но что если при автозагрузке требуется поддерживать как именование классов в стиле PEAR, так и пространства имен *одновременно*? Для этого можно объединить реализации двух приведенных выше функций автозагрузки в одну пользовательскую функцию или воспользоваться стеком функций автозагрузки, который организуется в функции `spl_autoload_register()`, как показано ниже:

```
// Листинг 5.35
$underscores = function(string $classname)
{
    $path = str_replace('_', DIRECTORY_SEPARATOR, $classname);
    $path = __DIR__ . "/$path";

    if (\stream_resolve_include_path("{$path}.php") !== false)
    {
        require_once("{$path}.php");
    }
}
$namespaces = function(string $path)
{
    if (preg_match('/\\\\/', $path))
    {
        $path = str_replace('\\', DIRECTORY_SEPARATOR, $path);
    }

    if (\stream_resolve_include_path("{$path}.php") !== false)
    {
        require_once("{$path}.php");
    }
};
\spl_autoload_register($namespaces);
\spl_autoload_register($underscores);
$blah = new util_Blah();
$blah->wave();
$obj = new util\LocalPath();
$obj->wave();
```

Когда интерпретатор PHP обнаружит неизвестный класс, он поочередно вызовет функции автозагрузки (в соответствии с порядком их регистрации), останавливаясь, когда инстанцирование станет возможным или когда все варианты будут исчерпаны.

Очевидно, что при использовании описанного выше стека функций автозагрузки производительность снижается из-за высоких накладных расходов. Так почему же он все-таки поддерживается в PHP? Дело в том, что в реальных проектах пространства имен зачастую сочетаются с именами

классов, разделяемыми знаком подчеркивания. Но в некоторых компонентах крупных систем и в сторонних библиотеках может возникнуть потребность реализовать собственный механизм автозагрузки. Поддержка стека функций автозагрузки как раз позволяет зарегистрировать независимые механизмы автозагрузки, не мешающие один другому. Кроме того, библиотека, в которой требуется лишь кратковременно воспользоваться механизмом автозагрузки, может передать имя своей специальной функции автозагрузки (или ссылку на нее, если она анонимная) в качестве параметра функции `spl_autoload_unregister()`, чтобы удалить ее из стека и таким образом произвести самоочистку.

Функции для исследования классов и объектов

В языке PHP предусмотрен широкий набор функций для проверки классов и объектов. Для чего это нужно? Вы ведь, скорее всего, сами создали большинство классов, которые используются в сценарии.

На самом деле во время выполнения не всегда известно, какие именно классы используются. Например, вы могли создать систему для "прозрачной" работы с переопределяемыми классами, разработанными сторонними производителями. Обычно экземпляр объекта в PHP получается только на основании имени класса. Однако в языке PHP разрешается ссылаться на классы динамически с использованием символьных строк, как показано ниже:

```
// Листинг 5.36
namespace tasks;
class Task
{
    public function doSpeak()
    {
        print "Привет\n";
    }
}

// Листинг 5.37
$classname = "Task";
require_once("tasks/{$classname}.php");
$classname = "tasks\\$classname";
$myObj = new $classname();
$myObj->doSpeak();
```

Символьную строку, присвоенную переменной `$classname`, можно получить из файла конфигурации или путем сравнения результатов веб-запроса с содержимым каталога. Затем эту строку можно использовать для загрузки файла класса и создания экземпляра объекта. Обратите внимание на то, что в приведенном выше фрагменте кода перед именем класса указано пространство имен.

Как правило, подобные операции выполняются, когда требуется, чтобы система могла запускать созданные пользователем подключаемые модули. Но прежде, чем делать такие рискованные вещи в реальном проекте, необходимо убедиться, что указанный класс существует, в нем имеются нужные вам методы и т.д.

На заметку Даже приняв все меры предосторожности, вы должны быть предельно внимательны при динамической установке сторонних подключаемых модулей. Нельзя автоматически запускать код, полученный от неизвестных пользователей. Дело в том, что любой установленный вами подключаемый модуль, как правило, запускается с теми же правами доступа, что и основной код вашего приложения. В итоге злоумышленник, написавший такой модуль, может полностью вывести из строя ваше приложение и даже сервер, на котором оно выполняется.

Но сказанное выше вовсе не означает, что следует отказаться от подключаемых модулей. Расширение функциональных возможностей базовой системы с помощью сторонних подключаемых модулей позволяет добиться большей гибкости. Для повышения уровня безопасности необходимо создать специальный каталог, в котором будут размещаться подключаемые модули. Доступ к этому каталогу следует предоставить только системному администратору, который и должен заниматься установкой подключаемых модулей как непосредственно, так и через специальное приложение, защищенное паролем. А перед установкой подключаемого модуля администратор должен лично проконтролировать его исходный код или получить его из заслуживающего доверия источника. Именно так реализована работа с подключаемыми модулями на популярной платформе WordPress для разработки веб-сайтов и блогов.

Некоторые функции для работы с классами заменены более эффективным прикладным интерфейсом Reflection API, который будет рассматриваться далее в этой главе. Но простота и удобство использования функций иногда делают их более предпочтительными. Именно по этой причине мы и рассмотрим их в первую очередь.

Поиск классов

Функции `class_exists()` передается строка, содержащая имя класса. Она возвращает логическое значение `true`, если класс существует, а иначе — логическое значение `false`. С помощью этой функции можно сделать более безопасным фрагмент кода из предыдущего примера, как показано ниже:

```
// Листинг 5.38
$base = __DIR__;
$classname = "Task";
$path = "{$base}/tasks/{$classname}.php";

if (! file_exists($path))
{
    throw new \Exception("Файл {$path} не найден");
}

require_once($path);
$qclassname = "tasks\\$classname";

if (! class_exists($qclassname))
{
    throw new Exception("Класс {$qclassname} не найден");
}

$myObj = new $qclassname();
$myObj->doSpeak();
```

Безусловно, нет никакой гарантии, что конструктору рассматриваемого здесь класса не потребуются аргументы. Для достижения такого уровня безопасности при программировании необходимо обратиться к интерфейсу Reflection API, описанному ниже в этой главе. Тем не менее с помощью функции `class_exists()` можно убедиться, что нужные классы существуют, прежде чем обращаться к этому интерфейсу.

Чтобы получить массив всех классов, определенных в сценарии, можно воспользоваться функцией `get_declared_classes()`, как показано ниже:

```
// Листинг 5.39
print_r(get_declared_classes());
```

На заметку Не забывайте, что к данным, полученным из внешних источников, всегда следует относиться с осторожностью. Об этом мы говорили выше. Перед использованием всегда проверяйте полученные данные и выполняйте их предварительную обработку. Так, если от пользователя получен путь к файлу, все точки и разделители каталогов в нем следует экранировать или удалить, предотвратив тем самым возможность взлома сайта, когда недобросовестный пользователь может заменить имена каталогов и включить в них нежелательные файлы. Но когда выше описывались способы расширения базовой системы подключаемыми модулями, то имелось в виду, что их установкой должен заниматься человек, которому предоставлены соответствующие полномочия, а не посторонний пользователь.

В итоге будет выведен список встроенных и определенных пользователем классов. Не следует, однако, забывать, что данная функция возвращает только те классы, которые были объявлены к моменту ее вызова. Ведь, выполнив в дальнейшем инструкцию `require()` или `require_once()`, можно тем самым дополнить число классов, применяемых в сценарии.

Получение сведений об объекте или классе

Как вам, должно быть, уже известно, с помощью объявлений типов классов можно ограничить типы аргументов метода некоторого объекта. Но даже используя эту возможность, не всегда можно быть уверенным в типе объекта. Для проверки типа объекта имеется целый ряд инструментальных средств. Прежде всего, класс, а следовательно, и тип объекта можно проверить с помощью функции `get_class()`. Этой функции в качестве аргумента передается любой объект, а она возвращает имя его класса в виде символьной строки, как демонстрируется в следующем примере кода:

```
// Листинг 5.40
$product = self::getProduct();

if (get_class($product) === 'popp\ch05\batch05\CDProduct')
{
    print "\$product является объектом класса CDProduct\n";
}
```

В данном фрагменте кода мы получаем *нечто* из метода `getProduct()`. Чтобы убедиться, что это объект типа `CDProduct`, мы вызываем функцию `get_class()`.

На заметку Классы `CDProduct` и `BookProduct` описаны в главе 3, "Основные положения об объектах".

Ниже приведено определение метода `getProduct()`:

```
// Листинг 5.41
public static function getProduct()
{
    return new CDProduct(
        "Классическая музыка. Лучшее",
        "Антонио",
        "Вивальди",
        10.99,
        60.33
    );
}
```

Метод `getProduct()` просто создает экземпляр объекта типа `CDProduct` и возвращает его. Мы еще не раз воспользуемся преимуществами данного метода в этом разделе.

Функция `get_class()` выдает слишком конкретную информацию. Обычно же требуются более общие сведения о принадлежности проверяемого объекта к семейству классов. Допустим, требуется выяснить, принадлежит ли объект семейству класса `ShopProduct`, но при этом не имеет значения, к какому конкретно классу он относится: `BookProduct` или `CDProduct`. Для этой цели в языке PHP предусмотрена оператор `instanceof`.

На заметку В версии PHP 4 оператор `instanceof` не поддерживается. Вместо него в версии PHP 4 была предусмотрена функция `is_a()`, которая в версии PHP 5.0 объявлена устаревшей. Начиная с версии PHP 5.3 этой функцией можно пользоваться снова.

Оператор `instanceof` выполняется над двумя операндами: объектом, который требуется проверить (этот операнд указывается слева от ключевого слова `instanceof`), и именем класса или интерфейса, указываемого справа. В итоге операция `instanceof` возвращает значение `true`, если объект является экземпляром класса указанного типа:

```
// Листинг 5.42
$product = self::getProduct();

if ($product instanceof \popp\ch05\batch05\CDProduct)
{
    print "\$product является экземпляром класса CDProduct\n";
}
```

Получение полностью квалифицированной строковой ссылки на класс

Благодаря внедрению пространств имен удалось избежать множества затруднений, которые возникали при использовании объектно-ориентированных средств языка PHP. В итоге мы навсегда избавились от необходимости создавать длинные и уродливые имена классов, а также от опасности возникновения конфликта имен, разумеется, кроме устаревшего кода. С другой стороны, при использовании псевдонимов классов или относительных ссылок на пространства имен иногда нелегко выяснить, являются ли пути к некоторым классам полностью квалифицированными.

Ниже приведено несколько примеров, когда нелегко определить имя класса:

```
// Листинг 5.43
namespace mypackage;
use util as u;
use util\db\Querier as q;
class Local
{
}
// Чему соответствует:

// Псевдоним пространства имен
// u\Writer;

// Псевдоним класса
// q;

// Ссылка на класс в локальном контексте
// Local
```

На самом деле не так уж и трудно определить, во что превращаются приведенные выше ссылки на класс. Намного труднее написать код, в котором обрабатывались бы все возможные ситуации. Так, чтобы проанализировать ссылку `u\Writer`, средству автоматического разрешения имен придется определить, что `u` — это псевдоним `util` и что `util` не является пространством имен. К счастью, в версии PHP 5.5 был введен синтаксис *ИмяКласса*`::class`. Он позволяет добавить к ссылке на класс операцию разрешения области видимости и ключевое слово `class`, чтобы выяснить полностью квалифицированное имя класса:

```
// Листинг 5.44
print u\Writer::class . "\n";
print q::class . "\n";
print Local::class . "\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
util\Writer
util\db\Querier
mypackage\Local
```

Начиная с PHP 8 вы можете также вызывать `::class` для объекта. Так, например, для экземпляра `ShopProduct` я могу получить полное имя класса:

```
// Листинг 5.45
$bookp = new BookProduct(
        "Классическая музыка. Лучшее",
        "Антонио",
        "Вивальди",
        10.99,
        60.33
    );
print $bookp::class;
```

Этот код приводит к выводу

```
popp\ch04\batch02\BookProduct
```

Обратите внимание, что этот удобный синтаксис не предлагает новую функциональность — вы уже встречались с функцией `get_class()`, которая обеспечивает тот же результат.

Получение информации о методах

Чтобы получить список всех методов класса, можно воспользоваться функцией `get_class_methods()`, как показано ниже. В качестве аргумента ей передается имя класса, а она возвращает массив, содержащий имена всех методов класса:

```
// Листинг 5.46
print_r(get_class_methods('\\popp\\ch04\\batch02\\BookProduct'));
```

Если класс `BookProduct` существует, то вы можете увидеть результат наподобие следующего:

```
Array
(
    [0] => __construct
    [1] => getNumberOfPages
    [2] => getSummaryLine
    [3] => getPrice
    [4] => setID
    [5] => getProducerFirstName
    [6] => getProducerMainName
    [7] => setDiscount
    [8] => getDiscount
    [9] => getTitle
    [10] => getProducer
    [11] => getInstance
)
```

В этом примере передается строка, содержащая имя класса функции `get_class_methods()`, а возвращаемый ею массив выводится с помощью функции `print_r()`. С таким же успехом можно было бы передать функции `get_class_methods()` *объект* и получить аналогичный результат. Но в этом случае в возвращаемый список будут включены только имена открытых методов.

Как было показано выше, имя метода можно сохранить в строковой переменной и вызвать его динамически вместе с объектом следующим образом:

```
// Листинг 5.47
$product = self::getProduct();
$method = "getTitle";       // Имя метода
print $product->$method(); // Вызов метода
```

Безусловно, такой подход таит в себе опасность. Что, например, произойдет, если метода не существует? Как и следовало ожидать, сценарий завершится ошибкой. Выше уже рассматривался один способ проверки существования метода:

```
// Листинг 5.48
if (in_array($method, get_class_methods($product)))
{
    print $product->$method(); // Вызов метода
}
```

Прежде чем вызывать метод, мы проверяем, присутствует ли его имя в массиве, возвращенном функцией `get_class_methods()`.

В PHP для этой цели предусмотрены более специализированные инструментальные средства. Имена методов так или иначе можно проверить

с помощью двух функций: `is_callable()` и `method_exists()`. Из них более сложной оказывается функция `is_callable()`. В качестве первого аргумента ей передается строковая переменная, определяющая имя функции. Если указанная функция существует и ее можно вызвать, функция `is_callable()` возвращает логическое значение `true`. Чтобы организовать такую же проверку для метода, вместо имени функции придется передать массив. Этот массив должен содержать ссылку на объект или имя класса в качестве первого элемента и имя проверяемого метода — в качестве второго. Функция `is_callable()` возвратит логическое значение `true`, если указанный метод существует в классе:

```
// Листинг 5.49
if (is_callable([$product, $method]))
{
    print $product->$method(); // Вызов метода
}
```

У функции `is_callable()` имеется и второй необязательный аргумент, принимающий логическое значение. Если задать в нем логическое значение `true`, то данная функция проверит корректность синтаксиса указанного имени функции или метода, но не будет проверять его реальное существование. Функция может принять и дополнительный, третий, аргумент, который должен быть переменной. Если эта переменная предоставлена, она будет заполнена строковым представлением переданного метода.

Далее я вызываю `is_callable()` с этим необязательным третьим аргументом, который затем вывожу:

```
// Листинг 5.50
if (is_callable([$product, $method], false, $callableName))
{
    print $callableName;
}
```

А вот какой вывод я получаю:

```
popp\ch05\batch05\CDProduct::getTitle
```

Такая функциональность может пригодиться при документировании или журналировании.

Функции `method_exists()` передаются ссылка на объект (или имя класса) и имя метода, как показано ниже. Она возвращает логическое значение `true`, если указанный метод существует в классе объекта:

```
// Листинг 5.51
if (method_exists($product, $method))
{
    print $product->$method(); // Вызов метода
}
```

На заметку Само существование метода еще не означает, что его можно вызвать. Функция `method_exists()` возвращает логическое значение `true` для обнаруживаемых закрытых (`private`), защищенных (`protected`) и открытых (`public`) методов.

Получение информации о свойствах

Подобно запрашиванию списка методов, можно запросить список свойств из класса. С этой целью функции `get_class_vars()` передается имя класса, а она возвращает ассоциативный массив. Имена свойств сохраняются в виде ключей этого массива, а содержимое полей — в виде значений. Произведем проверку объекта класса `CDProduct`. Для большей наглядности примера введем в этот класс открытое свойство: `CDProduct::$coverUrl`. В результате вызова

```
// Листинг 5.52
print_r(get_class_vars('\\popp\\ch05\\batch05\\CDProduct'));
```

будет выведено только открытое свойство:

```
Array
(
    [coverUrl] => cover url
)
```

Получение сведений о наследовании

Используя функции для обращения с классами, можно также выявлять отношения наследования. Так, с помощью функции `get_parent_class()` можно выяснить имя родительского класса для указанного класса. Этой функции передается ссылка на объект или имя класса, а она возвращает имя суперкласса, если таковой существует. Если же такого класса не существует, т.е. если у проверяемого класса нет родительского класса, то функция `get_parent_class()` возвращает логическое значение `false`. Например, в результате вызова

```
// Листинг 5.53
print get_parent_class('\\popp\\ch04\\batch02\\BookProduct');
```

будет получено имя родительского класса ShopProduct, как и следовало ожидать.

С помощью функции is_subclass_of() можно также проверить, является ли класс дочерним для другого класса. Этой функции передаются ссылка на дочерний объект и имя родительского класса, как показано ниже. Она возвращает логическое значение true, если класс, указанный в качестве второго аргумента, является суперклассом для класса, указанного в качестве первого аргумента:

```
// Листинг 5.54
$product = self::getBookProduct(); // Получение объекта

if (is_subclass_of($product,
                  '\\popp\\ch04\\batch02\\ShopProduct'))
{
    print "BookProduct является подклассом ShopProduct\n";
}
```

Функция is_subclass_of() сообщит сведения только об отношениях наследования в классе, но не поможет выяснить, реализован ли в этом классе некоторый интерфейс. Для этой цели следует применить оператор instanceof. Кроме того, можно воспользоваться функцией class_implements(), как показано ниже. Эта функция входит в состав библиотеки SPL (Standard PHP Library — стандартная библиотека PHP). Ей передается имя класса или ссылка на объект, а она возвращает массив имен интерфейсов:

```
// Листинг 5.55
if (in_array('someInterface', class_implements($product)))
{
    print "BookProduct реализует batqc someInterface\n";
}
```

Вызов методов

Выше уже рассматривался пример, в котором строковая переменная использовалась для динамического вызова метода:

```
// Листинг 5.56
$product = self::getProduct();
$method = "getTitle";      // Имя метода
print $product->$method(); // Вызов метода
```

Для этой цели в языке PHP предусмотрена также функция `call_user_func()`, которая может вызывать любые вызываемые сущности (такие, как имена функций или анонимные методы). Чтобы вызвать функцию, в качестве первого аргумента следует указать строку, содержащую имя этой функции:

```
$returnVal = call_user_func("myFunction");
```

Для вызова метода в качестве первого аргумента указывается массив. Первым элементом массива должен быть объект, а вторым — имя вызываемого метода:

```
$returnVal = call_user_func(array[ $myObj, "methodName"]);
```

Вызываемой функции или методу можно передать любое количество аргументов, указав их в качестве дополнительных аргументов функции `call_user_func()`:

```
// Листинг 5.57
$product = self::getBookProduct(); // Объект BookProduct
call_user_func([$product, 'setDiscount'], 20);
```

Этот динамический вызов эквивалентен следующему:

```
$product->setDiscount( 20 );
```

Функция `call_user_func()` не особенно облегчает жизнь, поскольку в равной степени вместо имени метода можно употребить строковую переменную, как показано ниже:

```
// Листинг 5.58
$method = "setDiscount";
$product->$method(20);
```

Намного большее впечатление производит родственная ей функция `call_user_func_array()`. Что касается выбора нужного метода или функции, то она действует аналогично функции `call_user_func()`. Но решающее значение приобретает то обстоятельство, что данной функции можно передать в виде массива любые аргументы, требующиеся для вызова указанного метода.

На заметку Имейте в виду, что аргументы, требующиеся для вызова указанного метода или функции, не передаются функции `call_user_func()` по ссылке.

В чем же здесь польза? Иногда аргументы приходится получать в виде массива. Если количество обрабатываемых аргументов заранее неизвестно, то передать их функции будет очень трудно. В главе 4, "Расширенные возможности", были рассмотрены методы-перехватчики, которые можно применять для создания делегирующих классов. Ниже приведен простой пример применения метода `__call()`:

```
// Листинг 5.59
public function __call(string $method, array $args): mixed
{
    if (method_exists($this->thirdpartyShop, $method))
    {
        return $this->thirdpartyShop->$method();
    }
}
```

Как пояснялось выше, метод `__call()` вызывается в том случае, когда в клиентском коде предпринимается попытка вызвать неопределенный метод. В данном примере мы используем объект, сохраненный в свойстве `$thirdpartyShop`. Если у этого объекта существует метод, имя которого указано в первом аргументе `$method`, то он вызывается. Мы легкомысленно предположили, что метод, который необходимо вызвать, не требует никаких аргументов. Но именно здесь начинаются затруднения! При создании метода `__call()` заранее нельзя знать, насколько крупным может оказаться массив `$args` при разных вызовах этого метода. Если мы передадим массив `$args` непосредственно делегирующему методу, то передадим единственный аргумент в виде массива, а не отдельные аргументы, как того, возможно, ожидает делегирующий метод. Поэтому функция `call_user_func_array()` идеально разрешает данное затруднение следующим образом:

```
// Листинг 5.60
public function __call(string $method, array $args): mixed
{
    if (method_exists($this->thirdpartyShop, $method))
    {
        return call_user_func_array(
            [
                $this->thirdpartyShop,
                $method
            ],
            $args
        );
    }
}
```

Reflection API

Программный интерфейс Reflection API для PHP — это то же самое, что и пакет `java.lang.reflect` для Java. Он состоит из встроенных классов для анализа свойств, методов и классов. В какой-то степени он напоминает такие рассмотренные выше функции манипулирования объектами, как, например, функция `get_class_vars()`, но в то же время он предоставляет больше удобств и сведений об анализируемых элементах прикладного кода. Он предназначен также для более эффективной работы с такими объектно-ориентированными средствами PHP, как управление доступом, интерфейсы и абстрактные классы, по сравнению с ограниченными функциональными возможностями прежних классов.

Краткое введение в Reflection API

Интерфейс Reflection API можно использовать для исследования не только классов. Например, класс `ReflectionFunction` предоставляет сведения об указанной функции, а класс `ReflectionExtension` — сведения о скомпилированных расширениях языка PHP. В табл. 5.1 перечислены некоторые классы интерфейса Reflection API.

Некоторые классы интерфейса Reflection API позволяют во время выполнения программы получить беспрецедентную информацию об объектах, функциях и расширениях языка, содержащихся в сценарии, добыть которую раньше было невозможно.

Из-за более широких возможностей и большей эффективности во многих случаях следует использовать интерфейс Reflection API, а не функции для работы с классами и объектами. Вскоре вы поймете, что это незаменимое инструментальное средство для исследования классов. Например, с его помощью можно создавать диаграммы классов или документацию, сохранять сведения об объекте в базе данных и исследовать методы доступа (установки и получения), применяемые для извлечения значений полей (или свойств) объекта. Создание каркаса, в котором вызываются методы из классов модуля в соответствии с принятой схемой именования, служит еще одним примером применения интерфейса Reflection API.

Таблица 5.1. Некоторые классы интерфейса Reflection API

Класс	Описание
`Reflection`	Содержит статический метод `export()`, предоставляющий итоговые сведения о классе
`ReflectionAttribute`	Контекстная информация о классах, свойствах, константах и параметрах
`ReflectionClass`	Информация о классе и инструментальные средства
`ReflectionClassConstant`	Информация о константах
`ReflectionException`	Класс ошибки
`ReflectionExtension`	Информация о расширениях PHP
`ReflectionFunction`	Информация о функции и инструментальные средства
`ReflectionGenerator`	Информация о генераторе
`ReflectionMethod`	Информация о методе класса и инструментальные средства
`ReflectionNamedType`	Информация о возвращаемом типе функции или метода (возвращаемое объединение описывается с помощью `ReflectionUnionType`)
`ReflectionObject`	Информация об объекте и инструментальные средства
`ReflectionParameter`	Информация об аргументах метода
`ReflectionProperty`	Информация о свойствах класса
`ReflectionType`	Информация о возвращаемом типе функции или метода
`ReflectionUnionType`	Набор объектов `ReflectionType` для объявления типа объединения
`ReflectionZendExtension`	Сведения о расширении Zend языка PHP

Время засучить рукава

В приведенных выше примерах уже демонстрировались некоторые функции для исследования атрибутов классов. И хотя они полезны, обычно они ограничены. А теперь рассмотрим инструментальное средство, специально предназначенное для выполнения такой работы. Им служит класс ReflectionClass, предоставляющий методы для сбора сведений обо всех аспектах указанного класса: как внутреннего, так и определяемого пользователем. В качестве единственного аргумента конструктору класса ReflectionClass передается имя исследуемого класса, интерфейса (или экземпляра объекта), как показано ниже:

```
// Листинг 5.61
$prodclass = new \ReflectionClass(CDProduct::class);
print $prodclass;
```

Создав объект ReflectionClass, вы можете немедленно получить всевозможную информацию о классе, просто обращаясь к нему в строковом контексте. Вот фрагмент вывода, полученного при печати экземпляра ReflectionClass для ShopProduct:

```
Class [ <user> class popp\ch04\batch02\CDProduct extends
     popp\ch04\batch02\ShopProduct ] {
    @@ /var/popp/src/ch04/batch02/CDProduct.php 6-37

  - Constants [2] {
    Constant [ public int AVAILABLE ] { 0 }
    Constant [ public int OUT_OF_STOCK ] { 1 }
}

  - Static properties [0] {
}

  - Static methods [1] {
    Method [ <user, inherits popp\ch04\batch02\ShopProduct>
        static public method getInstance ] {
      @@ /var/popp/src/ch04/batch02/ShopProduct.php 93 - 130

      - Parameters [2] {
        Parameter #0 [ <required> int $id ]
        Parameter #1 [ <required> PDO $pdo ]
    }
    - Return [ popp\ch04\batch02\ShopProduct ]
  }
}
```

```
- Properties [3] {
    Property [ private $playLength = 0 ]
    Property [ public $status = NULL ]
    Property [ protected int|float $price ]
}
...
```

На заметку Вспомогательный метод `Reflection::export()` был когда-то стандартным способом получения информации от `ReflectionClass`. Он объявлен устаревшим в PHP 7.4 и полностью удален в PHP 8.0.

Как видите, `ReflectionClass` обеспечивает удобный доступ к информации об исследуемом классе. Строковый вывод предоставляет итоговые сведения практически о каждом аспекте класса `CDProduct`, включая состояние управления доступом к свойствам, методам и аргументам, требующимся для каждого метода, а также местоположение каждого метода в сценарии. Сравните это с более традиционной функцией отладки. Так, функция `var_dump()` служит универсальным инструментальным средством для вывода итоговых данных. Вы должны инстанцировать объект перед выводом информации о нем с помощью функции `var_dump()`, но даже тогда вы не увидите того беспрецедентного объема информации, который можно получить с помощью `ReflectionClass`:

```
// Листинг 5.62
$cd = new CDProduct("cd1", "bob", "bobbleson", 4, 50);
var_dump($cd);
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
object(popp\ch04\batch02\CDProduct)#15 (8) {
  ["playLength":"popp\ch04\batch02\CDProduct":private]=>
  int(50)
  ["status"]=>
  NULL
  ["title":"popp\ch04\batch02\ShopProduct":private]=>
  string(3) "cd1"
  ["producerMainName":"popp\ch04\batch02\ShopProduct":private]=>
  string(9) "Вивальди"
  ["producerFirstName":"popp\ch04\batch02\ShopProduct":private]=>
  string(3) "Антонио"
  ["price":protected]=>
  float(4)
```

```
  ["discount":"popp\ch04\batch02\ShopProduct":private]=>
  int(0)
  ["id":"popp\ch04\batch02\ShopProduct":private]=>
  int(0)
}
```

Функция `var_dump()` и родственная ей функция `print_r()` служат невероятно удобными инструментальными средствами для отображения данных в сценариях. Но для исследования классов и функций интерфейс Reflection API позволяет выйти на совершенно новый уровень отладки.

Исследование класса

`ReflectionClass` предоставляет немало полезных сведений для отладки прикладного кода, но API можно использовать и более специализированным образом. Давайте поработаем непосредственно с классами `Reflection`.

Как было показано выше, создать экземпляр объекта `ReflectionClass` можно следующим образом:

```
// Листинг 5.63
$prodclass = new \ReflectionClass(CDProduct::class);
```

Теперь я использую объект `ReflectionClass`, чтобы исследовать класс `CDProduct` в процессе выполнения сценария. В данном случае необходимо выяснить, к какому именно типу класса он относится и можно ли создать его экземпляр. Ниже приведена функция, которая поможет найти ответы на эти вопросы:

```
// Листинг 5.64
// class ClassInfo
public static function getData(\ReflectionClass $class): string
{
    $details = "";
    $name = $class->getName();
    $details .= ($class->isUserDefined()) ?
    "$name - определен пользователем\n" : "" ;
    $details .= ($class->isInternal()) ?
    "$name - встроенный класс\n"        : "" ;
    $details .= ($class->isInterface())?
    "$name - интерфейс\n"               : "" ;
    $details .= ($class->isAbstract()) ?
    "$name - абстрактный класс\n"       : "" ;
    $details .= ($class->isFinal())    ?
    "$name - завершенный класс\n"       : "" ;
```

```
    $details .= ($class->isInstantiable()) ?
    "$name может быть инстанцирован\n"  :
    "$name не может быть инстанцирован\n" ;
    $details .= ($class->isCloneable())?
    "$name может быть клонирован\n"     :
    "$name не может быть клонирован\n" ;
    return $details;
}

// Листинг 5.65
$prodclass = new \ReflectionClass(CDProduct::class);
print ClassInfo::getData($prodclass);
```

Я создаю объект `ReflectionClass` путем передачи конструктору класса `CDProduct::class`, и присваиваю его переменной `$prodclass`. Затем эта переменная передается методу `ClassInfo::classData()`, который демонстрирует некоторые из методов, которые можно использовать для запроса класса.

Суть методов должна быть понятна из названий; тем не менее вот краткое описание некоторых из них.

- Метод `ReflectionClass::getName()` возвращает имя исследуемого класса.
- Метод `ReflectionClass::isUserDefined()` возвращает логическое значение `true`, если класс был объявлен в коде PHP, а метод `ReflectionClass::isInternal()` — если класс является встроенным.
- С помощью метода `ReflectionClass::isAbstract()` можно проверить, является ли класс абстрактным, а с помощью метода `ReflectionClass::isInterface` — является ли исследуемый элемент кода интерфейсом.
- Если требуется получить экземпляр класса, то с помощью метода `ReflectionClass::isInstantiable()` можно проверить, насколько это осуществимо.
- С помощью метода `ReflectionClass::isCloneable()` можно выяснить, является ли класс клонируемым.
- Можно даже исследовать исходный код класса, определенного пользователем. Объект `ReflectionClass` обеспечивает доступ к имени файла класса и номера первой и последней строк класса в файле.

Ниже приведен простой пример, в котором объект типа ReflectionClass используется для доступа к исходному коду указанного класса:

```
// Листинг 5.66
class ReflectionUtil
{
    public static function getClassSource(\ReflectionClass $class):
        string
    {
        $path = $class->getFileName();
        $lines = @file($path);
        $from = $class->getStartLine();
        $to = $class->getEndLine();
        $len = $to - $from + 1;
        return implode(array_slice($lines, $from - 1, $len));
    }
}

// Листинг 5.67
print ReflectionUtil::getClassSource(
    new \ReflectionClass(CDProduct::class)
);
```

Как видите, класс ReflectionUtil очень прост и содержит единственный статический метод ReflectionUtil::getClassSource(). В качестве единственного аргумента этому методу передается объект типа ReflectionClass, а он возвращает исходный код исследуемого класса. Метод ReflectionClass::getFileName() возвращает абсолютное имя файла класса, поэтому код из данного примера должен без проблем открыть указанный исходный файл. Функция file() возвращает массив всех строк кода в исходном файле, метод ReflectionClass::getStartLine() — номер начальной строки кода, а метод ReflectionClass::getEndLine() — номер последней строки, в которой содержится определение класса. Теперь для получения нужных строк кода достаточно воспользоваться функцией array_slice().

Ради краткости в данном примере кода опущена обработка ошибок (путем добавления символа @ перед вызовом file()). Но в реальном приложении нужно непременно проверять аргументы и возвращаемые коды состояния.

Исследование методов

Подобно тому как ReflectionClass служит для исследования класса, класс ReflectionMethod служит для исследования метода.

Вы можете получить массив объектов ReflectionMethod из ReflectionClass::getMethods(). В качестве альтернативного способа, если интерес представляет конкретный метод, его имя следует передать методу ReflectionClass::getMethod(), который возвратит соответствующий объект типа ReflectionMethod.

Можно также инстанцировать ReflectionMethod непосредственно, передавая либо строку с именем класса и метода, либо объект и имя метода.

Вот как могут выглядеть в коде описанные выше варианты:

```
// Листинг 5.68
$cd = new CDProduct("cd1", "Антонио", "Вивальди", 4, 50);
$classname = CDProduct::class;
$rmethod1 = new \ReflectionMethod("{$classname}::construct");
// Строка класс/метод
$rmethod2 = new \ReflectionMethod($classname, "construct");
// Имя класса и имя метода
$rmethod3 = new \ReflectionMethod($cd, "construct");
// Объект и имя метода
```

Далее мы используем ReflectionClass::getMethods():

```
// Листинг 5.69
$prodclass = new \ReflectionClass(CDProduct::class);
$methods = $prodclass->getMethods();

foreach ($methods as $method)
{
    print ClassInfo::methodData($method);
    print "\n----\n";
}

// Листинг 5.70
// class ClassInfo
public static function methodData(\ReflectionMethod $method):
    string
{
    $details = "";
    $name = $method->getName();
    $details .= ($method->isUserDefined()) ?
    "$name - определен пользователем\n"     : "" ;
```

```
    $details .= ($method->isInternal())    ?
    "$name - встроенный метод\n"             : "" ;
    $details .= ($method->isAbstract())    ?
    "$name - абстрактный метод\n"            : "" ;
    $details .= ($method->isPublic())      ?
    "$name - открытый метод\n"               : "" ;
    $details .= ($method->isProtected())   ?
    "$name - защищенный метод\n"             : "" ;
    $details .= ($method->isPrivate())     ?
    "$name - закрытый метод\n"               : "" ;
    $details .= ($method->isStatic())      ?
    "$name - статический метод\n"            : "" ;
    $details .= ($method->isFinal())       ?
    "$name - завершенный метод\n"            : "" ;
    $details .= ($method->isConstructor()) ?
    "$name - конструктор\n"                  : "" ;
    $details .= ($method->returnsReference()) ?
    "$name - возвращает ссылку (а не значение)\n" : "" ;
    return $details;
}
```

В приведенном коде для получения массива объектов типа ReflectionMethod вызывается метод ReflectionClass::getMethods(). Затем в цикле выполняется передача каждого элемента массива методу methodData().

Имена методов, вызываемых в функции methodData(), отражают их назначение: в коде проверяется, является ли метод определяемым пользователем, встроенным, абстрактным, открытым, защищенным, статическим или завершенным. Можно также проверить, является ли метод конструктором для своего класса и что он возвращает: ссылку или значение.

Но здесь необходимо сделать следующую оговорку: метод ReflectionMethod::returnsReference() не возвращает логическое значение true в случае, если проверяемый метод просто возвращает объект, хотя в версии PHP 5 объекты передаются и присваиваются по ссылке. Этот метод возвращает логическое значение true только в том случае, если в исследуемом методе было явно объявлено, что он возвращает ссылку. С этой целью перед именем метода в его объявлении должен быть указан знак амперсанда.

Как и следовало ожидать, доступ к исходному коду метода можно получить тем же способом, который использовался ранее для доступа к исследуемому классу с помощью класса ReflectionClass:

```
// Листинг 5.71
// class ReflectionUtil
public static function getMethodSource(\ReflectionMethod $method):
    string
{
    $path = $method->getFileName();
    $lines = @file($path);
    $from = $method->getStartLine();
    $to = $method->getEndLine();
    $len = $to - $from + 1;
    return implode(array_slice($lines, $from - 1, $len));
}

// Листинг 5.72
$class = new \ReflectionClass(CDProduct::class);
$method = $class->getMethod('getSummaryLine');
print ReflectionUtil::getMethodSource($method);
```

В классе ReflectionMethod имеются методы getFileName(), getStartLine() и getEndLine(), что позволяет очень просто получить исходный код исследуемого метода.

Исследование аргументов методов

Теперь, когда стало возможным ограничивать типы аргументов с помощью сигнатур методов, чрезвычайно полезной кажется возможность исследования аргументов, объявленных в сигнатуре метода. Именно для этой цели в интерфейсе Reflection API предусмотрен класс ReflectionParameter. Чтобы получить объект типа ReflectionParameter, потребуется помощь объекта типа ReflectionMethod. В частности, метод ReflectionMethod::getParameters() возвращает массив объектов типа ReflectionParameter.

Вы также можете инстанцировать объект ReflectionParameter непосредственно обычным способом. Конструктор ReflectionParameter требует вызываемого аргумента и либо целого числа, представляющего номер параметра (начиная с нуля), либо строки, представляющей имя аргумента.

Все четыре варианта эквивалентны. Каждый устанавливает объект ReflectionParameter в качестве второго аргумента конструктора класса CDProduct:

```
// Листинг 5.73
$classname = CDProduct::class;
$rparam1 = new \ReflectionParameter([$classname, "__construct"], 1);
```

```
$rparam2 = new \ReflectionParameter([$classname, "__construct"],
                                     "firstName");
$cd = new CDProduct("cd1", "Антонио", "Вивальди", 4, 50);
$rparam3 = new \ReflectionParameter([$cd, "__construct"], 1);
$rparam4 = new \ReflectionParameter([$cd, "__construct"],
                                     "firstName");
```

ReflectionParameter может указать имя аргумента и передан ли он по ссылке (с предшествующим символом & в объявлении метода). Он также может подсказать вам требуемый в качестве аргумента класс и принимает ли метод нулевое значение в качестве аргумента.

Вот некоторые из методов ReflectionParameter в действии:

```
// Листинг 5.74
$class = new \ReflectionClass(CDProduct::class);
$method = $class->getMethod("__construct");
$params = $method->getParameters();

foreach ($params as $param)
{
    print ClassInfo::argData($param) . "\n";
}

// Листинг 5.75
// class ClassInfo
public static function argData(\ReflectionParameter $arg): string
{
    $details = "";
    $declaringclass = $arg->getDeclaringClass();
    $name = $arg->getName();
    $position = $arg->getPosition();
    $details .= "\$$name имеет позицию $position\n";

    if ($arg->hasType())
    {
        $type = $arg->getType();
        $typenames = [];

        if ($type instanceof \ReflectionUnionType)
        {
            $types = $type->getTypes();

            foreach ($types as $utype)
            {
                $typenames[] = $utype->getName();
            }
        }
```

```
        else
        {
            $typenames[] = $type->getName();
        }

        $typename = implode("|", $typenames);
        $details .= "\$$name ljk;ty иметь тип {$typename}\n";
    }

    if ($arg->isPassedByReference())
    {
        $details .= "\${$name} передается по ссылке\n";
    }

    if ($arg->isDefaultValueAvailable())
    {
        $def = $arg->getDefaultValue();
        $details .= "\${$name} имеет значение по умолчанию: $def\n";
    }

    if ($arg->allowsNull())
    {
        $details .= "\${$name} может быть null\n";
    }

    return $details;
}
```

Используя `ReflectionClass::getMethod()`, код получает объект `ReflectionMethod`. Затем вызывается метод `ReflectionClass::getParameters()` для получения массива объектов типа `ReflectionParameter`, соответствующих данному методу. Далее в цикле функции `argData()` передается объект типа `ReflectionParameter`, а она возвращает информацию об аргументе.

Сначала в данном примере выясняется имя переменной аргумента с помощью метода `ReflectionParameter::getName()`. Метод `ReflectionParameter::getType()` возвращает объект типа `ReflectionType`, если тип определен в сигнатуре метода, или `ReflectionUnionType`, если этот тип является типом объединения. Что бы ни было возвращено, создается строковое представление требуемого типа. Далее с помощью метода `isPassedByReference()` проверяется, является ли аргумент ссылкой. И, наконец, с помощью метода `isDefaultValueAvailable()` проверяется, имеет ли данный аргумент значение по умолчанию.

Использование интерфейса Reflection API

Зная основы интерфейса Reflection API, теперь можно воспользоваться им на практике.

Допустим, требуется создать класс, динамически вызывающий объекты типа `Module`. Это означает, что данный класс должен уметь работать с модулями, написанными сторонними разработчиками, которые можно автоматически подключать к приложению, не занимаясь трудоемким кодированием. Чтобы добиться этого, можно определить метод `execute()` в интерфейсе `Module` или абстрактном базовом классе, вынуждая разработчиков определять его реализацию во всех дочерних классах. Пользователям системы можно разрешить указывать классы типа `Module` во внешнем XML-файле конфигурации. Эти сведения могут использоваться в системе для группирования ряда объектов типа `Module`, перед тем как вызывать метод `execute()` для каждого из них.

Но что произойдет, если каждому объекту типа `Module` для выполнения его обязанностей потребуются *свои* (отличные от других) данные? На этот случай в XML-файле конфигурации могут быть предоставлены ключи и значения свойств для каждого объекта типа `Module`, а создатель объекта типа `Module` должен предоставить методы установки для каждого имени свойства. И на этом основании в прикладном коде можно обеспечить вызов нужного метода установки по имени соответствующего свойства.

Ниже приведено объявление интерфейса `Module` и пара классов, в которых он реализован:

```
// Листинг 5.76
class Person
{
    public $name;
    public function __construct(string $name)
    {
        $this->name = $name;
    }
}

// Листинг 5.77
interface Module
{
    public function execute(): void;
}
```

```
// Листинг 5.78
class FtpModule implements Module
{
    public function setHost(string $host): void
    {
        print "FtpModule::setHost(): $host\n";
    }
    public function setUser(string | int $user): void
    {
        print "FtpModule::setUser(): $user\n";
    }
    public function execute(): void
    {
        // Некоторые действия
    }
}

// Листинг 5.79
class PersonModule implements Module
{
    public function setPerson(Person $person): void
    {
        print "PersonModule::setPerson(): {$person->name}\n";
    }
    public function execute(): void
    {
        // Некоторые действия
    }
}
```

В классах PersonModule и FtpModule из приведенного выше кода предусмотрены пустые реализации метода execute(). В каждом классе также реализованы методы установки значений, которые ничего не делают, но сообщают, что они были вызваны. В рассматриваемой здесь системе принято соглашение, что всем методам установки должен передаваться только один аргумент: символьная строка или объект, экземпляр которого можно создать с помощью одного строкового аргумента. Методу PersonModule::setPerson() должен передаваться объект типа Person, поэтому в данный пример был включен несложный класс Person.

Следующей стадией в работе с классами PersonModule и FtpModule является создание класса ModuleRunner. В нем используется многомерный массив, проиндексированный по имени модуля и содержащий сведения о параметрах конфигурации, полученные из XML-файла. Ниже приведено определение класса ModuleRunner:

```
// Листинг 5.80
class ModuleRunner
{
    private array $configData = [
                    PersonModule::class => ['person' => 'bob'],
                    FtpModule::class => [
                        'host' => 'example.com',
                        'user' => 'anon'
                    ]
                ];
    private array $modules = [];
    // ...
}
```

Свойство `ModuleRunner::$configData` содержит массив параметров для двух классов типа `Module`. В каждом элементе этого массива содержится вложенный массив, содержащий набор свойств для каждого модуля. Метод `init()` из класса `ModuleRunner` отвечает за создание соответствующих объектов типа `Module`, как показано в следующем фрагменте кода:

```
// Листинг 5.81
// class ModuleRunner
public function init(): void
{
    $interface = new \ReflectionClass(Module::class);
    foreach ($this->configData as $modulename => $params)
    {
        $module_class = new \ReflectionClass($modulename);

        if (! $module_class->isSubclassOf($interface))
        {
            throw new Exception(
                    "Неизвестный тип модуля: $modulename");
        }

        $module = $module_class->newInstance();

        foreach ($module_class->getMethods() as $method)
        {
            $this->handleMethod($module, $method, $params);
            // Метод handleMethod() будет в будущем листинге!
        }

        array_push($this->modules, $module);
    }
}
```

```
// Листинг 5.82
$test = new ModuleRunner();
$test->init();
```

В методе `init()` в цикле перебираются все элементы массива `ModuleRunner::$configData` и для каждого указанного в нем имени класса типа `Module` предпринимается попытка создать объект типа `ReflectionClass`. Когда конструктору класса `ReflectionClass` передается имя несуществующего класса типа `Module`, генерируется исключение. Поэтому в реальное приложение необходимо включить также код обработки ошибок. Чтобы проверить, относится ли класс модуля к типу `Module`, вызывается метод `ReflectionClass::isSubclassOf()`.

Прежде чем вызвать метод `execute()` для каждого объекта типа `Module`, необходимо создать экземпляр этого объекта. Для этой цели служит метод `ReflectionClass::newInstance()`. Ему можно передать произвольное количество аргументов, которые будут далее переданы конструктору соответствующего класса. Если все пройдет удачно, метод возвратит экземпляр исследуемого класса. В реальном приложении перед получением экземпляра объекта типа `Module` следует непременно убедиться, что методу конструктора каждого объекта типа `Module` не требуется передавать никаких аргументов.

Метод `ReflectionClass::getMethods()` возвращает массив всех объектов типа `ReflectionMethod`, имеющихся для исследуемого класса. Для каждого элемента массива в рассматриваемом здесь примере кода вызывается метод `ModuleRunner::handleMethod()`, которому передаются экземпляр объекта типа `Module`, объект типа `ReflectionMethod` и массив свойств для связывания с текущим объектом типа `Module`. Переданные данные проверяются в методе `handleMethod()`, после чего вызываются методы установки текущего объекта типа `Module`, как показано ниже:

```
// Листинг 5.83
// class ModuleRunner
public function handleMethod(Module $module,
                             \ReflectionMethod $method,
                             array $params): bool
{
    $name = $method->getName();
    $args = $method->getParameters();

    if (count($args) != 1 || substr($name, 0, 3) != "set")
    {
        return false;
    }
```

```
    $property = strtolower(substr($name, 3));

    if (! isset($params[$property]))
    {
        return false;
    }

    if (! $args[0]->hasType())
    {
        $method->invoke($module, $params[$property]);
        return true;
    }

    $arg_type = $args[0]->getType();

    if (!($arg_type instanceof \ReflectionUnionType) && class_exists(
            $arg_type->getName()))
    {
        $method->invoke(
          $module,
          (new \ReflectionClass($arg_type->getName()))->newInstance(
              $params[$property])
        );
    }
    else
    {
        $method->invoke($module, $params[$property]);
    }

    return true;
}
```

В методе `handleMethod()` сначала проверяется, является ли исследуемый метод допустимым методом установки. В данном примере кода допустимый метод установки должен называться `set`*`XXXX`*`()` и содержать только один аргумент.

В предположении, что аргумент проверен, код извлекает корректное имя из имени метода. Для этого в начале имени метода удаляется слово `"set"` и полученная в итоге подстрока преобразуется в строчные буквы. Эта строка служит для проверки элемента массива `$params`. Напомним, что в этом массиве содержатся предоставляемые пользователем значения свойств, связанных с объектом типа `Module`. Если требуемое свойство не обнаружено в массиве `$params`, то возвращается логическое значение `false`.

Если имя свойства, извлеченное из имени метода, содержащегося в объекте типа Module, соответствует элементу массива $params, то для него можно вызвать соответствующий метод установки. Но прежде необходимо проверить тип первого (и только первого) аргумента метода установки. Если параметр имеет объявление типа (ReflectionParameter::hasType()) и указанный тип разрешается в класс, то мы знаем, что метод ожидает передачи объекта. В противном случае мы предполагаем, что ему следует передать значение примитивного типа.

Для вызова метода установки потребуется новый метод ReflectionMethod::invoke() из интерфейса Reflection API. Этому методу передаются объект (в данном случае — типа Module) и произвольное количество аргументов, которые будут переданы далее методу установки объекта типа Module. Метод ReflectionMethod::invoke() генерирует исключение, если переданный ему объект не соответствует методу, определенному в объекте типа ReflectionMethod. Метод ReflectionMethod::invoke() можно вызвать одним из двух способов. Так, если методу установки требуется передать значение элементарного типа, то метод ReflectionMethod::invoke() вызывается с переданным ему строковым значением свойства, предоставляемым пользователем. А если методу установки требуется объект (что можно проверить с использованием class_exists с именем типа), то строковое значение свойства служит для получения экземпляра объекта требуемого типа, который затем передается методу установки.

В данном примере кода предполагается, что для получения экземпляра требуемого объекта его конструктору требуется передать только один строковый аргумент. Но это лучше все же проверить, прежде чем вызывать метод ReflectionClass::newInstance().

После того как метод ModuleRunner::init() завершит свою работу, объект типа ModuleRunner будет содержать ряд объектов типа Module (в массиве свойств $modules), причем все они будут заполнены данными. Остается лишь создать в классе ModuleRunner специальный метод, перебирающий в цикле все объекты типа Module и вызывающий для каждого из них метод execute().

Атрибуты

Многие языки предоставляют механизм, с помощью которого специальные дескрипторы (теги) в исходных файлах могут быть доступны для кода. Такие дескрипторы часто известны как *аннотации*. Несмотря на то что еще до PHP 8 в пакетах PHP имелись некоторые пользовательские реализации таких механизмов, на уровне языка никакой поддержки этой функциональности не было. Все изменилось с внедрением *атрибутов*.

По сути, атрибут — это специальный дескриптор, который позволяет добавить дополнительную информацию в класс, метод, свойство, параметр или константу. Эта информация становится доступной для системы через рефлексию.

Так для чего же могут использоваться аннотации? Как правило, метод может обеспечить дополнительную информацию о том, что он собирается использовать. Например, клиентский код может просканировать класс для обнаружения методов, которые обязательно должны работать. Я упомяну о других случаях использования, когда мы к ним подойдем.

Давайте объявим и получим доступ к аннотации:

```
// Листинг 5.84
namespace popp\ch05\batch09;

#[info]
class Person
{
}
```

Аннотация объявляется с использованием строкового токена, заключенного между `#[` и `]`. В данном случае это `#[info]`. Во многих примерах кода я исключаю объявление пространства имен, потому что код будет работать одинаково хорошо в объявленном пространстве имен или в `main`. В данном случае стоит упомянуть пространство имен. Я еще вернусь к этому моменту.

Теперь обратимся к аннотации:

```
// Листинг 5.85
$rpers = new \ReflectionClass(Person::class);
$attrs = $rpers->getAttributes();

foreach ($attrs as $attr)
{
    print $attr->getName() . "\n";
}
```

Я инстанцировал объект ReflectionClass, чтобы иметь возможность исследовать класс Person. Затем я вызываю метод getAttributes(), который возвращает массив объектов ReflectionAttribute. Вызов ReflectionAttribute::getName() возвращает имя атрибута, который я объявил.

Вот вывод этого кода:

```
popp\ch05\batch09\info
```

В моем выводе аннотация включает пространство имен. Часть имени popp\ch05\batch09 неявная. Я могу ссылаться на аннотацию в соответствии с теми же правилами и псевдонимами, что и при обращении к классам. Так, объявление #[info] в пространстве имен popp\ch05\batch09 эквивалентно объявлению #[popp\ch05\batch09\info] в другом месте.

Аннотации могут быть применимы к различным аспектам PHP. В табл. 5.2 перечислены сущности, которые могут быть аннотированы, а также соответствующие классы Reflection API.

Таблица 5.2. *Применение аннотаций PHP*

Сущность	Метод получения
Класс	ReflectionClass::getAttributes()
Свойство	ReflectionProperty::getAttributes()
Функция/метод	ReflectionFunction::getAttributes()
Константа	ReflectionConstant::getAttributes()

Вот пример применения атрибута к методу:

```
// Листинг 5.86
#[moreinfo]
public function setName(string $name): void
{
    $this->name = $name;
}
```

Теперь обратимся к нему. Этот процесс должен быть для вас знаком:

```
// Листинг 5.87
$rpers = new \ReflectionClass(Person::class);
$rmeth = $rpers->getMethod("setName");
$attrs = $rmeth->getAttributes();
```

```
foreach ($attrs as $attr)
{
    print $attr->getName() . "\n";
}
```

Вывод данного фрагмента кода также должен быть знаком — это полный путь пространства имен к `moreinfo`:

```
popp\ch05\batch09\moreinfo
```

Тот код, который приведен выше, уже представляет собой определенное использование атрибутов. Мы можем включать атрибут как флаг некоторого рода. Например, атрибут `Debug` может быть связан с методами, которые могут вызываться только в процессе разработки программы. Но атрибуты нужны не только для этого. Мы можем определять типы и предоставить дополнительную информацию об аргументах. Атрибуты открывают нам новые возможности. Например, в системе событий атрибут может сигнализировать о том, что класс или метод должен быть связан с определенным событием.

В следующем примере я определяю атрибут, который включает два аргумента:

```
// Листинг 5.88
#[ApiInfo("Идентификатор компании из 3 цифр",
          "Дескриптор отдела из 5 символов")]
public function setInfo(int $companyid, string $department): void
{
    $this->companyid = $companyid;
    $this->department = $department;
}
```

Получив объект `ReflectionAttribute`, я могу обращаться к аргументам с использованием метода `getArguments()`:

```
// Листинг 5.89
$rpers = new \ReflectionClass(Person::class);
$rmeth = $rpers->getMethod("setInfo");
$attrs = $rmeth->getAttributes();

foreach ($attrs as $attr)
{
    print $attr->getName() . "\n";

    foreach ($attr->getArguments() as $arg)
    {
```

```
        print " - $arg\n";
    }
}
```

Вот как выглядит вывод этого фрагмента кода:

```
popp\ch05\batch09\ApiInfo
 - Идентификатор компании из 3 цифр
 - Дескриптор отдела из 5 символов
```

Как я упоминал, можно явно отображать атрибут на класс. Вот простой класс `ApiInfo`:

```
// Листинг 5.90
namespace popp\ch05\batch09;

use Attribute;
#[Attribute]
class ApiInfo
{
    public function __construct(public string $compinfo,
                                public string $depinfo)
    {
    }
}
```

Для того чтобы правильно связать атрибут и свой класс, я должен не забыть добавить `use Attribute` и применить встроенный `#[Attribute]` к классу.

В момент инстанцирования любые аргументы связанного атрибута автоматически передаются соответствующему конструктору класса. В этом случае я просто назначаю данные соответствующим свойствам. В реальном приложении я бы, вероятно, выполнил некоторую дополнительную обработку или предоставил связанную функциональность в объявлении класса.

Важно понимать, что класс атрибута автоматически не вызывается. Мы должны делать это посредством `ReflectionAttribute::newInstance()`. Далее я адаптирую клиентский код для работы с новым классом:

```
// Листинг 5.91
$rpers = new \ReflectionClass(Person::class);
$rmeth = $rpers->getMethod("setInfo");
$attrs = $rmeth->getAttributes();
```

```
foreach ($attrs as $attr)
{
    print $attr->getName() . "\n";
    $attrobj = $attr->newInstance();
    print " - " . $attrobj->compinfo . "\n";
    print " - " . $attrobj->depinfo . "\n";
}
```

Хотя я получаю доступ к данным атрибута через объект `ApiInfo`, результат остается тем же. Я вызываю `ReflectionAttribute::newInstance()`, а затем обращаюсь к заполненным свойствам.

Но подождите — в этом последнем примере есть серьезный и потенциально фатальный изъян. К методу может быть добавлено несколько атрибутов. Поэтому мы не можем быть уверены, что каждый атрибут, назначенный методу `setInfo()`, представляет собой экземпляр `ApiInfo`. Обращения к свойствам `ApiInfo::$compinfo` и `ApiInfo::$depinfo` неизбежно завершатся ошибкой для любого атрибута, который не имеет тип `ApiInfo`.

К счастью, мы можем применить фильтр к `getAttributes()`:

```
// Листинг 5.92
$rpers = new \ReflectionClass(Person::class);
$rmeth = $rpers->getMethod("setInfo");
$attrs = $rmeth->getAttributes(ApiInfo::class);
```

Теперь будут возвращены только точные совпадения для `ApiInfo::class` — остальная часть кода безопасна. Можно сделать еще шаг в этом направлении:

```
// Листинг 5.93
$rpers = new \ReflectionClass(Person::class);
$rmeth = $rpers->getMethod("setInfo");
$attrs = $rmeth->getAttributes(ApiInfo::class,
                 \ReflectionAttribute::IS_INSTANCEOF);
```

Путем передачи второго параметра `ReflectionAttribute::IS_INSTANCEOF` в `ReflectionAttribute::getAttributes()` я ослабляю фильтр так, чтобы он соответствовал указанному классу, а также любым расширяющим или реализующим дочерним классам или интерфейсам.

В табл. 5.3 перечислены методы `ReflectionAttribute`, с которыми мы встречались.

Таблица 5.3. Некоторые методы `ReflectionAttribute`

Метод	Описание
`getName()`	Возвращает полностью квалифицированный пространствами имен тип атрибута
`getArguments()`	Возвращает массив всех аргументов, связанных с указанным атрибутом
`newInstance()`	Инстанцирует и возвращает экземпляр класса атрибута, передавая аргументы конструктору

На заметку В главе 9, "Генерация объектов", вы встретитесь с гораздо более сложным использованием атрибутов.

Резюме

В этой главе были рассмотрены некоторые способы и средства для управления библиотеками и классами. В ней были описаны пространства имен как языковое средство PHP и было показано, что их можно успешно применять для гибкой организации классов и пакетов наряду с путями включения файлов, автозагрузкой и каталогами файловой системы.

Кроме того, в этой главе были рассмотрены функции для работы с объектами и классами в PHP, а на более высоком уровне — эффективный и удобный интерфейс Reflection API. Классы Reflection API были использованы для простого примера, демонстрирующего одно из возможных применений этого интерфейса. Наконец мы объединили классы `Reflection` с атрибутами, что является одной из важных новых функциональных возможностей PHP 8.

ГЛАВА 6

Объекты и проектирование

Подробно рассмотрев механизм поддержки объектов в PHP, отвлечемся от подробностей и выясним, почему лучше всего пользоваться теми средствами, с которыми вы уже познакомились. В этой главе мы остановимся на некоторых вопросах, касающихся объектов и проектирования. В ней будет также рассмотрен UML — эффективный графический язык для описания объектно-ориентированных систем.

В этой главе рассматриваются следующие вопросы.

- *Основы проектирования.* Что понимается под проектированием и чем объектно-ориентированная структура кода отличается от процедурной.
- *Область видимости класса.* Как решить, что следует включить в класс.
- *Инкапсуляция.* Сокрытие реализации и данных в интерфейсе класса.
- *Полиморфизм.* Использование общего супертипа с целью разрешить прозрачную подстановку специализированных подтипов во время выполнения программы.
- *UML.* Использование диаграмм для описания объектно-ориентированных архитектур.

Определение программного проекта

Один из аспектов программного проекта касается определения системы: выяснение требований к системе, ее контекста и целей. Что должна делать система? Для кого она должна это делать? Каковы выходные данные системы? Отвечают ли они поставленным требованиям? На нижнем уровне *проектирование* можно понимать как процесс, посредством которого определяются участники системы и устанавливается связь между ними. Эта глава посвящена второму аспекту: определению и расположению классов и объектов.

Кто же такой участник? Объектно-ориентированная система состоит из классов. И очень важно решить, каким будет характер этих классов в проектируемой системе. Классы отчасти состоят из методов. Поэтому при определении классов необходимо решить, какие методы следует объединить, чтобы они составляли одно целое. Но, как будет показано в этой главе, классы часто объединяются в отношения наследования, чтобы подчиняться общим интерфейсам. Именно эти интерфейсы, или типы, должны стать первыми участниками проектируемой системы.

Существуют и другие отношения, которые можно определить для классов. Так, можно создать классы, состоящие из других типов или управляющие списками экземпляров других типов. Кроме того, можно спроектировать классы, в которых просто используются другие объекты. Возможность для составления или использования таких отношений встроена в классы (например, через объявления типов классов в сигнатурах методов). Но реальные отношения между объектами вступают в действие во время выполнения программы, что позволяет сделать процесс проектирования более гибким. Из этой главы вы узнаете, как моделировать эти отношения, а в остальной части книги они будут исследованы более подробно.

В процессе проектирования требуется решить, когда операция должна относиться к некоторому типу, а когда — к классу, используемому этим типом. Все, с чем нам приходится иметь дело, постоянно ставит нас перед необходимостью делать тот или иной выбор, который приведет к ясному и изящному проектному решению или затянет в болото компромиссов.

В этой главе основное внимание уделяется вопросам, способным повлиять на выбор проектных решений.

Объектно-ориентированное и процедурное программирование

Чем объектно-ориентированный проект отличается от более традиционного процедурного кода? Так и хочется сказать: "Главное отличие в том, что в объектно-ориентированном коде имеются объекты". Но это неверно. С одной стороны, в языке PHP часто случается так, что в процедурном коде используются объекты. А с другой стороны, могут встретиться классы, в которых содержатся фрагменты процедурного кода. Наличие классов не может служить гарантией объектно-ориентированного характера проекта,

даже в таком языке, как Java, в котором практически все операции выполняются в пределах класса.

Коренное отличие объектно-ориентированного кода от процедурного заключается в распределении ответственности. Процедурный код имеет форму последовательности команд и вызовов функций. Управляющий код обычно несет ответственность за обработку различных ситуаций. Это управление сверху вниз может привести к дублированию и возникновению зависимостей в проекте. Объектно-ориентированный код пытается свести к минимуму эти зависимости, передавая ответственность за управление задачами от клиентского кода к объектам в системе.

В этом разделе рассматривается простая задача, которая анализируется с точки зрения объектно-ориентированного и процедурного подходов, чтобы продемонстрировать их различия. Эта задача состоит в том, чтобы создать простое средство для считывания информации из конфигурационных файлов и записи в них. Чтобы сосредоточить внимание на структуре кода, в приведенных далее примерах опускается код их реализации.

Начнем с процедурного подхода к решению данной задачи. Организуем сначала чтение и запись текста в следующем формате:

```
ключ:значение
```

Для этой цели нам потребуются только две функции:

```
// Листинг 6.1
function readParams(string $source): array
{
    $params = [];
    // Чтение текстовых параметров из $source
    return $params;
}
function writeParams(array $params, string $source): void
{
    // Запись текстовых параметров в $source
}
```

Функции `readParams()` нужно передать имя файла конфигурации. Она пытается открыть его и прочитать из него каждую строку, отыскивая в ней пары “ключ–значение”. По ходу дела эта функция создает ассоциативный массив и возвращает этот массив управляющему коду. А функции `writeParams()` передаются ассоциативный массив и имя файла конфигурации. Она перебирает этот ассоциативный массив в цикле, записывая

каждую пару “ключ–значение” в файл конфигурации. Ниже приведен пример клиентского кода, в котором применяются эти функции:

```
// Листинг 6.2
$file = "/tmp/params.txt";
$params = [
              "key1" => "val1",
              "key2" => "val2",
              "key3" => "val3",
           ];
writeParams($params, $file);
$output = readParams($file);
print_r($output);
```

Этот код относительно компактный, и его легко сопровождать. При вызове функции `writeParams()` создается файл `param.txt`, в который записывается приведенная ниже информация:

```
key1:val1
key2:val2
key3:val3
```

Функция `readParams()` производит синтаксический анализ в том же самом формате. Многие проекты постепенно расширяют свои рамки и развиваются. Сымитируем эту особенность проектов, введя новое требование, в соответствии с которым код должен также обрабатывать структуру XML-файла конфигурации, которая выглядит следующим образом:

```
<params>
    <param>
        <key>Мой ключ</key>
        <val>Мое значение</val>
    </param>
</params>
```

Если данный конфигурационный файл имеет расширение `.xml`, то для его обработки необходимо задействовать средства обработки XML-разметки, встроенные в PHP. И хотя такую задачу совсем нетрудно решить, существует угроза, что прикладной код станет намного труднее сопровождать. Для этого у нас имеются две возможности: проверить расширение файла в управляющем коде или выполнить такую проверку в теле функций чтения и записи. Остановимся на второй возможности:

```
// Листинг 6.3
function readParams(string $source): array
{
```

```
    $params = [];

    if (preg_match("/\.xml$/i", $source))
    {
        // Чтение XML-параметров из $source
    }
    else
    {
        // Чтение текстовых параметров из $source
    }

    return $params;
}

function writeParams(array $params, string $source): void
{
    if (preg_match("/\.xml$/i", $source))
    {
        // Запись XML-параметров в $source
    }
    else
    {
        // Запись текстовых параметров в $source
    }
}
```

На заметку Код любого примера всегда несет в себе элемент трудного компромисса. Он должен быть достаточно понятным, чтобы объяснять суть дела, но это часто означает, что приходится жертвовать проверкой ошибок и пригодностью кода для той цели, для которой он был написан. Иными словами, рассматриваемый здесь пример кода предназначен для иллюстрации вопросов проектирования и дублирования кода, а не лучшего способа синтаксического анализа и записи данных в файл. По этой причине конкретная реализация будет опускаться там, где это не имеет отношения к обсуждаемому вопросу.

Как видите, расширение конфигурационного файла `.xml` пришлось проверять в каждой функции. Именно это повторение в итоге может стать причиной осложнений. Если же заказчики попросят включить в проектируемую систему поддержку еще одного формата параметров, то нужно внести изменения в функции `readParams()` и `writeParams()`.

А теперь попробуем решить ту же самую задачу с помощью простых классов. Создадим сначала абстрактный базовый класс, в котором будет определяться интерфейс заданного типа:

```
// Листинг 6.4
abstract class ParamHandler
{
    protected array $params = [];
    public function __construct(protected string $source)
    {
    }
    public function addParam(string $key, string $val): void
    {
        $this->params[$key] = $val;
    }
    public function getAllParams(): array
    {
        return $this->params;
    }
    public static function getInstance(string $filename):ParamHandler
    {
        if (preg_match("/\.xml$/i", $filename))
        {
            return new XmlParamHandler($filename);
        }

        return new TextParamHandler($filename);
    }
    abstract public function write(): void;
    abstract public function read(): void;
}
```

В приведенном выше фрагменте кода мы определили метод `addParam()`, чтобы пользователь мог вводить параметры в защищенное свойство `$params`, и метод `getAllParams()` — чтобы предоставить доступ к копии массива параметров.

Кроме того, мы создали статический метод `getInstance()`, в котором проверяется расширение файла и возвращается объект конкретного подкласса в соответствии с результатами проверки. А самое главное — мы определили два абстрактных метода, `read()` и `write()`, гарантировав тем самым, что любые подклассы будут поддерживать этот интерфейс.

На заметку Разместить статический метод для формирования дочерних объектов в родительском классе, конечно, удобно. Но такое проектное решение будет иметь отрицательные последствия, поскольку возможности класса `ParamHandler` теперь станут существенно ограниченными в отношении работы с его конкретными подклассами, возвращаемыми рассматриваемым статическим методом из центрального условного оператора. Что, например, произойдет, если понадобится работать с еще одним форматом файлов конфигурации?

Безусловно, если вы сопровождаете код класса `ParamHandler`, то всегда можете исправить метод `getInstance()`. Но если вы создаете клиентский код, то изменение этого библиотечного класса может быть не таким простым делом (на самом деле изменить его несложно, но в перспективе вам придется вносить исправление при каждой переустановке пакета, в котором применяется этот класс). Вопросы создания объектов мы обсудим в главе 9, "Генерация объектов".

Определим далее подклассы, снова опуская подробности реализации ради ясности примера кода:

```
// Листинг 6.5
class XmlParamHandler extends ParamHandler
{
    public function write(): void
    {
        // Запись XML
        // с использованием $this->params
    }
    public function read(): void
    {
        // Чтение XML
        // и заполнение $this->params
    }
}

// Листинг 6.6
class TextParamHandler extends ParamHandler
{
    public function write(): void
    {
        // Запись текста
        // с использованием $this->params
    }
    public function read(): void
    {
        // Чтение текста
        // и заполнение $this->params
    }
}
```

В этих классах просто обеспечена реализация методов `read()` и `write()`. Каждый класс будет читать и записывать данные в соответствующем формате.

В итоге из клиентского кода можно будет совершенно прозрачно записывать параметры конфигурации в текстовый или XML-файл в зависимости от расширения имени файла:

```
// Листинг 6.7
$test = ParamHandler::getInstance(__DIR__ . "/params.xml");
$test->addParam("key1", "val1");
$test->addParam("key2", "val2");
$test->addParam("key3", "val3");
$test->write(); // Запись в формате XML
```

Параметры конфигурации можно также прочитать из файла любого поддерживаемого формата следующим образом:

```
// Листинг 6.8
$test = ParamHandler::getInstance(__DIR__ . "/params.txt");
$test->read(); // Чтение в текстовом формате
$params = $test->getAllParams();
print_r($params);
```

Итак, какие выводы можно сделать из рассмотренных здесь двух подходов?

Ответственность

Управляющий код в примере процедурного кода несет ответственность за выбор формата, но принимает это решение не один раз, а дважды. И хотя условный код перенесен в функции, это лишь скрывает факт принятия решений в одном потоке. Вызов функции `readParams()` всегда происходит в ином контексте, чем вызов функции `writeParams()`, поэтому мы вынуждены повторять проверку расширения файла в каждой функции (или выполнять разновидности этой проверки).

В объектно-ориентированном примере кода выбор формата файла делается в статическом методе `getInstance()`, в котором расширение файла проверяется только один раз и создается нужный подкласс. Клиентский код не несет ответственности за реализацию. Он использует предоставленный объект, не зная и даже не интересуясь, какому именно подклассу он принадлежит. Ему известно только, что он работает с объектом типа `ParamHandler`, в котором поддерживаются методы `read()` и `write()`. Если процедурный код занят подробностями реализации, то объектно-ориентированный код работает только с интерфейсом, не заботясь о подробностях реализации. А поскольку ответственность за реализацию лежит на объектах, а не на клиентском коде, то поддержку новых форматов будет нетрудно внедрить прозрачным образом.

Связность

Связность — это степень, в которой соседние процедуры зависят одна от другой. В идеальном случае следует создавать компоненты, в которых ответственность разделена совершенно четко и ясно. Если по всему коду разбросаны связанные процедуры, то вскоре обнаружится, что их трудно сопровождать, потому что придется прилагать усилия для поиска мест, где необходимо внести изменения.

В рассматриваемых здесь классах типа `ParamHandler` связанные процедуры собраны в общем контексте. Методы для обработки данных в формате XML совместно используют один контекст, в котором они могут обмениваться данными и в котором необходимые изменения в одном методе могут быть легко отражены в другом, если, например, потребуется изменить имя элемента XML-разметки. И это дает основания считать, что классы `ParamHandler` имеют сильную связность.

А в процедурном примере кода связанные процедуры разделены. Код для обработки данных формата XML распределен по нескольким функциям.

Тесная связь

Тесная связь (coupling) возникает в том случае, когда отдельные части кода системы тесно связаны одна с другой, так что изменение в одной части вызывает потребность вносить изменения в других частях. Тесная связь не обязательно возникает в процедурном коде, но последовательный характер такого кода приводит к определенным осложнениям.

Такой вид тесной связи можно наблюдать в примере процедурного кода. Функции `readParams()` и `writeParams()` производят одну и ту же проверку расширения файла, чтобы определить порядок обработки данных. Любое изменение в логике работы одной функции должно быть реализовано и в другой. Так, если бы потребовалось добавить новый формат файла, для этого пришлось бы привести в соответствие все функции, чтобы новое расширение файла обрабатывалось в них одинаково. По мере внедрения новых функций, обрабатывающих другие форматы файлов, эта проблема будет только усугубляться.

А в примере объектно-ориентированного кода классы не связаны ни между собой, ни с клиентским кодом. И если бы потребовалось внедрить поддержку нового формата файла, то было бы достаточно создать

новый подкласс, изменив единственную проверку в статическом методе `getInstance()`.

Ортогональность

Практически идеальное сочетание компонентов с четко определенными обязанностями наряду с независимостью от более обширного контекста системы иногда называют *ортогональностью*. Этот предмет обсуждается, например, в книге Эндрю Ханта (Andrew Hunt) и Дэвида Томаса (David Thomas) *The Pragmatic Programmer, 20th Anniversary Edition*[1].

Как утверждают авторы данной книги, ортогональность способствует повторному использованию кода, поскольку готовые компоненты можно включать в новые системы без специальной их настройки. Такие компоненты должны иметь четко определенные входные и выходные данные, независимые от какого-либо более обширного контекста. В ортогональный код легче вносить изменения, поскольку изменение реализации будет локализовано в том компоненте, где вносятся изменения. И, наконец, ортогональный код безопаснее. Последствия ошибок будут ограничены в определенном контексте. В то же время ошибка в чрезвычайно взаимозависимом коде может легко оказать эффект цепной реакции на остальную систему в более обширном контексте.

Но слабая связь и сильная связность не являются автоматическими признаками в контексте класса. Ведь в конечном счете целый пример процедурного кода можно вставить в один неверный класс. Как же тогда достичь золотой середины в разрабатываемом коде? Как правило, начинать рекомендуется с рассмотрения классов, которые должны присутствовать в проектируемой системе.

Выбор классов

Порой определить границы классов бывает необычайно сложно. И это особенно трудно сделать по мере развития классов вместе с проектируемой системой.

При моделировании реальной ситуации эта задача может показаться нетрудной. Как правило, объектно-ориентированные системы являются

---

[1] Хант, Э., Томас, Д. *Программист-прагматик: 2-е юбилейное изд.* : Пер. с англ. — СПб. : ООО "Диалектика", 2020.

программным представлением реальных вещей, поскольку в них нередко применяются классы вроде Person, Invoice и Shop. Следовательно, можно предположить, что определение классов — это вопрос выявления некоторых *объектов* в системе и наделения их определенными функциями с помощью методов. И хотя это неплохая отправная точка, она не лишена скрытых опасностей. Если рассматривать класс как существительное, которым оперирует произвольное количество глаголов, то окажется, что он будет все больше расширяться, потому что в ходе разработки и внесения изменений потребуется, чтобы класс выполнял все больше и больше обязанностей.

Рассмотрим в качестве примера класс ShopProduct, который был создан в главе 3, “Азы объектов”. В контексте системы, предназначенной для продажи товаров покупателям, определение класса ShopProduct является вполне очевидным решением. Но является ли это решение единственным? Для доступа к данным о товаре предоставляются методы getTitle() и getPrice(). Если заказчики попросят предоставить механизм для вывода кратких сведений о товарах для счетов-фактур и уведомлений о доставке товаров, то, вероятно, имеет смысл определить метод write(). А если заказчики попросят предоставить механизм выдачи подобных сведений в разных форматах, то придется снова обратиться к классу ShopProduct и надлежащим образом создать методы writeXML() и writeXHTML() в дополнение к методу write(). С другой стороны, в тело метода write() можно ввести условные операторы, чтобы выводить данные в различных форматах в соответствии с признаком, устанавливаемым в дополнительном аргументе.

Но в любом случае проблема заключается в том, что класс ShopProduct теперь пытается делать слишком много. Кроме хранения сведений о самом товаре, этот класс должен еще выбирать способы представления этих данных.

Так как же *следует* определять классы? Наилучший подход состоит в том, чтобы рассматривать класс наделенным основной обязанностью и сделать эту обязанность как можно более конкретной и единственной. Эту обязанность следует выразить словами. Считается, что обязанность класса должна описываться не более чем 25 словами с редкими включениями союзов “и” и “или”. И если предложение, описывающее обязанность класса, становится слишком длинным или перегружено подчиненными предложениями, значит, настало время подумать об определении новых классов, наделив их некоторыми из описанных обязанностей.

Итак, обязанность класса ShopProduct — хранить сведения о товаре. И если мы добавляем в него методы для вывода данных в разных форматах, то тем самым наделяем его новыми обязанностями представлять сведения о товарах. Как было показано в главе 3, "Азы объектов", на основе этих отдельных обязанностей были определены два разных класса. В частности, класс ShopProduct остался ответственным за хранение сведений о товарах, а класс ShopProductWriter взял на себя обязанность за представление этих сведений. Уточнение этих обязанностей осуществляется в отдельных подклассах.

На заметку Далеко не все правила проектирования являются жесткими. Иногда встречается код, предназначенный для хранения объектных данных и находящийся в другом классе, совершенно не связанном с текущим классом. Очевидно, что самым удобным местом для размещения таких функциональных возможностей служит текущий класс, хотя такой подход, на первый взгляд, нарушает правило наделения класса только одной обязанностью. Такое удобство объясняется тем, что локальный метод будет иметь полный доступ к полям экземпляра класса. Использование локальных методов для целей сохраняемости позволит также не создавать параллельную иерархию классов сохраняемости, которые являются зеркальным отображением классов сохраняемых объектов, что неизбежно приведет к их связанности. Другие стратегии обеспечения сохраняемости объектов будут рассмотрены в главе 12, "Шаблоны корпоративных приложений". Старайтесь не воспринимать установленные правила проектирования как догму, поскольку они не могут заменить анализ поставленной задачи. Придерживайтесь больше духа, а не буквы правил.

Полиморфизм

Полиморфизм, или замена классов, — это общее свойство объектно-ориентированных систем. Его проявление уже встречалось в примерах кода из данной книги.

Полиморфизм — это поддержка нескольких реализаций на основе общего интерфейса. На первый взгляд такое определение может показаться сложным, но на самом деле это понятие должно быть вам уже знакомо. О необходимости полиморфизма обычно говорит наличие в коде большого количества условных операторов.

Когда класс ShopProduct был впервые создан в главе 3, "Азы объектов", мы экспериментировали с этим единственным классом, используя его для хранения сведений о книгах и компакт-дисках, а также о других товарах

общего назначения. Для вывода краткой сводки о товаре мы полагались на ряд условных инструкций, как показано ниже.

```
// Листинг 6.9
public function getSummaryLine(): string
{
    $base = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";

    if ($this->type == 'book')
    {
        $base .= ": {$this->numPages} стр.";
    }

    elseif($this->type == 'cd')
    {
        $base .= ": Время звучания - {$this->playLength}";
    }
    return $base;
}
```

Именно эти условные инструкции и определили функциональные возможности двух подклассов: `CDProduct` и `BookProduct`. Аналогичным образом в рассмотренном ранее примере процедурного кода именно в условных инструкциях, проверяющих значения параметров, было заложено основание для объектно-ориентированной структуры, к которой мы в конечном итоге пришли. Но одни и те же условия проверяются в двух разных частях сценария, как показано ниже.

```
// Листинг 6.10
function readParams(string $source): array
{
    $params = [];

    if (preg_match("/\.xml$/i", $source))
    {
        // Чтение XML-параметров из $source
    }
    else
    {
        // Чтение текстовых параметров из $source
    }

    return $params;
}
```

```
function writeParams(array $params, string $source): void
{
    if (preg_match("/\.xml$/i", $source))
    {
        // Запись XML-параметров в $source
    }
    else
    {
        // Запись текстовых параметров в $source
    }
}
```

В каждой из этих частей напрашивается решение выделить их в отдельные подклассы `XmlParamHandler` и `TextParamHandler`, которые в конце концов и были созданы. В этих подклассах реализованы методы `write()` и `read()` из абстрактного базового класса `ParamHandler`, который они расширяют:

```
// Листинг 6.11
// Может возвращать XmlParamHandler или TextParamHandler
$test = ParamHandler::getInstance($file);
$test->read();  // Может быть XmlParamHandler::read() или
                // TextParamHandler::read()
$test->addParam("newkey1", "newval1");
$test->write(); // Может быть XmlParamHandler::write() или
                // TextParamHandler::write()
```

Важно отметить, что полиморфизм не отрицает использование условных инструкций. Обычно в методах наподобие `ParamHandler::getInstance()` с помощью конструкции `switch` или инструкции `if` определяется, какие именно объекты следует возвратить. Но это ведет к сосредоточению кода с условными инструкциями в единственном месте.

Как было показано ранее, в языке PHP приходится создавать интерфейсы, определяемые в абстрактных классах. Это удобно, потому что дает уверенность, что в конкретном дочернем классе будут в точности поддерживаться те же самые сигнатуры методов, которые были определены в абстрактном родительском классе. В них будут включены все объявления типов классов и модификаторы доступа. Поэтому все дочерние классы общего суперкласса можно рассматривать в клиентском коде как взаимозаменяемые, при условии, что он опирается только на функциональные возможности, определенные в родительском классе.

Инкапсуляция

Инкапсуляция — это просто сокрытие данных и функциональных возможностей от клиентского кода. И в то же время это ключевое понятие объектно-ориентированного программирования.

На самом элементарном уровне данные инкапсулируются путем объявления свойств как `private` или `protected`. Скрывая свойство от клиентского кода, мы соблюдаем общий интерфейс и тем самым предотвращаем случайное повреждение данных объекта.

Полиморфизм демонстрирует другой вид инкапсуляции. Скрывая различные реализации за общим интерфейсом, мы прячем основополагающие стратегии от клиентского кода. Это означает, что любые изменения, внесенные за этим интерфейсом, являются прозрачными для более обширного контекста остальной системы. Мы можем ввести новые классы или изменить исходный код в классе, и это не приведет к ошибкам. Значение имеет не интерфейс, а действующий за ним механизм. Чем более независимы подобные механизмы, тем меньше вероятность, что внесенные изменения или поправки окажут эффект цепной реакции на ваши проекты.

В каком-то отношении инкапсуляция — это ключ к объектно-ориентированному программированию. Наша цель — сделать каждую часть проектируемой системы как можно более независимой от остальных ее частей. Классы и методы должны получать столько информации, сколько требуется для выполнения стоящих перед ними задач, которые должны быть ограничены конкретными рамками и ясно определены.

Внедрение ключевых слов `private`, `protected` и `public` облегчает инкапсуляцию. Но инкапсуляция — это и образ мышления. В версии PHP 4 не было предусмотрено формальной поддержки для сокрытия данных. О конфиденциальности необходимо было предупреждать с помощью документации и соглашений об именовании. Например, символ подчеркивания — это обычный способ предупреждения о закрытом свойстве:

```
var $_touchezpas;
```

Безусловно, написанный код приходилось тщательно проверять, потому что конфиденциальность не была строго соблюдена. Но любопытно, что ошибки случались редко, потому что структура и стиль кода довольно четко определяли, какие свойства трогать нельзя.

И даже в версии PHP 5 можно нарушить правила и выяснить точный подтип объекта, который используется в контексте замены классов, просто выполнив операцию instanceof:

```
// Листинг 6.12
public function workWithProducts(ShopProduct $prod)
{
    if ($prod instanceof CDProduct)
    {
        // обработать данные о компакт-диске
    } elseif($prod instanceof BookProduct)

    {
        // обработать данные о книге
    }
}
```

Для выполнения подобных действий должна быть серьезная причина, поскольку в целом это вносит некоторую неопределенность. Запрашивая конкретный подтип, как в данном примере, мы устанавливаем жесткую зависимость. Если же особенности подтипа скрываются полиморфизмом, то можно совершенно безболезненно изменить иерархию наследования класса ShopProduct, не опасаясь негативных последствий. Но в рассматриваемом здесь коде этому положен предел. Если теперь нам потребуется усовершенствовать классы CDProduct и BookProduct, мы должны не забывать о возможном проявлении нежелательных побочных эффектов в методе workWithProducts().

Из данного примера мы должны вынести два урока. Во-первых, инкапсуляция помогает создать ортогональный код. И во-вторых, степень инкапсуляции, которую можно соблюсти, к делу не относится. Инкапсуляция — это методика, которую должны в равной степени соблюдать как сами классы, так и их клиенты.

Забудьте, как это делается

Если вы похожи на меня, то упоминание о конкретной задаче заставит ваш ум интенсивно работать в поисках подходящего решения. Вы можете выбрать функции, которые могут решить данную задачу, поискать регулярные выражения, исследовать пакеты Composer. Возможно, у вас есть фрагмент кода из старого проекта, который выполняет нечто подобное, и его можно вставить в новый код. На стадии разработки вы выиграете, если

на некоторое время отложите все это в сторону, выбросив из головы всякие процедуры и механизмы.

Думайте только о ключевых участниках системы: требующихся типах данных и интерфейсах. Безусловно, знание программируемого процесса окажет помощь в ваших размышлениях. В частности, классу, в котором открывается файл, необходимо передать имя этого файла; код базы данных должен оперировать именами таблиц, паролями и т.д. Но старайтесь, чтобы ход ваших мыслей следовал структурам и отношениям в разрабатываемом коде. В итоге вы обнаружите, что реализация легко встанет на свое место за правильно определенным интерфейсом. И тогда вы получите широкие возможности заменить, улучшить или расширить реализацию, если в этом возникнет потребность, не затрагивая остальную систему в более обширном контексте.

Чтобы сосредоточить основное внимание на интерфейсе, старайтесь мыслить категориями абстрактных базовых классов, а не конкретных дочерних классов. Так, в рассмотренном выше примере кода для извлечения параметров интерфейс является самым важным аспектом проектирования. Для этого требуется тип данных, предназначенный для чтения и записи пар "имя–значение". Для такого типа данных важна именно эта обязанность, а не реальный носитель, на котором будут сохраняться данные или способы их хранения и извлечения. Проектирование системы необходимо сосредоточить вокруг абстрактного класса `ParamHandler`, внедрив только конкретные стратегии, чтобы в дальнейшем можно было читать и записывать параметры. Таким образом, полиморфизм и инкапсуляция внедряются в проектируемую систему с самого начала. Такая структура уже тяготеет к использованию замены классов.

Но мы, конечно, знали с самого начала, что должны существовать текстовая и XML-реализации класса `ParamHandler`, которые, безусловно, оказывают влияние на проектируемый интерфейс. При проектировании интерфейсов всегда есть над чем подумать, прежде чем выбрать подходящее решение.

"Банда четырех" в упоминавшейся ранее книге *Design Patterns* сформулировала этот принцип следующим образом: "*Программируйте на основе интерфейса, а не его реализации*". Это высказывание заслуживает того, чтобы вы занесли его как памятку в свою записную книжку.

Четыре явных признака недоброкачественного кода

Очень немногие разработчики принимают совершенно правильные решения еще на стадии проектирования. Большинство из них исправляют код по мере изменения требований к нему или в результате более полного понимания характера решаемой задачи.

Но по мере изменения кода он может в конечном счете выйти из-под контроля. Если ввести метод в одном месте, а класс — в другом, то система станет постепенно усложняться. Но, как было показано ранее, сам код может указывать пути для его совершенствования. Такие признаки в разрабатываемом коде иногда называют *запахами кода (code smells)*, или *недоброкачественным кодом* (буквально — кодом "с душком"). Речь идет о тех местах в коде, где *необходимо* внести конкретные исправления или, по крайней мере, еще раз проанализировать проектное решение. В этом разделе некоторые уже рассмотренные моменты будут выделены в четыре явных признака недоброкачественного кода, которые необходимо всегда иметь в виду в процессе программирования.

Дублирование кода

Дублирование считается одним из самых крупных недостатков прикладного кода. Если во время написания процедуры у вас появляется странное ощущение уже виденного, то дело, вероятнее всего, в дублировании кода.

Выявите повторяющиеся компоненты в проектируемой системе. Возможно, они каким-то образом связаны вместе. Дублирование обычно свидетельствует о наличии тесной связи компонентов. Если вы изменяете что-нибудь основательное в одной процедуре, то понадобится ли вносить исправления в похожие процедуры? И если это именно так, то, вероятнее всего, они относятся к одному и тому же классу.

Класс, который слишком много знал

Передавать параметры из одного метода в другой может быть очень неудобно. Почему бы не упростить дело, воспользовавшись глобальной переменной? С помощью глобальной переменной всякий может получить доступ к общим данным.

Глобальные переменные, безусловно, важны, но к ним нужно относиться с большим подозрением. Используя глобальную переменную или наделяя класс любыми сведениями о более обширной предметной области, вы привязываете класс к его контексту, делая его менее пригодным для повторного использования и зависимым от кода, который находится вне его контроля. Помните, что классы и процедуры необходимо развязывать друг от друга, избегая их взаимной зависимости. Постарайтесь ограничить осведомленность класса о его контексте. Некоторые стратегии воплощения этого принципа проектирования мы рассмотрим далее в книге.

На все руки мастер

А что, если класс пытается сразу делать слишком много? В таком случае попробуйте составить список обязанностей класса. Вполне возможно, что одна из них послужит основанием для создания отдельного класса.

Оставить без изменения класс, выполняющий слишком много обязанностей, — значит вызвать определенные осложнения при создании его подклассов. Какую обязанность вы расширяете с помощью подкласса? Что вы будете делать, если вам понадобится подкласс для выполнения нескольких обязанностей? Вполне возможно, что у вас получится слишком много подклассов или слишком сильная зависимость от условного кода.

Условные инструкции

У вас, вероятно, будут веские причины для употребления инструкций `if` и `switch` в своем коде. Но иногда наличие подобных структур служит сигналом прибегнуть к полиморфизму.

Если вы обнаружите, что проверяете некоторые условия в классе слишком часто, особенно если эти проверки повторяются в нескольких методах, то, возможно, это сигнализирует о том, что один класс нужно разделить на два класса или более. Выясните, не предполагает ли структура условного кода обязанности, которые можно выразить в отдельных классах. В этих новых классах должны быть реализованы методы общего абстрактного базового класса. Вполне вероятно, что затем вам придется найти способ передать нужный класс клиентскому коду. О некоторых проектных шаблонах для создания объектов речь пойдет в главе 9, “Генерация объектов”.

Язык UML

В рассмотренных выше примерах код не требовал особых пояснений. В этих кратких примерах демонстрировались такие понятия, как наследование и полиморфизм. И это было полезно, потому что в центре нашего внимания находился сам язык PHP. Но по мере увеличения размеров и сложности примеров использование только исходного кода для демонстрации широкого спектра проектных решений становится нецелесообразным. Согласитесь, нелегко увидеть общую картину в нескольких строках кода.

Сокращение "UML" означает "Unified Modeling Language" (Унифицированный язык моделирования). История создания этого языка очень интересна. По мнению Мартина Фаулера (из книги *UML Distilled*, Addison-Wesley Professional, 1999[2]), язык UML возник как стандарт после многолетних интеллектуальных и бюрократических споров в сообществе приверженцев объектно-ориентированного проектирования.

Результатом этих споров стал эффективный графический синтаксис для описания объектно-ориентированных систем. В этом разделе мы рассмотрим его только в общих чертах, но вскоре вы поймете, что "скромный малый" UML прошел долгий путь развития.

Диаграммы классов позволяют описывать структуры и шаблоны таким образом, чтобы их смысл становился ясным и понятным. Такой кристальной ясности трудно достичь с помощью фрагментов кода или маркированных списков.

Диаграммы классов

Диаграммы классов — это всего лишь один аспект UML, но именно они чаще всего употребляются в объектно-ориентированном проектировании. А поскольку они очень полезны для описания объектно-ориентированных связей, мы будем часто пользоваться ими в этой книге.

Представление классов

Как и следовало ожидать, классы являются главными составными элементами диаграмм классов. На такой диаграмме класс представлен в виде именованного прямоугольника (рис. 6.1).

---

[2] Фаулер, М. *UML. Основы. Краткое руководство по стандартному языку объектного моделирования* : Пер. с англ. — Изд-во "Символ-Плюс", 2011.

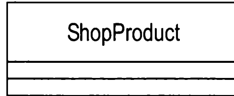


Рис. 6.1. *Представление класса в UML*

Прямоугольник, представляющий класс, делится на три части, в первой из которых отображается имя класса. Разделительные линии необязательны, если больше никакой дополнительной информации, кроме имени класса, не предоставляется. Составляя диаграммы классов, можно обнаружить, что для некоторых классов достаточно того уровня детализации, который представлен на рис. 6.1. На диаграмме классов совсем не обязательно обозначать все поля и методы и даже все классы.

Для представления абстрактных классов на диаграмме имя класса выделяется курсивом (рис. 6.2) или же дополняется обозначением `{abstract}`, как показано на рис. 6.3. Первый способ считается более распространенным, а второй — более удобным для того, чтобы делать заметки в записной книжке.

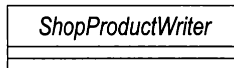


Рис. 6.2. *Пример представления абстрактного класса*

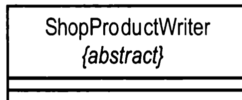


Рис. 6.3. *Пример представления абстрактного класса с дополнительным ограничением*

На заметку Обозначение `{abstract}` — это пример ограничения. На диаграммах классов ограничения служат для описания порядка применения отдельных элементов. Специальные правила для указания текста в фигурных скобках не определены. Он просто должен ясно пояснять условия, устанавливаемые для элемента.

Интерфейсы определяются так же, как и классы, но должны включать стереотип (т.е. расширение словаря UML), как показано на рис. 6.4.

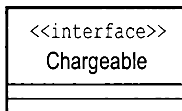


Рис. 6.4. *Пример представления интерфейса*

Атрибуты

Вообще говоря, атрибуты описывают свойства класса. Атрибуты перечисляются в той части диаграммы, которая располагается непосредственно под именем класса (рис. 6.5).

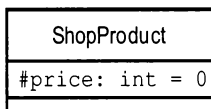


Рис. 6.5. *Пример обозначения атрибута*

Рассмотрим подробнее атрибут из примера, приведенного на рис. 6.5. Первый знак (#) в этом атрибуте обозначает уровень видимости атрибута или управления доступом к нему. В табл. 6.1 перечислены знаки для обозначения уровня видимости.

Таблица 6.1. *Знаки уровня видимости*

Знак	Уровень видимости	Описание
+	Общедоступный	Доступность для всего кода
-	Закрытый	Доступность только для текущего класса
#	Защищенный	Доступность только для текущего класса и его подклассов

После знака уровня видимости указывается имя атрибута. В данном случае описывается свойство `ShopProduct::$price`. Двоеточие служит для отделения имени атрибута от его типа, а возможно, и от его стандартного значения. Следует еще раз подчеркнуть, что в описание атрибута можно включить столько подробностей, сколько потребуется для ясности.

Операции

Операции описывают методы, а точнее — вызовы, которые могут быть сделаны для экземпляра класса. На рис. 6.6 показаны две операции, обозначаемые в классе ShopProduct.

ShopProduct
#price: int = 0
+setDiscount(amount:int) +getTitle(): String

Рис. 6.6. *Представление операций*

Как видите, для обозначения операций служит такой же синтаксис, как и для обозначения атрибутов. Знак уровня видимости предшествует имени метода. Список параметров заключается в круглые скобки. А тип значения, возвращаемого методом, если оно вообще возвращается, отделяется двоеточием. Параметры разделяются запятыми, и для их обозначения используется такой же синтаксис, как и для атрибута: имя атрибута отделяется от его типа двоеточием.

Как и следовало ожидать, это очень удобный синтаксис. Признак уровня видимости и тип возвращаемого значения можно опустить. Параметры часто обозначаются только их типом, поскольку имя аргумента обычно не имеет особого значения.

Описание наследования и реализации

В языке UML отношения наследования описываются в виде обобщений. Это отношение обозначается линией, ведущей от подкласса к родительскому классу. Эта линия оканчивается незаполненной замкнутой стрелкой. В качестве примера на рис. 6.7 показана связь между классом ShopProduct и его дочерними классами.

Средствами языка UML описывается также связь между интерфейсом и классами, в которых он реализован. Так, если бы в классе ShopProduct был реализован интерфейс Chargeable, мы бы добавили его в диаграмму класса, как показано на рис. 6.8.

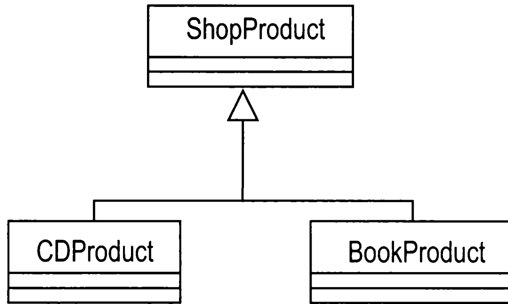


Рис. 6.7. Описание наследования

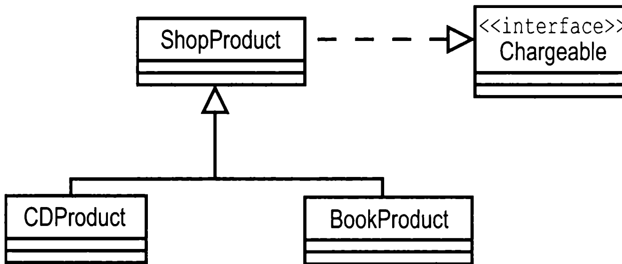


Рис. 6.8. Описание реализации интерфейса

Ассоциации

Наследование — это только одно из отношений в объектно-ориентированной системе. Ассоциация происходит в том случае, когда в классе объявляется свойство, в котором содержится ссылка на экземпляр (или экземпляры) другого класса. В качестве примера на рис. 6.9 моделируются два класса и образуется ассоциация между ними.

Рис. 6.9. Ассоциация класса

На данном этапе нам неизвестна природа этих отношений. Мы только указали, что у объекта типа `Teacher` будет ссылка на один или несколько объектов типа `Pupil`, или наоборот. Это отношение может быть или не быть взаимным.

Стрелки на диаграмме классов служат для описания направления ассоциации. Так, если в классе `Teacher` имеется ссылка на экземпляр класса `Pupil`, а не наоборот, то их ассоциацию следует определить с помощью стрелки, направленной от класса `Teacher` к классу `Pupil`. Такая ассоциация называется однонаправленной и показана на рис. 6.10.

Рис. 6.10. *Пример однонаправленной ассоциации*

Если в каждом классе имеется ссылка на другой класс, то для обозначения двунаправленного отношения служит двунаправленная стрелка, как показано на рис. 6.11.

Рис. 6.11. *Двунаправленная ассоциация*

Имеется также возможность указать количество экземпляров класса, на которые ссылается другой класс в ассоциации. Для этого достаточно указать число или диапазон чисел рядом с каждым классом. Можно также использовать знак звездочки (`*`), обозначающий любое число. На рис. 6.12 показано, что может существовать только один объект типа `Teacher` и нуль или больше объектов типа `Pupil`.

Рис. 6.12. *Пример определения множественности для ассоциации*

На рис. 6.13 показано, что в ассоциации может быть один объект типа `Teacher` и от пяти до десяти объектов типа `Pupil`.

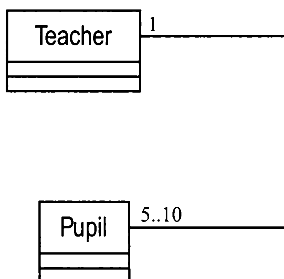


Рис. 6.13. *Еще один вариант определения множественности для ассоциации*

Агрегирование и композиция

Агрегирование и композиция подобны ассоциации. Все эти термины служат для описания ситуации, когда в классе содержится постоянная ссылка на один или несколько экземпляров другого класса. Но с помощью агрегирования и композиции экземпляры, на которые ссылаются, формируют внутреннюю часть ссылающегося объекта.

Что касается агрегирования, то объекты составляют основную часть объекта-контейнера, который их содержит, но они могут одновременно содержаться и в других объектах. Отношение агрегирования обозначается линией, которая начинается с незаполненного ромба. В качестве примера на рис. 6.14 определяются два класса, `SchoolClass` и `Pupil`, причем класс `SchoolClass` агрегирует класс `Pupil`.

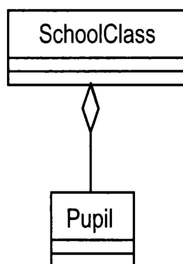


Рис. 6.14. *Пример агрегирования*

Класс `Pupil` состоит из учеников, но на один и тот же объект типа `Pupil` могут одновременно ссылаться различные экземпляры класса

SchoolClass. Если же понадобится распустить школьный класс, для этого необязательно удалять ученика, который может посещать другие классы.

Композиция представляет собой еще более сильное отношение, чем описанное выше. В композиции на содержащийся объект может ссылаться только содержащий его объект-контейнер. И он должен быть удален вместе с удаляемым объектом-контейнером. Отношения композиции обозначаются таким же образом, как и отношения агрегирования, только с заполненным ромбом. Пример отношения композиции наглядно показан на рис. 6.15.

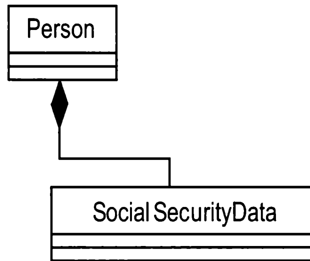


Рис. 6.15. *Пример композиции*

В классе Person имеется постоянная ссылка на объект типа SocialSecurityData. Экземпляр этого объекта может принадлежать только содержащему его объекту типа Person.

Описание отношений использования

В языке UML отношение использования описывается в виде зависимости. Это самое неустойчивое из всех отношений, рассматриваемых в данном разделе, потому что оно не описывает постоянную связь между классами. Используемый класс может передаваться в качестве аргумента или быть получен в результате вызова метода.

В классе Report, приведенном на рис. 6.16, используется объект типа ShopProductWriter. Отношение использования обозначается пунктирной линией со стрелкой, имеющей незамкнутый контур на конце. Эта линия соединяет рассматриваемые здесь класс и объект. Но при таком отношении ссылка на объект не хранится в виде свойства, как, например, в объекте типа ShopProductWriter, в котором хранится массив ссылок на объекты типа ShopProduct.

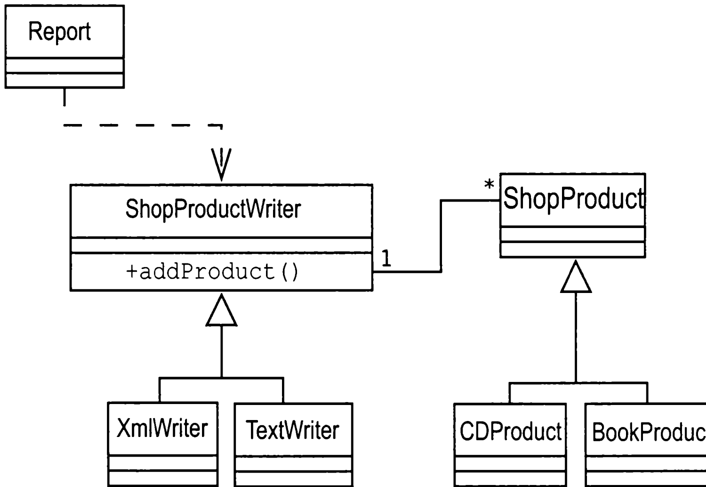


Рис. 6.16. *Пример отношения использования*

Использование примечаний

На диаграммах классов отображается структура системы, но они не дают достаточного представления о самом процессе. Как следует из рис. 6.16, в объекте типа `Report` используется объект типа `ShopProductWriter`, но механизм этого процесса неизвестен. На рис. 6.17 используются примечания, которые помогают немного прояснить эту ситуацию.

Как видите, примечание обозначается на диаграмме прямоугольником с уголком, загнутым справа вверху. Обычно в примечании содержатся фрагменты псевдокода.

Примечание вносит некоторую ясность в рассматриваемую здесь диаграмму классов. Теперь мы видим, что объект типа `ShopProductWriter` используется в объекте типа `Report` для вывода данных о товаре. На самом деле отношения использования не всегда настолько очевидны. В некоторых случаях даже примечание может не предоставить достаточной информации. Правда, взаимодействие объектов в проектируемой системе, а также структуру классов можно смоделировать.

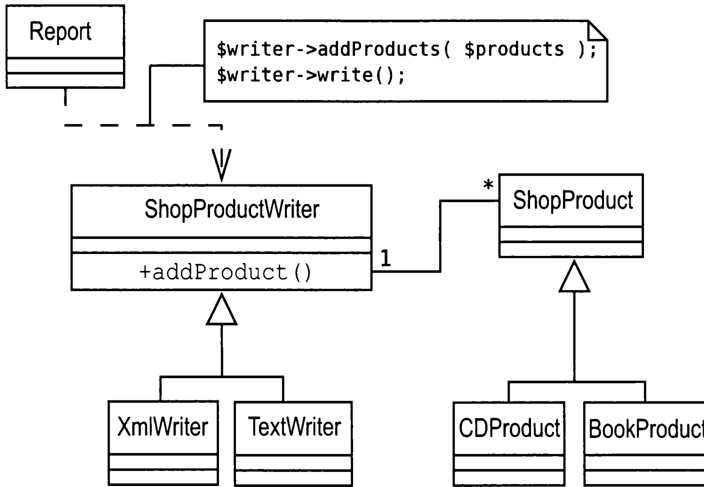


Рис. 6.17. *Использование примечаний для прояснения зависимости*

Диаграмма последовательностей

В основе *диаграммы последовательности* лежит объект, а не класс. Она служит для поэтапного моделирования процесса в проектируемой системе.

Составим простую диаграмму последовательности, моделируя порядок вывода данных о продукте из объекта типа `Report`. На диаграмме последовательности объекты из проектируемой системы представлены справа налево, как показано на рис. 6.18.

Рис. 6.18. *Пример представления объектов из проектируемой системы на диаграмме последовательности*

Объекты обозначены на этой диаграмме только именем класса. Если бы имелось несколько действующих независимо экземпляров одного класса, имя каждого объекта следовало бы указать в формате *`метка:класс`* (например, `product1:ShopProduct`).

Время жизни моделируемого процесса отображается на диаграмме последовательности сверху вниз, как показано на рис. 6.19.

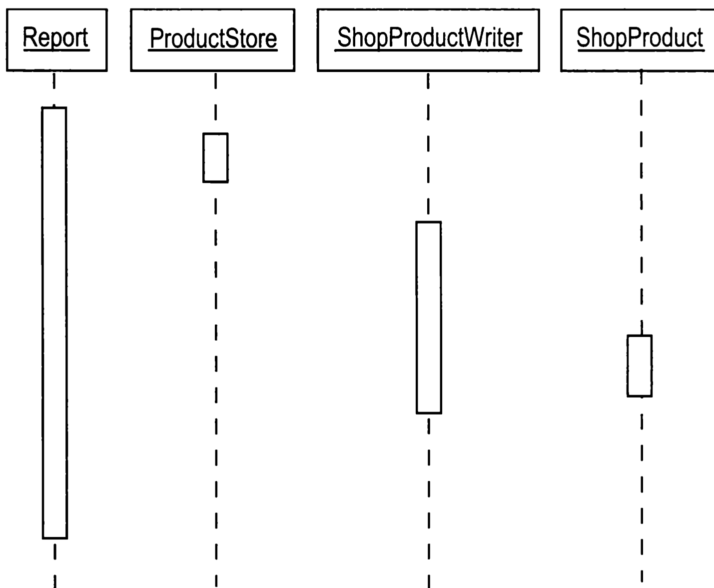


Рис. 6.19. *Пример обозначения линий жизни объектов на диаграмме последовательности*

Вертикальными пунктирными линиями на этой диаграмме обозначается время существования объектов в системе, а прямоугольниками на линиях жизни — направленность процесса. Если просмотреть рис. 6.19 сверху вниз, то можно увидеть, как процесс продвигается по объектам в системе. Но это трудно понять без сообщений, которыми обмениваются объекты. Такие сообщения добавлены на рис. 6.20.

Стрелками обозначены сообщения, которыми обмениваются объекты. Возвращаемые значения зачастую остаются неявными, хотя они могут быть обозначены пунктирной линией, идущей от вызываемого объекта к объекту, отправляющему сообщение. Каждое сообщение помечается вызовом соответствующего метода. Для меток имеется свой синтаксис, в котором квадратные скобки обозначают заданное условие. Так, запись

```
[okToPrint]
write()
```

означает, что вызов метода `write()` может произойти только при выполнении указанного условия. Знак звездочки служит для того, чтобы обозна-

чить повторение, как показано ниже. А далее может последовать соответствующее пояснение в квадратных скобках, хотя это и необязательно:

```
*[для каждого ShopProduct]
write()
```

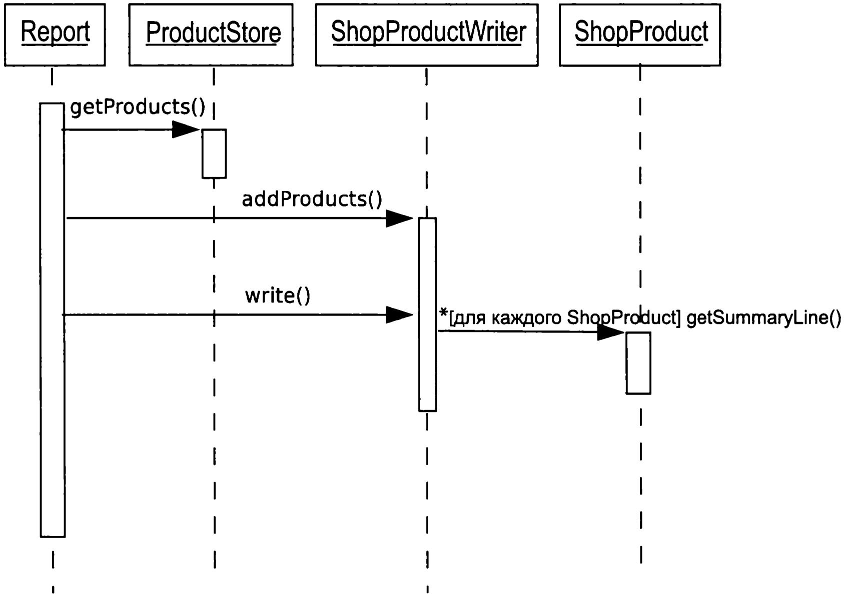


Рис. 6.20. *Пример завершенной диаграммы последовательности*

Теперь диаграмму последовательности, приведенную на рис. 6.20, можно интерпретировать сверху вниз. В частности, объект типа `Report` получает список объектов типа `ShopProduct` из объекта типа `ProductStore`. Этот объект передает полученный список объекту типа `ShopProductWriter`, в котором сохраняются ссылки на данные объекты, хотя, глядя на диаграмму последовательности, об этом можно только догадываться. Объект типа `ShopProductWriter` вызывает метод `ShopProduct::getSummaryLine()` для каждого объекта типа `ShopProduct`, ссылка на который хранится в массиве, добавляя полученный результат к выводимым данным.

Как видите, диаграммы последовательностей позволяют моделировать процессы, фиксируя срезы динамического взаимодействия и представляя их с необыкновенной ясностью.

На заметку Посмотрите внимательно на рис. 6.16 и 6.20. Обратите внимание, как на диаграмме класса проиллюстрирован полиморфизм (наглядно показаны классы, которые происходят от классов ShopProductWriter и ShopProduct). А теперь посмотрите, насколько этот факт становится очевидным, когда мы моделируем взаимодействие между объектами. Нам требуется, чтобы объекты работали по возможности с самыми общими из имеющихся типов данных и чтобы можно было скрыть подробности их реализации.

Резюме

В этой главе мы немного отошли от конкретных особенностей объектно-ориентированного программирования и рассмотрели некоторые важные вопросы проектирования. В частности, мы исследовали такие понятия, как инкапсуляция, связность, слабая и тесная связь, которые являются важными аспектами проектирования гибкой и повторно используемой объектно-ориентированной системы. Мы также рассмотрели средства языка UML как очень важное основание для работы с шаблонами, о которых речь пойдет далее в этой книге.

Часть II

Проектные шаблоны

ГЛАВА 7

Назначение и применение проектных шаблонов

Большинство задач, которые нередко приходится решать программистам, уже давно решены другими членами их сообщества. Проектные шаблоны (patterns; иногда их так и называют — "паттерн") как раз и являются тем средством, с помощью которого люди могут делиться друг с другом накопленным опытом. Как только шаблон становится всеобщим достоянием, он обогащает наш лексикон и позволяет легко обмениваться с другими новыми идеями проектирования и последствиями их осуществления. С помощью проектных шаблонов легко выделяются общие задачи, определяются проверенные решения и описываются вероятные результаты. Во многих книгах и статьях основное внимание уделяется особенностям конкретных языков программирования, имеющимся функциям, классам и методам. А в каталогах шаблонов, наоборот, акцент делается на том, как перейти в своих проектах от этих основ ("Что делать?") к осмыслению задач и возможных решений ("Зачем и как делать?").

В этой главе будут представлены проектные шаблоны и проанализированы некоторые причины их широкой распространенности. В ней, в частности, будут рассмотрены следующие вопросы.

- *Основы шаблонов.* Что такое проектные шаблоны?
- *Структура шаблона.* Каковы основные элементы проектного шаблона?
- *Преимущества шаблонов.* Почему шаблоны стоят того, чтобы их изучать?

Что такое проектные шаблоны

В сфере программного обеспечения проектный шаблон — это реальное проявление наследственной памяти организации.

— Гради Буч (Grady Booch), из книги *Core J2EE Patterns*[1]

Проектный шаблон — это решение задачи в определенном контексте.

— "Банда четырех" (The Gang of Four), из книги *Design Patterns: Elements of Reusable Object-Oriented Software*[2]

Как следует из приведенных выше цитат, проектный шаблон обеспечивает анализ определенной задачи и описывает хорошие практики ее решения.

Задачи имеют свойство повторяться, и веб-программистам приходится решать их снова и снова. В частности, как обработать входящий запрос? Как преобразовать данные в команды для проектируемой системы? Как запросить данные? Как представить результаты? Со временем мы находим более или менее изящные ответы на все эти вопросы и создаем неформальный набор методик, которые затем используем снова и снова в своих проектах. Эти методики и есть проектные шаблоны.

С помощью проектных шаблонов описываются и формализуются типовые задачи и их решения. В результате опыт, который нарабатывается с большим трудом, становится доступным широкому сообществу программистов. Шаблоны должны быть построены главным образом по восходящему, а не нисходящему принципу. Они укоренены в практике, а не в теории. Но это совсем не означает, что в проектных шаблонах отсутствует элемент теории, как будет показано в следующей главе. Шаблоны основаны на реальных методах, применяемых программистами на практике. Известный приверженец проектных шаблонов Мартин Фаулер (Martin Fowler) говорит, что он открывает шаблоны, а не создает их. Поэтому многие шаблоны будут вызывать у вас ощущение уже пройденного и виденного ранее, ведь вы будете узнавать в них методики, которыми пользуетесь сами.

Каталог шаблонов — это не книга кулинарных рецептов. Рецептам можно следовать буквально, а код можно скопировать и вставить в проект с

---

[1] Алур, Д., Крупи, Дж., Малкс, Д. *Образцы J2EE. Лучшие решения и стратегии проектирования* : Пер. с англ. — Изд-во "Лори", 2013.

[2] Гамма, Э., Хелм, Р., Джонсон, Р., Влиссидес, Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* : Пер. с англ. — Изд-во "Питер", 2007.

незначительными изменениями. Понимать весь код, используемый в рецепте, обязательно не всегда. А проектные шаблоны описывают *подходы* к решению конкретных задач. Подробности реализации могут существенно изменяться в зависимости от более широкого контекста. От этого контекста зависят выбор используемого языка программирования, характер приложения, масштабы проекта и особенности решения конкретной задачи.

Допустим, в проекте требуется создать систему шаблонной обработки. На основании имени файла шаблона необходимо синтаксически проанализировать его содержимое и построить дерево объектов, представляющих обнаруженные дескрипторы разметки.

Сначала синтаксический анализатор просматривает текст на предмет поиска триггерных лексем. Обнаружив соответствие, он передает лексему другому объекту-анализатору, который специализируется на чтении содержимого дескрипторов разметки. В итоге данные шаблона продолжают анализироваться до тех пор, пока не произойдет синтаксическая ошибка, будет достигнут их конец или найдена другая триггерная лексема. Обнаружив такую лексему, объект-анализатор должен также передать ее на обработку соответствующей программе — скорее всего, анализатору аргументов. Все вместе эти компоненты образуют так называемый рекурсивный нисходящий синтаксический анализатор.

Следовательно, участниками проектируемой системы можно назвать следующие классы: `MainParser`, `TagParser` и `ArgumentParser`. Потребуется также класс `ParserFactory`, создающий и возвращающий эти объекты.

Но все, конечно, идет не так гладко, как хотелось бы, и в дальнейшем на одном из совещаний выясняется, что в шаблонах нужно поддерживать несколько синтаксисов. И теперь нужно создать параллельный набор объектов-анализаторов в соответствии с конкретным синтаксисом: `OtherTagParser`, `OtherArgumentParser` и т.д.

Таким образом, поставлена следующая задача: формировать разные наборы объектов в зависимости от конкретной ситуации, и эти наборы объектов должны быть более или менее прозрачными для других компонентов системы. Но оказывается, что “Банда четырех” в своей книге определила следующую задачу для проектного шаблона `Abstract Factory` (Абстрактная фабрика): “Предусмотреть интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов”.

Этот проектный шаблон вполне подходит в данном случае! Сама суть поставленной задачи определяет и очерчивает рамки применения данного

шаблона. Но решение этой задачи не имеет ничего общего с операциями вырезания и вставки, как поясняется в главе 9, “Генерация объектов”, где подробно рассматривается проектный шаблон Abstract Factory.

Название шаблона уже само по себе очень ценно. Подобным образом создается нечто вроде общего словаря, который годами накапливается и используется в среде профессионалов. Такие условные обозначения помогают в совместных разработках, когда оцениваются и проверяются альтернативные подходы и разные последствия их применения. Например, при обсуждении семейств альтернативных синтаксических анализаторов достаточно сказать коллегам по работе, что в проектируемой системе каждый набор создается по проектному шаблону Abstract Factory. Одни кивнут головой с умным видом, кто-то сразу поймет, о чем идет речь, а кто-нибудь отметит про себя, что нужно будет ознакомиться с этим шаблоном позже. Но суть в том, что у такой системы набора понятий и разных результатов их воплощения имеется абстрактный описатель, способствующий созданию более кратких обозначений, как демонстрируется далее в этой главе.

И наконец, некорректно писать о проектных шаблонах, не процитировав Кристофера Александера (Christopher Alexander), профессора архитектуры, работы которого оказали огромное влияние на первых сторонников объектно-ориентированных проектных шаблонов. Вот что он пишет в своей книге *A Pattern Language* (Oxford University Press, 1977)[3].

> *Каждый проектный шаблон описывает задачу, которая возникает снова и снова, а затем описывает суть решения данной задачи, так что вы можете использовать это решение миллион раз, всякий раз делая это по-другому.*

Очень важно, что это определение, хотя оно относится к архитектурным задачам и решениям, начинается с формулировки задачи и ее более широкого контекста и переходит к решению. За последние годы прозвучало немало критики по поводу того, что проектными шаблонами злоупотребляют, особенно неопытные программисты. Причина в том, что решения применялись там, где не была сформулирована задача и отсутствовал соответствующий контекст. Шаблоны — это нечто большее, чем конкретная организация классов и объектов, совместно работающих определенным

[3] Александер, К., Исикава, С., Силверстайн, М. *Язык шаблонов. Города. Здания. Строительство* : Пер. с англ. — Изд-во студии А. Лебедева, 2014.

образом. Шаблоны создаются для определения условий, в которых должны применяться решения, а также для обсуждения результатов этих решений.

В данной книге основной акцент сделан на особенно влиятельном направлении в сфере проектных шаблонов: форме, описанной в упоминавшейся в начале этой главы книге *Design Patterns: Elements of Reusable Object-Oriented Software*. Она посвящена применению проектных шаблонов при разработке объектно-ориентированного программного обеспечения, и в ней описываются некоторые классические шаблоны, которые имеются в большинстве современных объектно-ориентированных проектов.

Эта книга "Банды четырех" очень важна не только потому, что в ней описываются основные проектные шаблоны, но и потому, что в ней рассматриваются принципы проектирования, которые положены в основу этих шаблонов. Некоторые из этих принципов мы рассмотрим в следующей главе.

На заметку Проектные шаблоны, описанные "Бандой четырех" и в данной книге, служат настоящими примерами языка шаблонов, т.е. каталога задач и решений, объединенных вместе таким образом, чтобы они дополняли друг друга и формировали взаимозависимое целое. Существуют языки шаблонов для решения задач из других предметных областей, включая визуальное проектирование, управление проектами и, конечно, архитектуру. Но когда в данной книге обсуждаются проектные шаблоны, то имеются в виду задачи и решения в области разработки объектно-ориентированного программного обеспечения.

Краткий обзор проектных шаблонов

По существу, проектный шаблон состоит из четырех частей: названия, постановки задачи, описания ее решения и последствий.

Название

Выбор названия для шаблона очень важен. Несколькими краткими словами можно обозначить довольно сложные задачи и их решения, поэтому при выборе названий для проектных шаблонов следует соблюдать золотую середину между краткостью и описательностью. "Банда четырех" утверждает: "Поиск удачных названий был одной из самых трудных задач при разработке нашего каталога". И с этим согласен Мартин Фаулер, заявляя: "Названия для шаблонов чрезвычайно важны, потому что одна из це-

лей шаблонов — создать словарь, позволяющий разработчикам общаться более эффективно" (из книги *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002)[4].

В своей книге *Patterns of Enterprise Application Architecture* Мартин Фаулер совершенствует шаблон доступа к базе данных, описание которого я впервые обнаружил в упоминавшейся в начале этой главы книге *Core J2EE Patterns*. В частности, Фаулер определяет два шаблона, которые описывают специализацию старого шаблона. Логика такого подхода ясна и точна: один из новых шаблонов моделирует объекты предметной области, в то время как другой — таблицы базы данных, тогда как в предыдущем труде это различие было нечетким. Тем не менее заставить себя мыслить категориями новых шаблонов было нелегко. Я лично пользовался названием первоначального шаблона при проектировании и документировании так долго, что оно прочно вошло в мой лексикон.

Постановка задачи

Какими бы изящными ни были решения, а некоторые из них действительно очень изящны, постановка задачи и ее контекста служит основанием для проектного шаблона. Поставить задачу намного труднее, чем выбрать какое-нибудь решение из каталога шаблонов. И это одна из причин, по которым некоторые шаблоны могут применяться неправильно.

В шаблонах очень тщательно описывается предметная область решаемой задачи. Сначала кратко описывается сама задача, а затем — ее контекст. Обычно приводится типичный пример, дополняемый одной или несколькими диаграммами. Далее анализируются особенности задачи и различные ее проявления. Описываются также все признаки, которые могут помочь в постановке задачи.

Решение

Решение сначала кратко описывается вместе с задачей. Оно также описывается подробно, как правило, с помощью диаграмм классов и взаимодействия на языке UML. В шаблон обычно включается пример кода.

И хотя код может быть предоставлен, в качестве решения никогда нельзя использовать способ вырезания и вставки. Напомним, что шаблон описывает только подход к решению задачи, поскольку в его реали-

---

[4] Фаулер, М. *Шаблоны корпоративных приложений* : Пер. с англ. — ИД "Вильямс", 2009.

зации могут проявиться сотни нюансов. Рассмотрим в качестве примера инструкции, как сеять хлеб. Если вы слепо выполните все эти инструкции, то, скорее всего, будете голодать после сбора урожая. Намного более полезным оказывается основанный на шаблоне подход, в котором описываются различные условия применения этого шаблона. Основное решение задачи (вырастить хлеб) всегда будет одним и тем же (посеять семена, поливать, собрать урожай), но реальные шаги, которые нужно будет предпринять, зависят от самых разных факторов, включая тип почвы, местность, наличие вредных насекомых и т.д. Мартин Фаулер называет решения, описанные в шаблонах, "полуфабрикатами", т.е. программист должен взять основной замысел решения и самостоятельно довести его до логического завершения.

Следствия

Каждое решение, принимаемое при проектировании, будет иметь обширный ряд следствий. Безусловно, это должно быть удовлетворительное решение поставленной задачи. Однажды осуществленное решение может идеально подходить для работы и с другими шаблонами, но делать это следует с особой осторожностью.

Формат "Банды четырех"

Теперь, когда я пишу эти строки, передо мной на рабочем столе лежат пять каталогов проектных шаблонов. Даже поверхностный взгляд на шаблоны в каждом каталоге говорит о том, что ни в одном из них не используется такая же структура, как и в других. Одни из них более формализованы, другие сильно детализированы (содержат немало подразделов), а третьи менее упорядочены.

Существует ряд ясно определенных структур шаблонов, включая первоначальную форму, разработанную Кристофером Александером (александрийская форма), а также описательный подход, применяемый в Портлендском хранилище шаблонов (Portland Pattern Repository). А поскольку "Банда четырех" имеет большое влияние, и мы будем рассматривать многие из описанных ими шаблонов, исследуем несколько разделов, которые они включили в свои шаблоны. Ниже приведено краткое описание этих разделов.

- *Назначение.* Краткая формулировка цели шаблона. Необходимо уметь с первого взгляда понять суть шаблона.
- *Мотивация.* Задача описывается, как правило, для типичной ситуации. На конкретных примерах легче понять, что представляет собой шаблон.
- *Применимость.* Исследование различных ситуаций, в которых можно применить шаблон. В то время как в разделе мотивации описывается типичная задача, в данном разделе определяются конкретные ситуации и оцениваются преимущества решения в контексте каждой из них.
- *Структура/взаимодействие.* Эти разделы могут содержать диаграммы классов и взаимодействия, описывающие на языке UML отношения между классами и объекты в конкретном решении.
- *Реализация.* В данном разделе рассматриваются подробности решения. В нем исследуются любые вопросы, которые могут возникнуть при применении предлагаемой методики, а также даются советы по ее применению.
- *Пример кода.* Я всегда перехожу сразу же к этому разделу, поскольку считаю, что простой пример кода помогает лучше разобраться в шаблоне. Этот пример часто сведен к минимуму с целью продемонстрировать саму суть решения. Пример может быть написан на любом объектно-ориентированном языке. Разумеется, в следующих примерах кода всегда будет употребляться язык PHP.
- *Примеры применения.* Реальные системы, в которых встречается данный шаблон (задача, контекст и решение). Существует мнение, что для того, чтобы считаться настоящим, шаблон должен присутствовать по крайней мере в трех широко известных контекстах. Это называется иначе “правилом трех”.
- *Связанные шаблоны.* Одни шаблоны порождают другие. Применяя одно решение, можно создать контекст, в котором станет полезным другое решение. Именно такие связи исследуются в этом разделе, где могут также обсуждаться шаблоны, некоторые сходства в задаче и решении, а также любые предшествующие шаблоны, определенные где-нибудь еще и служащие основанием для построения текущего шаблона.

Причины для применения проектных шаблонов

Так в чем же преимущества шаблонов? Учитывая, что шаблон — это поставленная задача и описанное решение, ответ, казалось бы, очевиден. Шаблоны помогают решать распространенные задачи. Но шаблон — это, без сомнения, нечто большее.

Шаблоны определяют задачи

Сколько раз вы доходили до какой-то стадии в проекте и обнаруживали, что дальше идти некуда? Вполне вероятно, вам придется вернуться немного назад, прежде чем начать снова.

Определяя распространенные задачи, шаблоны помогают улучшить проект. А иногда первый шаг к решению — это осознание возникшего затруднения.

Шаблоны определяют решения

Определив и осознав возникшее затруднение (и убедившись, что оно действительно составляет проблему), вы получаете с помощью проектного шаблона доступ к решению, а также к анализу последствий его применения. И хотя шаблон совсем не избавляет вас от необходимости рассмотреть последствия выбранного решения, вы по крайней мере будете уверены, что пользуетесь проверенной методикой.

Шаблоны не зависят от языка программирования

Проектные шаблоны определяют объекты и решения в терминах объектно-ориентированного программирования. Это означает, что многие шаблоны одинаково применимы во многих языках программирования. Начав пользоваться проектными шаблонами, я изучал примеры кода на C++ и Smalltalk и реализовывал свои решения на Java. На другие языки шаблоны переносятся с некоторыми изменениями в их реализации или в получаемых результатах, но они всегда остаются правомерными. В любом случае проектные шаблоны помогают при переходе от одного языка к другому. Аналогично приложение, написанное на четких принципах объектно-ориентированного проектирования, легко перенести с одного языка

на другой (хотя при этом всегда остаются трудности, которые приходится разрешать).

Шаблоны определяют словарь

Предоставляя разработчикам названия методик, шаблоны обогащают процесс общения и передачи информации. Представим совещание, посвященное вопросам проектирования. Я уже описал мое решение с помощью шаблона Abstract Factory и теперь должен изложить стратегию обработки данных, которые собирает система. Я рассказываю о своих планах Бобу в следующем диалоге.

Я: Я собираюсь применить шаблон Composite.

Боб: Мне кажется, ты не продумал это как следует.

Что ж, Боб не согласен со мной. Он всегда со мной не согласен. Но он знает, о чем я говорю и почему моя идея плоха. А теперь проиграем эту сцену еще раз без применения словаря.

Я: Я собираюсь использовать три объекта с одним и тем же общим типом данных. В интерфейсе этого типа данных будут определены методы для добавления дочерних объектов собственного типа. Таким образом, мы можем построить сложную комбинацию реализованных объектов во время выполнения программы.

Боб: Точно?

Шаблоны или методики, которые в них описаны, имеют тенденцию к взаимодействию. Так, шаблон Composite взаимодействует с шаблоном Visitor.

Я: А затем можно воспользоваться шаблоном Visitor для получения итоговых данных.

Боб: Ты упустил суть.

Не будем обращать внимание на Боба, долго и мучительно описывая версию этого разговора без употребления терминологии проектных шаблонов. Подробнее о шаблоне `Composite` речь пойдет в главе 10, “Шаблоны для программирования гибких объектов”, а о шаблоне `Visitor` — в главе 11, “Выполнение задач и представление результатов”.

Дело в том, что эти методики можно применять и без терминологии проектных шаблонов. Сами методики всегда появляются *до* того, как им будет присвоено название, и они будут организованы в виде шаблона. Если же такой шаблон отсутствует, его можно создать. А любое инструментальное средство, которое применяется достаточно часто, в конце концов получит свое название.

Шаблоны проверяются и тестируются

Итак, если с помощью проектных шаблонов описываются нормы передовой практики, то будет ли выбор названия самым важным элементом при создании каталогов шаблонов? В каком-то смысле будет. Шаблоны как нормы передовой практики объектно-ориентированного программирования могут показаться некоторым очень опытным программистам упражнением в выражении очевидного иными словами. Но для всех остальных шаблоны предоставляют доступ к задачам и решениям, которые в противном случае приходилось бы находить нелегким путем.

Шаблоны делают проектирование более доступным. Каталоги шаблонов появляются для все большего количества специализированных областей, поэтому даже очень опытный специалист может извлечь из них пользу. Например, разработчик графического пользовательского интерфейса может быстро получить доступ к задачам и решениям при создании корпоративного приложения, а разработчик веб-приложения — быстро наметить стратегию, как избежать ошибок и подводных камней, которые могут возникнуть в проектах для планшетных компьютеров и смартфонов.

Шаблоны предназначены для совместной работы

По своему характеру проектные шаблоны должны быть порождающими и составляемыми. Это означает возможность применить один шаблон и тем самым создать условия, подходящие для применения другого шаблона. Иначе говоря, используя шаблон, можно обнаружить открывающиеся новые перспективы. Каталоги шаблонов обычно разрабатываются с расчетом на такого рода совместную работу, а возможность составления шаблонов всегда документируется в самом шаблоне.

Шаблоны способствуют удачным проектам

В проектных шаблонах демонстрируются и применяются принципы объектно-ориентированного проектирования. Поэтому изучение проектных шаблонов может дать больше, чем конкретное решение в определенном контексте. Это может привести к новому представлению, как объединить объекты и классы для достижения поставленной цели.

Шаблоны применяются в распространенных каркасах

В данной книге процесс проектирования описан практически с самого начала. Принципы и проектные шаблоны, рассмотренные в ней, должны побудить вас к созданию ряда собственных базовых каркасов, которые лягут в основу всех ваших проектов. Ведь лень иногда оказывается во благо, и поэтому вы можете вполне воспользоваться такими стандартными каркасами, как Zend, Code Igniter или Symfony, или же скопировать код из готовых проектов. Ясное понимание основ проектных шаблонов поможет вам быстро освоить программный интерфейс API этих каркасов.

PHP и проектные шаблоны

В этой главе мало что относится непосредственно к PHP, что в некоторой степени характерно для рассматриваемой в ней темы. Большинство проектных шаблонов применимо во многих языках программирования, в которых имеются возможности манипулирования объектами и достаточно решить лишь некоторые вопросы реализации, если они вообще возникают.

Но это, конечно, не всегда именно так. Некоторые шаблоны корпоративных приложений отлично подходят для тех языков, в которых прикладной процесс не останавливает свою работу в перерывах между запросами к серверу. Но сценарии на PHP действуют иначе, поскольку для обработки каждого поступившего запроса сценарий запускается заново. Это означает, что некоторые шаблоны требуют более аккуратного обращения. Например, для реализации шаблона Front Controller часто требуется довольно много времени. Хорошо, если инициализация имеет место один раз при запуске приложения, но гораздо хуже, если она происходит при каждом запросе. Но это совсем не означает, что применять данный шаблон нельзя.

В своей практике я применял его с очень хорошими результатами. При обсуждении шаблона мы просто должны принять во внимание вопросы, связанные с PHP. Именно язык PHP формирует контекст для всех шаблонов, которые исследуются в данной книге.

В этом разделе уже упоминались языки программирования, в которых имеется возможность манипулировать объектами. А в PHP можно программировать, вообще не определяя никаких классов. Но объекты и принципы ООП положены, за некоторыми исключениями, в основу большинства проектов и библиотек на PHP.

Резюме

В этой главе были представлены проектные шаблоны, рассмотрена их структура (с помощью формата "Банды четырех") и проанализирован ряд причин, по которым у вас может возникнуть необходимость пользоваться проектными шаблонами в своих сценариях.

Не следует, однако, забывать, что проектные шаблоны — это не готовые решения, которые можно объединять, как компоненты, при создании проекта. Шаблоны — это предлагаемые подходы к решению распространенных задач. В этих решениях воплощены некоторые основные принципы проектирования. Именно их мы и будем исследовать в следующей главе.

ГЛАВА 8

Некоторые принципы проектных шаблонов

Несмотря на то что проектные шаблоны просто описывают решения задач, в них обычно делается акцент на решениях, которые способствуют их повторному использованию и гибкости. Для достижения этих целей в шаблонах реализуется ряд основных принципов объектно-ориентированного проектирования. Некоторые из этих принципов будут исследованы в этой главе, а более подробно проектные шаблоны будут рассматриваться на протяжении остальной части данной книги.

В этой главе рассматриваются следующие принципы действия шаблонов.

- *Композиция.* Использование агрегирования объектов для достижения большей гибкости, чем с помощью одного только наследования.
- *Развязка.* Сокращение взаимной зависимости элементов в системе.
- *Потенциальные возможности интерфейса.* Шаблоны и полиморфизм.
- *Категории шаблонов.* Типы шаблонов, описываемых в данной книге.

Открытие шаблонов

Я начал работать с объектами в языке Java. Как и следовало ожидать, прошло некоторое время, прежде чем какие-то идеи пригодились и воплотились. Причем произошло это очень быстро, почти как откровение. Изящество принципов наследования и инкапсуляции меня просто поразило. Я почувствовал, что это другой способ определения и построения систем. Я *пользовался* полиморфизмом при работе с типами данных и переключении между реализациями во время выполнения программы. Мне казалось, что эти знания и понимание помогут решить большинство моих проектных задач и позволят создавать прекрасные и изящные системы.

В то время все книги на моем рабочем столе были посвящены возможностям языков программирования и многочисленным интерфейсам API,

доступным программисту на Java. Если не считать краткого определения полиморфизма, в этих книгах почти не было попыток исследовать стратегии проектирования.

Одни только возможности языка не порождают объектно-ориентированные проекты. Хотя мои проекты удовлетворяли поставленным функциональным требованиям, тип проекта, который можно было создать с помощью наследования, инкапсуляции и полиморфизма, по-прежнему ускользал от меня.

Мои иерархии наследования становились все более обширными и глубокими, поскольку я пытался создавать новые классы по каждому поводу. Структура моих систем затрудняла передачу сообщений от одного уровня к другому таким образом, чтобы не давать промежуточным классам слишком много сведений об их окружении, не привязывать их к приложению и не делать их непригодными в новом контексте.

И только открыв для себя упоминавшуюся ранее книгу *Design Patterns*, которая иначе называется книгой "Банды четырех", я понял, что упустил целый аспект проектирования. К тому времени я уже самостоятельно открыл для себя несколько основных шаблонов, но знакомство с другими шаблонами изменило мой образ мышления.

Я обнаружил, что, пытаясь внести слишком большую функциональность в свои классы, я переоценил наследование. Но где еще можно разместить функциональные возможности в объектно-ориентированной системе?

Ответ я нашел в композиции. Программные компоненты можно определять во время выполнения путем комбинирования объектов, находящихся в гибких отношениях. "Банда четырех" сформулировала это в следующем принципе: "Предпочитайте композицию наследованию". Шаблоны описывали способы объединения объектов во время выполнения программы с целью достичь уровня гибкости, который невозможен при применении только одного дерева наследования.

Композиция и наследование

Наследование — это эффективный способ описания меняющихся обстоятельств или контекста. Но из-за этого можно потерять в гибкости, особенно если на классы возложено несколько функциональных обязанностей.

Проблема

Как вам, должно быть, известно, дочерние классы наследуют методы и свойства родительских классов, при условии, что они относятся к открытым или защищенным элементам. Этим фактом можно воспользоваться для создания дочерних классов, обладающих особыми функциональными возможностями.

На рис. 8.1 приведен простой пример наследования, демонстрируемого с помощью диаграммы UML.

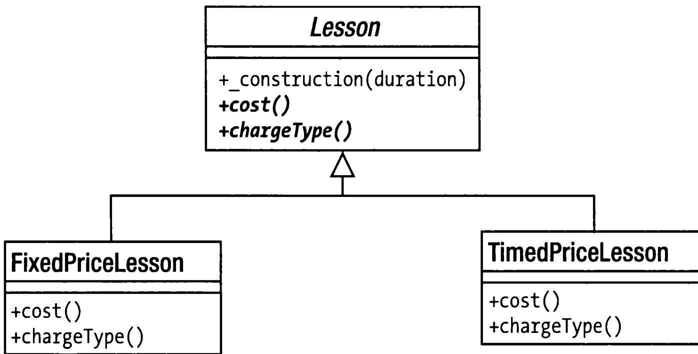


Рис. 8.1. *Родительский класс и два дочерних класса*

Абстрактный класс `Lesson` на рис. 8.1 моделирует обучение в колледже и определяет абстрактные методы `cost()` и `chargeType()`. На диаграмме показаны два реализующих их класса, `FixedPriceLesson` и `TimedPriceLesson`, которые обеспечивают разные механизмы оплаты занятий.

С помощью этой схемы наследования можно без особого труда изменить реализацию обучения. Клиентскому коду будет известно лишь то, что он имеет дело с объектом типа `Lesson`, поэтому подробности механизма оплаты будут прозрачными.

Но что произойдет, если потребуется внедрить новый ряд специализаций? Допустим, потребуется работать с такими элементами, как лекции и семинары. Они подразумевают разные способы регистрации учащихся и создания рабочих материалов к занятиям, для них нужны отдельные классы. Теперь у нас есть две силы, работающие с моим проектом. Мне нужно обрабатывать стратегии оплаты и разделять лекции и семинары.

На рис. 8.2 показано решение задачи "в лоб".

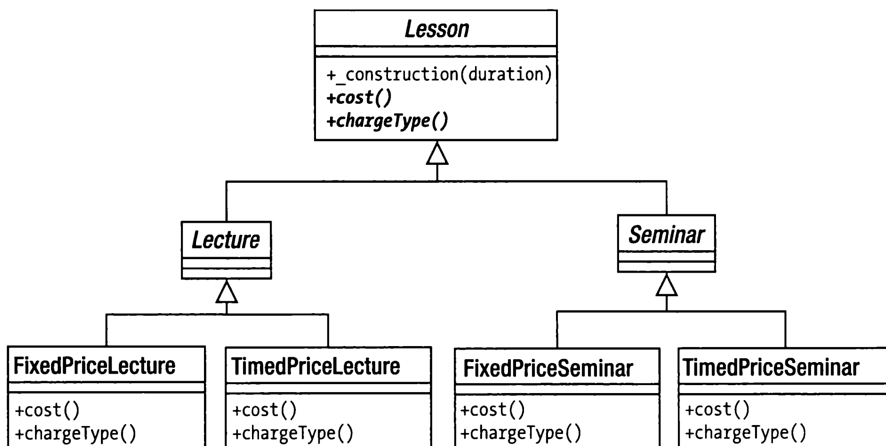


Рис. 8.2. *Неудачная структура наследования*

На рис. 8.2 приведена явно неудачная иерархия. Такое дерево наследования нельзя в дальнейшем использовать для того, чтобы управлять механизмами оплаты, не дублируя большие блоки функциональных средств. Эти механизмы оплаты повторяются в семействах классов `Lecture` и `Seminar`.

На этом этапе необходимо рассмотреть применение условных инструкций в суперклассе `Lesson`, чтобы избавиться от дублирования. В сущности, мы полностью удаляем логику оплаты из дерева наследования, перемещая ее вверх по иерархии в суперкласс. Это полная противоположность традиционному рефакторингу, когда условные инструкции заменяются полиморфизмом. Ниже показано, как выглядит исправленный в итоге класс `Lesson`:

```
// Листинг 8.1
abstract class Lesson
{
    public const FIXED = 1;
    public const TIMED = 2;
    public function __construct(protected int $duration,
                                private int $costtype = 1)
    {
    }
    public function cost(): int
    {
```

```
        switch ($this->costtype)
        {
            case self::TIMED:
                return (5 * $this->duration);
                break;

            case self::FIXED:
                return 30;
                break;

            default:
                $this->costtype = self::FIXED;
                return 30;
        }
    }
    public function chargeType(): string
    {
        switch ($this->costtype)
        {
            case self::TIMED:
                return "Почасовая оплата";
                break;

            case self::FIXED:
                return "Фиксированная ставка";
                break;

            default:
                $this->costtype = self::FIXED;
                return "Фиксированная ставка";
        }
    }
    // Другие методы класса...
}

// Листинг 8.2
class Lecture extends Lesson
{
    // Реализации, специфичные для класса Lecture...
}

// Листинг 8.3
class Seminar extends Lesson
{
    // Реализации, специфичные для класса Seminar...
}
```

Вот как можно работать с этими классами:

```
// Листинг 8.4
$lecture = new Lecture(5, Lesson::FIXED);
print "{$lecture->cost()} ({$lecture->chargeType()})\n";
$seminar = new Seminar(3, Lesson::TIMED);
print "{$seminar->cost()} ({$seminar->chargeType()})\n";
```

В итоге мы получим следующий вывод:

```
30 (Фиксированная ставка)
15 (Почасовая оплата)
```

Диаграмма нового класса показана на рис. 8.3.

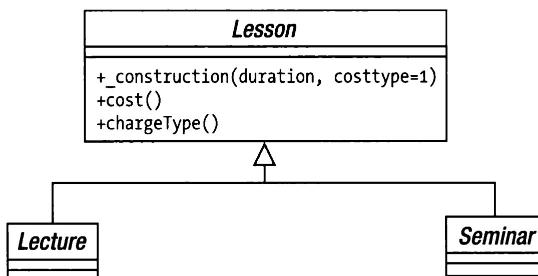


Рис. 8.3. *Иерархия наследования улучшена в результате удаления расчетов стоимости занятий из подклассов*

Мы сделали структуру класса намного более управляемой, хотя и дорогой ценой. Употребление условных инструкций в рассматриваемом здесь коде — это шаг назад. Как правило, условную инструкцию лучше заменить полиморфизмом, а здесь мы поступили наоборот. Как видите, это вынудило нас продублировать условную инструкцию в методах `chargeType()` и `cost()`. Похоже, что мы обречены на дублирование кода.

Применение композиции

Для решения данной задачи можно воспользоваться шаблоном Strategy, который служит для перемещения ряда алгоритмов в отдельный тип данных. Перемещая код для расчета стоимости занятия в другой класс, можно упростить тип `Lesson` (рис. 8.4).

Мы создали еще один абстрактный класс, `CostStrategy`, в котором определены абстрактные методы `cost()` и `chargeType()`. Методу `cost()` необходимо передать экземпляр класса `Lesson`, который он будет исполь-

зовать для расчета стоимости занятия. Мы предоставляем две реализации класса CostStrategy. Объекты типа Lesson оперируют только типом CostStrategy, а не конкретной реализацией, поэтому можно в любое время добавить новые алгоритмы расчета стоимости, создавая подклассы на основе класса CostStrategy. И для этого не придется вносить вообще никаких изменений в классы Lesson.

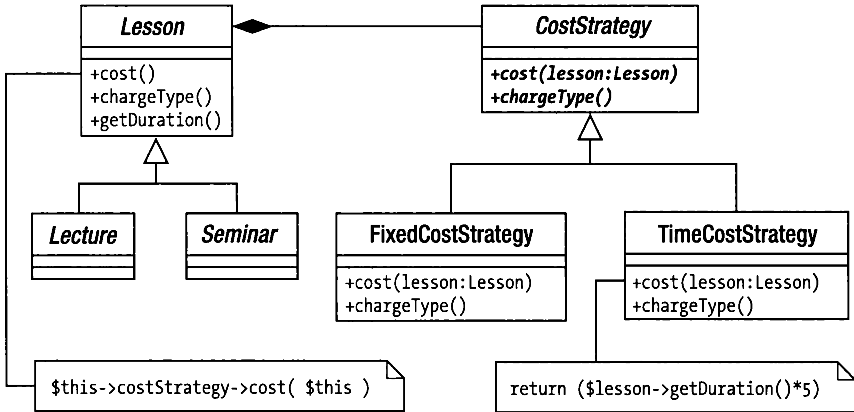


Рис. 8.4. *Перемещение алгоритмов в отдельный тип данных*

Приведем упрощенную версию нового класса Lesson, показанного на рис. 8.4:

```
// Листинг 8.5
abstract class Lesson
{
    public function __construct(private int $duration,
                                private CostStrategy
                                $costStrategy)
    {
    }
    public function cost(): int
    {
        return $this->costStrategy->cost($this);
    }
    public function chargeType(): string
    {
        return $this->costStrategy->chargeType();
    }
    public function getDuration(): int
    {
        return $this->duration;
```

```
    }
    // Другие методы Lesson...
}

// Листинг 8.6
class Lecture extends Lesson
{
    // Реализации, специфичные для класса Lecture...
}

// Листинг 8.7
class Seminar extends Lesson
{
    // Реализации, специфичные для класса Seminar...
}
```

Конструктору класса `Lesson` передается объект типа `CostStrategy`, который он сохраняет в виде свойства. В методе `Lesson::cost()` просто делается вызов `CostStrategy::cost()`. Аналогично в методе `Lesson::chargeType()` делается вызов `CostStrategy::chargeType()`. Такой явный вызов метода из другого объекта для выполнения запроса называется делегированием. В рассматриваемом здесь примере объект типа `CostStrategy` является делегатом класса `Lesson`. А класс `Lesson` снимает с себя ответственность за расчет стоимости занятия и возлагает эту обязанность на реализацию класса `CostStrategy`. Ниже показано, каким образом осуществляется делегирование:

```
// Листинг 8.8
public function cost(): int
{
    return $this->costStrategy->cost($this);
}
```

А вот определение класса `CostStrategy` вместе с реализующими его дочерними классами:

```
// Листинг 8.9
abstract class CostStrategy
{
    abstract public function cost(Lesson $lesson): int;
    abstract public function chargeType(): string;
}

// Листинг 8.10
class TimedCostStrategy extends CostStrategy
{
```

```
    public function cost(Lesson $lesson): int
    {
        return ($lesson->getDuration() * 5);
    }
    public function chargeType(): string
    {
        return "Почасовая оплата";
    }
}

// Листинг 8.11
class FixedCostStrategy extends CostStrategy
{
    public function cost(Lesson $lesson): int
    {
        return 30;
    }
    public function chargeType(): string
    {
        return "Фиксированная ставка";
    }
}
```

Теперь во время выполнения программы можно легко изменить способ расчета стоимости занятий, выполняемый любым объектом типа `Lesson`, передав ему другой объект типа `CostStrategy`. Такой подход способствует созданию довольно гибкого кода. Вместо того чтобы статично встраивать функциональность в структуры кода, можно комбинировать объекты и менять их сочетания динамически:

```
// Листинг 8.12

$lessons[] = new Seminar(4, new TimedCostStrategy());
$lessons[] = new Lecture(4, new FixedCostStrategy());

foreach ($lessons as $lesson) {
  print "Оплата за занятие {$lesson->cost()}. ";
  print " Тип оплаты: {$lesson->chargeType()}\n";
}
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Оплата за занятие 20. Тип оплаты: Почасовая оплата
Оплата за занятие 30. Тип оплаты: Фиксированная ставка
```

Как видите, одно из следствий принятия такой структуры состоит в том, что мы распределили обязанности классов. Объекты типа `CostStrategy` ответственны только за расчет стоимости занятия, а объекты типа `Lesson` управляют данными занятия.

Итак, композиция позволяет сделать код намного более гибким, поскольку можно комбинировать объекты и решать задачи динамически намного бо́льшим количеством способов, чем при использовании одной лишь иерархии наследования. Но при этом исходный код может стать неудобочитаемым. В результате композиции, как правило, создается больше типов данных с отношениями, которые не настолько предсказуемы, как отношения наследования. Поэтому понять отношения в такой системе немного труднее.

Развязка

Как пояснялось в главе 6, "Объекты и проектирование", имеет смысл создавать независимые компоненты, поскольку систему, состоящую из зависимых классов, намного труднее сопровождать. Дело в том, что внесение изменений в одном месте программы может повлечь за собой ряд соответствующих изменений в других частях кода программы.

Проблема

Повторное использование — одна из основных целей объектно-ориентированного проектирования, а тесная связь вредит этой цели. Тесная связь возникает, когда изменение в одном компоненте системы ведет к необходимости вносить множество изменений повсюду. Необходимо стремиться создавать независимые компоненты, чтобы можно было вносить изменения, не опасаясь эффекта цепной реакции непредвиденных последствий. Когда изменяется компонент, степень его независимости влияет на вероятность того, что эти изменения вызовут ошибки в других частях системы.

На рис. 8.2 был наглядно показан пример тесной связи. В частности, логика схем оплаты занятий повторяется в типах `Lecture` и `Seminar`, и поэтому изменения в компоненте типа `TimedPriceLecture` приведут к необходимости внесения параллельных изменений в ту же самую логику в компоненте типа `TimedPriceSeminar`. Обновляя один класс и не обновляя другой, мы нарушаем нормальную работу системы. При этом интер-

претатор PHP не выдает никакого предупреждения. Поэтому наше первое решение употребить условную инструкцию породило аналогичную зависимость между методами `cost()` и `chargeType()`.

Применяя шаблон Strategy, мы преобразовали алгоритмы оплаты занятий в тип `CostStrategy`, разместили их за общим интерфейсом и реализовали каждый из них только один раз. Тесная связь другого рода может возникнуть в том случае, если в системе многие классы внедрены явным образом в платформу или в среду. Допустим, требуется создать систему, которая работает с базой данных MySQL. Для запросов к серверу базы данных можно вызвать метод `mysql::query()`.

Но если же понадобится развернуть систему на сервере, который не поддерживает базу данных MySQL, для этого придется внести изменения в весь проект, чтобы использовать базу данных, например, SQLite. При этом изменения придется вносить по всему коду, а это сулит неприятную перспективу поддерживать две параллельные версии приложения.

И дело здесь не в зависимости системы от внешней платформы. Такая зависимость неизбежна, поскольку приходится работать с кодом, который связывается с базой данных. Проблема возникает в том случае, когда такой код разбросан по всему проекту. Взаимодействие с базами данных — это не главная обязанность большинства классов в системе, поэтому наилучшая стратегия состоит в том, чтобы выделить такой код и сгруппировать его за общим интерфейсом. И это будет способствовать независимости классов. В то же время собирание кода шлюза в одном месте создает условия для более легкого перехода на новую платформу, где не придется вносить изменения в остальную часть системы. Такой процесс сокрытия реализации за ясным интерфейсом называется *инкапсуляцией*. Данную проблему позволяет разрешить библиотека баз данных Doctrine из проекта `DBAL` (Database Abstraction Layer — абстрактный уровень базы данных), в которой предоставляется единственная точка доступа ко многим базам данных.

В частности, в классе `DriverManager` определен статический метод `getConnection()`, которому в качестве аргумента передается аргумент массива параметров. В соответствии со структурой этого массива данный метод возвращает конкретную реализацию интерфейса в классе `Doctrine\DBAL\Driver`. Структура этого класса приведена на рис. 8.5.

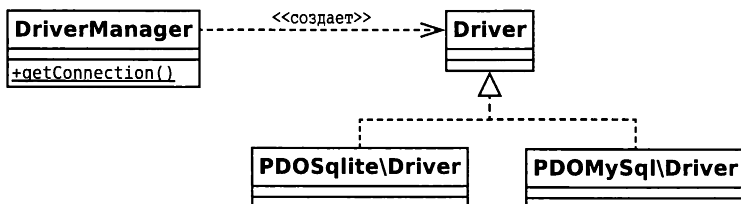


Рис. 8.5. *Пакет* `DBAL` *развязывает клиентский код от объектов базы данных*

На заметку Статические атрибуты и операции в UML должны быть подчеркнуты.

Пакет DBAL позволяет отвязать прикладной код от конкретных особенностей платформы базы данных. Это дает возможность организовать взаимодействие одной системы с MySQL, SQLite, MSSQL и прочими базами данных, не изменяя ни одной строки кода, разумеется, кроме параметров конфигурации.

Ослабление связанности

Чтобы сделать код взаимодействия с базой данных гибким и управляемым, необходимо развязать логику приложения от конкретных особенностей платформы базы данных. Такая развязка компонентов позволит извлечь немало выгод в разрабатываемых проектах.

Допустим, что в рассматриваемую здесь систему автоматизации учебного процесса требуется включить регистрационный компонент, в обязанности которого входит ввод в систему новых занятий. Процедура регистрации должна предусматривать рассылку уведомлений администратору после ввода нового занятия. Но пользователи данной системы никак не могут решить, в каком виде эти уведомления должны рассылаться: по электронной почте или в виде коротких текстовых сообщений (SMS). На самом деле они так любят спорить, что от них вполне можно ожидать изменения формы общения в ближайшем будущем. Более того, они могут потребовать, чтобы уведомления рассылались всеми возможными видами связи. Поэтому изменение режима уведомления в одном месте кода может повлечь за собой аналогичные изменения во многих других местах системы.

Если использовать в коде явные ссылки на классы `Mailer` и `Texter`, то система станет жестко привязанной к конкретному типу рассылки уведомлений. Это примерно то же самое, что привязаться к конкретному типу

платформы базы данных, используя в коде вызовы ее специализированных функций из интерфейса API.

Ниже приведен фрагмент кода, в котором подробности реализации конкретной системы уведомления скрыты от кода, в котором она используется:

```
// Листинг 8.13
class RegistrationMgr
{
    public function register(Lesson $lesson): void
    {
        // Некоторые действия с Lesson
        //и отправка кому-нибудь сообщения
        $notifier = Notifier::getNotifier();
        $notifier->inform("new lesson: cost ({$lesson->cost()})");
    }
}

// Листинг 8.14
abstract class Notifier
{
    public static function getNotifier(): Notifier
    {
        // Получить конкретный класс в соответствии с
        // конфигурацией или иной логикой
        if (rand(1, 2) === 1)
        {
            return new MailNotifier();
        }
        else
        {
            return new TextNotifier();
        }
    }
    abstract public function inform($message): void;
}

// Листинг 8.15
class MailNotifier extends Notifier
{
    public function inform($message): void
    {
        print "Уведомление почтой: {$message}\n";
    }
}
```

```
// Листинг 8.16
class TextNotifier extends Notifier
{
    public function inform($message): void
    {
        print "Уведомление текстом: {$message}\n";
    }
}
```

В данном примере мы создали класс RegistrationMgr, который является простым клиентским классом для классов типа Notifier. Несмотря на то что класс Notifier объявлен как абстрактный, в нем реализован статический метод getNotifier(), который создает и возвращает конкретный объект типа Notifier (TextNotifier или MailNotifier) в зависимости от сложившейся ситуации. В реальном проекте выбор конкретного типа объекта должен определяться каким-нибудь гибким способом, например параметром в файле конфигурации. Здесь же мы немного слукавили, выбрав тип объекта случайным образом. Классы MailNotifier и TextNotifier практически ничего не делают. Они просто выводят с помощью метода inform() сведения, полученные в качестве параметра, дополняя их идентификатором, чтобы было видно, из какого класса делается вызов.

Обратите внимание на то, что только в методе Notifier::getNotifier() известно, какой именно объект типа Notifier должен быть создан. В итоге уведомления можно посылать из самых разных мест системы, а при изменении типа уведомлений соответствующие коррективы потребуется внести только в один метод из класса Notifier.

Ниже приведен фрагмент кода, в котором метод register() вызывается из класса RegistrationMgr:

```
// Листинг 8.17
$lessons1 = new Seminar(4, new TimedCostStrategy());
$lessons2 = new Lecture(4, new FixedCostStrategy());
$mgr = new RegistrationMgr();
$mgr->register($lessons1);
$mgr->register($lessons2);
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Уведомление текстом: Новое занятие: стоимость - (20)
Уведомление почтой: Новое занятие: стоимость - (30)
```

На рис. 8.6 показана диаграмма классов в проектируемой системе. Обратите внимание на то, что структура, приведенная на рис. 8.6, очень сильно напоминает структуру компонентов Doctrine, представленную на рис. 8.5.

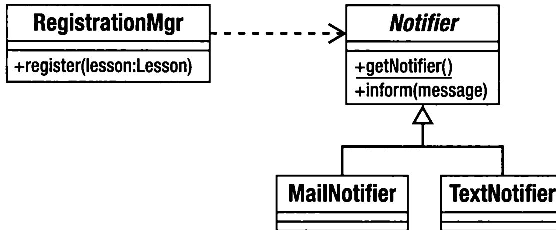


Рис. 8.6. *Класс `Notifier` развязывает клиентский код от различных реализаций уведомителей*

Программируйте на основе интерфейса, а не его реализации

Этот принцип служит одним из главных лейтмотивов данной книги, пронизывая весь ее материал. Как было показано в главе 6, “Объекты и проектирование”, и в предыдущем разделе, разные реализации можно скрыть за общим интерфейсом, определенным в суперклассе. В итоге появляется возможность оперировать в клиентском коде объектом, относящимся к суперклассу, а не к реализующему его классу, не особенно заботясь о его конкретной реализации.

Параллельно используемые условные инструкции, подобные введенным в методы `Lesson::cost()` и `Lesson::chargeType()`, служат явным признаком того, что необходимо прибегнуть к полиморфизму. Условные инструкции усложняют сопровождение кода, потому что изменение в одном условном выражении ведет к необходимости изменений во всех родственных ему выражениях. Об условных инструкциях иногда говорят, что они реализуют “сымитированное наследование”.

Размещая алгоритмы оплаты в отдельных классах, реализующих интерфейс `CostStrategy`, мы избавляемся от дублирования. Мы также намного упрощаем процесс внедрения новых стратегий оплаты, если возникнет такая потребность. С точки зрения клиентского кода иногда полезно потребовать, чтобы в параметрах метода задавались абстрактные или общие

типы данных. А требуя более конкретные типы, можно ограничить гибкость кода во время выполнения программы.

Но, конечно, степень обобщенности, выбираемая при объявлении типов аргументов, требует рассудительности. Если выбрать слишком обобщенный вариант, то метод может стать менее безопасным. А если потребовать конкретные функциональные возможности от подтипа, то передача методу совершенно иначе оснащенного родственного элемента может оказаться рискованной. Но если выбрать слишком ограниченные объявления типов аргументов, то потеряются все преимущества полиморфизма. Рассмотрим следующий видоизмененный фрагмент кода из класса `Lesson`:

```
// Листинг 8.18
public function __construct(private int $duration,
                            private FixedCostStrategy $costStrategy)
{
}
```

Проектное решение, демонстрируемое в данном примере, вызывает два вопроса. Во-первых, объект типа `Lesson` теперь привязан к определенной стратегии стоимости, и поэтому мы больше не можем формировать динамические компоненты. И во-вторых, явная ссылка на класс `FixedPriceStrategy` вынуждает нас поддерживать эту конкретную реализацию.

Требуя общий интерфейс, можно сочетать объект типа `Lesson` с любой реализацией интерфейса `CostStrategy`:

```
// Листинг 8.19
public function __construct(private int $duration,
                            private CostStrategy $costStrategy)
{
}
```

Иными словами, мы отвязали класс `Lesson` от особенностей расчетов стоимости занятия. Здесь самое главное — это интерфейс и гарантия, что предоставленный объект будет ему соответствовать.

Безусловно, программирование на основе интерфейса зачастую позволяет лишь на время отложить вопрос о том, как создавать экземпляры объектов. Говоря, что во время выполнения программы объекту типа `Lesson` можно передать любой объект, поддерживающий интерфейс `CostStrategy`, мы уклоняемся от вопроса, откуда возьмется объект типа `CostStrategy`?

При создании абстрактного суперкласса всегда возникают вопросы, как создавать экземпляры его дочерних объектов и какой дочерний объект выбрать и в соответствии с какими условиями? Эти вопросы образуют отдельную категорию в каталоге шаблонов "Банды четырех", и мы подробнее исследуем ее в следующей главе.

Меняющаяся концепция

После того как проектное решение осуществлено, его легко объяснить. Но как узнать, с чего начинать?

"Банда четырех" рекомендует "инкапсулировать меняющуюся концепцию". В рассматриваемом здесь примере организации занятий меняющаяся концепция — это алгоритм оплаты. Но это может быть не только расчет стоимости для одной из двух возможных стратегий оплаты, как в данном примере. Очевидно, что эту стратегию можно расширить, введя специальные предложения, тарифы для иностранных учащихся, вступительные скидки и другие возможности.

Мы быстро установили, что создание подклассов в условиях постоянно меняющейся ситуации неприемлемо, и поэтому прибегли к условным инструкциям. Разместив в одном классе все варианты оплаты, мы тем самым подчеркнули его пригодность для инкапсуляции.

"Банда четырех" рекомендует активно искать меняющиеся элементы в классах и оценивать их пригодность для инкапсуляции в новом типе. Можно извлечь каждый альтернативный вариант в анализируемой условной инструкции и сформировать отдельный класс, расширяющий общий абстрактный родительский класс. Затем этот новый тип можно использовать в тех классах, из которых он был извлечен. В результате будет достигнуто следующее.

- Сосредоточение обязанностей
- Поддержка гибкости благодаря композиции
- Большая компактность и сосредоточенность иерархий наследования
- Сокращение дублирования

Но как выявить изменение концепции? Одним из его признаков служит злоупотребление наследованием и, в частности, применение наследования под одновременным воздействием разных обстоятельств (лекция/семинар

и фиксированная/повременная оплата) или создание подклассов на основе алгоритма, который является второстепенным по отношению к основной обязанности типа. Другим признаком изменения концепции, пригодной для инкапсуляции, служит условное выражение, как было показано выше.

Проблемы применения шаблонов

Одна из задач, для которых не существует шаблонов, состоит в необязательном или неподходящем использовании шаблонов. Такое положение вещей создало проектным шаблонам плохую репутацию в некоторых кругах. Решения на основе шаблонов изящны и ясны, и поэтому возникает искушение применять их везде, где только можно, независимо от того, есть ли в этом реальная необходимость.

Методика экстремального программирования (eXtreme Programming — XP) предлагает ряд принципов, применимых в данной ситуации. Первый принцип гласит: "Вам это не понадобится" (обычно для его обозначения употребляется сокращение "YAGNI" от *You aren't going to need it*). Как правило, данный принцип применяется к функциональным средствам приложения, но он имеет смысл и в шаблонах.

Работая над крупными проектами на PHP, я обычно разбиваю приложение на уровни, отделяя логику приложения от представления данных и уровня сохранения данных. И в этом случае я пользуюсь всевозможными видами основных и промышленных шаблонов, а также их комбинациями.

Но если меня попросят создать простую форму для обратной связи небольшого веб-сайта, я могу запросто воспользоваться процедурным кодом, разместив его на одной странице с кодом HTML. В этом случае мне не требуется гибкость в крупных масштабах, потому что я не буду что-либо строить на этой первоначальной основе. Мне не потребуются и шаблоны, позволяющие решать соответствующие задачи в более крупных системах. Поэтому я применяю второй принцип экстремального программирования: "Делайте самое простое, что только может работать".

Работая с каталогом шаблонов, думайте о структуре и процессе решения и обращайте внимание на пример кода. Но прежде чем применить шаблон, подумайте о том, когда его использовать, найдите соответствующий раздел и прочитайте о результатах применения шаблона. В некоторых контекстах лечение может оказаться хуже болезни.

Шаблоны

Эта книга — не каталог шаблонов. Тем не менее в последующих главах мы рассмотрим несколько основных проектных шаблонов, которые применяются в настоящее время, предоставив примеры их реализации на PHP и обсудив их в широком контексте программирования на PHP.

Описываемые здесь шаблоны взяты из основных каталогов, включая упоминавшиеся ранее книги *Design Patterns* и *Patterns of Enterprise Application Architecture* Мартина Фаулера и *Core J2EE Patterns* Дипака Алура. В качестве отправной точки далее используется разделение шаблонов на категории, принятое “Бандой четырех”.

Шаблоны для формирования объектов

Эти шаблоны предназначены для создания экземпляров объектов. Это важная категория, если учитывать принцип кодирования на основе интерфейса. Если в своем проекте мы работаем с абстрактными родительскими классами, то должны разработать стратегии создания экземпляров объектов на основе конкретных подклассов. Эти объекты будут передаваться по всей системе.

Шаблоны для организации объектов и классов

Эти шаблоны помогают упорядочить композиционные отношения объектов. Проще говоря, эти шаблоны показывают, как объединять объекты и классы.

Шаблоны, ориентированные на задачи

Эти шаблоны описывают механизмы, посредством которых классы и объекты взаимодействуют для достижения целей.

Промышленные шаблоны

Мы рассмотрим некоторые шаблоны, описывающие типичные задачи и решения программирования для Интернета. Эти шаблоны, взятые в основном из книг *Patterns of Enterprise Application Architecture* и *Core J2EE Patterns*, имеют отношение к представлению данных и логике приложения.

Шаблоны баз данных

Мы также сделаем краткий обзор шаблонов, которые помогают сохранять и извлекать данные из баз данных и устанавливать соответствие между объектами базы данных и приложения.

Резюме

В этой главе мы рассмотрели некоторые принципы, лежащие в основе многих проектных шаблонов. Мы обсудили применение композиции, позволяющей сочетать и воссоединять объекты во время выполнения программы. Это предоставляет возможность создавать более гибкие структуры, чем те, в которых применяется только наследование. Мы представили также развязывание — норму практики выделения программных компонентов из их контекста, чтобы применять их в более обширном контексте. И, наконец, мы проанализировали особое значение интерфейса как средства развязки клиентского кода от подробностей реализации. В последующих главах мы подробнее исследуем некоторые проектные шаблоны, основные категории которых были вкратце описаны в этой главе.

ГЛАВА 9

Генерация объектов

Создание объектов — довольно хлопотное дело, поэтому во многих объектно-ориентированных проектах применяются изящные и четко определенные абстрактные классы. Это позволяет извлечь выгоды из впечатляющей гибкости, которую обеспечивает полиморфизм (смена конкретных реализаций во время выполнения программы). Но чтобы добиться такой гибкости, необходимо разработать стратегии генерации объектов. Именно эту тему мы здесь и обсудим.

В этой главе мы рассмотрим следующие вопросы.

- *Шаблон Singleton.* Специальный класс, формирующий один и только один экземпляр объекта.
- *Шаблон Factory Method.* Создание иерархии наследования классов создателя.
- *Шаблон Abstract Factory.* Создание групп функционально связанных программных продуктов.
- *Шаблон Prototype.* Применение операции `clone` для формирования объектов.
- *Шаблон Service Locator.* Запрос объектов у системы.
- *Шаблон Dependency Injection.* Наделение системы полномочиями предоставлять объекты.

Формирование объектов: задачи и решения

Создание объекта является слабым местом объектно-ориентированного проектирования. В предыдущей главе мы описали принцип программирования на основе интерфейса, а не его реализации, побуждающий нас оперировать абстрактными супертипами в рассматриваемых здесь классах. Ведь это делает код более гибким, позволяя применять объекты, экземпляры которых получаются из разных конкретных подклассов во время вы-

полнения программы. Побочным эффектом такого подхода оказывается отложенный характер получения экземпляров объектов.

Вот абстрактный класс, конструктору которого передается строка с именем сотрудника, для которого и создается экземпляр конкретного объекта:

```
// Листинг 9.1
abstract class Employee
{
    public function __construct(protected string $name)
    {
    }
    abstract public function fire(): void;
}
```

А вот конкретный класс, расширяющий класс `Employee`:

```
// Листинг 9.2
class Minion extends Employee
{
    public function fire(): void
    {
        print "{$this->name}: я уберу со стола\n";
    }
}
```

Теперь — класс клиента, который работает с объектами `Minion`:

```
// Листинг 9.3
class NastyBoss
{
    private array $employees = [];
    public function addEmployee(string $employeeName): void
    {
        $this->employees[] = new Minion($employeeName);
    }
    public function projectFails(): void
    {
        if (count($this->employees) > 0)
        {
            $emp = array_pop($this->employees);
            $emp->fire();
        }
    }
}
```

Соединив вместе

```
// Листинг 9.4
$boss = new NastyBoss();
$boss->addEmployee("Игорь");
$boss->addEmployee("Владимир");
$boss->addEmployee("Мария");
$boss->projectFails();
```

получим следующий вывод:

```
Мария: я уберу со стола
```

Как видите, мы определяем абстрактный базовый класс `Employee` и реализующий его класс `Minion`. Получив символьную строку с именем, метод `NastyBoss::addEmployee()` создает экземпляр нового объекта типа `Minion`. Каждый раз, когда злобный начальник, представленный объектом типа `NastyBoss`, попадает в неприятную ситуацию (из-за вызова метода `NastyBoss::projectFails()`, т.е. провала проекта), он ищет подходящего подчиненного, представленного объектом типа `Minion`, чтобы уволить его.

Инстанцируя объект типа `Minion` непосредственно в классе `NastyBoss`, мы ограничиваем гибкость последнего. Вот если бы объект `NastyBoss` мог оперировать *любым* экземпляром объекта типа `Employee`, мы могли бы сделать код доступным для изменения непосредственно во время выполнения программы, просто добавляя дополнительные специализации класса `Employee`. Полиморфизм, представленный на рис. 9.1, должен выглядеть для вас уже знакомым.

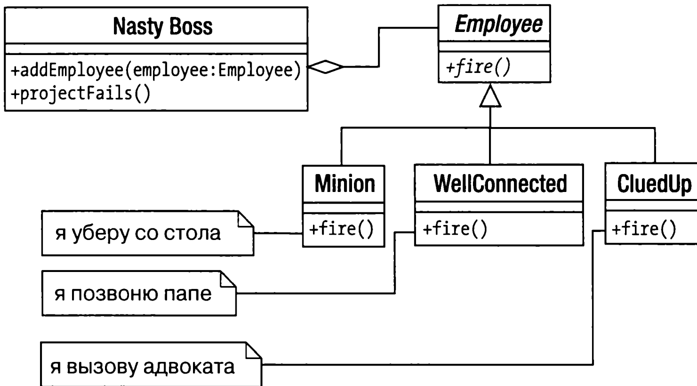


Рис. 9.1. *Оперирование абстрактным типом активизирует полиморфизм*

Если класс NastyBoss не создаст экземпляр объекта Minion, то откуда он возьмется? Авторы часто уклоняются от решения данной проблемы, ограничивая тип аргумента в объявлении метода, а затем с легкостью опуская демонстрацию создания экземпляров везде, за исключением тестового контекста:

```
// Листинг 9.5
class NastyBoss
{
    private array $employees = [];
    public function addEmployee(Employee $employee): void
    {
        $this->employees[] = $employee;
    }
    public function projectFails(): void
    {
        if (count($this->employees))
        {
            $emp = array_pop($this->employees);
            $emp->fire();
        }
    }
}

// Листинг 9.6
class CluedUp extends Employee
{
    public function fire(): void
    {
        print "{$this->name}: я вызову адвоката\n";
    }
}

// Листинг 9.7
$boss = new NastyBoss();
$boss->addEmployee(new Minion("Игорь"));
$boss->addEmployee(new CluedUp("Владимир"));
$boss->addEmployee(new Minion("Мария"));
$boss->projectFails();
$boss->projectFails();
$boss->projectFails();
```

```
Мария:  я уберу со стола
Владимир: я вызову адвоката
Игорь:  я уберу со стола
```

Несмотря на то что новая версия класса NastyBoss оперирует типом Employee и поэтому получает выгоды от полиморфизма, мы все еще не определили стратегию создания объекта. Получение экземпляров объектов — малоприятная, но необходимая процедура. В этой главе речь пойдет о классах и объектах, которые оперируют конкретными классами таким образом, чтобы избавить остальные классы от этой обязанности.

И если здесь требуется сформулировать принцип, то он должен гласить: "Делегировать получение экземпляров объекта". Мы сделали это неявно в предыдущем примере, потребовав, чтобы методу NastyBoss::addEmployee() передавался объект типа Employee. Но мы могли бы точно так же делегировать эту функцию отдельному классу или методу, который принимает на себя ответственность формировать объекты типа Employee. Итак, добавим в класс Employee статический метод, в котором реализована стратегия создания объекта:

```
// Листинг 9.8
abstract class Employee
{
    private static $types = ['Minion', 'CluedUp', 'WellConnected'];
    public static function recruit(string $name): Employee
    {
        $num = rand(1, count(self::$types)) - 1;
        $class = __NAMESPACE__ . "\\" . self::$types[$num];
        return new $class($name);
    }
    public function __construct(protected string $name)
    {
    }
    abstract public function fire(): void;
}

// Листинг 9.9
class WellConnected extends Employee
{
    public function fire(): void
    {
        print "{$this->name}: я позвоню папе\n";
    }
}
```

Как видите, данному методу передается символьная строка с именем сотрудника, которая служит для получения экземпляра конкретного подтипа Employee, выбранного случайным образом. Теперь мы можем деле-

гировать подробности получения экземпляра объекта методу `recruit()` из класса `Employee`:

```
// Листинг 9.10
$boss = new NastyBoss();
$boss->addEmployee(Employee::recruit("Игорь"));
$boss->addEmployee(Employee::recruit("Владимир"));
$boss->addEmployee(Employee::recruit("Мария"));
```

Вы видели простой пример такого класса в главе 4, “Расширенные возможности”. Я помещал статический метод `getInstance()` в класс `ShopProduct`.

На заметку В этой главе часто употребляется термин *фабрика*. Фабрика — это класс или метод, отвечающий за формирование объектов.

Метод `getInstance()` отвечал за формирование подходящего подкласса, производного от класса `ShopProduct`, на основании данных, полученных в результате запроса к базе данных. Поэтому класс `ShopProduct` выполняет двойную роль. Он определяет тип `ShopProduct`, но действует и как фабрика по созданию конкретных объектов этого типа, как показано ниже:

```
// Листинг 9.11
public static function getInstance(int $id,
                                   \PDO $pdo): ShopProduct
{
    $stmt = $pdo->prepare("select * from products where id=?");
    $result = $stmt->execute([$id]);
    $row = $stmt->fetch();

    if (empty($row))
    {
        return null;
    }

    if ($row['type'] == "book")
    {
        // Инстанцируется объект BookProduct
    } elseif($row['type'] == "cd")

    {
        // Инстанцируется объект CDProduct
    }
```

```
    else
    {
        // Инстанцируется объект ShopProduct
    }

    $product->setId((int) $row['id']);
    $product->setDiscount((int) $row['discount']);
    return $product;
}
```

В методе `getInstance()` для определения конкретных подклассов для инстанцирования используется довольно крупная условная инструкция. Подобные условные инструкции распространены в коде фабрик. Обычно в проектах следует избавляться от таких крупных условных инструкций; это зачастую приводит к тому, что выполнение условных инструкций откладывается до момента формирования объектов. Как правило, это не представляет серьезных трудностей, потому что параллельные условные инструкции удаляются из исходного кода, приурочивая принятие решения к данному моменту. В этой главе мы исследуем некоторые из основных шаблонов "Банды четырех" для формирования объектов.

Шаблон Singleton

Глобальная переменная — это один из самых крупных источников проблем для разработчика, пользующегося методикой объектно-ориентированного программирования. Причины этого к настоящему моменту уже должны быть вам понятны. Глобальные переменные привязывают классы к их контексту, подрывая основы инкапсуляции (подробнее об этом — в главах 6, "Объекты и проектирование", и 8, "Некоторые принципы проектных шаблонов"). Если в классе применяется глобальная переменная, то его невозможно извлечь из одного приложения и применить в другом, не убедившись сначала, что в новом приложении определяются такие же глобальные переменные.

Незащищенный характер глобальных переменных может стать причиной серьезных осложнений, несмотря на то что ими удобно пользоваться. Как только вы начнете пользоваться глобальными переменными, останется только ждать, когда в одной из библиотек будет объявлена глобальная переменная, которая в конечном счете вступит в конфликт с другой глобальной переменной, объявленной где-нибудь еще. Мы уже отмечали, что язык PHP уязвим к конфликтам имен классов, но конфликт

глобальных переменных оказывается намного более серьезным. Ведь интерпретатор PHP не предупредит вас, когда произойдет такой конфликт. Вы узнаете об этом только тогда, когда ваша программа начнет вести себя ненормально. А еще хуже, если вы вообще не заметите никаких затруднений на стадии разработки. Но, используя глобальные переменные, вы потенциально подвергаете пользователей угрозе новых конфликтов, когда они попытаются воспользоваться вашей библиотекой вместе с другими ресурсами.

Но искушение воспользоваться глобальными переменными все равно остается. Дело в том, что иногда недостатки глобальных переменных становятся именно той ценой, которую приходится платить за предоставление всем классам доступа к одному объекту.

Как пояснялось выше, избежать подобного рода затруднений частично помогают пространства имен. Они по крайней мере позволяют собрать переменные в отдельном пакете, а это означает, что библиотеки сторонних разработчиков уже в меньшей степени будут вступать в конфликт с исходным кодом проектируемой системы. Но даже в таком случае существует риск возникновения конфликтов в самом пространстве имен.

На заметку Константы и функции в дополнение к переменным также имеют области видимости пространств имен. Когда переменная, константа или функция вызывается без явного указания пространства имен, PHP сначала ищет ее локально, а затем — в глобальном пространстве имен.

Проблема

Как правило, в удачно спроектированных системах экземпляры объектов передаются в виде параметров при вызове методов. При этом каждый класс сохраняет свою независимость от более обширного контекста и взаимодействует с другими частями системы через очевидные каналы связи. Но иногда можно обнаружить, что некоторые классы приходится использовать как каналы передачи данных для объектов, которые не имеют к ним никакого отношения, создавая зависимости ради грамотного проектирования.

Рассмотрим в качестве примера класс `Preferences`, в котором хранятся данные, используемые в процессе выполнения приложения. Мы можем воспользоваться объектом типа `Preferences` для хранения таких параметров, как символьные строки DSN (Data Source Names — имена источников

данных, строки, содержащие информацию, необходимую для подключения к базе данных), URL сайта, пути к файлам и т.д. Очевидно, что подобные сведения могут меняться в зависимости от конкретной установки. Данный объект может также использоваться в качестве доски объявлений (или центра размещения сообщений), которая размещается или извлекается объектами системы, не связанными с ним никак иначе.

Но идея передавать объект типа `Preferences` от одного объекта к другому не всегда оказывается удачной. Многие классы, в которых этот объект вообще не используется, будут вынуждены принимать его лишь для того, чтобы передать его объектам, с которыми они взаимодействуют. В итоге получается лишь другой вид тесной связи.

Мы также должны быть уверены, что все объекты в проектируемой системе взаимодействуют с одним и тем же объектом типа `Preferences`. Нам не нужно, чтобы одни объекты устанавливали значения в каком-то объекте, тогда как другие читали данные из совершенно иного объекта.

Итак, выделим действующие факторы рассматриваемой здесь проблемы.

- Объект типа `Preferences` должен быть доступен любому объекту в проектируемой системе.
- Объект типа `Preferences` не должен сохраняться в глобальной переменной, значение которой может быть случайно перезаписано.
- В проектируемой системе не должно быть больше одного объекта типа `Preferences`. Это означает, что *объект Y* может установить свойство в объекте типа `Preferences`, а *объект Z* — извлечь то же самое значение этого свойства, причем без непосредственного взаимодействия объектов между собой (предполагается, что оба эти объекта имеют доступ к объекту типа `Preferences`).

Реализация

Чтобы решить данную задачу, начнем с установления контроля над получением экземпляров объектов. Создадим для этого класс, экземпляр которого нельзя получить за его пределами. На первый взгляд может показаться, что сделать это трудно. Но на самом деле это лишь вопрос определения закрытого конструктора:

```
// Листинг 9.12
class Preferences
{
    private array $props = [];
    private function __construct()
    {
    }
    public function setProperty(string $key, string $val): void
    {
        $this->props[$key] = $val;
    }
    public function getProperty(string $key): string
    {
        return $this->props[$key];
    }
}
```

Безусловно, в данный момент класс Preferences совершенно бесполезен, поскольку мы довели ограничение доступа до абсурдного уровня. В частности, конструктор объявлен как private, и никакой клиентский код не сможет получить с его помощью экземпляр объекта. Поэтому методы setProperty() и getProperty() оказываются лишними.

Здесь можно воспользоваться статическим методом и статическим свойством, чтобы получить экземпляр объекта через посредника, как показано ниже:

```
// Листинг 9.13
class Preferences
{
    private array $props = [];
    private static Preferences $instance;
    private function __construct()
    {
    }
    public static function getInstance(): Preferences
    {
        if (empty(self::$instance))
        {
            self::$instance = new Preferences();
        }

        return self::$instance;
    }
    public function setProperty(string $key, string $val): void
    {
        $this->props[$key] = $val;
    }
```

```
    public function getProperty(string $key): string
    {
        return $this->props[$key];
    }
}
```

Свойство `$instance` объявлено закрытым и статическим, поэтому оно недоступно за пределами его класса, но доступно из метода `getInstance()`. Ведь этот метод объявлен как открытый и статический, а следовательно, его можно вызвать через класс из любого места в сценарии, как показано ниже:

```
// Листинг 9.14
$pref = Preferences::getInstance();
$pref->setProperty("name", "Иван");

unset($pref); // Удаление ссылки

// Демонстрация, что значение не потеряно:
$pref2 = Preferences::getInstance();
print $pref2->getProperty("name") . "\n";
```

В итоге будет выведено единственное значение параметра `name`, которое было ранее добавлено в объект типа `Preferences` и получено посредством отдельного обращения к нему:

```
Иван
```

Статический метод не может получить доступ к свойствам объектов, потому что он по определению вызывается из класса, а не из контекста объекта. Но он может получить доступ к статическому свойству. Когда, например, вызывается метод `getInstance()`, проверяется свойство `Preferences::$instance`. Если оно пусто, то создается экземпляр класса `Preferences`, который сохраняется в этом свойстве, а ссылка на него возвращается в вызывающий код. А поскольку статический метод `getInstance()` входит в состав класса типа `Preferences`, то получение экземпляра класса `Preferences` не вызывает особых затруднений, даже несмотря на закрытый конструктор этого класса.

Действие шаблона Singleton в данном примере показано на рис. 9.2.

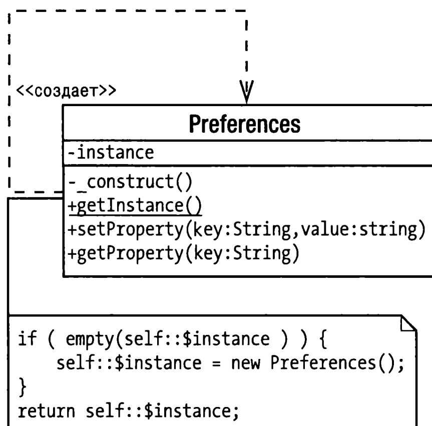


Рис. 9.2. Пример применения шаблона Singleton

Следствия

Насколько хорош подход к применению шаблона Singleton по сравнению с глобальными переменными? Начнем с плохого. Шаблоном Singleton и глобальными переменными часто злоупотребляют. Доступ к синглтонам можно получить из любого места в системе, и поэтому они могут способствовать созданию зависимостей, которые затрудняют отладку приложения. Изменения в шаблоне Singleton повлияют на классы, в которых он применяется. Сами зависимости не представляют особых трудностей, ведь мы создаем зависимость всякий раз, когда объявляем, что методу требуется передать аргумент определенного типа. Трудность состоит в том, что глобальный характер шаблона Singleton дает программисту возможность обойти каналы связи, определяемые в интерфейсах классов. Когда применяется шаблон Singleton, зависимость скрыта в теле метода и не объявляется в его сигнатуре. Это затрудняет отслеживание связей в системе, поэтому классы-синглтоны должны применяться редко и очень осторожно.

Тем не менее я считаю, что умеренное использование шаблона Singleton может улучшить проектирование системы, избавив ее от излишнего загромождения при передаче ненужных объектов в системе.

Объекты-синглтоны являются шагом вперед по сравнению с применением глобальных переменных в объектно-ориентированном контексте, поскольку перезаписать такие объекты неправильными данными нельзя.

Кроме того, вы можете группировать операции и пакеты данных в классе Singleton, что делает его гораздо более превосходным вариантом, чем ассоциативный массив или набор скалярных переменных.

Шаблон Factory Method

В объектно-ориентированном проекте акцент делается на абстрактном классе, а не на его реализации, т.е. на обобщениях, а не на частностях. Шаблон Factory Method (Фабричный метод) решает задачу получения экземпляров объектов, когда в исходном коде применяются абстрактные типы. В чем же состоит решение? Оно состоит в том, чтобы поручить получение экземпляров объектов специальным классам.

Проблема

Рассмотрим проект личного органайзера. Среди прочего в нем придется иметь дело с объектом типа `Appointment`, представляющего назначенную встречу. Допустим, что бизнес-группа нашей компании установила взаимоотношения с другой компанией, и нам поручено передать им информацию о назначенной встрече в формате BloggsCal. Но нас предупредили, что со временем могут понадобиться и другие форматы.

Оставаясь на уровне одного только интерфейса, мы можем сразу определить двух участников. В частности, нам потребуется кодировщик данных для преобразования объектов типа `Appointment` во внутренний формат BloggsCal. Присвоим этому классу имя `ApptEncoder`. Кроме того, нам потребуется управляющий класс, который будет выполнять поиск кодировщика данных, а возможно, взаимодействовать с ним для установления связи с третьей стороной. Присвоим этому классу имя `CommsManager`. Используя терминологию данного шаблона, можно сказать, что класс `CommsManager` — это производитель, а класс `ApptEncoder` — продукт. Такая структура показана на рис. 9.3.

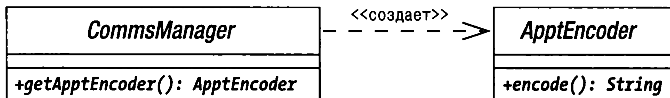


Рис. 9.3. *Абстрактные классы производителя и продукта*

Но как на самом деле получить конкретный объект типа `ApptEncoder`?

Для этого можно потребовать, чтобы объект типа ApptEncoder передавался объекту типа CommsManager, хотя это лишь откладывает решение данной проблемы, а нам нужно решить ее безотлагательно. Я инстанцирую BloggsApptEncoder непосредственно в классе CommsManager:

```
// Листинг 9.15
abstract class ApptEncoder
{
    abstract public function encode(): string;
}

// Листинг 9.16
class BloggsApptEncoder extends ApptEncoder
{
    public function encode(): string
    {
        return "Данные о встрече в формате BloggsCal\n";
    }
}

// Листинг 9.17
class CommsManager
{
    public function getApptEncoder(): ApptEncoder
    {
        return new BloggsApptEncoder();
    }
}
```

Класс CommsManager отвечает за формирование объектов типа BloggsApptEncoder. А когда корпоративные обязательства изменятся и нас попросят приспособить спроектированную систему к новому формату MegaCal, мы просто добавим условный оператор в метод CommsManager::getApptEncoder(). Ведь это та же стратегия, которой мы пользовались раньше. Итак, создадим новую реализацию класса CommsManager для поддержки форматов BloggsCal и MegaCal:

```
// Листинг 9.18
class CommsManager
{
    public const BLOGGS = 1;
    public const MEGA = 2;
    public function __construct(private int $mode)
    {
    }
```

```
    public function getApptEncoder(): ApptEncoder
    {
        switch ($this->mode)
        {
            case (self::MEGA):
                return new MegaApptEncoder();

            default:
                return new BloggsApptEncoder();
        }
    }
}

// Листинг 9.19
class MegaApptEncoder extends ApptEncoder
{
    public function encode(): string
    {
        return "Данные о встрече в формате MegaCal\n";
    }
}

// Листинг 9.20
$man = new CommsManager(CommsManager::MEGA);
print(get_class($man->getApptEncoder())) . "\n";
$man = new CommsManager(CommsManager::BLOGGS);
print(get_class($man->getApptEncoder())) . "\n";
```

В качестве флагов, определяющих два режима, в которых может работать сценарий, мы используем константы класса CommsManager MEGA и BLOGGS. В методе getApptEncoder() используется конструкция switch, чтобы проверить свойство $mode и получить экземпляр соответствующей реализации абстрактного класса ApptEncoder.

Но описанный выше подход не совсем удачен. Использование условных инструкций иногда считается дурным тоном и признаком недоброкачественного кода, хотя их нередко приходится применять в какой-то момент при создании объектов. Но не стоит слишком снисходительно относиться к проникновению в исходный код дубликатов условных инструкций. Класс CommsManager предоставляет функциональные средства для передачи календарных данных. Допустим, что в используемых нами протоколах требуется вывести данные в верхнем и нижнем колонтитулах, чтобы очертить границы каждой назначенной встречи. Расширим с этой целью предыдущий пример, определив метод getHeaderText():

```
// Листинг 9.21
class CommsManager
{
    public const BLOGGS = 1;
    public const MEGA = 2;
    public function __construct(private int $mode)
    {
    }
    public function getApptEncoder(): ApptEncoder
    {
        switch ($this->mode)
        {
            case (self::MEGA):
                return new MegaApptEncoder();

            default:
                return new BloggsApptEncoder();
        }
    }
    public function getHeaderText(): string
    {
        switch ($this->mode)
        {
            case (self::MEGA):
                return "MegaCal header\n";

            default:
                return "BloggsCal header\n";
        }
    }
}
```

Как видите, необходимость в поддержке вывода данных в верхнем и нижнем колонтитулах вынудила нас продублировать проверку типа протокола с помощью оператора `switch` уже в методе `getHeaderText()`. Но по мере внедрения поддержки новых протоколов код становится все более громоздким, особенно если ввести в него еще и метод `getFooterText()`.

Подытожим все, что нам удалось выяснить в отношении рассматриваемой здесь проблемы.

- До момента выполнения программы мы не знаем, какой тип объекта нам понадобится создать (`BloggsApptEncoder` или `MegaApptEncoder`).

- Необходимо иметь возможность достаточно просто добавлять новые типы объектов (например, для поддержки протокола SyncML в соответствии с очередным коммерческим требованием).
- Каждый вид продукта связан с контекстом, в котором требуются иные специализированные операции (например, методы `getHeaderText()` и `getFooterText()`).

Следует также отметить, что в данном случае используются условные инструкции, но, как было показано ранее, их можно естественным образом заменить полиморфизмом. Шаблон Factory Method позволяет применять наследование и полиморфизм, чтобы инкапсулировать создание конкретных продуктов. Иными словами, для каждого протокола создается свой подкласс типа `CommsManager`, в котором реализован свой метод `getApptEncoder()`.

Реализация

В шаблоне Factory Method классы производителей отделены от продуктов, которые они должны формировать. Производитель — это класс фабрики, в котором определен метод для генерации объекта отдельного продукта. Если стандартная реализация этого метода не предусмотрена, то создание экземпляров объектов поручается дочерним классам производителя. Как правило, в каждом подклассе производителя получается экземпляр параллельного дочернего класса продукта.

Переопределим класс `CommsManager` как абстрактный. Таким образом, мы сохраним гибкий суперкласс и разместим весь код, связанный с конкретным протоколом, в отдельных подклассах (рис. 9.4).

Ниже приведен пример несколько упрощенного кода:

```
// Листинг 9.22
abstract class ApptEncoder
{
    abstract public function encode(): string;
}

// Листинг 9.23
class BloggsApptEncoder extends ApptEncoder
{
    public function encode(): string
    {
        return "Данные о встрече в формате BloggsCal\n";
    }
}
```

```
// Листинг 9.24
abstract class CommsManager
{
    abstract public function getHeaderText(): string;
    abstract public function getApptEncoder(): ApptEncoder;
    abstract public function getFooterText(): string;
}

// Листинг 9.25
class BloggsCommsManager extends CommsManager
{
    public function getHeaderText(): string
    {
        return "Верхний колонтитул BloggsCal\n";
    }
    public function getApptEncoder(): ApptEncoder
    {
        return new BloggsApptEncoder();
    }
    public function getFooterText(): string
    {
        return "Нижний колонтитул BloggsCal\n";
    }
}

// Листинг 9.26
$mgr = new BloggsCommsManager();
print $mgr->getHeaderText();
print $mgr->getApptEncoder()->encode();
print $mgr->getFooterText();
```

Вот вывод этого кода:

```
BloggsCal верхний колонтитул
Данные о встрече в формате BloggsCal
BloggsCal нижний колонтитул
```

Когда от нас потребуют реализовать поддержку протокола MegaCal, для этого достаточно будет написать новую реализацию абстрактных классов. Такие классы для поддержки протокола MegaCal приведены на рис. 9.5.

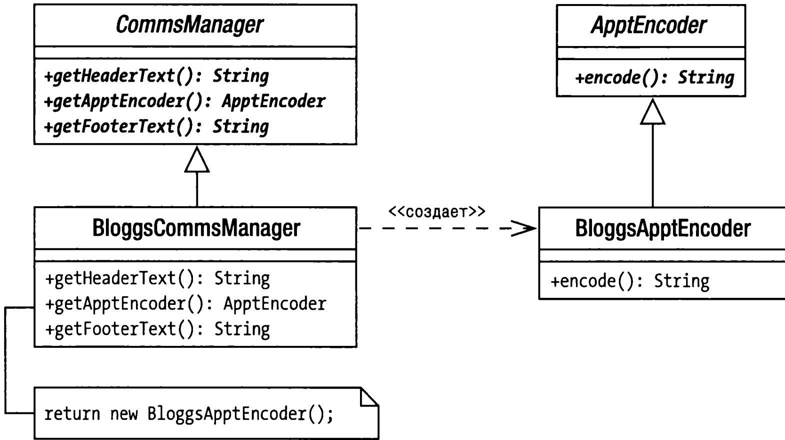


Рис. 9.4. *Конкретные классы производителя и продукта*

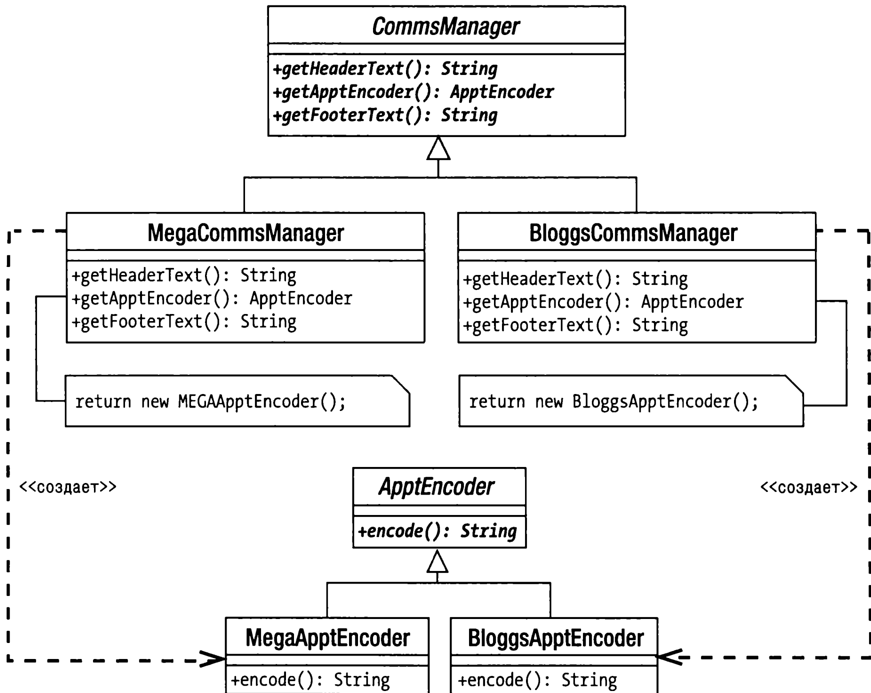


Рис. 9.5. *Расширение проекта для поддержки нового протокола*

Следствия

Обратите внимание на то, что классы производителей отражают иерархию продуктов. Это обычный результат, получаемый в результате применения шаблона Factory Method. Некоторые программисты считают этот шаблон особым видом дублирования кода и поэтому часто испытывают к нему антипатию. Еще один недостаток шаблона Factory Method заключается в том, что он нередко способствует ненужной подклассификации. Следовательно, если применение шаблона Factory Method не предполагается для формирования подклассов производителей, а иных причин для его использования не существует, то рекомендуется сначала хорошенько подумать. Именно поэтому мы продемонстрировали в приведенном выше примере ограничения, которые накладывает поддержка верхнего и нижнего колонтитулов.

В данном примере мы сосредоточились только на поддержке данных для назначенных встреч. Если же немного расширить этот пример, включив в него элементы заданий и контактов, то можно столкнуться с новой проблемой. Нам понадобится структура, которая должна одновременно обращаться с целым рядом связанных реализаций. Поэтому шаблон Factory Method нередко применяется вместе с шаблоном Abstract Factory, как поясняется в следующем разделе.

Шаблон Abstract Factory

В крупных приложениях могут понадобиться фабрики, формирующие связанные совокупности классов. И эту проблему позволяет решить шаблон Abstract Factory (Абстрактная фабрика).

Проблема

Вновь обратимся к примеру реализации личного органайзера. Ранее мы написали код для поддержки двух форматов и соответствующих протоколов BloggsCal и MegaCal. Эту структуру можно легко нарастить по горизонтали, добавив поддержку дополнительных форматов для кодирования данных. Но как нарастить ее по вертикали, чтобы добавить кодировщики данных для разных типов объектов личного органайзера? Но на самом деле мы уже работаем по данному шаблону.

На рис. 9.6 показаны параллельные семейства продуктов, с которыми нам предстоит работать. Это назначенные встречи (Appt), задания (Things To Do — Ttd) и контакты (Contact).

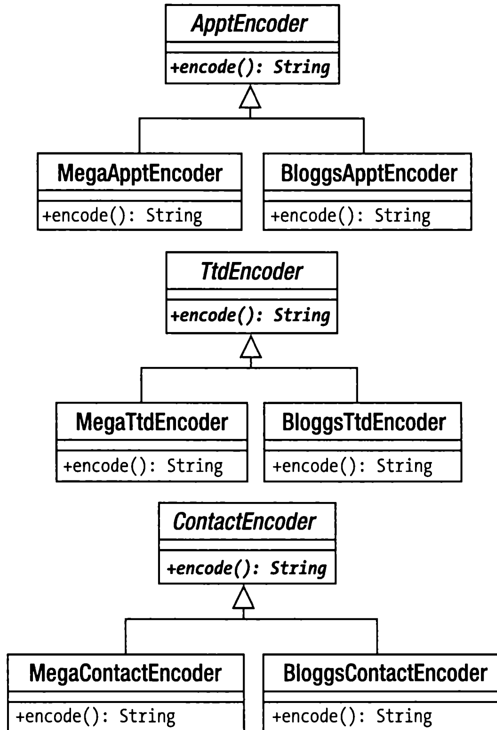


Рис. 9.6. *Три семейства продуктов*

Как видите, классы, поддерживающие формат BloggsCal, не связаны посредством наследования, хотя могут реализовывать общий интерфейс. Но в то же время они выполняют параллельные функции. Если система работает в настоящее время с классом BloggsTtdEncoder, то она должна работать и с классом BloggsContactEncoder.

Чтобы понять, как соблюсти данное требование, начнем с интерфейса, как мы это делали раньше, применяя шаблон Factory Method (рис. 9.7).

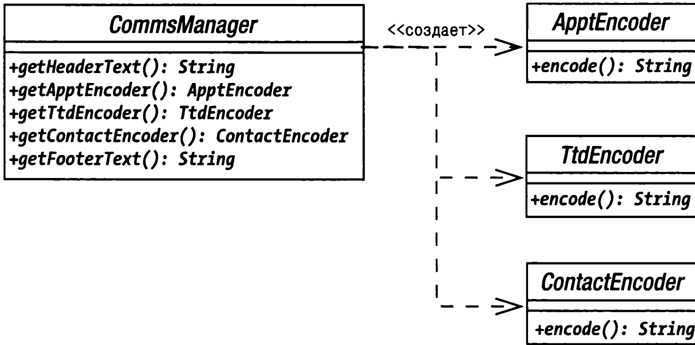


Рис. 9.7. *Абстрактный производитель и его абстрактные продукты*

Реализация

В абстрактном классе CommsManager определяется интерфейс для формирования каждого из трех продуктов, представленных объектами типа ApptEncoder, TtdEncoder и ContactEncoder. Поэтому необходимо реализовать конкретный объект создателя, чтобы формировать конкретные объекты продуктов для определенного семейства. Как это можно сделать для поддержки формата BloggsCal, показано на рис. 9.8.

Ниже приведен вариант исходного кода для классов CommsManager и BloggsCommsManager:

```
// Листинг 9.27
abstract class CommsManager
{
    abstract public function getHeaderText(): string;
    abstract public function getApptEncoder(): ApptEncoder;
    abstract public function getTtdEncoder(): TtdEncoder;
    abstract public function getContactEncoder(): ContactEncoder;
    abstract public function getFooterText(): string;
}

// Листинг 9.28
class BloggsCommsManager extends CommsManager
{
    public function getHeaderText(): string
    {
        return "BloggsCal header\n";
    }
    public function getApptEncoder(): ApptEncoder
```

```
    {
        return new BloggsApptEncoder();
    }
    public function getTtdEncoder(): TtdEncoder
    {
        return new BloggsTtdEncoder();
    }
    public function getContactEncoder(): ContactEncoder
    {
        return new BloggsContactEncoder();
    }
    public function getFooterText(): string
    {
        return "BloggsCal footer\n";
    }
}
```

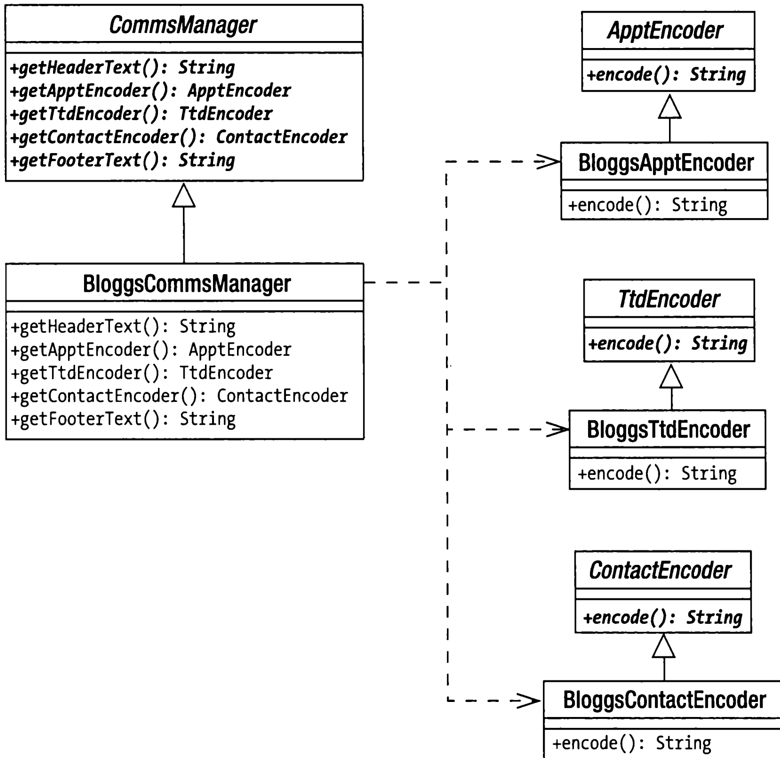


Рис. 9.8. *Добавление конкретного производителя и некоторых конкретных продуктов*

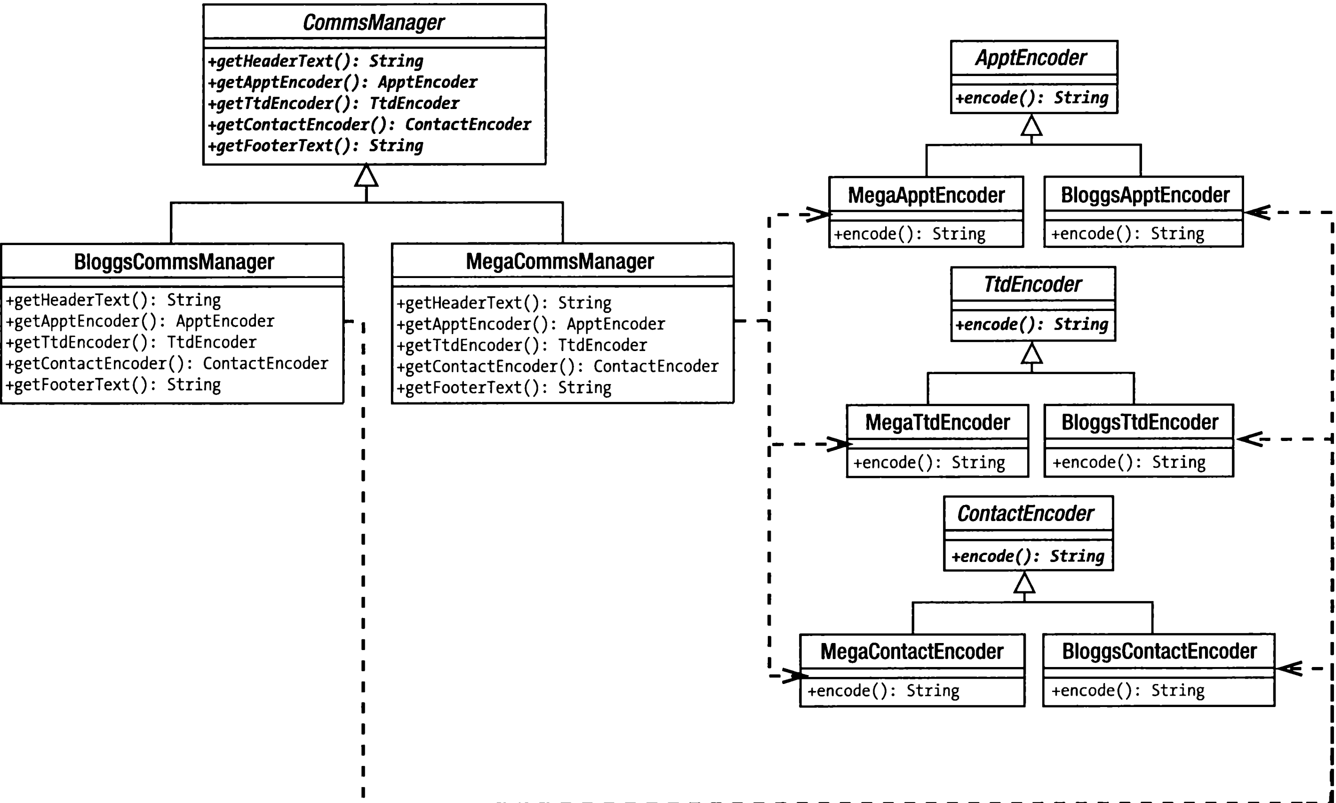


Рис. 9.9. *Добавление конкретных производителей и некоторых конкретных продуктов*

Обратите внимание на то, что в данном примере применяется шаблон Factory Method. Метод `getContactEncoder()` объявлен абстрактным в классе `CommsManager` и реализован в классе `BloggsCommsManager`. В результате проектные шаблоны работают совместно, причем один создает контекст, который служит для другого шаблона. На рис. 9.9 показано добавление поддержки формата MegaCal.

Следствия

Так что же дает применение шаблона Abstract Factory?

- Во-первых, мы отвязали проектируемую систему от подробностей реализации. В данном примере можно добавлять или удалять любое количество форматов кодирования, не опасаясь каких-нибудь осложнений.
- Во-вторых, мы ввели в действие группу функционально связанных элементов проектируемой системы. Поэтому применение класса `BloggsCommsManager` гарантирует, что работать придется только с классами, связанными с форматом BloggsCal.
- В-третьих, добавление новых продуктов может оказаться затруднительным. Для этого нам придется не только создать конкретные реализации новых продуктов, но и внести изменения в абстрактный класс производителя, а также создать каждый конкретный реализатор, чтобы его поддержать.

Во многих реализациях шаблона Abstract Factory применяется шаблон Factory Method. Возможно, причина заключается в том, что большинство примеров написано на Java или C++. Но в языке PHP не накладываются обязательные ограничения на тип, возвращаемый из метода (хотя теперь это можно делать). И это обеспечивает определенную гибкость, которой можно выгодно воспользоваться.

Вместо того чтобы создавать отдельные методы для каждого объекта в соответствии с шаблоном Factory Method, можно создать один метод `make()` и передать ему в качестве аргумента флаг, определяющий тип возвращаемого объекта, как показано ниже:

```
// Листинг 9.29
interface Encoder
{
    public function encode(): string;
}
```

```
// Листинг 9.30
abstract class CommsManager
{
    public const APPT = 1;
    public const TTD = 2;
    public const CONTACT = 3;
    abstract public function getHeaderText(): string;
    abstract public function make(int $flag_int): Encoder;
    abstract public function getFooterText(): string;
}

// Листинг 9.31
class BloggsCommsManager extends CommsManager
{
    public function getHeaderText(): string
    {
        return "Верхний колонтитул BloggsCal\n";
    }
    public function make(int $flag_int): Encoder
    {
        switch ($flag_int)
        {
            case self::APPT:
                return new BloggsApptEncoder();

            case self::CONTACT:
                return new BloggsContactEncoder();

            case self::TTD:
                return new BloggsTtdEncoder();
        }
    }
    public function getFooterText(): string
    {
        return "Нижний колонтитул BloggsCal\n";
    }
}
```

Как видите, мы сделали интерфейс класса более компактным, но заплатили за это достаточно дорогую цену. Используя шаблон Factory Method, мы определяем четкий интерфейс и вынуждаем все конкретные объекты фабрики подчиняться ему. Используя единственный метод `make()`, мы должны помнить о поддержке всех объектов продуктов во всех конкретных производителях. Мы также используем параллельные условные инструкции, поскольку в каждом конкретном производителе должны быть

реализованы одинаковые проверки флагов. Клиентский класс не может быть уверен, что конкретные производители формируют весь набор продуктов, поскольку внутренняя организация метода `make()` в каждом отдельном случае может отличаться.

С другой стороны, можно построить более гибкие производители. В базовом классе производителя можно предусмотреть метод `make()`, который будет гарантировать стандартную реализацию для каждого семейства продуктов. И тогда конкретные дочерние классы могут избирательно видоизменять такое поведение. Классам, реализующим производителя, будет предоставлено право выбора, вызывать стандартный метод `make()` после собственной реализации или не вызывать. Еще один вариант шаблона Abstract Factory мы рассмотрим в следующем разделе.

Шаблон Prototype

Когда применяется шаблон Factory Method, появление параллельных иерархий наследования может вызывать осложнения. Этот вид тесной связи вызывает у некоторых программистов ощущение дискомфорта. Всякий раз, когда добавляется новое семейство продуктов, приходится создавать связанного с ним конкретного производителя (например, кодировщикам данных формата BloggsCal соответствует класс `BloggsCommsManager`). В системе, которая быстро расширяется и включает в себя много продуктов, поддержание такого рода связи может быстро стать очень трудоемким делом.

Чтобы исключить подобную зависимость, можно, в частности, воспользоваться ключевым словом `clone` языка PHP, чтобы продублировать существующие конкретные продукты. И тогда конкретные классы продуктов сами станут основой для формирования их собственных объектов. Это, по существу, означает действовать в соответствии с шаблоном Prototype (Прототип), который позволяет заменить наследование композицией. Такой подход, в свою очередь, способствует гибкости во время выполнения программы и сокращает количество классов, которые требуется создать.

Проблема

Представьте веб-игру типа "Цивилизация", в которой элементы действуют на поле в клетку. Каждая клетка может представлять море, равнину или лес. Тип местности может ограничивать движение и возможности элемен-

тов занять клетку. Допустим, у нас имеется объект типа TerrainFactory (Фабрика местности), который позволяет создавать объекты типа Sea (море), Forest (лес) и Plains (равнины). Мы решили предоставить пользователю возможность выбирать совершенно разные типы окружающей среды. Следовательно, объект типа Sea относится к абстрактному суперклассу, реализуемому в классах MarsSea и EarthSea. Аналогичным образом реализуются объекты типа Forest и Plains. В данном случае можно применить шаблон Abstract Factory. У нас имеются различные иерархии продуктов (Sea, Forest, Plains) с сильными родственными отношениями, включающими наследование (Earth, Mars). На рис. 9.10 представлена диаграмма классов, наглядно показывающая, каким образом можно применить шаблоны Abstract Factory и Factory Method для работы с этими продуктами.

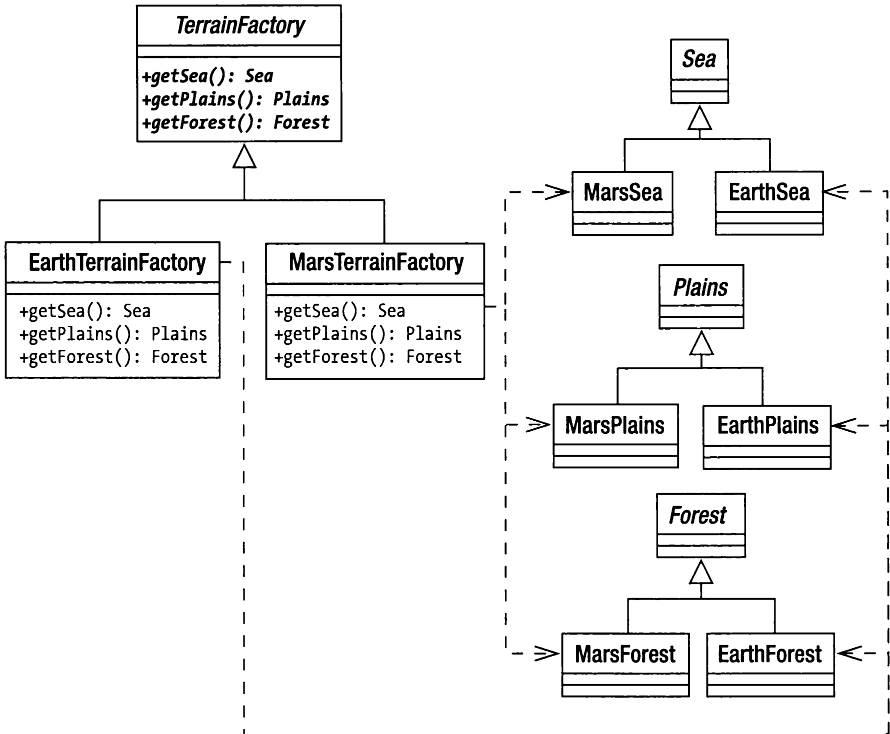


Рис. 9.10. *Обращение с различными типами местности в соответствии с шаблоном Abstract Factory*

Как видите, мы используем наследование, чтобы сгруппировать семейство типов местности для продуктов, которые будут произведены фабрикой. Это работоспособное решение, требующее крупной иерархии наследования и не обладающее достаточной гибкостью. Если параллельные иерархии наследования не нужны, но требуется максимальная гибкость во время выполнения, можно применить шаблон Prototype как более эффективный вариант шаблона Abstract Factory.

Реализация

Применяя шаблоны Abstract Factory и Factory Method, мы должны решить в определенный момент, с каким конкретно производителем нам требуется работать. Вероятно, это можно осуществить, проанализировав значения некоторого флага. А поскольку нам так или иначе необходимо это сделать, то почему бы не создать просто класс фабрики для хранения конкретных продуктов и их наполнения во время инициализации? Таким образом мы сможем избавиться от пары классов и, как мы вскоре увидим, воспользоваться другими преимуществами. Ниже приведен пример простого кода, в котором в фабрике применяется шаблон Prototype:

```
// Листинг 9.32
class Plains
{
}

// Листинг 9.33
class Forest
{
}

// Листинг 9.34
class Sea
{
}

// Листинг 9.35
class EarthPlains extends Plains
{
}
```

```
// Листинг 9.36
class EarthSea extends Sea
{
}

// Листинг 9.37
class EarthForest extends Forest
{
}

// Листинг 9.38
class MarsSea extends Sea
{
}

// Листинг 9.39
class MarsForest extends Forest
{
}

// Листинг 9.40
class MarsPlains extends Plains
{
}

// Листинг 9.41
class TerrainFactory
{
    public function __construct(private Sea $sea,
                                private Plains $plains,
                                private Forest $forest)
    {
    }
    public function getSea(): Sea
    {
        return clone $this->sea;
    }
    public function getPlains(): Plains
    {
        return clone $this->plains;
    }
    public function getForest(): Forest
    {
        return clone $this->forest;
    }
}
```

```
// Листинг 9.42
$factory = new TerrainFactory(
    new EarthSea(),
    new EarthPlains(),
    new EarthForest()
);
print_r($factory->getSea());
print_r($factory->getPlains());
print_r($factory->getForest());
```

Вот как выглядит вывод данного кода:

```
popp\ch09\batch11\EarthSea Object
(
)
popp\ch09\batch11\EarthPlains Object
(
)
popp\ch09\batch11\EarthForest Object
(
)
```

Как видите, в экземпляр конкретной фабрики типа `TerrainFactory` загружаются экземпляры объектов разных продуктов. Когда в клиентском коде вызывается метод `getSea()`, ему возвращается клон объекта типа `Sea`, который во время инициализации размещается в кеше. В итоге нам удалось не только сократить пару классов, но и достичь определенной гибкости. Хотите, чтобы игра происходила на новой планете с морями и лесами, как на Земле, и с равнинами, как на Марсе? Для этого не нужно создавать новый класс производителя; достаточно изменить набор классов, который добавляется в фабрику `TerrainFactory`, как показано ниже:

```
// Листинг 9.43
$factory = new TerrainFactory(
    new EarthSea(),
    new MarsPlains(),
    new EarthForest()
);
```

Таким образом, шаблон Prototype позволяет выгодно воспользоваться гибкостью, которую обеспечивает композиция. Но мы получили нечто большее, ведь мы сохраняем и клонируем объекты во время выполнения программы, а значит — воспроизводим состояние объектов, когда формируем новые продукты. Допустим, что у объектов типа `Sea` имеется свойство `$navigability` (судоходность). От него зависит количество энергии

движения, которое клетка моря отнимает у судна. С помощью этого свойства можно регулировать уровень сложности игры:

```
// Листинг 9.44
class Sea
{
    public function __construct(private int $navigability)
    {
    }
}
```

Теперь, когда инициализируется объект типа `TerrainFactory`, можно ввести объект типа `Sea` со свойством модификатора судоходности. И тогда это будет иметь силу для всех объектов типа `Sea`, создаваемых с помощью фабрики `TerrainFactory`:

```
// Листинг 9.45
$factory = new TerrainFactory(
    new EarthSea(-1),
    new EarthPlains(),
    new EarthForest()
);
```

Такая гибкость становится очевидной и в том случае, когда объект, который требуется сформировать, состоит из других объектов.

На заметку Клонирование объектов подробно пояснялось в главе 4, "Расширенные возможности". При выполнении операции клонирования с помощью ключевого слова `clone` получается неполная (или поверхностная) копия любого объекта, над которым она выполняется. Это означает, что объект продукта будет обладать теми же самыми свойствами, что и исходный объект. Если же какие-нибудь свойства исходного объекта оказываются объектами, они не будут скопированы в объект продукта. Вместо этого объект продукта будет ссылаться на *те же самые* свойства объекта. И вам решать, следует ли изменить этот стандартный способ копирования объектов и применить какой-нибудь другой способ, реализовав метод `__clone()`. Этот метод вызывается автоматически при выполнении операции клонирования с помощью ключевого слова `clone`.

Все объекты типа `Sea` могут содержать объекты типа `Resource` (`FishResource`, `OilResource` и т.д.). В соответствии со значением свойства можно определить, что по умолчанию все объекты типа `Sea` содержат объекты типа `FishResource`. Но не нужно забывать, что, если в продуктах имеются ссылки на другие объекты, следует реализовать метод `__clone()`, чтобы создать полную (или глубокую) копию этого продукта:

```
// Листинг 9.46
class Contained
{
}

// Листинг 9.47
class Container
{
    public Contained $contained;
    public function __construct()
    {
        $this->contained = new Contained();
    }
    public function __clone()
    {
        // Обеспечить, чтобы клонированный объект
        // содержал клон объекта, хранящегося в
        // свойстве self::$contained,
        // а не ссылку на него
        $this->contained = clone $this->contained;
    }
}
```

Доведение до крайности: шаблон Service Locator

Как было обещано, в этой главе должна идти речь о логике создания объектов, а не о конкретных примерах объектно-ориентированного программирования. Но в некоторых рассмотренных здесь шаблонах имеется ловко скрытое принятие решений о создании объектов, если не само создание объектов.

Шаблон Singleton в этом обвинить нельзя. Встроенная в него логика создания объектов совершенно однозначна. В шаблоне Abstract Factory создание семейств продуктов группируется в разных конкретных классах производителей. Но как правильно выбрать конкретного производителя? Аналогичное затруднение возникает, когда применяется шаблон Prototype. Оба эти шаблона занимаются созданием объектов, но откладывают решение, какой именно объект (или группу объектов) следует создавать.

Решение о выборе конкретного производителя обычно принимается в соответствии с заданным значением параметра конфигурации. Этот параметр может находиться в базе данных, в файле конфигурации или в

файле на сервере (например, в файле конфигурации уровня каталогов на сервере Apache; обычно он называется `.htaccess`) или может быть даже жестко закодирован в виде переменной или свойства в PHP. А поскольку конфигурация приложений на PHP может быть изменена каждым запросом или вызовом CLI, то инициализация сценария должна происходить максимально безболезненно. Поэтому я нередко прибегаю к жесткому кодированию значений флагов конфигурации в коде PHP. Можно сделать это вручную или написать сценарий для автоматического формирования файла класса. Ниже приведен пример конфигурационного класса, в который включен параметр для обозначения типа календарного протокола:

```
// Листинг 9.48
class Settings
{
    public static string $COMMSTYPE = 'Mega';
}
```

А теперь, когда имеется значение параметра, хотя это и сделано не совсем изящно, можно создать класс, в котором данный параметр используется при принятии решения, какой объект типа `CommsManager` следует предоставить по запросу. В подобных случаях нередко применяется шаблон Singleton в сочетании с шаблоном Abstract Factory, поэтому так и поступим:

```
// Листинг 9.49
class AppConfig
{
    private static ? AppConfig $instance = null;
    private CommsManager $commsManager;
    private function __construct()
    {
        // Выполняется единственный раз
        $this->init();
    }
    private function init(): void
    {
        switch (Settings::$COMMSTYPE)
        {
            case 'Mega':
                $this->commsManager = new MegaCommsManager();
                break;

            default:
                $this->commsManager = new BloggsCommsManager();
```

```
        }
    }
    public static function getInstance(): AppConfig
    {
        if (is_null(self::$instance))
        {
            self::$instance = new self();
        }

        return self::$instance;
    }
    public function getCommsManager(): CommsManager
    {
        return $this->commsManager;
    }
}
```

Класс `AppConfig` — это стандартный класс-синглтон. Поэтому мы можем легко получить ссылку на экземпляр класса `AppConfig` в любом месте нашей программы, и это всегда будет ссылка на один и тот же экземпляр. Метод `init()` вызывается конструктором класса и поэтому выполняется только один раз в процессе выполнения программы. В этом методе проверяется значение свойства `Settings::$COMMSTYPE`, и в соответствии с ним создается экземпляр конкретного объекта типа `CommsManager`, как показано ниже. Теперь в сценарии можно получить объект типа `CommsManager` и оперировать им, даже не зная о его конкретных реализациях или конкретных классах, которые он формирует:

```
$commsMgr = AppConfig::getInstance()->getCommsManager();
print $commsMgr->getApptEncoder()->encode();
```

Класс `AppConfig` организует поиск и создание компонентов автоматически, и поэтому служит характерным примером применения шаблона Service Locator (Определитель служб). Это изящный шаблон, хотя он и вносит зависимость в менее легкой форме, чем при непосредственной инициализации. В любых классах, в которых применяется столь монолитная служба, созданная по такому шаблону, ее приходится вызывать явным образом, что привязывает их к остальной системе в более обширном контексте. Именно поэтому некоторые программисты предпочитают другой подход.

Блестящее одиночество: шаблон Dependency Injection

В примере из предыдущего раздела для выбора на обслуживание одного из классов, расширяющих класс `CommsManager`, были использованы специальный признак и условная инструкция, хотя это решение и оказалось не таким гибким, как хотелось бы. Доступные для выбора классы были жестко закодированы в одном определителе служб, а выбор обоих компонентов встроен в условную инструкцию. Однако подобная негибкость является скорее недостатком продемонстрированного выше кода, чем самого шаблона Service Locator. В данном примере можно было бы воспользоваться любым количеством стратегий, чтобы найти и получить экземпляр и возвратить объекты от имени клиентского кода. Но истинная причина, по которой шаблон Service Locator нередко вызывает подозрения, заключается в том, что определитель служб должен вызываться в компоненте явным образом. Такой подход кажется слишком глобальным, а разработчики объектно-ориентированных приложений с большим подозрением относятся ко всему глобальному.

Проблема

Всякий раз, когда выполняется операция `new`, закрывается возможность для полиморфизма в данной области видимости. Рассмотрим в качестве примера следующий метод, в котором применяется жестко закодированный объект типа `BloggsApptEncoder`:

```
// Листинг 9.50
class AppointmentMaker
{
    public function makeAppointment(): string
    {
        $encoder = new BloggsApptEncoder();
        return $encoder->encode();
    }
}
```

Этого может оказаться достаточно для первоначальных потребностей, но перейти к любой другой реализации класса `ApptEncoder` во время выполнения программы все же не удастся. Тем самым ограничиваются возможности применения данного класса и затрудняется его тестирование.

На заметку Модульные тесты обычно разрабатываются таким образом, чтобы сосредоточиться на определенных классах и методах в изоляции от более широкой системы. Если тестируемый класс включает непосредственно инстанцированный объект, то могут быть выполнены все виды кода, не относящиеся к тесту, — возможно, вызывая ошибки и неожиданные побочные эффекты. Если же, с другой стороны, тестируемый класс получает объекты, с которыми он работает каким-либо образом, кроме непосредственного создания экземпляра, то для целей тестирования он может быть представлен поддельными объектами (*имитациями* или *заглушками*). Подробности тестирования рассматриваются в главе 18, "Тестирование средствами PHPUnit".
Непосредственные инстанцирования затрудняют тестирование кода. Преодолению именно такой негибкости и посвящена большая часть этой главы. Но, как отмечалось вскользь в предыдущем разделе, экземпляр должен быть получен в каком-то *другом* месте, несмотря на применение шаблона Prototype или Abstract Factory. Ниже приведен еще один фрагмент кода, в котором объект создается в соответствии с шаблоном Prototype:

```
// Листинг 9.51
$factory = new TerrainFactory(
    new EarthSea(),
    new EarthPlains(),
    new EarthForest()
);
```

Класс `TerrainFactory` — это шаг в правильном направлении. Для получения его экземпляра требуются обобщенные типы `Sea`, `Plains` и `Forest`. Этот класс предоставляет клиентскому коду самостоятельно выбирать, какую именно его реализацию следует предоставить. Но как это сделать?

Реализация

В большей части рассматриваемого здесь кода требуется обращение к фабрикам. И, как было показано ранее, именно такая модель называется шаблоном Service Locator, в соответствии с которым метод делегирует поставщику, которому он вполне доверяет, обязанность найти и обслужить требующийся тип данных. Совсем иначе дело обстоит в примере применения шаблона Prototype, в котором просто ожидается, что во время вызова в коде получения экземпляра будут предоставлены соответствующие реализации. И в этом нет ничего необычного, поскольку нужно лишь указать необходимые типы данных в сигнатуре конструктора вместо того, чтобы

создавать их в самом методе. А иначе можно предоставить методы установки, чтобы клиенты смогли передать объекты, прежде чем вызывать метод, в котором они применяются.

Внесем следующие коррективы в класс AppointmentMaker:

```
// Листинг 9.52
class AppointmentMaker2
{
    public function __construct(private ApptEncoder $encoder)
    {
    }
    public function makeAppointment(): string
    {
        return $this->encoder->encode();
    }
}
```

Итак, мы добились искомой гибкости благодаря тому, что класс AppointmentMaker2 вернул управление, — объект типа BloggsApptEncoder в нем больше не создается. Но каким же образом действует логика создания объектов и где находятся внушающие трепет операторы new? Для выполнения этих обязанностей потребуется компонент-сборщик. Обычная стратегия применяет файл конфигурации, в котором определяются конкретные реализации для получения экземпляров. И хотя здесь можно было бы воспользоваться вспомогательными инструментальными средствами, данная книга посвящена способам выхода из подобных затруднительных положений самостоятельно. Поэтому попробуем сначала создать весьма прямолинейную реализацию, начав с незамысловатого файла конфигурации формата XML, в котором описываются отношения между абстрактными классами и их предпочтительными реализациями:

```
// Листинг 9.53
<objects>
    <class name="popp\ch09\batch06\ApptEncoder">
    <instance inst="popp\ch09\batch06\BloggsApptEncoder" />
    </class>
</objects>
```

Здесь утверждается, что, когда мы запрашиваем ApptEncoder, наш инструмент должен генерировать BloggsApptEncoder. Конечно, нам нужно создать сборщик:

```
// Листинг 9.54
class ObjectAssembler
```

```
{
    private array $components = [];
    public function __construct(string $conf)
    {
        $this->configure($conf);
    }
    private function configure(string $conf): void
    {
        $data = simplexml_load_file($conf);

        foreach ($data->class as $class)
        {
            $name = (string)$class['name'];
            $resolvedname = $name;

            if (isset($class->instance))
            {
                if (isset($class->instance[0]['inst']))
                {
                    $resolvedname =
                            (string)$class->instance[0]['inst'];
                }
            }

            $this->components[$name] =
                            function() use($resolvedname)
            {
                $rclass = new \ReflectionClass($resolvedname);
                return $rclass->newInstance();
            };
        }
    }
    public function getComponent(string $class): object
    {
        if (isset($this->components[$class]))
        {
            $inst = $this->components[$class]();
        }
        else
        {
            $rclass = new \ReflectionClass($class);
            $inst = $rclass->newInstance();
        }

        return $inst;
    }
}
```

Это на первый взгляд немного трудная для понимания реализация, поэтому рассмотрим ее вкратце. Основное действие происходит в методе configure(), которому передается путь к файлу конфигурации, полученный из конструктора. Для синтаксического анализа XML-файла конфигурации в нем применяется расширение simplexml. Если в реальном проекте потребуется более основательная обработка ошибок, то в данном примере мы вполне доверяем содержимому анализируемого XML-файла конфигурации.

Для каждого элемента разметки <class> я извлекаю полностью квалифицированное имя класса, которое сохраняю в переменной $name. Я также создаю переменную $resolvedname, которая будет содержать имя конкретного класса, который мы сгенерируем. Предполагая, что элемент <instance> найден (а в более поздних примерах вы увидите, что он будет присутствовать не всегда), я присваиваю корректное значение переменной $resolvedname.

Я не хочу создавать объект до тех пор, пока он не понадобится, поэтому я создаю анонимную функцию, которая будет создавать объект при вызове и добавлять его в свойство $components.

Метод getComponent() принимает имя заданного класса и преобразует его в экземпляр. Это делается одним из двух способов. Если предоставленное имя класса является ключом в массиве $components, то я извлекаю и запускаю соответствующую анонимную функцию. Если же я не могу найти запись для предоставленного класса, я все еще могу попытаться создать экземпляр. Наконец я возвращаю полученный результат.

Давайте протестируем этот код:

```
// Листинг 9.55
$assembler = new
    ObjectAssembler("src/ch09/batch14_1/objects.xml");
$encoder = $assembler->getComponent(ApptEncoder::class);
$apptmaker = new AppointmentMaker2($encoder);
$out = $apptmaker->makeAppointment();
print $out;
```

Поскольку ApptEncoder::class разрешается в popp\ch09\batch06\ApptEncoder — ключ установлен в файле objects.xml, — создается и возвращается экземпляр объекта BloggsApptEncoder. Вы можете увидеть это в выводе данного фрагмента:

```
Данные о встрече в формате BloggsCal
```

Как видите, данный код достаточно умен, чтобы создать конкретный объект, даже если его нет в файле конфигурации:

```
// Листинг 9.56
$assembler = new
    ObjectAssembler("src/ch09/batch14_1/objects.xml");
$encoder = $assembler->getComponent(MegaApptEncoder::class);
$apptmaker = new AppointmentMaker2($encoder);
$out = $apptmaker->makeAppointment();
print $out;
```

В файле конфигурации нет ключа `MegaApptEncoder`, но поскольку класс `MegaApptEncoder` существует и может быть инстанцирован, класс `ObjectAssembler` может создать и вернуть экземпляр.

А как насчет объектов с конструкторами, требующими аргументов? Мы можем решить этот вопрос без особого труда. Помните самый последний класс `TerrainFactory`? Для него требуются объекты `Sea`, `Plains` и `Forest`. Здесь я изменяю свой формат XML для удовлетворения этого требования:

```
// Листинг 9.57
<objects>
    <class name="popp\ch09\batch11\TerrainFactory">
        <arg num="0" inst="popp\ch09\batch11\EarthSea" />
        <arg num="1" inst="popp\ch09\batch11\MarsPlains" />
        <arg num="2" inst="popp\ch09\batch11\Forest" />
    </class>
    <class name="popp\ch09\batch11\Forest">
        <instance inst="popp\ch09\batch11\EarthForest" />
    </class>
    <class name="popp\ch09\batch14\AppointmentMaker2">
        <arg num="0" inst="popp\ch09\batch06\BloggsApptEncoder" />
    </class>
</objects>
```

В этой главе я описал два класса: `TerrainFactory` и `AppointmentMaker2`. Я хочу создать экземпляр `TerrainFactory` с объектами `EarthSea`, `MarsPlains` и `EarthForest`. Я также хочу передать объект `AppointmentMaker2` объекту `BloggsApptEncoder`. Поскольку `TerrainFactory` и `AppointmentMaker2` — конкретные классы, мне в любом случае не нужно предоставлять элементы `<instance>`.

Хотя `EarthSea` и `MarsPlains` являются конкретными классами, обратите внимание, что `Forest` является абстрактным. Это изящная логическая рекурсия. Хотя сам `Forest` не может быть создан, есть соответствующий элемент `<class>`, который определяет конкретный экземпляр. Как

вы думаете, новая версия `ObjectAssembler` сможет справиться с этими требованиями?

```
// Листинг 9.58
class ObjectAssembler
{
    private array $components = [];
    public function __construct(string $conf)
    {
        $this->configure($conf);
    }
    private function configure(string $conf): void
    {
        $data = simplexml_load_file($conf);

        foreach ($data->class as $class)
        {
            $args = [];
            $name = (string)$class['name'];
            $resolvedname = $name;

            foreach ($class->arg as $arg)
            {
                $argclass = (string)$arg['inst'];
                $args[(int)$arg['num']] = $argclass;
            }

            if (isset($class->instance))
            {
                if (isset($class->instance[0]['inst']))
                {
                    $resolvedname =
                            (string)$class->instance[0]['inst'];
                }
            }

            ksort($args);
            $this->components[$name] =
                    function() use($resolvedname,$args)
            {
                $expandedargs = [];

                foreach ($args as $arg)
                {
                    $expandedargs[] = $this->getComponent($arg);
                }
```

```
                $rclass = new \ReflectionClass($resolvedname);
                return $rclass->newInstanceArgs($expandedargs);
            };
        }
    }
    public function getComponent(string $class): object
    {
        if (isset($this->components[$class]))
        {
            $inst = $this->components[$class]();
        }
        else
        {
            $rclass = new \ReflectionClass($class);
            $inst = $rclass->newInstance();
        }

        return $inst;
    }
}
```

Давайте подробнее рассмотрим, что здесь нового.

Во-первых, в методе `configure()` я перебираю все элементы `<arg>` в каждом элементе `<class>` и создаю список имен классов:

```
// Листинг 9.59
foreach ($class->arg as $arg)
{
    $argclass = (string)$arg['inst'];
    $args[(int)$arg['num']] = $argclass;
}
```

В таком случае, чтобы расширить каждый из этих элементов в экземпляры объекта для передачи в конструктор моего класса, мне не требуется выполнять много работы в анонимной функции построения. В конце концов, я уже создал для этого метод `getComponent()`:

```
// Листинг 9.60
ksort($args);
$this->components[$name] = function() use($resolvedname, $args)
{
    $expandedargs = [];

    foreach ($args as $arg)
    {
        $expandedargs[] = $this->getComponent($arg);
    }
```

```
    $rclass = new \ReflectionClass($resolvedname);
    return $rclass->newInstanceArgs($expandedargs);
};
```

На заметку Если вы собираетесь создавать сборщик/контейнер в соответствии с шаблоном Dependency Injection, рассмотрите следующие возможные средства: Pimple и Symfony DI. Подробнее о Pimple можно узнать по адресу `http://pimple.sensiolabs.org/`, а о компоненте Symfony DI — по адресу `http://symfony.com/doc/current/components/dependency_injection/introduction.html`.

Как бы там ни было, гибкость компонентов можно все же сохранить, организуя инстанцирование в динамическом режиме. Опробуем класс `ObjectAssembler` в следующем примере кода:

```
// Листинг 9.61
$assembler = new ObjectAssembler("src/ch09/batch14/objects.xml");
$apptmaker = $assembler->getComponent(AppointmentMaker2::class);
$out = $apptmaker->makeAppointment();
print $out;
```

При наличии класса `ObjectAssembler` для получения экземпляра объекта достаточно единственной инструкции. Теперь класс `AppointmentMaker2` свободен от прежней жестко закодированной зависимости от экземпляра класса `ApptEncoder`. Разработчик может воспользоваться файлом конфигурации, чтобы проконтролировать те классы, которые применяются во время выполнения, а также протестировать класс `AppointmentMaker2` отдельно от остальной системы в более обширном контексте.

Dependency Injection и атрибуты

Можно также воспользоваться атрибутами, появившимися в PHP 8, чтобы переместить часть рассматриваемой логики из конфигурационного класса в сами файлы, и мы можем сделать это, не жертвуя функциональностью, которую мы уже определили.

На заметку Атрибуты рассматривались в главе 5, "Средства для работы с объектами".

Вот еще один XML-файл. Я не ввожу в нем никаких новых возможностей. Фактически файл конфигурации берет на себя ответственность за *меньшее* количество логики:

```
// Листинг 9.62
<objects>
    <class name="popp\ch09\batch06\ApptEncoder">
        <instance inst="popp\ch09\batch06\BloggsApptEncoder" />
    </class>
    <class name="popp\ch09\batch11\Sea">
        <instance inst="popp\ch09\batch11\EarthSea" />
    </class>
    <class name="popp\ch09\batch11\Plains">
        <instance inst="popp\ch09\batch11\MarsPlains" />
    </class>
    <class name="popp\ch09\batch11\Forest">
        <instance inst="popp\ch09\batch11\EarthForest" />
    </class>
</objects>
```

Я хочу сгенерировать новую версию `TerrainFactory`. Если определение для него в файле конфигурации отсутствует, то где я могу его найти? Ответ лежит в самом классе `TerrainFactory`:

```
// Листинг 9.63
class TerrainFactory
{
    #[InjectConstructor(Sea::class, Plains::class, Forest::class)]
    public function __construct(private Sea $sea,
                                private Plains $plains,
                                private Forest $forest)
    {
    }
    public function getSea(): Sea
    {
        return clone $this->sea;
    }
    public function getPlains(): Plains
    {
        return clone $this->plains;
    }
    public function getForest(): Forest
    {
        return clone $this->forest;
    }
}
```

Это просто класс `TerrainConstructor` для шаблона Prototype, который вы уже видели, но с важным добавлением атрибута `InjectConstructor`. Это требует определения шаблона класса:

```
// Листинг 9.64
use Attribute;
#[Attribute]
public class InjectConstructor
{
    function __construct()
    {
    }
}
```

Итак, атрибут `InjectConstructor` определяет необходимое поведение. Я хочу, чтобы в примере использования Dependency Injection обеспечивались конкретные экземпляры абстрактных классов `Sea`, `Plains` и `Forest`. Для этого необходимо еще раз поработать над классом `ObjectAssembler`:

```
// Листинг 9.65
class ObjectAssembler
{
    private array $components = [];
    public function __construct(string $conf)
    {
        $this->configure($conf);
    }
    private function configure(string $conf): void
    {
        $data = simplexml_load_file($conf);
        foreach ($data->class as $class)
        {
            $args = [];
            $name = (string)$class['name'];
            $resolvedname = $name;
            foreach ($class->arg as $arg)
            {
                $argclass = (string)$arg['inst'];
                $args[(int)$arg['num']] = $argclass;
            }
            if (isset($class->instance))
            {
                if (isset($class->instance[0]['inst']))
                {
                    $resolvedname =
                        (string)$class->instance[0]['inst'];
                }
```

```
            }
            ksort($args);
            $this->components[$name] = function() use($resolvedname,
                                                       $args)
            {
                $expandedargs = [];
                foreach ($args as $arg)
                {
                    $expandedargs[] = $this->getComponent($arg);
                }
                $rclass = new \ReflectionClass($resolvedname);
                return $rclass->newInstanceArgs($expandedargs);
            };
        }
    }
    public function getComponent(string $class): object
    {
        // Создание $inst - экземпляр нашего объекта
        // и список объектов \ReflectionMethod
        if (isset($this->components[$class]))
        {
            // Экземпляр, найденный в конфигурационном файле
            $inst = $this->components[$class]();
            $rclass = new \ReflectionClass($inst::class);
            $methods = $rclass->getMethods();
        }
        else
        {
            $rclass = new \ReflectionClass($class);
            $methods = $rclass->getMethods();
            $injectconstructor = null;
            foreach ($methods as $method)
            {
                foreach($method->getAttributes(
                            InjectConstructor::class)
                        as $attribute)
                {
                    $injectconstructor = $attribute;
                    break;
                }
            }
            if (is_null($injectconstructor))
            {
                $inst = $rclass->newInstance();
            }
            else
            {
```

```
                $constructorargs = [];
                foreach ($injectconstructor->getArguments() as $arg)
                {
                    $constructorargs[] = $this->getComponent($arg);
                }
                $inst = $rclass->newInstanceArgs($constructorargs);
            }
        }
        return $inst;
    }
}
```

Возможно, все это кажется вам устрашающим. Но я добавил не так уж и много. Давайте разберемся. Все добавленное находится в `getComponent()`. Найдя предоставленный ключ класса — переменную аргумента `$class` — в свойстве-массиве `$components`, я просто полагаюсь на соответствующую анонимную функцию, которая выполняет инстанцирование. Если же нет, то логику можно найти в атрибутах. Чтобы проверить это, я циклически обхожу все методы целевого класса в поисках атрибута `InjectConstructor`. Если я нахожу его, то рассматриваю соответствующий метод как конструктор. Я раскрываю каждый из аргументов атрибута в собственный экземпляр объекта, а затем передаю законченный список в `ReflectionClass::newInstanceArgs()`. Если же я не нахожу атрибут `InjectConstructor`, то я просто создаю экземпляр без аргументов, используя `ReflectionClass::newInstance()`.

Обратите внимание, что в этом примере я создаю массив с именем `$methods`, который содержит объекты `ReflectionMethod` для данного класса. Сейчас этот массив лишний, но вскоре мы найдем ему применение.

Далее еще раз показана описанная логика, извлеченная из метода `ObjectAssembler::getComponent()`:

```
// Листинг 9.66
$rclass = new \ReflectionClass($class);
$methods = $rclass->getMethods();
$injectconstructor = null;

foreach ($methods as $method)
{
    foreach ($method->getAttributes(InjectConstructor::class)
                 as $attribute)
    {
        $injectconstructor = $attribute;
        break;
```

```
    }
}

if (is_null($injectconstructor))
{
    $inst = $rclass->newInstance();
}
else
{
    $constructorargs = [];

    foreach ($injectconstructor->getArguments() as $arg)
    {
        $constructorargs[] = $this->getComponent($arg);
    }

    $inst = $rclass->newInstanceArgs($constructorargs);
}
```

Обратите внимание на использование рекурсии. Для того чтобы расширить аргумент атрибута в объект, я передаю имя класса назад — в `getComponent()`.

Теперь теоретически я могу генерировать заполненный объект `TerrainFactory`:

```
// Листинг 9.67
$assembler = new ObjectAssembler("src/ch09/batch15/objects.xml");
$terrainfactory = $assembler->getComponent(TerrainFactory::class);
$plains = $terrainfactory->getPlains(); // MarsPlains
```

Когда объект `ObjectAssembler` вызывается с именем `TerrainFactory`, метод `ObjectAssembler::getComponent()` сначала ищет в массиве `$components` соответствующий элемент конфигурации. В нашем случае он его там не находит. Поэтому далее он циклически обходит методы `TerrainFactory` и находит атрибут `InjectConstructor`. Этот атрибут имеет три аргумента. Для каждого из них рекурсивно вызывается метод `getComponent()`. В каждом из этих случаев он *находит* элемент конфигурации, который предоставляет класс, из которого может быть инстанцирован данный аргумент.

На заметку Этот пример кода не проверяет наличие циклической рекурсии. Но окончательная версия кода этого должна как минимум предотвращать слишком глубокие, многоуровневые рекурсивные вызовы `getComponent()`.

Наконец, давайте добавим новый атрибут. Inject похож на InjectConstructor, с тем отличием, что его следует применять к стандартным методам, которые будут вызываться после инстанцирования целевого объекта. Вот как мы используем этот атрибут:

```
// Листинг 9.68
class AppointmentMaker
{
    private ApptEncoder $encoder;
    #[Inject(ApptEncoder::class)]
    public function setApptEncoder(ApptEncoder $encoder)
    {
        $this->encoder = $encoder;
    }
    public function makeAppointment(): string
    {
        return $this->encoder->encode();
    }
}
```

Суть директивы в том, что класс AppointmentMaker после инстанцирования должен иметь объект ApptEncoder.

Вот шаблонный класс Inject, соответствующий атрибуту:

```
// Листинг 9.69
use Attribute;
#[Attribute]
class Inject
{
    public function __construct()
    {
    }
}
```

Как и InjectConstructor, этот класс в действительности не делает ничего полезного, кроме заполнения пространства имен. Теперь пришло время добавить поддержку Inject в ObjectAssembler:

```
// Листинг 9.70
public function getComponent(string $class): object
{
    // Создание $inst - экземпляр нашего объекта
    // и список объектов \ReflectionMethod
    $this->injectMethods($inst, $methods);
    return $inst;
}
```

```
public function injectMethods(object $inst, array $methods)
{
    foreach ($methods as $method)
    {
        foreach ($method->getAttributes(Inject::class)
                    as $attribute)
        {
            $args = [];
            foreach ($attribute->getArguments() as $argstring)
            {
                $args[] = $this->getComponent($argstring);
            }
            $method->invokeArgs($inst, $args);
        }
    }
}
```

Я опустил бо́льшую часть `getComponent()`, поскольку здесь он не изменяется. Единственное дополнение — вызов нового метода `injectMethods()`. Он принимает инстанцированный объект и массив объектов `ReflectionMethod`. Затем он исполняет знакомый нам танец: циклический обход методов с атрибутами `Inject`, получение аргументов атрибута и передачу каждого из них обратно в `getComponent()`. После составления списка аргументов происходит вызов метода для данного экземпляра.

Вот пример клиентского кода:

```
// Листинг 9.71
$assembler = new ObjectAssembler("src/ch09/batch15/objects.xml");
$apptmaker = $assembler->getComponent(AppointmentMaker::class);
$output = $apptmaker->makeAppointment();
print $output;
```

Итак, когда я вызываю `getComponent()`, он создает экземпляр `AppointmentMaker`. Затем он вызывает `injectMethods()`, который находит метод с атрибутом `Inject` в классе `AppointmentMaker`. Аргумент атрибута — `ApptEncoder`. Этот ключ класса рекурсивно передается в `getComponent()`. Поскольку наш файл конфигурации указывает `BloggsApptEncoder` в качестве разрешения для `ApptEncoder`, соответствующий объект инстанцируется и передается методу установки.

Все это демонстрирует вывод рассмотренного кода:

```
Данные о встрече в формате BloggsCal
```

Ниже показан полный код `ObjectAssembler`:

```
// Листинг 9.72
class ObjectAssembler
{
    private array $components = [];
    public function __construct(string $conf)
    {
        $this->configure($conf);
    }
    private function configure(string $conf): void
    {
        $data = simplexml_load_file($conf);
        foreach ($data->class as $class)
        {
            $args = [];
            $name = (string)$class['name'];
            $resolvedname = $name;
            foreach ($class->arg as $arg)
            {
                $argclass = (string)$arg['inst'];
                $args[(int)$arg['num']] = $argclass;
            }
            if (isset($class->instance))
            {
                if (isset($class->instance[0]['inst']))
                {
                    $resolvedname =
                        (string)$class->instance[0]['inst'];
                }
            }
            ksort($args);
            $this->components[$name] = function()
                    use($resolvedname, $args)
            {
                $expandedargs = [];
                foreach ($args as $arg)
                {
                    $expandedargs[] = $this->getComponent($arg);
                }
                $rclass = new \ReflectionClass($resolvedname);
                return $rclass->newInstanceArgs($expandedargs);
            };
        }
    }
    public function getComponent(string $class): object
    {
```

```
        // Создание $inst - экземпляр нашего объекта
        // и список объектов \ReflectionMethod
        if (isset($this->components[$class]))
        {
            // Экземпляр, найденный в конфигурационном файле
            $inst = $this->components[$class]();
            $rclass = new \ReflectionClass($inst::class);
            $methods = $rclass->getMethods();
        }
        else
        {
            $rclass = new \ReflectionClass($class);
            $methods = $rclass->getMethods();
            $injectconstructor = null;
            foreach ($methods as $method)
            {
                foreach(
                    $method->getAttributes(InjectConstructor::class)
                         as $attribute)
                {
                    $injectconstructor = $attribute;
                    break;
                }
            }
            if (is_null($injectconstructor))
            {
                $inst = $rclass->newInstance();
            }
            else
            {
                $constructorargs = [];
                foreach ($injectconstructor->getArguments() as $arg)
                {
                    $constructorargs[] = $this->getComponent($arg);
                }
                $inst = $rclass->newInstanceArgs($constructorargs);
            }
        }
        $this->injectMethods($inst, $methods);
        return $inst;
    }
    public function injectMethods(object $inst, array $methods)
    {
        foreach ($methods as $method)
        {
            foreach ($method->getAttributes(Inject::class)
                         as $attribute)
```

```
            {
                $args = [];
                foreach ($attribute->getArguments() as $argstring)
                {
                    $args[] = $this->getComponent($argstring);
                }
                $method->invokeArgs($inst, $args);
            }
        }
    }
}
```

Следствия

Мы рассмотрели две возможности создания объектов. С одной стороны, класс `AppConfig` послужил примером применения шаблона Service Locator, предоставляя возможность обнаруживать компоненты или службы от имени его клиента. Безусловно, внедрение зависимости способствует написанию более изящного кода. Так, в классе `AppointmentMaker2` ничего не известно о стратегиях создания объектов. Он просто выполняет свои обязанности, что, конечно, идеально для любого класса. Мы стремимся проектировать классы, способные сосредоточиваться на своих обязанностях в как можно большей степени обособленно от остальной системы. Но такая чистота не достигается бесплатно, поскольку для этой цели требуется отдельный компонент сборщика, выполняющий все необходимые действия. Его можно рассматривать в качестве "черного ящика", доверяя ему создавать объекты как по волшебству от нашего имени. И если такое волшебство действует исправно, то нас это вполне устраивает. Но в то же время будет нелегко отладить неожиданное поведение такого "черного ящика".

С другой стороны, шаблон Service Locator проще, хотя он вынуждает встраивать компоненты в остальную систему. Впрочем, шаблон Service Locator не затрудняет тестирование и не делает систему негибкой, если он применяется должным образом. Определитель служб в соответствии с шаблоном Service Locator можно настроить на обслуживание произвольных компонентов для целей тестирования или в соответствии с заданной конфигурацией. Но жестко закодированный вызов определителя служб делает компонент зависимым от него. А поскольку такой вызов делается в теле метода, то взаимосвязь клиентского кода с целевым компонентом, предоставляемым в соответствии с шаблоном Service Locator, оказывается в ка-

кой-то степени скрытой. Ведь такая взаимосвязь объявляется в сигнатуре метода конструктора.

Какой же подход лучше выбрать? В какой-то мере это дело личных предпочтений. Я лично предпочитаю начинать с простейшего решения, постепенно усложняя его по мере надобности. Именно поэтому я обычно выбираю шаблон Service Locator, в нескольких строках кода создаю класс в соответствии с шаблоном Registry (Реестр) и затем повышаю его гибкость в соответствии с конкретными требованиями. Разрабатываемые мной компоненты осведомлены один о другом чуть больше, чем мне бы хотелось, но поскольку я редко переношу классы из одной системы в другую, я не особенно страдаю от эффекта встраивания. Переместив системный класс в отдельную библиотеку, я, например, не обнаружил никаких трудностей в устранении зависимости, возникшей в соответствии с шаблоном Service Locator.

Шаблон Dependency Injection обеспечивает чистоту, но он требует встраивания иного вида и вынуждает полагаться на сборщика. Если вы пользуетесь каркасом, в котором подобные функциональные возможности уже предоставляются, вам вряд ли стоит отказываться от его услуг. Например, компонент Symfony DI предоставляет гибридное решение в соответствии с шаблонами Service Locator и Dependency Injection, и первый из них служит в качестве контейнера службы, который организует формирование экземпляров объектов в соответствии с заданной конфигурацией (или кодом, если угодно) и предоставляет клиентам интерфейс для получения этих объектов. Такой контейнер службы допускает даже использование фабрик для создания объектов. Но если вы предпочитаете пользоваться только своими разработками или же компонентами из различных каркасов, то, возможно, пожелаете упростить дело, хотя и ценой некоторого изящества.

Резюме

В этой главе мы рассмотрели ряд приемов, которыми вы можете воспользоваться для формирования объектов. Сначала мы рассмотрели шаблон Singleton, предоставляющий глобальный доступ к единственному экземпляру объекта, а затем — шаблон Factory Method, в котором для формирования объектов применяется принцип полиморфизма. Далее мы исследовали сочетание шаблонов Factory Method и Abstract Factory

для формирования классов производителей, которые получают экземпляры наборов связанных объектов. Кроме того, мы рассмотрели шаблон Prototype и показали, каким образом клонирование объектов позволяет воспользоваться композицией для формирования объектов. И наконец мы обсудили две стратегии создания объектов в соответствии с шаблонами Service Locator и Dependency Injection.

ГЛАВА 10

Шаблоны для программирования гибких объектов

После стратегии формирования объектов можно рассмотреть ряд стратегий структурирования классов и объектов. В этой главе мы, в частности, уделим основное внимание принципу, согласно которому композиция предоставляет большую гибкость, чем наследование. Исследуемые здесь шаблоны снова взяты из каталога "Банды четырех".

В этой главе мы рассмотрим следующие вопросы.

- *Шаблон Composite.* Создание структур, в которых группы объектов можно использовать так, как будто это отдельные объекты.
- *Шаблон Decorator.* Гибкий механизм комбинирования объектов во время выполнения программы для расширения функциональных возможностей.
- *Шаблон Facade.* Создание простого интерфейса для сложных или изменяющихся систем.

Структурирование классов для повышения гибкости объектов

Как пояснялось в главе 3, "Азы объектов", начинающие программисты часто путают объекты и классы, а остальные время от времени чешут в недоумении затылки, разглядывая диаграммы классов на языке UML и пытаясь согласовать статичные структуры наследования с динамическими отношениями, в которые могут вступать объекты.

Помните принцип действия шаблонов "Предпочитайте композицию наследованию"? В нем подчеркивается напряженность, возникающая между организацией классов и объектов. Для достижения необходимой гибкости в своих проектах мы структурируем классы таким образом, чтобы их

объекты можно было компоновать в удобные структуры во время выполнения программы.

Это общая тема, которая станет главной при рассмотрении двух первых шаблонов в данной главе. Наследование является важным средством в обоих этих шаблонах, но его важность отчасти объясняется предоставлением механизма, с помощью которого можно воспользоваться композицией для представления структур и расширения функциональных возможностей.

Шаблон Composite

Шаблон Composite (Композиция) — это, вероятно, наиболее экстремальный пример наследования, которое применяется для обслуживания композиции. У этого шаблона простая, но удивительно изящная конструкция. И вместе с тем он очень полезен. Но имейте в виду: из-за всех этих преимуществ у вас может возникнуть искушение пользоваться им слишком часто.

Шаблон Composite — это простое средство, позволяющее сначала объединить группы схожих объектов, а затем управлять ими. В итоге отдельный объект для клиентского кода становится неотличимым от коллекции объектов. На самом деле это очень простой шаблон, хотя и нередко вводящий в заблуждение. Объясняется это, в частности, простой структурой классов по данному шаблону для организации его объектов. Иерархии наследования представляют собой деревья, корнем которых является суперкласс, а ветвями — специализированные подклассы. Иерархическое дерево наследования *классов*, построенное в соответствии с шаблоном Composite, предназначено для того, чтобы упростить формирование и обход дерева *объектов*.

Если вы еще не знакомы с этим шаблоном, то можете теперь почувствовать некоторое замешательство. Обратимся к аналогии, чтобы проиллюстрировать, каким образом отдельные объекты можно трактовать как коллекции предметов. Имея такие ингредиенты, как крупа и мясо (или соя, в зависимости от предпочтений), можно произвести пищевой продукт, например колбасу. И тогда с полученным результатом можно обращаться как с единой сущностью. Мы можем покупать, готовить и есть колбасу, одним из ингредиентов которой является мясо. Мы можем сочетать колбасу с другими ингредиентами, чтобы приготовить пиццу, включив одну составную часть в другую, более крупную. Таким образом, мы обращаемся

с коллекциями таким же образом, как и с их составными частями. Шаблон Composite помогает моделировать отношение между коллекциями и компонентами в прикладном коде.

Проблема

Управление группами объектов может быть довольно сложной задачей, особенно если эти объекты также содержат собственные объекты. Это очень распространенная проблема в программировании. Представьте счет-фактуру, в которой указаны дополнительные товары или услуги, или список запланированных дел, в котором элементы сами содержат несколько подзадач. Управляя содержимым, можно оперировать деревьями подразделов, страниц, статей и мультимедийных компонентов. Но внешнее управление этими структурами может быстро превратиться в очень сложную задачу.

Вернемся к сценарию, описанному в предыдущей главе. Напомним, что мы проектируем систему на основе игры “Цивилизация”. Игрок может передвигать элементы по сотням клеток на игровом поле, которые составляют карту. При этом отдельные объекты можно группировать, чтобы перемещаться, сражаться и защищаться так, как будто это единый элемент. Определим пару типов боевых единиц (так называемых юнитов):

```
// Листинг 10.1
abstract class Unit
{
    abstract public function bombardStrength(): int;
}
class Archer extends Unit
{
    public function bombardStrength(): int
    {
        return 4;
    }
}
class LaserCannonUnit extends Unit
{
    public function bombardStrength(): int
    {
        return 44;
    }
}
```

В классе `Unit` определен абстрактный метод `bombardStrength()`, в котором задается атакующая сила юнита, обстреливающего соседнюю клетку. Этот метод реализуется в классах `Archer` и `LaserCannonUnit`, которые могут также содержать сведения о перемещении и возможностях защиты. Но остановимся пока на самом простом варианте. Мы можем определить отдельный класс для группирования юнитов следующим образом:

```
// Листинг 10.2
class Army
{
    private array $units = [];
    public function addUnit(Unit $unit): void
    {
        array_push($this->units, $unit);
    }
    public function bombardStrength(): int
    {
        $ret = 0;
        foreach ($this->units as $unit)
        {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}

// Листинг 10.3
$unit1 = new Archer();
$unit2 = new LaserCannonUnit();
$army = new Army();
$army->addUnit($unit1);
$army->addUnit($unit2);
print $army->bombardStrength();
```

У класса `Army` имеется метод `addUnit()`, которому передается объект типа `Unit`. Объекты типа `Unit` сохраняются в массиве свойств `$units`. Мы вычисляем объединенную силу нашей армии в методе `bombardStrength()`. Для этого перебираются объединенные объекты типа `Unit` и для каждого из них вызывается метод `bombardStrength()`. При этом мы получаем следующий вывод:

```
48
```

Такая модель оказывается идеальной до тех пор, пока задача остается настолько простой. Но что будет, если мы введем ряд новых требований?

Допустим, что одна армия должна быть способна объединяться с другими армиями. При этом каждая армия должна сохранять свою индивидуальность, чтобы впоследствии иметь возможность выйти из более крупного соединения. Сегодня у храбрых вояк эрцгерцога может быть общая с генералом Соамсом причина атаковать незащищенный фланг армии врага, но восстание у себя на родине может заставить его армию в любой момент поспешить домой. Поэтому мы не можем просто переместить подразделения из каждой армии в новое воинское соединение.

Сначала мы можем внести коррективы в класс `Army` таким образом, чтобы оперировать в нем объектами как типа `Army`, так и типа `Unit`:

```
// Листинг 10.4
public function addArmy(Army $army): void
{
    array_push($this->armies, $army);
}
```

Затем нужно внести приведенные ниже изменения в метод `bombardStrength()`, чтобы производить итерации по всем армиям и объектам типа `Unit`:

```
// Листинг 10.5
public function bombardStrength(): int
{
    $ret = 0;

    foreach ($this->units as $unit)
    {
        $ret += $unit->bombardStrength();
    }

    foreach ($this->armies as $army)
    {
        $ret += $army->bombardStrength();
    }

    return $ret;
}
```

Эта дополнительная сложность в данный момент не представляет особых трудностей. Но помните, что мы должны сделать нечто подобное и с другими методами, определяющими, например, защитную силу в методе `defensiveStrength()`, диапазон перемещения в методе `movementRange()` и т.д. Рассматриваемая здесь игра обещает быть функ-

ционально богатой. Командование уже вызывает военно-транспортные самолеты, которые могут вмещать до десяти юнитов, чтобы быстро перебросить войска и улучшить свои позиции в некоторых районах. Очевидно, что военно-транспортный самолет подобен армии в том отношении, что в нем группируются объекты типа `Unit`. Но у него могут быть и свои характеристики. Мы можем снова внести коррективы в класс `Army`, чтобы управлять военно-транспортными самолетами (объектами типа `TroopCarrier`). Но ведь может появиться необходимость и в других типах группирования объектов типа `Unit`. Очевидно, что для этой цели нам потребуется более гибкая модель.

Рассмотрим еще раз модель, которую мы создаем. Всем созданным ранее классам требуется метод `bombardStrength()`. В действительности клиентскому коду не нужно различать армию, юнит или военно-транспортный самолет, поскольку они функционально идентичны, т.е. должны уметь перемещаться, атаковать и защищаться. А одним объектам, которые содержат другие объекты, требуются методы добавления и удаления. И эти сходные особенности приводят нас к неизбежному выводу: объекты-контейнеры имеют общий интерфейс с объектами, которые они содержат, и поэтому они должны, что вполне естественно, относиться к одному семейству типов.

Реализация

В шаблоне Composite определяется единственная иерархия наследования, которая устанавливает два различных набора функциональных обязанностей. Мы их уже видели в рассматриваемом здесь примере. Классы в этом шаблоне должны поддерживать общий набор операций как основную ответственность. Для нас это означает метод `bombardStrength()`. Классы также должны поддерживать методы для добавления и удаления объектов.

На рис. 10.1 показана диаграмма класса, наглядно демонстрирующая применение шаблона Composite для решения рассматриваемой здесь задачи.

Как видите, все элементы рассматриваемой здесь модели являются производными от класса `Unit`. Поэтому клиентский код может быть уверен, что любой объект типа `Unit` будет поддерживать метод `bombardStrength()`. А это означает, что с объектом типа `Army` можно обращаться таким же образом, как и с объектом типа `Archer`.

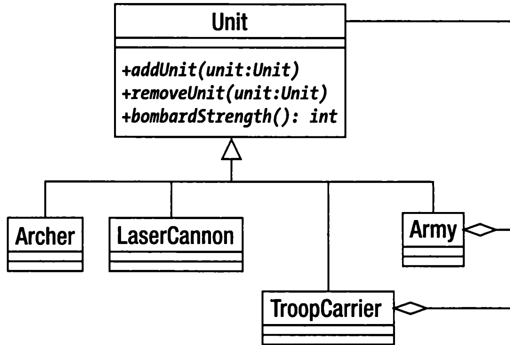


Рис. 10.1. *Шаблон Composite в действии*

Классы `Army` и `TroopCarrier` являются *составными типами данных* и предназначены для хранения объектов типа `Unit`. Классы `Archer` и `LaserCannon` относятся к *листьям*, предназначенным для поддержки операций над объектами типа `Unit`. В них не могут содержаться другие объекты типа `Unit`. И здесь возникает вопрос: должны ли листья подчиняться тому же интерфейсу, что и составные типы, как показано на диаграмме, приведенной на рис. 10.1? Как следует из этой диаграммы, в классах `TroopCarrier` и `Army` агрегируются другие объекты типа `Unit`, хотя в классах листьев также необходимо реализовать метод `addUnit()`. Мы еще вернемся к данному вопросу, а до тех пор определим абстрактный класс `Unit`:

```
// Листинг 10.6
abstract class Unit
{
    abstract public function addUnit(Unit $unit): void;
    abstract public function removeUnit(Unit $unit): void;
    abstract public function bombardStrength(): int;
}
```

Как видите, мы определили основные функции для всех объектов типа `Unit`. А теперь покажем, как реализовать эти абстрактные методы в составном объекте:

```
// Листинг 10.7
class Army extends Unit
{
    private array $units = [];
    public function addUnit(Unit $unit): void
    {
        if (in_array($unit, $this->units, true))
```

```
        {
            return;
        }
        $this->units[] = $unit;
    }
    public function removeUnit(Unit $unit): void
    {
        $idx = array_search($unit, $this->units, true);
        if (is_int($idx))
        {
            array_splice($this->units, $idx, 1, []);
        }
    }
    public function bombardStrength(): int
    {
        $ret = 0;
        foreach ($this->units as $unit)
        {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}
```

Объект типа `Unit`, переданный методу `addUnit()` в качестве параметра, сохраняется в закрытом массиве свойств `$units`. Перед этим проверяется, чтобы в массиве `$units` не было дубликатов объектов типа `Unit`. Аналогичная проверка производится и в методе `removeUnit()`, и если объект типа `Unit` обнаружен, то он удаляется из закрытого массива свойств `$units`.

На заметку При проверке, не добавлял ли я уже конкретный объект через метод `addUnit()`, я использую `in_array()` с третьим логическим аргументом `true`. Это ужесточает строгость `in_array()`, так что он будет проверять ссылки на один и тот же объект. Третий аргумент `array_search()` работает точно так же, возвращая индекс массива, только если предоставленное значение для поиска является эквивалентной ссылкой на объект, найденный в массиве.

В объектах типа `Army` могут храниться ссылки на любые объекты типа `Unit`, включая другие объекты типа `Army`, а также листья типа `Archer` или `LaserCannonUnit`. Метод `bombardStrength()` гарантированно поддерживается во всех объектах типа `Unit`, и поэтому в методе `Army::bombardStrength()` перебираются все дочерние объекты типа `Unit`, сохраненные в свойстве `$units`, и для каждого из них вызывается один и тот же метод.

У шаблона Composite имеется один серьезный недостаток: реализация функций добавления и удаления элементов. В классическом шаблоне методы `add()` и `remove()` размещаются в абстрактном суперклассе. Этим гарантируется, что во всех классах шаблона совместно используется общий интерфейс. Но, как показано ниже, это также означает, что в классах листьев необходимо обеспечить его реализацию:

```
// Листинг 10.8
class UnitException extends \Exception
{
}

// Листинг 10.9
class Archer extends Unit
{
    public function addUnit(Unit $unit): void
    {
        throw new UnitException(get_class($this) .
                                " является листом");
    }
    public function removeUnit(Unit $unit): void
    {
        throw new UnitException(get_class($this) .
                                " является листом");
    }
    public function bombardStrength(): int
    {
        return 4;
    }
}
```

Нам не требуется возможность добавлять объект типа `Unit` в объект типа `Archer`, поэтому при вызове метода `addUnit()` или `removeUnit()` генерируется исключение. А поскольку это придется делать для всех объектов листьев, проектное решение можно улучшить, заменив абстрактные методы `addUnit()` и `removeUnit()` в объекте типа `Unit` реализациями по умолчанию:

```
// Листинг 10.10
abstract class Unit
{
    public function addUnit(Unit $unit): void
    {
        throw new UnitException(get_class($this) .
                                " является листом");
    }
```

```
    public function removeUnit(Unit $unit): void
    {
        throw new UnitException(get_class($this) .
                                " является листом");
    }
    abstract public function bombardStrength(): int;
}

// Листинг 10.11
class Archer extends Unit
{
    public function bombardStrength(): int
    {
        return 4;
    }
}
```

Такой подход позволяет избавиться от дублирования кода в классах листьев. Но у него имеется серьезный недостаток: класс составного типа не обязан обеспечивать во время компиляции реализацию методов `addUnit()` и `removeUnit()`, что в конечном итоге может вызвать осложнения.

Некоторые недостатки шаблона Composite мы рассмотрим подробнее в следующем разделе. А этот раздел завершим, перечислив преимущества данного шаблона.

- *Гибкость*. Во всех элементах шаблона Composite используется общий супертип, что заметно упрощает ввод новых объектов составного типа или листьев в проектное решение, не меняя более обширный контекст программы.
- *Простота*. Клиентский код, использующий структуру шаблона Composite, имеет простой интерфейс. Клиентскому коду не нужно проводить различие между объектом, состоящим из других объектов, и объектом листа, за исключением случая добавления новых компонентов. Вызов метода `Army::bombardStrength()` может стать причиной серии делегированных внутренних вызовов, но для клиентского кода процесс и результат оказываются точно такими же, как и при вызове метода `Archer::bombardStrength()`.
- *Неявная досягаемость*. В шаблоне Composite объекты организованы в древовидную структуру. В каждом составном объекте содержатся ссылки на дочерний объект. Поэтому операция над определенной

частью дерева может иметь более обширный эффект. В частности, можно удалить один объект типа `Army` из его родительского объекта типа `Army` и добавить к другому объекту типа `Army`. Это простое действие выполняется над одним объектом, но в конечном итоге изменяется состояние всех объектов типа `Unit`, на которые ссылается объект типа `Army`, а также состояние их дочерних объектов.

- *Явная досягаемость*. В древовидной структуре можно легко выполнить обход всех ее узлов. Для получения сведений следует последовательно перебрать все ее узлы или выполнить необходимые преобразования. Довольно эффективные методы выполнения подобных действий мы рассмотрим в следующей главе при изучении шаблона Visitor.

Как правило, увидеть реальное проявление преимуществ шаблона Composite можно только в клиентском коде, поэтому создадим пару армий:

```
// Листинг 10.12
// Создание армии
$main_army = new Army();

// Добавление юнитов
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCannonUnit());

// Создание новой армии
$sub_army = new Army();

// Добавление юнитов
$sub_army->addUnit(new Archer());
$sub_army->addUnit(new Archer());
$sub_army->addUnit(new Archer());

// Добавление второй армии в первую
$main_army->addUnit($sub_army);

// Все вычисления выполняются "за кулисами"
print "Атака с силой: {$main_army->bombardStrength()}\n";
```

В данном примере создается новый объект типа `Army` и в него вводится несколько юнитов типа `Unit`. Этот процесс повторяется для второго объекта типа `Army`, который затем добавляется к первому объекту. Когда вызывается метод `Unit::bombardStrength()` для первого объекта

типа Army, вся сложность построенной структуры оказывается полностью скрытой. Вот какой вывод мы получаем:

```
Атака с силой: 60
```

Следствия

Если вы в чем-то похожи на меня, то, увидев фрагмент исходного кода из класса Archer, должны были сильно призадуматься. Для чего нужны лишние методы addUnit() и removeUnit() в классах листьев, которые по логике вещей не должны в них поддерживаться? Ответ заключается в прозрачности типа Unit.

Если клиентский код оперирует объектом типа Unit, то ему известно, что метод addUnit() всегда присутствует. В результате выполняется принцип действия шаблона Composite, который заключается в том, что у примитивных классов (листьев) должен быть такой же интерфейс, как и у составных классов. Хотя это не особенно помогает, потому что мы по-прежнему не знаем, насколько безопасно вызывать метод addUnit() для любого объекта типа Unit, с которым мы можем столкнуться.

Если переместить эти методы добавления и удаления вниз по иерархии классов, чтобы они были доступны только в классах составного типа, то при передаче объекта типа Unit методу возникает затруднение из-за того, что исходно нам неизвестно, поддерживает ли он метод addUnit(). Тем не менее оставлять методы-заглушки в классах листьев кажется неправильным. Пользы от этого никакой, а проект усложняется, поскольку интерфейс дает ложные сведения о собственных функциональных возможностях.

Классы составного типа можно легко разделить на подтипы CompositeUnit. Прежде всего исключим методы добавления и удаления юнитов из класса Unit:

```
// Листинг 10.13
abstract class Unit
{
    public function getComposite(): ? CompositeUnit
    {
        return null;
    }
    abstract public function bombardStrength(): int;
}
```

Обратите внимание на новый метод getComposite(). Мы еще вернемся к нему некоторое время спустя. А до тех пор нам потребуется новый абстрактный класс, в котором будут поддерживаться методы addUnit() и removeUnit(). Мы можем обеспечить даже его стандартные реализации:

```
// Листинг 10.14
abstract class CompositeUnit extends Unit
{
    private array $units = [];
    public function getComposite(): ? CompositeUnit
    {
        return $this;
    }
    public function addUnit(Unit $unit): void
    {
        if (in_array($unit, $this->units, true))
        {
            return;
        }

        $this->units[] = $unit;
    }
    public function removeUnit(Unit $unit): void
    {
        $idx = array_search($unit, $this->units, true);

        if (is_int($idx))
        {
            array_splice($this->units, $idx, 1, []);
        }
    }
    public function getUnits(): array
    {
        return $this->units;
    }
}
```

Класс CompositeUnit объявлен абстрактным, несмотря на то что в нем не определены никакие абстрактные методы. Тем не менее он расширяет класс Unit и не реализует абстрактный метод bombardStrength(). Класс Army и любые другие классы составного типа теперь могут расширять класс CompositeUnit. Классы в рассматриваемом здесь примере теперь организованы, как показано на рис. 10.2.

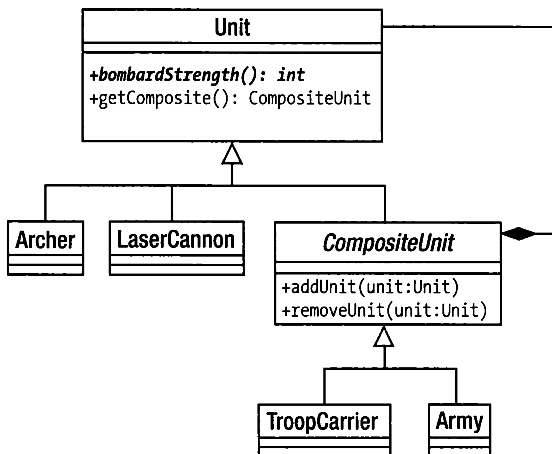


Рис. 10.2. *Перемещение методов добавления и удаления юнитов из базового класса в отдельный класс*

Итак, мы избавились от бесполезной реализации методов добавления и удаления юнитов в классах листьев, но в клиентском коде перед вызовом метода `addUnit()` теперь приходится проверять, относится ли объект к типу `CompositeUnit`. Именно здесь и вступает в свои права метод `getComposite()`. По умолчанию этот метод возвращает пустое значение. Только в классе `CompositeUnit` он возвращает ссылку на объект типа `CompositeUnit`. Поэтому, если в результате вызова данного метода возвращается объект, у нас должна появиться возможность вызвать для него метод `addUnit()`. Ниже показано, как этим приемом можно воспользоваться в клиентском коде:

```
// Листинг 10.15
class UnitScript
{
    public static function joinExisting(
        Unit $newUnit,
        Unit $occupyingUnit
    ): CompositeUnit
    {
        $comp = $occupyingUnit->getComposite();

        if (! is_null($comp))
        {
            $comp->addUnit($newUnit);
        }
```

```
        else
        {
            $comp = new Army();
            $comp->addUnit($occupyingUnit);
            $comp->addUnit($newUnit);
        }

        return $comp;
    }
}
```

Методу joinExisting() передаются два объекта типа Unit. Первый из них — это объект, вновь прибывающий в клетку, а второй — объект, занимавший клетку до этого. Если второй объект типа Unit принадлежит классу CompositeUnit, то первый объект попытается присоединиться к нему. В противном случае будет создан новый объект типа Army, включающий в себя оба объекта типа Unit. Изначально мы не можем выяснить, содержит ли аргумент $occupyingUnit объект класса CompositeUnit. Но вызов метода getComposite() позволит разрешить данное затруднение. Если метод getComposite() возвращает объект, мы можем непосредственно ввести в него новый объект типа Unit. В противном случае мы создадим новый объект типа Army и добавим в него оба объекта.

С одной стороны, данную модель можно еще больше упростить, сделав так, чтобы метод Unit::getComposite() возвратил объект типа Army, содержащий текущий объект типа Unit. С другой стороны, можно вернуться к предыдущей модели, в которой нет структурного разделения между объектами составного типа и листьями, а метод Unit::addUnit() делает то же самое, создавая объект типа Army и вводя в него оба объекта типа Unit. Это понятно, когда предполагается, что заранее известен составной тип, который требуется использовать для объединения боевых единиц типа Unit. Предположения, которые предстоит сделать при проектировании таких методов, как getComposite() и addUnit(), будут определяться логикой разрабатываемого приложения.

Подобные перекосы свидетельствуют о недостатках шаблона Composite. Простота достигается благодаря гарантии, что все классы происходят от общего базового класса. Но за простоту иногда приходится платить безопасностью использования типа данных. Чем сложнее становится модель, тем больше вероятность, что проверку типов придется выполнять вручную. Предположим, у нас имеется объект типа Cavalry (кавалерия). Если по правилам рассматриваемой здесь игры лошадей нельзя перемещать на

бронетранспортере, то сделать это автоматически с помощью шаблона Composite не удастся:

```
// Листинг 10.16
class TroopCarrier extends CompositeUnit
{
    public function addUnit(Unit $unit): void
    {
        if ($unit instanceof Cavalry)
        {
            throw new UnitException("Лошади не ездят на БТР");
        }

        parent::addUnit($unit);
    }
    public function bombardStrength(): int
    {
        return 0;
    }
}
```

Здесь мы обязаны воспользоваться операцией `instanceof`, чтобы проверить тип объекта, переданного методу `addUnit()`. Имеется слишком много такого рода ситуаций, так что недостатки шаблона Composite начинают перевешивать его преимущества. Шаблон Composite наиболее эффективен, когда большинство компонентов являются взаимозаменяемыми.

Еще один вопрос, о котором не следует забывать, имеет отношение к затратности некоторых операций, выполняемых в соответствии с шаблоном Composite. Характерным тому примером служит метод `Army::bombardStrength()`, инициирующий целый ряд вызовов одного и того же метода вниз по иерархическому дереву. Для крупного иерархического дерева с множеством вложенных армий единственный вызов может стать причиной целой лавины внутренних вызовов. Метод `bombardStrength()` сам по себе не является слишком затратным, но что произойдет, если в некоторых листьях дерева должны выполняться сложные вычисления для получения возвращаемых значений? Этот вопрос можно разрешить, сохранив результат вызова подобного метода в родительском объекте, чтобы последующие вызовы не были столь затратными. Но здесь следует проявить особое внимание, следя за тем, чтобы сохраненное значение не устарело. Необходимо выработать стратегии удаления всех сохраненных значений, когда в иерархическом дереве выполняются

какие-нибудь операции. Для этого, возможно, придется предоставить дочерним объектам ссылки на их родительские объекты.

И наконец сделаем замечание по поводу сохраняемости. Шаблон Composite довольно изящен, но не слишком пригоден для сохранения в реляционной базе данных. Причина в том, что по умолчанию обращение ко всей структуре происходит по целому каскаду ссылок. Поэтому, чтобы построить естественным образом структуру в соответствии с шаблоном Composite, придется сделать немало затратных запросов к базе данных. Эту задачу можно решить, присвоив идентификатор всему иерархическому дереву, чтобы все компоненты можно было извлечь из базы данных единым запросом. Но даже получив все нужные объекты, все равно придется воссоздавать ссылки типа "родитель–потомок", которые, в свою очередь, должны сохраняться в базе данных. Это несложное, но довольно хлопотное занятие.

Хотя, как упоминалось ранее, составные объекты нелегко сохранять в реляционной базе данных, они отлично подходят для хранения в формате XML или JSON, а следовательно, в различных не-SQL базах данных, таких как MongoDB, CouchDB и Elasticsearch. Это объясняется тем, что элементы обеих разметок, как правило, сами состоят из деревьев вложенных элементов.

Резюме

Шаблон Composite оказывается полезным, когда требуется обращаться с коллекцией объектов таким же образом, как и с отдельным объектом, или потому, что коллекция, по существу, такая же, как и компонент (например, армии и лучники), или же потому, что контекст придает коллекции такие же характеристики, как и у компонента (например, позиции в счете-фактуре). Составные объекты, создаваемые с помощью шаблона Composite, организованы в виде иерархических деревьев, и поэтому операция над целым деревом может затронуть отдельные его части, а данные из частей прозрачно доступны всему дереву. Шаблон Composite делает такие операции и запросы прозрачными и для клиентского кода. К тому же иерархические деревья легко обходить, как будет показано в следующей главе. Структуры, созданные в соответствии с шаблоном Composite, нетрудно дополнить новыми типами компонентов.

Недостаток шаблона Composite заключается в том, что составные объекты зависят от сходства их частей. Стоит лишь ввести сложные правила,

определяющие, какие именно компоненты могут содержать составные объекты, и прикладной код тут же становится трудно управляемым. Составные объекты плохо подходят для хранения в реляционной базе данных.

Шаблон Decorator

Если шаблон Composite помогает создавать гибкое представление из совокупности компонентов, то в шаблоне Decorator (Декоратор) применяется сходная структура для оказания помощи в видоизменении функциональности отдельных компонентов. И в этом шаблоне особая роль принадлежит композиции во время выполнения программы. Наследование — это ясный способ создания объектов с характеристиками, заложенными в родительском классе. Но такая ясность может привести к жесткому кодированию вариантов в иерархиях наследования, что, как правило, становится причиной отсутствия гибкости.

Проблема

Внедрение всех функциональных возможностей в структуру наследования может привести к бурному росту количества классов в проектируемой системе. Хуже того, попытки внести аналогичные изменения в разных ветвях иерархического дерева наследования, скорее всего, приведут к дублированию кода.

Вернемся к рассматривавшейся ранее игре, определив класс `Tile` и его производный тип следующим образом:

```
// Листинг 10.17
abstract class Tile
{
    abstract public function getWealthFactor(): int;
}

// Листинг 10.18
class Plains extends Tile
{
    private int $wealthfactor = 2;
    public function getWealthFactor(): int
    {
        return $this->wealthfactor;
    }
}
```

Итак, мы определили класс `Tile`, представляющий квадратную клетку, в которой могут находиться боевые единицы. У каждой клетки имеются определенные характеристики. В данном случае мы определили метод `getWealthFactor()`, влияющий на доход, который может вырабатывать определенная клетка, если ею владеет игрок. Как видите, у объектов типа `Plains` коэффициент благосостояния равен `2`. Но у клеток, конечно, могут быть и другие характеристики. Они также могут хранить ссылку на сведения об изображении для рисования игрового поля. Постараемся и в этом случае не усложнять стоящую перед нами задачу.

Нам необходимо видоизменить поведение объекта типа `Plains` (равнины), чтобы обрабатывать сведения о природных ресурсах и губительном воздействии на природу человеческого фактора. В частности, нам требуется смоделировать месторождение алмазов на местности, а также вред, наносимый загрязнением окружающей среды. Один из подходов состоит в том, чтобы воспользоваться наследованием от объекта типа `Plains`:

```
// Листинг 10.19
class DiamondPlains extends Plains
{
    public function getWealthFactor(): int
    {
        return parent::getWealthFactor() + 2;
    }
}

// Листинг 10.20
class PollutedPlains extends Plains
{
    public function getWealthFactor(): int
    {
        return parent::getWealthFactor() - 4;
    }
}
```

Теперь можно легко получить загрязненную клетку:

```
// Листинг 10.21
$tile = new PollutedPlains();
print $tile->getWealthFactor();
```

Вывод данного кода имеет следующий вид:

```
-2
```

Диаграмма классов для данного примера показана на рис. 10.3.

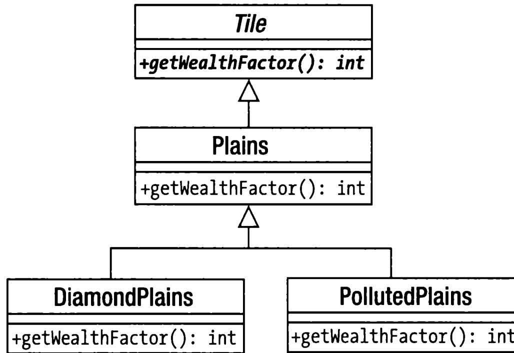


Рис. 10.3. *Построение видоизмененного варианта иерархического дерева наследования*

Очевидно, что этой структуре недостает гибкости. Мы можем получить равнины с алмазами, а также загрязненные равнины. Но можно ли получить и то, и другое? Очевидно, нет, если только мы не хотим создать нечто ужасное вроде `PollutedDiamondPlains`. Ситуация становится еще хуже, когда мы вводим класс `Forest`, в котором также могут быть и алмазы, и загрязнения.

Это, конечно, пример крайнего случая, но он наглядно демонстрирует суть дела. Если при определении функциональных возможностей полагаться только на наследование, это приведет к резкому увеличению количества классов и проявится тенденция к дублированию.

А теперь рассмотрим более общий пример. Серьезным веб-приложениям обычно требуется выполнить ряд действий после поступления запроса и до того, как инициируется задача для формирования ответа на запрос. Например, нужно аутентифицировать пользователя и зарегистрировать запрос в системном журнале. Вероятно, мы должны как-то обработать запрос, чтобы создать структуру данных на основе необработанных исходных данных, и выполнить основную обработку данных. Таким образом, перед нами встала та же самая задача.

Мы можем расширить функциональные возможности на основе класса `ProcessRequest` с дополнительной обработкой в производном классе `LogRequest`, в классе `StructureRequest`, а также в классе `AuthenticateRequest`. Эта иерархия классов показана на рис. 10.4.

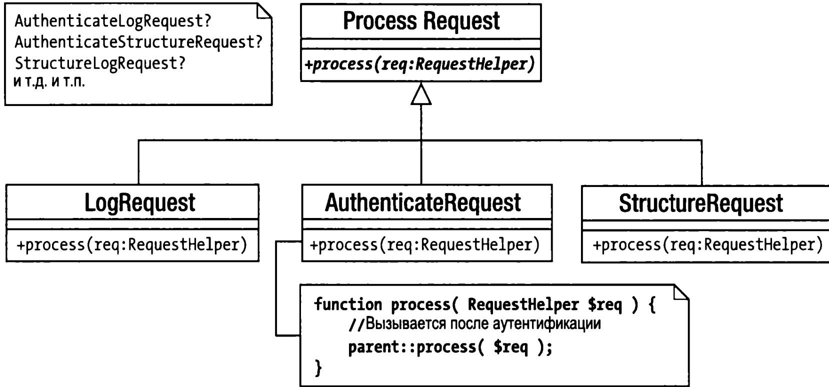


Рис. 10.4. *Дополнительные жестко закодированные вариации*

Но что произойдет, если потребуется выполнить регистрацию и аутентификацию, но не подготовку данных? Не создать ли нам класс `LogAndAuthenticateProcessor`? Очевидно, настало время найти более гибкое решение.

Реализация

Для решения задачи меняющихся функциональных возможностей вместо одного только наследования в шаблоне Decorator применяются композиция и делегирование. По существу, классы, создаваемые в соответствии с шаблоном Decorator, управляют ссылкой на экземпляр другого класса своего типа. Шаблон Decorator реализует сам процесс выполнения операции и вызывает аналогичную операцию над объектом, на который у него имеется ссылка (до или после выполнения собственных действий). Таким образом, во время выполнения программы можно создать конвейер объектов Decorator.

Чтобы продемонстрировать это, перепишем рассматриваемый здесь пример игры следующим образом:

```
// Листинг 10.22
abstract class Tile
{
    abstract public function getWealthFactor(): int;
}

// Листинг 10.23
class Plains extends Tile
```

```
{
    private int $wealthfactor = 2;
    public function getWealthFactor(): int
    {
        return $this->wealthfactor;
    }
}

// Листинг 10.24
abstract class TileDecorator extends Tile
{
    protected Tile $tile;
    public function construct(Tile $tile)
    {
        $this->tile = $tile;
    }
}
```

Здесь мы, как и прежде, объявили классы `Tile` и `Plains`, но в то же время ввели новый класс `TileDecorator`. В нем не реализован метод `getWealthFactor()`, поэтому он должен быть объявлен абстрактным. Кроме того, мы определили конструктор, которому передается объект типа `Tile`, а ссылка на него сохраняется в свойстве `$tile`. Мы объявили это свойство защищенным (`protected`), чтобы сделать его доступным из дочерних классов. Переопределим далее классы `Pollution` и `Diamond`:

```
// Листинг 10.25
class DiamondDecorator extends TileDecorator
{
    public function getWealthFactor(): int
    {
        return $this->tile->getWealthFactor() + 2;
    }
}

// Листинг 10.26
class PollutionDecorator extends TileDecorator
{
    public function getWealthFactor(): int
    {
        return $this->tile->getWealthFactor() - 4;
    }
}
```

Каждый из этих классов расширяет класс `TileDecorator`. Это означает, что у них имеется ссылка на объект типа `Tile`. Когда вызывается метод

getWealthFactor(), каждый из этих классов сначала вызывает такой же метод по ссылке на объект типа Tile, а затем выполняет собственную коррекцию значения.

Используя композицию и делегирование подобным образом, мы легко можем комбинировать объекты во время выполнения программы. Все объекты, создаваемые по данному шаблону, расширяют класс Tile, поэтому клиентскому коду совсем не обязательно знать, какой именно комбинацией объектов он оперирует. Можно быть уверенным, что метод getWealthFactor() доступен для любого объекта типа Tile независимо от того, декорирует ли он "за сценой" другой объект:

```
// Листинг 10.27
$tile = new Plains();
print $tile->getWealthFactor(); // 2
```

Класс Plains является компонентом системы, поэтому его конструктор просто возвращает значение 2:

```
// Листинг 10.28
$tile = new DiamondDecorator(new Plains());
print $tile->getWealthFactor(); // 4
```

В объекте типа DiamondDecorator хранится ссылка на объект типа Plains. Перед прибавлением собственного значения 2 он вызывает метод getWealthFactor() из объекта типа Plains:

```
// Листинг 10.29
$tile = new PollutionDecorator(
            new DiamondDecorator(new Plains()));
print $tile->getWealthFactor(); // 0
```

В объекте типа PollutionDecorator хранится ссылка на объект типа DiamondDecorator, а у того — собственная ссылка на другой объект типа Tile. Диаграмма классов для данного примера показана на рис. 10.5.

Данную модель очень легко расширять. В нее совсем не трудно добавлять новые декораторы и компоненты. Имея в своем распоряжении декораторы, можно строить очень гибкие структуры времени выполнения. Класс компонентов системы (в данном примере — Plains) можно заметно видоизменить разными способами, не прибегая к встраиванию всей совокупности видоизменений в иерархию классов. Проще говоря, это означает, что можно создать загрязненную равнину (объект типа Plains) с месторождением алмазов, не создавая объект типа PollutedDiamondPlains.

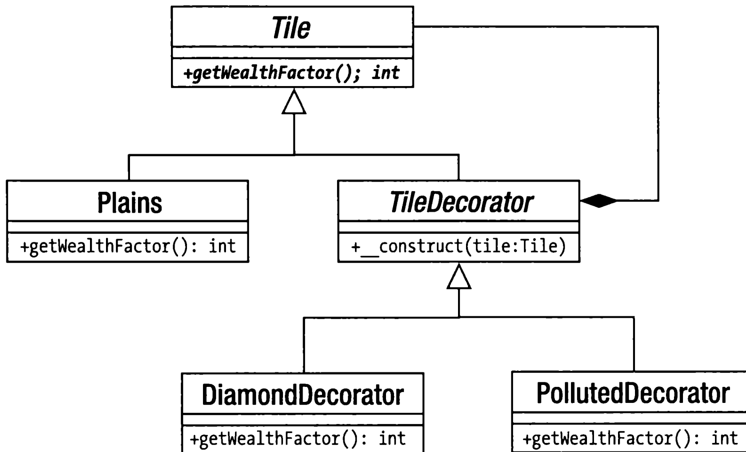


Рис. 10.5. *Диаграмма классов в соответствии с шаблоном Decorator*

В соответствии с шаблоном Decorator строятся конвейеры, очень удобные для создания фильтров. Классы декораторов очень эффективно используются в пакете `java.io` языка Java. Программист, разрабатывающий клиентский код, может комбинировать объекты декораторов с основными компонентами, чтобы внедрить фильтрацию, буферизацию, сжатие и прочие операции в базовые методы наподобие `read()`. В рассматриваемом здесь примере веб-запрос может быть также преобразован в конфигурируемый конвейер. Ниже приведен пример простой реализации, в которой применяется шаблон Decorator:

```
// Листинг 10.30
class RequestHelper
{
}

// Листинг 10.31
abstract class ProcessRequest
{
    abstract public function process(RequestHelper $req): void;
}

// Листинг 10.32
class MainProcess extends ProcessRequest
{
    public function process(RequestHelper $req): void
    {
        print __CLASS__ . ": выполнение запроса\n";
```

```
    }
}

// Листинг 10.33
abstract class DecorateProcess extends ProcessRequest
{
    public function __construct(protected ProcessRequest
                                $processrequest)
    {
    }
}
```

Как и ранее, мы определяем абстрактный суперкласс (`ProcessRequest`), конкретный компонент (`MainProcess`) и абстрактный декоратор (`DecorateProcess`). Метод `MainProcess::process()` не делает ничего, кроме вывода сообщения о том, что он был вызван. В классе `DecorateProcess` сохраняется ссылка на объект типа `ProcessRequest`, указывающая на один из его дочерних объектов. Ниже приведены примеры простых классов конкретных декораторов:

```
// Листинг 10.34
class LogRequest extends DecorateProcess
{
    public function process(RequestHelper $req): void
    {
        print __CLASS__ . ": запрос журналирования\n";
        $this->processrequest->process($req);
    }
}

// Листинг 10.35
class AuthenticateRequest extends DecorateProcess
{
    public function process(RequestHelper $req): void
    {
        print __CLASS__ . ": запрос аутентификации\n";
        $this->processrequest->process($req);
    }
}

// Листинг 10.36
class StructureRequest extends DecorateProcess
{
    public function process(RequestHelper $req): void
    {
        print __CLASS__ . ": данные структурирующего запроса\n";
```

```
        $this->processrequest->process($req);
    }
}
```

Каждый метод process(), прежде чем вызвать метод process() объекта типа ProcessRequest, на который делается ссылка, выводит сообщение. Теперь экземпляры этих классов можно комбинировать во время выполнения программы, чтобы создавать фильтры, выполняющие различные действия по запросу, причем в разном порядке. Ниже приведен пример кода для комбинирования объектов из всех упоминаемых здесь конкретных классов в одном фильтре:

```
// Листинг 10.37
$process = new AuthenticateRequest(
    new StructureRequest(
        new LogRequest(
            new MainProcess()
        )
    )
);
$process->process(new RequestHelper());
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
popp\ch10\batch07\AuthenticateRequest: запрос аутентификации
popp\ch10\batch07\StructureRequest: данные структурирующего запроса
popp\ch10\batch07\LogRequest: запрос журналирования
popp\ch10\batch07\MainProcess: выполнение запроса
```

На заметку На самом деле мы привели пример шаблона корпоративных приложений Intercepting Filter. Этот шаблон описан в упоминавшейся ранее книге *Core J2EE Patterns: Best Practices and Design Strategies.*

Следствия

Как и шаблон Composite, шаблон Decorator может показаться трудным для понимания. Важно помнить, что композиция и наследование вступают в действие одновременно. Поэтому класс LogRequest наследует свой интерфейс от класса ProcessRequest, но, в свою очередь, служит в качестве оболочки для другого объекта типа ProcessRequest.

Объект декоратора формирует оболочку вокруг дочернего объекта, поэтому очень важно поддерживать интерфейс настолько разреженным, насколько это возможно. Если создать базовый класс со многими функ-

циональными возможностями, то объекты декораторов будут вынуждены делегировать эти функции всем открытым методам того объекта, который они содержат. Это можно сделать в абстрактном классе декоратора, но в конечном итоге образуется такая тесная связь, которая может привести к программным ошибкам.

Некоторые программисты создают декораторы, не разделяющие общий тип с объектами, которые они видоизменяют. И до тех пор, пока они работают в рамках того же интерфейса, что и эти объекты, такая стратегия оказывается эффективной. При этом можно выгодно пользоваться встроенными методами-перехватчиками для автоматизации делегирования, реализовав метод `__call()`, чтобы перехватывать вызовы несуществующих методов и автоматически вызывать тот же самый метод дочернего объекта. Но в конечном счете теряется безопасность, которую обеспечивает проверка типа класса. В приведенных выше примерах клиентский код мог потребовать в своем списке аргументов объект типа `Tile` или `ProcessRequest` и быть уверенным в его интерфейсе независимо от того, является ли этот объект сильно декорированным.

Шаблон Facade

Вам, возможно, уже приходилось встраивать сторонние системы в свои проекты. И независимо от того, является ли сторонний код объектно-ориентированным, он, как правило, очень сложный, длинный и непонятный. А ваш код может, в свою очередь, вызвать трудности у программиста клиентского кода, которому требуется всего лишь получить доступ к нескольким функциональным средствам. Шаблон Facade — это удобное средство предоставления простого и понятного интерфейса для сложных систем.

Проблема

Обычно в процессе проектирования системы объем исходного кода, который на самом деле полезен только в пределах самой системы, постепенно увеличивается. В удачно спроектированных системах разработчики с помощью классов определяют понятный общедоступный интерфейс и прячут поглубже внутреннее содержание системы. Но не всегда очевидно, какие именно части системы должны использоваться в клиентском коде, а какие лучше скрыть.

Работая с такими подсистемами, как веб-форумы или приложения для создания галерей, вам, возможно, приходилось делать глубокие вызовы логики кода. Если исходный код подсистемы должен со временем меняться, а ваш код взаимодействует с ним в самых разных местах, то по мере развития подсистемы у вас могут возникнуть серьезные трудности при сопровождении вашего кода.

Аналогично, когда вы создаете собственные системы, имеет смысл разнести отдельные ее части на разные уровни. Как правило, первый уровень отвечает за логику приложения, второй — за взаимодействие с базой данных, третий — за представление данных и т.д. Вы должны стремиться поддерживать эти уровни независимыми один от другого, насколько это возможно, чтобы изменение в одной части проекта минимально отражалось на других частях. Если код одного уровня тесно интегрирован в код другого уровня, этой цели будет трудно достичь.

Ниже приведен пример специально запутанного процедурного кода, в котором простая задача получения информации из текстовых файлов и ее преобразования в данные объекта превращается в нечто очень сложное:

```
// Листинг 10.38
function getProductFileLines(string $file): array
{
    return file($file);
}

function getProductObjectFromId(string $id,
                                string $productname): Product
{
    // Поиск в некоторой базе данных
    return new Product($id, $productname);
}

function getNameFromLine(string $line): string
{
    if (preg_match("/.*-(.*)\s\d+/", $line, $array))
    {
        return str_replace('_', ' ', $array[1]);
    }
    return '';
}

function getIDFromLine($line): int | string
{
    if (preg_match("/^(\d{1,3})-/", $line, $array))
```

```
    {
        return $array[1];
    }
    return -1;
}

class Product
{
    public string $id;
    public string $name;
    public function __construct(string $id, string $name)
    {
        $this->id = $id;
        $this->name = $name;
    }
}
```

Допустим, что внутреннее содержимое приведенного выше кода сложнее, чем есть на самом деле, и нам придется воспользоваться им, чтобы не переписывать его заново. Допустим также, что приведенные ниже строки из файла требуется преобразовать в массив объектов:

```
234-Свитер_женский 55
532-Шляпа_мужская 44
```

С этой целью нам придется вызвать все упомянутые выше функции (ради краткости мы не извлекаем из исходных строк последнее число, обозначающее цену на товар):

```
// Листинг 10.39
$lines = getProductFileLines(__DIR__ . '/test2.txt');
$objects = [];

foreach ($lines as $line)
{
    $id = getIDFromLine($line);
    $name = getNameFromLine($line);
    $objects[$id] = getProductObjectFromID($id, $name);
}

print_r($objects);
```

Вывод имеет следующий вид:

```
Array
(
    [234] => Product Object
```

```
    (
        [id] => 234
        [name] => Свитер женский
    )
    [532] => Product Object
    (
        [id] => 532
        [name] => Шляпа мужская
    )
)
```

Если мы вызовем эти функции непосредственно в проекте, как показано выше, то наш код будет тесно вплетен в подсистему, которая в нем используется. И это может вызвать осложнения, если подсистема изменится или если мы решим отказаться от нее полностью. Поэтому нам следует внедрить шлюз между системой и остальным кодом.

Реализация

В качестве примера ниже приведено определение простого класса, предоставляющего интерфейс для процедурного кода, упоминавшегося в предыдущем разделе:

```
// Листинг 10.40
class ProductFacade
{
    private array $products = [];
    public function __construct(private string $file)
    {
        $this->compile();
    }
    private function compile(): void
    {
        $lines = getProductFileLines($this->file);
        foreach ($lines as $line)
        {
            $id = getIDFromLine($line);
            $name = getNameFromLine($line);
            $this->products[$id] =
                    getProductObjectFromID($id, $name);
        }
    }
    public function getProducts(): array
    {
        return $this->products;
    }
```

```
    public function getProduct(string $id): ? \Product
    {
        if (isset($this->products[$id]))
        {
            return $this->products[$id];
        }
        return null;
    }
}
```

С точки зрения клиентского кода доступ к объектам типа `Product` из текстового файла теперь намного упрощен:

```
// Листинг 10.41
$facade = new ProductFacade(__DIR__ . '/test2.txt');
$object = $facade->getProduct("234");
```

Следствия

В основу шаблона Facade на самом деле положено очень простое понятие, которое означает создание единственной точки входа на другой уровень или в подсистему. В итоге мы получаем ряд преимуществ, поскольку отдельные части проекта отделяются одна от другой. Программистам клиентского кода полезно и удобно иметь доступ к простым методам, которые выполняют понятные и очевидные действия. Это дает возможность сократить количество ошибок, сосредоточив обращение к подсистеме в одном месте, и тогда изменения в этой подсистеме вызовут сбой в предсказуемом месте. Кроме того, классы, создаваемые в соответствии с шаблоном Facade, сводят к минимуму ошибки в сложных подсистемах, где внутренние функции могли бы в противном случае неверно использоваться в клиентском коде.

Несмотря на всю простоту шаблона Facade, можно очень легко забыть воспользоваться им, особенно если вы знакомы с подсистемой, с которой работаете. Но здесь необходимо, конечно, найти золотую середину. С одной стороны, преимущества создания простых интерфейсов для сложных систем очевидны, а с другой — можно необдуманно абстрагировать сначала системы, а затем сами абстракции. Если вы делаете значительные упрощения для пользы клиентского кода и/или ограждаете его от систем, которые могут изменяться, то у вас, вероятно, есть все основания для реализации шаблона Facade.

Резюме

В этой главе мы рассмотрели несколько способов организации классов и объектов в системе. В частности, мы остановились на том, что композицию можно использовать для обеспечения гибкости там, где этого не может дать наследование. В шаблонах Composite и Decorator наследование применяется для поддержки композиции и определения общего интерфейса, который дает гарантии для клиентского кода.

Мы также показали, что в этих шаблонах эффективно используется делегирование. И наконец здесь был рассмотрен простой, но очень эффективный шаблон Facade. Это один из тех проектных шаблонов, которыми программисты пользуются годами, даже не зная, что он так называется. Шаблон Facade позволяет создать удобную и понятную точку входа для уровня или подсистемы. В языке PHP шаблон Facade применяется также для создания объектов-оболочек, в которые инкапсулированы блоки процедурного кода.

ГЛАВА 11

Выполнение задач и представление результатов

В этой главе мы начнем действовать более активно, в частности рассмотрим шаблоны, которые помогут решить такие задачи, как интерпретация мини-языка и инкапсуляция алгоритма.

В этой главе будут рассмотрены следующие вопросы.

- *Шаблон Interpreter.* Построение интерпретатора мини-языка, предназначенного для создания приложений, которые можно написать на языке сценариев.
- *Шаблон Strategy.* Определение алгоритмов в системе и их инкапсуляция в собственные типы.
- *Шаблон Observer.* Создание перехватчиков для извещения несовместимых объектов о событиях в системе.
- *Шаблон Visitor.* Выполнение операции над всеми узлами иерархического дерева объектов.
- *Шаблон Command.* Создание командных объектов, которые можно сохранять и передавать.
- *Шаблон Null Object.* Применение бездействующих объектов вместо пустых значений.

Шаблон Interpreter

Компиляторы и интерпретаторы языков программирования обычно пишутся на других языках программирования (по крайней мере, так было с самого начала). Например, сам интерпретатор языка PHP написан на языке C. Точно так же, каким бы странным это не показалось, на языке PHP можно определить и создать интерпретатор собственного языка. Безусловно, любой интерпретатор, создаваемый подобным образом, будет работать сравнительно медленно и окажется в какой-то степени ограни-

ченным. Однако, как будет показано в этой главе, мини-языки могут оказаться очень полезными.

Проблема

Создавая на PHP интерфейсы для веб или командной строки, мы, по существу, предоставляем пользователю доступ к функциональным средствам. При разработке интерфейса приходится находить компромисс между эффективностью и простотой использования. Как правило, чем больше возможностей предоставляется пользователю, тем более усложненным и запутанным становится интерфейс. Это, конечно, поможет делу, если удачно спроектировать интерфейс. Но если 90% пользователей употребляют один и тот же ряд функций (примерно треть от общего их числа), то вряд ли стоит расширять функциональные возможности приложения. Возможно, будет даже решено упростить систему в расчете на большинство пользователей. Но как быть с 10% пользователей, применяющих расширенные возможности системы? Им, вероятно, можно помочь как-то иначе. В частности, предложив таким пользователям предметно-ориентированный язык программирования, который обычно называется "Domain Specific Language" (DSL), можно действительно повысить потенциал приложения.

На самом деле в нашем распоряжении уже имеется язык программирования, который называется "PHP". Ниже показано, как разрешить пользователям создавать на нем сценарии в проектируемой системе:

```
// Листинг 11.1
$form_input = $_REQUEST['form_input'];
// Содержит: "print file_get_contents('/etc/passwd');"
eval( $form_input );
```

Очевидно, что такой способ поддержки написания сценариев пользователями нельзя назвать нормальным. Но если вам неясно, почему, то этому есть две причины: безопасность и сложность. Проблема безопасности хорошо продемонстрирована в данном примере. Разрешая пользователям выполнять код PHP через сценарий, мы, по существу, предоставляем им доступ к серверу, на котором выполняется сценарий. Проблема сложности столь же серьезна. Независимо от того, насколько понятен исходный код веб-приложения, маловероятно, чтобы среднестатистический пользователь смог легко его расширить, особенно с помощью только окна браузера.

Мини-язык может оказать помощь в решении обеих этих проблем. Можно сделать его достаточно гибким, уменьшить вероятность каких-ли-

бо нарушений из-за вмешательства пользователя и сосредоточиться на самом необходимом.

Рассмотрим в качестве примера приложение для авторской разработки викторин. Создатели викторины придумывают вопросы и устанавливают правила оценки ответов ее участников. Существует требование, что ответы на вопросы викторины должны оцениваться без вмешательства человека, хотя некоторые ответы пользователи могут вводить в текстовых полях.

Вот пример вопроса такой викторины:

```
Сколько членов в группе разработки проектных шаблонов?
```

В качестве правильных ответов можно принять "четыре" или "4". Можно даже создать веб-интерфейс, позволяющий создателям викторины применить регулярное выражение для оценки ответов:

```
^4|четыре$
```

Но, к сожалению, не все разработчики владеют регулярными выражениями. И чтобы облегчить их задачу, можно реализовать более удобный для пользователей механизм оценки ответов:

```
$input equals "4" or $input equals "четыре"
```

В данном случае предлагается язык, в котором поддерживаются переменные, операция `equals` и булева логика (логические операции `or` и `and`). Программисты любят все как-то называть, поэтому назовем такой язык "MarkLogic". Этот язык должен легко расширяться, поскольку нетрудно предвидеть немало запросов на более сложные функции. Оставим пока что в стороне вопрос синтаксического анализа входных данных и сосредоточимся на механизме соединения отдельных элементов во время выполнения программы для формирования ответа. Именно здесь, как и следовало ожидать, пригодится шаблон Interpreter (Интерпретатор).

Реализация

Мини-язык MarkLogic состоит из выражений, вместо которых подставляются некоторые значения. Как следует из табл. 11.1, даже в таком скромном языке, как MarkLogic, приходится отслеживать немало элементов.

В табл. 11.1 приведены имена в форме EBNF. А что такое EBNF? Это синтаксический метаязык, который можно использовать для описания грамматики языка. Сокращение "EBNF" означает "Extended Backus-Naur Form" (расширенная форма Бэкуса–Наура). Она состоит из набора строк,

которые называются *продукциями*. Каждая продукция состоит из имени и описания, которое принимает форму ссылок на другие продукции и на терминалы (т.е. элементы, которые сами не состоят из ссылок на другие продукции). Ниже показано, как можно описать грамматику мини-языка MarkLogic с помощью EBNF:

```
Expr     = operand { orExpr | andExpr }
Operand  = ( '(' expr ')' | ? string literal ? | variable ) {eqExpr}
orExpr   = 'or' operand
andExpr  = 'and' operand
eqExpr   = 'equals' operand
variable = '$' , ? word ?
```

***Таблица 11.1.** Элементы грамматики мини-языка MarkLogic*

Описание	Метаидентификатор EBNF	Имя класса	Пример
Переменная	`variable`	`VariableExpression`	`$input`
Строковый литерал	`stringLiteral`	`LiteralExpression`	`"четыре"`
Логическое И	`andExpr`	`BooleanAndExpression`	`$input equals '4' and $other equals '6'`
Логическое ИЛИ	`orExpr`	`BooleanOrExpression`	`$input equals '4' or $other equals '6'`
Проверка равенства	`eqExpr`	`BooleanEqualsExpression`	`$input equals '4'`

Некоторые знаки имеют специальный смысл (они должны быть вам знакомы по регулярным выражениям), например знак `'*'` означает "нуль или больше", а знак `'|'` — "или". Группировать элементы можно с помощью скобок. Так, в приведенном выше примере выражение (`expr`) состоит из лексемы `operand`, за которой следует нуль или больше лексем, `orExpr` или `andExpr`. В качестве лексемы `operand` может служить выражение в скобках, строковая константа (продукция для него опущена) или переменная, после которой следует нуль или больше лексем `eqExpr`. Как только вы поймете, как ссылаться в одной продукции на другую, форма EBNF станет очень простой для чтения и понимания.

На рис. 11.1 элементы грамматики рассматриваемого здесь языка представлены в виде диаграммы классов.

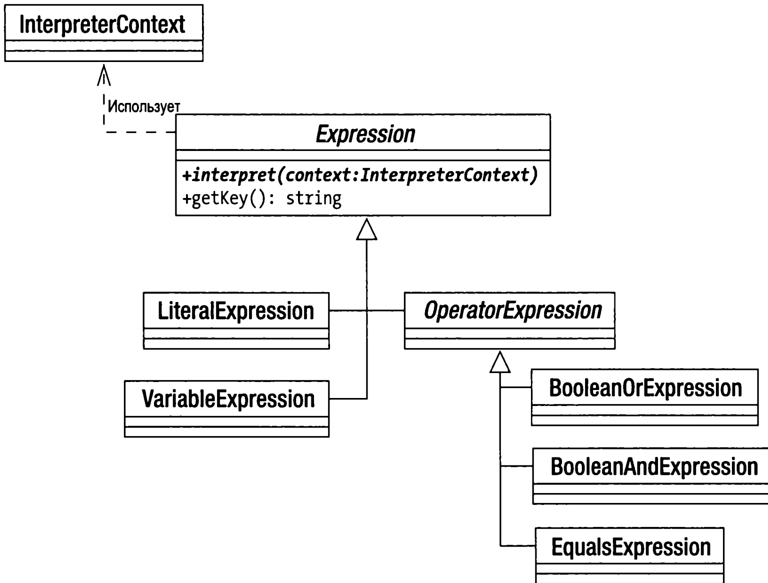


Рис. 11.1. *Классы, образующие мини-язык MarkLogic в соответствии с шаблоном Interpreter*

Как видите, класс `BooleanAndExpression` и родственные ему классы унаследованы от класса `OperatorExpression`. Дело в том, что все эти классы выполняют операции над другими объектами типа `Expression`. А объекты типа `VariableExpression` и `LiteralExpression` оперируют непосредственно значениями.

Во всех объектах типа `Expression` реализован метод `interpret()`, определенный в абстрактном базовом классе `Expression`. Методу `interpret()` передается объект типа `InterpreterContext`, который служит в качестве общего хранилища данных. Каждый объект типа `Expression` может сохранять данные в объекте типа `InterpreterContext`, который затем передается другим объектам типа `Expression`. В базовом классе `Expression` реализован также метод `getKey()`, возвращающий уникальный дескриптор и упрощающий извлечение данных из объекта типа `InterpreterContext`. Рассмотрим практическое применение мини-языка MarkLogic на примере реализации класса `Expression`:

```
// Листинг 11.2
abstract class Expression
{
    private static int $keycount = 0;
    private string $key;
    abstract public function interpret(InterpreterContext $context);
    public function getKey(): string
    {
        if (! isset($this->key))
        {
            self::$keycount++;
            $this->key = (string)self::$keycount;
        }
        return $this->key;
    }
}

// Листинг 11.3
class LiteralExpression extends Expression
{
    private mixed $value;
    public function __construct(mixed $value)
    {
        $this->value = $value;
    }
    public function interpret(InterpreterContext $context): void
    {
        $context->replace($this, $this->value);
    }
}

// Листинг 11.4
class InterpreterContext
{
    private array $expressionstore = [];
    public function replace(Expression $exp, mixed $value): void
    {
        $this->expressionstore[$exp->getKey()] = $value;
    }
    public function lookup(Expression $exp): mixed
    {
        return $this->expressionstore[$exp->getKey()];
    }
}

// Листинг 11.5
$context = new InterpreterContext();
```

```
$literal = new LiteralExpression('четыре');
$literal->interpret($context);
print $context->lookup($literal) . "\n";
```

Вот что получится в результате выполнения приведенного выше фрагмента кода:

```
четыре
```

Начнем анализ данного примера кода с класса `InterpreterContext`. Как видите, он на самом деле представляет собой только внешний интерфейс для ассоциативного массива `$expressionstore`, который служит для хранения данных. Методу `replace()` передаются ключ и значение, которые сохраняются в ассоциативном массиве `$expressionstore`. В качестве ключа используется объект типа `Expression`, а значение может быть любого типа. В классе `InterpreterContext` также реализован метод `lookup()` для извлечения данных.

В классе `Expression` определены абстрактный метод `interpret()` и конкретный метод `getKey()`, оперирующий статическим значением счетчика. Именно оно и возвращается в качестве дескриптора выражения. Этот метод используется в методах `InterpreterContext::lookup()` и `InterpreterContext::replace()` для индексирования данных.

В классе `LiteralExpression` определен конструктор, которому передается значение аргумента. Методу `interpret()` необходимо передать объект типа `InterpreterContext`. В нем просто вызывается метод `replace()` этого объекта, которому передаются ключ (ссылка на сам объект типа `LiteralExpression`) и значение переменной `$value`. В методе `replace()` объекта типа `InterpreterContext` для определения численного значения ключа используется метод `getKey()`. По мере исследования других классов типа `Expression` подобный шаблонный подход станет для вас уже привычным. Метод `interpret()` всегда сохраняет свои результаты в объекте типа `InterpreterContext`.

В приведенный выше пример включен также фрагмент клиентского кода, в котором создаются экземпляры объектов типа `InterpreterContext` и `LiteralExpression` (со значением `'четыре'`). Объект типа `InterpreterContext` передается методу `LiteralExpression::interpret()`. Этот метод сохраняет пару “ключ-значение” в объекте типа `InterpreterContext`, из которого затем извлекается значение, вызывая метод `lookup()`.

Определим теперь оставшийся терминальный класс. `VariableExpression` немного сложнее:

```
// Листинг 11.6
class VariableExpression extends Expression
{
    public function __construct(private string $name,
                                private mixed $val =
                                    null)
    {
    }
    public function interpret(InterpreterContext $context): void
    {
        if (! is_null($this->val))
        {
            $context->replace($this, $this->val);
            $this->val = null;
        }
    }
    public function setValue(mixed $value): void
    {
        $this->val = $value;
    }
    public function getKey(): string
    {
        return $this->name;
    }
}

// Листинг 11.7
$context = new InterpreterContext();
$myvar = new VariableExpression('input', 'четыре');
$myvar->interpret($context);
print $context->lookup($myvar) . "\n";
// Вывод: четыре

$newvar = new VariableExpression('input');
$newvar->interpret($context);
print $context->lookup($newvar) . "\n";
// Вывод: четыре

$myvar->setValue("пять");
$myvar->interpret($context);
print $context->lookup($myvar) . "\n";
// output: пять
print $context->lookup($newvar) . "\n";
// output: пять
```

Конструктору класса VariableExpression передаются два аргумента (имя и значение), которые сохраняются в свойствах объекта. В данном классе реализован метод setValue() для того, чтобы значение переменной можно было в любой момент изменить в клиентском коде.

В методе interpret() проверяется, имеет ли свойство $val ненулевое значение. Если в свойстве $val имеется некоторое значение, оно сохраняется в объекте типа InterpreterContext. Далее в свойстве $val устанавливается пустое значение null. Это делается для того, чтобы при повторном вызове метода interpret() не нарушилось значение переменной с тем же именем, сохраненной в объекте InterpreterContext другим экземпляром объекта VariableExpression. Возможности этой переменной довольно ограничены, поскольку ей могут быть присвоены только строковые значения. Если бы потребовалось расширить мини-язык MarkLogic, то пришлось бы сделать так, чтобы он работал с другими объектами Expression, содержащими результаты выполнения логических и других операций. Но пока что класс VariableExpression будет делать то, что нам от него нужно. Обратите внимание на то, что метод getKey() был переопределен, чтобы значения переменных были связаны с их именами, а не с произвольными статическими идентификаторами.

В мини-языке MarkLogic все операторные выражения оперируют двумя другими объектами типа Expression. Поэтому имеет смысл, чтобы они расширяли общий суперкласс. Ниже приведено определение класса OperatorExpression:

```
// Листинг 11.8
abstract class OperatorExpression extends Expression
{
    public function __construct(protected Expression $l_op,
                                protected Expression $r_op)
    {
    }
    public function interpret(InterpreterContext $context): void
    {
        $this->l_op->interpret($context);
        $this->r_op->interpret($context);
        $result_l = $context->lookup($this->l_op);
        $result_r = $context->lookup($this->r_op);
        $this->doInterpret($context, $result_l, $result_r);
    }
    abstract protected function doInterpret(
        InterpreterContext $context,
        $result_l,
```

```
        $result_r
    ): void;
}
```

В абстрактном классе OperatorExpression реализован метод interpret(), а также определен абстрактный метод doInterpret().

Конструктору этого класса передаются два объекта типа Expression для левого и правого операндов ($l_op и $r_op), которые он сохраняет в свойствах объекта.

Выполнение метода interpret() начинается с вызовов методов interpret() для обоих операндов, хранящихся в свойствах (если вы читали предыдущую главу, то, вероятно, заметили, что здесь был получен экземпляр класса в соответствии с шаблоном Composite). После этого в методе interpret() определяются значения левого и правого операндов с помощью вызова метода InterpreterContext::lookup() для каждого из них. Затем вызывается метод doInterpret(), чтобы дочерние классы могли решить, какую именно операцию необходимо выполнить над полученными значениями операндов.

На заметку Метод doInterpret() служит характерным примером применения шаблона Template Method. В соответствии с этим шаблоном в родительском классе определяется и вызывается абстрактный метод, реализация которого оставляется дочерним классам. Это может упростить разработку конкретных классов, поскольку совместно используемыми функциями управляет суперкласс, оставляя дочерним классам задачу сконцентрироваться на ясных и понятных целях.

Ниже приведено определение класса BooleanEqualsExpression, в котором проверяется равенство двух объектов типа Expression:

```
// Листинг 11.9
class BooleanEqualsExpression extends OperatorExpression
{
    protected function doInterpret(
        InterpreterContext $context,
        mixed $result_l,
        mixed $result_r
    ): void
    {
        $context->replace($this, $result_l == $result_r);
    }
}
```

В классе `BooleanEqualsExpression` реализован только метод `doInterpret()`. В нем проверяется равенство значений двух операндов, переданных из метода `interpret()` и полученных из объекта типа `InterpreterContext`.

Чтобы завершить набор классов типа `Expression`, приведем определение классов `BooleanOrExpression` и `BooleanAndExpression`:

```
// Листинг 11.10
class BooleanOrExpression extends OperatorExpression
{
    protected function doInterpret(
        InterpreterContext $context,
        mixed $result_l,
        mixed $result_r
    ): void
    {
        $context->replace($this, $result_l || $result_r);
    }
}

// Листинг 11.11
class BooleanAndExpression extends OperatorExpression
{
    protected function doInterpret(
        InterpreterContext $context,
        mixed $result_l,
        mixed $result_r
    ): void
    {
        $context->replace($this, $result_l && $result_r);
    }
}
```

Вместо проверки на равенство в классе `BooleanOrExpression` используется логическая операция ИЛИ, а полученный результат сохраняется в контексте с помощью метода `InterpreterContext::replace()`. В классе `BooleanAndExpression`, естественно, используется логическая операция "И".

Итак, мы получили код, с помощью которого можно интерпретировать фрагмент приведенного ранее выражения на мини-языке MarkLogic. Повторим его еще раз.

```
$input equals "4" or $input equals "четыре"
```

Ниже показано, как описать это выражение с помощью классов типа Expression:

```
// Листинг 11.12
$context = new InterpreterContext();
$input = new VariableExpression('input');
$statement = new BooleanOrExpression(
    new BooleanEqualsExpression($input,
                                new LiteralExpression('четыре')),
    new BooleanEqualsExpression($input, new LiteralExpression('4'))
);
```

В данном случае сначала создается переменная экземпляра $input, но ей пока еще не присваивается значение. Затем создается объект типа BooleanOrExpression, предназначенный для сравнения результатов, получаемых из двух объектов типа BooleanEqualsExpression. В первом случае сравнивается объект типа VariableExpression, который сохраняется в переменной $input, с объектом типа LiteralExpression, содержащим символьную строку "четыре". А во втором случае сравнивается значение переменной $input с объектом типа LiteralExpression, содержащим символьную строку "4".

Теперь, имея подготовленный оператор, мы подошли к тому, чтобы присвоить значение переменной $input и выполнить код:

```
// Листинг 11.13
foreach ([ "четыре", "4", "52" ] as $val)
{
    $input->setValue($val);
    print "$val:\n";
    $statement->interpret($context);

    if ($context->lookup($statement))
    {
        print "Правильный ответ!\n\n";
    }
    else
    {
        print "Вы ошиблись!\n\n";
    }
}
```

Фактически код запускается на выполнение три раза с тремя разными значениями. Но первый раз во временной переменной $val устанавливается значение "четыре", которое затем присваивается переменной $input

типа VariableExpression. Для этого вызывается метод setValue(). Далее вызывается метод interpret() для объекта типа Expression самого верхнего уровня (т.е. объекта типа BooleanOrExpression, который содержит ссылки на все другие выражения в операторе). Рассмотрим поэтапно, что же происходит непосредственно в этом вызове.

- Вызывается метод interpret() для свойства $l_op объекта по ссылке в переменной $statement (это первый объект типа BooleanEqualsExpression).
- Вызывается метод interpret() для свойства $l_op первого объекта типа BooleanEqualsExpression (в нем содержится ссылка на объект типа VariableExpression (переменная $input), для которого в настоящий момент установлено значение "четыре").
- Объект типа VariableExpression, содержащийся в переменной $input, записывает свое текущее значение в предоставленный объект типа InterpreterContext, вызывая метод InterpreterContext::replace().
- Вызывается метод interpret() для свойства $r_op первого объекта типа BooleanEqualsExpression (объекту типа LiteralExpression присвоено значение "четыре").
- Объект типа LiteralExpression регистрирует имя и значение своего ключа в объекте типа InterpreterContext.
- Первый объект типа BooleanEqualsExpression извлекает значения для свойств $l_op ("четыре") и $r_op ("четыре") из объекта типа InterpreterContext.
- Первый объект BooleanEqualsExpression сравнивает эти два значения, проверяя их на равенство, и регистрирует результат (логическое значение true) вместе с именем ключа в объекте типа InterpreterContext.
- На вершине иерархического дерева для объекта типа BooleanOrExpression в переменной $statement вызывается метод interpret() для его свойства $r_op. В итоге значение (в данном случае это логическое значение false) получается таким же образом, как при использовании свойства $l_op.

- Объект в переменной `$statement` извлекает значения для каждого из своих операндов из объекта `InterpreterContext` и сравнивает их в логической операции `||`. В итоге сравниваются логические значения `true` и `false`, и поэтому будет получено логическое значение `true`. И этот окончательный результат сохраняется в объекте типа `InterpreterContext`.

И все это — лишь первый шаг рассматриваемого здесь цикла. Окончательный результат приведен ниже:

```
четыре:
Правильный ответ!

4:
Правильный ответ!

52:
Вы ошиблись!
```

Возможно, вам придется прочитать этот раздел несколько раз, прежде чем все станет ясно. Вас может смутить здесь давний вопрос различия иерархических деревьев классов и объектов. Если *классы* `Expression` организуются в иерархию при наследовании, то *объекты* `Expression` объединяются в иерархическое дерево во время выполнения программы. Не забывайте об этом отличии, когда будете снова просматривать код из данного примера.

На рис. 11.2 показана полная диаграмма классов для рассматриваемого здесь примера.

Трудности реализации шаблона Interpreter

Как только основные классы будут подготовлены к реализации шаблона Interpreter, расширить их иерархию будет нетрудно. Цена, которую за это приходится платить, определяется лишь количеством классов, которые необходимо создать. Поэтому шаблон Interpreter пригоден в большей степени для разработки относительно небольших языков. Если же требуется полноценный язык программирования, то для этой цели лучше поискать сторонние инструментальные средства.

Классы, созданные в соответствии с шаблоном Interpreter, нередко выполняют очень схожие задачи, поэтому стоит следить за создаваемыми классами, чтобы не допустить их дублирования. Многие из тех, кто в пер-

вый раз обращается к шаблону Interpreter, после некоторых начальных экспериментов разочаровываются, обнаружив, что он не позволяет выполнять синтаксический анализ. Это означает, что мы пока еще не можем предложить пользователям хороший и удобный мини-язык. В приложении Б, "Простой синтаксический анализатор", приведен примерный вариант кода, иллюстрирующего одну из стратегий синтаксического анализа мини-языка.

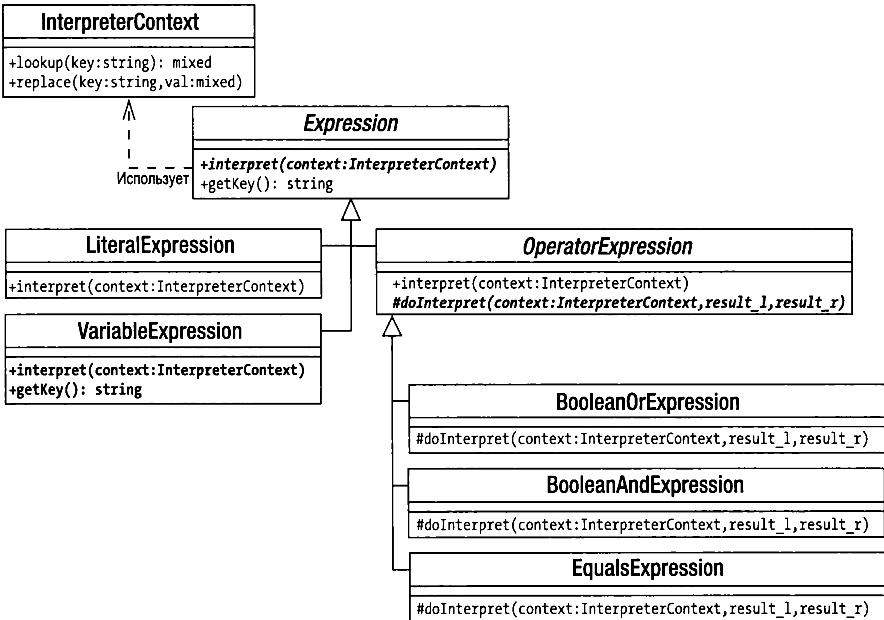


Рис. 11.2. *Пример применения шаблона Interpreter*

Шаблон Strategy

Разработчики часто пытаются наделить классы слишком большими обязанностями. И это понятно, поскольку вы обычно создаете класс, выполняющий ряд связанных вместе действий. Как и код, некоторые из этих действий приходится иногда изменять в зависимости от обстоятельств. В то же время классы необходимо разбивать на подклассы. И прежде чем вы это поймете, ваш проект начнет рассыпаться как карточный домик.

Проблема

В предыдущем разделе мы создали интерпретатор мини-языка, поэтому обратимся снова к примеру с викториной. Для викторин нужны вопросы, поэтому мы создали класс `Question` и реализовали в нем метод `mark()`. Но все это замечательно до тех пор, пока не приходится поддерживать разные механизмы оценки.

Допустим, нас попросили о поддержке простого мини-языка MarkLogic, в котором оценка получается в результате простого сравнения полученных ответов с заданными, а также проверки с помощью регулярных выражений. Первой мыслью, вероятно, будет создание подклассов, в которых учитываются подобные различия (рис. 11.3).

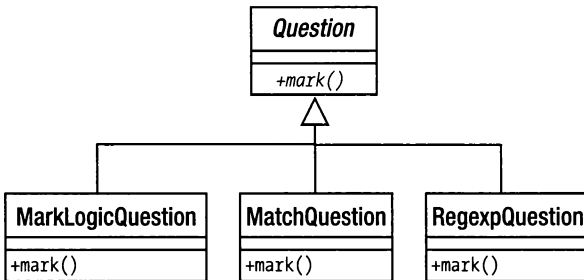


Рис. 11.3. *Определение подклассов в соответствии со стратегиями оценки*

Но такое положение дел устраивает нас до тех пор, пока оценка является единственным меняющимся аспектом класса. Но допустим, что нас попросили поддержать различные виды вопросов, например текстовые и мультимедийные. И тогда, если все эти факторы потребуется объединить в одном иерархическом дереве наследования (рис. 11.4), у нас возникнет затруднение.

В итоге не только увеличится количество классов в иерархии, но и неизбежно возникнет дублирование. Ведь логика оценки повторяется в каждой ветви иерархического дерева наследования. Всякий раз, когда вы обнаруживаете, что алгоритм повторяется в родственных узлах иерархического дерева наследования классов (в результате создания подклассов или в повторяющихся условных операторах), подумайте о том, чтобы выделить этот алгоритм в отдельный класс.

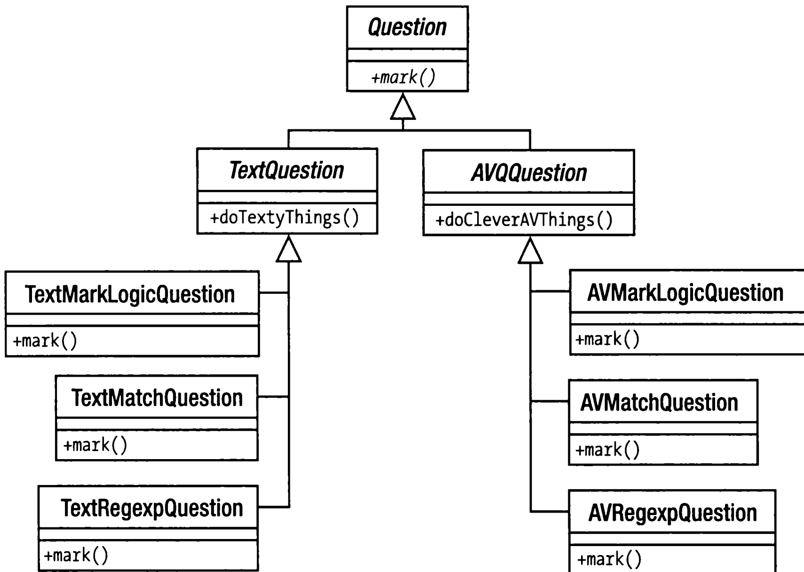


Рис. 11.4. *Определение подклассов на основе двух внешних факторов*

Реализация

Как и все лучшие проектные шаблоны, шаблон Strategy прост и эффективен. Если классы должны поддерживать несколько реализаций интерфейса (например, несколько механизмов оценки), то зачастую эти реализации лучше всего выделять и размещать в классе отдельного типа, а не расширять первоначальный класс, чтобы оперировать ими.

Так, в рассматриваемом здесь примере алгоритм оценки можно разместить в классе `Marker`. Полученная в итоге новая структура приведена на рис. 11.5.

Помните принцип "Банды четырех": "отдавайте предпочтение композиции, а не наследованию"? Данный пример наглядно демонстрирует этот принцип. Определяя и инкапсулируя алгоритмы оценки, мы сокращаем количество создаваемых подклассов и повышаем степень гибкости. Мы можем в любой момент добавить новые стратегии оценки, причем для этого нам не потребуется изменять классы `Question`. Этим классам известно лишь то, что в их распоряжении имеется экземпляр класса `Marker` и что через свой интерфейс он гарантированно поддерживает метод `mark()`, а подробности реализации — это вообще не их дело.

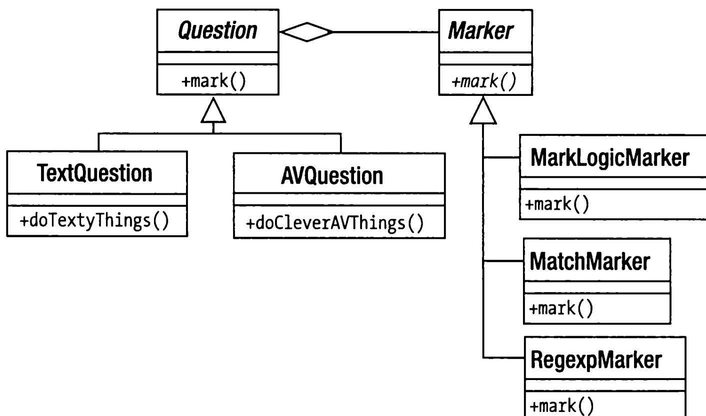


Рис. 11.5. *Выделение алгоритмов в их класс отдельного типа*

Ниже приведено определение семейства классов `Question`:

```
// Листинг 11.14
abstract class Question
{
    public function __construct(protected string $prompt,
                                protected Marker $marker)
    {
    }
    public function mark(string $response): bool
    {
        return $this->marker->mark($response);
    }
}

// Листинг 11.15
class TextQuestion extends Question
{
    // Обработка вопроса в текстовом виде
}

// Листинг 11.16
class AVQuestion extends Question
{
    // Обработка вопроса в мультимедийном виде
}
```

Как видите, реализация отличительных особенностей классов `TextQuestion` и `AVQuestion` оставлена на ваше усмотрение. Все необходимые функции обеспечиваются в базовом классе `Question`, в котором, кроме

всего прочего, хранится свойство, содержащее вопрос и объект типа `Marker`. Когда вызывается метод `Question::mark()` с ответом конечного пользователя, этот метод просто поручает решение задачи своему объекту `Marker`.

А теперь определим ряд простых объектов типа `Marker`:

```
// Листинг 11.17
abstract class Marker
{
    public function __construct(protected string $test)
    {
        abstract public function mark(string $response): bool;
    }
}

// Листинг 11.18
class MarkLogicMarker extends Marker
{
    private MarkParse $engine;
    public function __construct(string $test)
    {
        parent:: __construct($test);
        $this->engine = new MarkParse($test);
    }
    public function mark(string $response): bool
    {
        return $this->engine->evaluate($response);
    }
}

// Листинг 11.19
class MatchMarker extends Marker
{
    public function mark(string $response): bool
    {
        return ($this->test == $response);
    }
}

// Листинг 11.20
class RegexpMarker extends Marker
{
    public function mark(string $response): bool
    {
        return (preg_match("$this->test", $response) === 1);
    }
}
```

Как видите, в классах Marker, в общем-то, нет ничего неожиданного. Обратите внимание на то, что объект типа MarkParse предназначен для работы с простым синтаксическим анализатором, простая реализация которого приведена в приложении Б, "Простой синтаксический анализатор". Самое главное здесь — определение структуры, а не подробности реализации самих стратегий. Можно, например, заменить класс RegexpMarker классом MatchMarker, и это никак не повлияет на класс Question.

Разумеется, необходимо еще решить, каким образом выбирать конкретные объекты типа Marker. На практике для решения этой задачи применяются два подхода. В первом случае для выбора предпочитаемой стратегии оценки используются кнопки-переключатели, а во втором — сама структура условия оценки, в которой операция сравнения остается простой. Так, для оценки "пять" перед оператором мини-языка MarkLogic ставится двоеточие:

```
:$input equals 'пять'
```

а в регулярном выражении употребляются знаки косой черты:

```
/п.ть/
```

Ниже приведен код, позволяющий проверить классы Marker на практике:

```
// Листинг 11.21
$markers = [
               new RegexpMarker("/п.ть/"),
               new MatchMarker("пять"),
               new MarkLogicMarker('$input equals "пять"')
             ];

foreach ($markers as $marker)
{
    print get_class($marker) . "\n";
    $question =
        new TextQuestion("Сколько лучей у пятиконечной звезды",
                        $marker);
    foreach ([ "пять", "четыре" ] as $response)
    {
        print " Ответ: $response: ";

        if ($question->mark($response))
        {
            print "Верно\n";
        }
```

```
        else
        {
            print "Неверно\n";
        }
    }
}
```

Мы создали три объекта, содержащие стратегии оценки ответов, которые по очереди используются для создания объекта типа `TextQuestion`. Затем объект типа `TextQuestion` проверяется по двум вариантам ответа. Выполнение приведенного выше фрагмента кода приведет к следующему результату (включая пространства имен):

```
popp\ch11\batch02\RegexpMarker
    Ответ: пять: Верно
    Ответ: четыре: Неверно

popp\ch11\batch02\MatchMarker
    Ответ: пять: Верно
    Ответ: четыре: Неверно

popp\ch11\batch02\MarkLogicMarker
    Ответ: пять: Верно
    Ответ: четыре: Неверно
```

В этом примере мы передали введенные пользователем данные (они содержатся в переменной `$response`) от клиента к объекту стратегии оценки через метод `mark()`. Но иногда возникают ситуации, когда заранее неизвестно, сколько информации потребуется объекту стратегии оценки для выполнения своих обязанностей. Поэтому решение, какие именно данные следует получать, можно поручить самому объекту стратегии оценки, передав ему экземпляр клиентского объекта. И тогда объект стратегии оценки сам запросит у клиента нужные ему данные.

Шаблон Observer

Преимущества ортогональности уже обсуждались. Одной из целей разработчиков должно быть создание компонентов, которые можно изменять или перемещать с минимальным воздействием на другие компоненты. Если каждое изменение, которое вносится в один компонент, влечет за собой необходимость изменений в других местах кодовой базы, то задача разработки быстро раскрутится по спирали внесения и устранения программных ошибок.

Разумеется, об ортогональности можно только мечтать как о недостижимой цели. Компоненты проектируемой системы должны содержать встроенные ссылки на другие компоненты. Но для сведения таких ссылок к минимуму можно применить различные стратегии. Ранее были продемонстрированы различные примеры использования полиморфизма, в которых клиентскому коду понятен интерфейс компонента, но сам конкретный компонент может меняться во время выполнения программы.

Однако иногда может возникнуть потребность вбить еще больший клин между компонентами. Рассмотрим в качестве примера следующий класс, отвечающий за управление доступом пользователя к системе:

```
// Листинг 11.22
classLogin
{
    public const LOGIN_USER_UNKNOWN = 1;
    public const LOGIN_WRONG_PASS = 2;
    public const LOGIN_ACCESS = 3;
    private array $status = [];
    public function handleLogin(string $user, string $pass,
                                string $ip):
        bool
    {
        $isvalid = false;
        switch (rand(1, 3))
        {
        case 1:
            $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
            $isvalid = true;
            break;
        case 2:
            $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
            $isvalid = false;
            break;
        case 3:
            $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
            $isvalid = false;
            break;
        }
        print "возврат " .(($isvalid) ? "true" : "false") . "\n";
        return $isvalid;
    }
    private function setStatus(int $status,
                               string $user, string $ip): void
    {
        $this->status = [$status, $user, $ip];
    }
```

```
    public function getStatus(): array
    {
        return $this->status;
    }
}
```

Разумеется, в реальном примере метод handleLogin() должен проверять учетные данные пользователя, хранящиеся где-нибудь в системе. Но в данном случае класс Login имитирует процесс входа в систему с помощью функции rand(). Возможны три разных результата вызова метода handleLogin(). В частности, код состояния устанавливается в соответствии со значением константы LOGIN_ACCESS, LOGIN_WRONG_PASS или LOGIN_USER_UNKNOWN.

Класс Login выполняет функции привратника, охраняющего сокровища и тайны коммерческой деятельности вашей компании, и поэтому он может привлекать большое внимание как на стадии проектирования системы, так и при последующей ее эксплуатации. Например, вас могут вызвать в отдел маркетинга и попросить организовать ведение журнала регистрации IP-адресов пользователей системы. С этой целью вы можете добавить вызов соответствующего метода из класса Logger в вашей системе, как показано ниже:

```
// Листинг 11.23
public function handleLogin(string $user, string $pass,
                            string $ip): bool
{
    switch (rand(1, 3))
    {
    case 1:
        $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
        $isvalid = true;
        break;
    case 2:
        $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
        $isvalid = false;
        break;
    case 3:
        $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
        $isvalid = false;
        break;
    }

    Logger::logIP($user, $ip, $this->getStatus());
    return $isvalid;
}
```

Заботясь о безопасности, системные администраторы могут попросить вас вести учет неудачных попыток входа в систему. В этом случае вы можете вернуться к методу управления входом в систему и добавить новый вызов:

```
// Листинг 11.24
if (! $isvalid)
{
    Notifier::mailWarning(
        $user,
        $ip,
        $this->getStatus()
    );
}
```

В отделе развития коммерческой деятельности могут объявить о привязке к конкретному поставщику услуг Интернета (ISP) и попросить, чтобы после входа определенных пользователей в систему им высылались cookie-файлы. И так далее и тому подобное.

Все эти требования нетрудно удовлетворить, хотя и за счет изменений проектного решения. Класс `Login` вскоре станет очень плотно встроенным в данную конкретную систему. Его уже нельзя будет извлечь и включить в другой программный продукт, не проанализировав построчно его исходный код и не удалив все, что связано с прежней системой. Безусловно, это не так уж и трудно сделать, но в конечном счете вы скатитесь на путь простого копирования и вставки. Теперь, когда у вас имеются два похожих, но все же разных класса `Login` в ваших системах, вы обнаружите, что улучшение в одном классе приведет к необходимости аналогичных изменений в другом классе. И так будет продолжаться до тех пор, пока эти классы неизбежно перестанут быть похожими.

Так что же можно сделать, чтобы как-то спасти класс `Login`? Призвать на помощь шаблон Observer.

Реализация

В основу шаблона Observer положен принцип отвязки клиентских элементов (наблюдателей) от центрального класса (субъекта). Наблюдатели должны быть осведомлены, когда происходят события, о которых известно субъекту. В то же время нежелательно, чтобы у субъекта была жестко закодированная связь с классами его наблюдателей.

Чтобы достичь этого, можно разрешить наблюдателям регистрироваться в субъекте. С этой целью добавим в класс `Login` три новых метода, `attach()`, `detach()` и `notify()`, с помощью интерфейса `Observable`, как показано ниже:

```
// Листинг 11.25
interface Observable
{
    public function attach(Observer $observer): void;
    public function detach(Observer $observer): void;
    public function notify(): void;
}

// Листинг 11.26
class Login implements Observable
{
    private array $observers = [];
    public const LOGIN_USER_UNKNOWN = 1;
    public const LOGIN_WRONG_PASS = 2;
    public const LOGIN_ACCESS = 3;
    public function attach(Observer $observer): void
    {
        $this->observers[] = $observer;
    }
    public function detach(Observer $observer): void
    {
        $this->observers = array_filter(
            $this->observers,
            function($a) use($observer)
        {
            return (!($a === $observer));
        }
        );
    }
    public function notify(): void
    {
        foreach ($this->observers as $obs)
        {
            $obs->update($this);
        }
    }
    // ...
}
```

Итак, класс `Login` управляет списком объектов наблюдателей. Они могут быть добавлены в этот список третьей стороной с помощью метода `attach()` и удалены из него с помощью метода `detach()`. Метод `notify()` вызывается, чтобы сообщить наблюдателям, что произошло нечто интересное. Он просто перебирает в цикле список наблюдателей, вызывая для каждого из них метод `update()`.

В самом классе `Login` метод `notify()` вызывается из его метода `handleLogin()`, определяемого следующим образом:

```
// Листинг 11.27
public function handleLogin(string $user, string $pass,
                            string $ip): bool
{
    switch (rand(1, 3))
    {
    case 1:
        $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
        $isvalid = true;
        break;
    case 2:
        $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
        $isvalid = false;
        break;
    case 3:
        $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
        $isvalid = false;
        break;
    }

    $this->notify();
    return $isvalid;
}
```

А теперь определим интерфейс для класса `Observer`:

```
// Листинг 11.28
interface Observer
{
    public function update(Observable $observable): void;
}
```

Любой объект, использующий этот интерфейс, может быть добавлен к классу `Login` с помощью метода `attach()`. Ниже приведен конкретный экземпляр:

```
// Листинг 11.29
class LoginAnalytics implements Observer
{
    public function update(Observable $observable): void
    {
        // Небезопасно с точки зрения типов!
        $status = $observable->getStatus();
        print __CLASS__ . ": обработка информации о состоянии\n";
    }
}
```

Обратите внимание на то, как в объекте наблюдателя используется переданный ему экземпляр типа Observable, чтобы получить дополнительные сведения о событии. Класс субъекта должен предоставить методы, которые могут вызвать наблюдатели, чтобы узнать о состоянии субъекта. В данном случае мы определили метод getStatus(), который наблюдатели могут вызвать, чтобы получить сведения о текущем состоянии субъекта.

Но такое дополнение выявляет также следующее затруднение. При вызове метода Login::getStatus() классу LoginAnalytics передается больше сведений, чем требуется для их безопасного использования. Этот вызов делается для объекта типа Observable, но нет никакой гарантии, что это также будет объект типа Login. Выйти из такой ситуации можно двумя путями. С одной стороны, можно расширить интерфейс Observable, включив в него объявление метода getStatus(). При этом, вероятно, придется переименовать интерфейс в нечто вроде ObservableLogin, чтобы показать, что он связан непосредственно с классом Login.

С другой стороны, можно сохранить интерфейс Observable общим, но сделать так, чтобы классы наблюдателей типа Observer были ответственными за работу с субъектами правильного типа. Они даже могут выполнять работу по присоединению себя к своему субъекту. А поскольку у нас будет не один наблюдатель типа Observer и мы собираемся выполнять некоторые служебные операции, общие для всех наблюдателей, то создадим абстрактный суперкласс, который будет заниматься этой рутинной работой:

```
// Листинг 11.30
abstract class LoginObserver implements Observer
{
    private Login $login;
    public function __construct(Login $login)
    {
        $this->login = $login;
        $login->attach($this);
    }
```

```
    public function update(Observable $observable): void
    {
        if ($observable === $this->login)
        {
            $this->doUpdate($observable);
        }
    }
    abstract public function doUpdate(Login $login): void;
}
```

Конструктору класса `LoginObserver` необходимо передать объект типа `Login`. Он сохраняет ссылку на него и вызывает метод `Login::attach()`. При вызове метода `update()` класса `LoginObserver` сначала проверяется, правильная ли ссылка на объект типа `Observable` ему передана. Затем вызывается шаблонный метод `doUpdate()`. Теперь можно создать ряд объектов типа `LoginObserver`, и все они могут быть уверены, что работают с объектом типа `Login`, а не только с любым прежним объектом, поддерживающим интерфейс `Observable`:

```
// Листинг 11.31
class SecurityMonitor extends LoginObserver
{
    public function doUpdate(Login $login): void
    {
        $status = $login->getStatus();

        if ($status[0] == Login::LOGIN_WRONG_PASS)
        {
            // Отправление письма сисадмину
            print __CLASS__ . ": письмо сисадмину\n";
        }
    }
}

// Листинг 11.32
class GeneralLogger extends LoginObserver
{
    public function doUpdate(Login $login): void
    {
        $status = $login->getStatus();
        // Добавление данных о входе в журнал
        print __CLASS__ . ": добавление данных о входе в журнал\n";
    }
}
```

```
// Листинг 11.33
class PartnershipTool extends LoginObserver
{
    public function doUpdate(Login $login): void
    {
        $status = $login->getStatus();
        // Проверка $ip-адреса
        // Установка cookie при соответствии списку
        print __CLASS__ .
            ": Установка cookie при соответствии списку\n";
    }
}
```

Теперь создание и подключение к субъекту класса типа `LoginObserver` выполняются сразу, во время инстанцирования:

```
// Листинг 11.34
$login = new Login();
new SecurityMonitor($login);
new GeneralLogger($login);
new PartnershipTool($login);
```

Таким образом, мы установили гибкую связь между классами субъектов и наблюдателей. Диаграмма классов для рассматриваемого здесь примера приведена на рис. 11.6.

В языке PHP обеспечивается встроенная поддержка шаблона Observer через входящее в комплект стандартное расширение SPL (Standard PHP Library — стандартная библиотека PHP), которое является набором инструментальных средств, помогающих решать распространенные задачи объектно-ориентированного программирования. Расширение SPL как универсальный аспект шаблона Observer состоит из следующих трех элементов: `SplObserver`, `SplSubject` и `SplObjectStorage`. В частности, `SplObserver` и `SplSubject` — это интерфейсы, которые являются точной аналогией интерфейсов `Observer` и `Observable` из примера, приведенного ранее в данном разделе, а `SplObjectStorage` — вспомогательный класс, обеспечивающий улучшенное сохранение и удаление объектов. Ниже приведена измененная версия реализации шаблона Observer:

```
// Листинг 11.35
class Login implements \SplSubject
{
    private \SplObjectStorage $storage;
    // ...
    public function __construct()
```

```
    {
        $this->storage = new \SplObjectStorage();
    }
    public function attach(\SplObserver $observer): void
    {
        $this->storage->attach($observer);
    }
    public function detach(\SplObserver $observer): void
    {
        $this->storage->detach($observer);
    }
    public function notify(): void
    {
        foreach ($this->storage as $obs)
        {
            $obs->update($this);
        }
    }
    // ...
}

// Листинг 11.36
abstract class LoginObserver implements \SplObserver
{
    public function __construct(private Login $login)
    {
        $login->attach($this);
    }
    public function update(\SplSubject $subject): void
    {
        if ($subject === $this->login)
        {
            $this->doUpdate($subject);
        }
    }
    abstract public function doUpdate(Login $login): void;
}
```

Что касается интерфейсов `SplObserver` и `SplSubject`, то не существует никаких реальных их отличий от интерфейсов `Observer` и `Observable`, за исключением, конечно, того, что больше не нужно объявлять интерфейсы, а нужно только изменить указания типов в параметрах методов в соответствии с новыми именами. А класс `SplObjectStorage` предоставляет действительно полезные услуги. Вы, вероятно, заметили, что в первоначальном примере при реализации метода `Login::detach()`

мы перебирали в цикле все объекты типа Observable, сохраненные в массиве $observable, чтобы найти и удалить объект аргумента. Класс SplObjectStorage выполняет всю эту рутинную работу за сценой. В нем реализованы методы attach() и detach(), а его объект можно указать для итерирования в цикле foreach, подобно массиву.

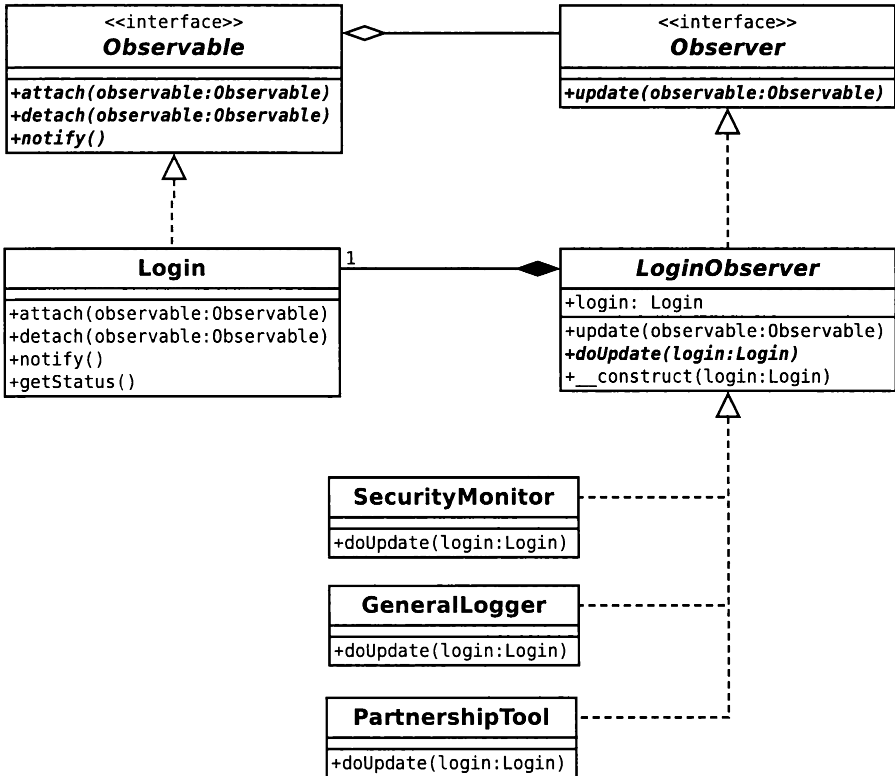


Рис. 11.6. Пример применения шаблона Observer

На заметку За более подробными сведениями о расширении SPL обращайтесь к документации на PHP по адресу http://www.php.net/spl. Там вы, в частности, найдете немало инструментальных средств итераторов. А о встроенном в PHP интерфейсе Iterator речь пойдет в главе 13, "Шаблоны баз данных".

Еще один подход к решению задачи взаимодействия субъекта типа Observable с наблюдателем типа Observer состоит в том, что методу update() можно передать конкретные сведения о состоянии, а не экзем-

пляр субъекта типа `Observable`. Такой подход обычно выбирается в качестве скороспелого решения, поэтому в данном примере методу `update()` следовало бы передавать код состояния, имя пользователя и IP-адрес (возможно, в массиве ради переносимости), а не экземпляр класса `Login`. Благодаря этому нам не пришлось бы создавать в классе `Login` отдельный метод для получения состояния. С другой стороны, класс субъекта хранит немало сведений о состоянии, так что передача его экземпляра методу `update()` дает наблюдателям возможность действовать намного более гибко.

Кроме того, тип аргумента можно зафиксировать полностью, чтобы класс `Login` оперировал только классом наблюдателя определенного типа (вероятно, типа `LoginObserver`). Для этого следует предусмотреть во время выполнения программы проверку типов объектов, переданных методу `attach()`. В противном случае, возможно, придется полностью пересмотреть интерфейс `Observable`.

И в данной ситуации мы снова воспользовались композицией во время выполнения программы, чтобы построить гибкую и расширяемую модель. Класс `Login` можно извлечь из контекста и включить в совсем другой проект безо всяких изменений, и там он сможет работать с другим рядом наблюдателей.

Шаблон Visitor

Как было показано ранее, многие шаблоны предназначены для создания структур данных во время выполнения программы, следуя принципу, по которому композиция обладает большей гибкостью, чем наследование. Характерным тому примером служит вездесущий шаблон Composite. Когда вы работаете с коллекциями объектов, вам может понадобиться выполнять над структурой различные операции, в результате которых задействуется каждый ее отдельный компонент. Такие операции могут быть встроены в сами компоненты. Ведь компоненты зачастую лучше всего предназначены для взаимного вызова.

Такой подход не лишен недостатков. У вас не всегда имеются сведения обо всех операциях, которые, возможно, потребуется выполнить над структурой. Если вы добавляете поддержку новых операций в классы от случая к случаю, то в результате ваш интерфейс может обрасти ненужными функциями, которые на самом деле не характерны для него. Как вы уже, вероятно, догадались, эти недостатки позволяет устранить шаблон Visitor.

Проблема

Напомним пример применения шаблона Composite из предыдущей главы, в которой для игры мы создали армию компонентов таким образом, чтобы можно было попеременно обращаться как с целым, как и с отельными его частями. В данном примере было показано, что операции можно встраивать в компоненты. Как правило, объекты листьев выполняют операцию, а составные объекты вызывают свои дочерние объекты, чтобы выполнить эту операцию, как показано ниже:

```
// Листинг 11.37
class Army extends CompositeUnit
{
    public function bombardStrength(): int
    {
        $strength = 0;

        foreach ($this->units() as $unit)
        {
            $strength += $unit->bombardStrength();
        }

        return $strength;
    }
}

// Листинг 11.38
class LaserCanonUnit extends Unit
{
    public function bombardStrength(): int
    {
        return 44;
    }
}
```

Если конкретная операция является неотъемлемой частью и ее выполнение входит в обязанности класса составного типа, то никакого затруднения не возникает. Но существуют и второстепенные задачи, которые могут не так удачно вписаться в интерфейс.

Ниже приведен пример операции, выводящей текстовую информацию об узлах-листьях. Ее можно добавить в качестве метода в абстрактный класс `Unit`:

```
// Листинг 11.39
abstract class Unit
{
    // ...
    public function textDump($num = 0): string
    {
        $txtout = "";
        $pad = 4 * $num;
        $txtout .= sprintf("%{$pad}s", "");
        $txtout .= get_class($this) . ": ";
        $txtout .= "Огневая мощь: "
                . $this->bombardStrength() . "\n";
        return $txtout;
    }
    // ...
}
```

Данный метод можно затем перекрыть в классе `CompositeUnit` следующим образом:

```
// Листинг 11.40
abstract class CompositeUnit extends Unit
{
    // ...
    public function textDump($num = 0): string
    {
        $txtout = parent::textDump($num);

        foreach ($this->units as $unit)
        {
            $txtout .= $unit->textDump($num + 1);
        }

        return $txtout;
    }
}
```

Теперь, продолжая в том же духе, мы должны создать методы для подсчета количества боевых единиц в иерархическом дереве для сохранения компонентов в базе данных и для подсчета количества единиц пищи, потребленной армией. Почему нужно включать эти методы в интерфейс составного типа? На это есть только один действительно убедительный ответ. Эти несопоставимые операции следует включать в интерфейс потому, что именно здесь метод, выполняющий подобную операцию, может легко получить доступ к связанным вместе узлам в структуре составного типа.

И хотя легкость обхода иерархического дерева действительно является одной из примечательных особенностей шаблона Composite, это еще не означает, что каждая операция, в которой требуется обойти иерархическое дерево, должна по этой причине требовать себе место в интерфейсе в соответствии с шаблоном Composite.

Итак, задача формулируется следующим образом: необходимо извлечь все выгоды из легкого обхода иерархического дерева, предоставляющего структуру объектов, но сделать это, не раздувая чрезмерно интерфейс.

Реализация

Начнем с интерфейсов и определим в абстрактном классе `Unit` метод `accept()`:

```
// Листинг 11.41
abstract class Unit
{
    // ...
    public function accept(ArmyVisitor $visitor): void
    {
        $refthis = new \ReflectionClass(get_class($this));
        $method = "Посещение " . $refthis->getShortName();
        $visitor->$method($this);
    }
    protected function setDepth($depth): void
    {
        $this->depth = $depth;
    }
    public function getDepth(): int
    {
        return $this->depth;
    }
}
```

Как видите, метод `accept()` ожидает, что ему будет передан объект типа `ArmyVisitor`. В языке PHP можно динамически определить метод в объекте типа `ArmyVisitor`, который требуется вызвать. Поэтому мы составили имя такого метода на основании имени текущего класса и вызвали его для переданного в качестве параметра объекта типа `ArmyVisitor`. Так, если текущим является класс `Army`, вызывается метод `ArmyVisitor::visitArmy()`, а если это класс `TroopCarrier`, то вызывается метод `ArmyVisitor::visitTroopCarrier()` и т.п. Это дает возможность не реализовывать метод `accept()` в каждом листовом узле иерар-

хии классов. Для большего удобства мы ввели еще два метода, `getDepth()` и `setDepth()`, чтобы извлекать и сохранять величину глубины вложенности элемента в иерархическом дереве. В частности, метод `setDepth()` вызывается из метода `CompositeUnit::addUnit()` родителем элемента, когда тот добавляет его в иерархическое дерево:

```
// Листинг 11.42
abstract class CompositeUnit extends Unit
{
    // ...
    public function addUnit(Unit $unit): void
    {
        foreach ($this->units as $thisunit)
        {
            if ($unit === $thisunit)
            {
                return;
            }
        }

        $unit->setDepth($this->depth + 1);
        $this->units[] = $unit;
    }
    public function accept(ArmyVisitor $visitor): void
    {
        parent::accept($visitor);

        foreach ($this->units as $thisunit)
        {
            $thisunit->accept($visitor);
        }
    }
}
```

В приведенный выше фрагмент кода было также включено определение метода `accept()`. Вызов метода `Unit::accept()` аналогичен вызову метода `visit()` для предоставляемого объекта типа `ArmyVisitor`. На самом деле метод `accept()` переопределяет аналогичную операцию в родительском классе, поэтому данный метод позволяет делать следующее:

- вызывать подходящий метод `visit()` для текущего компонента;
- передавать объект посетителя всем текущим дочерним элементам с помощью метода `accept()` при условии, что текущий компонент является составным.

Мы должны также определить интерфейс для класса ArmyVisitor. И какую-то информацию для этого должны предоставить методы accept(). В классе посетителя должны быть определены методы accept() для каждого конкретного класса в иерархии. Это позволит обеспечить разные функциональные возможности для различных объектов. В приведенной ниже версии данного класса определен также стандартный метод visit(), который вызывается автоматически, если в реализующих его классах принимается решение не выполнять специальную обработку для определенных классов Unit:

```
// Листинг 11.43
abstract class ArmyVisitor
{
    abstract public function visit(Unit $node);
    public function visitArcher(Archer $node): void
    {
        $this->visit($node);
    }
    public function visitCavalry(Cavalry $node): void
    {
        $this->visit($node);
    }
    public function visitLaserCanonUnit(LaserCanonUnit $node): void
    {
        $this->visit($node);
    }
    public function visitTroopCarrierUnit(TroopCarrierUnit $node):
        void
    {
        $this->visit($node);
    }
    public function visitArmy(Army $node): void
    {
        $this->visit($node);
    }
}
```

Теперь остается только вопрос предоставления реализаций класса ArmyVisitor, и мы готовы его решить. Ниже приведен простой пример кода, выводящего текстовую информацию и заново реализованного на основе объекта типа ArmyVisitor:

```
// Листинг 11.44
class TextDumpArmyVisitor extends ArmyVisitor
{
```

```
    private string $text = "";
    public function visit(Unit $node): void
    {
        $txt = "";
        $pad = 4 * $node->getDepth();
        $txt .= sprintf("%{$pad}s", "");
        $txt .= get_class($node) . ": ";
        $txt .= "Огневая мощь: " . $node->bombardStrength() . "\n";
        $this->text .= $txt;
    }
    public function getText(): string
    {
        return $this->text;
    }
}
```

Теперь рассмотрим клиентский код, а далее пройдемся по всему процессу:

```
// Листинг 11.45
$main_army = new Army();
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCanonUnit());
$main_army->addUnit(new Cavalry());
$textdump = new TextDumpArmyVisitor();
$main_army->accept($textdump);
print $textdump->getText();
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
popp\ch11\batch08\Army: Огневая мощь: 50
    popp\ch11\batch08\Archer: Огневая мощь: 4
    popp\ch11\batch08\LaserCannonUnit: Огневая мощь: 44
    popp\ch11\batch08\Cavalry: Огневая мощь: 2
```

В данном примере мы создаем объект типа `Army`. Этот объект является составным, и поэтому в нем имеется метод `addUnit()`, с помощью которого можно добавлять дополнительные объекты типа `Unit`. Затем мы создаем объект типа `TextDumpArmyVisitor` и передаем его методу `Army::accept()`. В этом методе на ходу составляется имя вызываемого метода и далее делается вызов `TextDumpArmyVisitor::visitArmy()`. В данном случае мы не обеспечили специальной обработки объектов типа `Army`, и поэтому составленный в итоге вызов передается обобщенному методу `visit()` наряду со ссылкой на объект типа `Army`. Он вызывает свои методы (включая и вновь добавленный метод `getDepth()`, который сообщает всем, кому это следует знать, степень глубины вложения элемента в дереве композиции), чтобы сформировать итоговые дан-

ные. По завершении вызова метода `visitArmy()` выполняется операция `Army::accept()`, в ходе которой метод `accept()` по очереди вызывается для дочерних объектов, и ему передается объект посетителя. Таким образом, в классе `ArmyVisitor` происходит посещение каждого объекта в иерархическом дереве.

Добавив всего пару методов, мы создали в итоге механизм, посредством которого можно внедрять новые функциональные возможности в классы составного типа, не ухудшая их интерфейс и не дублируя в большом количестве код обхода иерархического дерева.

В некоторых клетках в рассматриваемой здесь игре армии должны платить налоги. Сборщик налогов посещает армию и берет плату за каждую обнаруженную в ней боевую единицу (или подразделение). Разные подразделения должны платить разные суммы налогов. И здесь мы можем воспользоваться преимуществами специализированных методов в классе посетителя:

```
// Листинг 11.46
class TaxCollectionVisitor extends ArmyVisitor
{
    private int $due = 0;
    private string $report = "";
    public function visit(Unit $node): void
    {
        $this->levy($node, 1);
    }
    public function visitArcher(Archer $node): void
    {
        $this->levy($node, 2);
    }
    public function visitCavalry(Cavalry $node): void
    {
        $this->levy($node, 3);
    }
    public function
        visitTroopCarrierUnit(TroopCarrierUnit $node): void
    {
        $this->levy($node, 5);
    }
    private function levy(Unit $unit, int $amount): void
    {
        $this->report .= "Налог для " . get_class($unit);
        $this->report .= ": $amount\n";
        $this->due += $amount;
    }
```

```
    public function getReport(): string
    {
        return $this->report;
    }
    public function getTax(): int
    {
        return $this->due;
    }
}
```

В этом простом примере мы не пользуемся объектами типа `Unit`, передаваемыми различным методам типа `visit`, непосредственно. Но в то же время мы пользуемся специализированной природой этих методов, взимая различные суммы налогов, исходя из конкретного типа вызывающего объекта типа `Unit`.

Ниже приведен конкретный пример клиентского кода:

```
// Листинг 11.47
$main_army = new Army();
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCanonUnit());
$main_army->addUnit(new Cavalry());
$taxcollector = new TaxCollectionVisitor();
$main_army->accept($taxcollector);
print $taxcollector->getReport();
print "ВСЕГО: ";
print $taxcollector->getTax() . "\n";
```

Объект типа `TaxCollectionVisitor` передается, как и прежде, методу `accept()` объекта типа `Army`. И снова объект типа `Army` передает ссылку на себя методу `visitArmy()`, прежде чем вызывать метод `accept()` для своих дочерних объектов. Но этим компонентам ничего неизвестно о том, какие именно операции выполняет их посетитель. Они просто взаимодействуют с их общедоступным интерфейсом, причем каждый из них добросовестно передаст методу соответствующего типа ссылку на самого себя.

В дополнение к методам, определенным в классе `ArmyVisitor`, класс `TaxCollectionVisitor` предоставляет два итоговых метода — `getReport()` и `getTax()`. В результате вызова этих методов предоставляются данные, которые и предполагалось получить:

```
Налог для popp\ch11\batch08\Army: 1
Налог для popp\ch11\batch08\Archer: 2
Налог для popp\ch11\batch08\LaserCanonUnit: 1
Налог для popp\ch11\batch08\Cavalry: 3
ВСЕГО: 7
```

На рис. 11.7 показаны классы из данного примера.

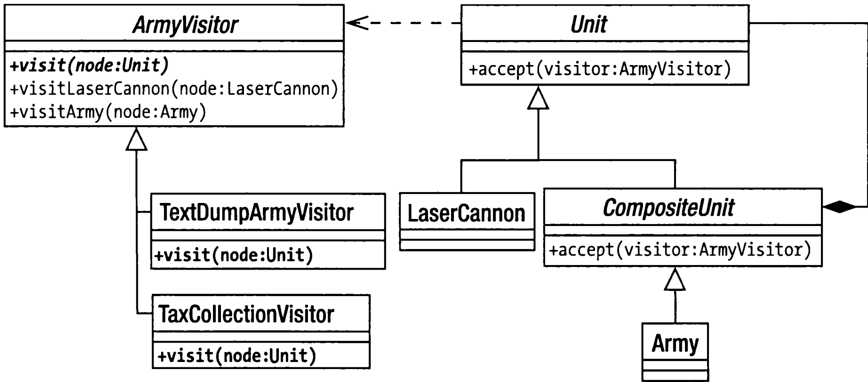


Рис. 11.7. *Пример применения шаблона Visitor*

Трудности реализации шаблона Visitor

Шаблон Visitor является еще одним проектным шаблоном, сочетающим в себе простоту и эффективность. Но применяя этот шаблон, не следует забывать некоторые особенности его реализации.

Несмотря на то что шаблон Visitor идеально приспособлен для применения вместе с шаблоном Composite, на самом деле его можно применять с любой коллекцией объектов. В частности, его можно применять вместе со списком объектов, в котором каждый объект сохраняет ссылку на родственные объекты (т.е. на узлы одного уровня в иерархическом дереве).

Но, вынося операции наружу, мы рискуем нарушить инкапсуляцию — нам, возможно, придется раскрывать внутреннее содержимое посещаемых объектов, чтобы посетители могли сделать с ними что-нибудь полезное. Например, в первом примере применения шаблона Visitor мы были вынуждены ввести дополнительный метод в интерфейсе класса `Unit`, чтобы предоставить необходимые сведения для объектов типа `TextDumpArmyVisitor`. С этой дилеммой мы уже сталкивались в шаблоне Observer.

Сам процесс итерации отделен от операций, которые выполняют объекты посетителей, поэтому в какой-то степени придется ослабить контроль. Нельзя, например, легко и просто создать метод `visit()`, который что-то делает как до, так и после обхода дочерних узлов иерархического дерева. Это затруднение можно разрешить, например, передав ответственность за выполнение итерации объектам посетителей. Но дело в том, что это может

привести к дублированию кода обхода узлов иерархического дерева в каждом посетителе.

По умолчанию код обхода узлов иерархического дерева лучше оставлять в посещаемых объектах. Но его вынесение наружу даст следующее явное преимущество: возможность изменять способ обработки посещаемых объектов в зависимости от конкретного посетителя.

Шаблон Command

В последние годы я редко завершал веб-проект, не применяя этот шаблон. Командные объекты, первоначально рассматриваемые в контексте проектирования графического интерфейса, способствуют удачному проектированию корпоративного приложения, поддерживают разделение между уровнями контроллера (обработки запросов и диспетчеризации) и моделью предметной области (логики работы приложения). Проще говоря, шаблон Command помогает создавать хорошо организованные системы, которые легко поддаются расширению.

Проблема

Во всех проектируемых системах должно приниматься решение, что делать в ответ на запрос пользователя. В языке PHP процесс принятия решения часто осуществляется с помощью ряда отдельных контактных страниц. Выбирая такую страницу (`feedback.php`), пользователь явно дает понять функциональным средствам и интерфейсу, что́ именно ему требуется. Все чаще программисты на PHP делают выбор в пользу единственной контактной страницы (этот подход будет обсуждаться в следующей главе). Но в любом случае получатель запроса должен передать полномочия уровню, более тесно связанному с логикой приложения. Такое делегирование полномочий особенно важно, если пользователь может сделать запросы к разным страницам. В противном случае дублирование кода в проекте неизбежно.

Допустим, что у нас имеется проект с рядом задач, которые необходимо решить. В частности, проектируемая система одним пользователям должна разрешать входить в систему, а другим — оставлять отклики. Мы можем создать страницы `login.php` и `feedback.php`, на которых решаются поставленные задачи, получая экземпляры соответствующих специализированных классов, которые и выполнят всю рутинную работу. К сожалению, пользовательский интерфейс проектируемой системы редко соответствует

в точности задачам, для решения которых предназначена данная система. Например, функции входа в систему и оставления откликов могут понадобиться на каждой странице. Если страницы должны решать много разных задач, то мы, вероятно, должны представлять себе задачи как нечто такое, что можно инкапсулировать. Таким способом мы упростим внедрение новых задач в проектируемую систему и построим границу между уровнями этой системы. И это, конечно, приведет нас к шаблону Command.

Реализация

Интерфейс для командного объекта вряд ли может быть проще! Он требует реализовать только один метод `execute()`. На рис. 11.8 класс `Command` представлен как абстрактный. На таком уровне простоты его можно было бы определить как интерфейс. Но в данном случае мы склонны воспользоваться абстрактным классом, потому что базовый класс также может предоставить полезные общие функциональные возможности для своих производных объектов.

Рис. 11.8. *Класс* `Command`

В шаблоне Command может быть до трех других участников: клиент, который создает экземпляр командного объекта; вызывающий участник, который использует этот объект; а также получатель, которым оперирует команда.

Получатель может быть передан командному объекту клиентским кодом через конструктор или запрошен из какого-нибудь объекта фабрики. Более предпочтительным выглядит второй подход, который состоит в применении конструктора класса `Command` без аргументов. И тогда экземпляры всех объектов класса `Command` могут быть созданы одним и тем же способом.

Ниже приведено определение абстрактного класса `Command`:

```
// Листинг 11.48
abstract class Command
{
    abstract public function execute(CommandContext $context): bool;
}
```

А вот как выглядит определение конкретного класса, расширяющего класс Command:

```
// Листинг 11.49
class LoginCommand extends Command
{
    public function execute(CommandContext $context): bool
    {
        $manager = Registry::getAccessManager();
        $user = $context->get('username');
        $pass = $context->get('pass');
        $user_obj = $manager->login($user, $pass);

        if (is_null($user_obj))
        {
            $context->setError($manager->getError());
            return false;
        }

        $context->addParam("user", $user_obj);
        return true;
    }
}
```

Класс LoginCommand предназначен для работы с объектом воображаемого класса AccessManager. Этот класс предназначен для управления механизмом входа пользователей в систему. Обратите внимание на то, что методу Command::execute() требуется передать объект типа CommandContext (в упоминавшейся ранее книге *Core J2EE Patterns* он называется RequestHelper). Это механизм, посредством которого данные запроса могут быть переданы объектам типа Command, а ответы могут быть отправлены назад, на уровень представления. Использовать объект подобным способом полезно, потому что командам можно передать разные параметры, не нарушая интерфейс. Класс CommandContext, по существу, служит оболочкой для ассоциативного массива переменных, хотя он часто расширяется для выполнения дополнительных полезных задач. Ниже приведен пример простой реализации класса CommandContext:

```
// Листинг 11.50
class CommandContext
{
    private array $params = [];
    private string $error = "";
    public function __construct()
    {
```

```
        $this->params = $_REQUEST;
    }
    public function addParam(string $key, $val): void
    {
        $this->params[$key] = $val;
    }
    public function get(string $key): string
    {
        if (isset($this->params[$key]))
        {
            return $this->params[$key];
        }
        return null;
    }
    public function setError($error): string
    {
        $this->error = $error;
    }
    public function getError(): string
    {
        return $this->error;
    }
}
```

Итак, снабженный объектом типа `CommandContext`, класс `LoginCommand` может получить доступ к полученным данным запроса: имени пользователя и паролю. Мы используем простой класс `Registry` со статическими методами для формирования общих объектов. Он возвращает объект типа `AccessManager`, которым будет оперировать класс `LoginCommand`. Если при выполнении метода `login()` из объекта типа `AccessManager` происходит ошибка, то сообщение об ошибке сохраняется в объекте типа `CommandContext`, чтобы его можно было отобразить на уровне представления, а также возвращается логическое значение `false`. Если же все в порядке, то метод `execute()` из объекта типа `LoginCommand` просто возвращает логическое значение `true`. Обратите внимание на то, что сами объекты типа `Command` выполняют мало логических операций. Они проверяют входные данные, обрабатывают ошибки и сохраняют данные, а также вызывают методы из других объектов для выполнения соответствующих операций. Если окажется, что логика приложения вкралась в командные классы, это послужит явным признаком, что следует рассмотреть возможность реорганизации кода. Ведь такой код способствует дублированию, поскольку он неизбежно копируется и вставляется из одной команды в другую. Следует хотя бы выяснить, к чему относятся такие

функциональные средства. Возможно, их лучше всего переместить в объекты, содержащие логику приложения, или на уровень фасада. В рассматриваемом здесь примере нам еще недостает клиента — класса, формирующего командные объекты, и вызывающего участника — класса, который оперирует сформированной командой. Самый простой способ выбрать экземпляры командных объектов, которые требуется получить в веб-проекте, — указать параметр в самом запросе. Ниже приведен пример упрощенной реализации клиента:

```
// Листинг 11.51
class CommandFactory
{
    private static string $dir = 'commands';
    public static function
        getCommand(string $action = 'Default'): Command
    {
        if (preg_match('/\W/', $action))
        {
            throw new \Exception("Неверные символы в команде");
        }

        $class = __NAMESPACE__ . "\\commands\\" .
            UCFirst(strtolower($action)) . "Command";

        if (! class_exists($class))
        {
            throw new CommandNotFoundException(
                            "Класс '$class' не обнаружен");
        }

        $cmd = new $class();
        return $cmd;
    }
}
```

Класс `CommandFactory` просто ищет определенный класс. Полностью квалифицированное имя класса создается из пространства имен класса `CommandFactory`, строки `"\commands\"` и параметра `$action` объекта `CommandContext`, который, в свою очередь, должен быть передан системе из запроса. Благодаря логике механизма автозагрузки классов проектного шаблона Composer нам не нужно явным образом включать класс в наш код. Если класс существует, объект инстанцируется и возвращается вызываемому коду. Можно также ввести еще больше операций проверки ошибок, чтобы убедиться, что найденный класс принадлежит семейству

Command и что конструктор не ожидает аргументов, но приведенный выше вариант полностью подходит для наших учебных целей. Преимущество такого подхода заключается в том, что новый объект типа Command можно в любой момент добавить в каталог commands, и система сразу же станет поддерживать его.

Теперь проще простого определить вызывающий объект, как показано ниже:

```
// Листинг 11.52
class Controller
{
    private CommandContext $context;
    public function __construct()
    {
        $this->context = new CommandContext();
    }
    public function getContext(): CommandContext
    {
        return $this->context;
    }
    public function process(): void
    {
        $action = $this->context->get('action');
        $action = (is_null($action)) ? "default" : $action;
        $cmd = CommandFactory::getCommand($action);

        if (! $cmd->execute($this->context))
        {
            // Обработка сбоя
        }
        else
        {
            // Удачный исход операции
        }
    }
}
```

Вот как выглядит код вызова класса:

```
// Листинг 11.53
$controller = new Controller();
$context = $controller->getContext();

$context->addParam('action', 'login');
$context->addParam('username', 'Иван');
$context->addParam('pass', 'tiddles');
```

```
$controller->process();

print $context->getError();
```

Прежде чем вызвать метод Controller::process(), мы имитируем веб-запрос, задавая его параметры в объекте типа CommandContext, экземпляр которого создан в конструкторе контроллера. В методе process() запрашивается значение параметра "action", и если оно не существует, то в качестве запасного варианта выбирается символьная строка "default". Затем метод process() делегирует объекту типа CommandFactory создание экземпляров командных объектов, после чего он вызывает метод execute() для возвращенного командного объекта. Обратите внимание на то, что контроллеру вообще неизвестно внутреннее содержимое командного объекта. Именно эта независимость от подробностей выполнения команды дает нам возможность вводить новые классы Command, не оказывая никакого воздействия на систему в целом.

Ниже приведен еще один класс типа Command:

```
// Листинг 11.54
class FeedbackCommand extends Command
{
    public function execute(CommandContext $context): bool
    {
        $msgSystem = Registry::getMessageSystem();
        $email = $context->get('email');
        $msg = $context->get('msg');
        $topic = $context->get('topic');
        $result = $msgSystem->send($email, $msg, $topic);
        if (! $result)
        {
            $context->setError($msgSystem->getError());
            return false;
        }
        return true;
    }
}
```

На заметку Мы еще вернемся к шаблону Command в главе 12, "Шаблоны корпоративных приложений", где будет представлена более полная реализация класса фабрики Command. Описанная здесь структура выполнения команд является упрощенной версией другого шаблона — Front Controller, — рассматриваемого далее в книге.

Этот класс будет вызван в ответ на получение из запроса символьной строки "feedback", соответствующей параметру 'action'. И для этого не придется вносить никаких изменений в контроллер или класс CommandFactory.

На рис. 11.9 приведены все участники шаблона Command.

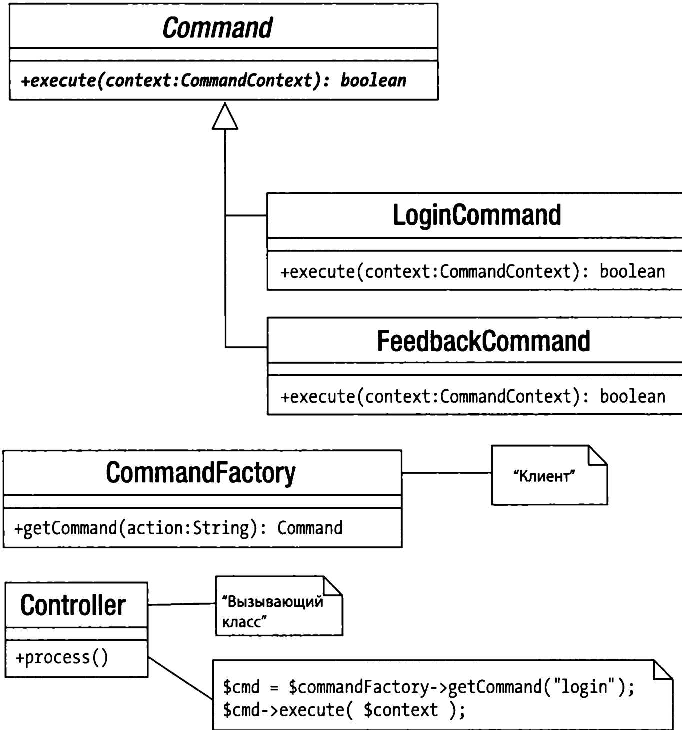


Рис. 11.9. *Участники шаблона Command*

Шаблон Null Object

Половина затруднений, с которыми сталкиваются программисты, связана с типами данных. Именно поэтому в языке PHP постепенно развивалась поддержка контроля типов в объявлениях методов и при возвращении из них значений. Если трудности вызывает обращение с переменной, содержащей значение неверного типа, то не меньше хлопот связано с полным отсутствием значения какого-нибудь типа в переменной. И это проис-

ходит постоянно, поскольку многие функции возвращают значение `null`, когда в них не удается сформировать какое-нибудь полезное значение. Избавить себя и других от этой напасти можно, воспользовавшись в своих проектах шаблоном Null Object. Как будет показано далее, если другие проектные шаблоны, рассмотренные ранее в этой главе, служат для того, чтобы сделать что-нибудь существенное, то шаблон Null Object предназначен для того, чтобы вообще ничего не делать, причем как можно корректнее.

Проблема

Если на метод возложена обязанность найти нужный объект и он не смог этого сделать, то все, что ему остается, — просигнализировать о неудачном исходе операции. Причиной тому может служить устаревшая информация, предоставленная вызывающим кодом, или недоступность искомого ресурса. Если неудачный исход операции имеет катастрофические последствия, то, возможно, следует выбрать вариант генерации соответствующего исключения. Но зачастую требуется чуть более снисходительное решение. В таком случае возврат значения `null` может оказаться удобным способом известить клиента о неудачном исходе операции.

Обратимся снова к рассматривавшемуся ранее примеру игры и допустим, что в классе `TileForces` отслеживаются сведения о боевых единицах, находящихся в отдельной клетке. В данной игре поддерживаются локально сохраняемые сведения о боевых единицах во всей системе, а компонент `UnitAcquisition` отвечает за преобразование этих метаданных в массив объектов.

Ниже приведено объявление класса `TileForces` и его конструктора:

```
// Листинг 11.55
class TileForces
{
    private int $x;
    private int $y;
    private array $units = [];
    public function __construct(int $x, int $y,
                                UnitAcquisition $acq)
    {
        $this->x = $x;
        $this->y = $x;
        $this->units = $acq->getUnits($this->x, $this->y);
    }
    // ...
}
```

Объект типа `TileForces` лишь поручает объекту типа `UnitAcquisition`, передаваемому его конструктору в качестве параметра, получить массив объектов типа `Unit`. Создадим фиктивный объект типа `UnitAcquisition` следующим образом:

```
// Листинг 11.56
class UnitAcquisition
{
    public function getUnits(int $x, int $y): array
    {
        // 1. Найти координаты x и y в локальных данных и
        //    получить список идентификаторов боевых единиц
        // 2. Обратиться к источнику и получить
        //    полноценные сведения о боевых единицах
        // А пока что подставим какие-нибудь фиктивные сведения о них
        $army = new Army();
        $army->addUnit(new Archer());
        $found = [
            new Cavalry(),
            null,
            new LaserCanonUnit(),
            $army
        ];
        return $found;
    }
}
```

В объекте данного класса сознательно не приведен процесс получения данных типа `Unit`. Но в реальной системе должен, конечно, выполняться поиск конкретных данных, а здесь мы довольствовались лишь несколькими операциями непосредственного получения экземпляров объектов. Обратите, однако, внимание на коварное пустое значение `null`, заданное в массиве `$found`. Оно требуется на тот случай, если, например, клиент этой сетевой игры хранит метаданные, которые уже не совпадают с текущим состоянием данных на сервере.

Имея в своем распоряжении массив объектов типа `Unit`, класс `TileForces` может предоставить некоторые функциональные возможности, как показано ниже:

```
// Листинг 11.57
// TileForces
public function firepower(): int
{
    $power = 0;
```

```
    foreach ($this->units as $unit)
    {
        $power += $unit->bombardStrength();
    }

    return $power;
}
```

А теперь проверим написанный нами код в действии:

```
// Листинг 11.58
$acquirer = new UnitAcquisition();
$tileforces = new TileForces(4, 2, $acquirer);
$power = $tileforces->firepower();
print "Огневая мощь: {$power}\n";
```

Из-за скрытого значения `null` этот код приводит к следующей ошибке:

```
Error: Call to a member function bombardStrength() on null
```

В методе `TileForces::firepower()` циклически итерируется массив `$units` и для каждого объекта типа `Unit` вызывается метод `bombard Strength()`. Но попытка вызвать этот метод для значения `null`, безусловно, приведет к ошибке. Наиболее очевидный выход из положения в данном случае состоит в том, чтобы проверять каждый элемент массива, прежде чем его обрабатывать:

```
// Листинг 11.59
// TileForces
public function firepower(): int
{
    $power = 0;
    foreach ($this->units as $unit)
    {
        if (! is_null($unit))
        {
            $power += $unit->bombardStrength();
        }
    }
    return $power;
}
```

Сама по себе обработка пустого значения не вызывает особых трудностей. Но представьте версию класса `TileForces`, в котором выполняются самые разные операции над элементами массива `$units`. Как только мы начнем размножать проверку с помощью функции `is_null()` в нескольких местах, мы снова придем к конкретному образцу недоброкачественно-

го кода. И зачастую избавиться от параллельных фрагментов клиентского кода можно, воспользовавшись полиморфизмом вместо нескольких условных операторов. Это можно сделать и в данном случае.

Реализация

Шаблон Null Object позволяет поручить классу ничего не делать с предполагаемым типом данных. В данном случае создадим по этому шаблону класс `NullUnit` следующим образом:

```
// Листинг 11.60
class NullUnit extends Unit
{
    public function bombardStrength(): int
    {
        return 0;
    }
    public function getHealth(): int
    {
        return 0;
    }
    public function getDepth(): int
    {
        return 0;
    }
}
```

В данном случае в классе `NullUnit` мы реализовали интерфейс `Unit` с помощью пустых методов, в которых ничего конкретного не делается. А теперь можно внести исправления в класс `UnitAcquisition` и создать объект типа `NullUnit` вместо пустого значения `null`:

```
// Листинг 11.61
public function getUnits(int $x, int $y): array
{
    $army = new Army();
    $army->addUnit(new Archer());
    $found = [
        new Cavalry(),
        new NullUnit(),
        new LaserCanonUnit(),
        $army
    ];
    return $found;
}
```

Теперь из клиентского кода в классе `TileForces` можно вызывать любые методы элементов массива `$units`, в том числе для объекта типа `NullUnit`, без каких бы то ни было осложнений и ошибок:

```
// Листинг 11.62
// TileForces
public function firepower(): int
{
    $power = 0;
    foreach ($this->units as $unit)
    {
        $power += $unit->bombardStrength();
    }
    return $power;
}
```

Проанализируйте любой крупный проект и подсчитайте количество некрасивых проверок, которые разработчикам пришлось ввести в методы, возвращающие пустые значения. Сколько таких проверок могло быть распространено по всему проекту, если бы большинство разработчиков пользовались шаблоном Null Object?

Безусловно, иногда *требуется* заранее знать, что придется оперировать пустым объектом. Самый очевидный способ сделать это состоит в том, чтобы проверить объект с помощью операции `instanceof`. Но это менее изящный способ, чем приведенная ранее проверка с помощью функции `is_null()`.

Возможно, лучшим решением может стать добавление приведенного ниже метода `isNull()` как в базовый класс (с возвратом логического значения `false`), так и в класс, созданный в соответствии с шаблоном Null Object (с возвратом логического значения `true`):

```
// Листинг 11.63
if (! $unit->isNull())
{
    // Некоторые действия
}
else
{
    print "null - ничего не делаем!\n";
}
```

Это дает нам обоюдовыгодное решение. С одной стороны, можно благополучно вызвать любой метод из объекта типа `NullUnit`, а с другой — можно запросить состояние любого объекта типа `Unit`.

Резюме

Этой главой мы завершили исследование шаблонов "Банды четырех". В ней мы разработали мини-язык и построили его интерпретатор в соответствии с шаблоном Interpreter. Кроме того, мы обнаружили в шаблоне Strategy другой способ применения композиции с целью увеличить степень гибкости и уменьшить потребность в повторном создании подклассов. А шаблон Observer позволяет решить задачу извещения совершенно разных и меняющихся компонентов о событиях в системе.

Затем мы снова воспользовались примером применения шаблона Composite и с помощью шаблона Visitor выяснили, как осуществить вызов каждого компонента из иерархического дерева и выполнить над ним многие операции. Мы также показали, каким образом шаблон Command может помочь в создании расширяемой многоуровневой системы. И наконец мы выяснили, как сэкономить на целом ряде проверок пустых значений, воспользовавшись шаблоном Null Object.

В следующей главе мы выйдем за рамки шаблонов "Банды четырех", чтобы исследовать некоторые проектные шаблоны, предназначенные для применения при разработке корпоративных приложений.

ГЛАВА 12

Шаблоны корпоративных приложений

Язык PHP, прежде всего, предназначен для применения в веб-среде. И благодаря существенно расширенной поддержке объектов можно выгодно воспользоваться преимуществами шаблонов, разработанных в контексте других объектно-ориентированных языков программирования, в частности — Java.

В этой главе мы будем пользоваться единственным примером для демонстрации всех описываемых в ней шаблонов. Но не забывайте, что, решив применить один шаблон, вы не обязаны использовать все шаблоны, которые подходят для работы с ним. Не считайте также, что представленные в этой главе реализации — это единственный способ применения рассматриваемых в ней шаблонов. Используйте приведенные здесь примеры для того, чтобы лучше понять суть описываемых здесь шаблонов, и не стесняйтесь извлекать из них все полезное для своих проектов.

В данной главе необходимо изложить довольно обширный материал, и поэтому это одна из самых больших и сложных глав в книге, а следовательно, прочитать ее за один присест будет трудно. Она разделена на введение и две основные части, что поможет вам освоить материал поэтапно.

Отдельные шаблоны сначала описаны в разделе "Краткий обзор архитектуры". И хотя они в какой-то степени взаимозависимы, вы можете свободно перейти к описанию любого конкретного шаблона и проработать его отдельно от других шаблонов, а в свободное время изучить связанные с ним проектные шаблоны.

В этой главе будут рассмотрены следующие вопросы.

- *Краткий обзор архитектуры.* Обзор уровней, из которых обычно состоит корпоративное приложение.
- *Шаблон Registry.* Управление данными приложения.
- *Уровень представления данных.* Средства для управления, а также для ответа на запросы и представления данных для пользователя.
- *Уровень логики приложения.* Обращение к настоящей цели проектируемой системы для решения коммерческих задач.

Краткий обзор архитектуры

Поскольку в этой главе необходимо изложить довольно объемный материал, начнем с обзора рассматриваемых здесь проектных шаблонов, а затем продолжим кратким введением в построение многоуровневых приложений.

Шаблоны

В этой главе мы рассмотрим перечисленные ниже шаблоны. Можете читать главу от начала до конца или же описание только тех шаблонов, которые вам нужны или интересны в данный момент.

- *Шаблон Registry.* Служит для того, чтобы сделать данные доступными для всех классов в текущем процессе. Если аккуратно пользоваться сериализацией, то данный шаблон можно также применять для сохранения сведений на протяжении всего сеанса работы или даже целого приложения.
- *Шаблон Front Controller.* Предназначен для крупных систем, в которых требуется максимально возможная степень гибкости при управлении различными представлениями и командами.
- *Шаблон Application Controller.* Позволяет создавать классы для управления логикой представления данных и выбором команд.
- *Шаблон Template View.* Позволяет создавать страницы для управления только отображением данных и пользовательским интерфейсом, а также динамически встраивать информацию в статические страницы, используя как можно меньше кода.
- *Шаблон Page Controller.* Это более легковесный, но менее гибкий, чем Front Controller, шаблон, решающий те же самые задачи. Он применяется для управления запросами и логикой представления данных, если требуется быстро получить результаты, хотя сложность системы от этого существенно не увеличится.
- *Шаблон Transaction Script.* Если требуется быстро решать задачи, прибегая в минимальной степени к предварительному планированию, для создания логики приложения следует использовать процедурный библиотечный код. Этот шаблон плохо поддается масштабированию.

- *Шаблон Domain Model.* Является полной противоположностью шаблона Transaction Script и служит для создания объектных моделей бизнес-процессов и их участников.

На заметку Шаблон Command здесь отдельно не описывается, поскольку это уже было сделано в главе 11, "Выполнение задач и представление результатов". Но он снова применяется в этой главе вместе с шаблонами Front Controller и Application Controller.

Приложения и уровни

Многие (а по существу, большинство) шаблоны, описываемые в этой главе, предназначены для того, чтобы способствовать независимой работе нескольких различных уровней в приложении. Уровни корпоративной системы, подобно классам, представляют специализацию обязанностей, но только в более крупных масштабах. Типичное разделение системы на уровни показано на рис. 12.1.

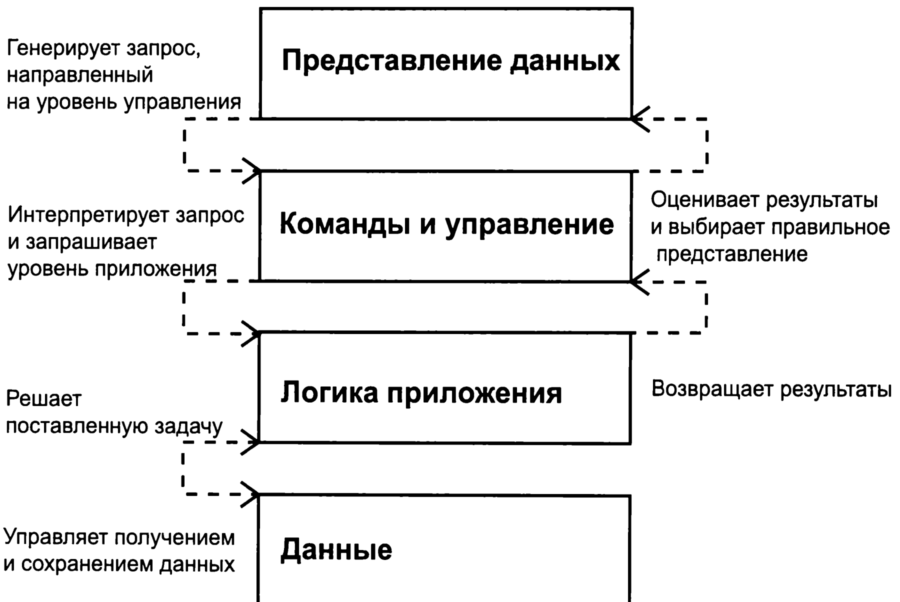


Рис. 12.1. Уровни типичной корпоративной системы

Структура, приведенная на рис. 12.1, не является чем-то неизменным: некоторые уровни в ней можно объединить, а для сообщения между ними могут применяться другие стратегии, хотя это зависит от сложности проектируемой системы. Тем не менее на рис. 12.1 наглядно показана модель, в которой делается акцент на гибкость и повторное использование кода, и многие корпоративные приложения в значительной степени следуют этой модели.

- *Уровень представления данных.* Содержит интерфейс, который реально видят и с которым взаимодействуют пользователи системы. Он отвечает за представление результатов запроса пользователя и обеспечение механизма, посредством которого можно сделать следующий запрос системы.
- *Уровень команд и управления.* Обрабатывает запрос от пользователя. На основе этого анализа он делегирует уровню логики приложения полномочия по любой обработке, необходимой для выполнения запроса. Затем он решает, какой шаблон лучше всего подходит для представления результатов пользователю. На практике данный уровень часто объединяется с уровнем представления данных в единый *уровень представления.* Но даже в этом случае роль отображения должна быть строго отделена от ролей обработки запросов и вызова логики приложения.
- *Уровень логики приложения.* Отвечает за обработку запроса. Выполняет все необходимые вычисления и упорядочивает полученные в итоге данные.
- *Уровень данных.* Отделяет остальную систему от механизма хранения и получения необходимой информации. Уровень данных используется в одних системах на уровне управления и команд для получения объектов приложения, с которыми требуется работать. А в других системах уровень данных скрывается настолько, насколько это возможно.

Так какой же смысл разделять систему подобным образом? Как и в отношении многого другого в этой книге, ответ кроется в ослаблении связей. Поддерживая логику приложения независимой от уровня представления данных, вы делаете возможным добавление новых интерфейсов в систему вообще без изменений или же с небольшими коррективами ее исходного кода.

Представьте себе систему управления списками событий (до конца этой главы мы разберем данный пример достаточно подробно). Конечному

пользователю требуется привлекательный HTML-интерфейс, что вполне естественно, тогда как администраторам, поддерживающим систему, может понадобиться интерфейс командной строки для встраивания в автоматизированные системы. В то же время для работы с мобильными телефонами и другими портативными устройствами необходимо разработать разные версии системы, а возможно, и применить для этого интерфейс SOAP или RESTful API.

Если вы первоначально объедините логику, лежащую в основе системы, с уровнем представления в формате HTML (что до сих пор является распространенной методикой, несмотря на многочисленные критические замечания по этому поводу), эти требования могут вынудить вас сразу же приступить к переписыванию системы. А если вы спроектируете многоуровневую систему, то новые методики представления данных можно будет внедрить, не пересматривая уровни логики приложения и данных.

Но в то же время методики сохранения данных подвержены изменениям. И в этом случае должна быть возможность выбирать методики сохранения данных с минимальным воздействием на другие уровни в системе.

Тестирование — это еще одна хорошая причина для проектирования систем с раздельными уровнями. Общеизвестно, что веб-приложения с трудом поддаются тестированию. В системе, которая в недостаточной степени разделена на уровни, автоматические тесты должны, с одной стороны, согласовывать HTML-интерфейс, а с другой — рискованно запускать произвольные запросы базы данных, даже если они преследуют совсем другую цель. И хотя всякое тестирование лучше, чем полное его отсутствие, такие тесты неизбежно вызывают беспорядок. В многоуровневой системе классы, которые обращаются к другим уровням, часто создаются таким образом, чтобы расширять абстрактный суперкласс или реализовывать определенный интерфейс. Этот супертип тогда может поддерживать полиморфизм. В контексте тестирования весь уровень может быть заменен рядом фиктивных объектов, которые нередко называются "заглушками" или "имитирующими" объектами. Подобным образом можно протестировать логику работы приложения, например, с помощью фиктивного уровня данных. Подробнее о тестировании речь пойдет в главе 18, "Тестирование средствами PHPUnit".

Уровни полезны даже в том случае, когда вы считаете, что тестирование никому не нужно и что у вашей системы всегда будет только один интерфейс. Создавая уровни с различными обязанностями, вы строите систему, составные части которой легче расширять и отлаживать. Сохраняя код, выполняющий одинаковые операции, в одном месте, а не пронизывая

систему, например, многочисленными обращениями к базе данных или стратегиями отображения информации, вы тем самым уменьшаете вероятность его дублирования. Ввести что-нибудь в такую систему относительно просто, потому что изменения будут происходить упорядоченно по вертикали, а не беспорядочно по горизонтали.

Новому функциональному средству в многоуровневой системе могут потребоваться новый компонент интерфейса, дополнительная обработка запросов, дополнительная логика приложения и улучшение механизма хранения данных. Такое изменение называется *вертикальным*. А в одноуровневой системе можно ввести новое функциональное средство, но затем вспоминать, на скольких (пяти, а может быть, и на шести) страницах применяется измененная таблица базы данных. В такой системе могут существовать десятки мест, где вполне вероятен вызов нового интерфейса. Поэтому придется просмотреть всю систему, добавляя в нужных местах соответствующий код. Такое изменение называется *горизонтальным*.

Разумеется, на практике вряд ли удастся полностью избежать подобного рода горизонтальных зависимостей, особенно если речь идет об элементах навигации в интерфейсе. Но многоуровневая система поможет свести к минимуму потребность в горизонтальных изменениях.

На заметку Несмотря на то что многие из описываемых здесь шаблонов существуют уже давно (ведь шаблоны являются отражением испытанных методов), их названия и области применения взяты из упоминавшегося ранее основного труда Мартина Фаулера по шаблонам корпоративных приложений *Patterns of Enterprise Application Architecture*[1] или из авторитетного труда Дипака Алура *Core J2EE Patterns*[2]. При расхождениях в этих двух источниках ради согласованности здесь употребляются условные обозначения имен шаблонов, принятые у Фаулера. Дело в том, что работа Фаулера в меньшей степени сфокусирована на одной технологии и поэтому имеет более широкое применение. Алур и другие в своей книге склонны ориентироваться на технологию Enterprise Java Beans, а это означает, что многие шаблоны оптимизированы для распределенных архитектур. Очевидно, что это вопрос ниши в мире PHP.

Если вы найдете эту главу полезной для себя, то я бы порекомендовал вам обе упомянутые книги в качестве следующего этапа обучения. Даже если вы не знаете Java, то как специалист, занимающийся объектно-ориентированным программированием на PHP, вы достаточно легко разберетесь в приведенных в них примерах.

---

[1] Фаулер, М. *Шаблоны корпоративных приложений* : Пер. с англ. — ИД "Вильямс", 2009.

[2] Алур, Д., Крупи Дж., Малкс, Д. *Образцы J2EE. Лучшие решения и стратегии проектирования* : Пер. с англ. — Изд-во "Лори", 2013.

Все примеры из этой главы связаны с вымышленной системой управления текстовыми списками с причудливым названием “Woo”, которое означает “What's On Outside” (Что идет во внешнем мире). К участникам системы относятся различные культурные заведения (театры, клубы, кинотеатры), места проведения культурных мероприятий (киноэкран или театральные подмостки) и названия мероприятий (например, фильмы *Долгая страстная пятница* и *Как важно быть серьезным*). К описываемым далее операциям относятся создание культурного заведения, добавление в него места проведения мероприятий и вывод списка всех заведений в системе.

Не следует, однако, забывать, что назначение этой главы — продемонстрировать основные проектные шаблоны корпоративных приложений, а не спроектировать рабочую систему. По своему характеру проектные шаблоны взаимозависимы, и поэтому исходный код в большинстве представленных здесь примеров частично совпадает. Этот код предназначен в основном для демонстрации корпоративных шаблонов и по большей части не удовлетворяет всем критериям рабочей корпоративной системы. В частности, проверка ошибок опущена ради большей ясности кода примеров. Поэтому рассматривайте эти примеры как средства для демонстрации реализуемых шаблонов, а не как стандартные блоки для построения каркаса или приложения.

Нарушение правил с самого начала

Большинство шаблонов, представленных в этой книге, вписываются естественным образом в отдельные уровни архитектуры корпоративного приложения. Но некоторые шаблоны являются настолько основополагающими, что выходят за пределы этой архитектуры. Характерным тому примером служит шаблон Registry. По существу, он предоставляет эффективный способ выхода за пределы ограничений, накладываемых разделением на уровни. Это исключение, которое только подтверждает общее правило.

Шаблон Registry

Этот шаблон предназначен для того, чтобы предоставлять доступ к объектам во всей системе. Это своеобразный символ веры в то, что глобальные переменные — зло. Но, как и остальные грехи, глобальные данные чертовски привлекательны. Поэтому архитекторы объектно-ориентированных

систем почувствовали потребность изобрести глобальные данные заново, но под другим именем. В главе 9, "Генерация объектов", уже рассматривался шаблон Singleton, хотя объекты-одиночки, создаваемые в соответствии с шаблоном Singleton, в действительности не страдают всеми болезнями, которыми заражены глобальные переменные. В частности, синглтон нельзя случайно затереть. Следовательно, синглтоны — это здоровые глобальные переменные. Но эти объекты все равно вызывают подозрение, потому что побуждают привязывать классы к системе, внося тем самым излишне тесную связь. Тем не менее синглтоны иногда настолько полезны, что многие программисты (включая меня) не могут от них отказаться.

Проблема

Как пояснялось ранее, многие корпоративные системы разделены на уровни, причем каждый уровень сообщается с соседними уровнями только по строго определенным каналам. Такое разделение на уровни делает приложение гибким. В частности, каждый уровень можно заменить или переделать, оказав минимальное воздействие на остальную систему. Но что произойдет, если полученная на каком-то уровне информация впоследствии понадобится на другом несмежном уровне?

Допустим, что сведения о конфигурации приложения загружаются в классе `ApplicationHelper`, как показано ниже:

```
// Листинг 12.1
class ApplicationHelper
{
    public function getOptions(): array
    {
        $optionfile = __DIR__ . "/data/woo_options.xml";

        if (! file_exists($optionfile))
        {
            throw new AppException("Не могу найти файл настроек");
        }

        $options = \simplexml_load_file($optionfile);
        $dsn = (string)$options->dsn;
        // Что теперь с этим делать?
        // ...
    }
}
```

Получить эту информацию очень просто, но как доставить ее на тот уровень данных, где она будет использована в дальнейшем? И как насчет всех остальных сведений о конфигурации, которые должны быть доступны во всей системе? В качестве ответа на эти вопросы можно передавать информацию в системе от одного объекта к другому: от объекта контроллера, ответственного за обработку запросов, через объекты на уровне логики приложения к объекту, отвечающему за взаимодействие с базой данных.

И это вполне осуществимо. По существу, передать можно сам объект типа `ApplicationHelper` или же более специализированный объект типа `Context`. Но в любом случае контекстно-зависимая информация из объекта типа `Context` передается через уровни системы тому объекту или нескольким объектам, которым она требуется.

Но здесь неизбежен компромисс: чтобы сделать это, придется изменить интерфейс всех объектов и включить в него объект типа `Context` независимо от того, нужно ли им его использовать. Очевидно, что это в какой-то степени подрывает слабую связанность объектов в приложении. Альтернативный вариант предоставляет упоминавшийся ранее шаблон `Registry`, но его применение имеет свои последствия.

Реестр — это просто класс, который предоставляет доступ к данным (как правило, но не исключительно, к объектам) через статические методы (или методы экземпляра, вызываемые для одноэлементных объектов). Поэтому у каждого объекта в системе имеется доступ к этим объектам.

Термин *реестр* (Registry) взят из упоминавшейся ранее книги Мартина Фаулера *Patterns of Enterprise Application Architecture*, но, как это часто бывает с наименованиями многих проектных шаблонов, они употребляются в разных контекстах. Так, в своей книге *The Pragmatic Programmer: from Journeyman to Master* Дэвид Хант (David Hunt) и Дэвид Томас (David Thomas) сравнили класс реестра с доской объявлений о происшествиях в полиции. Агенты, работающие в первую смену, оставляют на доске информацию о фактах и заметки, которые затем забирают агенты, работающие во вторую смену. Мне также приходилось видеть использование шаблона Registry под названиями “Whiteboard” (Белая доска) и “Blackboard” (Черная доска).

Реализация

На рис. 12.2 показан объект типа `Registry`, предназначенный для сохранения и возврата объектов типа `Request` по запросу.

Ниже приведено определение классов `Registry`:

```
// Листинг 12.2
class Registry
{
    private static ? Registry $instance = null;
    private ? Request $request = null;
    private function __construct()
    {
    }
    public static function instance(): self
    {
        if (is_null(self::$instance))
        {
            self::$instance = new self();
        }
        return self::$instance;
    }
    public function getRequest(): Request
    {
        if (is_null($this->request))
        {
            $this->request = new Request();
        }
        return $this->request;
    }
}

// Листинг 12.3
class Request
{
}
```

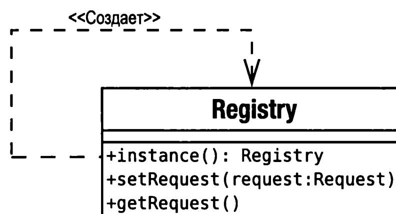


Рис. 12.2. *Простой реестр*

Таким образом, доступ к одному и тому же объекту типа `Request` можно получить в любой части системы, как показано ниже:

```
// Листинг 12.4

$reg = Registry::instance();
print_r($reg->getRequest());
```

Как видите, `Registry` — это обычный синглтон (напомним, что такие классы создаются в соответствии с шаблоном Singleton; см. главу 9, “Генерация объектов”). В приведенном выше фрагменте кода с помощью метода `instance()` сначала получается и возвращается единственный экземпляр класса `Registry`, который затем используется для получения объекта типа `Request`.

Отбросив всякие сомнения, воспользуемся системой сохранения параметров на основе ключей следующим образом:

```
// Листинг 12.5
class Registry
{
    private static ? Registry $instance = null;
    private array $values = [];
    private function __construct()
    {
    }
    public static function instance(): self
    {
        if (is_null(self::$instance))
        {
            self::$instance = new self();
        }
        return self::$instance;
    }
    public function get(string $key): mixed
    {
        if (isset($this->values[$key]))
        {
            return $this->values[$key];
        }
        return null;
    }
    public function set(string $key, mixed $value): void
    {
        $this->values[$key] = $value;
    }
}
```

Преимущество такого подхода заключается в том, что отпадает необходимость создавать методы для каждого объекта, который требуется сохранять и обслуживать, а недостаток — в том, что глобальные переменные снова вводятся, пусть и, так сказать, через "черный ход". Использование произвольных символьных строк в качестве ключей для сохраняемых объектов означает следующее: ничто не помешает изменить пару "ключ–значение" при добавлении объекта в одной из частей системы. Я обнаружил, что на стадии разработки удобно воспользоваться структурой отображения ключей на значения; когда станет ясно, какие именно данные требуется сохранить и извлечь, можно будет перейти к явно именованным методам.

На заметку Шаблон Registry является не единственным средством организации служб, которые требуются в проектируемой системе. Аналогичная стратегия с использованием шаблона Dependency Injection уже рассматривалась в главе 9, "Генерация объектов". Она применяется в таких распространенных каркасах, как Symfony.

Объекты реестра можно использовать также в качестве фабрик для распространенных объектов системы. Вместо сохранения предоставленного объекта класс-реестр создает экземпляр, а затем сохраняет в кеш-памяти ссылку на него. Кроме того, можно незаметно осуществить некоторую настройку, например извлечь данные из файла конфигурации или объединить ряд объектов:

```
// Листинг 12.6
// class Registry

private ? TreeBuilder $treeBuilder = null;
private ? Conf $conf = null;
// ...
public function treeBuilder(): TreeBuilder
{
    if (is_null($this->treeBuilder))
    {
        $this->treeBuilder = new TreeBuilder(
            $this->conf()->get('treedir'));
    }
    return $this->treeBuilder;
}
public function conf(): Conf
{
    if (is_null($this->conf))
    {
```

```
        $this->conf = new Conf();
    }
    return $this->conf;
}
```

Классы TreeBuilder и Conf — вымышленные и включены в данный пример ради демонстрации сути рассматриваемого вопроса. Клиентский класс, которому требуется объект типа TreeBuilder, может просто вызвать статический метод Registry::treeBuilder(), не особенно беспокоясь о сложностях инициализации. К таким сложностям могут относиться данные уровня приложения, например из вымышленного объекта типа Conf, и большинство классов в системе должны быть изолированы от них.

Объекты реестра можно использовать также для тестирования. В частности, статический метод instance() можно вызвать для обслуживания объекта класса, производного от класса Registry и заполненного пустыми объектами. Ниже показано, как внести коррективы в метод instance(), чтобы достичь этой цели:

```
// Листинг 12.7
// class Registry
private static $testmode = false;
// ...
public static function testMode(bool $mode = true): void
{
    self::$instance = null;
    self::$testmode = $mode;
}
public static function instance(): self
{
    if (is_null(self::$instance))
    {
        if (self::$testmode)
        {
            self::$instance = new MockRegistry();
        }
        else
        {
            self::$instance = new self();
        }
    }
    return self::$instance;
}
```

Для того чтобы испытать спроектированную систему, можно воспользоваться режимом тестирования, чтобы перейти к сымитированному

реестру. Он может предоставить заглушки (объекты, имитирующие реальную среду для целей тестирования) или имитирующие объекты (аналогичные объекты, предназначенные для анализа сделанных для них вызовов и оценивания их правильности):

```
// Листинг 12.8

Registry::testMode();
$mockreg = Registry::instance();
```

Подробнее о заглушках и имитирующих объектах речь пойдет в главе 18, "Тестирование средствами PHPUnit".

Реестр, область видимости и язык PHP

Термин *область видимости* (scope) часто употребляется для описания видимости объекта или значения в контексте структур кода. Период существования переменной можно также измерять во времени. Имеются три уровня области видимости, которые можно рассматривать в этом смысле. Стандартом служит период, охватываемый HTTP-*запросом*. В языке PHP предусмотрена также встроенная поддержка сеансовых переменных. В конце запроса они сериализуются и сохраняются в файле или базе данных, а затем снова восстанавливаются при поступлении следующего запроса. Идентификатор сеанса связи, который сохраняется в cookie-файле или передается в строке запроса, используется для отслеживания сведений о владельце данного сеанса. Поэтому можно считать, что у некоторых переменных имеется своя сеансовая область видимости. Этим преимуществом можно воспользоваться, сохраняя некоторые объекты в промежутках между запросами и тем самым избавляя себя от необходимости обращаться к базе данных. Безусловно, это следует делать очень аккуратно, чтобы не получить в итоге несколько версий одного и того же объекта. Поэтому нужно также подумать о стратегии блокировки, если в результате проверки объекта в сеансе связи окажется, что аналогичный объект уже существует в базе данных.

В других языках, особенно в Java и Perl (который реализован в виде модуля ModPerl веб-сервера Apache), существует понятие области видимости приложения. Переменные, занимающие это пространство, доступны всем экземплярам объектов приложения. Но это совсем не свойственно языку PHP, хотя в крупных приложениях очень удобно иметь доступ к пространству всего приложения, чтобы иметь возможность обращаться к переменным конфигурации.

В предыдущих изданиях этой книги демонстрировались примеры применения классов реестров в области видимости сеанса связи и приложения. Но с тех пор, как около десяти лет назад я впервые написал пример подобного кода, у меня так и не появилось веских оснований пользоваться ничем другим, кроме реестра в области видимости запроса. За такой подход на уровне отдельных запросов приходится расплачиваться инициализацией, но в качестве выхода из этого положения, как правило, применяются стратегии кеширования.

Следствия

Объекты реестра делают свои данные глобально доступными. Это означает, что любой класс, являющийся клиентом реестра, будет зависим от него, причем эта зависимость в его интерфейсе не объявляется. И это может вызвать серьезные трудности в проектируемой системе, стоит только положиться во многом на данные из объектов реестра. Объектами реестра лучше всего пользоваться аккуратно для хранения вполне определенных элементов данных.

Уровень представления данных

Когда делается запрос к спроектированной вами системе, она должна каким-то образом интерпретировать указанные в нем данные, а затем выполнить всю необходимую обработку и возвратить ответ. В простых сценариях весь этот процесс обычно полностью происходит в самом представлении, и только сложные вычислительные процедуры и повторяющиеся фрагменты кода размещаются в библиотеках.

На заметку *Представление* — отдельный элемент уровня отображения данных. Это может быть PHP-страница (или совокупность составных элементов представления), предназначенная главным образом для того, чтобы отображать данные и обеспечивать механизм, посредством которого пользователь может генерировать новые запросы. Это может быть также один из шаблонов, применяемых в таких системах, как Twig.

По мере увеличения масштабов приложения стандартная стратегия становится менее подходящей для обработки запросов, вызова логики приложения, поскольку логика диспетчеризации представления дублируется с одной страницы на другую.

В этом разделе мы рассмотрим стратегии управления этими тремя ключевыми обязанностями на уровне представления. А поскольку границы между уровнем представления и уровнем команд и управления, как правило, весьма размыты, то их целесообразно рассматривать вместе под общим названием "уровень представления".

Шаблон Front Controller

Этот шаблон диаметрально противоположен традиционному приложению на языке программирования PHP с его несколькими точками входа. Шаблон Front Controller (Фронтальный контроллер) предоставляет центральную точку доступа для обработки всех входящих запросов и в конечном итоге поручает уровню представления вывод полученных результатов для просмотра пользователем. Это основной шаблон из семейства шаблонов корпоративных приложений Java. Он очень подробно описан в упоминавшейся ранее книге *Core J2EE Patterns: Best Practices and Design Strategies*, которая остается одним из самых авторитетных источников по шаблонам корпоративных приложений. Но нельзя сказать, что все члены сообщества программирующих на PHP являются приверженцами этого шаблона, отчасти из-за издержек на инициализацию, которые иногда неизбежны.

В большинстве разрабатываемых мною систем, как правило, используется шаблон Front Controller. И хотя я могу поначалу использовать его не полностью, я знаю, какие шаги необходимо предпринять, чтобы развить проект в реализацию шаблона Front Controller, если мне понадобится та степень гибкости, которую он предоставляет.

Проблема

Когда запросы обрабатываются в нескольких точках системы, очень трудно избежать дублирования кода. Вполне возможно, что потребуется аутентифицировать пользователя, перевести элементы интерфейса на другие языки или просто получить доступ к общим данным. Так, если для запроса требуется выполнить общие для разных представлений действия, это неизбежно приведет к копированию и вставке кода. Это усложнит внесение изменений, поскольку даже самое простое исправление придется повторить в нескольких местах приложения. И по этой причине может быть нарушено соответствие одних частей кода другим. Безусловно, общие опе-

рации можно централизовать, выделив их в библиотечный код, но все равно останутся вызовы библиотечных функций или методов, разбросанные по всему приложению.

Трудность в организации перехода от одного представления к другому — это еще одно осложнение, которое может возникнуть в системе, в которой управление распределено по представлениям. В сложной системе передача запроса на обработку в одном представлении может привести к произвольному количеству страниц результатов в соответствии с входными данными и к успешному выполнению необходимых операций на уровне логики приложения. Логика перехода от одного представления к другому может стать запутанной, особенно если одно и то же представление используется в разных потоках.

Реализация

По существу, шаблон Front Controller определяет центральную точку входа для обработки каждого запроса. Он организует обработку запроса, используя его для того, чтобы выбрать операцию для последующего выполнения. Операции обычно определяются в специальных командных объектах, организованных в соответствии с шаблоном Command.

На рис. 12.3 приведен общий вид реализации шаблона `Front Controller`.

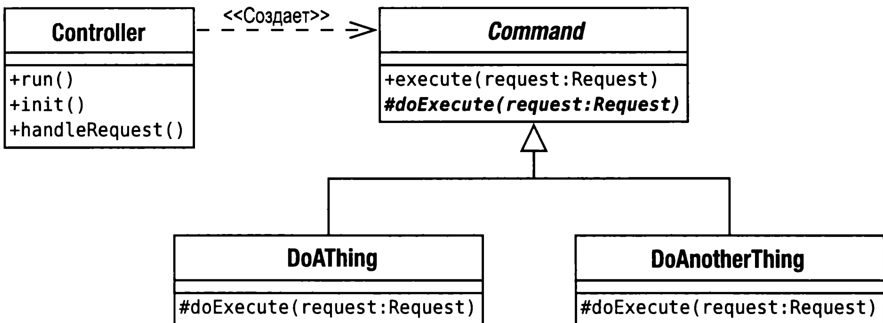


Рис. 12.3. *Класс `Controller` и иерархия команд*

На самом деле, чтобы как-то сгладить процесс разработки, скорее всего, придется воспользоваться несколькими вспомогательными классами. Начнем, однако, с основных участников данного процесса. Ниже приведено определение простого класса `Controller`:

```
// Листинг 12.9
class Controller
{
    private Registry $reg;
    private function __construct()
    {
        $this->reg = Registry::instance();
    }
    public static function run(): void
    {
        $instance = new self();
        $instance->init();
        $instance->handleRequest();
    }
    private function init(): void
    {
        $this->reg->getApplicationHelper()->init();
    }
    private function handleRequest(): void
    {
        $request = $this->reg->getRequest();
        $resolver = new CommandResolver();
        $cmd = $resolver->getCommand($request);
        $cmd->execute($request);
    }
}
```

Как видите, класс `Controller` очень прост и лишен всякой обработки ошибок (но в нем больше ничего пока что и не требуется). Контроллер находится на вершине системы, делегируя полномочия другим классам. Именно эти другие классы выполняют бóльшую часть работы. Статический метод `run()` добавлен ради удобства, чтобы вызывать в нем методы `init()` и `handleRequest()`. А поскольку конструктор данного класса объявлен закрытым, единственная возможность начать выполнение проектируемой системы в клиентском коде состоит в том, чтобы вызвать метод `run()`. Обычно это делается в файле `index.php`, который содержит лишь несколько строк кода, как показано ниже:

```
// Листинг 12.10
require_once(__DIR__ . "/../../../vendor/autoload.php");

use \popp\ch12\batch05\Controller;
Controller::run();
```

Обратите внимание на неприглядный вид инструкции require. Она нужна для того, чтобы в остальной части системы можно было не беспокоиться о включении необходимых файлов. Сценарий из исходного файла autoload.php автоматически формируется средствами Composer. Он управляет логикой загрузки файлов классов по мере надобности. Если это ни о чем вам не говорит, не отчаивайтесь — мы еще вернемся к подробному обсуждению механизма автозагрузки в главе 15, "Стандарты PHP".

На самом деле различия между методами init() и handleRequest() в языке PHP едва уловимы. В некоторых языках метод init() выполняется только при запуске приложения, а метод handleRequest() или его аналог — при получении каждого запроса пользователя. В нашем же классе мы придерживаемся такого же различия между инициализацией и обработкой запросов, хотя метод init() вызывается для каждого запроса.

В методе init() происходит обращение к классу ApplicationHelper через класс Registry, ссылка на который находится в свойстве $reg класса Controller. В классе ApplicationHelper хранятся данные конфигурации для всего приложения. Вызов Controller::init() приводит, в свою очередь, к вызову аналогичного метода init() из класса ApplicationHelper, где инициализируются данные, применяемые в приложении, как будет показано далее. А в методе handleRequest() используется класс CommandResolver с целью получить объект типа Command для выполнения соответствующей команды путем вызова метода Command::execute().

Класс ApplicationHelper

Этот класс не так важен для реализации шаблона Front Controller. Но в большинстве его реализаций используются данные конфигурации, поэтому для них необходимо выработать соответствующую стратегию. Ниже приведен пример простого определения класса ApplicationHelper:

```
// Листинг 12.11
class ApplicationHelper
{
    private string $config = __DIR__ . "/data/woo_options.ini";
    private Registry $reg;

    public function __construct()
    {
        $this->reg = Registry::instance();
    }
```

```
    public function init(): void
    {
        $this->setupOptions();

        if (defined('STDIN'))
        {
            $request = new CliRequest();
        }
        else
        {
            $request = new HttpRequest();
        }

        $this->reg->setRequest($request);
    }
    private function setupOptions(): void
    {
        if (! file_exists($this->config))
        {
            throw new AppException("Файл не найден");
        }

        $options = parse_ini_file($this->config, true);
        $this->reg->setConf(new Conf($options['config']));
        $this->reg->setCommands(new Conf($options['commands']));
    }
}
```

В данном классе просто выполняется чтение содержимого файла конфигурации, и в реестр вводятся различные объекты, благодаря чему они становятся доступными остальной системе. В методе `init()` вызывается закрытый метод `setupOptions()`, в котором читается содержимое файла с расширением `.ini` и объекту типа `Registry` передаются два массива, каждый из которых заключен в оболочку класса `Conf`. В классе `Conf` определены лишь методы `get()` и `set()`, хотя в более развитых классах могут быть организованы поиск и синтаксический анализ файлов, а также обнаружение данных. Один из массивов типа `Conf` предназначен для хранения общих значений конфигурации и передается вызываемому методу `Registry::setConf()`. Еще один массив служит для преобразования путей URL в классы типа `Command` и передается при вызове метода `Registry::setCommands()`.

В методе `init()` предпринимается также попытка выяснить, выполняется ли приложение в контексте веба или запущено из командной строки (для этого проверяется, определена ли константа `STDIN`). В зависимости от

результата данной проверки в этом методе создается и передается объекту типа `Registry` соответствующий объект отдельного подкласса, производного от класса `Request`.

Поскольку в классах типа `Registry` выполняется совсем немного действий (в них лишь сохраняются и возвращаются объекты), их исходный код вряд ли стоит приводить в листингах. Но ради полноты примера ниже представлены дополнительные методы из класса `Registry`, применяемые или подразумеваемые в классе `ApplicationHelper`:

```
// Листинг 12.12

// Должен быть инициализирован каким-нибудь более
// интеллектуальным компонентом

public function setRequest(Request $request): void
{
    $this->request = $request;
}
public function getRequest(): Request
{
    if (is_null($this->request))
    {
        throw new \Exception("Request не установлен");
    }

    return $this->request;
}
public function getApplicationHelper(): ApplicationHelper
{
    if (is_null($this->applicationHelper))
    {
        $this->applicationHelper = new ApplicationHelper();
    }

    return $this->applicationHelper;
}
public function setConf(Conf $conf): void
{
    $this->conf = $conf;
}
public function getConf(): Conf
{
    if (is_null($this->conf))
    {
        $this->conf = new Conf();
    }
```

```
    return $this->conf;
}
public function setCommands(Conf $commands): void
{
    $this->commands = $commands;
}
public function getCommands(): Conf
{
    if (is_null($this->commands))
    {
        $this->commands = new Conf();
    }

    return $this->commands;
}
```

Ниже приведен пример простого файла конфигурации:

```
[config]
dsn=sqlite:/var/popp/src/ch12/batch05/data/woo.db

[commands]
/=\popp\ch12\batch05\DefaultCommand
```

Класс CommandResolver

Контроллеру требуется какой-нибудь способ, позволяющий решить, как интерпретировать HTTP-запрос, чтобы в результате можно было вызвать нужный код для обработки этого запроса. Такую логику можно легко включить в сам класс Controller, но для этой цели лучше воспользоваться специальным классом. При необходимости это позволит без особого труда реорганизовать код для целей полиморфизма.

Контроллер, создаваемый в соответствии с шаблоном Front Controller, обращается к логике приложения, выполняя команду из объекта типа Command (подробнее о шаблоне Command — в главе 11, "Выполнение задач и представление результатов"). Этот объект выбирается в соответствии со структурой запрошенного URL или значением параметра, указанного в запросе типа GET (последний вариант встречается реже). Но в любом случае получается отличительный признак или шаблон, с помощью которого можно выбрать команду. Для выбора команды с помощью URL имеются разные способы. С одной стороны, можно сопоставить отличительный признак с конкретным классом типа Command с помощью файла конфигурации или структуры данных (такая стратегия называется *логической*). А с другой стороны, его

можно непосредственно отобразить на файл класса, расположенный в файловой системе (такая стратегия называется *физической*).

В предыдущей главе демонстрировался пример фабрики команд, в котором применялась физическая стратегия. Выберем на этот раз логическую стратегию, преобразовав фрагменты URL в классы, создаваемые в соответствии с шаблоном Command:

```
// Листинг 12.13
class CommandResolver
{
    private static ? \ReflectionClass $refcmd = null;
    private static string $defaultcmd = DefaultCommand::class;
    public function __construct()
    {
        // Этот объект можно сделать конфигурируемым
        self::$refcmd = new \ReflectionClass(Command::class);
    }
    public function getCommand(Request $request): Command
    {
        $reg = Registry::instance();
        $commands = $reg->getCommands();
        $path = $request->getPath();
        $class = $commands->get($path);
        if (is_null($class))
        {
            $request->addFeedback("Путь '$path' не годится");
            return new self::$defaultcmd();
        }
        if (! class_exists($class))
        {
            $request->addFeedback("Класс '$class' не найден");
            return new self::$defaultcmd();
        }
        $refclass = new \ReflectionClass($class);
        if (! $refclass->isSubClassOf(self::$refcmd))
        {
            $request->addFeedback(
                "Команда '$refclass' не является Command");
            return new self::$defaultcmd();
        }
        return $refclass->newInstance();
    }
}
```

В этом простом классе сначала из реестра извлекается объект типа Conf, а затем используется путь, полученный из URL в результате вызова метода

Request::getPath(), чтобы попытаться получить имя класса. Если имя класса найдено и этот класс существует и является экземпляром базового класса Command, то сначала создается его экземпляр, а затем он возвращается в вызываемую программу.

Если любое из упомянутых выше условий не удовлетворяется, то метод getCommand() корректно выходит из сложившейся ситуации, возвращая стандартный объект типа Command.

В более сложной реализации (например, при применении логики маршрутизации в Symfony) в подобных путях используются метасимволы.

В связи с изложенным выше у вас может возникнуть вопрос, почему в данном коде принимается на веру, что конструктору класса Command, который в нем создается, не нужно передавать никаких параметров:

```
return $refclass->newInstance();
```

Ответ следует искать в сигнатуре самого класса Command:

```
// Листинг 12.14
abstract class Command
{
    final public function __construct()
    {
    }
    public function execute(Request $request): void
    {
        $this->doExecute($request);
    }
    abstract protected function doExecute(Request $request): void;
}
```

Объявляя метод конструктора как final, мы делаем невозможным его переопределение в дочернем классе. Поэтому ни один из классов типа Command никогда не потребует аргументы для своего конструктора.

Создавая командные классы, старайтесь по возможности не допускать включения в них логики приложения. Как только они начнут выполнять то, что должно делать само приложение, они превратятся в некое подобие запутанного сценария транзакций, и тогда дублирование кода не заставит себя долго ждать. Команды можно сравнить с ретрансляционной станцией: они должны интерпретировать запрос, вызывать логику приложения для выполнения операций с какими-нибудь объектами, а затем передавать данные для отображения на уровне представления. И как только они начнут делать что-нибудь более сложное, вероятно, настанет время реор-

ганизовать код. Правда, реорганизовать код совсем нетрудно. Для этого достаточно найти место, где команда пытается сделать много лишнего, а решение для устранения этого недостатка, как правило, очевидно: переместить функциональные возможности команды вниз по иерархии во вспомогательный класс или же в класс предметной области.

Запрос

Запросы волшебным образом обрабатываются для нас средствами PHP и аккуратно укладываются в суперглобальные массивы. Вы, вероятно, заметили, что для представления запроса мы все еще пользуемся классом. В частности, объект типа `Request` передается сначала объекту типа `CommandResolver`, а затем — объекту типа `Command`.

Почему мы не позволяем этим классам напрямую обращаться к элементам суперглобальных массивов `$_REQUEST`, `$_POST` или `$_GET`? Мы могли бы, конечно, это сделать, но, собирая операции обработки запросов в одном месте, мы открываем для себя новые возможности.

Например, к входящим запросам можно применить фильтры или, как показано в примере далее, получить параметры не из HTTP-запроса, а из другого источника, что позволит запускать приложение из командной строки или из тестового сценария.

Объект типа `Request` может также служить удобным хранилищем данных, которые требуется передать на уровень представления. На самом деле во многих системах для этих целей предоставляется отдельный объект типа `Response`, но мы не будем усложнять дело до такой степени и приведем ниже пример простого класса `Request`:

```
// Листинг 12.15
abstract class Request
{
    protected array $properties = [];
    protected array $feedback = [];
    protected string $path = "/";
    public function __construct()
    {
        $this->init();
    }
    abstract public function init(): void;
    public function setPath(string $path): void
    {
        $this->path = $path;
    }
```

```
    public function getPath(): string
    {
        return $this->path;
    }
    public function getProperty(string $key): mixed
    {
        if (isset($this->properties[$key]))
        {
            return $this->properties[$key];
        }
        return null;
    }
    public function setProperty(string $key, mixed $val): void
    {
        $this->properties[$key] = $val;
    }
    public function addFeedback(string $msg): void
    {
        array_push($this->feedback, $msg);
    }
    public function getFeedback(): array
    {
        return $this->feedback;
    }
    public function getFeedbackString($separator = "\n"): string
    {
        return implode($separator, $this->feedback);
    }
    public function clearFeedback(): void
    {
        $this->feedback = [];
    }
}
```

Как видите, бо́льшую часть этого класса занимают механизмы установки и получения значений свойств. В частности, метод `init()` отвечает за наполнение закрытого массива `$properties` для обработки в дочерних классах. Следует особо подчеркнуть, что в данном примере реализации совершенно игнорируются методы запроса, чего не стоит делать в реальном проекте. В полноценной реализации следует организовать обработку массивов из запросов по методам `GET`, `POST` и `PUT`, а также предоставить единообразный механизм обработки запросов. Как только будет получен объект типа `Request`, вы должны обращаться к параметрам HTTP-запроса исключительно с использованием вызова метода `getProperty()`. Этому методу передается ключ в виде символьной строки, а он возвращает со-

ответствующее значение, хранящееся в массиве `$properties`. Кроме того, нужные данные можно ввести с помощью метода `setProperty()`.

В данном классе поддерживается также массив `$feedback`. Это простой канал связи, по которому классы контроллеров могут передавать сообщения пользователю. А в полноценной реализации, вероятнее всего, придется различать сообщения об ошибках от информационных сообщений.

Как упоминалось ранее, в классе `ApplicationHelper` создается экземпляр класса `HttpRequest` или `CliRequest`. Ниже приведено определение первого из них:

```
// Листинг 12.16
class HttpRequest extends Request
{
    public function init(): void
    {
        // Ради удобства здесь игнорируются различия
        // в методах запросов POST, GET и так далее, но
        // этого нельзя делать в реальном проекте!
        $this->properties = $_REQUEST;
        $this->path = $_SERVER['PATH_INFO'];
        $this->path = (empty($this->path)) ? "/" : $this->path;
    }
}
```

В классе `CliRequest` аргументы командной строки интерпретируются в виде пары "ключ–значение" и разделяются на отдельные свойства. Кроме того, в этом классе выполняется поиск аргумента с префиксом `path:`, а его значение присваивается свойству `$path` текущего объекта, как показано ниже:

```
// Листинг 12.17
class CliRequest extends Request
{
    public function init(): void
    {
        $args = $_SERVER['argv'];
        foreach ($args as $arg)
        {
            if (preg_match("/^path:(\S+)/", $arg, $matches))
            {
                $this->path = $matches[1];
            }
            else
            {
                if (strpos($arg, '='))
```

```
                {
                    list($key, $val) = explode("=", $arg);
                    $this->setProperty($key, $val);
                }
            }
        }
        $this->path = (empty($this->path)) ? "/" : $this->path;
    }
}
```

Команда

Вы уже знакомы с базовым классом Command, а шаблон Command подробно описывался в главе 11, "Выполнение задач и представление результатов", поэтому не будем повторяться. Но подытожив все, что об этом известно, рассмотрим простую реализацию объекта типа Command:

```
// Листинг 12.18
class DefaultCommand extends Command
{
    protected function doExecute(Request $request): void
    {
        $request->addFeedback("Welcome to Woo");
        include(__DIR__ . "/main.php");
    }
}
```

Такой объект типа Command автоматически выбирается классом CommandResolver, если не было получено явного запроса на конкретный объект типа Command. Как вы, вероятно, заметили, в абстрактном базовом классе Command реализован метод execute(), из которого вызывается метод doExecute(), реализованный в дочернем классе. Это позволяет вводить код для начальной установки и очистки во все команды, просто изменяя базовый класс.

Методу execute() передается объект типа Request, через который обеспечивается доступ к данным, введенным пользователем, а также к методу setFeedback(). В классе DefaultCommand данный метод служит для установки приветственного сообщения.

И наконец команда передает управление представлению, просто вызывая метод include(). Внедрение кода представления в классы типа Command — это самый простой механизм диспетчеризации, но для небольших систем он подходит идеально. О более гибкой стратегии речь пойдет далее, в разделе "Шаблон Application Controller".

В файле main.php содержится HTML-код разметки страницы и вызов метода getFeedback() объекта Request для отображения сообщений, предназначенных для пользователя (вскоре я расскажу о представлениях более подробно). Теперь у нас имеются все компоненты, необходимые для запуска системы. Ниже показано, как это будет выглядеть на экране:

```
<html>
<head>
<title>Woo! Это программа Woo!</title>
</head>
<body>

<table>
<tr>
<td>
Добро пожаловать в Woo!
</td>
</tr>
</table>

</body>
</html>
```

Как видите, сообщение, выведенное в теле стандартной команды, оказалось на экране пользователя. А теперь рассмотрим весь процесс, который привел к такому результату.

Общее представление

Подробное описание рассматриваемых здесь классов может дать неверное представление о простоте шаблона Front Controller. Поэтому на рис. 12.4 приведена диаграмма последовательности, иллюстрирующая жизненный цикл запроса.

Как видите, Front Controller делегирует инициализацию объекту типа ApplicationHelper, в котором используется кеширование с целью сократить время на обработку параметров. Затем класс Controller получает объект типа Command от объекта типа CommandResolver. И наконец вызывается метод Command::execute(), чтобы выполнить логику приложения.

В данной реализации шаблона Front Controller сам объект типа Command отвечает за делегирование полномочий на уровень представления. Усовершенствованный вариант реализации данного шаблона будет приведен в следующем разделе.

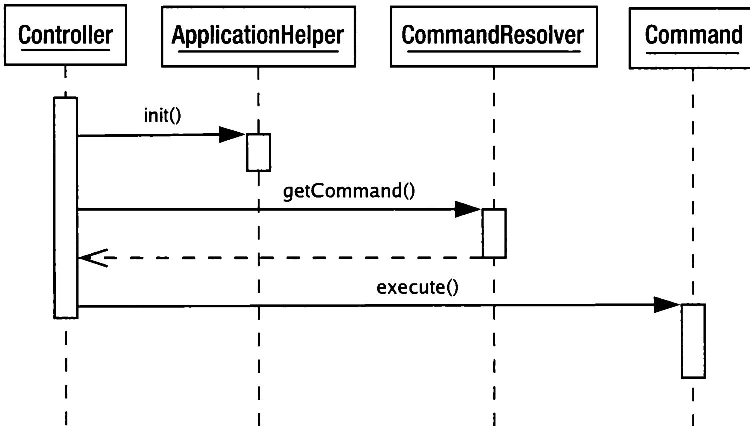


Рис. 12.4. Шаблон Front Controller в действии

Следствия

Шаблон Front Controller — не для слабонервных. Для работы с ним вам придется уделить немало внимания предварительной разработке, прежде чем вы реально почувствуете все его преимущества. И если проект настолько мал, что в конечном итоге каркас шаблона Front Controller может оказаться сложнее всей остальной системы, или если проект требуется очень быстро реализовать, это становится серьезным препятствием для применения данного шаблона.

Но, успешно применив шаблон Front Controller в одном проекте, вы обнаружите, что можете повторно использовать его и в других проектах с молниеносной быстротой. В частности, большинство его функций можно разместить в библиотечном коде, создав таким образом эффективный повторно используемый каркас.

Еще одним недостатком данного шаблона является обязательное требование, чтобы все сведения о конфигурации загружались по каждому запросу. В какой-то степени от этого страдают все шаблоны, но для шаблона Front Controller часто необходимы дополнительные сведения, например логические соответствия команд и представлений.

Указанный недостаток можно значительно уменьшить благодаря кешированию данных конфигурации. Самый эффективный способ сделать это — организовать ввод данных конфигурации системы в виде встроенных в PHP структур данных. Такой подход вполне пригоден для тех разра-

ботчиков, которые самостоятельно сопровождают систему. Но если вашей системой пользуются люди, не владеющие языком PHP, то для них придется создать файл конфигурации. Впрочем, вы можете автоматизировать упомянутый выше подход с помощью собственных средств PHP, написав программу, которая сначала читает содержимое файла конфигурации, а затем создает на его основе структуры данных PHP, которые хранятся в оперативной памяти. Как только структуры данных конфигурации будут созданы, они могут использоваться в вашей системе до тех пор, пока не будут внесены изменения в исходный файл конфигурации и не возникнет потребность обновить содержимое кеша.

Преимущество шаблона Front Controller заключается в том, что в нем централизована логика управления представлениями в системе. Это означает возможность осуществлять централизованный контроль над обработкой запросов и выбирать представления (но в любом случае в одном наборе классов). Это позволит сократить дублирование кода и уменьшить вероятность ошибок.

Шаблон Front Controller предоставляет также немало возможностей для расширения. Как только основная часть приложения будет спроектирована и готова к работе, ввести новые классы типа `Command` и представления не составит большого труда.

В описанном здесь примере сами команды управляют диспетчеризацией собственных представлений. Если шаблон Front Controller применяется вместе с объектом, помогающим выбрать представление, а возможно, и команду, то он предоставляет отличные возможности для управления навигацией, которую труднее эффективно поддерживать, если управление представлениями распределено по всей системе. Такой объект будет описан в следующем разделе.

Шаблон Application Controller

В небольших системах вполне допускается, чтобы команды вызывали собственные представления, хотя это и не идеальный вариант. Намного предпочтительнее отделить команды от уровня представления данных настолько, насколько это возможно.

Шаблон Application Controller (Контроллер приложения) берет на себя ответственность за соответствие запросов командам и команд — их представлениям. Такое разделение ответственности означает, что в приложении можно очень легко сменять совокупности представлений, не за-

трагивая кодовую базу. Это также дает владельцу системы возможность изменять ход выполнения приложения, не затрагивая, опять-таки, его внутреннее строение. Допуская логическую систему разрешения команд в соответствии с шаблоном Command, шаблон Application Controller упрощает также применение одной и той же команды в разных контекстах проектируемой системы.

Проблема

Напомним о характере проблемы в рассматриваемом здесь примере. Администратору требуется возможность вводить культурное заведение в систему и связывать с ним место проведения культурного мероприятия. Поэтому система может поддерживать команды типа `AddVenue` и `AddSpace`. Как следует из рассмотренных до сих пор примеров, эти команды выбираются путем прямого отображения пути (`/addVenue`) на класс (`AddVenue`).

Вообще говоря, успешный вызов команды типа `AddVenue` должен привести к первоначальному вызову команды типа `AddSpace`. Эта связь может быть жестко закодирована в самих классах, чтобы при успешном выполнении команды типа `AddVenue` она вызывала команду типа `AddSpace`. Затем в команду типа `AddSpace` может быть включено представление, содержащее форму для ввода места проведения культурного мероприятия в соответствующее заведение.

Обе команды могут быть связаны по меньшей мере с двумя различными представлениями: основным — для отображения формы ввода входных данных и вспомогательным — для вывода сообщения об ошибке или выражения благодарности на экран. По обсуждавшейся ранее логике эти представления могут быть включены в сами классы типа `Command` (а точнее, в проверки условий, в которых решается, какое именно представление следует воспроизводить в конкретной ситуации).

Такой уровень жесткого кодирования приемлем до тех пор, пока команды применяются одинаково. Но ситуация начнет ухудшаться, когда потребуется специальное представление для команды типа `AddVenue` и придется изменить логику, по которой одна команда приводит к другой (вероятно, в выполняющемся потоке понадобится вывести дополнительный экран после успешного ввода культурного заведения и перед началом ввода места проведения культурного мероприятия). Если каждая из команд применяется только один раз в одной взаимосвязи с другими командами и одним представлением, то взаимосвязи команд друг с другом и с их представле-

ниями можно жестко закодировать. В противном случае продолжайте читать, чтобы узнать, каким образом класс контроллера приложения может взять на себя управление такой логикой, освободив классы типа `Command`, чтобы они могли сосредоточиться на своих непосредственных обязанностях, т.е. обрабатывать входные данные, вызывать логику приложения и обрабатывать полученные результаты.

Реализация

Как всегда, ключом к данному шаблону служит интерфейс. Контроллер приложения — это класс (или ряд классов), который шаблон Front Controller может применять с целью получить команды на основе запроса пользователя и найти нужное представление, которое будет затем выведено после выполнения команды. Подобная взаимосвязь контроллеров наглядно показана на рис. 12.5.

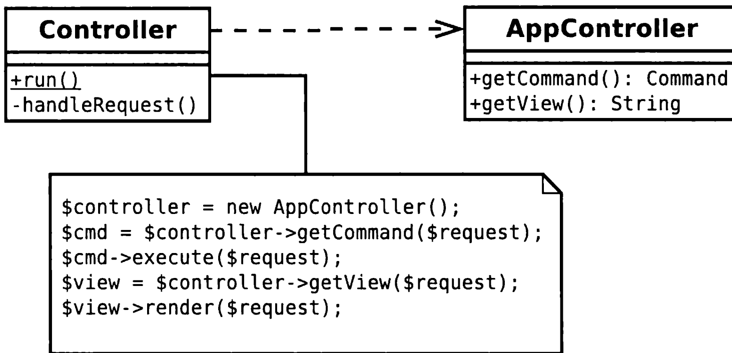


Рис. 12.5. *Шаблон Application Controller*

Назначение данного шаблона, как и всех остальных шаблонов, рассматриваемых в этой главе, — максимально упростить задачу клиентского кода. Этим и объясняется спартанский вид класса `Controller`, представляющего Front Controller. Но за интерфейсом необходимо развернуть реализацию. Применяемый здесь подход демонстрирует лишь одну из возможностей сделать это. Прорабатывая материал этого раздела, не забывайте, что суть данного шаблона состоит во взаимодействии участников (контроллера приложения, команд и представлений), а не в особенностях рассматриваемой здесь реализации. Итак, начнем с того кода, в котором применяется контроллер приложения.

Класс Controller

Ниже показано, каким образом класс Controller может взаимодействовать с классом AppController. Данный пример кода упрощен и не предполагает обработку ошибок:

```
// Листинг 12.19
// Controller
private function __construct()
{
    $this->reg = Registry::instance();
}
private function handleRequest(): void
{
    $request = $this->reg->getRequest();
    $controller = new AppController();
    $cmd = $controller->getCommand($request);
    $cmd->execute($request);
    $view = $controller->getView($request);
    $view->render($request);
}
public static function run(): void
{
    $instance = new self();
    $instance->init();
    $instance->handleRequest();
}
private function init(): void
{
    $this->reg->getApplicationHelper()->init();
}
```

Принципиальное отличие от предыдущего примера состоит в том, что теперь компонент представления извлекается вместе с командой, помимо замены имени класса CommandResolver именем AppController, хотя это и косметическая операция. Обратите также внимание на то, что для получения объекта типа Request в данном примере применяется объект типа Registry. Объект типа AppController можно было бы сохранить в реестре, даже если он и не применяется в других компонентах. Классы, в которых исключается непосредственное получение экземпляров, как правило, оказываются более гибкими и легче поддаются тестированию.

Так по какой же логике в классе AppController становится известно, какое именно представление следует связать с конкретной командой? Как

и всегда в объектно-ориентированном коде, интерфейс важнее, чем реализация. Выберем, однако, наиболее приемлемый подход.

Краткий обзор реализации

На различных стадиях выполнения операции в классе Command могут потребоваться разные представления. Представлением по умолчанию для команды типа AddVenue может быть форма ввода данных. Если пользователь вводит неверный тип данных, форма может быть выведена снова или будет выведена страница с сообщением об ошибке. Если все пойдет нормально и культурное заведение будет создано в системе, можно перейти к другому звену в цепочке команд, представленных объектами типа Command (возможно, к команде типа AddSpace).

Объекты типа Command сообщают системе о своем текущем состоянии, устанавливая специальный флаг состояния. Ниже приведены коды, которые распознаются в рассматриваемой здесь минимальной реализации (они задаются в виде свойств в суперклассе Command):

```
// Листинг 12.20
public const CMD_DEFAULT = 0;
public const CMD_OK = 1;
public const CMD_ERROR = 2;
public const CMD_INSUFFICIENT_DATA = 3;
```

Контроллер приложения находит и получает экземпляр нужного класса типа Command, используя объект типа Request. Как только контроллер приложения начнет выполняться, объект типа Command будет связан с соответствующим состоянием. Сочетание объекта типа Command и состояния можно затем сравнить с внутренней структурой данных, чтобы выяснить, какая именно команда должна быть выполнена следующей или же какое представление следует отобразить, если никаких команд выполнять больше не требуется.

Файл конфигурации

Владелец системы может определить, каким образом команды и представления должны работать совместно, указав набор директив в файле конфигурации. Ниже приведен фрагмент такого файла:

```
// Листинг 12.21
<woo-routing>
  <control>
    <command path="/" class="\popp\ch12\batch06\DefaultCommand">
```

```
      <view name="main" />
      <status value="CMD_ERROR">
        <view name="error" />
      </status>
    </command>

    <command path="/listvenues"
             class="\popp\ch12\batch06\ListVenues">
      <view name="listvenues" />
    </command>

    <command path="/quickaddvenue"
             class="\popp\ch12\batch06\AddVenue">
      <view name="quickadd" />
    </command>

    <command path="/addvenue" class="\popp\ch12\batch06\AddVenue">
      <view name="addvenue" />
      <status value="CMD_OK">
        <forward path="/addspace" />
      </status>
    </command>
    <command path="/addspace" class="\popp\ch12\batch06\AddSpace">
      <view name="addspace" />
      <status value="CMD_OK">
        <forward path="/listvenues" />
      </status>
    </command>
  </control>
</woo-routing>
```

На примере данного упрощенного фрагмента кода XML демонстрируется одна из стратегий абстрагирования потока команд и его взаимосвязи с представлениями в классах типа Command. Все эти директивы содержатся в элементе control.

В каждом элементе command определяются атрибуты path и class, в которых описывается элементарное соответствие команд имен классов. Но логика для представлений оказывается более сложной. Так, в элементе view на верхнем уровне разметки команды определяется ее взаимосвязь с представлением по умолчанию. Иными словами, если ни одно из более конкретных условий не совпадает, для данной команды выбирается данное представление view. Конкретные условия определяются в элементах status, в которых значения атрибутов value должны совпадать с одним из упомянутых выше состояний команд. Так, если выполнение коман-

ды приводит к состоянию CMD_OK, то, если данное состояние определено в XML-документе, используется соответствующий элемент view.

В элементе view определяется атрибут name, значение которого служит для составления пути к файлу шаблона, который затем включается в проект.

Элементы command и status могут содержать элемент forward, а не view. Рассматриваемая здесь система интерпретирует элемент forward как представление особого рода, повторно вызывающее приложение по новому пути вместо воспроизведения файла шаблона.

Рассмотрим следующий фрагмент XML-разметки в свете приведенного выше пояснения:

```
// Листинг 12.22
<command path="/addvenue" class="\popp\ch12\batch06\AddVenue">
    <view name="addvenue" />
    <status value="CMD_OK">
        <forward path="/addspace" />
    </status>
</command>
```

Когда из запроса извлекается заданный путь /addvenue, вызывается команда типа AddVenue. Затем формируется код состояния CMD_DEFAULT, CMD_OK, CMD_ERROR или CMD_INSUFFICIENT_DATA. Далее для любого кода состояния, кроме CMD_OK, вызывается шаблон addvenue. Но если команда возвращает код состояния CMD_OK, то это означает, что условие совпадает. Элемент status мог бы просто содержать другое представление, которое должно быть выбрано вместо представления по умолчанию. Но в данном случае вступает в действие элемент разметки forward. Отсылая к другой команде, файл конфигурации возлагает на новый элемент разметки всю ответственность за обработку представлений. И тогда система будет снова запущена на выполнение по новому запросу с заданным путем /addspace.

Синтаксический анализ файла конфигурации

Благодаря расширению SimpleXML отпадает необходимость в непосредственном выполнении синтаксического анализа, поскольку это делается автоматически. Остается лишь обойти структуру данных SimpleXML и сформировать собственные данные. Ниже приведено определение класса ViewComponentCompiler, в котором все это реализуется на практике:

```
// Листинг 12.23
class ViewComponentCompiler
```

```
{
    private static $defaultcmd = DefaultCommand::class;
    public function parseFile(string $file): Conf
    {
        $options = \simplexml_load_file($file);
        return $this->parse($options);
    }
    public function parse(\SimpleXMLElement $options): Conf
    {
        $conf = new Conf();
        foreach ($options->control->command as $command)
        {
            $path = (string) $command['path'];
            $cmdstr = (string) $command['class'];
            $path = (empty($path)) ? "/" : $path;
            $cmdstr = (empty($cmdstr)) ? self::$defaultcmd:$cmdstr;
            $pathobj = new ComponentDescriptor($path, $cmdstr);
            $this->processView($pathobj, 0, $command);
            if (isset($command->status) &&
                isset($command -> status['value']))
            {
                foreach ($command->status as $statusel)
                {
                    $status = (string)$statusel['value'];
                    $statusval = constant(Command::class
                                          . "::" . $status);
                    if (is_null($statusval))
                    {
                        throw new AppException(
                            "Неизвестное состояние: {$status}");
                    }
                    $this->processView($pathobj, $statusval,
                                       $statusel);
                }
            }
            $conf->set($path, $pathobj);
        }
        return $conf;
    }
    public function processView(ComponentDescriptor $pathobj,
                                int $statusval,
                                \SimpleXMLElement $el): void
    {
        if (isset($el->view) && isset($el->view['name']))
        {
            $pathobj->setView($statusval,
                              new TemplateViewComponent((string)
```

```
                            $el->view['name']));
        }

        if (isset($el->forward) && isset($el->forward['path']))
        {
            $pathobj->setView($statusval,
                new ForwardViewComponent(
                    (string)$el->forward['path']));
        }
    }
}
```

Реальное действие происходит в методе `parse()`, которому передается объект типа `SimpleXMLElement` для обхода. Сначала получается объект типа `Conf` (напомним, что этот объект служит всего лишь оболочкой для массива), а затем осуществляется обход элементов команды в XML-документе. Для каждой команды извлекаются значения атрибутов `path` и `class`, а полученные данные передаются конструктору класса `ComponentDescriptor`. Объект этого класса обработает пакет информации, связанной с элементом разметки `command`. Этот класс будет представлен ниже.

Далее вызывается закрытый метод `processView()`, которому передаются объект типа `ComponentDescriptor`, нулевое целочисленное значение, поскольку обрабатывается стандартное состояние, а также ссылка на текущий элемент XML-разметки, фактически содержащий конкретную команду. В зависимости от того, что будет обнаружено во фрагменте XML-разметки, в методе `processView()` будет создан объект либо типа `TemplateViewComponent`, либо типа `ForwardViewComponent`, который далее передается вызываемому методу `ComponentDescriptor::setView()`. Если же никакого совпадения не произойдет, то ничего не будет вызвано, что, конечно, нежелательно. Поэтому в более полной реализации такое условие следовало бы трактовать как ошибочное.

Обработка атрибутов элемента разметки `status` начинается в методе `parse()`. С этой целью снова вызывается метод `processView()`, но на этот раз с целочисленным значением, соответствующим символьной строке из атрибута `value` элемента `status`. Иными словами, символьная строка `"CMD_OK"` преобразуется в целочисленное значение `1`, символьная строка `"CMD_ERROR"` — в целочисленное значение `2` и т.д. Метод PHP `constant()` аккуратно выполняет такое преобразование. В итоге методу `processView()` передаются ненулевое целочисленное значение, а также элемент `status`.

Объект типа ComponentDescriptor снова заполняется в методе processView() любыми обнаруженными объектами, реализующими компоненты представлений.

И наконец объект типа ComponentDescriptor сохраняется в объекте типа Conf по индексу значения пути к компоненту команды. По окончании цикла возвращается объект типа Conf.

Возможно, чтобы разобраться в рассмотренном здесь процессе, вам придется прочитать его описание еще раз, но на самом деле он довольно прост. В классе ViewComponentCompiler создается массив объектов типа ComponentDescriptor, заключаемый, как и прежде, в оболочку объекта типа Conf. В каждом объекте типа ComponentDescriptor хранятся сведения о пути, команде, реализуемой в классе Command, а также о массиве объектов, реализующих компоненты представлений для отображения или вызова другой команды. Этот массив проиндексирован по коду состояния (принимающему нулевое значение для представления по умолчанию).

Несмотря на весь этот антураж, сам описываемый здесь процесс, в общем, очень прост и состоит в создании взаимосвязей между потенциальными запросами с одной стороны и командами — с другой (рис. 12.6).

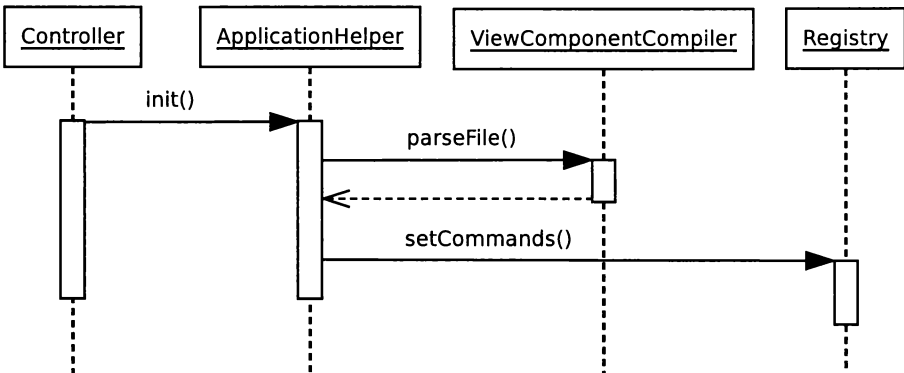


Рис. 12.6. *Компиляция команд и представлений*

Управление данными компонентов

Как было показано выше, объекты типа ComponentDescriptor хранятся в объекте типа Conf, который служит оболочкой для получения и установки элементов ассоциативного массива. В качестве ключей этого массива служат пути, распознаваемые в системе (например, / или /addvenue).

Рассмотрим класс `ComponentDescriptor`, предназначенный для управления командами, представлениями и пересылки информации:

```
// Листинг 12.24
class ComponentDescriptor
{
    private array $views = [];
    public function __construct(private string $path,
                                private string $cmdstr)
    {
    }
    public function getCommand(): Command
    {
        $class = $this->cmdstr;
        if (is_null($class))
        {
            throw new AppException("Неизвестный класс '$class'");
        }
        if (! class_exists($class))
        {
            throw new AppException("Класс '$class' не найден");
        }
        $refclass = new \ReflectionClass($class);
        if (! $refclass->isSubClassOf(Command::class))
        {
            throw new AppException("'$class' не является Command");
        }
        return $refclass->newInstance();
    }
    public function setView(int $status, ViewComponent $view): void
    {
        $this->views[$status] = $view;
    }
    public function getView(Request $request): ViewComponent
    {
        $status = $request->getCmdStatus();
        $status = (is_null($status)) ? 0 : $status;
        if (isset($this->views[$status]))
        {
            return $this->views[$status];
        }
        if (isset($this->views[0]))
        {
            return $this->views[0];
        }
        throw new AppException("no view found");
    }
}
```

Как видите, в данном классе не только организуются хранение и извлечение данных, но и делается кое-что еще. Сведения о команде (полное имя класса, реализующего команду) вводятся через конструктор и только по требованию преобразуются в объект типа Command при вызове метода getCommand(). Процедура получения экземпляра и проверки происходит в закрытом методе resolveCommand(). Его исходный код выглядит знакомо, поскольку фактически он заимствован из класса CommandResolver, упоминавшегося ранее в этой главе, вернее — навеян его эквивалентными функциональными возможностями.

Таким образом, этот небольшой класс обеспечивает всю логику, необходимую для обработки элементов разметки command из XML-файла. Сам же класс контроллера приложения оказывается относительно скромным, поскольку все операции выполняются во вспомогательных классах. Ниже приведено определение этого класса:

```
// Листинг 12.25
class AppController
{
    private static string $defaultcmd = DefaultCommand::class;
    private static string $defaultview = "fallback";
    public function getCommand(Request $request): Command
    {
        try
        {
            $descriptor = $this->getDescriptor($request);
            $cmd = $descriptor->getCommand();
        }
        catch (AppException $e)
        {
            $request->addFeedback($e->getMessage());
            return new self::$defaultcmd();
        }
        return $cmd;
    }
    public function getView(Request $request): ViewComponent
    {
        try
        {
            $descriptor = $this->getDescriptor($request);
            $view = $descriptor->getView($request);
        }
        catch (AppException)
        {
            return new TemplateViewComponent(self::$defaultview);
```

```
        }
        return $view;
    }
    private function getDescriptor(Request $request):
        ComponentDescriptor
    {
        $reg = Registry::instance();
        $commands = $reg->getCommands();
        $path = $request->getPath();
        $descriptor = $commands->get($path);

        if (is_null($descriptor))
        {
            throw new AppException("Нет дескриптора для {$path}",
                                   404);
        }

        return $descriptor;
    }
}
```

В этом классе в действительности присутствует немного логики, поскольку бо́льшая часть сложных операций вынесена в различные вспомогательные классы. В методах `getCommand()` и `getView()` вызывается метод `getDescriptor()` с целью получить объект типа `ComponentDescriptor` для текущего запроса. Текущий путь получается из объекта типа `Request` в методе `getDescriptor()` и используется далее для извлечения объекта типа `ComponentDescriptor` из объекта типа `Conf`, который также сохраняется в реестре и возвращается методом `getCommands()`. Напомним, что этот объект служит оболочкой для массива объектов типа `ComponentDescriptor` и ранее был заполнен потенциальными путями в качестве ключей с помощью объекта типа `ViewComponentCompiler`.

Как только будет получен объект типа `ComponentDescriptor`, в каждом из методов `getCommand()` и `getView()` можно вызвать соответствующий метод. Так, в методе `getCommand()` вызывается метод `ComponentDescriptor::getCommand()`, а в методе `getView()` — метод `ComponentDescriptor::getView()`.

Прежде чем продолжить обсуждение, необходимо уточнить некоторые подробности. Теперь команды, реализуемые объектами типа `Command`, сами больше не вызывают представления, и поэтому для воспроизведения шаблонов требуется какой-нибудь механизм. Для этой цели служат объ-

екты класса TemplateViewComponent, реализующего приведенный ниже интерфейс ViewComponent:

```
// Листинг 12.26
interface ViewComponent
{
    public function render(Request $request): void;
}
```

Ниже приведено определение класса TemplateViewDisplay для отображения представлений шаблонов:

```
// Листинг 12.27
class TemplateViewComponent implements ViewComponent
{
    public function __construct(private string $name)
    {
    }
    public function render(Request $request): void
    {
        $reg = Registry::instance();
        $conf = $reg->getConf();
        $path = $conf->get("templatepath");

        if (is_null($path))
        {
            throw new AppException("Не найден каталог шаблонов");
        }
        $fullpath = "{$path}/{$this->name}.php";
        if (! file_exists($fullpath))
        {
            throw new AppException("Нет шаблона в {$fullpath}");
        }
        include($fullpath);
    }
}
```

Экземпляр этого класса получается с именем, которое затем используется в методе render() во время визуализации вместе со значением пути к шаблону из файла конфигурации. Ниже приведено определение класса ForwardViewComponent для пересылки компонентов представлений:

```
// Листинг 12.28
class ForwardViewComponent implements ViewComponent
{
```

```
    public function __construct(private ? string $path)
    {
    }
    public function render(Request $request): void
    {
        $request->forward($this->path);
    }
}
```

В этом классе просто вызывается метод `forward()` предоставленного объекта типа `Request`. Конкретная реализация метода `forward()` зависит от подтипа, производного от `Request`. Так, для типа `HttpRequest` в этом методе устанавливается заголовок `"Location"` (Местоположение), как показано ниже:

```
// Листинг 12.29
//HttpRequest
public function forward(string $path): void
{
    header("Location: {$path}");
    exit;
}
```

Что же касается типа `CliRequest`, то для пересылки нельзя полагаться на сервер, поэтому придется выбрать другой подход:

```
// Листинг 12.30
// CliRequest
public function forward(string $path): void
{
    // Добавить новый путь к концу списка аргументов,
    // где преимущество получает последний аргумент
    $_SERVER['argv'][] = "path:{$path}";
    Registry::reset();
    Controller::run();
}
```

Здесь выгодно используется тот факт, что во время синтаксического анализа пути в массиве аргументов последнее обнаруженное совпадение в конечном итоге сохраняется в объекте типа `Request`. Для этого нужно лишь ввести аргумент, задающий путь, очистить реестр и запустить контроллер на выполнение сверху.

А поскольку полный цикл на этом завершается, здесь уместно сделать краткий обзор!

Стратегии, применяемые в контроллере приложения для получения представлений и команд, могут сильно разниться, но самое главное, что они скрыты от остальной системы. На рис. 12.7 показан в общих чертах процесс, в ходе которого контроллер приложения, построенный в классе `AppController` в соответствии с шаблоном Application Controller, применяется в классе `Controller`, созданном в соответствии с шаблоном Front Controller, для получения сначала первой команды, реализуемой объектом типа `Command`, а затем и представления.

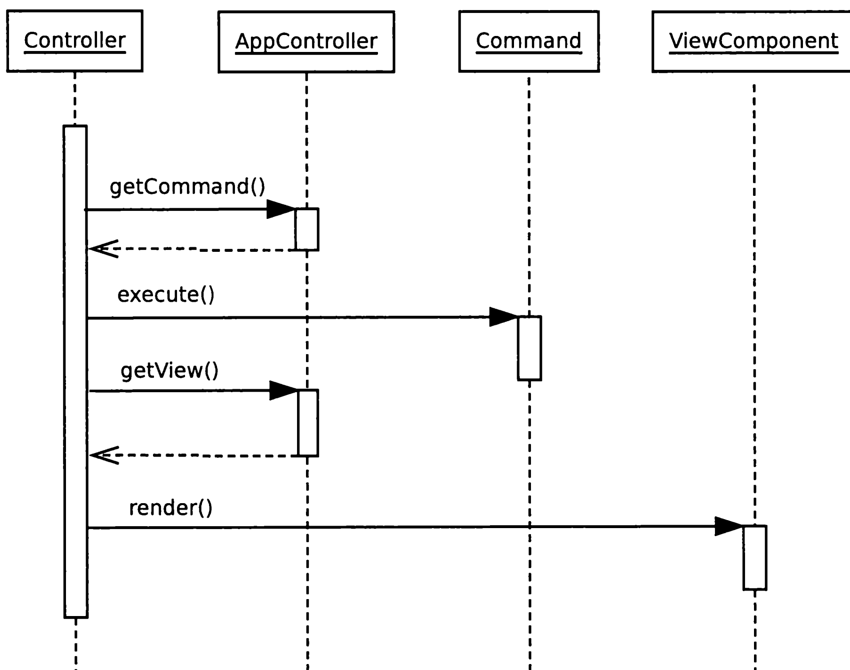


Рис. 12.7. *Получение команд и представлений с помощью контроллера приложения*

Следует, однако, иметь в виду, что визуализируемое представление, показанное на рис. 12.7, может относиться к типу `ForwardViewComponent` (который начинает весь процесс заново по новому пути) или же к типу `TemplateViewComponent` (который включит файл шаблона). Напомним, что данные, необходимые для получения объектов типа `Command` и `ViewComponent`, реализующих команды и компоненты представлений, как показано на рис. 12.7, были сформированы в упоминавшемся ранее

классе ApplicationHelper. Ниже для справки приведен общий вид кода, позволяющий достичь этой цели:

```
// Листинг 12.31
private function setupOptions(): void
{
    //...
    $vcfile = $conf->get("viewcomponentfile");
    $cparse = new ViewComponentCompiler();
    $commandandviewdata = $cparse->parseFile($vcfile);
    $this->reg->setCommands($commandandviewdata);
}
```

Класс Command

А теперь, когда команды больше не отвечают за отображение своих шаблонов, целесообразно рассмотреть вкратце их базовый класс и его реализацию. Как было показано ранее, в классе Command устанавливаются коды состояния, но в нем выполняются и другие служебные операции, заслуживающие упоминания:

```
// Листинг 12.32
abstract class Command
{
    public const CMD_DEFAULT = 0;
    public const CMD_OK = 1;
    public const CMD_ERROR = 2;
    public const CMD_INSUFFICIENT_DATA = 3;
    final public function __construct()
    {
    }
    public function execute(Request $request): void
    {
        $status = $this->doExecute($request);
        $request->setCmdStatus($status);
    }
    abstract protected function doExecute(Request $request): int;
}
```

В хорошем примере применения шаблона Template Method в методе execute() вызывается абстрактный метод doExecute(), а возвращаемое значение сохраняется в объекте типа Request. Это значение будет использовано некоторое время спустя в объекте типа ComponentDescriptor для выбора подходящего возвращаемого представления.

Конкретная команда

Ниже показано, насколько просто может быть реализована конкретная команда типа AddVenue:

```
// Листинг 12.33
class AddVenue extends Command
{
    protected function doExecute(Request $request): int
    {
        $name = $request->getProperty("venue_name");
        if (is_null($name))
        {
            $request->addFeedback("Имя не предоставлено");
            return self::CMD_INSUFFICIENT_DATA;
        }
        else
        {
            // Некоторые действия
            $request->addFeedback("'{$name}' added");
            return self::CMD_OK;
        }
        return self::CMD_DEFAULT;
    }
}
```

На самом деле здесь отсутствует функциональный код, связанный с построением объекта типа Venue и его сохранением в базе данных, но мы до всего этого еще дойдем. Пока что важно подчеркнуть, что выполняемая команда возвращает разные коды состояния в зависимости от сложившихся обстоятельств. Как было показано ранее, разные коды состояния приведут к выбору и возврату разных представлений из контроллера приложения. Так, если воспользоваться приведенным ранее примером XML-файла конфигурации, то в результате возврата кода состояния CMD_OK механизм пересылки запустит ее только по пути /addspace, причем это произойдет только для пути /addvenue. Если в запросе, по которому вызывается данная команда, указан путь /quickaddvenue, то никакой пересылки не произойдет и отобразится представление quickaddvenue, хотя команде типа AddVenue об этом ничего неизвестно. Она просто придерживается своих основных обязанностей.

Следствия

Реализовать полноценный пример шаблона Application Controller нелегко, потому что придется немало потрудиться, чтобы получить и применить метаданные, описывающие взаимосвязи команды с запросом, представлением и с другими командами.

Поэтому я стараюсь реализовать нечто подобное, когда совершенно очевидно, что это необходимо для приложения. Мне словно кто-то нашептывает это, когда я ввожу условные операторы в команды, вызывающие разные представления или другие команды в зависимости от обстоятельств. И в этот момент я понимаю, что поток команд и логика отображения начинают выходить из-под моего контроля.

Безусловно, в контроллере приложения могут применяться всевозможные механизмы для установления взаимосвязей команд с представлениями, а не только тот подход, который был применен здесь. Даже если начать с фиксированной взаимосвязи строки запроса с наименованием команды и представлением, все равно можно извлечь немалую выгоду из построения контроллера приложения, в котором все это инкапсулировано. Это обеспечит значительную степень гибкости, когда потребуется реорганизовать код, чтобы приспособиться к более высокому уровню сложности.

Шаблон Page Controller

Несмотря на то что мне очень нравится шаблон Front Controller, применять его целесообразно далеко не всегда. Применение данного шаблона на стадии предварительного проектирования может иметь положительные последствия для крупных систем и отрицательные — для простых проектов, которые нужно быстро реализовать. Здесь нам может пригодиться шаблон Page Controller (Контроллер страниц), с которым вы, вероятно, уже знакомы, поскольку это достаточно распространенная стратегия проектирования. Тем не менее некоторые вопросы его применения все же стоит обсудить.

Проблема

И в данном случае задача состоит в том, чтобы управлять взаимосвязью запроса, логики приложения и представления данных. В корпоративных проектах это практически постоянная задача, и изменяются только накладываемые ограничения.

Так, если имеется относительно простой проект, в котором развертывание крупной многофункциональной структуры приведет только к срыву сроков сдачи проекта, не дав практически никаких преимуществ, то шаблон Page Controller станет удобным инструментальным средством управления запросами и представлениями.

Допустим, требуется вывести страницу, на которой отображается список всех культурных заведений в рассматриваемой здесь системе. Даже если у нас уже имеется готовый код для извлечения такого списка из базы данных, без применения шаблона Page Controller перед нами все равно встанет трудная задача достижения такого простого на первый взгляд результата.

Итак, список культурных заведений должен отображаться на странице представления, и для его получения необходимо выполнить запрос. При обработке запроса допускается возможность возникновения ошибок, поэтому в полученном результате не всегда будет отображаться новое представление, как можно было бы ожидать при решении сложной задачи. Поэтому в данном случае проще и эффективнее всего связать представление с контроллером, что зачастую делается на одной и той же странице.

Реализация

Несмотря на то что при применении шаблона Page Controller могут возникать определенные трудности, сам шаблон довольно прост. В частности, контроллер, в соответствии с этим шаблоном, связан с отдельным представлением или рядом представлений. В простейшем случае это означает, что контроллер находится в самом представлении, хотя его можно отделить, особенно если одно представление тесно связано с другими представлениями (т.е. в том случае, если в различных ситуациях требуется отсылать браузер на разные страницы). Ниже приведен простейший пример реализации шаблона Page Controller:

```
// Листинг 12.34
<?php
namespace popp\ch12\batch07;

try {
    $venuemapper = new VenueMapper();
    $venues = $venuemapper->findAll();
} catch (\Exception) {
    include('error.php');
```

```
    exit(0);
}

// Далее идет страница по умолчанию
?>
<html>
<head>
<title>Список заведений</title>
</head>
<body>
    <h1>Список заведений</h1>
    <?php foreach ($venues as $venue) { ?>
        <?php print $venue->getName(); ?><br />
    <?php } ?>
</body>
</html>
```

Этот HTML-документ состоит из двух элементов разметки. Элемент разметки представления управляет отображением, а элемент разметки контроллера управляет запросом и вызывает логику приложения. И хотя представление и контроллер находятся на одной и той же странице, они строго разделены.

К этому примеру мало что можно добавить, кроме взаимодействия с базой данных, которое выполняется за кулисами (о чем подробнее речь пойдет в следующей главе). Блок кода PHP вверху страницы пытается получить список всех объектов типа `Venue`, который он сохраняет в глобальной переменной `$venues`.

Если возникает ошибка, данная страница поручает ее обработку странице `error.php` с помощью функции `include()`, после вызова которой следует вызов функции `exit()`, чтобы прекратить любую дальнейшую обработку текущей страницы. С тем же успехом можно было бы применить механизм переадресации сетевого протокола HTTP. Если же метод `include()` не вызывается, то внизу страницы отображается заданное представление, размеченное в коде HTML.

Комбинация контроллеров и представлений показана на рис. 12.8.

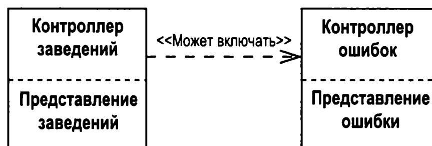


Рис. 12.8. *Контроллеры страниц, встроенные в представления*

Как видите, мы довольно быстро реализовали шаблон Page Controller в виде контрольного примера, но при создании крупных систем, вероятно, потребуется приложить намного больше усилий, чем в данном случае. Если раньше код контроллера страниц в соответствии с шаблоном Page Controller был неявно отделен от представления, то теперь сделаем это разделение явным, начав с определения элементарного базового класса `PageController`:

```
// Листинг 12.35
abstract class PageController
{
    private Registry $reg;
    abstract public function process(): void;
    public function __construct()
    {
        $this->reg = Registry::instance();
    }
    public function init(): void
    {
        if (isset($_SERVER['REQUEST_METHOD']))
        {
            $request = new HttpRequest();
        }
        else
        {
            $request = new CliRequest();
        }
        $this->reg->setRequest($request);
    }
    public function forward(string $resource): void
    {
        $request = $this->getRequest();
        $request->forward($resource);
    }
    public function render(string $resource, Request $request): void
    {
        include($resource);
    }
    public function getRequest(): Request
    {
        return $this->reg->getRequest();
    }
}
```

В данном классе используются уже рассматривавшиеся ранее средства и, в частности, классы `Request` и `Registry`. Основное назначение класса

PageController — обеспечить доступ к объекту типа Request и управлять включением представлений. Это назначение класса PageController быстро расширится в реальном проекте по мере того, как у дочерних классов будет возникать потребность в общих функциональных возможностях.

Дочерний класс может находиться в самом представлении, и поэтому оно, как и прежде, отображается по умолчанию или может находиться отдельно от представления. Второй подход кажется более подходящим, поэтому мы выбрали именно его. Приведем пример реализации класса PageController, в котором предпринимается попытка добавить новое культурное заведение в рассматриваемую здесь систему:

```
// Листинг 12.36
class AddVenueController extends PageController
{
    public function process(): void
    {
        $request = $this->getRequest();
        try
        {
            $name = $request->getProperty('venue_name');
            if (is_null($request->getProperty('submitted')))
            {
                $request->addFeedback("Укажите название заведения");
                $this->render(__DIR__ . '/view/add_venue.php',
                              $request);
            }
            elseif(is_null($name))
            {
                $request->addFeedback("Название — обязательное поле");
                $this->render(__DIR__ . '/view/add_venue.php',
                              $request);
                return;
            }
            else
            {
                // Добавление в базу данных
                $this->forward('listvenues.php');
            }
        }
        catch (Exception)
        {
            $this->render(__DIR__ . '/view/error.php', $request);
        }
    }
}
```

В классе AddVenueController реализован только метод process(), который отвечает за проверку данных, введенных пользователем. Если пользователь не заполнил форму или сделал это неправильно, то включается представление по умолчанию (add_venue.php), обеспечивающее реакцию на действия пользователя и выводящее форму. Если же новое заведение добавлено успешно, то в данном методе вызывается метод forward(), который отсылает пользователя к контроллеру страниц типа ListVenues.

Обратите внимание на формат, использованный для представления. Файлы представлений лучше отличать от файлов классов, используя в именах первых строчные буквы, а в именах вторых — смешанное написание, при котором на границах слов, обозначаемых строчными буквами, употребляются прописные буквы.

Как видите, в классе AddVenueController ничего не предусмотрено для его приведения в действие. Код, приводящий этот класс в действие, можно было бы разместить в том же самом файле, но это затруднило бы тестирование, поскольку сама операция включения файла данного класса привела бы к выполнению его методов. Именно по этой причине мы создали отдельный запускающий сценарий для каждой страницы. Ниже представлен пример такого сценария из файла addvenue.php для приведения в действие класса AddVenueController и выполнения соответствующей команды:

```
// Листинг 12.37
$addvenue = new AddVenueController();
$addvenue->init();
$addvenue->process();
```

А вот как выглядит представление, связанное с классом AddVenueController:

```
// Листинг 12.38
<html>
    <head>
        <title>Добавление заведения</title>
    </head>
    <body>
        <h1>Добавление заведения</h1>
        <table>
            <tr>
            <td>
            <?php
                print $request->getFeedbackString(
                             "</td></tr><tr><td>");
            ?>
```

```
            </td>
            </tr>
            </table>
            <form action="/addvenue.php" method="get">
                <input type="hidden" name="submitted" value="yes"/>
                <input type="text" name="venue_name" />
            </form>
    </body>
</html>
```

Как видите, в этом представлении не делается ничего, кроме отображения данных и обеспечения механизма формирования нового запроса. Этот запрос адресован классу `PageController` (через запускающий сценарий из файла `/addvenue.php`), а не обратно представлению. Напомним, что именно класс `PageController` отвечает за обработку запросов.

На рис. 12.9 схематически показан более сложный вариант реализации шаблона Page Controller.

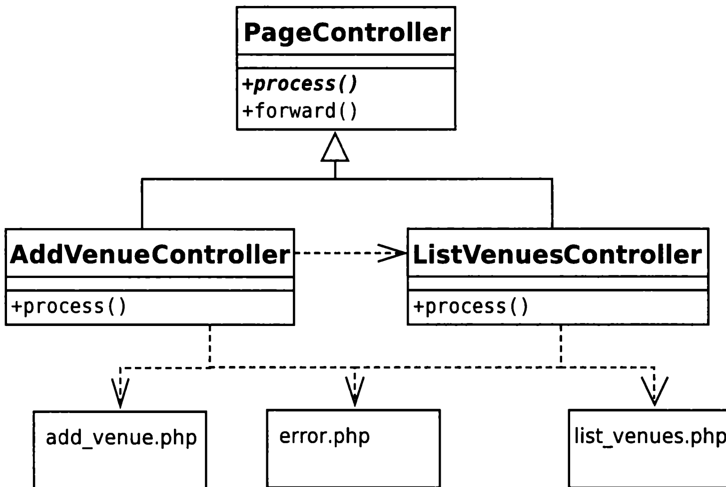


Рис. 12.9. *Иерархия класса* `PageController` *и его включаемые взаимосвязи*

Следствия

Представленному здесь подходу присуще следующее явное преимущество: он понятен всем, кто имеет хотя бы минимальный опыт разработки веб-приложений. Мы делаем запрос на вывод перечня культурных заведений из файла `venues.php` и получаем именно то, что нужно. Даже ошибки

находятся в пределах ожидания, поскольку с сообщениями типа "Server error" (Ошибка на сервере) и "Page not found" (Страница не найдена) приходится сталкиваться постоянно.

Но дело немного усложнится, если отделить представление от класса контроллера страниц, хотя тесная взаимно однозначная связь между участниками останется вполне очевидной. Контроллер страниц включает свое представление по окончании обработки, хотя иногда он может отослать на другую страницу. Следовательно, когда, например, по команде типа AddVenue успешно вводится культурное заведение, отображать дополнительную форму для ввода данных больше не придется, поскольку эта обязанность поручается другому контроллеру страниц типа ListVenues. И это делается в методе forward() из класса PageController, в котором, как и в упоминавшемся ранее классе ForwardViewComponent, метод forward() просто вызывается для объекта типа Request.

Несмотря на то что класс контроллера страниц может делегировать свои полномочия объектам типа Command, преимущество такого подхода не так очевидно, как при реализации шаблона Front Controller. В классах, создаваемых в соответствии с шаблоном Front Controller, необходимо сначала выяснить цель запроса, тогда как в классах, создаваемых в соответствии с шаблоном Page Controller, об этом уже известно. Простая проверка параметров запроса и вызовы логики приложения, которые можно было бы разместить в классе Command, с той же легкостью размещаются в классе контроллера страниц. И при этом можно извлечь выгоду из того факта, что для выбора объектов типа Command не требуется отдельный механизм.

Дублирование может причинить хлопоты, которые, впрочем, можно преодолеть, воспользовавшись общим суперклассом. Это позволит также сократить время инициализации приложения, поскольку можно исключить загрузку данных, которые не понадобятся в текущем контексте. Безусловно, это можно сделать и в соответствии с шаблоном Front Controller, но выяснить, что же действительно нужно, а без чего вообще можно обойтись, будет намного сложнее.

Настоящий недостаток шаблона Page Controller проявляется в тех случаях, когда пути через представления оказываются сложными, особенно когда одно и то же представление используется по-разному в разное время (характерным тому примером служат экраны для ввода и редактирования данных). В процессе проектирования можно легко запутаться в условных операторах и проверках состояния и в конечном итоге утратить общее представление о проектируемой системе.

Нельзя, однако, начать с шаблона Page Controller и затем перейти к шаблону Front Controller. И это особенно справедливо, если используется суперкласс `PageController`. Если, по моим оценкам, мне требуется меньше недели, чтобы завершить проектирование системы, и в дальнейшем расширять ее не придется, то я, как правило, выбираю шаблон Page Controller, извлекая выгоду из быстрого выполнения проекта. Если же я работаю над крупным проектом со сложной логикой представлений, который со временем будет постепенно расширяться, то я обычно останавливаю свой выбор на шаблоне Front Controller.

Шаблоны Template View и View Helper

Шаблон Template View (Представление шаблона) дает практически то же самое, чего можно добиться в PHP по умолчанию, разместив на одной странице разметку представления (в виде HTML-кода) вместе с кодом нашего веб-приложения (непосредственно на PHP). Но, как упоминалось ранее, это и хорошо, и плохо. Простота, с которой все это можно объединить, вызывает искушение разместить логику приложения и отображения в одном месте. Но последствия такого решения могут быть весьма плачевными.

К тому же программирование представления на PHP — это вопрос многих ограничений. И если ваш код выполняет не только отображение данных, то относитесь к нему с большим недоверием.

С этой целью в шаблоне View Helper (Вспомогательное средство представления; см. упоминавшуюся ранее книгу Дипака Алура и др.) предоставляется вспомогательный класс, который может быть характерным для конкретного представления или общим для нескольких представлений. В этом классе размещается весь программный код, требующийся для решения конкретной задачи.

Проблема

В настоящее время SQL-запросы и другая логика приложения все реже помещаются непосредственно на страницы отображения, но иногда это все же случается. Об этой напасти немало упоминалось в предыдущих главах, поэтому будем кратки.

С веб-страницами, содержащими слишком много кода, трудно работать дизайнерам, поскольку компоненты представления опутаны циклами и условными операторами.

Логика приложения, внедренная на уровне представления, вынуждает придерживаться именно такого интерфейса. А без переноса большого объема прикладного кода так просто перейти к новому представлению не удастся.

Системы, в которых представление отделено от их логики, легко поддаются тестированию, поскольку тестированию подвергаются функциональные возможности на уровне логики, изолированные от перегруженного и отвлекающего внимание уровня представления.

Кроме того, в тех системах, в которых логика приложения встраивается на уровне представления, часто возникают нарушения системы безопасности. В таких системах код запросов к базе данных и код обработки введенных пользователем данных нередко оказываются распределенными по разным таблицам, формам и спискам, что сильно затрудняет анализ и выявление потенциальных угроз нарушения безопасности.

Многие операции повторяются от одного представления к другому, и поэтому те системы, в которых прикладной код встроен в шаблон, обычно становятся жертвой дублирования, потому что одни и те же структуры кода переносятся с одной страницы на другую. И когда это происходит, неизбежно возникают программные ошибки и трудности сопровождения системы.

Чтобы предотвратить описанные выше затруднения, управление работой приложения необходимо перенести в другое место, а представлениям следует дать возможность управлять только отображением данных. Как правило, этого можно достичь, сделав представления пассивными получателями данных. Но если в представлении требуется сделать запрос к системе, то объекту класса, построенному в соответствии с шаблоном View Helper, целесообразно предоставить возможность выполнить всю рутинную работу от имени представления.

Реализация

Стоит создать общий каркас приложения, так как программирование на уровне представления перестанет быть слишком сложным. Безусловно, остается еще большой вопрос проектирования информационной архитектуры, но его рассмотрение выходит за рамки данной книги.

Шаблон Template View был назван так Мартином Фаулером. Это основной шаблон для большинства разработчиков корпоративных приложений. В некоторых языках программирования его реализация может включать

создание системы шаблонной обработки, в которой дескрипторы преобразуются в значения, установленные в системе. В языке PHP также имеется подобная возможность. В частности, механизм шаблонной обработки можно использовать подобно замечательному инструментальному средству Twig. Мы же предпочитаем пользоваться встроенными средствами PHP, но делать это осторожно.

Чтобы представление смогло что-нибудь отобразить, у него должна быть возможность получать данные. С этой целью определим вспомогательный класс в соответствии с шаблоном View Helper, чтобы воспользоваться им в представлениях. Ниже приведен пример определения простого класса в соответствии с шаблоном View Helper:

```
// Листинг 12.39
class ViewHelper
{
    public function sponsorList(): string
    {
        // Некоторая сложная работа по
        // получению списка спонсоров
        return "Обувной супермаркет Боба";
    }
}
```

Все, что в настоящий момент делает этот класс, — это предоставляет список спонсоров в форме символьной строки. Допустим, имеется относительно сложный процесс получения или форматирования этих данных и последующего их встраивания в сам шаблон. Этот процесс можно расширить, чтобы предоставить дополнительные функциональные возможности по мере дальнейшего развития разрабатываемого приложения. Если же для выполнения какой-нибудь операции в представлении требуется больше двух строк кода, то, скорее всего, такое представление относится к шаблону View Helper. В более крупном приложении можно предоставить несколько объектов из иерархии классов, построенных в соответствии с шаблоном View Helper, чтобы обеспечить тем самым разные инструментальные средства для отдельных частей проектируемой системы.

Вспомогательный объект класса, построенного в соответствии с шаблоном View Helper, можно получить из какой-нибудь фабрики — возможно, даже из реестра. С точки зрения шаблона проще всего сделать экземпляр вспомогательного объекта доступным в методе `render()` следующим образом:

```
// Листинг 12.40
public function render(string $resource, Request $request): void
{
    $vh = new ViewHelper();
    // Теперь в шаблоне будет переменная $vh из include($resource);
}
```

Ниже приведено простое представление, в котором используется вспомогательный объект типа ViewHelper:

```
// Листинг 12.41
<html>
    <head>
        <title>Список заведений</title>
    </head>
    <body>
        <h1>Список заведений</h1>
        <div>
            Спонсоры: <?php echo $vh->sponsorList(); ?>
        </div> Listing venues
    </body>
</html>
```

Это представление (из файла list_venues.php) создается в экземпляре класса ViewHelper, хранящегося в переменной $vh. В нем вызывается метод sponsorList() и выводятся соответствующие результаты.

Очевидно, что в данном примере полностью удалить код из представления не удастся, но в нем строго ограничены объем и тип кода, который требуется написать. Страница содержит простые операторы вывода на печать и несколько вызовов методов, но веб-дизайнер сможет разобраться в этом коде без малейших усилий.

Несколько бóльшую трудность представляют условные операторы if и циклы. Их обязанности трудно поручить классу ViewHelper, потому что они обычно связаны с выводом форматированных данных. Как правило, простые условные операторы и циклы, которые зачастую используются при построении таблиц, в которых отображаются строки данных, лучше размещать в классе, созданном в соответствии с шаблоном Template View. Но для того чтобы сделать их как можно более простыми, обязанности по проверке условий лучше делегировать везде, где это только возможно.

Следствия

В том, как данные передаются уровню представления, есть повод для беспокойства. Дело в том, что у представления на самом деле отсутствует фиксированный интерфейс, гарантирующий свое окружение. Каждое представление можно рассматривать как вступающее в контакт со всей системой в целом. Такое представление фактически сообщает приложению следующее: "Если я вызвано, значит, мне дано право получать доступ к `Этому`, `Тому` или `Другому_объекту`". И здесь все зависит от конкретного приложения (разрешать такой доступ или не разрешать).

Представления можно сделать более строгими, предоставив методы доступа во вспомогательных классах. Но зачастую оказывается намного проще регистрировать объекты динамически на уровне представления через объекты типа `Request`, `Response` или `Context`.

Как правило, шаблоны представлений являются пассивными элементами, в которых хранятся данные, полученные в результате выполнения последнего запроса. Но возможны и ситуации, когда в представлении требуется сделать вспомогательный запрос. Для обеспечения именно таких функциональных возможностей вполне подходит шаблон View Helper. В итоге механизм доступа к данным оказывается надежно скрытым от самого представления. Но даже в этом случае класс, построенный в соответствии с шаблоном View Helper, должен выполнять как можно меньше обязанностей, делегируя их командам или связываясь с уровнем приложения через фасад (в соответствии с шаблоном Facade).

На заметку Шаблон `Facade` был представлен в главе 10, "Шаблоны для программирования гибких объектов". Дипак Алур и другие авторы упоминавшейся ранее книги рассматривают только одно его применение — при программировании корпоративных приложений вместе с шаблоном Session Facade, предназначенным для ограничения мелких сетевых транзакций. Мартин Фаулер также описывает шаблон под названием "Service Layer", предоставляющий простую точку доступа к сложным средствам на уровне обслуживания.

Уровень логики приложения

Если на уровне управления организуется взаимодействие с внешним миром и формируется реакция системы на него, то на уровне логики приложения решается, собственно, задача, *ради* которой приложение и было

создано. На данном уровне должно быть как можно меньше хлопот, связанных с анализом строк запросов, созданием HTML-таблиц и составлением ответных сообщений. На уровне логики приложения должны решаться только те задачи, которые диктуются истинным назначением приложения. Все остальное существует только для того, чтобы поддерживать эти задачи.

В классическом объектно-ориентированном приложении уровень логики приложения, как правило, состоит из классов, моделирующих задачи, которые призвана решать система. Как будет показано далее, это гибкое проектное решение, требующее серьезного предварительного планирования. Поэтому начнем с самого быстрого способа настройки и запуска системы на выполнение.

Шаблон Transaction Script

Этот шаблон, взятый из упоминавшейся ранее книги Мартина Фаулера *Patterns of Enterprise Application Architecture*, описывает способ пущенного на самотек развития многих систем. Он прост, интуитивно понятен и эффективен, хотя все эти качества ухудшаются по мере роста системы. Шаблон Transaction Script (Сценарий транзакций) обрабатывает запрос внутри, а не делегирует его специализированным объектам. Это типичный способ получить быстрый результат. Данный шаблон трудно отнести к какой-нибудь категории, поскольку он сочетает элементы из разных уровней, описанных в этой главе. Он представлен здесь как часть уровня логики приложения, поскольку основное назначение данного шаблона — реализовать цели, стоящие перед системой.

Проблема

Каждый запрос должен каким-то образом обрабатываться. Как пояснялось ранее, во многих системах предусмотрен уровень, на котором анализируются и фильтруются входящие данные. Но в идеальном случае на этом уровне должны затем вызываться классы, предназначенные для выполнения запроса. Эти классы можно разделить по выполняемым функциям и обязанностям — возможно, с помощью интерфейса в соответствии с шаблоном Facade. Но такой подход требует серьезного и внимательного отношения к проектированию. Для некоторых проектов (обычно небольших по объему и срочных по характеру) такие издержки разработки могут оказаться неприемлемыми. В таком случае, возможно, придется встроить логику приложения в набор процедурных операций. Каждая операция бу-

дет предназначена для обработки определенного запроса. Поэтому необходимо обеспечить быстрый и эффективный механизм достижения целей системы без потенциально дорогостоящих вложений в сложный проект.

Огромное преимущество данного шаблона заключается в скорости получения результатов. В каждом сценарии для достижения нужного результата обрабатываются входные данные и выполняются операции с базой данных. Помимо организации связанных вместе методов в одном классе и сохранения классов, построенных в соответствии с шаблоном Transaction Script, на их собственном уровне, как можно более независимом от уровней команд, управления и представления, для этого потребуется минимальное предварительное проектирование.

В то время как классы уровня логики приложения четко отделены от уровня представления, они в большей степени внедрены на уровне данных. Дело в том, что выборка и сохранение данных — это ключ к задачам, которые обычно решают подобные классы. Далее в этой главе будут представлены механизмы разделения объектов на уровнях логики приложения и базы данных. Но в классах, построенных в соответствии с шаблоном Transaction Script, обычно известно все о базе данных, хотя в них могут использоваться промежуточные классы для реализации подробностей обработки реальных запросов.

Реализация

Вернемся к примеру обработки списка событий. В данном случае в системе поддерживаются три таблицы реляционной базы данных: `venue`, `space` и `event`. У культурного заведения (`venue`) может быть несколько мест проведения культурных мероприятий (`space`; например, у театра может быть несколько сценических площадок, у танцевального клуба — несколько помещений и т.д.). Ниже показано, как создается схема такой базы данных на языке SQL:

```
CREATE TABLE 'venue' (
    'id' int(11) NOT NULL auto_increment,
    'name' text,
    PRIMARY KEY ('id')
)

CREATE TABLE 'space' (
    'id' int(11) NOT NULL auto_increment,
    'venue' int(11) default NULL,
    'name' text,
```

```
    PRIMARY KEY ('id')
)

CREATE TABLE 'event' (
    'id' int(11) NOT NULL auto_increment,
    'space' int(11) default NULL,
    'start' mediumtext,
    'duration' int(11) default NULL,
    'name' text,
    PRIMARY KEY ('id')
)
```

Очевидно, что в проектируемой системе требуются механизмы для ввода как культурных заведений (venue), так и культурных мероприятий (event). Каждая из этих операций представляет собой одну транзакцию. Следовательно, для каждого метода можно назначить свой класс (и организовать такие классы в соответствии с шаблоном Command, описанным в главе 11, "Выполнение задач и представление результатов"). Но в данном случае мы разместим эти методы в одном классе, хотя он и является частью иерархии наследования, структура которой в соответствии с шаблоном Transaction Script показана на рис. 12.10.

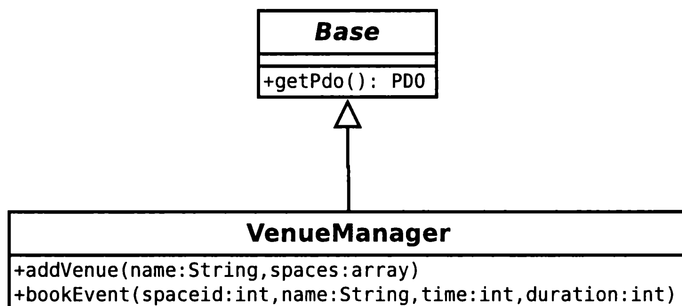


Рис. 12.10. *Структура класса и его суперкласса в соответствии с шаблоном Transaction Script*

Но зачем в данный пример включен абстрактный суперкласс? В сценарии любого размера мы, скорее всего, добавили бы больше конкретных классов к этой иерархии. Дело в том, что по крайней мере некоторые из базовых функциональных средств для работы с базой данных оказываются общими, поэтому их целесообразно разместить в общем родительском классе.

На самом деле это отдельный шаблон, который Мартин Фаулер назвал "Layer Supertype". Если классы имеют общие характеристики на одном уровне, то их целесообразно сгруппировать в один тип, вынеся служебные операции в базовый класс, что и будет продемонстрировано в остальной части этой главы.

В данном случае базовый класс получает объект типа PDO, который сохраняется в его свойстве:

```
// Листинг 12.42
abstract class Base
{
    private \PDO $pdo;
    private string $config = __DIR__ . "/data/woo_options.ini";
    public function __construct()
    {
        $reg = Registry::instance();
        $options = parse_ini_file($this->config, true);
        $conf = new Conf($options['config']);
        $reg->setConf($conf);
        $dsn = $reg->getDSN();
        if (is_null($dsn))
        {
            throw new AppException("DSN не определен");
        }
        $this->pdo = new \PDO($dsn);
        $this->pdo->setAttribute(\PDO::ATTR_ERRMODE,
                                 \PDO::ERRMODE_EXCEPTION);
    }
    public function getPdo(): \PDO
    {
        return $this->pdo;
    }
}
```

Для получения символьной строки DSN (имя источника данных), которая передается конструктору класса PDO, здесь применяется класс Registry. Объект типа PDO становится доступным через метод получения getPdo(). В действительности бо́льшую часть этой рутинной работы можно поручить самому объекту типа Registry. Именно такая стратегия принята в этой и последующей главах.

Ниже приведено начало определения класса VenueManager, в котором составляются SQL-запросы к базе данных:

```
// Листинг 12.43
class VenueManager extends Base
{
    private string $addvenue = "INSERT INTO venue
                                (name)
                                VALUES( ?)";
    private string $addspace = "INSERT INTO space
                                (name, venue)
                                VALUES( ?, ?)";
    private string $addevent = "INSERT INTO event
                                (name, space, start, duration)
                                VALUES( ?, ?, ?, ?)";
    // ...
```

Как видите, в этом классе не так уж и много нового. Указанные в нем операторы SQL будут использоваться в сценарии транзакций. Они составлены в формате, пригодном для вызова метода `prepare()` из класса PDO. Знаками вопросов обозначены места для подстановки значений, которые будут переданы методу `execute()`. А теперь определим первый метод, предназначенный для решения одной из конкретных задач на уровне логики приложения проектируемой системы:

```
// Листинг 12.44
// VenueManager
public function addVenue(string $name, array $spaces): array
{
    $pdo = $this->getPdo();
    $ret = [];
    $ret['venue'] = [$name];
    $stmt = $pdo->prepare($this->addvenue);
    $stmt->execute($ret['venue']);
    $vid = $pdo->lastInsertId();
    $ret['spaces'] = [];
    $stmt = $pdo->prepare($this->addspace);
    foreach ($spaces as $spacename)
    {
        $values = [$spacename, $vid];
        $stmt->execute($values);
        $sid = $pdo->lastInsertId();
        array_unshift($values, $sid);
        $ret['spaces'][] = $values;
    }
    return $ret;
}
```

Как видите, методу `addVenue()` передаются имя культурного заведения и массив наименования мест проведения культурных мероприятий.

Они используются в нем для заполнения таблицы культурных заведений (venue) и мест проведения культурных мероприятий (space). В этом методе создается также структура данных, предназначенная для хранения этих сведений вместе с вновь сформированными значениями идентификаторов (ID) каждой строки в упомянутых выше таблицах базы данных.

Следует, однако, иметь в виду, что если при выполнении операций в базе данных возникнет ошибка, то будет сгенерировано соответствующее исключение. В данном случае исключения не перехватываются, и поэтому любое исключение, сгенерированное в методе prepare(), будет сгенерировано и в данном методе. Именно такой результат здесь и нужен, хотя в документации к этому методу следует ясно указать, что в нем генерируются исключения.

Создав строку в таблице venue, мы обходим в цикле массив $space_array, вводя строку в таблицу space для каждого места проведения культурного мероприятия. Обратите внимание на то, что в каждую создаваемую строку в таблице space включается идентификатор культурного заведения в виде внешнего ключа. Подобным образом место проведения мероприятия связывается с конкретным культурным заведением.

Второй сценарий транзакций выглядит так же просто. Назначение этого сценария — ввести в таблицу events культурное мероприятие, связанное с местом его проведения:

```
// Листинг 12.45
// VenueManager
public function bookEvent(int $spaceid, string $name,
                          int $time, int $duration): void
{
    $pdo = $this->getPdo();
    $stmt = $pdo->prepare($this->addevent);
    $stmt->execute([$name, $spaceid, $time, $duration]);
}
```

Следствия

Шаблон Transaction Script является эффективным средством для быстрого получения результатов. Он относится к числу тех шаблонов, которыми многие программисты пользуются годами, даже не подозревая, что у него есть свое название. Используя несколько удобных вспомогательных методов наподобие тех, которые были добавлены здесь в базовый класс, можно сосредоточиться на логике приложения, не отвлекаясь на особенности взаимодействия с базой данных.

Мне приходилось наблюдать применение шаблона Transaction Script и в менее благоприятном контексте. В частности, я полагал, что разработал намного более сложное и насыщенное объектами приложение, чем те, для которых обычно подходит данный шаблон. Но когда приблизились сроки сдачи проекта, я обнаружил, что разместил слишком много логики там, где должен быть тонкий фасад модели предметной области в соответствии с шаблоном Domain Model, рассматриваемым в следующем разделе. И хотя результат получился менее изящным, чем хотелось бы, мне пришлось признать, что разработанное приложение не особенно пострадало от этой неявной доработки.

Как правило, шаблон Transaction Script следует применять в небольших проектах, если есть уверенность, что они не перерастут в нечто большее. Такой подход не вполне допускает масштабирование проектируемой системы, потому что дублирование кода обычно начинает проявляться, когда сценарии неизбежно пересекаются. Безусловно, можно реорганизовать код, но полностью избавиться от его дублирования вряд ли удастся.

В данном примере мы решили вставить код взаимодействия с базой данных непосредственно в классы сценариев транзакций. Но, как было показано выше, в таком коде необходимо отделить взаимодействие с базой данных от логики приложения. Такое разделение можно сделать окончательным, полностью изъяв код взаимодействия с базой данных из класса и создав промежуточный класс, назначение которого — управлять взаимодействием с базой данных от имени системы.

Шаблон Domain Model

Шаблон Domain Model (Модель предметной области) представляет собой чисто логический механизм, попытки создать, сформировать и защитить который предпринимаются во многих других шаблонах, описанных в этой главе. Это абстрактное представление сил, действующих в конкретном проекте. Это нечто вроде плоскости форм, на которой естественным образом разворачиваются решаемые бизнес-задачи, не обремененные такими неприятными материальными вопросами, как базы данных и веб-страницы.

Если приведенное выше описание кажется вам слишком витиеватым, обратимся к реальности. Шаблон Domain Model — это представление реальных участников проектируемой системы. Именно в шаблоне Domain Model объект практически проявляет свою истинную сущность в большей степени, чем где бы то ни было. Во всех других местах объекты стремятся

воплощать дополнительные обязанности, а в шаблоне Domain Model они, как правило, описывают набор свойств с дополнительными действующими силами. Здесь объект — это *нечто*, делающее *что-то* конкретное.

Проблема

Если вы пользовались шаблоном Transaction Script, то, вероятно, обнаружили, что дублирование кода становится трудноразрешимой задачей, когда в различных сценариях приходится выполнять одни и те же задачи. Эту задачу отчасти можно решить, вынеся повторяющиеся операции в отдельный модуль, но со временем такое решение все равно сводится к копированию и вставке кода.

Шаблон Domain Model можно применять для извлечения и объединения участников и процессов в проектируемой системе. Вместо того чтобы использовать сценарий для ввода сведений о месте проведения культурного мероприятия в базу данных, а затем связывать с ними сведения о событии, можно создать классы `Space` и `Event`. И тогда будет очень просто заказать культурное мероприятие в определенном месте, для чего достаточно вызвать метод `Space::bookEvent()`. Задача проверки временного конфликта превращается в вызов метода `Event::intersects()` и т.д.

Очевидно, что для такого простого примера, как проектирование рассматриваемой здесь системы Woo, шаблон Transaction Script будет более подходящим. Но по мере усложнения логики приложения более привлекательным становится альтернативный вариант: шаблон Domain Model. Если создать модель предметной области приложения, то справиться со сложной логикой будет проще, и для этого потребуется меньше кода с условными операторами.

Реализация

Проектировать предметные области в соответствии с шаблоном Domain Model относительно просто. Главная сложность здесь кроется в проектных шаблонах, которые должны сохранять чистоту модели, отделяя ее от других уровней в приложении.

Отделение участников модели предметной области в соответствии с шаблоном Domain Model от уровня представления связано главным образом с обеспечением полной изоляции. Отделить участников от уровня представления данных намного сложнее. И хотя в идеальном случае шаблон Domain Model следовало бы рассматривать только с точки зрения задач,

которые он представляет и решает, избежать реальных проблем, связанных с базой данных, будет очень трудно.

Как правило, классы, построенные в соответствии с шаблоном Domain Model, практически напрямую отображаются на таблицы реляционной базы данных, что существенно упрощает дело. В качестве примера на рис. 12.11 показана диаграмма классов, на которой отображены некоторые из участников рассматриваемой здесь системы Woo.

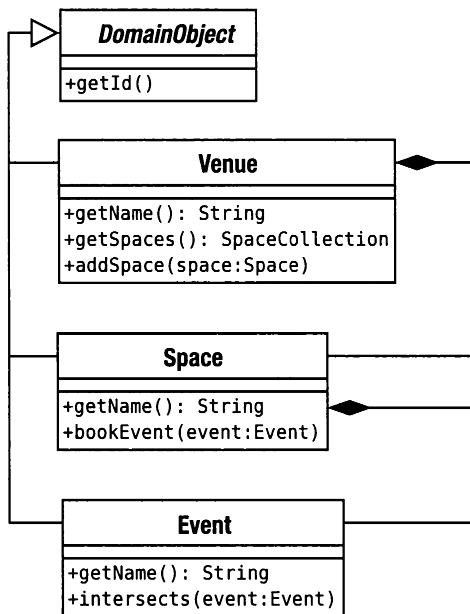


Рис. 12.11. *Фрагмент реализации шаблона Domain Model*

Объекты классов, приведенных на рис. 12.11, соответствуют таблицам, которые были созданы для рассмотренного ранее примера применения шаблона Transaction Script. Благодаря этой прямой связи управлять системой становится легче, хотя это не всегда возможно, особенно если приходится иметь дело со схемой базы данных, созданной еще до разработки приложения. Такая связь сама по себе может стать источником затруднений. Если вы не будете внимательны, то все ваши усилия будут сосредоточенны на моделировании базы данных, а не на решении насущных задач разработки приложения.

То, что шаблон Domain Model нередко отражает структуру базы данных, совсем не означает, что о ней должно быть хотя бы что-нибудь известно в классах, построенных по данному шаблону. Если отделить модель от базы

данных, то весь уровень будет легче протестировать и менее вероятно, что его затронут изменения, вносимые в схему базы данных или даже в весь механизм хранения данных. Это дает также возможность сосредоточить обязанности каждого класса на его основных задачах.

Ниже приведено определение упрощенного класса Venue вместе с его родительским классом:

```
// Листинг 12.46
abstract class DomainObject
{
    public function __construct(private int $id)
    {
    }
    public function getId(): int
    {
        return $this->id;
    }
    public static function getCollection(string $type): Collection
    {
        // Фиктивная реализация
        return Collection::getCollection($type);
    }
    public function markDirty(): void
    {
        // См. следующую главу!
    }
}

// Листинг 12.47
class Venue extends DomainObject
{
    private SpaceCollection $spaces;
    public function __construct(int $id, private string $name)
    {
        $this->name = $name;
        $this->spaces = self::getCollection(Space::class);
        parent:: __construct($id);
    }
    public function setSpaces(SpaceCollection $spaces): void
    {
        $this->spaces = $spaces;
    }
    public function getSpaces(): SpaceCollection
    {
        return $this->spaces;
    }
```

```
    public function addSpace(Space $space): void
    {
        $this->spaces->add($space);
        $space->setVenue($this);
    }
    public function setName(string $name): void
    {
        $this->name = $name;
        $this->markDirty();
    }
    public function getName(): string
    {
        return $this->name;
    }
}
```

В данном классе можно выделить несколько моментов, отличающих его от класса, предназначенного для выполнения без механизма сохраняемости. Вместо массива в данном классе применяется объект типа `SpaceCollection`, чтобы сохранять любые объекты типа `Space`, которые может содержать объект типа `Venue`. (Хотя можно было бы возразить, что типизированный массив дает несомненное преимущество независимо от того, работаете вы с базой данных или нет!) Данный класс оперирует специальным объектом коллекции, а не массивом объектов типа `Space`, и поэтому при обращении к нему в его конструкторе приходится получать экземпляры пустой коллекции. И с этой целью вызывается статический метод на уровне его супертипа:

```
$this->spaces = self::getCollection(Space::class);
```

Из этого метода в систему должны быть возвращены объекты коллекции, но о том, как получить их, речь пойдет в следующей главе. А до тех пор из суперкласса просто возвращается пустой массив.

На заметку В этой и следующей главах описываются изменения, которые требуется внести в объектах типа `Venue` и `Space`. Это простые объекты предметной области, разделяющие общие функциональные возможности. Если вы по ходу чтения данной книги занимаетесь программированием, вам не составит большого труда реализовать в обоих этих классах описываемые здесь принципы. Например, класс `Space` не обязан поддерживать коллекцию объектов типа `Space` (в нем могут точно таким же образом обрабатываться объекты типа `Event`).

Конструктору передается параметр `$id`, который мы передали суперклассу на хранение. Вряд ли вы будете удивлены, узнав, что параметр `$id` представляет однозначный идентификатор строки в таблице базы данных. Обратите также внимание на то, что из суперкласса вызывается метод `markDirty()`, который будет описан, когда речь пойдет о шаблоне Unit of Work.

Следствия

Сложность реализации шаблона Domain Model зависит от тех бизнес-процессов, которые требуется эмулировать. Его преимущество заключается в том, что вы можете сосредоточиться на решении конкретной задачи, проектируя модель и решая вопросы сохраняемости и представления на других уровнях — по крайней мере в теории.

На практике большинство разработчиков проектируют модели предметной области, ориентируясь на базу данных. Никто не хочет проектировать структуры, которые будут заставлять вас (или, еще хуже, ваших коллег) писать запутанный код, когда дело дойдет до ввода объектов в базу данных и их извлечения из нее.

За отделение модели предметной области от уровня представления данных с точки зрения проектирования и планирования приходится платить немалую цену. Код базы данных можно разместить непосредственно в модели предметной области, хотя вполне вероятно, что может возникнуть потребность спроектировать промежуточный класс для обработки реальных SQL-запросов. Для относительно простых моделей предметных областей, особенно для тех, в которых каждый класс явным образом отображается на таблицу базы данных, такой подход может быть очень выигрышным и позволит избежать значительного объема дополнительной работы по проектированию внешней системы для согласования объектов с базой данных.

Резюме

В этой главе изложено немало материала, хотя многое и было опущено для упрощения изложения. Пусть вас не пугают значительные объемы кода, включенные в материал этой главы. Проектные шаблоны следует применять тогда, когда это уместно, и сочетать их тогда, когда это полезно. Применяйте описанные в этой главе проектные шаблоны в случаях, если вы считаете, что они удовлетворяют потребностям вашего проекта, и

если вы не чувствуете, что придется создавать целый каркас еще до начала работы над проектом. С другой стороны, здесь содержится *достаточно* материала, чтобы сформировать основание такого каркаса или, что также вполне вероятно, дать представление об архитектуре некоторых готовых каркасов, которые можно выбрать для развертывания.

Есть и еще кое-что: я оставил вас в неустойчивом положении на краю сохраняемости, дав всего несколько провокационных намеков о коллекциях и преобразователях данных, чтобы немного вас подразнить. В следующей главе мы рассмотрим ряд шаблонов, предназначенных для работы с базами данных и обособления объектов от механизма хранения данных.

ГЛАВА 13

Шаблоны баз данных

В большинстве веб-приложений любой сложности в той или иной степени приходится иметь дело с *сохраняемостью* (или *персистентностью) данных*. В интернет-магазинах должна храниться информация о продаваемых товарах и их покупателях. В играх должны запоминаться игроки и их состояние. На сайтах социальных сетей должны отслеживаться 238 друзей пользователя и бессчетное количество его любимых групп 1980–1990-х годов. Каким бы ни было конкретное веб-приложение, вполне вероятно, что в нем незаметно ведется учет чего-нибудь. В этой главе мы рассмотрим ряд шаблонов, которые можно применять для решения упомянутых выше задач.

В этой главе будут рассмотрены следующие вопросы.

- *Интерфейс уровня хранения данных.* Шаблоны, которые определяют точки соприкосновения уровня хранения данных и остальной системы.
- *Наблюдение за объектами.* Отслеживание объектов, исключение дублирования, автоматизация операций сохранения и вставки.
- *Гибкие запросы.* Позволяют программистам клиентского кода создавать запросы, не задумываясь о формате базы данных, используемой в приложении.
- *Создание списка найденных объектов.* Создание коллекций, по которым можно осуществлять итерации.
- *Управление компонентами базы данных.* Долгожданное возвращение шаблона Abstract Factory.

Уровень хранения данных

В дискуссиях с клиентами обычно больше всего внимания уделяется уровню представления. Шрифты, цвета и простота использования — вот основные темы бесед с клиентами. Самой же распространенной темой в среде разработчиков является база данных, размер которой угрожающе растет. И беспокоит их не сама база данных, поскольку они уверены, что

она честно выполняет свои функции (кроме случаев ее неграмотного использования разработчиками). Разработчиков больше интересуют применяемые механизмы, которые преобразуют строки и столбцы таблицы базы данных в структуры данных и могут стать источниками серьезных проблем. В этой главе мы рассмотрим примеры кода, которые помогут в решении подобных проблем.

Не все, что описано далее в этой главе, относится непосредственно к самому уровню хранения данных. Здесь выделена часть шаблонов, которые помогают решать задачи сохраняемости. Все эти шаблоны описаны такими авторами, как Клифтон Нок (Clifton Nock), Мартин Фаулер (Martin Fowler), Дипак Алур (Deepak Alur) и др.

Шаблон Data Mapper

Если вы заметили, в разделе "Шаблон Domain Model" главы 12, "Шаблоны корпоративных приложений", мы обошли вниманием вопрос хранения и извлечения объектов типа `Venue` из базы данных. Теперь же пришло время дать ответы на некоторые вопросы взаимодействия с базами данных. Шаблон Data Mapper (Преобразователь данных) описан Дипаком Алуром в упоминавшейся ранее книге *Core J2EE Patterns: Best Practices and Design Strategies*[1] и Мартином Фаулером в еще одной упоминавшейся книге, *Patterns of Enterprise Application Architecture*[2], как Data Access Object (Объект доступа к данным). На самом деле наименование "Data Access Object" неточно описывает назначение шаблона Data Mapper, так как последний служит для формирования объектов передачи данных. Но поскольку подобные объекты выполняют на практике сходные задачи, шаблоны оказываются довольно схожими. Как и следовало ожидать, преобразователь данных в соответствии с шаблоном Data Mapper — это класс, который отвечает за управление передачей данных из базы данных в отдельный объект.

---

[1] Алур, Д., Крупи, Дж., Малкс, Д. *Образцы J2EE. Лучшие решения и стратегии проектирования* : Пер. с англ. — Изд-во "Лори", 2013.

[2] Фаулер, М. *Шаблоны корпоративных приложений* : Пер. с англ. — ИД "Вильямс", 2009.

Проблема

Объекты не организованы как таблицы в реляционной базе данных. Как известно, таблицы реляционной базы данных — это сеточные структуры, состоящие из строк и столбцов. Одна строка может связываться с другой строкой в другой (или даже в той же) таблице с помощью внешнего ключа. С другой стороны, объекты обычно связаны один с другим более естественным образом. Один объект может содержать ссылку на другой, а в различных структурах данных одни и те же объекты могут быть организованы разными способами. Их можно скомбинировать по-разному, и сформировать новые связи между ними можно прямо во время выполнения программы. Реляционные базы данных являются оптимальными для работы с большими объемами табличных данных, в то время как классы и объекты инкапсулируют небольшие специализированные фрагменты информации.

Такое отличие классов от реляционных баз данных нередко называется *объектно-реляционной потерей соответствия* (или просто потерей соответствия). Как же сделать такой переход, чтобы преодолеть подобное отличие? В качестве одного варианта можно создать класс (или ряд классов), отвечающий за решение только этой задачи, фактически скрыв базу данных от модели предметной области и сгладив неизбежные острые углы такого перехода.

Реализация

Приложив усилия и применив знания в программировании, можно создать один класс `Mapper`, предназначенный для обслуживания нескольких объектов. Но, как правило, в прикладном коде приходится наблюдать отдельный класс `Mapper`, созданный в качестве главного для модели предметной области в соответствии с шаблоном Domain Model.

На схеме, приведенной на рис. 13.1, показаны три конкретных класса `Mapper` и один абстрактный суперкласс.

На самом деле исходный код из класса `SpaceMapper` можно вынести в класс `VenueMapper`, поскольку объекты типа `Space` фактически являются подчиненными объектов типа `Venue`. Но ради упражнения мы будем пользоваться ими по отдельности в приведенных далее примерах кода.

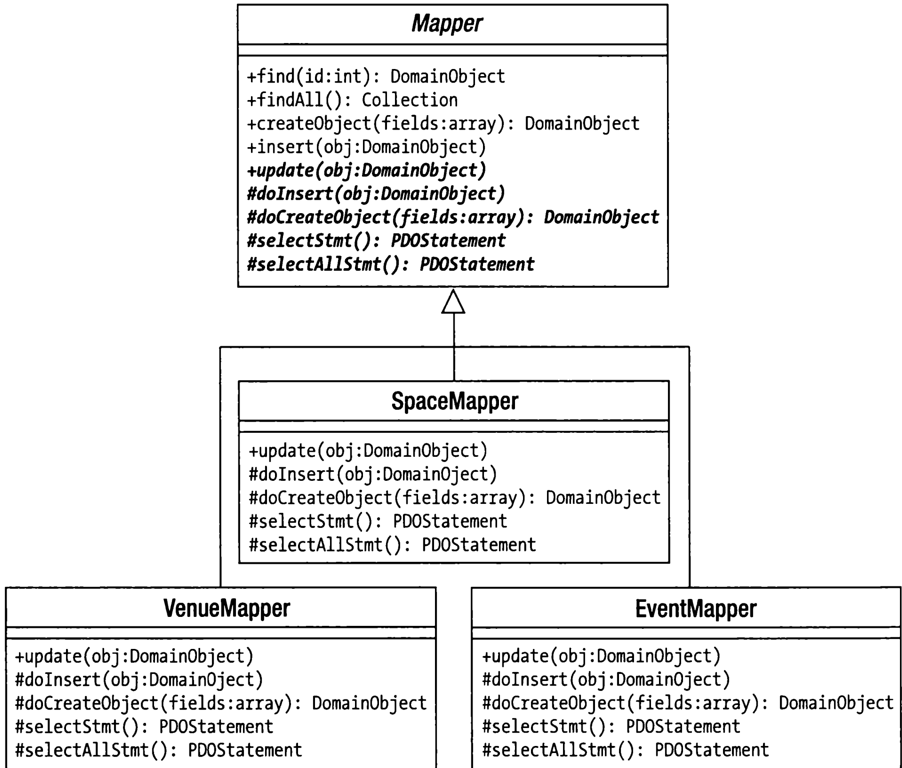


Рис. 13.1. *Классы* `Mapper`

Как видите, в этих классах представлены общие операции сохранения и загрузки данных. В базовом классе сосредоточены общие функциональные средства, а обязанности по выполнению операций над конкретными объектами делегируются дочерним классам. К этим операциям обычно относятся формирование реального объекта и составление запросов на выполнение операций в базе данных.

В базовом классе обычно выполняются вспомогательные операции до или после основной операции, поэтому для явного делегирования полномочий (например, для вызовов из таких конкретных методов, как `insert()`, в абстрактные методы, подобные `doInsert()`, и т.д.) применяется шаблон Template Method. Как будет показано ниже, реализация данного шаблона определяет, какие именно методы из базового класса сделаны конкретными подобным образом.

Ниже приведена упрощенная версия базового класса Mapper:

```
// Листинг 13.1
abstract class Mapper
{
    protected \PDO $pdo;
    public function __construct()
    {
        $reg = Registry::instance();
        $this->pdo = $reg->getPdo();
    }
    public function find(int $id): ? DomainObject
    {
        $this->selectstmt()->execute([$id]);
        $row = $this->selectstmt()->fetch();
        $this->selectstmt()->closeCursor();
        if (! is_array($row))
        {
            return null;
        }
        if (! isset($row['id']))
        {
            return null;
        }
        $object = $this->createObject($row);
        return $object;
    }
    public function createObject(array $raw): DomainObject
    {
        $obj = $this->doCreateObject($raw);
        return $obj;
    }
    public function insert(DomainObject $obj): void
    {
        $this->doInsert($obj);
    }
    abstract public function update(DomainObject $obj): void;
    abstract protected function doCreateObject(array $raw):
        DomainObject;
    abstract protected function doInsert(DomainObject $object): void;
    abstract protected function selectStmt(): \PDOStatement;
    abstract protected function targetClass(): string;
}
```

В методе-конструкторе применяется класс Registry для получения объекта типа PDO. Для подобных классов применять реестр на самом деле очень удобно, поскольку не всегда существует удобный путь от уровня

управления к объекту типа Mapper, по которому можно передать данные. Еще один способ создать объект типа Mapper состоит в том, чтобы поручить это классу Registry. Тогда конструктору класса Mapper, реализующего преобразователь данных, придется *передать* объект типа PDO в качестве аргумента, как показано ниже:

```
// Листинг 13.2
abstract class Mapper
{
    public function __construct(protected \PDO $pdo)
    {
    }
}
```

В клиентском коде новый объект типа VenueMapper получается из объекта типа Registry с помощью вызова такого метода, как getVenueMapper(). При этом создается экземпляр объекта типа Mapper, формирующий, в свою очередь, объект типа PDO. При последующих запросах этот метод возвратит ссылку на сохраняемый в кеше объект типа Mapper. Благодаря такому подходу удается избавить преобразователи данных от сложностей конфигурирования, даже если в качестве фасада используется реестр. Поэтому рекомендуется применять именно такой подход.

Обращаясь снова к классу Mapper, следует заметить, что метод insert() лишь делегирует свои полномочия методу doInsert(). В данном случае можно было бы отдать предпочтение абстрактному методу insert(), если бы не то обстоятельство, что применяемая здесь реализация окажется полезной в дальнейшем.

Метод find() отвечает за вызов подготовленного оператора, предоставляемого реализующим дочерним классом, а также за извлечение информации из строки базы данных. Завершая свою работу, он вызывает метод createObject(). Безусловно, подробности преобразования массива в объект будут меняться от случая к случаю, поэтому конкретная их реализация поручается абстрактному методу doCreateObject(). И в этом случае метод createObject(), по-видимому, лишь делегирует свои полномочия дочерней реализации. Но в дальнейшем мы добавим служебные операции, благодаря которым применение шаблона Template Method станет оправданным.

В дочерних классах будут также реализованы специальные методы обнаружения данных по заданным критериям (нам, например, нужно будет найти объекты типа Space, относящиеся к объектам типа Venue).

Этот процесс можно рассматривать и с точки зрения дочернего класса, как показано ниже:

```
// Листинг 13.3
class VenueMapper extends Mapper
{
    private \PDOStatement $selectStmt;
    private \PDOStatement $updateStmt;
    private \PDOStatement $insertStmt;
    public function __construct()
    {
        parent:: __construct();
        $this->selectStmt = $this->pdo->prepare(
                        "SELECT * FROM venue WHERE id=?");
        $this->updateStmt = $this->pdo->prepare(
                        "UPDATE venue SET name=?, id=? WHERE id=?");
        $this->insertStmt = $this->pdo->prepare(
                        "INSERT INTO venue ( name ) VALUES( ? )");
    }
    protected function targetClass(): string
    {
        return Venue::class;
    }
    public function getCollection(array $raw): VenueCollection
    {
        return new VenueCollection($raw, $this);
    }
    protected function doCreateObject(array $raw): Venue
    {
        $obj = new Venue(
            (int)$raw['id'],
            $raw['name']
        );
        return $obj;
    }
    protected function doInsert(DomainObject $obj): void
    {
        $values = [$obj->getName()];
        $this->insertStmt->execute($values);
        $id = $this->pdo->lastInsertId();
        $obj->setId((int)$id);
    }
    public function update(DomainObject $obj): void
    {
        $values = [
            $obj->getName(),
            $obj->getId(),
```

```
            $obj->getId()
        ];
        $this->updateStmt->execute($values);
    }
    public function selectStmt(): \PDOStatement
    {
        return $this->selectStmt;
    }
}
```

И в этом классе отсутствует ряд средств, которые могут понадобиться в дальнейшем, тем не менее он исправно делает свое дело. В конструкторе данного класса подготавливаются некоторые операторы SQL для последующего применения.

На заметку Обратите внимание, что `doCreateObject()` из `VenueMapper` объявляет возвращаемый тип `Venue`, а не `DomainObject`, как указано в родительском классе `Mapper`. То же самое верно и для `getCollection()`, который здесь объявляет возвращаемый тип `VenueCollection`, а не более общий тип `Collection`, указанный в классе `Mapper`. Это пример *ковариантности возвращаемого типа*, введенной в PHP 7.4, которая позволяет объявлять в дочерних классах более специализированные возвращаемые типы.

В родительском классе `Mapper` реализован метод `find()`, в котором вызывается метод `selectStmt()` с целью получить подготовленный оператор `SELECT` языка SQL. Если предположить, что все пройдет нормально, то в классе `Mapper` будет вызван метод `VenueMapper::doCreateObject()`. И здесь для формирования объекта типа `Venue` используется ассоциативный массив.

С точки зрения клиента данный процесс — сама простота:

```
// Листинг 13.4
$mapper = new VenueMapper();
$venue = $mapper->find(2);
print_r($venue);
```

Вызов функции `print_r()` позволяет быстро убедиться, что метод `find()` успешно выполнил свои обязанности. В рассматриваемой здесь системе, в таблице `venue` которой существует символьная строка с идентификатором `2`, выполнение приведенного выше фрагмента кода приведет к следующему результату:

```
popp\ch13\batch01\Venue Object
(
    [name:popp\ch13\batch01\Venue:private]      => The Likey Lounge
```

```
    [spaces:popp\ch13\batch01\Venue:private]    =>
    [id:popp\ch13\batch01\DomainObject:private] => 2
)
```

Методы `doInsert()` и `update()` выполняют действия, противоположные действиям, выполняемым методом `find()`. Каждому из них передается объект типа `DomainObject`, из которого извлекаются данные, которые должны быть занесены в строку таблицы базы данных, а затем вызывается метод `PDOStatement::execute()`. Обратите внимание на то, что в методе `doInsert()` запоминается идентификатор предоставленного объекта. Не следует, однако, забывать, что объекты в языке PHP передаются по ссылке, поэтому данное изменение будет обнаружено и в клиентском коде по его собственной ссылке.

Следует также заметить, что методы `doInsert()` и `update()` на самом деле не являются строго типизированными. Они примут любой подкласс, производный от класса `DomainObject`, безо всяких возражений.

Рассмотрим снова операции вставки и обновления данных с точки зрения клиента:

```
// Листинг 13.5
$mapper = new VenueMapper();
$venue = new Venue(-1, "The Likey Lounge");
// Добавление объекта в базу данных
$mapper->insert($venue);
// Еще раз ищем объект для проверки, что все работает
$venue = $mapper->find($venue->getId());
print_r($venue);
// Изменяем наш объект
$venue->setName("The Bibble Beer Likey Lounge");
// Вызов update для добавления измененных данных
$mapper->update($venue);
// Снова обращаемся к базе данных для проверки работоспособности
$venue = $mapper->find($venue->getId());
print_r($venue);
```

Почему я использовал отрицательное значение для идентификатора объекта `Venue`, который хочу добавить к базе данных? Как вы увидите позже в этой главе, я использую данное соглашение, чтобы отличать объекты, которым уже был назначен идентификатор базы данных, от объектов, его не имеющих. `VenueMapper::doInsert ()` не проверяет идентификатор — он просто использует имя для создания новой строки, а затем устанавливает сгенерированный идентификатор базы данных в предоставленном объекте `Venue`. Вот как выглядит метод `doInsert()`:

```
// Листинг 13.6
protected function doInsert(DomainObject $obj): void
{
    $values = [$obj->getName()];
    $this->insertStmt->execute($values);
    $id = $this->pdo->lastInsertId();
    $obj->setId((int)$id);
}
```

Обработка нескольких строк

Метод `find()` довольно прост, потому что ему необходимо возвратить только один объект. Но что делать, если из базы данных требуется извлечь много данных? Возможно, в поисках ответа на этот вопрос ваша первая мысль будет состоять в том, чтобы возвратить массив объектов. Можно, конечно, поступить и так, но у такого подхода имеется крупный недостаток.

Если возвращается массив объектов, то для каждого объекта из коллекции придется сначала получить экземпляр. Но если в итоге возвращается результирующее множество из тысячи объектов, то его обработка окажется чрезмерно затратной. В качестве альтернативы достаточно возвратить массив объектов, предоставив вызывающему коду возможность самому решать задачу инстанцирования объектов. И хотя это вполне допустимо, это нарушает само назначение классов типа `Mapper`.

Впрочем, есть способ сделать так, чтобы и волки были сыты, и овцы целы: воспользоваться встроенным интерфейсом `Iterator`.

Этот интерфейс требует, чтобы в реализующем его классе были определены методы для обработки списков. Удовлетворив данное требование, объект такого класса можно будет использовать в цикле `foreach` таким же образом, как и массив.

В табл. 13.1 перечислены методы, определяемые в интерфейсе `Iterator`.

Таблица 13.1. *Методы, определяемые в интерфейсе* `Iterator`

Имя	Описание
`rewind()`	Перемещает указатель в начало списка
`current()`	Возвращает элемент списка, находящийся в текущей позиции
`key()`	Возвращает текущий ключ (т.е. значение указателя)
`next()`	Перемещает указатель на следующую позицию списка
`valid()`	Подтверждает, что в текущей позиции имеется элемент списка

Чтобы реализовать интерфейс Iterator, необходимо реализовать его методы и отслеживать местоположение в наборе данных. А порядок получения данных, их сортировки или фильтрации нужно скрыть от клиента.

Ниже приведен пример реализации интерфейса Iterator, в котором массив заключен в оболочку реализующего класса, а его конструктору передается объект типа Mapper по причинам, которые станут понятными в дальнейшем:

```
// Листинг 13.7
abstract class Collection implements \Iterator
{
    protected int $total = 0;
    private int $pointer = 0;
    private array $objects = [];
    public function __construct(protected array $raw = [],
                                protected? Mapper $mapper = null)
    {
        $this->total = count($raw);
        if (count($raw) && is_null($mapper))
        {
            throw new
                AppException("Для генерации объекта нужен Mapper");
        }
    }
    public function add(DomainObject $object): void
    {
        $class = $this->targetClass();
        if (!($object instanceof $class))
        {
            throw new AppException("Это коллекция {$class}");
        }
        $this->notifyAccess();
        $this->objects[$this->total] = $object;
        $this->total++;
    }
    abstract public function targetClass(): string;
    protected function notifyAccess(): void
    {
        // Специально оставлен пустым!
    }
    private function getRow(int $num): ? DomainObject
    {
        $this->notifyAccess();
        if ($num >= $this->total || $num < 0)
        {
            return null;
        }
```

```
        if (isset($this->objects[$num]))
        {
            return $this->objects[$num];
        }
        if (isset($this->raw[$num]))
        {
            $this->objects[$num] = $
            this->mapper->createObject(
                $this->raw[$num]);
            return $this->objects[$num];
        }
        return null;
    }
    public function rewind(): void
    {
        $this->pointer = 0;
    }
    public function current(): ? DomainObject
    {
        return $this->getRow($this->pointer);
    }
    public function key(): mixed
    {
        return $this->pointer;
    }
    public function next(): void
    {
        $row = $this->getRow($this->pointer);
        if (! is_null($row))
        {
            $this->pointer++;
        }
    }
    public function valid(): bool
    {
        return (! is_null($this->current()));
    }
}
```

В конструкторе предполагается, что он будет вызываться без аргументов или с двумя аргументами (с исходными данными, которые получены из базы данных и в конечном итоге могут быть преобразованы в объекты, а также со ссылкой на объект типа `Mapper`, реализующий преобразователь данных). Допустим, клиент задал аргумент `$raw` (при этом нужно также указать объект типа `Mapper`). Он сохраняется в свойстве вместе с размером предоставленного набора данных. Если же предоставлены исходные

данные, то потребуется и экземпляр объекта типа Mapper, потому что именно он преобразовывает каждую строку из таблицы в объект. А если аргументы не были переданы конструктору данного класса, то он начнет свое функционирование с пустого массива (т.е. без исходных данных). Но обратите внимание на то, что для добавления данных в коллекцию имеется метод add().

В данном классе поддерживаются два массива: $objects и $raw. Если в клиентском коде требуется конкретный элемент, то в методе getRow() сначала выполняется поиск в массиве $objects, чтобы выяснить, не получен ли уже экземпляр такого элемента. Если экземпляр уже имеется, данный метод возвратит его. В противном случае метод getRow() продолжит поиск исходных данных в массиве $raw. Эти данные присутствуют в массиве $raw только в том случае, если присутствует и объект типа Mapper. Следовательно, данные для соответствующей строки таблицы могут быть переданы упоминавшемуся ранее методу Mapper::createObject(). Этот метод возвращает объект типа DomainObject, который сохраняется в массиве $objects по соответствующему индексу. В конечном итоге вновь созданный объект типа DomainObject возвращается пользователю.

В остальной части класса выполняется простая обработка свойства $pointer и вызывается метод getRow(). Он также включает метод notifyAccess(), который потребуется, когда дело дойдет до применения шаблона Lazy Load. Вы, вероятно, заметили, что класс Collection объявлен абстрактным, а следовательно, необходимо предоставить отдельные его реализации для каждого класса предметной области:

```
// Листинг 13.8
class VenueCollection extends Collection
{
    public function targetClass(): string
    {
        return Venue::class;
    }
}
```

Класс VenueCollection просто расширяет класс Collection; в нем реализован метод targetClass(). Благодаря этому, а также проверке типов в методе add() из суперкласса в коллекцию могут быть добавлены только объекты типа Venue. Ради еще большей безопасности можно также выполнить дополнительную проверку типов в конструкторе данного класса.

Обратите внимание, что я использовал синтаксис `::class`, чтобы получить полностью квалифицированное строковое представление класса. Эта возможность доступна только начиная с версии PHP 5.5. До этого мне пришлось бы быть осторожным и предоставить полный путь к пространству имен самостоятельно.

Очевидно, что данный класс должен действовать только вместе с классом `VenueMapper`. Но из практических соображений он реализует умеренно типизированную коллекцию, особенно если дело касается шаблона Domain Model.

Безусловно, для объектов типа `Event` и `Space` имеются параллельные классы. Некоторые классы типа `Collection` приведены на рис. 13.2.

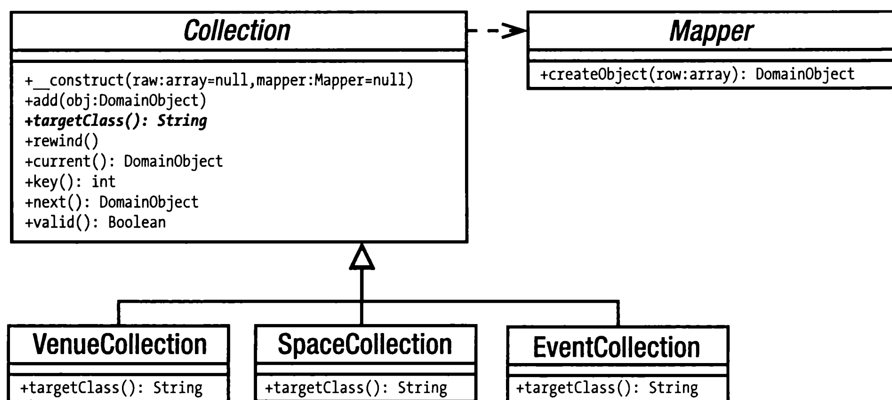


Рис. 13.2. *Классы для обработки множества строк таблицы с помощью коллекций*

Для получения пустых коллекций в классе `Registry` предусмотрены удобные соответствующие методы. Все дело в том, что в соответствии с шаблоном Domain Model необходимо получить экземпляры типа `Collection`, а в какой-то момент, возможно, потребуется сменить реализацию, особенно для целей тестирования. По мере расширения проектируемой системы эту обязанность можно поручить специально выделенной фабрике. Но на стадии разработки лучше выбрать простейший подход, применяя шаблон `Registry` для создания большинства объектов. В качестве примера ниже показано, как получить пустой объект типа `VenueCollection`:

```
// Листинг 13.9
$reg = Registry::instance();
$collection = $reg->getVenueCollection();
```

```
$collection->add(new Venue(-1, "Loud and Thumping"));
$collection->add(new Venue(-1, "Eeezy"));
$collection->add(new Venue(-1, "Duck and Badger"));

foreach ($collection as $venue)
{
    print $venue->getName() . "\n";
}
```

Здесь я опять использовал соглашение, согласно которому для объекта, который еще не был добавлен в базу данных, применяется идентификатор -1. Объект `Collection` не заботится о том, были ли уже добавлены его члены `DomainObject`.

Принимая во внимание осуществленную здесь реализацию, с такой коллекцией мало что можно еще сделать, хотя добавить методы `elementAt()`, `deleteAt()`, `count()` и другие аналогичные методы не составит большого труда. Оставляем это вам в качестве упражнения. (Выполните его и получите удовольствие!)

Применение генератора вместо интерфейса `Iterator`

Реализация интерфейса `Iterator` может оказаться довольно хлопотным, хотя и нетрудным делом. В версии PHP 5.5 появилась возможность использовать более простой (и зачастую более эффективный с точки зрения использования доступной памяти) механизм, называемый *генератором* (*generator*). По существу, генератор — это функция, которая может возвращать несколько значений, как правило, в цикле. В ней вместо ключевого слова `return` используется ключевое слово `yield`. Как только интерпретатор PHP обнаруживает в коде функции ключевое слово `yield`, он возвращает в вызывающий код итератор типа `Generator`, в котором реализован интерфейс `Iterator`. Над этим новым объектом можно выполнять те же самые действия, что и над любым другим итератором. Но хитрость состоит в том, что цикл, в котором для возврата нового значения используется ключевое слово `yield`, продолжит свое выполнение только в том случае, если объекту класса `Generator` поступит запрос на следующее значение из вызываемого метода `next()`. По сути дела, весь процесс выполнения кода выглядит следующим образом.

- В клиентском коде вызывается функция генератора, содержащая ключевое слово `yield`.

- В функции генератора запрограммирован цикл или какой-нибудь повторяющийся процесс, благодаря которому с помощью ключевого слова `yield` может возвращаться несколько значений. Как только интерпретатор PHP обнаружит в коде это ключевое слово, он создаст объект типа `Generator` и возвратит его клиентскому коду.
- На этом выполнение повторяющегося процесса в функции генератора приостанавливается.
- Полученный клиентским кодом объект типа `Generator` рассматривается как обычный итератор, которым можно оперировать в цикле (как правило, в цикле `foreach`).
- На каждом шаге цикла `foreach` объект типа `Generator` получает следующее значение из функции генератора.

Так почему бы нам не воспользоваться этой новой возможностью в упоминавшемся выше базовом классе `Collection`? Функция (точнее — метод) генератора возвращает объект типа `Generator`, поэтому в самом классе `Collection` больше не нужно поддерживать итеративную обработку. Вместо этого можно воспользоваться методом генератора как фабричным следующим образом:

```
// Листинг 13.10
abstract class GenCollection
{
    protected int $total = 0;
    private array $objects = [];
    public function __construct(protected array $raw = [],
                                protected ? Mapper $mapper = null)
    {
        $this->total = count($raw);

        if (count($raw) && is_null($mapper))
        {
            throw new
                AppException("Для генерации объекта нужен Mapper");
        }
    }
    public function add(DomainObject $object): void
    {
        $class = $this->targetClass();

        if (!($object instanceof $class))
```

```
        {
            throw new AppException("Это коллекция {$class}");
        }

        $this->notifyAccess();
        $this->objects[$this->total] = $object;
        $this->total++;
    }
    public function getGenerator(): \Generator
    {
        for ($x = 0; $x < $this->total; $x++)
        {
            yield $this->getRow($x);
        }
    }
    abstract public function targetClass(): string;
    protected function notifyAccess(): void
    {
        // Специально оставлен пустым!
    }
    private function getRow(int $num): ? DomainObject
    {
        $this->notifyAccess();
        if ($num >= $this->total || $num < 0)
        {
            return null;
        }
        if (isset($this->objects[$num]))
        {
            return $this->objects[$num];
        }
        if (isset($this->raw[$num]))
        {
            $this->objects[$num] = $this->mapper->createObject(
                $this->raw[$num]);
            return $this->objects[$num];
        }
        return null;
    }
}
```

Как видите, такое решение позволило создать более компактный базовый класс. Теперь отпала необходимость в реализации методов `current()`, `reset()` и им подобных. Единственный недостаток такого решения состоит в том, что сам класс `Collection` утратил свои итеративные возможности. Поэтому в клиентском коде придется вызвать функцию генерато-

ра getGenerator() и итеративно обработать возвращаемый оператором yield объект типа Generator, как показано ниже:

```
// Листинг 13.11
$genvencoll = new GenVenueCollection();
$genvencoll->add(new Venue(-1, "Loud and Thumping"));
$genvencoll->add(new Venue(-1, "Eeezy"));
$genvencoll->add(new Venue(-1, "Duck and Badger"));
$gen = $genvencoll->getGenerator();

foreach ($gen as $wrapper)
{
    print_r($wrapper);
}
```

Мы не собираемся внедрять эту новую возможность в рассматриваемую здесь систему и поэтому продолжим рассмотрение данного примера, применяя реализованную ранее версию итератора, определяемого в интерфейсе Iterator. Следует, однако, иметь в виду, что на самом деле генераторы предоставляют все средства, необходимые для того, чтобы создавать легковесные итераторы, прилагая минимум усилий для их установки.

Получение объектов типа Collection

Как говорилось ранее, мы решили воспользоваться классом Registry как фабрикой для коллекций. Но его можно использовать и для обслуживания объектов типа Mapper, как показано в следующем примере кода:

```
// Листинг 13.12
public function getVenueMapper(): VenueMapper
{
    return new VenueMapper();
}
public function getSpaceMapper(): SpaceMapper
{
    return new SpaceMapper();
}
public function getEventMapper(): EventMapper
{
    return new EventMapper();
}
public function getVenueCollection(): VenueCollection
{
    return new VenueCollection();
}
public function getSpaceCollection(): SpaceCollection
```

```
{
    return new SpaceCollection();
}
public function getEventCollection(): EventCollection
{
    return new EventCollection();
}
```

Таким образом, мы перешли границы того, что изящно вписывается в класс `Registry`. Стоило добавить ряд дополнительных методов получения, как возникла потребность реорганизовать этот код, чтобы воспользоваться шаблоном Abstract Factory. Но оставим это вам в качестве упражнения (за справкой обращайтесь, если потребуется, к главе 9, “Генерация объектов”).

Итак, упростив доступ к классам `Mapper` и `Collection`, можно расширить класс `Venue`, чтобы организовать сохраняемость объектов типа `Space`. В этом классе предоставляются методы для добавления отдельных объектов типа `Space` в коллекцию типа `SpaceCollection` или для создания совершенно новой коллекции того же самого типа:

```
// Листинг 13.13
// Venue
public function getSpaces(): SpaceCollection
{
    if (is_null($this->spaces))
    {
        $reg = Registry::instance();
        $this->spaces = $reg->getSpaceCollection();
    }
    return $this->spaces;
}
public function setSpaces(SpaceCollection $spaces): void
{
    $this->spaces = $spaces;
}
public function addSpace(Space $space): void
{
    $this->getSpaces()->add($space);
    $space->setVenue($this);
}
```

В данной реализации метода `setSpaces()` принимается на веру, что все объекты типа `Space` в коллекции ссылаются на текущий объект типа `Venue`. Добавить проверку в этот метод совсем не трудно, но ради простоты примера в данной версии мы не будем этого делать. Обратите внимание на то, что мы получаем экземпляр свойства `$spaces` только тогда, когда

вызывается метод getSpaces(). В дальнейшем будет показано, как расширить это отложенное получение экземпляров, чтобы ограничить запросы к базе данных.

В классе VenueMapper должна создаваться коллекция типа Space Collection для каждого создаваемого объекта типа Venue:

```
// Листинг 13.14
// VenueMapper
protected function doCreateObject(array $raw): Venue
{
    $obj = new Venue(
        (int)$raw['id'],
        $raw['name']
    );
    $spacemapper = new SpaceMapper();
    $spacecollection = $spacemapper->findByVenue($raw['id']);
    $obj->setSpaces($spacecollection);
    return $obj;
}
```

В методе VenueMapper::doCreateObject() создается объект типа SpaceMapper, из которого получается объект типа SpaceCollection. Как видите, в классе SpaceMapper реализован метод findByVenue(). Это приводит нас к запросам, по которым формируется несколько объектов. Ради краткости мы опустили метод Mapper::findAll() из первоначального листинга для класса woo\mapper\Mapper. В приведенном ниже листинге он присутствует снова:

```
// Листинг 13.15
// Mapper
public function findAll(): Collection
{
    $this->selectAllStmt()->execute([]);
    return $this->getCollection(
        $this->selectAllStmt()->fetchAll()
    );
}
abstract protected function selectAllStmt(): \PDOStatement;
abstract protected function getCollection(array $raw): Collection;
```

В данном методе вызывается дочерний метод selectAllStmt(). Как и метод selectStmt(), он должен содержать подготовленный оператор SQL, по которому из таблицы выбираются все строки. Ниже показано, каким образом объект типа PDOStatement создается в классе SpaceMapper:

```
// Листинг 13.16
// SpaceMapper::__construct()

$this->selectAllStmt = $this->pdo->prepare(
                            "SELECT * FROM space"
                        );
$this->findByVenueStmt = $this->pdo->prepare(
                            "SELECT * FROM space WHERE venue=?"
                        );
```

Здесь внедрен еще один оператор, `$findByVenueStmt`, предназначенный для выборки объектов типа `Space`, характерных для отдельного объекта типа `Venue`. В методе `findAll()` вызывается еще один новый метод `getCollection()`, которому передаются обнаруженные данные. Ниже приведено определение метода `SpaceMapper::getCollection()`:

```
// Листинг 13.17
public function getCollection(array $raw): SpaceCollection
{
    return new SpaceCollection($raw, $this);
}
```

В полной версии класса `Mapper` следует объявить методы `getCollection()` и `selectAllStmt()` абстрактными, чтобы все объекты типа `Mapper` могли возвращать коллекцию, содержащую их постоянно сохраняемые объекты предметной области. Но чтобы получить объекты типа `Space`, которые относятся к объекту типа `Venue`, потребуется более ограниченная коллекция. Ранее уже демонстрировался подготовленный оператор SQL для получения данных, а ниже приведен метод `SpaceMapper::findByVenue()`, в котором формируется коллекция:

```
// Листинг 13.18
public function findByVenue($vid): SpaceCollection
{
    $this->findByVenueStmt->execute([$vid]);
    return new SpaceCollection($this->findByVenueStmt->fetchAll(),
$this);
}
```

Метод `findByVenue()` идентичен методу `findAll()`, за исключением применяемой инструкции SQL. Если снова обратиться к классу `VenueMapper`, то полученная в итоге коллекция сохранится в объекте типа `Venue` с помощью метода `Venue::setSpaces()`. Поэтому объекты типа `Venue` теперь поступают непосредственно из базы данных со всеми при-

надлежащими им объектами типа Space лишь в строго типизированном списке. Ни один экземпляр объекта из этого списка не создается до того, как к нему будет сделан запрос.

На рис. 13.3 показан процесс, посредством которого клиентский класс может получить объект типа SpaceCollection, а также взаимодействие класса SpaceCollection с методом SpaceMapper::createObject() с целью преобразовать табличные данные в объект, возвращаемый клиенту.

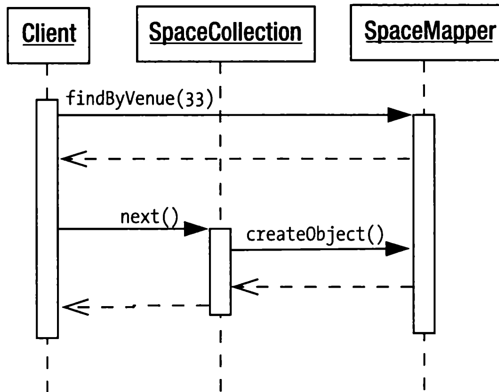


Рис. 13.3. *Получение коллекции типа* SpaceCollection, *а с ее помощью — объекта типа* Space

Следствия

Недостаток подхода, выбранного для добавления объектов типа Space в объект типа Venue, заключается в необходимости дважды обращаться к базе данных. Но, на мой взгляд, это справедливая цена, которую стоит заплатить. Следует также заметить, что действия, предпринимаемые в методе Venue::doCreateObject() для получения надлежащим образом заполненной коллекции типа SpaceCollection, можно перенести в метод Venue::getSpaces(), чтобы второе подключение к базе данных происходило только по требованию. Ниже показано, как определяется вариант такого метода:

```
// Листинг 13.19
// Venue
public function getSpaces2(): SpaceCollection
{
```

```
    if (is_null($this->spaces))
    {
        $reg = Registry::instance();
        $finder = $reg->getSpaceMapper();
        $this->spaces = $finder->findByVenue($this->getId());
    }

    return $this->spaces;
}
```

Но если встанет вопрос эффективности, то следует полностью реорганизовать исходный код класса `SpaceMapper` и получить все необходимые данные за один проход с помощью конструкции `JOIN` SQL-запроса. Безусловно, в результате такой реорганизации код может стать менее переносимым, но за оптимизацию ради эффективности всегда приходится платить!

Степень детализации классов `Mapper` будет, конечно, варьироваться. Если объект одного конкретного типа сохраняется только в объекте другого типа, то для контейнера можно предусмотреть только один преобразователь данных типа `Mapper`. Главное преимущество шаблона Data Mapper заключается в четком разделении уровня предметной области и базы данных. Объекты типа `Mapper` действуют "за кулисами" и могут приспосабливаться ко всем видам реляционных баз данных.

Вероятно, самым крупным недостатком данного шаблона является большой объем кропотливой работы по созданию конкретных классов `Mapper`. Но здесь обнаруживается немало стереотипного кода, который можно сгенерировать автоматически. Изящным средством генерации общих методов для классов типа `Mapper` служит рефлексия. В частности, можно сделать запрос объекта предметной области, исследовать его методы получения и установки (возможно, учитывая соглашение об именовании аргументов) и сгенерировать базовые классы типа `Mapper`, готовые для последующей коррекции. Именно так первоначально были созданы все классы типа `Mapper`, приведенные в этой главе.

Имея дело с классами типа `Mapper`, следует помнить об опасности загрузки слишком большого количества объектов одновременно. Но здесь на помощь может прийти реализация интерфейса `Iterator`. Сначала коллекция типа `Collection` содержит данные только из одной строки таблицы базы данных, и поэтому второй запрос (объекта типа `Space`) делается только при обращении к конкретному объекту типа `Venue` для последующего преобразования из массива в объект. Такую форму загрузки по требованию можно усовершенствовать еще больше, как будет показано далее.

Впрочем, волнообразная загрузка требует особого внимания. Создавая преобразователь данных типа Mapper, имейте в виду, что, используя другой преобразователь данных того же самого типа с целью получить значение свойства из объекта, вы затрагиваете лишь вершину большого айсберга. Ведь второй преобразователь данных может использоваться еще и для создания собственного объекта. И если вы проявите невнимательность, то в конечном итоге может оказаться, что простая на первый взгляд операция поиска повлечет за собой десятки других аналогичных операций.

Следует также принимать во внимание все инструкции по составлению эффективных запросов к базам данных и проявить готовность к их оптимизации для каждой базы данных, если в этом возникнет потребность. Операторы SQL, пригодные для запросов к разным базам данных, — это, конечно, хорошо, но еще лучше — быстродействующие приложения. Несмотря на то что внедрение условных операторов (или классов стратегий) для составления разных вариантов одних и тех же запросов — дело хлопотное и в первом случае дает неприглядные результаты, не следует забывать, что вся эта неприятная оптимизация скрыта от клиентского кода.

Шаблон Identity Map

Помните тот кошмар, который был связан с ошибками при передаче параметров по значению в версии PHP 4? Полная путаница, возникавшая в том случае, когда две переменные на первый взгляд указывали на один объект, а на самом деле оказывалось, что на разные, но коварно схожие объекты? К сожалению, этот кошмар возвратился.

Проблема

Рассмотрим вариант некоторого тестового кода, созданного для проверки примера применения шаблона Data Mapper:

```
// Листинг 13.20
$mapper = new VenueMapper();
$venue = new Venue(-1, "The Likey Lounge");
$mapper->insert($venue);

$venue1 = $mapper->find($venue->getId());
$venue2 = $mapper->find($venue->getId());

$venue1->setName("The Something Else");
```

```
$venue2->setName("The Bibble Beer Likey Lounge");

print $venue->getName() . "\n";
print $venue1->getName() . "\n";
print $venue2->getName() . "\n";
```

Назначение исходного варианта кода — продемонстрировать, что объект, добавленный в базу данных, может быть также извлечен с помощью объекта типа `Mapper` и окажется идентичным во всех отношениях, за исключением того, что это *не тот же самый* объект! Здесь я делаю проблему очевидной, работая с тремя версиями `Venue` — оригиналом и двумя экземплярами, извлеченными из базы данных. Я изменяю имена своих новых экземпляров и вывожу все три имени:

```
The Likey Lounge
The Something Else
The Bibble Beer Likey Lounge
```

Вспомните, что я использую соглашение, согласно которому совершенно новый объект `DomainObject` (т.е. тот, который отсутствует в базе данных) должен быть инстанцирован со значением идентификатора, равным `-1`. Благодаря методу `VenueMapper::insert()` мой исходный объект `Venue` будет обновлен с помощью значения, автоматически сгенерированного базой данных.

Ранее я обошел эту проблему, присваивая новый объект `Venue` поверх старого, так что у меня не получалось много клонообразных объектов. Но, к сожалению, контролировать ситуацию до такой степени можно далеко не всегда. Один и тот же объект может использоваться в разные моменты времени в пределах *одного* запроса. Если один его вариант изменяется и сохраняется в базе данных, то где гарантия, что в другом варианте этого объекта (вероятно, уже сохраненном в коллекции типа `Collection`) не будут перезаписаны внесенные ранее изменения?

Дубликаты объектов вызывают не только риск, но и непроизводительные издержки. Некоторые распространенные объекты могут загружаться по три-четыре раза в ходе выполнения процесса, и все эти обращения к базе данных, кроме одного, окажутся совершенно излишними. К счастью, решить эту проблему относительно просто.

Реализация

Identity Map — это просто объект, предназначенный для слежения за всеми остальными объектами в системе и помогающий исключить дублирование тех объектов, которые должны быть в единственном экземпляре:

```
// Листинг 13.21
class ObjectWatcher
{
    private array $all = [];
    private static ? ObjectWatcher $instance = null;
    private function __construct()
    {
    }
    public static function instance(): self
    {
        if (is_null(self::$instance))
        {
            self::$instance = new ObjectWatcher();
        }
        return self::$instance;
    }
    public function globalKey(DomainObject $obj): string
    {
        return get_class($obj) . "." . $obj->getId();
    }
    public static function add(DomainObject $obj): void
    {
        $inst = self::instance();
        $inst->all[$inst->globalKey($obj)] = $obj;
    }
    public static function exists(string $classname, int $id):
        ? DomainObject
    {
        $inst = self::instance();
        $key = "{$classname} . {$id}";
        if (isset($inst->all[$key]))
        {
            return $inst->all[$key];
        }
        return null;
    }
}
```

Как показано на рис. 13.4, класс, реализующий шаблон Identity Map, может сочетаться с другими рассмотренными ранее классами.

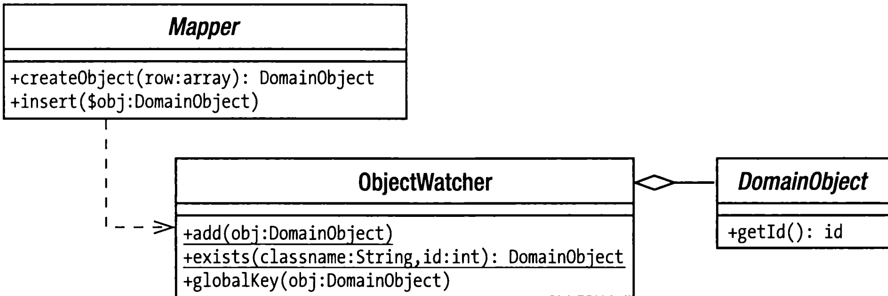


Рис. 13.4. *Сочетание класса, реализующего шаблон Identity Map, с другими классами*

Главная особенность шаблона Identity Map — это, естественно, идентификация объектов. Это означает, что каждый объект необходимо каким-то способом пометить. С этой целью можно воспользоваться самыми разными стратегиями. Ключ таблицы базы данных, который уже используется во всех объектах в системе, для этой цели не подходит, потому что нет никакой гарантии, что его идентификатор окажется уникальным для всех таблиц базы данных.

Можно также организовать отдельную базу данных и поддерживать в ней глобальную таблицу ключей. В таком случае, создавая всякий раз объект, следует обратиться к таблице ключей и связать глобальный ключ с объектом, находящимся в отдельной строке таблицы. Издержки в данном случае невелики и необременительны.

Как видите, здесь выбран более простой подход. В частности, имя класса дополняется его табличным идентификатором. Не может быть двух объектов типа `popp\ch13\batch03\Event` с идентификатором `4`, поэтому выбранный ключ `popp\ch13\batch03\Event.4` вполне подходит для наших целей.

Подробности реализации такого подхода вынесены в отдельный метод `globalKey()` класса `ObjectWatcher`. В этом классе предусмотрен также метод `add()` для добавления новых объектов. При этом каждый объект помечается уникальным ключом в массиве свойств `$all`.

Методу `exists()` передаются имя класса и идентификатор `$id`, а не сам объект. Чтобы узнать, существует ли объект, совсем необязательно получать его экземпляр! На основании этих данных в методе `exists()` создается ключ и проверяется, существует ли такой индекс в массиве свойств `$all`. Если объект найден, то возвращается ссылка, как и требуется.

Имеетсятолькоодинкласс,вкоторомприменяетсякласс`ObjectWatcher`, реализующий шаблон Identity Map. Это класс `Mapper`, предоставляющий функциональные возможности для формирования объектов, поэтому в него имеет смысл добавить проверку следующим образом:

```
// Листинг 13.22
// Mapper
public function find(int $id): ? DomainObject
{
    $old = $this->getFromMap($id);
    if (! is_null($old))
    {
        return $old;
    }
    // Работа с базой данных
    return $object;
}
abstract protected function targetClass(): string;
private function getFromMap($id): ? DomainObject
{
    return ObjectWatcher::exists(
        $this->targetClass(),
        $id
    );
}
private function addToMap(DomainObject $obj): void
{
    ObjectWatcher::add($obj);
}
public function createObject($raw): ? DomainObject
{
    $old = $this->getFromMap((int)$raw['id']);
    if (! is_null($old))
    {
        return $old;
    }
    $obj = $this->doCreateObject($raw);
    $this->addToMap($obj);
    return $obj;
}
public function insert(DomainObject $obj): void
{
    $this->doInsert($obj);
    $this->addToMap($obj);
}
```

В данном классе предусмотрены два удобных метода — addToMap() и getFromMap(). Это дает возможность не запоминать полный синтаксис статического обращения к классу ObjectWatcher. И, что еще важнее, в этих методах происходит обращение к дочерним реализациям (VenueMapper и т.д.) с целью получить имя класса, который в настоящий момент ожидает получения экземпляра.

Это достигается путем вызова абстрактного метода targetClass(), который реализуется во всех конкретных классах типа Mapper. Этот метод должен возвратить имя класса, объект которого должен сформировать преобразователь данных типа Mapper. Ниже приведена реализация метода targetClass() в классе SpaceMapper:

```
// Листинг 13.23
// SpaceMapper
protected function targetClass(): string
{
    return Space::class;
}
```

В методах find() и createObject() сначала проверяется, существует ли объект, путем передачи идентификатора таблицы методу getFromMap(). Если объект найден, он возвращается клиенту, и выполнение метода завершается. Но если никаких версий искомого объекта не существует, то получение его экземпляра продолжится. В методе createObject() этот новый объект передается методу addToMap(), чтобы предотвратить впоследствии любые конфликты.

Но зачем часть рассматриваемого здесь процесса приходится повторять дважды, вызывая метод getFromMap() в методах find() и createObject()? Ответ на этот вопрос связан с коллекциями типа Collection. Когда в них формируются объекты, с этой целью в них вызывается метод createObject(). При этом необходимо убедиться, что строка таблицы, инкапсулированная в объекте типа Collection, не устарела, и обеспечить возврат пользователю последней версии объекта.

Следствия

До тех пор, пока шаблон Identity Map применяется во всех контекстах, в которых объекты формируются из базы данных или добавляются в нее, вероятность дублирования объектов в ходе процесса практически равна нулю.

Разумеется, это правило действует только *в самом* процессе. В разных процессах будет неизбежно происходить одновременное обращение к разным версиям одного и того же объекта. И в этом случае важно учитывать возможности искажения данных в результате параллельного (т.е. одновременного) доступа к ним. Если это вызывает серьезные проблемы в вашей системе, то рекомендуется реализовать стратегию блокировки. Можно также рассмотреть возможность сохранения объектов в общей памяти или применения внешней системы кеширования объектов вроде `Memcached`. Подробнее о системе `Memcached` можно узнать по адресу `http://memcached.org/`, а об ее поддержке в PHP — по адресу `http://www.php.net/memcache`.

Шаблон Unit of Work

В какой момент вы сохраняете объекты? До тех пор, пока я не открыл для себя шаблон Unit of Work (Единица работы), описанный Дэвидом Райсом в упоминавшейся ранее книге Мартина Фаулера *Patterns of Enterprise Application Architecture*, по завершении команды я посылал запросы на сохранение из уровня представления. Но такое проектное решение оказалось довольно затратным. Шаблон Unit of Work помогает сохранять только те объекты, которые действительно нужно сохранить.

Проблема

Однажды я вывел набор своих операторов SQL в окне браузера и был поражен. Я обнаружил, что сохранял одни и те же данные снова и снова в одном и том же запросе. У меня была замечательная система составных команд, т.е. одна команда могла запустить несколько других, и каждая выполняла очистку после себя.

Я не только сохранял один и тот же объект дважды; я сохранял еще и объекты, которые вообще не нужно было сохранять.

В каком-то отношении данная задача аналогична той, которую решает шаблон Identity Map. Если этот шаблон позволяет решить задачу излишней загрузки объектов, то в данном случае задача находится на другом конце процесса. Обе эти задачи оказываются взаимодополняющими, как, впрочем, и их решения.

Реализация

Чтобы выяснять, какие именно необходимы операции с базой данных, придется отслеживать различные события, которые происходят в объектах. И самым лучшим местом, где это можно сделать, вероятно, являются сами объекты.

Необходимо также вести список объектов, составленный для каждой операции с базой данных (например, вставки, обновления, удаления). Рассмотрим пока что лишь операции вставки и обновления. Где лучше хранить список объектов? У нас уже есть объект класса `ObjectWatcher`, поэтому мы можем продолжить его разработку следующим образом:

```
// Листинг 13.24
// ObjectWatcher
private array $all = [];
private array $dirty = [];
private array $new = [];
private array $delete = []; // В данном примере не используется
private static ? ObjectWatcher $instance = null;
public static function addDelete(DomainObject $obj): void
{
    $inst = self::instance();
    $inst->delete[$inst->globalKey($obj)] = $obj;
}
public static function addDirty(DomainObject $obj): void
{
    $inst = self::instance();
    if (! in_array($obj, $inst->new, true))
    {
        $inst->dirty[$inst->globalKey($obj)] = $obj;
    }
}
public static function addNew(DomainObject $obj): void
{
    $inst = self::instance();
    // Пока что у нас нет id
    $inst->new[] = $obj;
}
public static function addClean(DomainObject $obj): void
{
    $inst = self::instance();
    unset($inst->delete[$inst->globalKey($obj)]);
    unset($inst->dirty[$inst->globalKey($obj)]);
    $inst->new = array_filter(
```

```
            $inst->new,
            function($a) use($obj)
            {
                return !($a === $obj);
            }
        );
    }
    public function performOperations(): void
    {
        foreach ($this->dirty as $key => $obj)
        {
            $obj->getFinder()->update($obj);
        }
        foreach ($this->new as $key => $obj)
        {
            $obj->getFinder()->insert($obj);
            print "Вставка " . $obj->getName() . "\n";
        }
        $this->dirty = [];
        $this->new = [];
    }
```

Класс `ObjectWatcher` по-прежнему соответствует шаблону Identity Map и продолжает выполнять свою функцию отслеживания всех объектов в системе с помощью свойства `$all`. В данном примере мы просто добавили в этот класс дополнительные функциональные возможности.

Отдельные аспекты реализации шаблона Unit of Work в классе `ObjectWatcher` показаны на рис. 13.5.

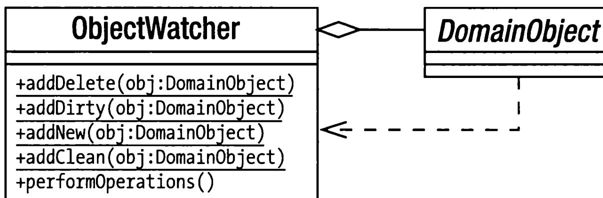


Рис. 13.5. *Аспекты шаблона Unit of Work в классе* `ObjectWatcher`

Объекты помечаются как "измененные", если они были модифицированы после извлечения из базы данных. Измененный объект сохраняется с помощью метода `addDirty()` в массиве свойств `$dirty` до тех пор, пока не придет время обновить базу данных. В клиентском коде может быть по собственным причинам решено, что измененный объект не должен подвергаться обновлению. С этой целью измененный объект может быть поме-

чен в клиентском коде как "сохраненный" с помощью метода addClean(). Как и следовало ожидать, вновь созданный объект должен быть добавлен в массив $new с помощью метода addNew(). Объекты добавляются из этого массива в базу данных по намеченному плану. В рассматриваемых здесь примерах функциональные средства удаления не реализуются, но общий принцип должен быть понятен.

В каждом из методов addDirty() и addNew() объект добавляется в массивы, хранящиеся в соответствующих свойствах. Но в методе addClean() заданный объект удаляется из массива $dirty, что помечает его как больше не ожидающий обновления.

Когда, наконец, приходит время обработать все объекты, сохраненные в этих массивах, должен быть вызван метод performOperations() (вероятно, из класса контроллера или его вспомогательного класса). Этот метод обходит в цикле массивы $dirty и $new, обновляя или добавляя объекты.

Класс ObjectWatcher теперь обеспечивает механизм обновления и вставки объектов. Но клиентскому коду все еще недостает средства добавления объектов в объект типа ObjectWatcher. А поскольку оперировать приходится именно этими объектами, сами они, вероятно, служат наилучшим местом для уведомления об этом. Ниже приведен ряд служебных методов, которые можно добавить в класс DomainObject. Обратите особое внимание на метод конструктора:

```
// Листинг 13.25
abstract class DomainObject
{
    public function __construct(private int $id = -1)
    {
        if ($id < 0)
        {
            $this->markNew();
        }
    }
    abstract public function getFinder(): Mapper;
    public function getId(): int
    {
        return $this->id;
    }
    public function setId(int $id): void
    {
        $this->id = $id;
    }
```

```
    public function markNew(): void
    {
        ObjectWatcher::addNew($this);
    }
    public function markDeleted(): void
    {
        ObjectWatcher::addDelete($this);
    }
    public function markDirty(): void
    {
        ObjectWatcher::addDirty($this);
    }
    public function markClean(): void
    {
        ObjectWatcher::addClean($this);
    }
}
```

Как видите, метод конструктора помечает текущий объект как новый, вызывая `markNew()`, если ему не было передано свойство `$id`.

На заметку Вспомните наше соглашение о том, что не вставленная в базу данных строка имеет идентификатор -1. Это позволяет нам всегда запрашивать целочисленное значение, а затем проверять, больше ли оно нуля, чтобы определить, должны ли данные строки считаться новыми. Конечно, вместо этого вы можете использовать для новых данных нулевое значение и изменить сигнатуру конструктора в `DomainObject` на `private ?int $id = null`.

Это похоже на своего рода волшебство, и здесь следует проявить внимательность. При таком виде данного кода новый объект для вставки в базу данных назначается безо всякого вмешательства со стороны создателя объектов. Представьте, что новый программист в вашей команде пишет сценарий для временного использования, чтобы протестировать работу программы, решающей поставленную задачу. В этом сценарии нет никаких признаков кода сохраняемости, поэтому все должно быть достаточно безопасно, не так ли? А теперь представьте эти тестовые объекты с интересными временными именами, которые попадают в постоянное хранилище! Волшебство — это, конечно, хорошо, но ясность — намного лучше. От клиентского кода, возможно, лучше потребовать, чтобы он передавал какой-нибудь флаг конструктору с целью поставить новый объект в очередь на добавление в базу данных.

В класс `Mapper` необходимо также добавить следующий фрагмент кода:

```
// Листинг 13.26
// Mapper
public function createObject($raw): DomainObject
{
    $old = $this->getFromMap($raw['id']);

    if (! is_null($old))
    {
        return $old;
    }

    $obj = $this->doCreateObject($raw);
    $this->addToMap($obj);
    return $obj;
}
```

Остается лишь добавить в методы классов, созданных в соответствии с шаблоном Domain Model, вызовы `markDirty()`. Напомним, что "измененным" считается объект, который был модифицирован после извлечения из базы данных. Именно этот аспект рассматриваемого здесь шаблона подозрительно напоминает шаблон Domain Model. Очевидно, важно гарантировать, чтобы все методы, которые модифицируют состояние объекта, помечали его как "измененный". Но поскольку это придется делать вручную, то вероятность ошибки вполне реальна, ведь человеку свойственно ошибаться.

Ниже приведены некоторые методы из класса `Space`, в которых вызывается `markDirty()`:

```
// Листинг 13.27
// Space
public function setVenue(Venue $venue): void
{
    $this->venue = $venue;
    $this->markDirty();
}
public function setName(string $name): void
{
    $this->name = $name;
    $this->markDirty();
}
```

Ниже приведен код для добавления новых объектов типа `Venue` и `Space` в базу данных, взятый из класса `Command`:

```
// Листинг 13.28
// Значение идентификатора -1 представляет новый Venue или Space
$venue = new Venue(-1, "The Green Trees");
```

```
$venue->addSpace(
    new Space(-1, 'The Space Upstairs')
);
$venue->addSpace(
    new Space(-1, 'The Bar Stage')
);

// Этот метод может быть вызван из контроллера
// или вспомогательного класса
ObjectWatcher::instance()->performOperations();
```

Мы добавили отладочный код в класс ObjectWatcher, что дает возможность увидеть, что происходит по окончании запроса:

```
Вставка The Green Trees
Вставка The Space Upstairs
Вставка The Bar Stage
```

Поскольку объекты Venue и Space были созданы с идентификаторами -1, они рассматриваются DomainObject как новые объекты. Внутренне в каждом случае конструктор объекта домена вызывает DomainObject::markNew(), который затем вызывает ObjectWatcher::addNew(). Когда в конечном итоге вызывается ObjectWatcher::performOperations(), эти объекты вносятся (а не обновляются) в базу данных, и срабатывает отладочный вывод.

Контроллер высокого уровня обычно вызывает метод performOperations(), и поэтому, как правило, достаточно создать или модифицировать объект, а класс (в данном случае — ObjectWatcher), созданный в соответствии с шаблоном Unit of Work, выполнит свои обязанности только один раз в конце запроса.

Следствия

Шаблон Unit of Work очень полезен, но имеется ряд моментов, о которых необходимо помнить. Применяя данный шаблон, следует быть уверенным, что во всех изменяющих объект операциях модифицированный объект помечается как "измененный". Невыполнение этого правила приведет к появлению ошибок, обнаружить которые будет очень трудно.

Возможно, придется поискать и другие способы тестирования измененных объектов. По-видимому, для этой цели подойдет рефлексия, но тогда необходимо оценить влияние такого тестирования на производительность. Ведь шаблон должен улучшать, а не ухудшать производительность.

Шаблон Lazy Load

Шаблон Lazy Load относится к числу основных шаблонов, к которым большинство разработчиков веб-приложений очень быстро приходят самостоятельно. Причина проста и состоит в том, что это важнейший механизм, позволяющий избегать массовых обращений к базе данных, т.е. делает именно то, к чему мы так стремимся.

Проблема

В примере, который рассматривается в этой главе, мы установили связь между культурными заведениями, местами проведения и культурными мероприятиями, т.е. объектами типа Venue, Space и Event. Когда создается объект типа Venue, автоматически получается коллекция типа SpaceCollection. Если бы нужно было вывести все объекты типа Space в данном объекте типа Venue, такое действие автоматически вызвало бы запрос к базе данных для получения всех объектов типа Event, связанных с каждым объектом типа Space. Они сохраняются в коллекции типа EventCollection. Если же просматривать события не требуется, то придется сделать несколько обращений к базе данных безо всякой причины. При наличии многих культурных заведений, у каждого из которых имеется два или три места проведения десятков или даже сотен культурных мероприятий, такой процесс получается слишком затратным.

Очевидно, в некоторых случаях придется прекращать подобное автоматическое включение коллекций. В качестве примера ниже приведен код, добавленный в класс SpaceMapper для получения данных типа Event:

```
// Листинг 13.29
// SpaceMapper
protected function doCreateObject(array $raw): Space
{
    $obj = new Space((int)$raw['id'], $raw['name']);
    $venmapper = new VenueMapper();
    $venue = $venmapper->find((int)$raw['venue']);
    $obj->setVenue($venue);
    $eventmapper = new EventMapper();
    $eventcollection = $eventmapper->findBySpaceId((int)$raw['id']);
    $obj->setEvents($eventcollection);
    return $obj;
}
```

Сначала в методе `doCreateObject()` получается объект типа `Venue`, с которым связан объект типа `Space`. Это не слишком затратная операция, потому что данный объект, вероятнее всего, уже сохранен в объекте типа `ObjectWatcher`. Затем в этом методе вызывается метод `EventMapper::findBySpaceId()`. Именно здесь и могут возникнуть осложнения в системе.

Реализация

Как вам, вероятно, уже известно, применить шаблон Lazy Load значит отложить получение значения свойства до тех пор, пока клиент его действительно не запросит. Как было показано ранее, самый простой способ добиться этой цели — сделать отсрочку в содержащем объекте явной. Ниже показано, как это можно сделать в классе `Space`:

```
// Листинг 13.30
// Space
public function getEvents2(): EventCollection
{
    if (is_null($this->events))
    {
        $reg = Registry::instance();
        $eventmapper = $reg->getEventMapper();
        $this->events = $eventmapper->findBySpaceId($this->getId());
    }

    return $this->events;
}
```

В данном методе проверяется, установлено ли свойство `$events`. Если оно не установлено, то запрашивается средство поиска (объект типа `Mapper`) и используется его собственное свойство `$id` для получения коллекции типа `EventCollection`, с которой оно связано. Очевидно, чтобы избавиться в этом методе от ненужных запросов к базе данных, придется внести коррективы в код класса `SpaceMapper`, чтобы он не выполнял автоматически предварительную загрузку объекта типа `EventCollection`, как этот делалось в предыдущем примере!

Это вполне работоспособный, хотя и немного беспорядочный подход. Не лучше ли избавиться от подобной беспорядочности? В конечном итоге мы возвращаемся к реализации интерфейса `Iterator`, в котором создается объект типа `Collection`. Мы уже скрыли за этим интерфейсом тот секретный факт, что исходные данные, извлекаемые из базы данных, могли

быть еще не использованы для получения экземпляра объекта предметной области в момент обращения к нему из клиентского кода. Но мы, вероятно, можем скрыть за этим интерфейсом еще больше.

Идея состоит в том, чтобы создать объект-коллекцию типа `EventCollection`, в котором обращение к базе данных откладывается до тех пор, пока не будет сделан запрос на него. Это означает, что клиентскому объекту (например, типа `Space`) вообще не должно быть известно, что он содержит пустую коллекцию типа `Collection` при своем первоначальном создании. После того как клиент начнет работу с данными, он будет содержать совершенно нормальную коллекцию типа `EventCollection`.

Ниже приведено определение класса `DeferredEventCollection`:

```
// Листинг 13.31
class DeferredEventCollection extends EventCollection
{
    private bool $run = false;
    public function __construct(
        Mapper $mapper,
        private \PDOStatement $stmt,
        private array $valueArray
    )
    {
        parent:: construct([], $mapper);
    }
    protected function notifyAccess(): void
    {
        if (! $this->run)
        {
            $this->stmt->execute($this->valueArray);
            $this->raw = $this->stmt->fetchAll();
            $this->total = count($this->raw);
        }

        $this->run = true;
    }
}
```

Как видите, этот класс расширяет стандартный класс `EventCollection`. Его конструктору требуются объекты типа `EventMapper` и `PDOStatement`, а также массив элементов, который должен соответствовать подготовленному оператору SQL. В первом случае класс ничего не делает, а только сохраняет свои свойства и ждет. Никаких запросов к базе данных не делается.

Следует иметь в виду, что в базовом классе Collection определяется пустой метод notifyAccess(), о котором упоминалось выше, в разделе “Шаблон Data Mapper”. Он вызывается из любого вызываемого извне метода.

В классе DeferredEventCollection этот метод переопределяется. Если теперь кто-нибудь попытается обратиться к коллекции типа Collection, в этом классе станет известно, что пришло время перестать притворяться, будто он делает что-то полезное, и получить, наконец, какие-нибудь реальные данные. С этой целью в нем вызывается метод PDOStatement::execute(). Вместе с вызовом метода PDOStatement::fetch() это позволяет получить массив полей, пригодных для передачи вызываемому методу Mapper::createObject().

Ниже приведено определение метода в классе EventMapper, в котором создается экземпляр класса DeferredEventCollection:

```
// Листинг 13.32
// EventMapper
public function findBySpaceId(int $sid): DeferredEventCollection
{
    return new DeferredEventCollection(
        $this,
        $this->selectBySpaceStmt,
        [$sid]
    );
}
```

Следствия

Загрузка по требованию — это привычка, которую стоит приобрести независимо от того, внедряете ли вы явным образом логику отложенной загрузки в классы предметной области. Помимо типовой безопасности, особое преимущество от применения для свойств не массива, а коллекции заключается в возможности откорректировать процесс загрузки по требованию, если в этом возникнет потребность.

Шаблон Domain Object Factory

Шаблон Data Mapper изящен, но у него есть некоторые недостатки. В частности, класс Mapper берет на себя слишком много обязанностей. В нем составляются операторы SQL, массивы преобразуются в объекты, а последние — обратно в массивы, готовые для добавления в базу данных.

Такая разносторонность делает класс Mapper удобным и эффективным, но в то же время она способна в какой-то степени уменьшить гибкость. Это особенно справедливо в том случае, если преобразователь данных типа Mapper должен обрабатывать самые разные виды запросов или если другие классы должны совместно использовать общие функциональные средства вместе с классом Mapper.

В оставшейся части главы будет показано, как разделить шаблон Data Mapper на ряд более специализированных шаблонов. При объединении этих мелкоструктурных шаблонов воспроизводятся все обязанности, которые выполнялись в шаблоне Data Mapper, причем некоторые из них (если не все!) можно использовать вместе с этим шаблоном. Эти шаблоны хорошо описаны Клифтоном Ноком (Clifton Nok) в книге *Data Access Patterns* (Addison Wesley, 2003), из которой взяты их названия там, где имеются разночтения. Итак, начнем с основной задачи — формирования объектов предметной области.

Проблема

Мы уже сталкивались с ситуацией, когда в классе Mapper проявляется естественное ограничение. Безусловно, метод createObject() используется внутренним образом в классе Mapper, но он требуется и коллекции типа Collection для создания объектов предметной области по запросу. Для этого необходимо передать ссылку на объект типа Mapper при создании коллекции типа Collection. И хотя нет ничего дурного в том, чтобы разрешить обратные вызовы (как это делалось в шаблонах Visitor и Observer), обязанность создавать объект предметной области лучше поручить его собственному типу. И тогда он может совместно использоваться в таких классах, как Mapper, Collection и им подобных. Шаблон Domain Object Factory (Фабрика объектов предметной области) описан в упоминавшейся выше книге *Data Access Patterns*.

Реализация

Допустим, имеется ряд классов типа Mapper, организованный, в общем, таким образом, чтобы каждый из них был нацелен на свой объект предметной области. Шаблон Domain Object Factory просто требует извлечь метод createObject() из каждого класса типа Mapper и разместить его в отдельном классе из параллельной иерархии. Эти новые классы показаны на рис. 13.6.

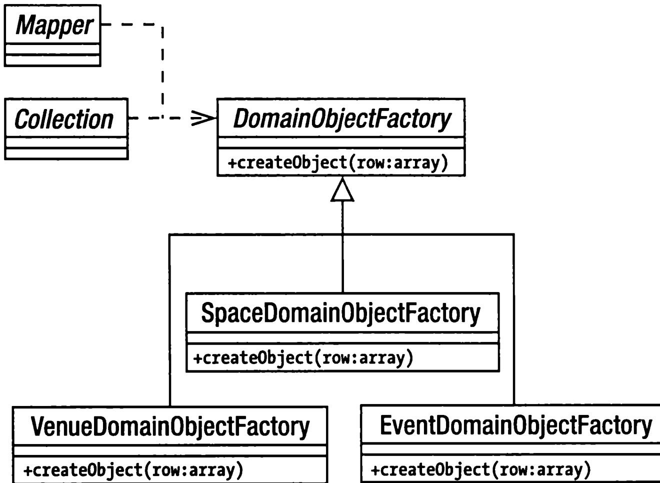


Рис. 13.6. *Классы, созданные в соответствии с шаблоном Domain Object Factory*

У классов, созданных в соответствии с шаблоном Domain Object Factory. имеется лишь одна основная обязанность, поэтому они, как правило, очень просты, как показано в следующем примере абстрактного класса:

```
// Листинг 13.33
abstract class DomainObjectFactory
{
    abstract public function createObject(array $row): DomainObject;
}
```

А вот как выглядит конкретная реализация этого абстрактного класса:

```
// Листинг 13.34
class VenueObjectFactory extends DomainObjectFactory
{
    public function createObject(array $row): Venue
    {
        $obj = new Venue((int)$row['id'], $row['name']);
        return $obj;
    }
}
```

Конечно, чтобы не допустить дублирования кода и предотвратить излишние обращения к базе данных, необходимо кешировать объекты, как это было сделано с классом `Mapper`. В приведенный выше класс можно также переместить методы `addToMap()` и `getFromMap()`, или можно устано-

вить взаимосвязь класса ObjectWatcher с методами createObject() по типу наблюдателя. Подробности оставляются вам в качестве упражнения, но не допускайте создания клонов объектов предметной области, чтобы система не стала неуправляемой!

Следствия

Шаблон Domain Object Factory отделяет исходные данные, полученные из базы данных, от данных из полей объектов. В теле метода createObject() можно произвести любое количество правок. Этот процесс прозрачен для клиента, обязанность которого — предоставлять исходные данные. Если убрать эти функциональные возможности из класса Mapper, они станут доступными для других компонентов. В качестве примера ниже приведена измененная реализация класса Collection:

```
// Листинг 13.35
abstract class Collection implements \Iterator
{
    protected int $total = 0;
    protected array $raw = [];
    private int $pointer = 0;
    private array $objects = [];
    // Collection
    public function __construct(array $raw = [],
                                protected ? DomainObjectFactory
$dofact = null)
    {
        if (count($raw) && ! is_null($dofact))
        {
            $this->raw = $raw;
            $this->total = count($raw);
        }

        $this->dofact = $dofact;
    }
    // ...
```

Объект типа DomainObjectFactory можно использовать для генерации объектов по запросу, как показано ниже:

```
// Листинг 13.36
private function getRow(int $num): ? DomainObject
{
    // ...
```

```
    if (isset($this->raw[$num]))
    {
        $this->objects[$num] = $this->dofact->createObject(
            $this->raw[$num]);
        return $this->objects[$num];
    }
}
```

Объекты типа DomainObjectFactory отделены от базы данных, поэтому их можно более эффективно использовать для тестирования. Например, можно создать имитирующий объект типа DomainObjectFactory, чтобы протестировать исходный код класса Collection. Сделать это намного проще, чем имитировать целый объект типа Mapper (подробнее о заглушках и имитирующих объектах речь пойдет в главе 18, "Тестирование средствами PHPUnit").

Одним из общих результатов разбиения единого компонента на составные части оказывается неизбежное размножение классов. Поэтому нельзя недооценивать вероятность путаницы. Даже если каждый компонент и его связи с другими равноправными компонентами логичны и явно определены, нередко бывает трудно составлять пакеты, содержащие десятки компонентов со сходными наименованиями.

Следует, однако, иметь в виду, что всегда перед тем, как стать лучше, ситуация ухудшается. Уже сейчас можно заметить еще одно ограничение, присущее шаблону Data Mapper. Так, метод Mapper::getCollection() был удобным, но ведь и в других классах может возникнуть потребность в получении коллекции типа Collection для определенного типа предметной области, не обращаясь к классу, нацеленному на операции с базой данных. Итак, у нас имеются два связанных вместе абстрактных компонента типа — Collection и DomainObjectFactory. Исходя из типа используемого объекта предметной области нам потребуются разные конкретные реализации, например VenueCollection и VenueDomainObjectFactory или SpaceCollection и SpaceDomainObjectFactory. И это затруднение, безусловно, приводит нас непосредственно к шаблону Abstract Factory. На рис. 13.7 приведен класс PersistenceFactory, реализующий этот шаблон. Мы воспользуемся им для организации различных компонентов, из которых можно составить ряд следующих шаблонов.

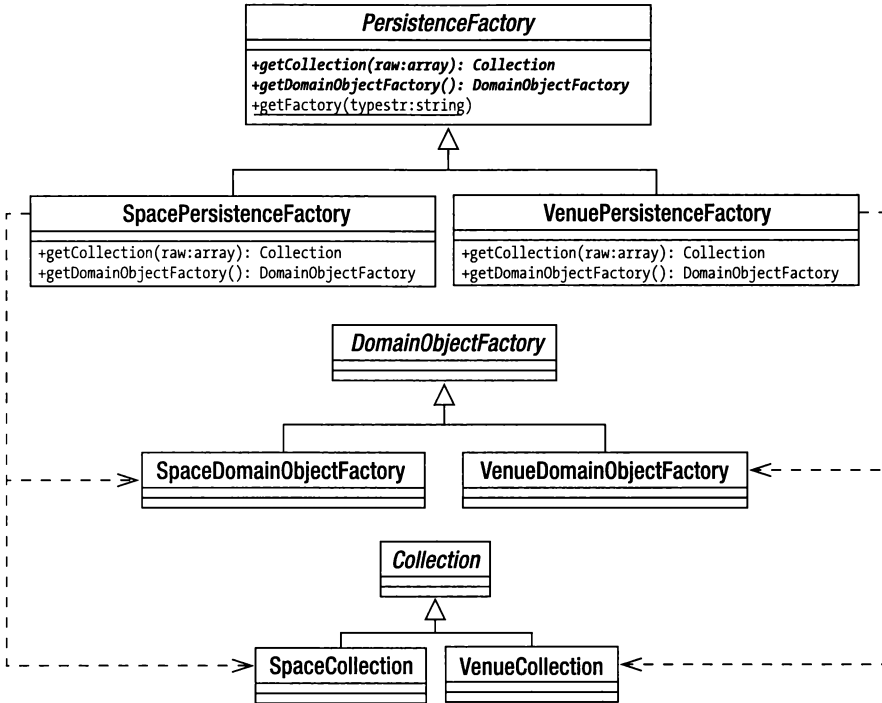


Рис. 13.7. *Применение шаблона Abstract Factory для организации связанных вместе компонентов*

Шаблон Identity Object

Представленная выше реализация преобразователя данных отличается недостаточной гибкостью, когда речь идет о выявлении объектов предметной области. Найти отдельный объект совсем не трудно, как, впрочем, и все подходящие объекты предметной области. Но если требуется сделать нечто среднее между этими операциями, то придется добавить специальный метод для подготовки запроса. В данном случае это метод `EventMapper::findBySpaceId()`.

Шаблон Identity Object (Объект идентичности), который у Алура и других называется “Data Transfer Object” (Объект передачи данных), инкапсулирует критерии запроса, развязывая тем самым систему от синтаксиса базы данных.

Проблема

Трудно знать заранее, что именно понадобится искать в базе данных вам или другим программистам. Чем сложнее объект предметной области, тем больше фильтров может потребоваться в запросе. Эту задачу можно в какой-то степени решить, добавляя от случая к случаю дополнительные методы в классы типа `Mapper`. Но это, конечно, не очень удобно и может привести к дублированию кода, когда потребуется подготовить много схожих, но все же различающихся запросов как в одном классе `Mapper`, так и во всех преобразователях данных, применяемых в системе.

Шаблон Identity Object инкапсулирует условные составляющие запроса к базе данных таким образом, чтобы их можно было сочетать в различных комбинациях во время выполнения. Так, если объект предметной области называется `Person`, то клиент может вызывать методы для объекта идентичности, класс которого создан в соответствии с шаблоном Identity Object, чтобы обозначить мужчину в возрасте от 30 до 40 лет ростом до 180 см. Класс должен быть спроектирован в соответствии с шаблоном Identity Object для удобства объединения условий в запросе, когда, например, рост человека не имеет значения или требуется исключить из условия нижнюю границу возраста. В какой-то степени шаблон Identity Object ограничивает возможности программиста клиентского кода. Так, если вы не написали код, в котором предусмотрено поле `income`, то его нельзя будет добавить в запрос, не внеся соответствующих изменений. Тем не менее возможность задавать условия в различных комбинациях — это шаг вперед в направлении гибкости. А теперь выясним, как этого добиться.

Реализация

Шаблон Identity Object обычно состоит из ряда методов, которые можно вызывать для составления критериев запроса. Установив состояние объекта, его можно передать методу, который отвечает за подготовку оператора SQL в запросе.

На рис. 13.8 показана типичная иерархия классов, созданных в соответствии с шаблоном Identity Object.

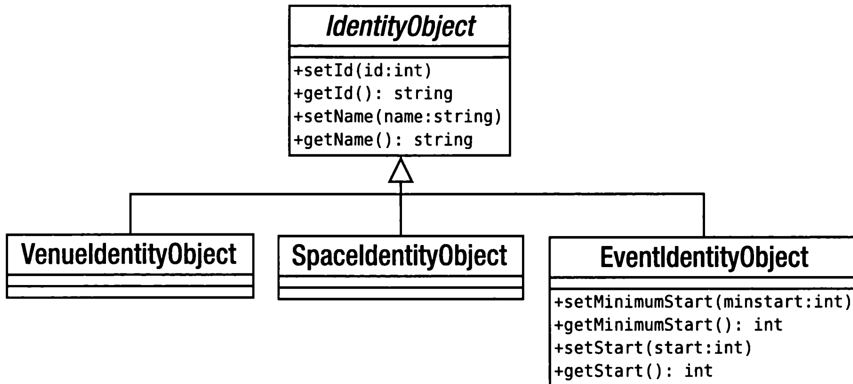


Рис. 13.8. *Иерархия классов, созданных в соответствии с шаблоном Identity Object для управления критериями запросов*

Чтобы управлять общими операциями и обеспечить совместное использование какого-нибудь общего типа в объектах критериев, можно определить базовый класс. Ниже приведена более простая реализация, чем иерархия классов, показанная на рис. 13.8:

```
// Листинг 13.37
abstract class IdentityObject
{
    private ? string $name = null;
    public function setName(string $name): void
    {
        $this->name = $name;
    }
    public function getName(): ? string
    {
        return $this->name;
    }
}
```

Здесь нет ничего слишком обременительного. В приведенных выше классах просто сохраняются предоставленные данные, которые отдаются по запросу. Ниже приведен код, который может быть использован в классе `EventIdentityObject` для составления предложения `WHERE` языка SQL:

```
// Листинг 13.38
$idobj = new EventIdentityObject();
$idobj->setMinimumStart(time());
```

```
$idobj->setName("A Fine Show");
$comps = [];
$name = $idobj->getName();
if (! is_null($name))
{
    $comps[] = "name = '{$name}'";
}
$minstart = $idobj->getMinimumStart();

if (! is_null($minstart))
{
    $comps[] = "start > {$minstart}";
}
$start = $idobj->getStart();
if (! is_null($start))
{
    $comps[] = "start = '{$start}'";
}
$clause = " WHERE " . implode(" and ", $comps);
print "{$clause}\n";
```

Такая модель окажется вполне работоспособной, но она вряд ли устроит ленивую натуру разработчика, поскольку для крупного объекта предметной области ему придется сначала создать изрядное количество методов получения и установки, а затем, следуя этой модели, написать код для вывода каждого условия, заданного в предложении `WHERE`. Трудно заставить себя учесть все возможные варианты в коде данного примера. Поэтому представьте удовольствие, которое испытает разработчик, создавая объект идентичности в соответствии с шаблоном Identity Object с целью решить конкретную практическую задачу.

К счастью, существуют различные стратегии, которые можно применять для автоматизации сбора данных и генерации кода SQL. Раньше, например, в базовом классе я заносил в ассоциативные массивы имена полей базы данных. Они сами индексировались в разнотипных операциях сравнения: больше, равно, меньше или равно. А в дочерних классах предусматривались удобные методы для добавления этих данных в базовую структуру. Построитель SQL-запросов мог затем обойти эту структуру в цикле, чтобы составить свой запрос динамически. Можно с уверенностью сказать, что реализация подобной системы — дело техники, поэтому мы рассмотрим здесь один из вариантов.

Итак, воспользуемся текучим интерфейсом (fluent interface). Это класс, в котором методы установки возвращают экземпляры объектов, предо-

ставляя пользователям возможность свободно объединять объекты в цепочку, как в естественном языке. И хотя такой подход потворствует лени, можно надеяться, что он даст программисту клиентского кода удобную возможность составлять критерии запроса.

Начнем, пожалуй, с создания класса `\woo\mapper\Field`, предназначенного для хранения данных сравнения для каждого поля, которое в конечном итоге окажется в предложении `WHERE`:

```
// Листинг 13.39
class Field
{
    protected array $comps = [];
    protected bool $incomplete = false;
    // Установка имени поля (например, age)
    public function __construct(protected string $name)
    {
    }
    // Добавление оператора и значения для тестирования
    // (например, больше 40), а также свойство $comps
    public function addTest(string $operator, $value): void
    {
        $this->comps[] = [
            'name' => $this->name,
            'operator' => $operator,
            'value' => $value
        ];
    }
    // $comps - это массив, поэтому одно поле
    // можно проверить не одним, а несколькими способами
    public function getComps(): array
    {
        return $this->comps;
    }
    // Если массив $comps не содержит элементов,
    // значит, данные сравнения с полем и
    // само поле не готовы для применения в запросе
    public function isIncomplete(): bool
    {
        return empty($this->comps);
    }
}
```

Этому простому классу передается имя поля, которое он сохраняет внутри. С помощью метода `addTest()` в данном классе создается массив, содержащий элементы `$operator` и `$value`. Это позволяет поддерживать

несколько операторов сравнения для одного поля. А теперь рассмотрим новый класс `IdentityObject`:

```
// Листинг 13.40
class IdentityObject
{
    protected ? Field $currentfield = null;
    protected array $fields = [];
    private array $enforce = [];
    // Объект идентичности может быть создан пустым
    // или же с отдельным полем
    public function __construct( ? string $field = null,
                                 ? array $enforce = null)
    {
        if (! is_null($enforce))
        {
            $this->enforce = $enforce;
        }
        if (! is_null($field))
        {
            $this->field($field);
        }
    }
    // Имена полей, на которые наложено данное ограничение
    public function getObjectFields(): array
    {
        return $this->enforce;
    }
    // Добавляет новое поле.
    // Генерирует ошибку, если текущее поле неполное
    // (т.е. age, а не age > 40).
    // Этот метод возвращает ссылку на текущий объект
    // и тем самым разрешает текучий синтаксис
    public function field(string $fieldname): self
    {
        if (!$this->isVoid() && $this->currentfield->isIncomplete())
        {
            throw new \Exception("Неполное поле");
        }
        $this->enforceField($fieldname);
        if (isset($this->fields[$fieldname]))
        {
            $this->currentfield = $this->fields[$fieldname];
        }
        else
        {
            $this->currentfield = new Field($fieldname);
```

```
            $this->fields[$fieldname] = $this->currentfield;
        }
        return $this;
    }
    // Имеются ли уже какие-нибудь поля
    // у объекта идентичности?
    public function isVoid(): bool
    {
        return empty($this->fields);
    }
    // Допустимо ли заданное имя поля?
    public function enforceField(string $fieldname): void
    {
        if (! in_array($fieldname, $this->enforce) &&
            ! empty($this->enforce))
        {
            $forcelist = implode(', ', $this->enforce);
            throw new \Exception(
            "{$fieldname} не является корректным полем ($forcelist)"
            );
        }
    }
    // Добавляет оператор равенства в текущее поле,
    // т.е. 'age' превращается в 'age=40'.
    // Возвращает ссылку на текущий объект через operator()
    public function eq($value): self
    {
        return $this->operator("=", $value);
    }
    // Меньше
    public function lt($value): self
    {
        return $this->operator("<", $value);
    }
    // Больше
    public function gt($value): self
    {
        return $this->operator(">", $value);
    }
    // Выполняет работу, чтобы методы операторов
    // получали текущее поле, и добавляет оператор
    // и проверяемое значение
    private function operator(string $symbol, $value): self
    {
        if ($this->isVoid())
        {
            throw new \Exception("Поле объекта не определено");
```

```
        }
        $this->currentfield->addTest($symbol, $value);
        return $this;
    }
    // Возвращает все полученные до сих пор результаты
    // сравнения из ассоциативного массива
    public function getComps(): array
    {
        $ret = [];
        foreach ($this->fields as $field)
        {
            $ret = array_merge($ret, $field->getComps());
        }
        return $ret;
    }
}
```

Чтобы разобраться, что же происходит в данном коде, проще всего начать с клиентского кода, постепенно продвигаясь в обратном направлении:

```
// Листинг 13.41
$idobj = new IdentityObject();
$idobj->field("name")
      ->eq("'The Good Show'")
      ->field("start")
      ->gt(time())
      ->lt(time() + (24 * 60 * 60));
```

Сначала в приведенном выше фрагменте кода создается объект типа `IdentityObject`. Вызов метода `field()` приводит к созданию объекта типа `Field` и присвоению ссылки на него свойству `$currentfield`. Обратите внимание на то, что метод `field()` возвращает ссылку на объект типа `IdentityObject`. Это позволяет добавить в цепочку вызовы других методов после вызова метода `field()`. В каждом методе сравнения `eq()`, `gt()` и так далее вызывается метод `operator()`, в котором проверяется, существует ли текущий объект типа `Field`, которым требуется оперировать. И если он существует, то передается знак операции и предоставленное значение. Метод `eq()` также возвращает ссылку на объект идентичности, что позволяет добавить новые проверки или снова вызвать метод `field()`, чтобы приступить к обработке нового поля.

Обратите внимание на то, что клиентский код почти всегда похож на обычное предложение: значения в поле `"name"` равно `"The Good Show"`, а значение в поле `"start"` больше, чем текущее время, но меньше, чем время через сутки. Безусловно, теряя жестко закодированные методы, мы

в то же время теряем в какой-то степени в безопасности. Именно для этой цели и предназначен массив `$enforce`. Базовый класс может вызываться из подклассов, хотя и с некоторыми ограничениями, как показано ниже:

```
// Листинг 13.42
class EventIdentityObject extends IdentityObject
{
    public function __construct(string $field = null)
    {
        parent:: construct(
            $field,
            ['name', 'id', 'start', 'duration', 'space']
        );
    }
}
```

Теперь класс `EventIdentityObject` вводит в действие ряд полей. Вот что получается при попытке обработать случайное имя поля:

```
// Листинг 13.43
try
{
    $idobj = new EventIdentityObject();
    $idobj->field("banana")
          ->eq("The Good Show")
          ->field("start")
          ->gt(time())
          ->lt(time() + (24 * 60 * 60));
    print $idobj;
}
catch (\Exception $e)
{
    print $e->getMessage();
}
```

А вот какой вид имеет вывод приведенного выше кода:

```
banana not a legal field (name, id, start, duration, space)
```

Следствия

Шаблон Identity Object дает программистам клиентского кода возможность определять критерии поиска без обращения к запросу к базе данных. Он также избавляет от необходимости создавать специальные методы запросов для различных видов операций поиска, которые могут понадобиться пользователю.

Одна из целей шаблона Identity Object — оградить пользователей от подробностей реализации базы данных. Так, если вырабатывается автоматическое решение наподобие текучего интерфейса из предыдущего примера, то очень важно, чтобы используемые метки ясно обозначали объекты предметной области, а не исходные имена столбцов в таблице базы данных. И если они не соответствуют друг другу, то придется создавать механизм для их сопоставления.

Если же применяются специализированные объекты-сущности (по одному на каждый объект предметной области), рекомендуется воспользоваться абстрактной фабрикой, построенной в соответствии с шаблоном Abstract Factory (например, классом `PersistenceFactory`, описанным в предыдущем разделе), чтобы обслуживать их вместе с другими объектами, связанными с текущим объектом предметной области. А теперь, когда мы можем представить критерии поиска, воспользуемся ими для составления самого запроса.

Шаблоны Selection Factory и Update Factory

Итак, мы уже забрали некоторые обязанности у классов типа `Mapper`. Имея в своем распоряжении рассматриваемые здесь шаблоны, в классе `Mapper` не нужно создавать объекты или коллекции. Если критерии запросов обрабатываются в объектах идентичности, создаваемых в соответствии с шаблоном Identity Object, то отпадает необходимость обрабатывать многие варианты в методе `find()`. А на следующей стадии предстоит исключить обязанности по составлению запроса.

Проблема

Любая система, которая обращается к базе данных, должна формировать запросы, но сама система организована вокруг объектов предметной области и логики приложения, а не базы данных. Можно сказать, что многие шаблоны, описанные в этой главе, наводят мосты между табличной базой данных и более естественными древовидными структурами предметной области. Но существует момент преобразования, т.е. точка, в которой данные предметной области преобразуются в форму, приемлемую для базы данных. Именно в этой точке и происходит настоящая развязка.

Реализация

Безусловно, многие из подобных функциональных возможностей можно обнаружить в рассматривавшемся ранее шаблоне Data Mapper. Но такая специализация позволяет выгодно воспользоваться дополнительными функциональными возможностями, предоставляемыми шаблоном Identity Object. Следовательно, составление запросов может стать более динамичным просто потому, что очень велико возможное количество вариантов.

На рис. 13.9 показаны простые фабрики выбора и обновления, построенные в соответствии с шаблонами Selection Factory и Update Factory.

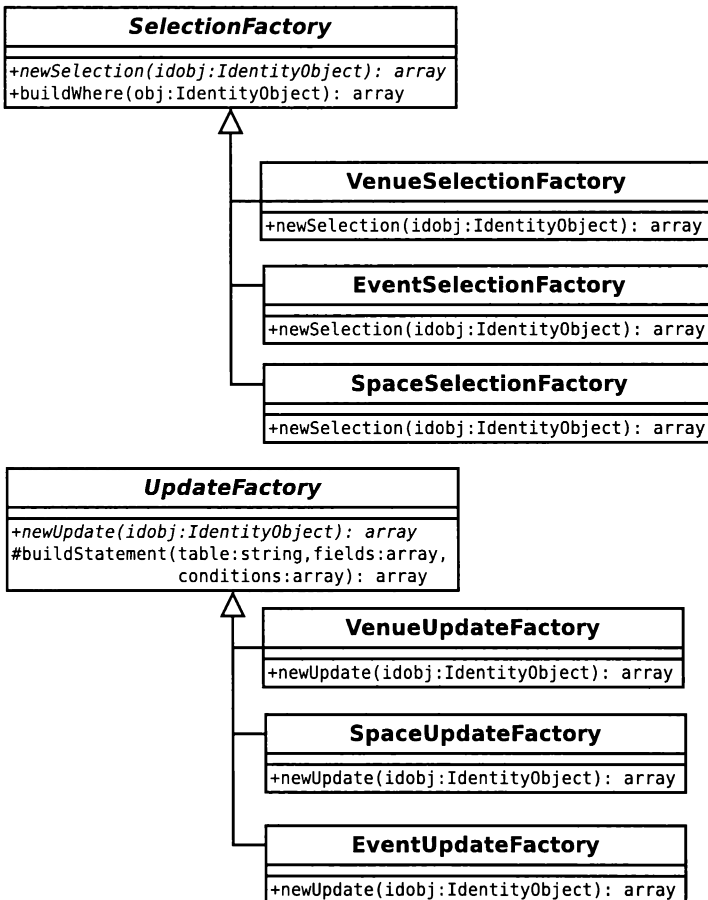


Рис. 13.9. *Фабрики выбора и обновления, построенные в соответствии с шаблонами Selection Factory и Update Factory*

Следует еще раз подчеркнуть, что шаблоны Selection Factory и Update Factory обычно организованы так, что они соответствуют объектам предметной области в системе, посредниками которых, возможно, являются объекты идентичности. Именно по этой причине они также подходят для создания класса `PersistenceFactory`, тогда как шаблон Abstract Factory служит в качестве арсенала для инструментальных средств сохраняемости объектов предметной области. Ниже приведена реализация базового класса в соответствии с шаблоном Update Factory:

```
// Листинг 13.44
abstract class UpdateFactory
{
    abstract public function newUpdate(DomainObject $obj): array;
    protected function buildStatement(string $table, array $fields,
                                ? array $conditions = null): array
    {
        $terms = array();
        if (! is_null($conditions))
        {
            $query = "UPDATE {$table} SET ";
            $query .= implode(" = ?,", array_keys($fields))." = ?";
            $terms = array_values($fields);
            $cond = [];
            $query .= " WHERE ";
            foreach ($conditions as $key => $val)
            {
                $cond[] = "$key = ?";
                $terms[] = $val;
            }
            $query .= implode(" AND ", $cond);
        }
        else
        {
            $qs = [];
            $query = "INSERT INTO {$table} (";
            $query .= implode(",", array_keys($fields));
            $query .= ") VALUES (";
            foreach ($fields as $name => $value)
            {
                $terms[] = $value;
                $qs[] = '?';
            }
            $query .= implode(",", $qs);
            $query .= ")";
        }
```

```
        return [$query, $terms];
    }
}
```

С точки зрения интерфейса в этом классе определяется лишь метод `newUpdate()`, возвращающий массив, содержащий строку запроса, а также список условий, которые требуется применить в ней. Метод `buildStatement()` выполняет общую работу по составлению запроса на обновление, а всю работу, а не конкретные действия над отдельными объектами предметной области, выполняют дочерние классы. Методу `buildStatement()` передаются имя таблицы, ассоциативный массив полей и их значений и аналогичный ассоциативный массив условий. Все это объединяется в данном методе для составления запроса. Ниже приведено определение конкретного класса `UpdateFactory`:

```
// Листинг 13.45
class VenueUpdateFactory extends UpdateFactory
{
    public function newUpdate(DomainObject $obj): array
    {
        // Обратите внимание на удаленную проверку типа
        $id = $obj->getId();
        $cond = null;
        $values['name'] = $obj->getName();

        if ($id > 0)
        {
            $cond['id'] = $id;
        }

        return $this->buildStatement("venue", $values, $cond);
    }
}
```

В данной реализации происходит непосредственная работа с объектом типа `DomainObject`. В тех системах, в которых одновременно может обновляться много объектов, можно воспользоваться объектом идентичности для определения совокупности объектов, которыми требуется оперировать. Тем самым формируется основание для массива `$cond`, который содержит только данные `$id`.

В методе `newUpdate()` извлекаются данные, необходимые для формирования запроса. Это процесс, посредством которого данные объекта преобразуются в информацию для базы данных. Обратите внимание на про-

верку значения $id. Если для идентификатора установлено значение -1, то это новый объект предметной области, и мы не будем предоставлять условное значение buildStatement(). buildStatement() использует наличие условных инструкций для выяснения, следует ли генерировать запросы INSERT и UPDATE.

Обратите внимание на то, что методу newUpdate() можно передать любой объект типа DomainObject. Дело в том, что все классы типа UpdateFactory могут совместно использовать общий интерфейс. К этому было бы неплохо добавить дополнительную проверку типов, чтобы гарантировать, что неверный объект не будет передан.

Ниже представлен небольшой фрагмент кода, в котором проверяется работа класса VenueUpdateFactory:

```
// Листинг 13.46
$vuf = new VenueUpdateFactory();
print_r($vuf->newUpdate(new Venue(334, "The Happy Hairband")));
Array
(
    [0] => UPDATE venue SET name = ? WHERE id = ?
                                   [1] => Array
                                   (
                                       [0] => The Happy Hairband
                                       [1] => 334
                                   )
)
```

А вот как выглядит генерация инструкции INSERT:

```
// Листинг 13.47
$vuf = new VenueUpdateFactory();
print_r($vuf->newUpdate(new Venue(-1, "The Lonely Hat Hive")));
Array
(
    [0] => INSERT INTO venue(name) VALUES( ?)
    [1] => Array
    (
        [0] => The Lonely Hat Hive
    )
)
```

Рассмотрим аналогичную структуру для классов SelectionFactory. Ниже приведено определение базового класса:

```
// Листинг 13.48
abstract class SelectionFactory
```

```
{
    abstract public function newSelection(IdentityObject $obj): array;
    public function buildWhere(IdentityObject $obj): array
    {
        if ($obj->isVoid())
        {
            return ["", []];
        }
        $compstrings = [];
        $values = [];
        foreach ($obj->getComps() as $comp)
        {
          $compstrings[] = "{$comp['name']} {$comp['operator']} ?";
          $values[] = $comp['value'];
        }
        $where = "WHERE " . implode(" AND ", $compstrings);
        return [$where, $values];
    }
}
```

И снова в классе определен общедоступный интерфейс в форме абстрактного класса. В методе newSelection() ожидается объект типа IdentityObject, который требуется также во вспомогательном методе buildWhere(), но он должен быть локальным по отношению к текущему типу. В самом методе buildWhere() вызывается метод IdentityObject::getComps() с целью получить сведения, требующиеся для создания предложения WHERE, а также составить список значений, причем и то, и другое возвращается в двухэлементном массиве.

Ниже приведено определение конкретного класса SelectionFactory:

```
// Листинг 13.49
class VenueSelectionFactory extends SelectionFactory
{
    public function newSelection(IdentityObject $obj): array
    {
        $fields = implode(',', $obj->getObjectFields());
        $core = "SELECT $fields FROM venue";
        list($where, $values) = $this->buildWhere($obj);
        return [$core . " " . $where, $values];
    }
}
```

В этом классе создается основа оператора SQL, а затем вызывается метод buildWhere(), чтобы добавить в этот оператор условное предложение. На самом деле единственное, что отличает один конкретный класс

SelectionFactory от другого в тестовом коде, — это имя таблицы. Если же в ближайшее время какая-то особая специализация не понадобится, эти подклассы можно убрать и пользоваться дальше одним конкретным классом SelectionFactory, в котором будет запрашиваться имя таблицы из объекта типа PersistenceFactory.

Приведем снова небольшой фрагмент клиентского кода:

```
// Листинг 13.50
$vio = new VenueIdentityObject();
$vio->field("name")->eq("The Happy Hairband");
$vsf = new VenueSelectionFactory();
print_r($vsf->newSelection($vio));
(
    [0] => SELECT name, id FROM venue WHERE name = ?
            [1] => Array
            (
                [0] => The Happy Hairband
            )
)
```

Следствия

Благодаря обобщенной реализации шаблона Identity Object упрощается применение единственного параметризированного класса SelectionFactory. Если вы предпочитаете создавать в соответствии с шаблоном Identity Object жестко закодированные объекты идентичности, состоящие из списка методов получения и установки, вам, скорее всего, придется создавать по одному классу SelectionFactory на каждый объект предметной области.

К числу самых важных преимуществ фабрик запросов, строящихся в соответствии с шаблоном Query Factory, вместе с объектами идентичности, создаваемыми в соответствии с шаблоном Identity Object, относится целый ряд запросов, которые можно формировать. Но это может вызвать трудности при кешировании. Фабричные методы формируют запросы динамически, и поэтому трудно сказать, не приведет ли это к дублированию. Возможно, имеет смысл создать средство сравнения объектов идентичности, чтобы возвратить кешированную символьную строку, избежав всех этих хлопот. Аналогичный вид объединения операторов в запросах базы данных можно рассмотреть и на более высоком уровне.

Еще один вопрос сочетания рассматривавшихся здесь шаблонов связан с тем, что они не *настолько* гибкие, насколько хотелось бы. Под этим подразумевается, что они должны обладать способностью хорошо приспосабливаться в установленных для них границах. Для исключительных случаев при этом остается мало возможностей. Классы типа `Mapper`, которые труднее создавать и поддерживать, очень легко приспосабливаются к любым проблемам производительности или операций с данными, которые, возможно, потребуется выполнять, скрывая за их аккуратно построенными интерфейсами API. Эти более изящные шаблоны страдают тем недостатком, что их конкретные обязанности и акцент на совместное применение могут сильно затруднить попытки сделать что-нибудь нелепое, но эффективное вопреки здравому смыслу.

К счастью, мы не потеряли интерфейс более высокого уровня. У нас еще есть уровень контроллера, на котором мы можем разумно продемонстрировать свою предусмотрительность, если потребуется.

Что теперь осталось от шаблона Data Mapper

Мы убрали из преобразователя данных, созданного в соответствии с шаблоном Data Mapper, функции формирования объектов, запросов и коллекций, не говоря уже об организации условных конструкций. Что же от него осталось? То, что требуется от преобразователя данных в зачаточной форме. Но нам по-прежнему требуется объект, который находится над другими созданными объектами и координирует их действия. Это поможет выполнять задачи кеширования и управлять подключением к базе данных, хотя обязанности по взаимодействию с базой данных могут быть делегированы еще выше. Клифтон Нок называет контроллеры, действующие на уровне представления данных, сборщиками объектов предметной области.

Обратим к конкретному примеру:

```
// Листинг 13.51
class DomainObjectAssembler
{
    protected \PDO $pdo;
    public function __construct(private PersistenceFactory $factory)
    {
        $reg = Registry::instance();
        $this->pdo = $reg->getPdo();
    }
```

```
    public function getStatement(string $str): \PDOStatement
    {
        if (! isset($this->statements[$str]))
        {
            $this->statements[$str] = $this->pdo->prepare($str);
        }

        return $this->statements[$str];
    }
    public function findOne(IdentityObject $idobj): DomainObject
    {
        $collection = $this->find($idobj);
        return $collection->next();
    }
    public function find(IdentityObject $idobj): Collection
    {
        $selfact = $this->factory->getSelectionFactory();
        list($selection, $values) = $selfact->newSelection($idobj);
        $stmt = $this->getStatement($selection);
        $stmt->execute($values);
        $raw = $stmt->fetchAll();
        return $this->factory->getCollection($raw);
    }
    public function insert(DomainObject $obj): void
    {
        $upfact = $this->factory->getUpdateFactory();
        list($update, $values) = $upfact->newUpdate($obj);
        $stmt = $this->getStatement($update);
        $stmt->execute($values);

        if ($obj->getId() < 0)
        {
            $obj->setId((int)$this->pdo->lastInsertId());
        }

        $obj->markClean();
    }
}
```

Как видите, это не абстрактный класс. Вместо его разделения по разным специализациям в нем применяется класс `PersistenceFactory`, чтобы гарантировать получение подходящих компонентов для текущего объекта предметной области.

На рис. 13.10 приведены участники, вынесенные на более высокий уровень из типа `Mapper`.

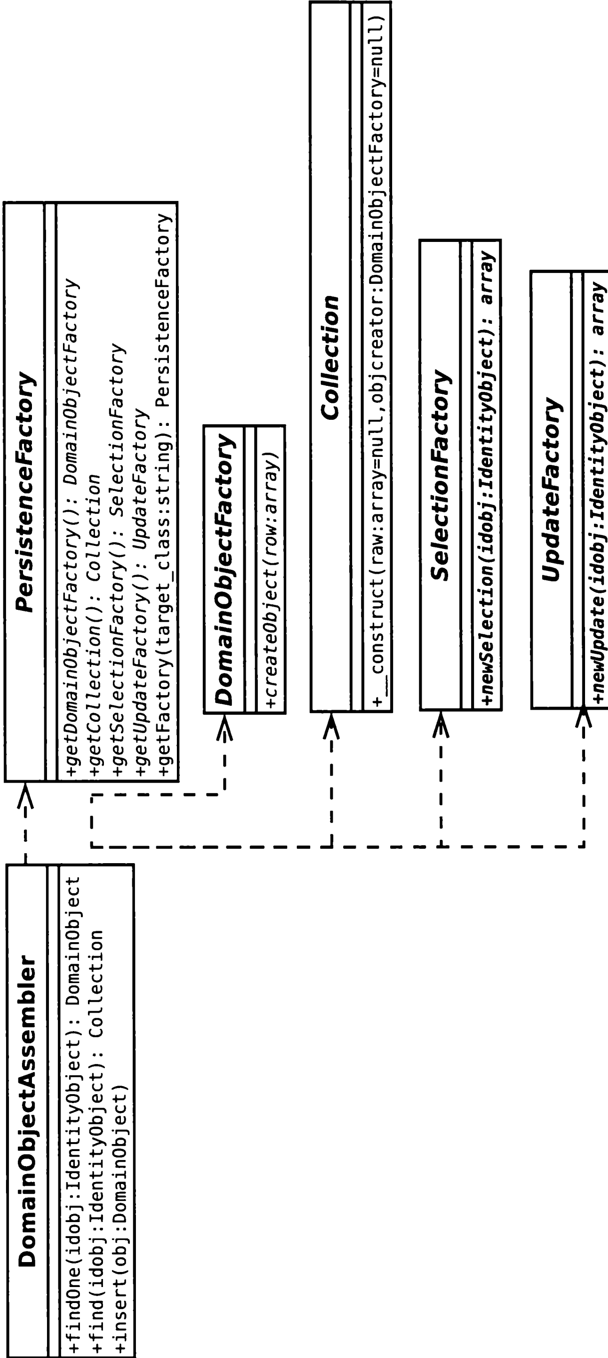


Рис. 13.10. *Некоторые постоянно хранимые классы, разработанные в этой главе*

Помимо установления связи с базой данных и выполнения запросов, упомянутый выше класс управляет объектами типа SelectionFactory и UpdateFactory. Если требуется выбрать коллекцию, в этом классе происходит обращение к соответствующему классу Collection, чтобы сформировать возвращаемые значения.

С точки зрения клиента создать объект типа DomainObjectFactory совсем не трудно. Для этого достаточно получить подходящий объект конкретного типа PersistenceFactory и передать его конструктору, как показано ниже:

```
// Листинг 13.52
$factory = PersistenceFactory::getFactory(Venue::class);
$finder = new DomainObjectAssembler($factory);
```

Разумеется, в качестве "клиента" здесь вряд ли подразумевается конечный клиент. Классы более высокого уровня можно обособить даже от такой сложности, добавляя метод getFinder() непосредственно в класс PersistenceFactory и превратив приведенный выше пример в единственную строку кода:

```
$finder = PersistenceFactory::getFinder(Venue::class);
```

Но оставим это читателю в качестве упражнения. Программист клиентского кода может затем перейти к получению коллекции объектов типа Venue:

```
// Листинг 13.53
$idobj = $factory->getIdentityObject()
                ->field('name')
                ->eq('The Eyeball Inn');
$collection = $finder->find($idobj);

foreach ($collection as $venue)
{
    print $venue->getName() . "\n";
}
```

Резюме

Как обычно, выбор шаблонов зависит от характера решаемой задачи. Я лично предпочитаю выбирать шаблон Data Mapper для работы с объектом идентичности, создаваемым в соответствии с шаблоном Identity Object. Мне нравятся аккуратные автоматизированные решения, но мне также нужно

знать, что я могу выйти за рамки системы и работать вручную, когда мне это понадобится, поддерживая в то же время ясный интерфейс и развязанный уровень базы данных. Например, у меня может возникнуть потребность оптимизировать SQL-запрос или воспользоваться соединением (`JOIN`), чтобы получить данные из нескольких таблиц базы данных. Даже применяя сложный сторонний каркас, основанный на шаблонах, вы можете обнаружить, что такое замысловатое объектно-реляционное преобразование дает не совсем то, что нужно. Одним из критериев качества каркаса и системы собственной разработки служит простота, с которой в нее можно внедрить новый модуль, не нарушая общей целостности всей системы. Мне нравятся изящные, красиво составленные решения, но я все же прагматик!

В этой главе был изложен довольно объемный материал. В частности, были представлены следующие шаблоны баз данных.

- *Data Mapper*. Создает специализированные классы для взаимного преобразования объектов, созданных в соответствии с шаблоном Domain Model, и данных, извлекаемых из реляционной базы данных.
- *Identity Map*. Отслеживает все объекты в системе, чтобы предотвратить повторное получение экземпляров и лишние обращения к базе данных.
- *Unit of Work*. Автоматизирует процесс, посредством которого объекты сохраняются в базе данных, а также гарантирует обновление только измененных объектов и добавление в нее только вновь созданных объектов.
- *Lazy Load*. Откладывает создание объектов и даже запросы базы данных до тех пор, пока они действительно не понадобятся.
- *Domain Object Factory*. Инкапсулирует функции создания объектов.
- *Identity Object*. Позволяет клиентам составлять критерии запросов, не обращаясь к основной базе данных.
- *Query (Selection и Update) Factory*. Инкапсулирует логику создания SQL-запросов.
- *Domain Object Assembler*. Создает контроллер, который управляет высокоуровневым процессом сохранения и извлечения данных.

В следующей главе мы сделаем долгожданную передышку, чтобы немного отвлечься от написания конкретного кода, и обсудим некоторые более общие практики, способствующие успешному выполнению проектов.

Часть III

Практика

ГЛАВА 14

Практика — хорошая (и плохая)

До сих пор мы уделяли основное внимание программированию, делая акцент на роли проектирования в построении гибких и повторно используемых инструментальных средств и приложений. Но разработка не заканчивается написанным кодом. Прочитав книги и пройдя курсы, можно получить прочные знания и навыки программирования на конкретном языке, но, как только дело дойдет до ввода в эксплуатацию и развертывания готового проекта, все еще могут возникнуть затруднения.

В этой главе мы отойдем от написания кода, чтобы представить ряд инструментальных средств и методик, служащих основанием для успешного процесса разработки. Здесь будут рассмотрены следующие вопросы.

- *Пакеты сторонних разработчиков.* Где их найти и когда применять.
- *Построение.* Создание и развертывание пакетов.
- *Контроль версий.* Гармонизация процесса разработки.
- *Документация.* Написание кода, который легко понять, использовать и расширить.
- *Модульное тестирование.* Инструментальное средство для автоматического обнаружения и предотвращения ошибок.
- *Стандарты.* Причины, по которым иногда полезно следовать принятым нормам.
- *Vagrant.* Инструментальное средство, в котором виртуализация применяется для того, чтобы все разработчики могли работать с системой, похожей на производственную среду, независимо от применяемого оборудования или операционной системы.
- *Модульное тестирование.* Инструмент для автоматического обнаружения и предотвращения ошибок.
- *Непрерывная интеграция.* Применение данной нормы практики и набора инструментальных средств для автоматизации процесса создания и тестирования проекта, которые способны предупредить разработчика о возникших осложнениях.

Не кодом единым

Когда я начал работать в команде разработчиков по окончании периода самостоятельной практики, я был поражен тем, как много знают мои коллеги по работе. Мы бесконечно спорили по вопросам, казалось бы, жизненной важности: какой текстовый редактор наилучший, должна ли команда стандартизировать интегрированную среду разработки, должны ли мы устанавливать стандарт для кодирования, как следует тестировать код, необходимо ли документировать процесс разработки? Иногда эти вопросы казались более важными, чем сам код, и, по-видимому, мои коллеги приобрели свои энциклопедические знания в этой области в ходе какого-то странного процесса осмоса.

В книгах по языкам программирования PHP, Perl и Java, которые мне приходилось читать, рассматривался в основном код. Как упоминалось ранее, в большинстве книг по платформам программирования, как правило, приводятся только описания функций и синтаксиса и не рассматриваются вопросы проектирования. Если проектирование оказывается не соответствующим теме книги, то можно быть уверенным, что такие вопросы, как контроль версий и тестирование, обсуждаться в подобных книгах не будут. Но это не означает, что я критикую такие книги. Если цель книги — осветить основные функции языка программирования, то не удивительно, если это все, что она дает читателю.

Но, изучая вопросы программирования, я понял, что пренебрегал многими механизмами ежедневного существования проекта, и обнаружил, что от некоторых из этих подробностей зависит успешный или неудачный исход проектов, в разработке которых я участвовал. В этой и последующих главах мы отойдем от рассмотрения исключительно кода и исследуем некоторые инструментальные средства и методики, от которых может зависеть успешное завершение проектов.

Снова изобретаем колесо

Столкнувшись в проекте с трудным, но конкретным требованием (например, с необходимостью сделать синтаксический анализатор определенного формата или использовать новый протокол при обращении к удаленному серверу), многие разработчики предпочтут создать компонент для решения данной задачи. Возможно, таким способом лучше всего обу-

читься своему ремеслу. Создавая пакет, вы проникаете в суть проблемы и отшлифовываете новые методики, которым можно найти более широкое применение. Тем самым вы вкладываете средства одновременно в проект и в свои знания и навыки. Внедрив функциональные средства в свою систему, вы избавляете пользователей от необходимости загружать сторонние пакеты, хотя иногда вам придется решать непростые вопросы лицензирования. И вы испытаете огромное чувство удовлетворения, когда, протестировав компонент, который спроектировали сами, обнаружите, что — о, чудо из чудес! — он работает, причем именно так, как вы и задумали, когда его писали.

Но у всего этого, конечно, есть и оборотная сторона. Многие пакеты представляют собой инвестиции в виде тысяч человеко-часов: ресурс, которого у вас, возможно, нет. В качестве выхода из столь затруднительного положения можно разработать только те функции, которые требуются непосредственно для проекта, в то время как сторонний продукт, как правило, может выполнять много других дополнительных функций. Но тогда остается вопрос: если уже имеется свободно доступное инструментальное средство, то зачем тратить свой талант на его воспроизводство? Есть ли у вас время и ресурсы, чтобы разрабатывать, тестировать и отлаживать пакет? Не лучше ли потратить время и силы с большей пользой?

Я очень категоричен в вопросе изобретения колеса. Анализировать задачи и придумывать их решения — это именно то, чем в основном занимаются программисты. Но серьезно заняться архитектурой намного полезнее и перспективнее, чем писать какое-то средство, связывающее вместе три-четыре имеющихся компонента. Когда мною овладевает подобное искушение, я напоминаю себе о прошлых проектах. И хотя в моей практике решение начать все снова никогда не было роковым для проекта, мне все же приходилось наблюдать, как подобное решение срывает все планы и съедает прибыль. И вот я сижу с маниакальным блеском в глазах, вынашивая схемы и разрабатывая диаграммы классов, и не замечаю, что я настолько погряз в деталях компонента, что уже не представляю в уме общую картину.

После составления плана проекта мне нужно понять, какие именно функции должны находиться в кодовой базе, а какие из них следует позаимствовать со стороны. Допустим, что веб-приложение может формировать (или читать) RSS-канал, в котором необходимо проверять правильность адресов электронной почты и автоматически отправлять ответные почтовые сообщения, аутентифицировать пользователей или читать файл

конфигурации стандартного формата. Все эти задачи можно выполнить с помощью внешних пакетов.

В предыдущих изданиях данной книги было сделано предположение, что PEAR (PHP Extension and Application Repository — хранилище расширений и приложений на PHP) — это самое подходящее средство для перехода к пакетам. Но времена изменились и разработка приложений на PHP, определенно, перешла на применение диспетчера зависимостей Composer и его стандартного хранилища Packagist (`https://packagist.org`). А поскольку диспетчер зависимостей Composer управляет пакетами по отдельным проектам, то он в меньшей степени подвержен зловещему синдрому зависимостей, когда для разных пакетов требуются несовместимые версии одной и той же библиотеки. Кроме того, переход к диспетчеру зависимостей Composer и хранилищу Packagist означает, что у разработчиков веб-приложений на PHP оказалось больше шансов найти именно то, что они искали. Более того, пакеты PEAR доступны через хранилище Packagist (см. `https://packagist.org/packages/pear/`).

Итак, определив свои потребности, посетите сначала веб-сайт Packagist по указанному выше адресу, а затем воспользуйтесь диспетчером зависимостей Composer, чтобы установить свой пакет и управлять его зависимостями. Более подробно о диспетчере зависимостей Composer речь пойдет в следующей главе.

Чтобы дать некоторое представление о возможностях диспетчера Composer и хранилища Packagist, ниже приведены некоторые операции, которые можно выполнять с помощью имеющихся там пакетов.

- Кеширование вывода средствами пакета `pear/cache_lite`.
- Тестирование эффективности кода средствами библиотеки эталонного тестирования `athletic/athletic`.
- Абстрагирование от подробностей доступа к базе данных средствами пакета `doctrine/dbal`.
- Выделение RSS-каналов средствами пакета `simplepie/simplepie`.
- Отправка почты с вложениями средствами пакета `pear/mail`.
- Синтаксический анализ форматов файлов конфигурации с помощью пакета `symfony/config`.
- Синтаксический анализ URL и манипулирование ими с помощью пакета `league/uri`.

На веб-сайте Packagist по указанному выше адресу предоставляется эффективный поисковый механизм, позволяющий найти пакеты по своим потребностям или закинуть сеть пошире, чтобы получить более богатый улов с помощью этого поискового механизма. Но в любом случае придется уделить время оценке существующих пакетов, прежде чем решаться снова изобретать колесо.

Наличие задачи и пакета для ее решения еще не означает соответственно начало и конец проводимого анализа. И хотя желательно воспользоваться пакетом, чтобы сэкономить на излишней разработке, иногда такое решение может означать дополнительные издержки без реальных выгод. Так, если клиенту требуется приложение для отправки почты, это еще не означает, что следует автоматически воспользоваться пакетом `pear/mail`. В языке PHP для этой цели предусмотрена очень хорошая функция `mail()`, поэтому начать разработку лучше всего именно с нее. Но как только станет ясно, что придется проверять правильность всех адресов электронной почты по стандарту RFC822 и отправлять по электронной почте вложенные изображения, можно приступить к анализу других вариантов. Оказывается, в пакете `pear/mail` поддерживаются обе эти операции электронной почты (последняя — с помощью функции `mail_mime()`).

Многие программисты, включая меня самого, зачастую уделяют слишком много внимания созданию оригинального кода — иногда даже в ущерб своим проектам. В основе такого стремления к авторскому коду нередко лежит желание создавать код, а не использовать его многократно.

На заметку Нежелание пользоваться сторонними инструментальными средствами и решениями нередко укоренено на организационном уровне. Подобная тенденция (относиться с подозрением к сторонним программным продуктам) иногда еще называется *синдромом неприятия чужой разработки*.

Грамотные программисты смотрят на оригинальный код как на одно из инструментальных средств, способствующих успешному завершению проекта. Такие программисты анализируют имеющиеся в наличии ресурсы и умело пользуются ими. Если существует пакет, который может взять на себя выполнение некоторой задачи, значит, это именно то, что нужно. Перефразируя известный афоризм из среды Perl, можно сказать, что хорошие программисты ленивы.

Ведите себя хорошо

Истинность знаменитого изречения Сартра "Ад — это другие люди" ежедневно доказывают некоторые программные проекты. Это изречение может служить описанием отношений между клиентами и разработчиками, когда отсутствие нормального общения ведет к утраченным возможностям и искаженным приоритетам. Но данное изречение подходит и для описания общительных и сотрудничающих членов команд, когда речь заходит о совместно используемом коде.

Как только над проектом начинают работать несколько разработчиков, встает вопрос контроля версий. Один программист может работать над кодом, сохраняя копии в рабочем каталоге в ключевые моменты разработки. Но стоит только подключить к работе над проектом второго программиста, и такая стратегия сразу же терпит неудачу. Если второй программист работает в том же самом каталоге, то существует вероятность, что при сохранении он затрет работу первого, если оба не проявят особую внимательность и не будут всегда работать с разными версиями файлов.

Возможен и другой вариант. Оба программиста получают версию кодовой базы, чтобы работать отдельно. И все идет нормально до тех пор, пока не наступает время согласовать эти две версии. Если программисты не работали над совершенно разными наборами файлов, то окажется, что задача объединения двух или более веток разработки превратится в серьезное затруднение.

Именно здесь и пригодятся системы контроля версий Git, Subversion и аналогичные инструментальные средства, предназначенные для управления командной работой программистов. С помощью системы контроля версий вы можете получить собственную версию кодовой базы и работать над ней до тех пор, пока не будет получен удовлетворительный результат. Затем вы можете обновить свою версию в соответствии со всеми изменениями, которые внесли ваши коллеги. Система контроля версий автоматически встроит эти изменения в ваши файлы, уведомляя вас о любых противоречиях, которые она не сможет разрешить. Протестировав этот новый гибридный вариант кода, вы можете сохранить его в центральном хранилище, тем самым сделав его доступным для других разработчиков.

Системы контроля версий предоставляют и другие преимущества. Они ведут полную запись всех этапов проекта, чтобы можно было всегда вернуться к любому моменту развития проекта или получить соответствую-

щую копию исходного кода. В такой системе можно также создавать разные ветки и поддерживать общедоступную версию одновременно с разрабатываемой.

Испытав возможности контроля версий в своем проекте, вы вряд ли захотите разрабатывать без него другой проект. Одновременно работать с несколькими ветками проекта нелегко, особенно поначалу, но преимущества быстро станут очевидными. Контроль версий слишком полезен, чтобы обходиться без него. Подробнее о системе контроля версий Git речь пойдет в главе 17, "Контроль версий средствами Git".

На заметку Настоящее издание этой книги было написано и отредактировано как обычный текст с использованием Git в качестве инструмента для совместной работы.

Дайте коду крылья

Возникала ли у вас когда-нибудь ситуация, когда ваш код не находил применения, потому что его трудно было инсталлировать? Это особенно справедливо для проектов, которые разрабатываются "на месте". Такие проекты устанавливаются в своем контексте, в котором пароли, каталоги, базы данных и вызовы вспомогательных приложений программируются непосредственно в коде. Внедрить такой проект нелегко. Командам программистов придется всякий раз прочесывать исходный код и корректировать параметры настройки, чтобы они соответствовали новой рабочей среде.

Это затруднение можно отчасти разрешить, предусмотрев централизованный файл конфигурации или класс, чтобы все настройки можно было изменить в одном месте. Но даже в этом случае инсталляция окажется непростым делом. От сложности или простоты установки будет в значительной степени зависеть популярность любого распространяемого приложения. Это обстоятельство также будет препятствовать или же способствовать многократному и частому развертыванию приложения во время разработки.

Как и любую повторяющуюся или отнимающую время задачу, установку программного обеспечения следует автоматизировать. Помимо выполнения прочих задач, программа установки должна определить стандартные значения для мест установки, проверить и изменить права доступа, со-

здать базы данных и инициализировать переменные. В действительности программа установки может сделать практически все, что необходимо для полного развертывания устанавливаемого приложения, скопировав его из исходного каталога в дистрибутиве. И хотя это не освобождает пользователя от обязанности ввести в устанавливаемое приложение информацию о своей среде, процесс развертывания упрощается настолько, что он сводится к ответу на несколько вопросов или указанию ряда параметров в командной строке.

Облачные программные продукты, подобные AWS Elastic Beanstalk компании Amazon, позволяют по мере необходимости создавать тестовые и рабочие среды. Удачные решения для построения и установки проектов очень важны, поскольку они позволяют извлечь наибольшую выгоду из этих ресурсов. Не имеет никакого смысла создавать специальный сервер для автоматизации задач построения, если вы не в состоянии динамически развернуть свою систему.

Для разработчиков имеются различные средства построения. Например, средства PEAR и Composer служат для управления установкой (PEAR — централизованно, а Composer — в локальном каталоге разработчика). Для любого из этих средств можно создать отдельные пакеты, которые могут быть затем без особого труда загружены и установлены пользователями. Но построение — это нечто большее, чем процесс размещения файла `A` в каталоге `B`. На примере проекта с открытым исходным кодом в главе 19, "Автоматическое построение средствами Phing", мы рассмотрим `Phing` — перенесенный с платформы Java вариант распространенного инструментального средства `Ant` для создания приложений на этой платформе. Инструментальное средство `Phing` написано на PHP и предназначено для написания приложений на PHP, но его архитектура подобна `Ant`, и в ней применяется тот же формат XML для файлов построения.

Диспетчер зависимостей Composer очень хорошо выполняет ограниченный ряд задач, предлагая самую простую конфигурацию. На первый взгляд `Phing` обескураживает, хотя и с уступкой в пользу изрядной гибкости. Его можно применять не только для автоматизации всего, что угодно: от копирования файлов до преобразования XSLT; вы также можете легко расширить данное инструментальное средство, написав и встроив в него собственные задачи. Инструментальное средство `Phing` написано с применением объектно-ориентированных средств языка PHP, и в его архитектуре основной акцент делается на модульности и простоте расширения.

Инструментальные средства построения и средства, предназначенные для управления пакетами или зависимостями, совсем не исключают друг друга. Как правило, такое инструментальное средство применяется на стадии разработки для выполнения тестов, упорядочения проектов и подготовки пакетов, которые в конечном итоге развертываются средствами PEAR, Composer или даже дистрибутива, основанного на таких системах управления пакетами, как RPM и Apt.

Стандарты

Как упоминалось ранее, в настоящем издании книги акцент смещен с PEAR на Composer. Означает ли это, что Composer намного лучше, чем PEAR? Мне лично многое нравится в Composer, и только этого было бы достаточно, чтобы сместить в данной книге акцент на Composer. Но главная причина заключается в том, что к применению Composer перешли все разработчики веб-приложений на PHP. Диспетчер Composer стал стандартным средством для управления зависимостями, а это означает, что когда я обнаруживаю пакет в хранилище Packagist, то вместе с ним, вероятнее всего, обнаружатся и все его зависимости от связанных с ним пакетов. В этом хранилище можно даже найти многие пакеты PEAR.

Таким образом, выбор стандартного средства для управления зависимостями гарантирует доступность и взаимозаменяемость. Но стандарты распространяются не только на пакеты и зависимости, но и на порядок работы систем и написания кода. Если протоколы согласованы, то системы и команды разработчиков могут гармонично взаимодействовать друг с другом. И по мере того, как все больше систем составляется из отдельных компонентов, подобное взаимодействие приобретает все большее значение.

А там, где требуется особый порядок обработки данных (например, протоколирование), было бы идеально принять наиболее подходящий протокол. Но качество рекомендации, диктующее выбор форматов, уровней протоколирования и так далее, вероятно, имеет меньшее значение, чем то обстоятельство, что все разработчики ее придерживаются. Вряд ли стоит внедрять самый лучший стандарт, если его соблюдает только кто-нибудь один.

Более подробно мы обсудим стандарты в главе 15, "Стандарты PHP", уделив особое внимание некоторым рекомендациям, составленным группой разработчиков PHP-FIG. Эти рекомендации сокращенно называются "PSR" (PHP Standards Recommendations — рекомендованные стандарты

PHP) и охватывают все вопросы разработки: от кеширования до безопасности. В этой главе основное внимание будет уделено рекомендациям стандартов PSR-1 и PSR-2, направленным на решение трудного вопроса выбора стиля программирования, включая размещение фигурных скобок и реагирование на чью-нибудь просьбу сменить избранный порядок оформления исходного кода. После этого будет рассмотрено главное преимущество рекомендаций стандарта PSR-4, которое заключается в автозагрузке. Между прочим, поддержка рекомендаций стандарта PSR-4 является еще одной особенностью, которая выгодно отличает Composer.

Vagrant

Какой операционной системой пользуется ваша команда разработчиков? В некоторых организациях специально предписывается применять определенное сочетание аппаратных и программных средств. Но зачастую применяются как обязательные, так и необязательные средства. Так, один разработчик может работать на настольном компьютере под управлением операционной системы Fedora, другой — на переносном компьютере MacBook, а третий — на ПК под управлением Windows, возможно, потому что он любит играть в компьютерные игры.

Вполне возможно, рабочая система будет работать совсем в другой операционной системе, например в CentOS. Обеспечить работоспособность системы на разных платформах будет нелегко, ведь существует риск, что ни одна из этих платформ не похожа на рабочую систему. После запуска системы в эксплуатацию нежелательно решать вопросы, связанные с операционной системой, применяемой в рабочей среде. На практике рабочая система сначала развертывается в среде обкатки. Но даже в этом случае не лучше ли выяснить подобные вопросы заранее?

Vagrant — это технология, в которой виртуализация применяется с целью предоставить всем членам команды среду разработки, в как можно большей степени похожую на рабочую среду, в которой предполагается эксплуатировать разрабатываемый программный продукт. В этом случае для запуска этого продукта в эксплуатацию достаточно выполнить одну или две команды, а самое главное, что все члены команды разработчиков будут пользоваться своими излюбленными аппаратными и программными средствами (я лично большой поклонник ОС Fedora). Более подробно о Vagrant речь пойдет в главе 20, “Виртуальная машина Vagrant”.

Тестирование

Создавая класс или компонент, вы, вероятно, уверены, что он работает. Ведь в конечном итоге вы проверите его в процессе разработки, запустите систему с этим компонентом и проверите, насколько хорошо он интегрирован, доступны ли новые функциональные возможности и работает ли все так, как нужно.

Но можно ли быть уверенным в том, что классы будут продолжать работать именно так, как предполагалось изначально? На первый взгляд такой вопрос может показаться нелепым. Ведь прикладной код был тщательно проверен; почему же он должен вдруг перестать нормально работать? Безусловно, ничто не происходит случайно, и если не вводить ни одной строки в прикладной код, то можно спать спокойно. Но если проект активно развивается, то контекст отдельного компонента будет неизбежно меняться, и вполне возможно, что и сам компонент будет изменен, причем самыми разными способами.

Рассмотрим все эти вопросы по очереди, и, прежде всего — как изменение контекста компонента может привести к ошибкам. Даже в системе, в которой компоненты аккуратно развязаны, они все равно остаются взаимозависимыми. Объекты, применяемые в классе, возвращают значения, выполняют действия и принимают данные. Если любой из этих типов поведения меняется, то результат работы класса может привести к такому виду ошибки, который легко обнаружить, когда происходит отказ системы с выдачей сообщения об ошибке, в котором указаны имя файла и номер строки. Но намного более коварен тип изменений, который не приводит к ошибке на уровне интерпретатора, но затрудняет нормальную работу компонента. Если в одном классе делается предположение на основании данных из другого класса, то изменение этих данных может привести к принятию неправильного решения. Таким образом, в классе возникают ошибки, несмотря на то что в нем не было изменено ни одной строки кода.

И вполне возможно, что в только что созданный вами и готовый к применению класс будут и далее вноситься изменения. Как правило, эти изменения будут настолько незначительными и очевидными, что вы даже не почувствуете потребности проводить такое же тщательное тестирование, как и на стадии разработки. Вы, скорее всего, вообще забудете о внесенных изменениях, если только не введете соответствующие комментарии в конце файла класса, как я иногда поступаю. Но даже незначительные измене-

ния могут привести к серьезным непредвиденным последствиям, которые можно предвидеть, если умело пользоваться средствами тестирования.

Средства тестирования — это набор автоматических тестов, которые можно применить к системе в целом или к отдельным ее классам. Правильно использованные средства тестирования помогают не допустить появления и *повторения* ошибок. Единственное изменение может привести к целому каскаду ошибок, и средства тестирования позволяют обнаружить и устранить этот недостаток. Это означает, что изменения можно вносить с некоторой уверенностью ничего не нарушить. Возможность внести изменения в систему и увидеть список неудачно завершившихся тестов приносит удовлетворение. Ведь все обнаруженные в итоге ошибки могли распространиться по системе, но теперь она не пострадает.

Непрерывная интеграция

Приходилось ли вам когда-нибудь планировать свою работу, которая была успешно и своевременно выполнена? Вы начинали с задания — это мог быть программный проект или домашнее задание в школе. Поначалу оно казалось большим, ужасным и обреченным на провал. Но вы брали лист бумаги и делили задание на поддающиеся выполнению части, определяя литературу, которую нужно прочитать, или компоненты, которые следует написать. Возможно, вы выделяли отдельные задания разным цветом. Взятые по отдельности они уже не казались такими ужасными, и вы чувствовали, что в состоянии их выполнить. И так, постепенно выполняя запланированную работу, вы продвигались к сроку ее сдачи. Если вы делали что-нибудь хотя бы понемногу каждый день, то все шло замечательно и в конце вы могли немного расслабиться.

Но иногда составленный план работ приобретает силу талисмана. Вы придерживаетесь его как защиты от сомнений и страха, что в какой-то момент все может неожиданно пойти прахом. И только по прошествии нескольких недель вы осознаете, что в самом плане нет ничего чудесного. Вы просто должны выполнить свою работу. К тому времени под влиянием внушающей силы составленного плана работ вы допускали, чтобы все шло своим чередом. И тогда вам не оставалось ничего другого, как составить новый план, но уже меньше веря в его всемогущую силу.

По существу, тестирование и построение проекта очень похоже на планирование работ. В частности, необходимо выполнять тесты, строить про-

екты, а затем снова и снова перестраивать их в новых условиях, иначе ничего не получится как по волшебству.

И если писать тесты хлопотно, то выполнять их — не менее хлопотно, особенно когда они усложняются, а их "непрохождение" нарушает планы. Безусловно, если выполнять тесты чаще, то их непрохождение, вероятнее всего, будет происходить реже и иметь отношение только к недавно написанному коду.

Достичь определенных удобств в своей "песочнице" нетрудно, если под рукой есть все необходимые "игрушки" вроде мелких и облегчающих жизнь сценариев, средств разработки и полезных библиотек. Но беда в том, что вашему проекту может быть слишком уютно в "песочнице". В нем могут использоваться незавершенные фрагменты кода или зависимости, которые вы забыли включить в файл построения. Это означает, что построение проекта завершится неудачей в любой другой среде, кроме той, в которой вы работаете. Единственный выход состоит в том, чтобы тщательно и многократно строить проект и проверять его, причем каждый раз это нужно делать в относительно "первозданной" среде.

Но одно дело — посоветовать, а совсем другое дело — взять и сделать! Программистам исконно нравится сам процесс написания кода. Они всегда стремятся свести все организационные вопросы к минимуму. И здесь им на помощь приходит *непрерывная интеграция*. Это одновременно методика и набор инструментальных средств, облегчающих, насколько это возможно, внедрение в жизнь этой методики. В идеальном случае процессы построения и тестирования должны быть полностью автоматизированы или хотя бы должны запускаться после ввода одной команды или щелчка кнопкой мыши. При этом любые затруднения будут зафиксированы и о них будет сообщено, прежде чем произойдет что-нибудь серьезное. Подробнее о непрерывной интеграции речь пойдет в главе 21, "Непрерывная интеграция".

Резюме

У разработчика всегда одна и та же цель — выпустить работоспособную систему, однако написание качественного кода — необходимое, но недостаточное условие для ее достижения.

В этой главе было дано краткое введение в управление зависимостями средствами Composer и Packagist. В ней также были рассмотрены два заме-

чательных вспомогательных средства для коллективной разработки программного обеспечения: система контроля версий и технология Vagrant. Кроме того, здесь пояснялось, что для контроля версий требуется автоматическое средство построения, подобное `Phing` — реализации на PHP средства построения `Ant`, исходно написанного на языке Java. И в конце главы обсуждались вопросы тестирования программ, а также было введено новое понятие непрерывной интеграции как средства, автоматизирующего процесс написания и тестирования программ.

ГЛАВА 15

Стандарты PHP

Если вы не юрист и не санитарный инспектор, то обсуждение стандартов вряд ли заставит ваше сердце биться учащенно. Тем не менее стандарты помогают достичь впечатляющих результатов, способствуют взаимозаменяемости и как следствие позволяют получить доступ к обширному ряду совместимых компонентов инструментальных средств и каркасов.

В этой главе будут рассмотрены следующие вопросы, касающиеся стандартов.

- *Зачем нужны стандарты.* Что такое стандарты и зачем они нужны.
- *Рекомендованные стандарты PHP.* Их происхождение и назначение.
- *PSR-1.* Основной стандарт программирования.
- *PSR-12.* Руководство по стилю программирования.
- *PSR-4.* Автозагрузка.

Зачем нужны стандарты

Проектные шаблоны совместимы; это заложено в их основе. Задача, описанная в проектном шаблоне, предполагает конкретное решение, которое, в свою очередь, приводит к определенным архитектурным последствиям, вполне разрешимым в других шаблонах. Кроме того, шаблоны помогают разработчикам эффективно взаимодействовать, поскольку они предоставляют для этой цели общий словарь. Объектно-ориентированные системы обладают привилегией вести себя хорошо.

Но по мере того, как разработчики все чаще обмениваются компонентами, такого неформального стремления к взаимодействию не всегда оказывается достаточно. Как пояснялось ранее, Composer (или другая система управления пакетами) позволяет применять разные инструментальные средства в разрабатываемых проектах. Ведь компоненты могут быть разработаны как самостоятельные библиотеки или входить в состав более обширных каркасов. Но в любом случае, как только система будет развернута, ее компоненты должны быть в состоянии работать как отдельно, так

и вместе с любым количеством других компонентов. Придерживаясь основных стандартов, мы можем уменьшить риск возникновения трудноразрешимых вопросов совместимости.

В каком-то смысле характер стандарта менее важен, чем его соблюдение. Мне лично нравится далеко не всё, что указано в рекомендациях стандарта PSR-12 относительно стиля программирования, хотя в большинстве случаев (в том числе в этой книге) я принимаю их как стандарт. При этом я надеюсь, что другим разработчикам из моей команды будет легче работать с моим кодом, поскольку он оформлен в привычном для них стиле. Что же касается других стандартов, включая автозагрузку, то несоблюдение общего стандарта способно привести к тому, что компоненты вообще не смогут работать вместе без дополнительного промежуточного программного обеспечения, служащего в качестве связующего звена.

Стандарты вряд ли можно считать самым привлекательным аспектом программирования. Но в их основе лежит любопытное противоречие. На первый взгляд стандарт сдерживает стремление к творчеству. Ведь стандарты предписывают, что можно и чего нельзя делать. А поскольку их приходится соблюдать, то может показаться, что они едва ли способствуют новшествам. Тем не менее заметный расцвет творчества, благодаря которому Интернет вошел в нашу жизнь, обязан тому обстоятельству, что каждый узел сети действует в строгом соответствии с имеющимися открытыми стандартами. Нестандартные, оригинальничающие системы вынуждены оставаться в пределах ограниченной области действия, а зачастую и ограниченной продолжительности существования, каким бы совершенным ни был их исходный код, а интерфейсы — искусными. Благодаря общепринятым сетевым протоколам Интернет гарантирует, что любой сайт может связаться с любым другим сайтом. В большинстве браузеров поддерживаются стандарты HTML, CSS и JavaScript. Интерфейсы, которые могут быть построены в соответствии с этими стандартами, не всегда впечатляют, хотя накладываемые на них ограничения не столь значительны, как прежде. Однако соблюдение стандартов делает результаты трудов разработчиков максимально доступными.

Если умело пользоваться стандартами, они будут способствовать открытости, сотрудничеству и в конечном итоге — творчеству. И это справедливо, даже если сам стандарт действует с некоторыми ограничениями.

Рекомендованные стандарты PHP

На конференции php[tek] в 2009 году группа разработчиков каркасов учредила организацию под названием "PHP-FIG" (PHP Framework Interop Group — Группа по взаимодействию каркасов PHP). С тех пор в эту организацию вступили разработчики компонентов из других ключевых областей. Их целью стало составление таких стандартов, которые могли бы обеспечить лучшее сосуществование их систем. Группа PHP-FIG ставит на голосование предлагаемые стандарты, которые проходят стадии проекта, рассмотрения и, наконец, принятия. Перечень стандартов, принятых группой PHP-FIG на момент написания данной книги, приведен в табл. 15.1.

Таблица 15.1. *Принятые рекомендованные стандарты PHP*

Номер PSR	Наименование	Описание
1	Basic Coding Standard (основной стандарт программирования)	Основные положения, касающиеся, например, дескрипторов PHP и основных соглашений об именовании
2	Coding Style Guide (руководство по стилю программирования)	Форматирование исходного кода, включая правила размещения фигурных скобок, списков аргументов и т.д.
3	Logger Interface (интерфейс регистраторов)	Правила определения уровней протоколирования и режимов работы регистраторов
4	Autoloading Standard (стандарт автозагрузки)	Соглашения об именовании имен классов и пространств имен, а также их отображение на файловую систему
6	Caching Interface (интерфейс кеширования)	Правила управления кешированием, включая типы данных, срок действия элементов в кеше, обработку ошибок и т.д.
7	HTTP Message Interfaces (интерфейсы HTTP-сообщений)	Соглашения для HTTP-запросов и ответов
11	Container interface (интерфейс контейнера)	Общий интерфейс для контейнеров зависимостей инъекций
12	Extended Coding Style Guide (руководство по расширенному стилю кодирования)	Форматирование кода, включая правила размещения скобок, списков аргументов и т.д.

Окончание табл. 15.1

Номер PSR	Наименование	Описание
13	Link definition interfaces (интерфейсы определения связей)	Интерфейсы для описания ссылок HyperMedia
14	Event dispatcher (диспетчер событий)	Определения для управления событиями
15	HTTP Handlers (обработчики HTTP)	Общие интерфейсы для обработчиков запросов к HTTP-серверу
16	Simple Cache (простой кеш)	Общий интерфейс библиотек кеширования библиотек (упрощение PSR-6)
17	HTTP Factories (фабрики HTTP)	Общий стандарт для фабрик, которые создают HTTP-объекты, соответствующие стандарту PSR-7
18	HTTP Client (клиент HTTP)	Интерфейс для отправки HTTP-запросов и получения HTTP-ответов

Особенности рекомендованных стандартов PSR

Так как же выбрать тот или иной стандарт? Оказывается, что организация PHP Framework Interop Group, являющаяся инициатором использования рекомендованных стандартов PSR, имеет довольно солидный послужной список, а следовательно, в ее стандартах есть свой смысл. Но имеются и такие стандарты, которые приняты в большинстве каркасов и компонентов. Если вы пользуетесь диспетчером зависимостей Composer для внедрения функциональных средств в свои проекты, то уже применяете код, соответствующий рекомендациям стандарта PSR. Используя соглашения по автозагрузке и рекомендации по стилевому оформлению, принятые в Composer, вы, вероятнее всего, строите код, готовый для сотрудничества с другими разработчиками и взаимодействия с другими компонентами.

Рекомендации стандартов PSR являются удачным выбором потому, что они поддерживаются в ключевых проектах каркасов и компонентов, включая Phing, Composer, PEAR, Symfony и Zend 2. Как и проектные шаблоны, стандарты заразительны, а следовательно, вы, вероятно, уже извлекаете из них выгоду.

На заметку Один набор стандартов не превосходит другой их набор. Когда приходится решать, стоит ли принимать стандарт, конкретный выбор может зависеть от оценки достоинств рекомендаций стандартов. С другой стороны, можно сделать прагматичный выбор исходя из того контекста, в котором приходится работать. Так, если вы работаете на платформе WordPress, то можете принять стиль оформления кода, определенный в документе *Core Contributor Handbook* (Справочник основного участника проекта; см. `https://make.wordpress.org/core/handbook/best-practices/coding-standards/php/`). Такой выбор отчасти объясняется назначением стандартов, призванных способствовать сотрудничеству разработчиков и взаимодействию программного обеспечения.

На кого рассчитаны рекомендации стандартов PSR

На первый взгляд, рекомендации стандартов PSR рассчитаны на создателей каркасов. Но вследствие того, что группа PHP-FIG быстро расширила свой круг участников, включая создателей инструментальных средств, а не только каркасов, ее стандартам нашлось более широкое применение. Так, если только вы не создаете регистратор, вас могут и не интересовать подробности рекомендаций стандарта PSR-3, помимо совместимости с ним любого инструментального средства, применяемого вами для протоколирования. Но если вы собираетесь прочитать остальную часть данной книги, то, скорее всего, вы не только создаете инструментальные средства, но и используете их. Следовательно, в действующих или будущих стандартах вы можете найти нечто подходящее для себя.

Кроме того, существуют стандарты, имеющие значение для всех разработчиков. И хотя они не столь привлекательны, как, например, рекомендации по стилю программирования, они касаются каждого программиста. Несмотря на то что правила автозагрузки действительно касаются лишь тех, кто создает автозагрузчики, а главная роль в этом, вероятно, принадлежит диспетчеру зависимостей Composer, они, по существу, оказывают также влияние на порядок организации классов, пакетов и файлов. Именно по этим причинам мы уделим основное внимание стилю программирования и автозагрузке в остальной части этой главы.

Программирование в избранном стиле

Обнаружив в запросе на включение исходного кода комментарии, подобные "фигурные скобки в вашем коде находятся не на том месте", я нахожу их непомерно раздражающими. Такие комментарии кажутся придирчивыми и опасливо похожими на волокитство, создающее *метафору закона тривиальности* (bike-shedding).

На заметку Если вас интересует происхождение термина *метафора закона тривиальности*, то он обозначает стремление некоторых экспертов критиковать неважные элементы обсуждаемого проекта. Дело в том, что такие элементы специально выбираются исходя из уровня компетентности комментаторов. Если такому эксперту предоставят для оценивания проект небоскреба, он может уделить свое внимание не сложной конструкции башни из стекла, стали и бетона, а намного более простому сооружению вроде стоянки для велосипедов на задворках этого здания. Любопытное описание ситуации можно найти в Википедии по адресу `https://ru.wikipedia.org/wiki/Закон_тривиальности`.

И мне еще нужно убедиться, что соблюдение общего стиля может оказать помощь в повышении качества исходного кода. Это главным образом относится к удобочитаемости исходного кода независимо от обоснования отдельного правила. Если команда разработчиков подчиняется одним и тем же правилам оформления отступов, размещения фигурных скобок, списков аргументов и так далее, то любой из разработчиков может очень быстро оценить исходный код своего коллеги и поучаствовать в его написании.

Таким образом, при подготовке настоящего издания книги я решил отредактировать все примеры исходного кода, чтобы привести их в соответствие рекомендациям стандартов PSR-1 и PSR-12. И я попросил моего коллегу и технического рецензента Пола Трегоинга (Paul Tregoing) поддержать меня в этом намерении. Когда я только планировал редактирование, оно казалось мне нетрудным для реализации, но в действительности оно потребовало намного бо́льших усилий, чем предполагалось изначально. И тогда я получил свой первый урок стилевого оформления исходного кода: стандарт следует принять на как можно более ранней стадии разработки проекта. Реорганизация исходного кода в соответствии с выбранным стилем его оформления, вероятнее всего, приведет к связыванию ресурсов и затруднит анализ отличий в коде, которые могут восходить к временам Великой Реформации. Так какие же изменения следует внести в исходный код? Начнем рассмотрение этого вопроса с самых основ.

Основные рекомендации стандарта PSR-1 по стилю программирования

В рекомендациях стандарта PSR-1 содержатся основные положения для стилевого оформления исходного кода на PHP. Подробнее с ними можно ознакомиться по адресу `http://www.php-fig.org/psr/psr-1/`. В следующих разделах приведено краткое описание рекомендаций стандарта PSR-1.

Открывающие и закрывающие дескрипторы

Прежде всего, раздел PHP включаемого файла должен начинаться с открывающего дескриптора <?php или <?=. Иными словами, вначале этого раздела нельзя употреблять ни более краткий открывающий дескриптор <?, ни какую-нибудь другую его разновидность. А завершаться раздел PHP должен только закрывающим дескриптором ?> (или вообще без дескриптора).

На заметку PSR используют такие термины, как "следует" (should) и "обязан" (must), которые определяют степень соответствия директиве. Точное их значение в контексте PSR определено по адресу `www.ietf.org/rfc/rfc2119.txt`.

Побочные эффекты

В исходном файле PHP должны быть объявлены классы, интерфейсы, функции и прочие элементы исходного кода либо выполнено конкретное действие (например, чтение или запись в файл или же отправка выводимого результата в браузер), но не следует делать одновременно и то, и другое. Если вы привыкли пользоваться функцией `require_once()` для включения других файлов, это собьет вас с верного пути, поскольку включение другого файла имеет побочный эффект. Подобно тому как одни проектные шаблоны порождают другие шаблоны, для одних стандартов требуются другие стандарты. Поэтому для правильного обращения с зависимостями классов следует применять автозагрузчик, соответствующий рекомендациям стандарта PSR-4.

Насколько допустимо для объявляемого вами класса выполнять запись в файл в одном из его методов? Это вполне допустимо, поскольку включение файла не даст никакого отрицательного эффекта. Иными словами, его выполнение не приведет к побочным эффектам.

В какого же рода файле можно выполнять действия, а не объявлять классы? К такого рода файлу следует отнести сценарий, запускающий веб-приложение. Ниже приведен пример кода, в котором действия выполняются как прямое следствие включения файла:

```
// Листинг 15.1
namespace popp\ch15\batch01;
require_once(__DIR__ . "/../../../vendor/autoload.php");
$tree = new Tree();
print "Загружен " . get_class($tree) . "\n";
```

В следующем примере представлен исходный файл PHP, в котором класс объявлен без побочных эффектов:

```
// Листинг 15.2
namespace popp\ch15\batch01;
class Tree
{
}
```

На заметку В других главах данной книги объявления пространств имен в объявлениях `namespace` опускаются для того, чтобы вы могли сосредоточить все внимание на самом коде. Но поскольку в этой главе рассматривается механизм форматирования файлов классов, приведенные в ней примеры кода включают инструкции `namespace` и `use` там, где это уместно.

Соглашение об именовании

Имена классов должны быть заданы в смешанном написании, иначе говоря — "верблюжьим" стилем, или стилем Pascal. При таком написании имя класса должно начинаться с прописной буквы, а остальная его часть должна быть обозначена строчными буквами, если только оно не состоит из нескольких слов. В последнем случае каждое слово в имени класса должно начинаться с прописной буквы:

```
class MyClassName
```

Для имен свойств можно выбрать любой стиль, хотя после этого его нужно будет неукоснительно придерживаться. Я предпочитаю использовать верблюжью форму записи так же, как и для имен классов, но при этом первую букву оставляю строчной:

```
private $myPropertyName
```

Методы должны быть объявлены в смешанном написании следующим образом:

```
public function myMethodName()
```

Константы следует объявлять в классе прописными буквами, разделяя слова знаками подчеркивания, как показано ниже:

```
const MY_NAME_IS = 'matt';
```

Другие правила и примеры

Классы, имена пространств и файлов должны объявляться в соответствии с рекомендациями стандарта PSR-4 для автозагрузки, к которым мы еще вернемся в этой главе. Документы PHP должны сохраняться в файлах в кодировке UTF-8.

Наконец, следуя рекомендациям стандарта PSR-1, сделаем сначала все неправильно, а затем — правильно. В качестве примера ниже приведен файл класса, нарушающий все правила:

```
// Листинг 15.3
<?
require_once("conf/ConfFile.ini");
class conf_reader {
    const ModeFile = 1;
    const Mode_DB = 2;
    private $conf_file;
    private $confValues = [];
    function read_conf() {
        // Реализация
    }
}
?>
```

Обратите внимание на ошибки в объявлении данного класса. Прежде всего, в его объявлении используется краткий открывающий дескриптор. Кроме того, здесь не объявлено пространство имен, хотя мы еще не рассматривали это требование подробно. В именовании данного класса используются знаки подчеркивания вместо прописных букв и смешанного написания, а также две формы обозначения имен констант, ни одна из которых не соответствует требованию объявлять их прописными буквами, разделяя слова знаками подчеркивания. И хотя имена обоих объявленных здесь свойств вполне допустимы, они обозначены несогласованно. В част-

ности, в имени свойства `$conf_file` используется знак подчеркивания, а в имени свойства `$confValues` — смешанное написание. И, наконец, в имени метода `read_conf()` используется знак подчеркивания вместо смешанного написания.

Теперь устраним описанные выше недостатки в объявлении данного класса:

```
// Листинг 15.4
<? php
namespace popp\ch15\batch01;
class ConfReader
{
    const MODEFILE = 1;
    const MODE_DB = 2;
    private $conf_file;
    private $confValues = [];
    function readConf()
    {
        // Реализация
    }
}
?>
```

Рекомендации стандарта PSR-12 по стилю программирования

Рекомендации стандарта PSR-12 по стилю программирования основываются на рекомендациях стандарта PSR-1. В следующих разделах вкратце описаны некоторые правила из этих рекомендаций.

Начало и окончание документа PHP

Как было показано выше, в рекомендациях стандарта PSR-1 требуется, чтобы блоки кода PHP начинались открывающим дескриптором <?php. А в рекомендациях стандарта PSR-12 оговаривается, что файлы, содержащие только исходный код PHP, должны завершаться не закрывающим дескриптором ?>, а одинарной пустой строкой. Ведь такой файл очень легко завершить закрывающим дескриптором, а затем добавить в него лишние символы новой строки. И это может привести к ошибкам форматирования, а также к ошибкам, которые возникают при определении HTTP-заголовков (это нельзя делать после того, как содержимое уже было отправлено браузеру).

В табл. 15.2 описаны элементы, могущие образовывать корректный PHP-документ.

Таблица 15.2. *Принятые рекомендованные стандарты PHP*

Элемент	Пример
Открывающий дескриптор	<?php
Блок документации уровня файла	/** * Документация */
Объявление	declare(strict_types=1)
Объявление пространства имен	namespace popp;
Инструкция импорта (классы)	use other\Service;
Инструкция импорта (функции)	use function other\{getAll, calculate}
Инструкция импорта (константы)	use constant other\{NAME, VERSION}

Документ PHP должен следовать структуре в табл. 15.2 (хотя любые элементы, которые не нужны PHP-коду, могут быть опущены). После объявлений пространств имен в операторах namespace, а также блока объявлений в операторах use должна следовать пустая строка. В одной строке кода должно присутствовать лишь *одно* объявление в операторе use, как показано ниже:

```
// Листинг 15.5
namespace popp\ch15\batch01;

use popp\ch10\batch06\PollutionDecorator;
use popp\ch10\batch06\DiamondDecorator;
use popp\ch10\batch06\Plains;

// Начало класса
```

Начало и окончание объявления класса

Ключевое слово class, имя класса и ключевые слова extends и implements должны быть размещены в одной и той же строке кода. Если в классе реализуется несколько интерфейсов, имя каждого интерфейса может быть указано в той же строке кода, в которой указано объявление класса, или же с отступом в отдельной строке кода. Если же выбрать раз-

мещение имен интерфейсов в нескольких строках кода, то первый элемент должен располагаться в отдельной строке, а не сразу после ключевого слова `implements`. Открывающие фигурные скобки должны быть указаны с новой строки *после* объявления класса, а закрывающие фигурные скобки — также с новой строки, но сразу же после содержимого класса. Таким образом, объявление класса может выглядеть следующим образом:

```
// Листинг 15.6
class EarthGame extends Game implements
    Playable,
    Savable
{
    // Тело класса
}
```

Имена интерфейсов могут также быть указаны в одной строке следующим образом:

```
// Листинг 15.7
class EarthGame extends Game implements Playable, Savable
{
    // Тело класса
}
```

Работа с трейтами

При добавлении трейта к классу вы должны добавить инструкцию `use` сразу после открывающей фигурной скобки класса. Хотя PHP позволяет сгруппировать трейты в одну строку, PSR-12 требует, чтобы вы помещали каждую инструкцию `use` в отдельную строку. Если ваш класс предоставляет собственные элементы в дополнение к инструкциям `use`, вы должны оставить пустую строку перед тем, как приступить к работе с содержимым, не связанным с трейтами. В противном случае вы обязаны закрыть блок класса в строке, следующей сразу за последней инструкцией `use`.

Вот класс, который импортирует два трейта и предоставляет собственный метод:

```
// Листинг 15.8
namespace popp\ch15\batch01;
class Tree
{
    use GrowTools;
    use TerrainUtil;
```

```
    public function draw(): void
    {
        // Реализация
    }
}
```

Если вы объявляете блок для инструкций `as` или `insteadof`, он должен распространяться на несколько строк. Открывающая скобка должна начинаться в той же строке, что и инструкция `use`. Затем следует использовать по одной строке для каждой инструкции. Наконец, закрывающая скобка должна заканчиваться в своей отдельной строке, например:

```
// Листинг 15.9
namespace popp\ch15\batch01;
class Marsh
{
    use GrowTools
    {
        GrowTools::dimension as size;
    }
    use TerrainUtil;
    public function draw(): void
    {
        // Реализация
    }
}
```

Объявление свойств и констант

Свойства и константы должны иметь объявленную область видимости, обозначаемую ключевым словом `public`, `private` или `protected`, тогда как употребление ключевого слова `var` не допускается. Формы обозначения имен свойств и констант уже рассматривались выше как часть рекомендаций стандарта PSR-1.

Начало и окончание объявлений методов

Все методы должны иметь объявленную область видимости, обозначаемую ключевым словом `public`, `private` или `protected`. Ключевое слово, обозначающее область видимости, должно следовать *после* ключевого слова `abstract` или `final`, но *перед* ключевым словом `static`. Аргументы методов со стандартными значениями должны быть указаны в конце списка аргументов.

Объявления методов в одной строке

Открывающие фигурные скобки должны быть указаны с новой строки *после* объявления метода, а закрывающие фигурные скобки — также с новой строки, но сразу же после тела метода. Список аргументов не должен начинаться или оканчиваться пробелом (т.е. они должны указываться вплотную к открывающим или закрывающим круглым скобкам). При указании каждого аргумента запятая должна следовать сразу после имени аргумента (или стандартного значения), но после нее должен быть указан пробел. Поясним это на конкретном примере:

```
// Листинг 15.10
final public static
        function generateTile(int $diamondCount,
                              bool $polluted = false): array
{
    // Реализация
}
```

Многострочные объявления методов

Объявлять методы в одной строке кода непрактично, если у метода имеется много аргументов. В таком случае список аргументов можно разделить таким образом, чтобы каждый аргумент, включая тип, имя аргумента, стандартное значение и запятую, располагался в отдельной строке кода, а закрывающая круглая скобка — в отдельной строке кода после списка аргументов, причем вплотную к началу объявления метода. Открывающая фигурная скобка должна быть указана после закрывающей круглой скобки в той же строке кода и отделена от нее пробелом. Тело метода должно начинаться с новой строки. Поскольку это описание может показаться сложным, для ясности приведем конкретный пример объявления метода в нескольких строках кода:

```
// Листинг 15.11
public function __construct(
    int $size,
    string $name,
    bool $wraparound = false,
    bool $aliens = false
) {
    // Реализация
}
```

Возвращаемые типы

Объявление типа возвращаемого значения должно находиться в той же строке, что и закрывающая скобка. Двоеточие должно следовать сразу за закрывающей круглой скобкой. Двоеточие следует отделять от типа возвращаемого значения одним пробелом. Для многострочных объявлений объявление типа возвращаемого значения должно предшествовать открывающей скобке в той же строке, разделенной пробелом:

```
// Листинг 15.12
final public static function findTilesMatching(
    int $diamondCount,
    bool $polluted = false
): array {
    // Реализация
}
```

PSR-12 не требует обязательного использования объявлений возвращаемого типа. Однако из-за введения типов `void`, смешанных и могущих принимать значение `null`, должна быть возможность предоставления объявления, соответствующего всем обстоятельствам.

Строки кода и отступы

Для обозначения отступа следует использовать четыре пробела вместо знака табуляции. С этой целью необходимо проверить установки в применяемом текстовом редакторе, настроив его на ввод пробелов вместо знака табуляции при нажатии клавиши <Tab>. Кроме того, длина строки текста не должна превышать 120 символов (хотя это и не является обязательным требованием). Строки должны заканчиваться символами перевода строки Unix, а не другими комбинациями, зависящими от платформы (например, CR — в Mac и CR/LF — в Windows).

Проверьте настройки своего редактора, поскольку он, скорее всего, будет использовать символы окончания строки по умолчанию для вашей операционной системы.

Вызов методов и функций

Между именем метода и открывающей круглой скобкой *нельзя* вводить пробел. Правила указания списка аргументов в объявлении метода распространяются и на список аргументов в вызове метода. Иными словами, если вызов указывается в одной строке кода, то после открывающей кру-

глой скобки или прежде закрывающей круглой скобки не нужно оставлять пробел. Запятая должна быть указана сразу же после каждого аргумента, а перед следующим аргументом — один пробел. Если же вызов метода требуется разместить в нескольких строках кода, каждый аргумент следует указать с отступом в отдельной строке, а закрывающую круглую скобку — с новой строки, как показано ниже:

```
// Листинг 15.13
$earthgame = new EarthGame(
    5,
    "earth",
    true,
    true
);
$earthgame::generateTile(5, true);
```

Управление потоком выполнения программы

После ключевых слов, используемых для управления потоком выполнения команд (`if`, `for`, `while` и т.д.), должен следовать один пробел, но только не после открывающей круглой скобки. Аналогично пробела не должно быть перед закрывающей круглой скобкой. Тело управляющего оператора должно быть заключено в фигурные скобки. В отличие от объявлений классов и функций (в одной строке кода), фигурная скобка, открывающая тело управляющего оператора, должна быть указана в той же строке кода, что и закрывающая круглая скобка. А фигурная скобка, закрывающая тело управляющего оператора, должна быть указана с новой строки. Ниже приведен краткий пример обозначения управляющего оператора:

```
// Листинг 15.14
$tile = [];

for ($x = 0; $x < $diamondCount; $x++) {
    if ($polluted) {
        $tile[] = new PollutionDecorator(
                new DiamondDecorator(new Plains()));
    }
    else
    {
        $tile[] = new DiamondDecorator(new Plains());
    }
}
```

Обратите внимание на пробел после управляющих операторов `for` и `if` и непосредственно перед открывающими круглыми скобками. В обоих случаях после закрывающей круглой скобки следует сначала пробел, а затем — фигурная скобка, открывающая тело управляющего оператора.

Выражения в круглых скобках могут быть разделены на несколько строк, каждая строка — хотя бы с одним отступом. Если выражения разделены, логические операторы могут располагаться либо в начале, либо в конце каждой строки, но ваш выбор должен быть последовательным:

```
// Листинг 15.15
$ret = [];

for (
    $x = 0;
    $x < count($this->tiles);
    $x++
) {
    if (
        $this->tiles[$x]->isPolluted() &&
        $this->tiles[$x]->hasDiamonds() &&
        !($this->tiles[$x]->isPlains())
    )
    {
        $ret[] = $x;
    }
}

return $ret;
```

Проверка и исправление исходного кода

Несмотря на то что в этой главе рекомендации стандарта PSR-12 описываются по отдельности, держать их все в уме очень трудно. Ведь разработчикам приходится больше думать о проектировании и реализации разрабатываемых систем, а не о форматировании кода. Но если согласиться, что стандарты на стиль программирования действительно ценны, то как в таком случае соблюдать их, не уделяя им достаточно времени и внимания? Разумеется, для этого можно воспользоваться подходящим инструментальным средством.

В частности, инструментальное средство PHP_CodeSniffer позволяет обнаруживать и даже исправлять нарушения принятых стандартов на стиль программирования, причем не только рекомендаций стандартов PSR. Его можно загрузить, следуя инструкциям, приведенным по адресу

`https://github.com/squizlabs/PHP_CodeSniffer`. Другие возможности предоставляются в Composer и PEAR, но в данном случае можно загрузить архивные файлы PHP следующим образом:

```
curl -OL https://squizlabs.github.io/PHP_CodeSniffer/phpcs.phar

curl -OL https://squizlabs.github.io/PHP_CodeSniffer/phpcbf.phar
```

Зачем загружать два архивных файла? Первый архив (`phpcs`) служит для диагностики и сообщения об обнаруженных нарушениях, а второй архив (`phpcbf`) — для устранения большинства из них. Проверим возможности инструментального средства `PHP_CodeSniffer` на следующем примере неверно отформатированного фрагмента кода:

```
// Листинг 15.16
namespace popp\ch15\batch01;
class ebookParser {
    function __construct(string $path , $format = 0) {
        if ($format>1)
            $this->setFormat( 1 );
    }
    function setformat(int $format) {
        // Некоторые действия с $format
    }
}
```

Вместо того чтобы устранять недостатки в стилевом оформлении приведенного выше фрагмента кода, поручим эту обязанность инструментальному средству `PHP_CodeSniffer`, выполнив следующую команду:

```
$ php phpcs.phar --standard=PSR12 src/ch15/batch01/phpcsBroken.php
```

В итоге будет получен следующий результат:

```
$ php phpcs.phar --standard=PSR12 src/ch15/batch01/phpcsBroken.php
FILE: /var/popp/src/ch15/batch01/phpcsBroken.php
----------------------------------------------------------------------
FOUND 16 ERRORS AFFECTING 6 LINES
----------------------------------------------------------------------
 5 | ERROR | [x] Header blocks must be separated by a single
               blank line
 6 | ERROR | [ ] Class name "ebookParser" is not in PascalCase format
 6 | ERROR | [x] Opening brace of a class must be on the line after
               the definition
 8 | ERROR | [ ] Visibility must be declared on method "__construct"
 8 | ERROR | [x] Expected 0 spaces between argument "$path" and comma;
               1 found
```

```
 8 | ERROR | [x] Incorrect spacing between argument "$format" and
               |     equals sign; expected 1 but found 0
 8 | ERROR | [x] Incorrect spacing between default value and equals
               |     sign for argument "$format"; expected 1 but found 0
 8 | ERROR | [x] Expected 0 spaces before closing parenthesis; 1 found
 8 | ERROR | [x] Opening brace should be on a new line
 9 | ERROR | [x] Inline control structures are not allowed
 9 | ERROR | [x] Expected at least 1 space before ">"; 0 found
 9 | ERROR | [x] Expected at least 1 space after ">"; 0 found
10 | ERROR | [x] Space after opening parenthesis of function call
               |     prohibited
10 | ERROR | [x] Expected 0 spaces before closing parenthesis; 1 found
13 | ERROR | [ ] Visibility must be declared on method "setformat"
13 | ERROR | [x] Opening brace should be on a new line
--------------------------------------------------------------------
PHPCBF CAN FIX THE 13 MARKED SNIFF VIOLATIONS AUTOMATICALLY
--------------------------------------------------------------------
Time: 82ms; Memory: 6MB
```

Обратите внимание, как много нарушений стиля программирования обнаружено лишь в нескольких строках кода. Правда, бо́льшую их часть можно устранить автоматически, как следует из приведенного выше результата диагностики данного фрагмента кода. С этой целью достаточно выполнить следующую команду:

```
$ php phpcbf.phar --standard=PSR12 src/ch15/batch01/EbookParser.php

PHPCBF RESULT SUMMARY
--------------------------------------------------------------------
FILE FIXED REMAINING
--------------------------------------------------------------------
/var/popp/src/ch15/batch01/EbookParser.php 13 3
--------------------------------------------------------------------
A TOTAL OF 13 ERRORS WERE FIXED IN 1 FILE
--------------------------------------------------------------------
Time: 96ms; Memory: 6MB
```

Если теперь выполнить файл phpcs снова, то можно обнаружить, что ситуация намного улучшилась:

```
$ php phpcs.phar --standard=PSR2 src/ch15/batch01/EbookParser.php

FILE: /var/popp/src/ch15/batch01/EbookParser.php
--------------------------------------------------------------------
FOUND 3 ERRORS AFFECTING 3 LINES
--------------------------------------------------------------------
7 | ERROR | Class name "ebookParser" is not in PascalCase format
```

```
10 | ERROR | Visibility must be declared on method " construct"
17 | ERROR | Visibility must be declared on method "setformat"
--------------------------------------------------------------------
Time: 76ms; Memory: 6MB
```

Теперь осталось лишь ввести объявления областей видимости для двух методов и правильно обозначить имя класса. В итоге мы получим файл класса, объявление которого совместимо с принятыми нормами стилевого оформления исходного кода:

```
// Листинг 15.17
namespace popp\ch15\batch01;
class EbookParser
{
    public function __construct(string $path, $format = 0) {
        if ($format > 1)        {
            $this->setFormat(1);
        }
    }
    private function setformat(int $format): void {
        // Некоторые действия с $format
    }
}
```

Рекомендации стандарта PSR-4 по автозагрузке

О поддержке автозагрузки в языке PHP речь шла в главе 5, "Средства для работы с объектами", где было показано, как пользоваться функцией spl_autoload_register() для автоматического включения файлов по имени еще незагруженного класса. Несмотря на всю эффективность механизма автозагрузки, его магия действует невидимо для программы. И если это допустимо в одном проекте, то в другом может вызвать немало недоразумений, если в нем применяются многие компоненты, в которых используются разные соглашения по включению файлов классов.

В соответствии с рекомендациями стандарта PSR-4 по автозагрузке требуется, чтобы каркасы отвечали общим правилам, а следовательно, в волшебный механизм автозагрузки должна быть внедрена определенная дисциплина, что вполне устроило бы разработчиков. Это означает, что механизмом автоматического включения файлов можно в той или иной степени пренебречь, уделив основное внимание зависимостям классов.

Самые важные правила

Основное назначение рекомендаций стандарта PSR-4 — установить правила для разработчиков автозагрузчиков. Но эти правила неизбежно определяют порядок объявления пространств имен и классов. Рассмотрим их основные положения.

Полностью определенное имя класса (т.е. имя самого класса, а также наименования его пространств имен) должно включать в себя пространство имен разработчика. Таким образом, у класса должно быть хотя бы одно пространство имен.

Допустим, у класса имеется пространство имен разработчика popp. В таком случае этот класс можно объявить следующим образом:

```
// Листинг 15.18
namespace popp;
class Services
{
}
```

Полностью определенное имя этот класса будет выглядеть как popp\Services. Начальные пространства имен в указанном пути должны соответствовать одному или нескольким базовым каталогам. Мы можем также отобразить ряд вложенных пространств имен на начальный каталог. Так, если требуется работать в пространстве имен popp\library и больше ни в каких вложенных пространствах имен popp, то его можно отобразить на каталог верхнего уровня, чтобы обойтись без создания пустых каталогов в каталоге popp/.

Создадим приведенный ниже файл composer.json, чтобы выполнить подобное отображение:

```
{
    "autoload": {
        "psr-4": {
            "popp\\library\\": "mylib"
        }
    }
}
```

Обратите внимание на то, что в данном случае не указан базовый каталог library. Это результат произвольного отображения пути popp\library на каталог mylib. Теперь в каталоге mylib можно создать файл приведенного ниже класса. Чтобы класс LibraryCatalogue был найден,

он должен быть размещен в файле под точно таким же именем, которое, очевидно, следует дополнить расширением .php:

```
// Листинг 15.19
// mylib/LibraryCatalogue.php

namespace popp\library;
use popp\library\inventory\Book;
class LibraryCatalogue
{
    private array $books = [];
    public function addBook(Book $book): void
    {
        $this->books[] = $book;
    }
}
```

После того как базовый каталог (mylib) будет связан с начальным пространством имен popp\library, у нас появится прямая взаимосвязь между последующими каталогами и вложенными пространствами имен. Но ведь мы уже обращались к классу popp\library\inventory\Book в классе LibraryCatalogue. Следовательно, файл данного класса должен быть размещен в каталоге mylib/inventory, как показано ниже:

```
// Листинг 15.20
// mylib/library/inventory/Book.php
namespace popp\library\inventory;

class Book
{
    // Реализация
}
```

Напомним правило, гласящее, что начальные пространства имен, указанные в пути, должны соответствовать одному или *нескольким* базовым каталогам. До сих пор мы установили взаимно однозначное соответствие пространства имен popp\library и каталога mylib. На самом деле не существует никаких причин, по которым нельзя отобразить пространство имен popp\library на более чем один базовый каталог. Поэтому добавим каталог additional в отображение. Ниже приведен исправленный вариант файла composer.json:

```
{
    "autoload": {
        "psr-4": {
```

```
            "popp\\library\\": ["mylib", "additional"]
        }
    }
}
```

Теперь можно создать каталоги `additional/inventory` для размещения в них класса:

```
// Листинг 15.21
// additional/inventory/Ebook.php
namespace popp\library\inventory;

class Ebook extends Book
{
    // Реализация
}
```

Далее создадим сценарий верхнего уровня в файле `index.php`, чтобы получить экземпляры созданных выше классов:

```
// Листинг 15.22
require_once("vendor/autoload.php");

use popp\library\LibraryCatalogue;

// Будет найден в каталоге mylib/
use popp\library\inventory\Book;

// Будет найден в каталоге additional/
use popp\library\inventory\Ebook;

$catalogue = new LibraryCatalogue();
$catalogue->addBook(new Book());
$catalogue->addBook(new Ebook());
```

На заметку Чтобы создать файл автозагрузки `vendor/autoload.php`, следует воспользоваться диспетчером зависимостей Composer. Кроме того, этот файл нужно каким-то образом явно включить в сценарий, чтобы получить доступ к логике, объявленной в файле `composer.json`. С этой целью можно выполнить команду `composer install` (или `composer dump-autoload`, если вы просто хотите регенерировать файл автозагрузки в среде, которая уже установлена). Подробнее о диспетчере зависимостей Composer речь пойдет в главе 16, "Создание и использование компонентов PHP средствами Composer".

Напомним правило о побочных эффектах. В исходном файле PHP следует либо объявлять классы, интерфейсы, функции и прочие элементы кода, либо выполнять действия, но не то и другое одновременно. Приведенный выше сценарий относится к категории выполнения действий. Очень важно, что в нем вызывается функция `require_once()` для включения кода автозагрузки, сгенерированного на основе файла конфигурации `composer.json`. Благодаря этому все указанные классы будут найдены, несмотря на то что класс `Ebook` был размещен в отдельном от остальных каталоге.

Но зачем поддерживать два отдельных каталога для одного и того же базового пространства имен? К возможным причинам этого, в частности, относится потребность хранить модульные тесты отдельно от рабочего кода. Это позволяет также подключать дополнительные модули и расширения, не поставляемые вместе с каждой версией проектируемой системы.

На заметку Внимательно следите за развитием всех рекомендованных стандартов по адресу `http://www.php-fig.org/psr/`. Эта область бурно развивается, и вы, вероятно, обнаружите, что подходящие вам стандарты еще не устоялись окончательно и по-прежнему находятся на стадии разработки.

Резюме

В этой главе была предпринята скромная попытка показать, что, несмотря на всю непривлекательность стандартов, соблюдать их полезно, и были продемонстрированы их основные возможности. Стандарты разрешают вопросы интеграции, поэтому их можно взять на вооружение и пойти дальше, чтобы творить замечательные вещи. В главе были, в частности, вкратце описаны рекомендации стандартов PSR-1 и PSR-12 по основам и стилю программирования, а также стандарта PSR-4 — по правилам автозагрузки. В конце главы был рассмотрен пример, созданный с помощью Composer и продемонстрировавший организацию автозагрузки по стандарту PSR-4 на практике.

ГЛАВА 16

Создание и использование компонентов PHP средствами Composer

Программисты стремятся создавать повторно используемый код. И в этом состоит одна из главных целей объектно-ориентированного программирования. Им нравится абстрагировать полезные функциональные возможности от неразберихи конкретного контекста, превращая конкретное решение в инструментальное средство, которое может использоваться неоднократно. Если посмотреть на это под другим углом зрения, то программисты любят повторно используемый код и ненавидят дублирующийся. Создавая библиотеки, которые можно применять повторно, программисты стараются избегать необходимости реализовывать сходные решения в разных проектах.

Но даже если удастся избежать дублирования кода, возникнет более обширный вопрос: насколько часто другие программисты реализовывали одно и то же проектное решение в каждом создаваемом инструментальном средстве? По большому счету это напрасная трата усилий. Не лучше ли программистам сотрудничать друг с другом, направляя коллективную энергию на то, чтобы сделать одно инструментальное средство вместо сотен вариаций на одну и ту же тему?

Чтобы сделать это, придется обратиться к услугам существующих библиотек. Но тогда в необходимых нам пакетах, вероятнее всего, потребуются другие библиотеки, чтобы они могли выполнить свои функции. Следовательно, нам понадобится инструментальное средство, способное управлять загрузкой и установкой пакетов, а также зависимостями. Именно здесь и пригодится диспетчер зависимостей Composer, поскольку он выполняет все эти и многие другие функции.

В этой главе рассматриваются следующие основные вопросы.

- *Установка*. Загрузка и установка Composer.
- *Требования*. Применение файла `composer.json` для получения пакетов.

- *Версии.* Указание версий, чтобы получить последний вариант исходного кода, не нарушая работы системы.
- *Сайт Packagist.* Конфигурирование исходного кода, чтобы сделать его общедоступным.
- *Частные хранилища.* Выгодное использование частного хранилища в Composer.

Что такое Composer

Строго говоря, Composer выполняет функции диспетчера зависимостей, а не пакетов. И это, по-видимому, объясняется тем, что он оперирует взаимосвязями компонентов локально, а не централизованно, как, например, `Yum` или `Apt`. И считая такое отличие слишком тонким, вы можете оказаться правы. Тем не менее мы его проводим, принимая во внимание то обстоятельство, что в Composer допускается указание пакетов. Этот диспетчер загружает пакеты в локальный каталог (`vendor`), обнаруживает и загружает все зависимости, а затем делает весь полученный в итоге код доступным в проекте через автозагрузчик.

Чтобы воспользоваться Composer, как и любым другим инструментальным средством, его необходимо сначала получить, а затем установить.

Установка Composer

Загрузить диспетчер зависимостей Composer можно по адресу `https://getcomposer.org/download/`, где находится инсталляционный файл. Выполнить загрузку и инсталляцию можно из командной строки, как показано ниже:

```
$ wget https://getcomposer.org/download/1.8.4/composer.phar
$ chmod 755 composer.phar
$ sudo mv composer.phar /usr/bin/composer
```

После загрузки архивного `phar`-файла выполняется команда `chmod`, делающая его выполняемым. Затем он копируется в общий каталог, чтобы его можно было легко запустить из любого места системы. Далее можно выполнить команду проверки версии Composer следующим образом:

```
$ composer --version
```

Выполнение этой команды приведет к результату наподобие следующего:

```
Composer version 2.0.8 2020-12-03 17:20:38
```

Установка пакетов

Почему здесь речь идет не об одном, а о целом ряде пакетов? А потому, что для работы одних пакетов неизбежно требуются другие пакеты, а иногда и их огромное количество.

Но мы начнем с автономной библиотеки. Допустим, необходимо написать приложение, которое должно взаимодействовать с Twitter. Несложный поиск приводит к пакету `abraham/twitteroath`. Чтобы установить его, придется сформировать файл формата JSON под именем `composer.json`, а затем определить в нем элемент `require` для данного пакета, как показано ниже:

```
{
    "require": {
        "abraham/twitteroauth": "2.0.*"
    }
}
```

Начнем с каталога, в котором не должно быть ничего, кроме файла `composer.json`. Но если выполнить следующую команду Composer, то обнаружится приведенное ниже изменение:

```
$ composer update

Loading composer repositories with package information
Updating dependencies
Lock file operations: 2 installs, 0 updates, 0 removals
  - Locking abraham/twitteroauth (2.0.1)
  - Locking composer/ca-bundle (1.2.8)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.2.8): Extracting archive
  - Installing abraham/twitteroauth (2.0.1): Extracting archive
    Generating autoload files
```

Так что же было сформировано в конечном итоге? Попробуем выяснить это, выполнив следующую команду:

```
$ ls
```

В итоге будет получен следующий результат:

```
composer.json composer.lock vendor
```

Диспетчер зависимостей Composer устанавливает пакеты в каталоге vendor/. Он также формирует файл блокировки composer.lock, в котором указаны конкретные версии всех установленных пакетов. Если применяется система контроля версий, этот файл следует зафиксировать в центральном хранилище. Если другой разработчик выполнит команду composer install при наличии файла блокировки composer.lock, версии пакета будут установлены в его системе именно так, как было указано. Таким образом, члены команды разработчиков могут синхронизировать свои действия, а рабочая среда будет соответствовать средам разработки и тестирования.

Файл блокировки можно перезаписать, выполнив команду composer update еще раз. Выполнение этой команды приведет к формированию нового файла блокировки. Как правило, эта команда выполняется с целью поддерживать актуальность текущих версий пакетов, если указать в ней метасимволы подстановки (как было сделано выше) или заданные диапазоны версий.

Установка пакетов из командной строки

Как было показано выше, в текстовом редакторе можно вручную создать файл composer.json. Но в то же время можно дать Composer команду сделать это автоматически. Это особенно удобно, если требуется запустить приложение на выполнение с единственным пакетом. Когда команда composer require вызывается из командной строки, диспетчер зависимостей Composer загружает указанный в ней файл и автоматически устанавливает его в каталоге vendor/, как показано ниже. Кроме того, он автоматически формирует файл composer.json, который можно будет затем отредактировать и дополнить:

```
$ composer require abraham/twitteroauth
```

Выполнение данной команды приведет к следующему результату:

```
Using version ^2.0 for abraham/twitteroauth
./composer.json has been created
Running composer update abraham/twitteroauth
Loading composer repositories with package information
Updating dependencies
```

```
Lock file operations: 2 installs, 0 updates, 0 removals
  - Locking abraham/twitteroauth (2.0.1)
  - Locking composer/ca-bundle (1.2.8)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.2.8): Extracting archive
  - Installing abraham/twitteroauth (2.0.1): Extracting archive
    Generating autoload files
```

Версии пакетов

Диспетчер зависимостей Composer предназначен для поддержки семантического контроля версий. Это, по существу, означает, что версии пакета обозначаются тремя разделяемыми точками номерами: *основным*, *дополнительным* и номером *исправления* (в виде так называемой "заплатки" ("патча")). Так, если исправляется программная ошибка, но никаких дополнительных функциональных средств не внедряется и обратная совместимость не нарушается, следует увеличить на единицу номер *исправления*. Если же внедряются новые функциональные средства, но обратная совместимость не нарушается, то на единицу следует увеличить указанный средним *дополнительный* номер версии. А если в новой версии нарушается обратная совместимость (иными словами, если внезапный переход на новую версию нарушит нормальное функционирование клиентского кода), то на единицу следует увеличить указанный первым *основной* номер версии.

На заметку Подробнее об условных обозначениях, принятых при семантическом контроле версий, можно узнать по адресу `https://semver.org`.

Описанные выше правила следует непременно иметь в виду, указывая версии в файле `composer.json`. Так, если слишком вольно использовать метасимволы подстановки или диапазоны при указании версий пакетов, нормальная работа системы может нарушиться в результате обновления. Некоторые способы обозначения версий пакетов в Composer перечислены в табл. 16.1.

Таблица 16.1. Обозначение версий пакетов в Composer

Тип	Пример	Примечание
Точный	1.2.2	Установить только указанную версию пакета
Метасимвол подстановки	1.2.*	Установить указанные номера версий, но в то же время найти последнюю доступную версию пакета, совпадающую с метасимволом подстановки
Диапазоны	1.0.0 - 1.1.7	Установить версию пакета не ниже первого номера и не выше последнего
Сравнение	>1.2.0<=1.2.2	Знаки <, <=, > и >= используются для указания сложных критериев диапазонов. Эти знаки можно сочетать с пробелами, что равнозначно логической операции И, или со знаками ||, обозначающими логическую операцию ИЛИ
Тильда (основная версия)	~1.3	Задает минимальный номер версии, а указанный номер последней версии может увеличиться. Таким образом, ~1.3 означает минимальный номер 1.3, а совпадение с версией 2.0.0 и выше невозможно
Символ вставки	^1.3	Обозначает совпадение вплоть до, но не включая, следующего нарушающего изменения. Так, если ~1.3.1 означает несовпадение с версией 1.4 и выше, то ^1.3.1 — это совпадение начиная с версии 1.3.1 и вплоть до версии 2.0.0, но исключая последнюю. Как правило, это самое удобное обозначение версий пакета

На заметку Вы можете дополнительно повлиять на способ выбора пакетов Composer, добавив суффиксы стабильности к строкам ограничений версии. Добавив `@`, за которым следует один из суффиксов `dev`, `alpha`, `beta` и `RC` (от наименее стабильной до наиболее стабильной), вы разрешите Composer учитывать в своих расчетах нестабильные версии. Composer может принимать решения по именам тегов git. Так, `1.2.*@dev` может соответствовать тегу `1.2.2-dev`. Вы также можете использовать флаг стабильности `stable`, чтобы указать, что вы не хотите использовать нестабильный код. Это будет дескриптор версий, которые не являются `dev`, `beta` и т.д.

Поле require-dev

Зачастую на стадии разработки требуются пакеты, которые обычно не нужны на стадии эксплуатации. Допустим, необходимо выполнить тесты локально, но на общедоступном веб-сайте пакет PHPUnit, предназначенный для модульного тестирования, вероятнее всего, не понадобится.

В диспетчере зависимостей Composer эта задача решается благодаря поддержке отдельного элемента `require-dev`, в котором можно указать пакеты таким же образом, как и в элементе `require`:

```
{
    "require-dev": {
        "phpunit/phpunit": "*"
    },
    "require": {
        "abraham/twitteroauth": "^2.0",
        "ext-xml": "*"
   }
}
```

Если теперь выполнить приведенную ниже команду `composer update`, будут загружены и установлены пакет PHPUnit и все зависимые пакеты:

```
$ composer update
```

Выполнение этой команды приведет к следующему результату:

```
Loading composer repositories with package information
        Updating dependencies
Lock file operations: 36 installs, 0 updates, 0 removals
  - Locking abraham/twitteroauth (2.0.1)
  - Locking composer/ca-bundle (1.2.8)
  - Locking doctrine/instantiator (1.4.0)
...
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 36 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.2.8): Extracting archive
  - Installing abraham/twitteroauth (2.0.1): Extracting archive
...
6 package suggestions were added by new dependencies, use `composer
suggest` to see details.
Generating autoload files
```

Если пакеты устанавливаются в рабочей среде, то в команде composer install можно указать параметр --no-dev, и тогда Composer загрузит только те пакеты, которые заданы в поле require, как показано ниже:

```
$ composer install --no-dev
```

Выполнение этой команды приведет к следующему результату:

```
Installing dependencies from lock file
Verifying lock file contents can be installed on current platform.
Package operations: 2 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.2.8): Extracting archive
  - Installing abraham/twitteroauth (2.0.1): Extracting archive
    Generating autoload files
```

На заметку Когда выполняется команда composer install, в текущем каталоге программа Composer создает файл composer.lock, в котором записывается конкретная версия каждого файла, установленного в каталоге vendor/. Если же выполнить команду composer install при наличии файла composer.lock, Composer установит версии пакетов, указанные в этом файле, если они отсутствуют. Это удобно, поскольку вы можете зафиксировать файл composer.lock в глобальном хранилище системы контроля версий в полной уверенности, что ваши коллеги загрузят те же самые версии всех установленных вами пакетов. Если же файл блокировки composer.lock необходимо переопределить, чтобы получить последние версии пакетов или в связи с изменениями в файле composer.json, то придется выполнить команду composer update.

Composer и автозагрузка

Подробно автозагрузка обсуждалась в главе 15, "Стандарты PHP", но ради полноты изложения ее стоит еще раз вкратце рассмотреть. Composer формирует файл autoload.php, организующий загрузку классов для пакетов, загружаемых этим диспетчером зависимостей. Вы можете этим выгодно воспользоваться, включив в свой код файл автозагрузки autoload.php — обычно с помощью оператора require_once(). Как только это будет сделано, любой класс, объявленный в проектируемой системе, будет автоматически найден и загружен в момент первого к нему обращения из вашего кода. Единственное условие — имена используемых вами каталогов и файлов должны соответствовать именам пространств имен и классов. Иными словами, класс poppbook\megaquiz\command\CommandContext

должен быть размещен в файле CommandContext.php, находящемся в каталоге poppbook/megaquiz/command/.

Если же потребуется вмешаться в этот процесс (например, пропустить один или два начальных каталога в искомом пути или ввести в него тестовый каталог), можно будет воспользоваться элементом autoload файла composer.json, чтобы отобразить пространство имен на структуру файлов, как показано ниже:

```
"autoload": {
    "psr-4": {
        "poppbook\\megaquiz\\": ["src", "test"]
    }
}
```

Чтобы сгенерировать последний файл autoload.php, нужно запустить composer install (что также установит все, что указано в файле блокировки) или composer update (при этом будут также установлены последние пакеты, соответствующие спецификации в composer.json). Если вы не хотите устанавливать или обновлять какие-либо пакеты, можете использовать команду composer dump-autoload, которая сгенерирует только файлы автозагрузки.

Теперь, поскольку файл autoload.php включен в проект, его классы нетрудно обнаружить. Благодаря конфигурации автозагрузки класс poppbook\megaquiz\command\CommandContext будет обнаружен в файле src/command/CommandContext.php. А благодаря ссылке на несколько целевых каталогов (test и src) в каталоге test/ также можно будет создать тестовые классы, относящиеся к пространству имен poppbook\megaquiz\. Более подробный пример организации автозагрузки приведен в разделе “Рекомендации стандарта PSR-4 по автозагрузке” главы 15.

Создание собственного пакета

Если вы работали раньше с хранилищем пакетов PEAR, то можете предложить, что в этом разделе будет рассматриваться создание совершенно нового файла пакета. На самом деле мы уже создали пакет в этой главе, поэтому нам остается лишь добавить немного дополнительных сведений о нем, чтобы сделать его доступным для других.

Добавление сведений о пакете

Чтобы сделать пакет жизнеспособным, совсем не обязательно вводить о нем много сведений, но совершенно необходимо указать имя пакета, чтобы его можно было найти. Кроме того, можно включить в эти сведения элементы description и authors, а также создать фиктивный продукт megaquiz, который будет периодически упоминаться в следующих главах:

```
"name": "poppbook/megaquiz",
"description": "a truly mega quiz",
"authors": [
    {
        "name": "matt zandstra",
        "email": "matt@getinstance.com"
    }
],
```

Содержимое элементов не требует особых пояснений, кроме элемента name, в котором указано пространство имен poppbook, отделенное от имени пакета косой чертой. Это так называемое *имя разработчика* (vendor name). Как и следовало ожидать, имя разработчика становится каталогом верхнего уровня в каталоге vendor/ при установке пакета. Зачастую это наименование организации, используемое владельцем пакета в хранилище GitHub или Bitbucket.

Когда все эти сведения о пакете окажутся на своих местах, можете зафиксировать пакет на сервере, выбранном вами в качестве системы контроля версий. Если же вы не уверены, что ввели все необходимые сведения о пакете, обратитесь к главе 17, "Контроль версий средствами Git", в которой этот вопрос рассматривается более подробно.

На заметку В диспетчере зависимостей Composer поддерживается также элемент version, но более подходящей нормой практики считается применение специального дескриптора в системе Git для отслеживания версии пакета. Этот дескриптор автоматически распознается в Composer.

Напомним, что весь каталог vendor не следует специально фиксировать на сервере системы контроля версий — по крайней мере так поступают не всегда, хотя из этого правила имеются вполне обоснованные исключения. Но зачастую сформированный файл composer.lock целесообразно отслеживать вместе с файлом composer.json.

Пакеты для конкретной платформы

Несмотря на то что Composer не подходит для установки системных пакетов, в нем можно задать системные требования, чтобы устанавливать пакеты только для той системы, для которой они предназначены. Пакет для конкретной платформы указывается с помощью единственного параметра, хотя иногда такой параметр дополняется типом пакета, обозначаемым через дефис. Доступные типы пакетов перечислены в табл. 16.2.

Таблица 16.2. *Типы пакетов для платформ*

Тип	Пример	Описание
PHP	`"php": "7.*"`	Версия PHP
Расширение	`"ext-xml": ">2"`	Расширение PHP
Библиотека	`"lib-iconv": "~2"`	Системная библиотека, используемая PHP
HHVM	`"hhvm": "~2"`	Версия HHVM, где HHVM — виртуальная машина, на которой поддерживается расширенная версия PHP

А теперь опробуем пакеты для платформ на практике. В следующем фрагменте кода указано, что для пакета требуются расширения PHP `xml` и `gd`:

```
{
    "require": {
        "abraham/twitteroauth": "2.0.*",
        "ext-xml": "*",
        "ext-gd": "*"
    }
}
```

Выполним обновление с помощью следующей команды:

```
$ composer update
```

В итоге будет получен следующий результат:

```
Loading composer repositories with package information
Updating dependencies
Your requirements could not be resolved to an installable set of
packages.

Problem 1
  - Root composer.json requires PHP extension ext-gd * but it is
    missing from your system. Install or enable PHP's gd extension.
```

Похоже, что мы установили расширение языка PHP для работы с XML, но о расширении GD для работы с изображениями забыли; поэтому диспетчер зависимостей Composer сгенерировал ошибку, гласящую, что в системе отсутствует требуемое расширение `ext-gd *` языка PHP.

Распространение пакетов через сайт Packagist

Если вы прорабатывали материал этой главы, то у вас, возможно, возникло недоумение, откуда происходят устанавливаемые нами пакеты. Это похоже на какое-то волшебство, но, как и следовало ожидать, за этим на самом деле стоит хранилище пакетов, которое называется "Packagist" и доступно по адресу `https://packagist.org`. При условии, что исходный код может быть найден в общедоступном хранилище типа Git, он может стать доступным и через хранилище Packagist.

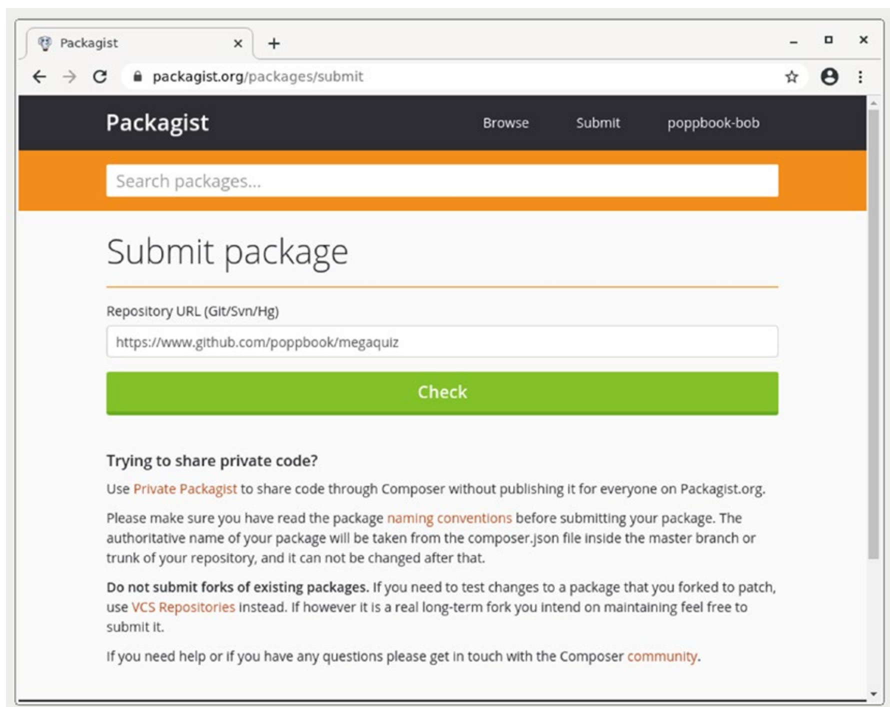


Рис. 16.1. *Добавление пакета в хранилище сайта Packagist*

Рассмотрим такую возможность на конкретном примере. Допустим, мы разместили свой пакет megaquiz в хранилище GitHub и теперь нам требуется уведомить об этом хранилище пакетов Packagist. Зарегистрировавшись в Packagist, нам достаточно указать URL своего хранилища, как показано на рис. 16.1.

После добавления пакета megaquiz сайт Packagist обратится к хранилищу в поисках файла composer.json, после чего будет отображена панель управления (рис. 16.2).

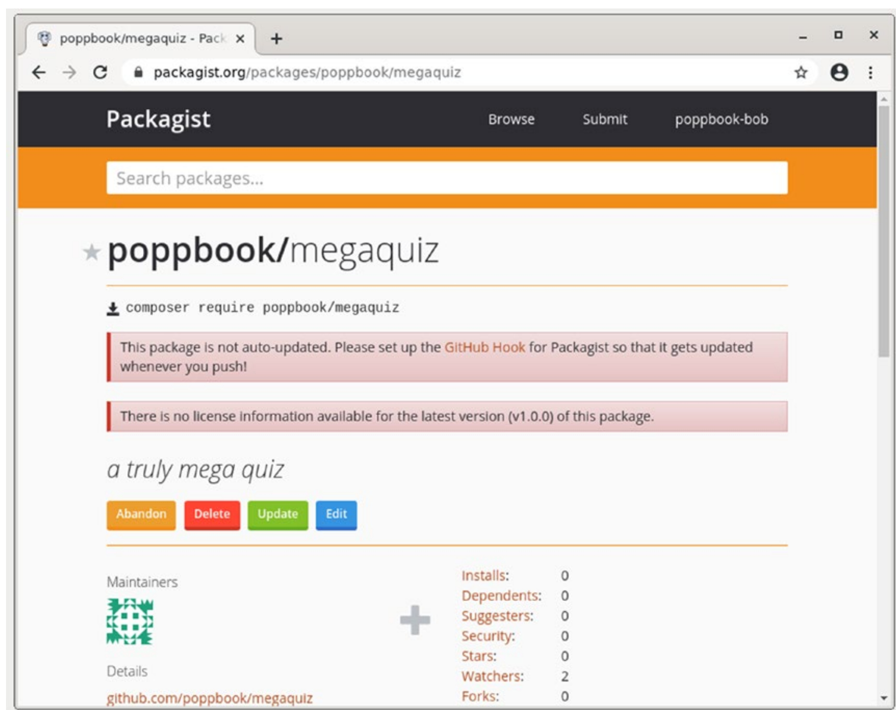


Рис. 16.2. *Панель управления пакетами*

Хранилище пакетов Packagist сообщит, что мы не предоставили сведения о лицензии. Этот недостаток можно исправить в любой момент, добавив элемент license в файл composer.json следующим образом:

```
"license": "Apache-2.0"
```

Кроме того, сайту Packagist не удалось обнаружить никаких сведений о версии пакета. Этот недостаток можно исправить, добавив дескриптор в хранилище GitHub следующим образом:

```
$ git tag -a '1.0.0' -m '1.0.0'
$ git push --tags
```

На заметку Если вы считаете, что я прибегаю к хитрости, лишь вскользь упоминая здесь Git, то вы совершенно правы. Более подробно Git и хранилище GitHub будут рассмотрены в главе 17, "Контроль версий средствами Git".

Теперь в хранилище пакетов Packagist известен номер версии моего пакета (рис. 16.3).

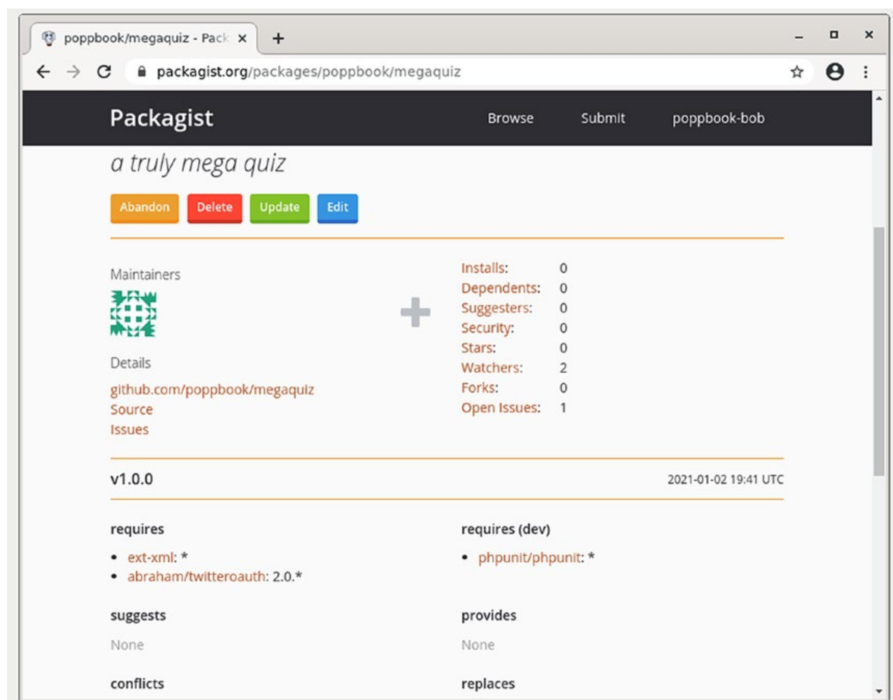


Рис. 16.3. *Теперь хранилищу пакетов Packagist известен номер версии пакета*

Итак, теперь кто угодно может включить пакет `megaquiz` из другого пакета. Ниже приведено минимальное содержимое файла `composer.json`:

```
{
    "require": {
        "poppbook/megaquiz": "*"
    }
}
```

На свой страх и риск я готов принять какую угодно версию пакета. Перейдем к его установке:

```
$ composer update
```

В итоге будет получен приведенный ниже результат (обратите внимание на то, что загружаются также зависимости, указанные нами при установке пакета `megaquiz`):

```
Loading composer repositories with package information
Updating dependencies
Lock file operations: 3 installs, 0 updates, 0 removals
  - Locking abraham/twitteroauth (2.0.1)
  - Locking composer/ca-bundle (1.2.8)
  - Locking poppbook/megaquiz (v1.0.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 3 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.2.8): Extracting archive
  - Installing abraham/twitteroauth (2.0.1): Extracting archive
  - Installing poppbook/megaquiz (v1.0.0): Extracting archive
    Generating autoload files
```

Работа с закрытыми пакетами

Разумеется, не всегда нужно предавать свой код гласности. Но иногда все же требуется поделиться им с небольшой частью авторизованных пользователей.

Ниже приведен пример закрытого пакета `getinstance/wtnlang-php`, который содержит библиотеку для языка сценариев:

```
{
    "name": "getinstance/wtnlang-php",
    "description": "it's a wtn language",
    "license": "private",
    "authors": [
        {
            "name": "matt zandstra",
            "email": "matt@getinstance.com"
        }
    ],
    "autoload": {
        "psr-4": {
            "getinstance\\wtnlang\\": ["src/", "test/unit"]
        }
```

```
    },
    "require": {
        "abraham/twitteroauth": "*",
        "aura/cli": "~2.1.0",
        "monolog/monolog": "^1.23"
    },
    "require-dev": {
        "phpunit/phpunit": "^7"
    }
}
```

Этот пакет размещается в закрытом хранилище Bitbucket и поэтому недоступен через хранилище пакетов Packagist. Так как же включить его в свой проект? Для этого достаточно сообщить диспетчеру зависимостей Composer, где искать данный пакет. С этой целью нужно добавить в файл `composer.json` элемент `repositories` следующим образом:

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "git@bitbucket.org:getinstance/wtnlang-php.git"
        }
    ],
    "require": {
        "poppbook/megaquiz": "*",
        "getinstance/wtnlang-php": "dev-develop"
    }
}
```

Я мог бы указать версию `getinstance/wtnlang-php` в блоке `require` так, чтобы она соответствовала дескриптору в репозитории git, но, используя префикс `dev-`, я могу обратиться к ветви репозитория. Это очень полезная во время разработки возможность. Итак, сейчас, если у меня есть доступ к `getinstance/wtnlang-php`, я могу установить и свой закрытый пакет, и `megaquiz` одновременно:

```
$ composer update

Loading composer repositories with package information
Updating dependencies
Nothing to modify in lock file
Installing dependencies from lock file (including require-dev)
Package operations: 7 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.2.8): Extracting archive
  - Installing psr/log (1.1.3): Extracting archive
```

```
 - Installing monolog/monolog (1.26.0): Extracting archive
 - Installing aura/cli (2.1.2): Extracting archive
 - Installing abraham/twitteroauth (2.0.1): Extracting archive
 - Installing getinstance/wtnlang-php (dev-develop de3bf14): Cloning
   de3bf1456c
 - Installing poppbook/megaquiz (v1.0.0): Extracting archive
   Generating autoload files
```

Резюме

Эта глава должна оставить у вас ощущение, что управлять пакетами и повышать эффективность проектов с помощью диспетчера зависимостей Composer очень легко. Используя файл `composer.json`, можно сделать свой код доступным другим пользователям как открыто (через хранилище Packagist), так и закрыто (через собственное хранилище). Такой подход позволяет автоматизировать загрузки зависимостей для пользователей и применять разработанные вами пакеты в сторонних пакетах, не прибегая к их явному включению.

ГЛАВА 17

Контроль версий средствами Git

Все неприятности достигают критического момента, когда порядок окончательно нарушается и события полностью выходят из-под контроля. Оказывались ли вы когда-нибудь в подобной ситуации, работая над проектом? Сумели ли вы вовремя распознать этот критический момент?

Возможно, это произошло, когда вы внесли "всего пару изменений" и обнаружили, что из-за этого весь проект оказался сломанным (и, что еще хуже, вы теперь не знаете, как вернуться к устойчивому состоянию, которое вы только что нарушили). Это могло произойти и в том случае, если три члена команды разработчиков работали над одним набором классов и сохранили свои изменения — каждый поверх изменений остальных. А может быть, вы обнаружили, что исправление программной ошибки, которое вы уже делали дважды, снова каким-то образом исчезло из кодовой базы? Разве вам не пригодилось бы в подобных случаях инструментальное средство, помогающее организовывать коллективный труд, делать моментальные снимки проектов и, если потребуется, производить их откат, а также объединять несколько ветвей разработки? В этой главе мы рассмотрим систему Git, которая позволяет сделать все это и даже больше.

В этой главе будут рассмотрены следующие вопросы.

- *Базовая конфигурация.* Несколько советов по настройке системы Git.
- *Импорт.* Начало нового проекта.
- *Фиксация изменений.* Сохранение работы в хранилище.
- *Обновление.* Объединение чужой работы со своей.
- *Ветвление.* Поддержка параллельных ветвей разработки.

Зачем нужен контроль версий

Контроль версий изменит вашу жизнь, если он этого еще не сделал (хотя бы ту ее сторону, которая связана с разработкой программного обеспечения). Сколько раз вы достигали устойчивого состояния в проекте, делали глубокий вдох и снова погружались в хаос разработки? Легко ли было возвращаться к устойчивой версии, когда нужно было продемонстрировать ход выполнения работ? Безусловно, достигнув устойчивого состояния, вы могли бы сохранить моментальный снимок своего проекта, сделав копию каталога разработки. А теперь представьте, что ваш коллега работает с той же кодовой базой. Возможно, он сохранил устойчивую копию кода, как это сделали вы. Но разница в том, что сделанная им копия — это моментальный снимок его, а не вашей работы. И конечно, у него такой же запутанный каталог разработки. Поэтому вам придется согласовывать четыре версии проекта. А теперь представьте проект, над которым работают четыре программиста и разработчик пользовательского веб-интерфейса... Кажется, вы побледнели... Может, вам лучше прилечь?

Система Git специально предназначена для разрешения описанных выше затруднений. С ее помощью все разработчики могут создать собственные копии кодовой базы, извлекаемые из центрального хранилища. Достигая всякий раз устойчивого состояния в коде, они могут получать последнюю версию кодовой базы с сервера и объединять ее с собственной работой. Когда все будет готово и все конфликты будут устранены, разработчики смогут перенести новую устойчивую версию кодовой базы в общее центральное хранилище на сервере.

Git представляет собой распределенную систему контроля версий. Это означает, что после получения из центрального хранилища ветви кодовой базы разработчики могут вносить изменения в собственное локальное хранилище, не устанавливая сетевое соединение с сервером центрального хранилища. Такой подход дает сразу несколько преимуществ. Прежде всего, он ускоряет выполнение ежедневных рутинных операций, а также позволяет работать над проектом в любом месте, например во время поездки в самолете, поезде или автомобиле. Где бы вы ни находились в данный момент, в конечном итоге вы сможете без особого труда сохранить результаты своей работы в центральном хранилище на сервере и поделиться ими с коллегами.

Тот факт, что каждый разработчик может объединить в центральном хранилище свою работу с работой других разработчиков, означает, что разработчикам не нужно больше беспокоиться о согласовании различных ветвей разработки и делать это вручную. Более того, они могут извлекать из хранилища версии кодовой базы по дате или некоторой метке. Когда код достигнет устойчивого состояния и можно будет продемонстрировать клиенту ход выполнения работ, достаточно будет пометить эту стадию разработки специальной меткой. Если заказчик вдруг нагрянет в ваш офис, то нужную кодовую базу с помощью этой метки можно будет без труда извлечь и произвести на заказчика благоприятное впечатление своими достижениями.

Но и это еще не все! Можно также управлять несколькими ветвями разработки одновременно. Если вам покажется, что это излишние сложности, представьте себе проект со стажем. Вы уже выпустили версию 1, и полным ходом идет разработка версии 2. Означает ли это, что версия 1.*n* уже не используется? Конечно, нет. Пользователи обнаруживают ошибки и постоянно требуют улучшений. До выпуска версии 2 остается несколько месяцев, так в какую же версию вносить изменения и какую тестировать? Система Git позволяет поддерживать различные ветви кодовой базы, так что вы можете создать специальную ветвь, предназначенную для исправления ошибок в версии 1.*n*, непосредственно в текущем рабочем коде. В ключевых точках эту ветвь можно объединить с кодом версии 2 (т.е. с основной ветвью), чтобы в новой версии извлечь выгоду из улучшений версии 1.*n*.

На заметку Git является далеко не единственной системой контроля версий. Вы можете познакомиться также с системами Subversion (`http://subversion.apache.org/`) и Mercurial (https://www.mercurial-scm.org/).

В этой главе приведен минимум того, что требуется знать в такой обширной области, как системы контроля версий. Тем, кто действительно хочет разобраться в особенностях работы системы Git, рекомендуем прочитать книгу *Pro Git* (Apress, 2009). Электронная версия ее второго издания доступна по адресу `https://git-scm.com/book/ru/v2`.

Теперь рассмотрим возможности системы Git на практике.

Установка Git

Если вы работаете в операционной системе Unix (например, Linux или FreeBSD), то клиент Git у вас уже, скорее всего, установлен и готов к применению.

На заметку В этой главе, как, впрочем, и в предыдущих главах, команды, вводимые в командной строке, начинаются с символа доллара ($), чтобы отличать их от вывода на экран при выполнении команд.

Попробуйте ввести в командной строке следующую команду:

```
$ git help
```

В итоге должна появиться информация, подтверждающая, что Git установлен и готов к работе. Если Git у вас еще не установлен, обратитесь к системной документации. Вы можете легко почти наверняка получить доступ к простому механизму установки Git (например, `Yum` или `Apt`) или загрузить Git непосредственно с веб-сайта по адресу `http://git-scm.com/downloads`.

На заметку Технический рецензент Пол Трегоинг (Paul Tregoing) также рекомендует Git для Windows (`https://gitforwindows.org/`), который, естественно, включает Git, а также набор полезных инструментов с открытым исходным кодом.

Использование онлайн-хранилища Git

Вы, вероятно, обратили внимание на то, что красной нитью через данную книгу проходит мысль, что вместо изобретения колеса вы должны по крайней мере разобраться в его конструкции, прежде чем покупать готовое колесо. Именно по этой причине мы рассмотрим механизм установки и ведения собственного центрального хранилища Git в следующем разделе. Обратимся, однако, к действительности. Для ведения хранилища можно почти наверняка воспользоваться специализированным сервером. Можно выбрать целый ряд вариантов, среди которых самыми крупными игроками на рынке являются BitBucket (`https://bitbucket.org`) и GitHub (`https://github.com/`).

Какой же вариант центрального хранилища выбрать? Пожалуй, лучше выбрать хранилище GitHub, которое, вероятно, стало уже стандартным для программных продуктов с открытым исходным кодом. Начинающие компании, производящие программное обеспечение, зачастую пользуются центральным хранилищем BitBucket, поскольку оно предоставляет свободный доступ к частным хранилищам и взимает плату только по мере разрастания команды разработчиков. Это, по-видимому, неплохой вариант, ведь если команда разработчиков разрастается, то можно обеспечить надежное финансирование проекта или даже получить какую-то прибыль. В таком случае можете похвалить себя за удачный выбор!

В какой-то степени выбор центрального хранилища для системы контроля версий подобен бросанию жребия. Попробуем зарегистрировать свой проект в GitHub. На рис. 17.1 показано, как сделать выбор между общедоступным и частным хранилищем. Выберем для данного проекта общедоступное хранилище.

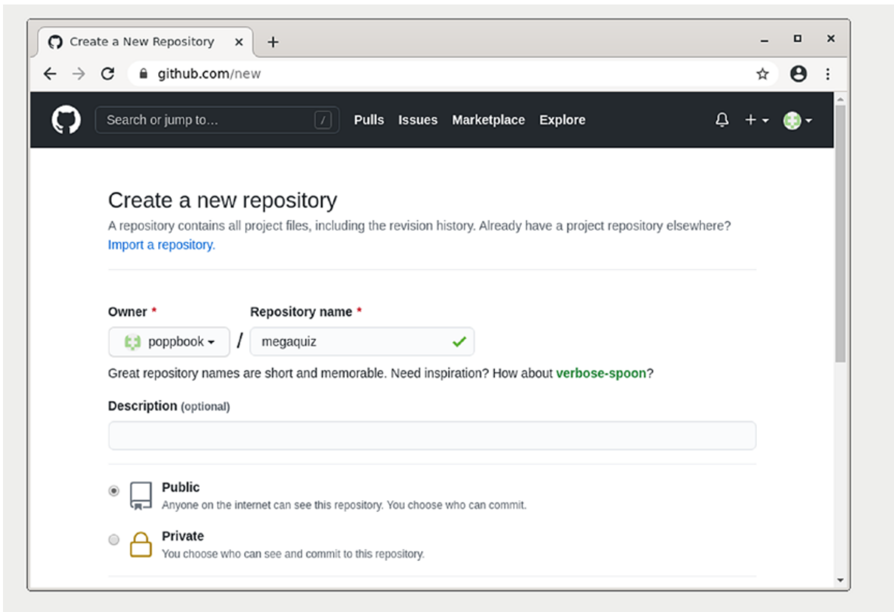


Рис. 17.1. *Начало проекта в GitHub*

На данном этапе GitHub предоставляет ряд полезных команд по импорту проекта. Они приведены на рис. 17.2.

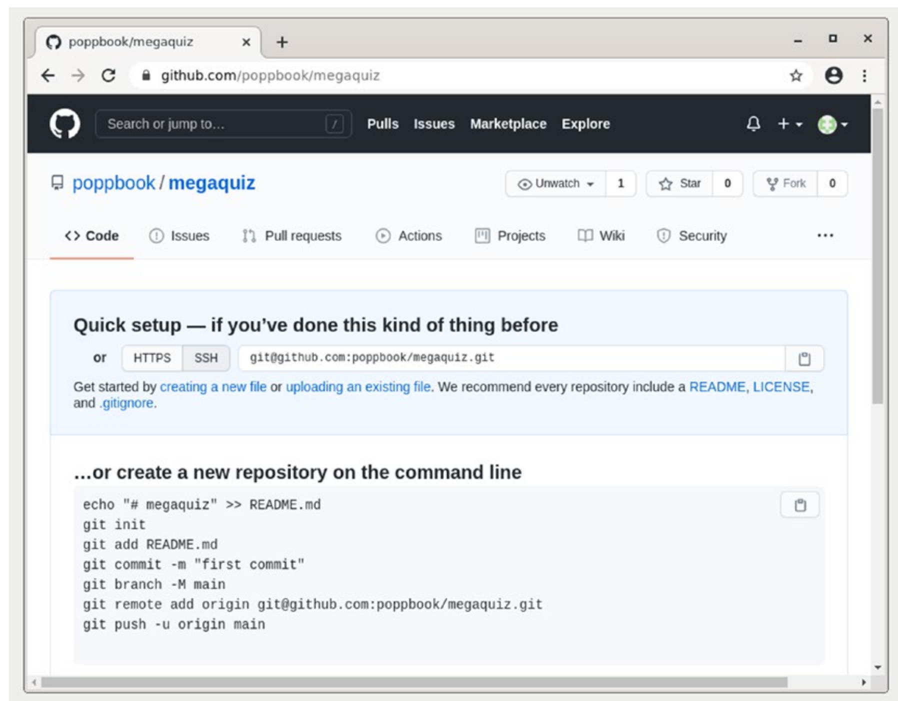


Рис. 17.2. *Команды по импорту проекта в GitHub*

Но мы пока еще не готовы выполнить эти команды. Дело в том, что сервер GitHub должен выполнить проверку пользователя при записи файлов в центральное хранилище. Для этого ему требуется открытый ключ пользователя. Один из способов генерации такого ключа описан ниже, в разделе “Конфигурирование сервера Git”. Получив открытый ключ, можно ввести его по ссылке SSH and GPG keys (Ключи SSH и GPG) на экране Personal Settings (Пользовательские настройки) в GitHub, как показано на рис. 17.3.

Теперь мы готовы приступить к добавлению файлов в хранилище. Но прежде необходимо отступить назад и потратить время на обустройство пути.

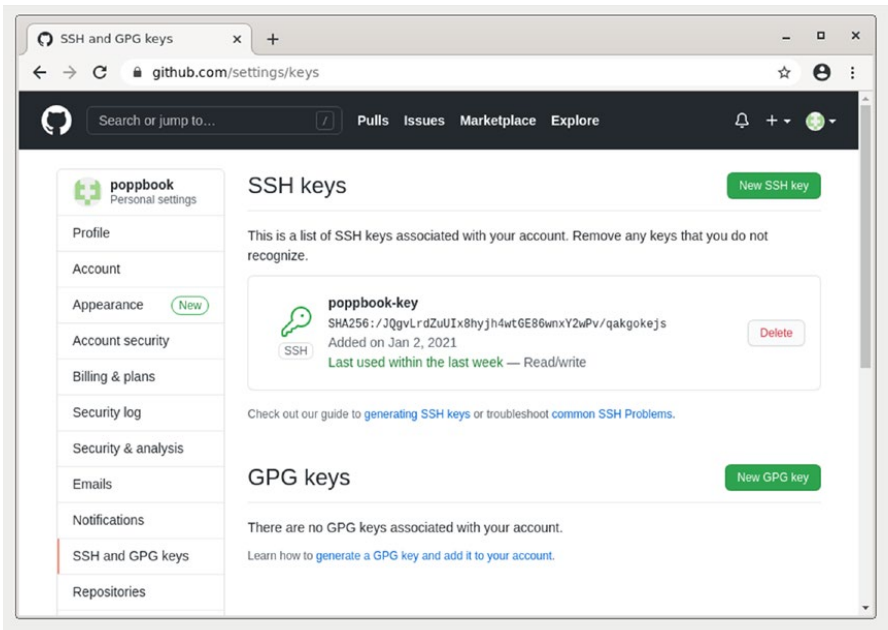


Рис. 17.3. *Ввод ключа SSH*

Конфигурирование сервера Git

Система Git имеет два принципиальных отличия от традиционных систем контроля версий. Во-первых, из-за особенностей внутренней структуры в Git хранятся копии самих файлов, а не изменения, внесенные в них с момента прошлой фиксации в хранилище. Во-вторых, и это нельзя не отметить, Git работает на локальном компьютере до того момента, когда пользователю нужно будет загрузить файлы из центрального хранилища или обновить их в центральном хранилище. Это означает, что для выполнения своей работы пользователям более не требуется постоянно подключение к Интернету.

Для работы с Git совершенно не обязательно иметь единое удаленное хранилище, но на практике почти всегда имеет смысл его создать, особенно если вы работаете в команде. В этом разделе мы рассмотрим действия, которые необходимо выполнить, чтобы установить и запустить в работу удаленный сервер Git. При этом предполагается наличие прав доступа суперпользователя (`root`) на машине под управлением Linux.

Создание удаленного хранилища

Чтобы создать удаленное хранилище Git, на сервере необходимо создать для него каталог. С этой целью подключитесь к удаленному серверу по сетевому протоколу SSH, введя свои учетные данные суперпользователя. Мы собираемся создать хранилище в папке `/var/git`. Поскольку создавать и модифицировать структуру системных каталогов может только суперпользователь, все приведенные ниже команды следует выполнять от его имени:

```
$ sudo mkdir -p /var/git/megaquiz
$ cd /var/git/megaquiz/
```

Здесь мы создаем родительский каталог `/var/git` для своих хранилищ, а в нем — подкаталог `megaquiz` для рассматриваемого здесь примера одноименного проекта. Теперь можно выполнить подготовку самого каталога с помощью следующей команды:

```
$ sudo git init --bare

Initialized empty Git repository in /var/git/megaquiz/
```

Параметр `--bare` предписывает Git инициализировать хранилище без рабочего каталога. Если попытаться перенести файлы в это хранилище, Git выдаст сообщение об ошибке.

На данном этапе работать с каталогом `/var/git` может только суперпользователь. Это положение можно изменить, создав специального пользователя `git` и одноименную группу и сделав их владельцами каталога, как показано ниже:

```
$ sudo adduser git
$ sudo chown -R git:git /var/git
```

Подготовка хранилища для локальных пользователей

Несмотря на то что сервер является удаленным, мы должны предоставить возможность фиксировать свои проекты в хранилище локальным пользователям. Если не предпринять специальных мер, при попытке записи файлов в хранилище локальный пользователь столкнется с проблемой отказа в доступе, особенно если до этого файлы в хранилище записывали

привилегированные пользователи. Поэтому необходимо выполнить следующую команду:

```
$ sudo chmod -R g+rws /var/git
```

Эта команда разрешает пользователям группы `git` доступ для записи файлов в каталог `/var/git` и заставляет присваивать всем вновь создаваемым файлам и подкаталогам в нем права доступа группы `git`. После того как мы принудительно назначили всем файлам права доступа группы `git`, локальные пользователи могут записывать свои файлы в удаленное хранилище. Более того, любые файлы, созданные в этом хранилище, будут доступны для записи всем членам группы `git`.

Чтобы добавить учетную запись локального пользователя в группу `git`, выполните приведенную ниже команду. В результате ее выполнения пользователь `bob` будет включен в группу `git`:

```
$ sudo usermod -aG git bob
```

Предоставление доступа пользователям

В предыдущем разделе был описан процесс добавления пользователя `bob` в группу `git`. Теперь, подключившись к удаленному серверу по сетевому протоколу SSH, этот пользователь сможет работать с хранилищем с помощью командной строки. Но, скорее всего, вы не собираетесь предоставлять доступ к серверу через командную оболочку всем пользователям. В любом случае большинство пользователей предпочтут воспользоваться распределенным характером Git, работая со своими локальными копиями файлов, полученными из центрального хранилища.

Чтобы предоставить пользователям доступ к удаленному хранилищу по сетевому протоколу SSH, можно, например, прибегнуть к аутентификации с открытым ключом. Такой ключ может уже иметься у пользователя. Так, на машине под управлением Linux он, скорее всего, будет находиться в файле `id_rsa.pub`, расположенном в конфигурационном подкаталоге `.ssh`. Если же такой файл отсутствует, сгенерировать новый ключ всегда можно без особого труда. На машине под управлением UNIX-подобной операционной системы для этого достаточно выполнить команду `ssh-keygen` и скопировать сгенерированный ключ в указанный файл, как показано ниже:

```
$ ssh-keygen
$ cat .ssh/id_rsa.pub
```

Копию этого ключа вы должны будете предоставить администратору хранилища. Получив такой ключ, администратор должен ввести его в настройки параметров сетевого протокола SSH для пользователя `git` на сервере удаленного хранилища. Для этого достаточно добавить скопированный открытый ключ пользователя в файл `.ssh/authorized_keys` в предварительно созданном конфигурационном подкаталоге `.ssh`, если ключ устанавливается впервые, выполнив приведенные ниже команды (которые выполняются в начальном каталоге пользователя `git`):

```
$ mkdir .ssh
$ chmod 0700 .ssh
# Создание файла authorized_keys и
# добавление в него ключа пользователя:
$ vi .ssh/authorized_keys
$ chmod 0700 .ssh/authorized_keys
```

На заметку Как правило, отказ в доступе по сетевому протоколу SSH происходит из-за того, что при создании конфигурационного подкаталога `.ssh` ему присваиваются слишком либеральные права доступа. Этот каталог должен быть доступен для чтения и записи только для владельца учетной записи. Исчерпывающее описание сетевого каталога SSH приведено в книге Майкла Штанке (Michael Stahnke) *Pro OpenSSH* (Apress, 2005).

Закрытие доступа к системной оболочке для пользователя `git`

Ни один сервер не должен быть открыт более, чем это строго необходимо. Вы можете захотеть позволить пользователю работать только с командами Git, но не более того.

В операционной системе Linux полный путь к системной оболочке, назначенной для конкретного пользователя, можно обнаружить в файле `/etc/passwd`. Ниже приведена строка из этого файла, относящаяся к пользователю `git` на моем удаленном сервере:

```
git:x:1001:1001::/home/git:/bin/bash
```

В состав Git входит специальная оболочка под названием `git-shell`, которая ограничивает возможности пользователя только выбранным на-

бором команд. Для включения этой программы нужно отредактировать соответствующую строку в файле `/etc/passwd`, как показано ниже[1]:

```
git:x:1001:1001::/home/git:/usr/bin/git-shell
```

Если теперь попытаться подключиться к удаленному серверу по сетевому протоколу SSH, то можно получить следующий результат:

```
$ ssh git@poppch17.vagrant.internal

Last login: Thu Dec 31 14:25:05 2020 from 192.168.33.1
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute
access.
Connection to 192.168.33.71 closed.
```

Начало проекта

После установки и настройки удаленного сервера Git и организации доступа к нему из локальной учетной записи настало время добавить рассматриваемый здесь проект в удаленное хранилище `/var/git/megaquiz`.

Прежде всего следует еще раз внимательно просмотреть файлы и каталоги и удалить все обнаруженные в них временные элементы.

Пренебрежение этой операцией является весьма распространенной и досадной ошибкой. Ко временным элементам, на которые следует обратить особое внимание, относятся автоматически сгенерированные файлы, включая пакеты Composer, каталоги построения, журналы регистрации из программы установки и т.д.

---

[1] Непосредственное редактирование файла `passwd` допускается не во всех UNIX-подобных системах. Для изменения пути к системной оболочке существуют специальные команды. Например, в BSD-совместимых системах необходимо ввести следующую команду:

```
pw usermod git -s /usr/local/bin/git-shell
```

А в системах Linux — такую:

```
usermod -s /usr/bin/git-shell git
```

За подробностями обращайтесь к руководству пользователя вашей ОС. — *Примеч. ред.*

На заметку В своем хранилище можно указать файлы и шаблоны имен файлов, которые необходимо игнорировать, создав файл `.gitignore`. Примеры содержимого этого файла в системе Linux можно просмотреть с помощью команды `man gitignore`. Вы можете увидеть, каким образом задаются шаблоны имен файлов, которые позволяют исключить различные файлы блокировки и временные каталоги, созданные процессами построения, редакторами и интегрированными средами разработки. Этот текст также доступен по адресу `http://git-scm.com/docs/gitignore`.

Прежде чем двигаться дальше, нам необходимо зарегистрировать свою идентификационную информацию в системе Git, выполнив следующие команды. Эта позволит отследить, кто и что делал в хранилище:

```
$ git config --global user.name "poppbook"
$ git config --global user.email "poppbook@getinstance.com"
```

Сообщив свои персональные данные и убедившись, что в проекте отсутствуют ненужные файлы, можно создать личное хранилище в своем начальном каталоге и выгрузить из него код проекта на удаленный сервер:

```
$ cd /home/mattz/work/megaquiz
$ git init

Initialized empty Git repository in /home/mattz/work/megaquiz/.git/
```

А теперь добавим сами файлы с помощью следующей команды:

```
$ git add .
```

Теперь Git отслеживает в локальном хранилище все файлы и каталоги, находящиеся в каталоге проекта `megaquiz`. Отслеживаемые файлы могут находиться в одном из трех состояний: *неизмененном* (unmodified), *измененном* (modified) и *проиндексированном* (staged) для фиксации. Чтобы проверить состояние файла, достаточно ввести следующую команду:

```
$ git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file: composer.json
#    new file: composer.lock
```

```
#    new file: main.php
#    new file: src/command/Command.php
#    new file: src/command/CommandContext.php
#    new file: src/command/FeedbackCommand.php
#    new file: src/command/LoginCommand.php
#    new file: src/quizobjects/User.php
#    new file: src/quiztools/AccessManager.php
#    new file: src/quiztools/ReceiverFactory.php
#
```

Благодаря команде `git add` все файлы данного проекта проиндексированы для фиксации в хранилище. Теперь можно выполнить следующую команду — фиксации данных в хранилище:

```
$ git commit -m'my first commit'

[master (root-commit) a5ca2d4] my first commit
 10 files changed, 1638 insertions(+)
 create mode 100644 composer.json
 create mode 100644 composer.lock
 create mode 100755 main.php
 create mode 100755 src/command/Command.php
 create mode 100755 src/command/CommandContext.php
 create mode 100755 src/command/FeedbackCommand.php
 create mode 100755 src/command/LoginCommand.php
 create mode 100755 src/quizobjects/User.php
 create mode 100755 src/quiztools/AccessManager.php
 create mode 100644 src/quiztools/ReceiverFactory.php
```

С помощью параметра `-m` в команде `commit` добавлено сообщение `'my first commit'`. Если этого не сделать, Git запустит текстовый редактор и попросит набрать в нем контрольное сообщение.

Если вы привыкли работать с такими системами контроля версий, как CVS и Subversion, то можете решить, что все необходимое сделано и больше ничего делать не нужно. И хотя мы можем продолжить благополучно редактировать файлы, добавлять их в хранилище, фиксировать их состояние и создавать отдельные ветви, имеется еще одна стадия, о которой необходимо не забыть, если мы собираемся размещать код данного проекта в центральном хранилище на сервере. Как будет показано далее в этой главе, Git позволяет создавать и поддерживать несколько ветвей разработки проекта. Благодаря этому можно поддерживать отдельную ветвь для каждого выпуска продукта и безопасно продолжать работу над новым кодом, не смешивая его с выпущенным ранее. При первом запуске на выполнение Git создает единственную ветвь разра-

ботки под именем master (главная). Чтобы убедиться в этом, достаточно ввести команду

```
$ git branch -a
* master
```

Параметр -a предписывает Git показать все ветви разработки проекта, кроме тех, которые находятся на удаленном сервере (это стандартное поведение данной системы). Как видите, система вывела по указанной выше команде только ветвь master.

На самом деле мы еще ничего не сделали, чтобы каким-то образом связать локальное хранилище с удаленным сервером. И теперь самое время сделать это с помощью следующей команды:

```
$ git remote add origin git@poppch17.vagrant.internal:/var/git/
megaquiz
```

К сожалению, эта команда удручающе немногословна и совсем не выводит никакой информации, в особенности учитывая количество работы, которую она выполняет. По сути, она говорит Git следующее: "Связать псевдоним origin с указанным хранилищем на сервере, а также отслеживать изменения между локальной ветвью разработки проекта под именем master и ее эквивалентом на удаленном сервере origin".

Чтобы убедиться в том, что псевдоним origin связан с удаленным сервером, выполните следующую команду:

```
$ git remote -v

origin git@poppch17.vagrant.internal:/var/git/megaquiz (fetch)
origin git@poppch17.vagrant.internal:/var/git/megaquiz (push)
```

Разумеется, если вы планируете использовать глобальную службу наподобие GitHub и загрузили файлы в ее хранилище, как показано на рис. 17.2, вам следует воспользоваться соответствующим вариантом команды git remote add. В моем случае она выглядит так:

```
$ git remote add origin git@github.com:poppbook/megaquiz.git
```

Однако не запускайте предыдущую команду, если только вы действительно не хотите поместить свои файлы в мой репозиторий GitHub!

Однако я до сих пор не отправил никаких файлов на мой сервер Git, так что это будет моим следующим шагом:

```
$ git push origin master

Counting objects: 16, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (16/16), 8.87 KiB | 0 bytes/s, done.
Total 16 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To git@github.com:poppbook/megaquiz.git
* [new branch] master -> master
```

Теперь можно выполнить команду `git branch` еще раз, чтобы убедиться, что на удаленном сервере появилась ветвь разработки `master`:

```
$ git branch -a

* master
  remotes/origin/master
```

Или чтобы увидеть только ветви на удаленном сервере:

```
$ git branch -r

origin/master
```

На заметку Я создал то, что называют *отслеживающей ветвью* (tracking branch). Это локальная ветвь, которая связана со своей удаленной копией.

Клонирование хранилища

Для целей этой главы добавим в нашу команду нового члена по имени "Боб", чтобы он работал вместе с нами над проектом MegaQuiz. Разумеется, Бобу понадобится личная версия исходного кода, чтобы работать с ней самостоятельно. Для этого мы добавили открытый ключ его учетной записи на сервер Git, и теперь Боб может приступить к работе. Мы поздравили Боба с присоединением к нашему проекту в параллельном мире GitHub. После этого Боб ввел свой открытый ключ для своей учетной записи. Результат оказался точно таким же. Чтобы запросить из удаленного хранилища персональную копию кода, Боб должен ввести следующую команду:

```
$ git clone git@github.com:poppbook/megaquiz.git

Cloning into 'megaquiz'...
remote: Enumerating objects: 16, done.
```

```
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 16 (delta 2), reused 16 (delta 2), pack-reused 0
Receiving objects: 100% (16/16), 8.87 KiB | 0 bytes/s, done.
Resolving deltas: 100% (2/2), done.
```

Теперь мы с Бобом можем работать самостоятельно над одним проектом на своих локальных компьютерах. Как только каждый член нашей команды закончит свою часть работы, мы сможем легко обменяться сделанными изменениями.

Обновление и фиксация изменений

Боб, конечно, — замечательный и талантливый парень. Но у него есть одна общая для всех программистов и весьма досадная черта: он не может оставить в покое код, написанный кем-то другим.

Боб умен и любознателен, быстро увлекается новыми направлениями в разработке и стремится помочь оптимизировать новый код. В итоге в кодовой базе, куда ни кинь взгляд, везде видна рука Боба. В частности, он добавил кое-что в документацию (реализовал идею, которую мы обсуждали с ним за чашкой кофе). Не пора ли его убить? Но пока что нам необходимо объединить код, над которым мы работаем, с тем кодом, который добавил Боб.

Ниже приведено содержимое исходного файла quizobjects/User.php. В настоящий момент в нем нет ничего, кроме голого скелета:

```
namespace poppbook\megaquiz\quizobjects;

class User
{
}
```

Мы решили немного дополнить документацию и сначала ввели комментарии в свою версию данного файла:

```
namespace popp\ch17\megaquiz\quizobjects;

/**
* @license http://www.example.com Borsetshire Open License
* @package quizobjects
*/
class User
{
}
```

Напомним, что каждый файл в хранилище может находиться в одном из трех состояний: *неизмененном*, *измененном* и *проиндексированном* для фиксации. Так вот, исходный файл `User.php` теперь перешел из *неизмененного* состояния в *измененное*. Чтобы убедиться в этом, достаточно выполнить следующую команду:

```
$ git status

# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes
#         in working directory)
#
#     modified: src/quizobjects/User.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Как видите, исходный файл `User.php` был изменен, но он пока еще не проиндексирован для фиксации. Чтобы изменить это положение, достаточно выполнить команду `git add` и получить приведенный ниже результат:

```
$ git add src/quizobjects/User.php
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     modified: src/quizobjects/User.php
#
```

Теперь все готово для фиксации изменений в хранилище:

```
$ git commit -m'added documentation' src/quizobjects/User.php

[master 997622c] added documentation
1 file changed, 5 insertions(+)
```

По команде `git commit` внесенные изменения были зафиксированы в локальном хранилище. Если мы считаем, что эти изменения очень важны для остальных участников проекта, то должны перенести свой новый код в удаленное хранилище с помощью следующей команды:

```
$ git push origin master
Counting objects: 9, done.
Delta compression using up to 2 threads.
```

```
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 537 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To git@github.com:poppbook/megaquiz.git
ce5a604..997622c master -> master
```

Тем временем, работая в собственной "песочнице", Боб, как всегда, проявил излишнюю инициативу, добавив к классу следующий комментарий:

```
namespace popp\ch17\megaquiz\quizobjects;

/**
* @package quizobjects
*/
class User
{
}
```

Теперь уже Бобу нужно внести изменения, зафиксировать их в локальном хранилище и перенести на сервер, выполнив последовательно команды `add`, `commit` и `push`. Но поскольку эти команды обычно выполняются совместно, в системе Git все это можно сделать одной командой следующим образом:

```
$ git commit -a -m'my great documentation'

[master 13de456] my great documentation
1 file changed, 4 insertions(+)
```

В итоге у нас появились две разные версии исходного файла `User.php`. Одна из них ранее перенесена нами в удаленное хранилище, а вторая — зафиксирована Бобом в его локальном хранилище, но еще не перенесена на сервер. Посмотрим, что же произойдет, если Боб попытается загрузить свою версию этого исходного файла в удаленное хранилище на сервере:

```
$ git push origin master

To git@github.com:poppbook/megaquiz.git
! [rejected] master -> master (fetch first)
error: failed to push some refs to
        'git@github.com:poppbook/megaquiz.git'
hint: Updates were rejected because the remote contains
         work that you do
hint: not have locally. This is usually caused by another
        repository pushing
hint: to the same ref. You may want to first merge the
        remote changes (e.g.,
```

```
hint: 'git pull') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help'
        for details.
```

Как видите, Git не позволяет выполнить перенос файла в хранилище, если требуется его обновление. Поэтому Боб должен сначала получить нашу версию исходного файла `User.php` с помощью команды

```
$ git pull origin master

remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 1), reused 5 (delta 1), pack-reused 0
        Unpacking objects: 100% (5/5), done.
From github.com:poppbook/megaquiz
* branch master -> FETCH_HEAD
Auto-merging src/quizobjects/User.php
CONFLICT (content): Merge conflict in src/quizobjects/User.php
Automatic merge failed; fix conflicts and then commit the result.
```

Git может благополучно объединить данные из двух файлов лишь в том случае, если внесенные в них изменения *не перекрываются*. В Git отсутствуют средства для обработки изменений, которые затронули одни и те же строки кода. Как ей разобраться в приоритетности изменений? Следует ли записать наши изменения поверх изменений Боба, или наоборот? Должны ли сосуществовать оба варианта изменений и какое из них должно быть первым? В этой ситуации Git не остается иного выбора, кроме как сообщить о конфликте и предоставить Бобу возможность его уладить.

Вот что увидит Боб, когда откроет данный файл:

```
/**
<<<<<<< HEAD
 * @package quizobjects
 */
=======
 * @license http://www.example.com Borsetshire Open License
 * @package quizobjects
 */
>>>>>>> f36c6244521dbd137b37b76414e3cea2071958d2
namespace poppbook\megaquiz\quizobjects;

class User
{
}
```

В файл включены комментарий Боба и все изменения, вызвавшие конфликт, вместе с метаданными, которые сообщают о происхождении отдельных частей исходного кода. Сведения о конфликте отделяются строкой знаков равенства. Изменения, внесенные Бобом, отмечены строкой знаков "меньше", после которой следует слово 'HEAD'. А изменения в удаленном хранилище отмечены с новой строки строкой после знаков "больше".

Выяснив причину конфликта, Боб может отредактировать исходный файл и устранить этот конфликт:

```
/**
 * @package quizobjects
 * @license http://www.example.com Borsetshire Open License
 * @package quizobjects
 */
namespace poppbook\megaquiz\quizobjects;

class User
{
}
```

Устранив конфликт, Боб должен проиндексировать этот файл для фиксации изменений в своем локальном хранилище с помощью следующих команд:

```
$ git add src/quizobjects/User.php
$ git commit -m'documentation merged'

[master c99d3f5] documentation merged
```

Только после этого он может перенести измененный файл в удаленное хранилище с помощью следующей команды:

```
$ git push origin master
```

Добавление и удаление файлов и каталогов

Проекты меняются по мере своего развития. Это обстоятельство должно приниматься во внимание в системах контроля версий, в которых пользователям должна быть предоставлена возможность добавлять новые файлы и удалять старые, которые в противном случае будут только мешать в работе над проектом.

Добавление файла

В приведенных выше примерах уже не раз демонстрировалось применение команды git add. Она, в частности, применялась на стадии создания проекта для добавления исходного кода в пустое хранилище megaquiz, а на последующих стадиях — для индексации файлов перед их фиксацией. Выполнив команду git add с указанным в ней неотслеживаемым файлом или каталогом, вы тем самым предписываете Git начать отслеживание изменений в нем и проиндексировать его для фиксации. Теперь добавим в рассматриваемый здесь проект новый документ в файле CompositeQuestion.php, выполнив следующие команды:

```
$ touch src/quizobjects/CompositeQuestion.php
$ git add src/quizobjects/CompositeQuestion.php
```

В реальной ситуации, вероятно, следовало бы начать с добавления некоторого содержимого в документ CompositeQuestion.php, но здесь мы ограничимся созданием пустого файла с помощью стандартной команды touch. После добавления указанного документа нужно еще выполнить подкоманду commit, чтобы зафиксировать внесенные изменения:

```
$ git commit -m'initial check in'

[master 323bec3] initial check in
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 src/quizobjects/CompositeQuestion.php
```

В итоге файл CompositeQuestion.php окажется в локальном хранилище.

Удаление файла

Если мы слишком поспешили с добавлением документа в хранилище, его придется удалить. Неудивительно, что для этого используется команда rm:

```
$ git rm src/quizobjects/CompositeQuestion.php

rm 'src/quizobjects/CompositeQuestion.php'
```

В этом случае для фиксации изменений также потребуется команда commit. Как и прежде, убедиться в успешно выполненном удалении можно, выполнив команду git status:

```
$ git status

# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     deleted: src/quizobjects/CompositeQuestion.php
#

$ git commit -m'removed Question'

[master 5bf88aa] removed CompositeQuestion
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 src/quizobjects/CompositeQuestion.php
```

Добавление каталога

С помощью команд `add` и `rm` можно также добавлять и удалять каталоги. Допустим, Бобу требуется создать новый каталог. С этой целью необходимо выполнить следующие команды:

```
$ mkdir resources
$ touch resources/blah.gif
$ git add resources/
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     new file: resources/blah.gif
#
```

Обратите внимание на то, что содержимое каталога `resources` автоматически добавляется в хранилище. Теперь Боб может зафиксировать изменения в своем локальном хранилище и перенести их на удаленный сервер, как обычно.

На заметку Будьте осторожны при использовании команды `git add` с каталогами. Это жадная команда, которая добавит все файлы и подкаталоги в текущем каталоге. Всегда проверяйте результаты ее работы с помощью команды `git status`.

Удаление каталогов

Как и следовало ожидать, для удаления каталогов служит команда `rm`. В данном случае требуется удалить не только каталог, но и его содержимое, поэтому в команде `rm` следует указать параметр `-r`, как показано ниже. Здесь мы категорически не согласны с Бобом в том, что в проект нужно добавить каталог `resources`:

```
$ git rm -r resources/

rm 'resources/blah.gif'
```

Метка о выпуске

Проект благополучно достигает состояния готовности, и можно перейти к его поставке или развертыванию. При выпуске готового продукта в хранилище следует оставить специальную метку, которая позволит в дальнейшем вернуться к текущему состоянию кода и внести в него необходимые изменения. Такая метка делается с помощью команды `git tag`:

```
git tag -a 'v1.0.0' -m'release 1.0.0'
```

Чтобы выяснить, какие метки имеются в хранилище, достаточно выполнить команду `git tag` без параметров:

```
$ git tag

v1.0.0
```

Мы добавили метки в локальном хранилище. Чтобы перенести их в удаленное хранилище, следует воспользоваться командой `git push`, указав в ней параметр `--tags`:

```
$ git push origin --tags

Counting objects: 1, done.
Writing objects: 100% (1/1), 159 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:poppbook/megaquiz.git
 * [new tag]     v1.0.0 -> v1.0.0
```

Применение параметра `--tags` в данной команде приведет к тому, что все локальные метки будут перенесены в удаленное хранилище.

Любое действие, выполняемое в хранилище GitHub, может быть отслежено на сайте. Так, на рис. 17.4 показана сделанная нами отметка выпуска.

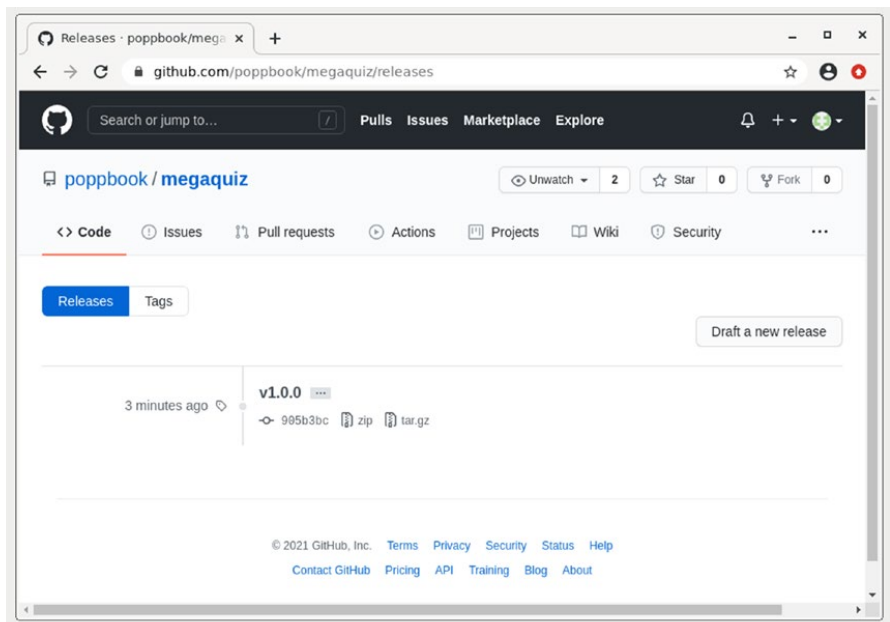


Рис. 17.4. *Просмотр отметки выпуска в хранилище GitHub*

После создания метки исходного кода в хранилище у вас, вероятно, возникнет вопрос, как ею воспользоваться, чтобы в дальнейшем вносить изменения в прежние версии своих программ? Сначала мы должны рассмотреть вопрос ветвления проекта, в котором особенно ярко проявляются преимущества Git!

Ветвление проекта

Теперь, когда проект завершен, можно подумать о том, чтобы сделать что-то новенькое, правда? В конце концов, наш код был так удачно написан и отлажен, что ошибки в нем просто невозможны, и так тщательно продуман, что пользователям не понадобятся новые возможности!

Но в действительности мы должны продолжать работу с кодовой базой по крайней мере на двух уровнях. Ведь нам уже начинают поступать сообщения об ошибках, а также требования реализовать в версии 1.2.0 новые

фантастические возможности. Как нам выйти из этой ситуации? Мы должны исправлять ошибки сразу по мере поступления сообщений о них и в то же время продолжать работу над основным кодом проекта. Безусловно, можно исправить ошибки в процессе разработки и выпустить исправления однократно, когда появится следующая устойчивая версия системы. Но в таком случае пользователям придется долго ждать, пока они увидят эти исправления, что совершенно неприемлемо. С другой стороны, мы можем выпустить проект в его текущем состоянии, но с исправленными ошибками, и продолжить над ним работу. В этом случае мы рискуем выпустить не до конца работоспособный код. Очевидно, нам нужны две ветви разработки. Тогда мы сможем, как и раньше, продолжать работу над проектом и добавлять новые и неотлаженные возможности в его основную ветвь, нередко называемую *стволом*[2], а исправления замеченных пользователями ошибок вносить в другую ветвь. Поэтому теперь нам нужно создать ветвь для только что выпущенной версии продукта, чтобы впоследствии в нее можно было вносить требуемые исправления.

На заметку Описанный выше порядок организации ветвей, безусловно, не является единственно возможным вариантом. Разработчики постоянно спорят по поводу наилучшего порядка организации ветвей, выпусков и исправлений программных ошибок. К числу самых распространенных для этих целей инструментальных средств относится `gitflow`, подробное описание которого находится по адресу `http://danielkummer.github.io/git-flow-cheatsheet/`. При этом подходе для выпуска будет служить ветвь `master`, а для нового кода — ветвь `develop`, которая объединяется с ветвью `master` во время выпуска. У каждой единицы активной разработки имеется свой набор функциональных возможностей, который объединяется с ветвью `develop`, когда она становится устойчивой.

Создать новую ветвь и переключиться на нее можно с помощью команды `git checkout`. Но сначала выясним, какие ветви разработки существуют в данный момент, выполнив следующую команду:

```
$ git branch -a

* master
  remotes/origin/master
```

---

[2] От англ. *trunk* — “основа”, “магистраль”. — *Примеч. ред.*

Как видите, у нас имеется всего одна ветвь разработки master и ее эквивалент на удаленном сервере. Теперь создадим новую ветвь разработки и переключимся на нее:

```
$ git checkout -b megaquiz-branch1.0

Switched to a new branch 'megaquiz-branch1.0'
```

Для отслеживания ветвей я воспользуюсь, например, файлом src/command/FeedbackCommand.php. По-видимому, я создал специальную ветвь для исправления ошибок как раз вовремя. Пользователи уже сообщили, что им не удается воспользоваться механизмом обратной связи, реализованным в системе. В итоге ошибку удалось локализовать:

```
//...
$result = $msgSystem->despatch($email, $msg, $topic);
if (! $user)
{
    $this->context->setError($msgSystem->getError());
    //...
```

Фактически нужно проверить значение переменной $result, а не $user. Вот необходимые изменения в исходном коде:

```
//...
$result = $msgSystem->dispatch($email, $msg, $topic);
if (! $result)
{
    $this->context->setError($msgSystem->getError());
    //...
```

Поскольку мы переключились на ветвь megaquiz-branch1.0 и работаем в ней, можно зафиксировать изменения в локальном хранилище следующим образом:

```
$ git add src/command/FeedbackCommand.php
$ git commit -m'bugfix'
[megaquiz-branch1.0 6e56ade] bugfix
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Эта фиксация, безусловно, локальная. Поэтому необходимо выполнить приведенную ниже команду git push, чтобы внести изменения в удаленное хранилище:

```
$ git push origin megaquiz-branch1.0

Counting objects: 9, done.
```

```
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 456 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote: Create a pull request for 'megaquiz-branch1.0' on GitHub
remote: by visiting: https://github.com/poppbook/megaquiz/pull
remote: /new/megaquiz-branch1.0
To git@github.com:poppbook/megaquiz.git
  * [new branch] megaquiz-branch1.0 -> megaquiz-branch1.0
```

Вернемся к Бобу. Он наверняка уже рьяно взялся за дело и также исправил некоторые ошибки. Для начала он должен выполнить команду `git fetch`, которая сообщит ему о новой информации на сервере. Затем он может ознакомиться с доступными ветвями на сервере с помощью команды `git branch -a`:

```
$ git fetch
$ git branch -a

* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/megaquiz-branch1.0
```

При этом Боб должен создать у себя локальную ветвь и связать ее с удаленной:

```
$ git checkout megaquiz-branch1.0

Branch megaquiz-branch1.0 set up to track remote
    branch megaquiz-branch1.0 from origin.
Switched to a new branch 'megaquiz-branch1.0'
```

Теперь Боб может свободно работать, внося любые изменения и фиксируя их в локальном хранилище. Все они окажутся в удаленном хранилище после выполнения команды `git push`.

Тем временем нам понадобилось внести несколько новых изменений в ствол, т.е. в ветвь разработки `master`. Рассмотрим вновь состояние наших ветвей разработки в локальном хранилище:

```
$ git branch -a

  master
* megaquiz-branch1.0
  remotes/origin/master
  remotes/origin/megaquiz-branch1.0
```

Для перехода на другую ветвь следует выполнить команду `git checkout`, но на этот раз параметр `-b` указывать не нужно:

```
$ git checkout master

Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

Если мы вновь просмотрим содержимое файла `command/FeedbackCommand.php`, то обнаружим, что внесенное нами ранее исправление ошибки загадочно исчезло! Оно, конечно, осталось в ветви `megaquiz-branch1.0` хранилища. В дальнейшем мы всегда можем объединить все внесенные изменения в главную ветвь разработки `master`, так что особенно волноваться не о чем. Вместо этого мы уделим пока что основное внимание добавлению нового фрагмента кода:

```
class FeedbackCommand extends Command
{
    public function execute(CommandContext $context): bool
    {
        // Новая рискованная разработка
        $msgSystem = ReceiverFactory::getMessageSystem();
        $email = $context->get('email');
        // ...
```

В этом фрагменте кода введен лишь комментарий, имитирующий добавление нового кода. Это изменение можно теперь зафиксировать и перенести в хранилище следующим образом:

```
$ git commit -am'new development on master'
$ git push origin master
```

Итак, теперь у нас имеются две параллельные ветви разработки. Разумеется, рано или поздно все исправления программных ошибок, сделанные в ветви `megaquiz-branch1.0`, нам придется внести в основную ветвь разработки.

Это можно сделать из командной строки, но прежде остановимся на одном средстве, которое поддерживается в GitHub и аналогичных службах, таких как BitBucket. Запрос на включение внесенных изменений позволяет запросить просмотр исходного кода перед объединением ветви. Следовательно, прежде чем исправления в ветви `megaquiz-branch1.0` достигнут главной ветви, мы можем попросить Боба проверить проделанную нами работу. Как показано на рис. 17.5, GitHub обнаруживает эту

ветвь, и у нас появляется возможность выдать запрос на включение внесенных изменений.

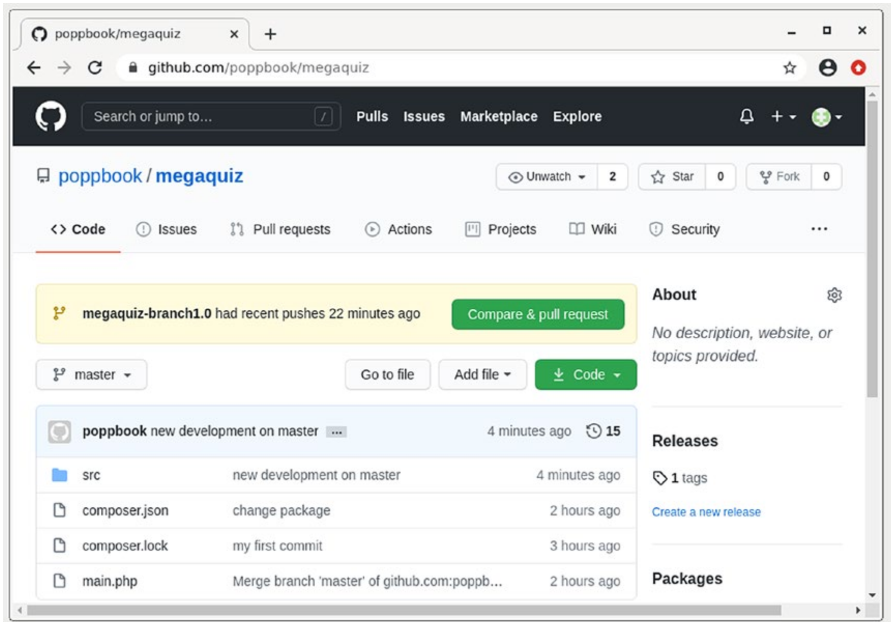


Рис. 17.5. *Хранилище GitHub упрощает выдачу запросов на включение внесенных изменений*

Щелкнув на соответствующей кнопке, введем комментарии, прежде чем выдать запрос на включение внесенных изменений. Полученный результат приведен на рис. 17.6.

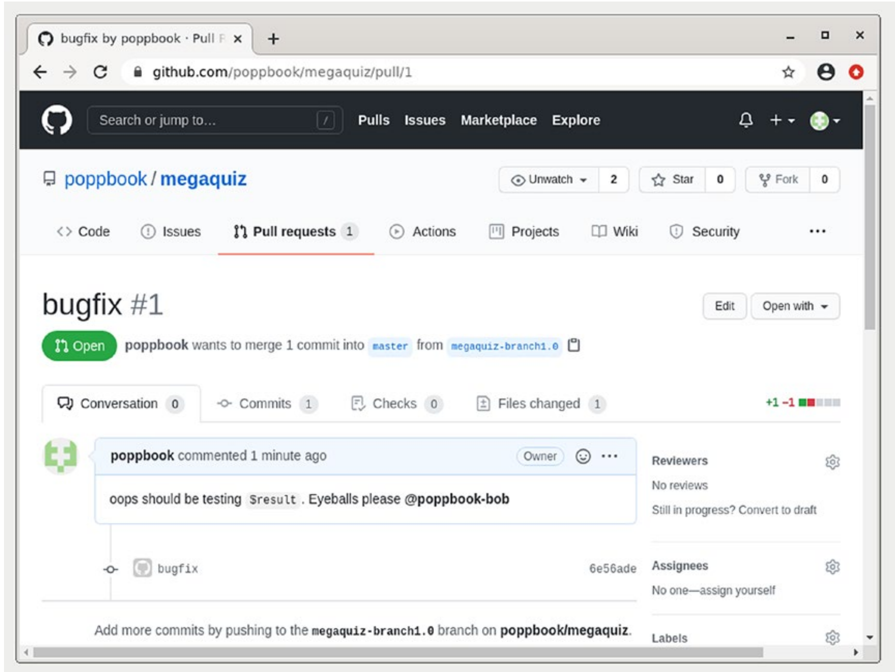


Рис. 17.6. Выдача запроса на включение внесенных изменений

Теперь Боб может проверить внесенные изменения и, если потребуется, добавить свои комментарии. GitHub показывает ему, что именно изменилось. Комментарии Боба приведены на рис. 17.7.

Как только Боб подтвердит наш запрос на включение внесенных изменений, мы можем выполнить объединение непосредственно в браузере или вернуться в режим командной строки. Сделать это нетрудно, и в Git для этой цели предусмотрена специальная команда — `merge`:

```
$ git checkout master

Already on 'master'
```

Итак, мы уже находимся в главной ветви, но убедиться в этом все же не помешает. Теперь можно выполнить объединение с помощью следующей команды:

```
$ git merge --no-commit megaquiz-branch1.0

Auto-merging src/command/FeedbackCommand.php
Automatic merge went well; stopped before committing as requested
```

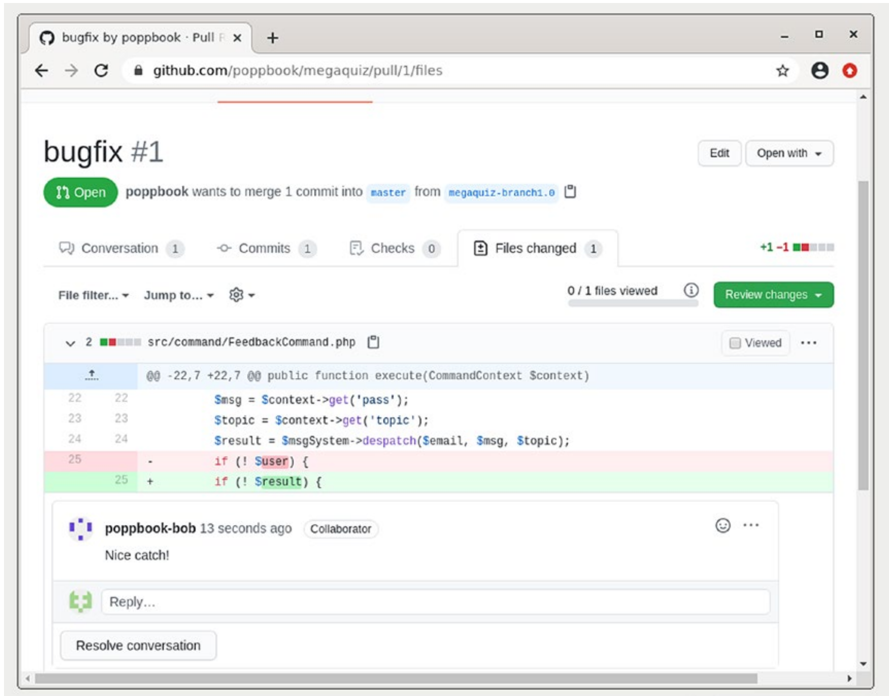


Рис. 17.7. *Изменения, охватываемые в запросе на включение внесенных изменений*

Передавая флаг `--no-commit`, я оставляю слияние нефиксированным, что дает мне еще один шанс хорошенько все проверить. Как только я буду удовлетворен, я смогу продолжить работу и зафиксировать изменения.

На заметку Объединять или не объединять? Этот выбор не всегда так прост, как может показаться на первый взгляд. Иногда, например, исправление ошибки может быть разновидностью временной меры, которая заменяется более тщательной реорганизацией кода в стволе или больше не применяется по причине изменений в спецификации. Это решение должно быть обязательно продумано. Но в большинстве команд разработчиков, в которых мне приходилось работать, весь проект обычно старались объединять в стволе, сводя к минимуму все работы в ветвях. Новые для разработчиков функциональные средства обычно появляются в стволе и быстро доходят до пользователей благодаря политике быстрого и частого выпуска.

Если мы посмотрим теперь на версию класса `FeedbackCommand` в стволе, то увидим, что все изменения были объединены:

```
public function execute(CommandContext $context): bool
{
    // Новая рискованная разработка
    $msgSystem = ReceiverFactory::getMessageSystem();
    $email = $context->get('email');
    $msg = $context->get('pass');
    $topic = $context->get('topic');
    $result = $msgSystem->despatch($email, $msg, $topic);
    if (! $result)
    {
        $this->context->setError($msgSystem->getError());
        return false;
    }
```

Метод `execute()` теперь включает и разработку из ветви `master`, и исправление программной ошибки.

Новая ветвь создана сразу после того, как была "выпущена" первая версия MegaQuiz 1.0. Именно в нее мы вносили исправления ошибок. Напомню, что на данном этапе мы создали метку в хранилище, и было обещано, что со временем будет показано, как с ней можно работать. Но на самом деле вы уже увидели, как это делается. Локальную ветвь разработки можно создать по метке таким же образом, как это сделал Боб, установив свою локальную версию ветви для исправления ошибок. Отличие лишь в том, что эта новая ветвь будет совершенно чистой. В ней не будут отслеживаться изменения ветви из удаленного хранилища:

```
$ git checkout -b v1.0.0-branch v1.0.0

Switched to a new branch 'v1.0.0-branch'
```

Создав новую ветвь, мы можем перенести ее в удаленное хранилище и сделать доступной для других участников команды, как было показано выше.

На заметку Git — удивительно универсальный и полезный инструмент. Как и в случае любого мощного инструмента, его использование иногда может привести к непредвиденным последствиям. Для моментов, когда ты загнал себя в угол и нужно быстро все сбросить, технический рецензент Пол Трегоинг (Paul Tregoing) рекомендует `https://dangitgit.com/ru`. На этом сайте полно рецептов, которые могут спасти ваш рассудок, так что его стоит добавить в закладки, если вы серьезно работаете с Git.

Две другие команды Git, которые стоит иметь в своем арсенале, — `git stash` и `git stash apply`. Когда вы по уши в локальном редактировании, но вас просят переключаться между ветками, то первое побуждение — зафиксировать незавершенную работу. Но вы можете не захотеть фиксировать недоделанный код. Вы можете решить, что ваш единственный выбор — выбросить локальные изменения или скопировать их во временные файлы. Но если вы запустите `git stash`, то все локальные изменения будут спрятаны для вас за кулисами, и ваша ветка вернется в состояние, которое было при последней фиксации. Вы можете выполнить срочную работу, а когда будете готовы, воспользуйтесь командой `git stash apply`, чтобы получить назад свою незавершенную работу. Это похоже на волшебство!

Резюме

Git содержит огромный арсенал инструментальных средств с множеством вариантов и возможностей применения. Поэтому в рамках этой главы было дано лишь краткое введение в систему контроля версий. Но если вы будете пользоваться хотя бы теми возможностями, которые описаны в этой главе, то почувствуете, насколько полезен Git как для защиты от потери данных, так и для повышения эффективности коллективного труда.

В этой главе были вкратце описаны лишь базовые команды Git. В ней было показано, как настроить хранилище перед импортом проекта, как извлечь, зафиксировать, обновить исходный код, а также как пометить и экспортировать его новую версию. В конце главы были рассмотрены ветви и продемонстрировано, насколько удобно поддерживать параллельные ветви для разработки и исправления ошибок в проекте.

Но остался еще один вопрос, который здесь не был затронут. Мы установили принцип, по которому разработчики должны сначала извлечь из удаленного хранилища собственные версии проекта. Но в общем случае проекты не стоят на месте. Чтобы проверить внесенные изменения, разработчики должны развернуть код локально на своем компьютере. Иногда для этого достаточно скопировать несколько каталогов. Но зачастую при развертывании приходится решать ряд вопросов, связанных с конфигурированием. В следующих главах мы рассмотрим некоторые методики автоматизации этого процесса.

ГЛАВА 18

Тестирование средствами PHPUnit

Успешное функционирование каждого компонента в системе зависит от согласованной работы и соблюдения интерфейса других компонентов. Только в этом случае возможно долговременное и бесперебойное функционирование системы. Но по мере разработки в системе, как правило, возникают проблемы. Улучшая классы и пакеты, нельзя забывать изменять код, который с ними связан. Так, внесение некоторых изменений может вызвать "волновой" эффект, в результате чего будут затронуты даже те компоненты, которые далеко отстоят от первоначально измененного кода. Зоркость и проницательность, а также энциклопедические знания зависимостей системы помогут в разрешении подобного затруднения. И хотя это отличные качества разработчика, сложность системы обычно нарастает так быстро, что вряд ли удастся легко предсказать всякий нежелательный эффект. Не последнюю роль в этом играет то обстоятельство, что над системой, как правило, работает много разработчиков. В качестве выхода из положения целесообразно регулярно тестировать каждый компонент. Это, конечно, сложная задача, которую к тому же приходится решать неоднократно, поэтому ее лучше автоматизировать.

Среди инструментальных средств тестирования, имеющихся в распоряжении программистов на PHP, самым распространенным и, безусловно, полнофункциональным является PHPUnit. В этой главе будут рассмотрены следующие вопросы применения PHPUnit.

- *Установка.* Применение Composer для установки PHPUnit.
- *Написание тестов.* Составление различных контрольных примеров и применение методов с утверждениями (assertion methods).
- *Обработка исключений.* Стратегии для подтверждения отказов.
- *Выполнение нескольких тестов.* Объединение тестов в наборы.
- *Построение логики утверждений.* Наложение ограничений.

- *Фиктивные компоненты.* Имитации и заглушки.
- *Написание веб-тестов.* Проведение тестов как с помощью дополнительных средств, так и без них.

Функциональные и модульные тесты

Тестирование имеет большое значение для любого проекта. И даже если этот процесс не формализован, то все равно придется составить неформальный список действий для проверки работоспособности системы и выявления в ней слабых мест. Но этот процесс может быстро стать надоедливым, а это может привести к тому, что проекты будут выполняться просто наудачу.

Один из подходов к тестированию начинается с интерфейса проекта, когда моделируются разные способы взаимодействия пользователя с проектируемой системой. Такой подход, вероятнее всего, пригоден для тестирования вручную, хотя существуют различные средства автоматизации данного процесса. Составляемые подобным способом функциональные тесты иногда называют приемочными, потому что список выполненных успешно действий может служить критерием завершения определенной стадии проекта. При таком подходе система обычно рассматривается как "черный ящик" — в тестах преднамеренно ничего не указывается о скрытых взаимодействующих компонентах, формирующих тестируемую систему.

Если функциональные тесты действуют извне, то модульные — изнутри. Модульное тестирование сосредоточено на классах, причем тестовые методы группируются в контрольные примеры. Каждый контрольный пример подвергает серьезному испытанию отдельный класс, проверяя, работает ли каждый метод, как предполагалось, и дает ли он сбой там, где это должно быть. Цель состоит в том, чтобы протестировать по возможности каждый компонент отдельно от более обширного контекста. И, как правило, это помогает лучше понять, насколько успешно развязаны отдельные части системы.

Тесты можно выполнять как часть процесса построения непосредственно из командной строки или даже через веб-страницу. В этой главе мы рассмотрим способ выполнения тестов из командной строки.

Модульное тестирование — это хороший способ убедиться в качестве проектирования системы. Модульные тесты выявляют обязанности отдельных классов и функций. Некоторые программисты даже отстаивают подход, предполагающий тестирование изначально: они считают, что

тесты следует писать еще до того, как начнется работа над классом. Это позволит установить назначение класса, создать понятный интерфейс и краткие, целенаправленные методы. Лично я никогда не стремлюсь к безукоризненности до такой степени, поскольку это не соответствует моему стилю программирования. Тем не менее я стараюсь писать тесты в процессе разработки. Поддержание средств тестирования обеспечивает уровень безопасности, требующийся мне для реорганизации кода. Я могу извлечь и заменить целые пакеты, если знаю, что у меня есть большие шансы найти в системе неожиданные ошибки.

Тестирование вручную

Как пояснялось в предыдущем разделе, тестирование очень важно для любого проекта. Но точнее было бы сказать, что тестирование *неизбежно* для любого проекта. Все мы что-нибудь тестируем. Но трагедия в том, что мы часто пренебрегаем этой важной обязанностью.

Итак, создадим несколько классов для тестирования. Ниже приведен класс, в котором хранятся сведения о пользователе и о том, откуда они извлекаются. С демонстрационными целями в этом классе формируются массивы, а не объекты типа `User`, которые обычно применяются в подобных случаях:

```
// Листинг 18.1

class UserStore
{
    private array $users = [];
    public function addUser(string $name, string $mail,
                            string $pass): bool
    {
        if (isset($this->users[$mail]))
        {
            throw new \Exception(
                "Пользователь {$mail} уже есть в системе"
            );
        }
        if (strlen($pass) < 5)
        {
            throw new \Exception(
                "Пароль должен быть не короче 5 символов"
            );
        }
```

```
        $this->users[$mail] = [
            'pass' => $pass,
            'mail' => $mail,
            'name' => $name
        ];
        return true;
    }
    public function notifyPasswordFailure(string $mail): void
    {
        if (isset($this->users[$mail]))
        {
            $this->users[$mail]['failed'] = time();
        }
    }
    public function getUser(string $mail): array
    {
        return ($this->users[$mail]);
    }
}
```

Сведения о пользователе передаются в этот класс с помощью метода `addUser()`, а для их извлечения вызывается метод `getUser()`. Адрес электронной почты пользователя служит в качестве ключа для извлечения сведений о нем. Если вы, как и я, пишете тестовый пример реализации в ходе разработки, чтобы убедиться, что все работает так, как и задумано, то такой пример будет выглядеть следующим образом:

```
// Листинг 18.2

$store = new UserStore();
$store->addUser(
    "bob williams",
    "bob@example.com",
    "12345"
);
$store->notifyPasswordFailure("bob@example.com");
$user = $store->getUser("bob@example.com");
print_r($user);
```

Вот как выглядит вывод этого кода:

```
Array
(
    [pass] => 12345
    [mail] => bob@example.com
    [name] => bob williams
    [failed] => 1609766967
)
```

Во время работы над классом этот фрагмент кода можно разместить в конце файла, который содержит класс. Безусловно, проверка правильности теста выполняется вручную: необходимо просмотреть результаты и убедиться, что данные, возвращаемые методом `UserStore::getUser()`, соответствуют сведениям о пользователе, которые были введены первоначально. Тем не менее это уже своего рода тест.

Ниже приведен клиентский класс, в котором класс `UserStore` используется для того, чтобы убедиться, что пользователь предоставил правильные сведения для аутентификации:

```
// Листинг 18.3

class Validator
{
    public function __construct(private UserStore $store)
    {
    }
    public function validateUser(string $mail, string $pass): bool
    {
        if (! is_array($user = $this->store->getUser($mail)))
        {
            return false;
        }

        if ($user['pass'] == $pass)
        {
            return true;
        }

        $this->store->notifyPasswordFailure($mail);
        return false;
    }
}
```

Данному классу требуется указать объект типа `UserStore`, ссылка на который сохраняется в свойстве `$store`. Это свойство используется в методе `validateUser()`, во-первых, чтобы убедиться, что пользователь по указанному адресу электронной почты действительно существует в хранилище, а во-вторых, чтобы проверить, что пароль пользователя совпадает с представленным аргументом. Если оба условия выполняются, метод `validateUser()` возвращает значение `true`. Тестирование можно выполнить по ходу разработки, как показано ниже:

```
// Листинг 18.4
$store = new UserStore();
$store->addUser("bob williams", "bob@example.com", "12345");
$validator = new Validator($store);

if ($validator->validateUser("bob@example.com", "12345"))
{
    print "Привет, друг!\n";
}
```

В данном фрагменте кода создается экземпляр `UserStore`, который заполняется данными и передается вновь созданному экземпляру `Validator`. Затем можно подтвердить правильность сочетания имени пользователя и пароля.

Удовлетворившись результатами своих трудов, эти тесты можно удалить или закомментировать. И хотя такие тесты служат ярким примером напрасного расходования ресурсов, они могут послужить основанием для тщательной проверки системы в процессе разработки. И в этом нам может помочь такое инструментальное средство, как PHPUnit.

Общее представление о PHPUnit

PHPUnit относится к семейству инструментальных средств тестирования xUnit. Им предшествовал каркас SUnit, созданный Кентом Беком (Kent Beck), для тестирования систем, построенных на языке Smalltalk. Система xUnit стала популярным инструментом благодаря реализации на Java, jUnit, и популярности таких методик, как экстремальное программирование (Extreme Programming — XP) и Scrum, в котором очень большое внимание уделяется тестированию.

Получить PHPUnit можно с помощью Composer:

```
{
    "require-dev": {
        "phpunit/phpunit": "^9"
    }
}
```

После запуска инсталлятора Composer в каталоге `vendor/bin` появится сценарий `phpunit`. Кроме того, вы можете загрузить архивный файл с расширением `.phar` и сделать его исполняемым, выполнив следующие команды:

```
$ wget https://phar.phpunit.de/phpunit.phar
$ chmod 755 phpunit.phar
$ sudo mv phpunit.phar /usr/local/bin/phpunit
```

На заметку Как и прежде, команды, которые должны вводиться в командной строке, предваряются символом доллара ($), чтобы отличать их от выводимых данных.

Создание контрольного примера

Имея такое инструментальное средство, как PHPUnit, можно написать тесты для проверки класса `UserStore`. Тесты каждого целевого компонента должны объединяться в одном классе, расширяющем класс `PHPUnit\Framework\TestCase`. Это один из классов, которые стали доступны благодаря пакету PHPUnit. Ниже показано, как создать минимальный класс для контрольного примера:

```
// Листинг 18.5

namespace popp\ch18\batch01;
use PHPUnit\Framework\TestCase;
class UserStoreTest extends TestCase
{
    protected function setUp(): void
    {
    }
    protected function tearDown(): void
    {
    }
}
```

Этому классу контрольного примера присвоено имя `UserStoreTest`. Нередко тест полезно разместить в том же месте, где находится тестируемый класс, чтобы упростить доступ к тестируемому классу и его партнерам, а также сделать структуру тестовых файлов такой же, как и в проектируемой системе. Напомним, что благодаря поддержке в Composer рекомендаций стандарта PSR-4 в одном и том же пакете можно поддерживать раздельные структуры каталогов для файлов классов.

Ниже показано, как это можно сделать в Composer:

```
"autoload": {
    "psr-4": {
        "popp\\": ["myproductioncode/", "mytestcode/"]
    }
}
```

Здесь указаны два каталога, отображающихся на пространство имен popp. Их можно поддерживать параллельно, чтобы упростить разделение кода на тестовый и рабочий.

Метод setUp() автоматически вызывается для каждого тестового метода, что позволяет создать устойчивую и соответствующим образом наполненную среду для теста. Метод tearDown() вызывается после выполнения каждого тестового метода. Если тесты изменяют более обширное окружение системы, то с помощью этого метода можно восстановить исходное состояние. Общая платформа, которой управляют методы setUp() и tearDown(), называется *тестовой конфигурацией (fixture)*.

Чтобы протестировать класс UserStore, потребуется его экземпляр. Мы можем получить его в методе setUp() и присвоить его свойству. Создадим также тестовый метод:

```
// Листинг 18.6

namespace popp\ch18\batch01;
use PHPUnit\Framework\TestCase;
class UserStoreTest extends TestCase
{
    private UserStore $store;
    protected function setUp(): void
    {
        $this->store = new UserStore();
    }
    protected function tearDown(): void
    {
    }
    public function testGetUser(): void
    {
        $this->store->addUser("bob williams", "a@b.com", "12345");
        $user = $this->store->getUser("a@b.com");
        $this->assertEquals("a@b.com", $user['mail']);
        $this->assertEquals("bob williams", $user['name']);
        $this->assertEquals("12345", $user['pass']);
    }
}
```

На заметку Напомним, что методы `setUp()` и `tearDown()` вызываются один раз для каждого тестового метода в проверяемом классе. Если вы хотите включить код, который будет запускаться один раз перед всеми методами тестирования в классе, можете реализовать метод `setUpBeforeClass()`. И наоборот, для кода, который должен запускаться после всех тестовых методов класса, реализуйте метод `tearDownAfterClass()`.

Тестовые методы следует называть таким образом, чтобы их имена начинались со слова `"test"` и не требовали аргументов. Дело в том, что работа с классом контрольного примера осуществляется с помощью рефлексии.

На заметку Подробнее о рефлексии читайте в главе 5, "Средства для работы с объектами".

Объект, выполняющий тесты, просматривает все методы в классе и вызывает только те из них, которые отвечают данному образцу (т.е. методы, имена которых начинаются со слова `"test"`).

В данном примере протестировано извлечение сведений о пользователе. Получать экземпляр типа `UserStore` для каждого теста не нужно, потому что это сделано в методе `setUp()`. А поскольку `setUp()` вызывается для каждого теста, свойство `$store` гарантированно содержит вновь полученный экземпляр объекта.

В теле метода `testGetUser()` фиктивные данные сначала передаются методу `UserStore::addUser()`, а затем извлекаются для проверки каждого их элемента.

Прежде чем выполнять рассматриваемый здесь тест, следует также иметь в виду, что в нем употребляются операторы `use` без конструкции `require` или `require_once`. Иными словами, мы полагаемся на автозагрузку. Поиск и включение файла автозагрузки выполняется автоматически, если вы установили PHPUnit с Composer и если файл автозагрузки для вашего проекта был сгенерирован в том же контексте. Однако это может быть не всегда. Например, я могу выполнить глобальную команду PHPUnit, которая ничего не знает о моей локальной автозагрузке, или у меня может быть загружен phar-файл. Как мне сообщить своим тестам в этом случае, как найти сгенерированный `autoload.php` файл? Я мог бы поместить инструкцию `require_once` в тестовый класс (или суперкласс), но это нарушит правило PSR-1, согласно которому файлы классов не должны иметь побочных эффектов. Самое простое — сообщить PHPUnit о файле `autoload.php` из командной строки:

```
$ phpunit src/ch18/batch01/UserStoreTest.php
            --bootstrap vendor/autoload.php

PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
. 1 / 1 (100%)
Time: 00:00.012, Memory: 4.00 MB
OK (1 test, 3 assertions)
```

Методы утверждений

В программировании утверждение — это инструкция или метод, позволяющий проверить предположения относительно некоторых аспектов системы. Используя утверждение, вы обычно определяете некоторое предположение, например о том, что переменная `$cheese` содержит значение `"blue"`, а переменная `$pie` — значение `"apple"`. Если это предположение не выполняется, генерируется некоторое предупреждение. Утверждения оказываются настолько удобным способом сделать систему более безопасной, что в PHP встроена их поддержка, которую можно отключить в контексте эксплуатации рабочей версии приложения.

На заметку Подробнее о поддержке утверждений в PHP читайте на соответствующей странице руководства по адресу `https://www.php.net/assert`.

PHPUnit поддерживает утверждения с помощью набора методов, которые можно вызывать либо статически, либо для экземпляра класса, расширяющего `PHPUnit\Framework\TestCase`.

В предыдущем примере я использовал метод `assertEquals()` класса `TestCase`. Он сравнивает два предоставленных аргумента и проверяет их эквивалентность. Если они не совпадают, метод тестирования помечается как неудачный тест. Подкласс `PHPUnit\Framework\TestCase` обеспечивает меня доступом к набору методов утверждения. Некоторые из этих методов перечислены в табл. 18.1.

Таблица 18.1. *Методы утверждений класса* `PHPUnit\Framework\TestCase`

Метод	Описание
`assertEquals($val1, $val2, $message)`	Завершается неудачно, если значения аргументов `$val1` и `$val2` не эквивалентны
`assertFalse($expression, $message)`	Вычисляет значение выражения `$expression`. Завершается неудачно, если значение выражения *не* равно `false`
`assertTrue($expression, $message)`	Вычисляет значение выражения `$expression`. Завершается неудачно, если значение выражения *не* равно `true`
`assertNotNull($val, $message)`	Завершается неудачно, если значение аргумента `$val` равно `null`
`assertNull($val, $message)`	Завершается неудачно, если значение аргумента `$val` не равно `null`
`assertSame($val1, $val2, $message)`	Завершается неудачно, если аргументы `$val1` и `$val2` *не* содержат ссылки на один и тот же объект или если они содержат переменные, имеющие разные типы и значения
`assertNotSame($val1, $val2, $message)`	Завершается неудачно, если аргументы `$val1` и `$val2` содержат ссылки на один и тот же объект или если они содержат переменные, имеющие одинаковые типы и значения
`assertMatchesRegularExpression($regexp, $val, $message)`	Завершается неудачно, если значение аргумента `$val1` не соответствует регулярному выражению в аргументе `$regexp`

Тестирование исключений

Цель программиста обычно состоит в том, чтобы заставить написанный им код *работать*, причем работать исправно. Как правило, такой подход распространяется и на тестирование, особенно если тестируется собственный код. При этом требуется проверить, работает ли метод так, как предполагалось. Поэтому очень легко забыть, насколько важно протестировать его *на отказ*, чтобы выяснить, насколько хорошо действует проверка ошибок в методе, генерирует ли он исключения, когда это необходимо, и делает

ли он это правильно, приводит ли он все в порядок после ошибки, если, например, операция перед тем, как возникла ошибка, была выполнена наполовину. Ваша роль как тестировщика — все это проверить. PHPUnit может оказать в этом помощь.

Ниже приведен тест, в котором проверяется поведение класса `UserStore`, когда операция завершается неудачно:

```
// Листинг 18.7
public function testAddUserShortPass(): void
{
    try
    {
        $this->store->addUser("bob williams",
                              "bob@example.com", "ff");
    }
    catch (\Exception $e)
    {
        $this->assertEquals(
                    "Пароль должен быть не короче 5 символов",
                    $e->getMessage());
        return;
    }

    $this->fail("Ожидается исключение слишком короткого пароля");
}
```

Если снова проанализировать метод `UserStore::addUser()`, то можно заметить, что в нем генерируется исключение, если пароль пользователя состоит менее чем из пяти символов. В приведенном выше тесте предпринимается попытка это подтвердить. Для этого в блоке `try` вводится пользователь с недопустимым паролем. Если при этом генерируется ожидаемое исключение, то все в порядке и выполнение кода переходит к блоку `catch`. Если же ожидаемое исключение не генерируется, то блок `catch` пропускается и управление передается следующей за ним инструкции, вызывающей метод `fail()`.

Чтобы проверить, сгенерировано ли исключение, можно также вызвать метод утверждения `expectException()`, которому передается имя ожидаемого типа генерируемого исключения (`Exception` или его подтипа). Если тестовый метод завершится без генерации правильного типа исключения, тест будет не пройден.

На заметку Метод `expectException()` появился в версии PHP 5.2.0.

Вот вторая реализация предыдущего теста:

```
// Листинг 18.8
public function testAddUserShortPassNew(): void
{
    $this->expectException(\Exception::class);
    $this->store->addUser("bob williams", "bob@example.com", "ff");
}
```

Если есть такой изящный способ тестирования исключений, то зачем я вообще показал старый подход? В большинстве случаев самый простой способ — использование `expectException()` — будет и самым лучшим. Однако иногда вы можете захотеть выполнить дальнейшие проверки исключений и состояния тестируемого объекта или убрать некоторые побочные эффекты. В таких случаях все еще может иметь смысл воспользоваться старым проверенным способом.

Выполнение наборов тестов

Если мы тестируем класс `UserStore`, то должны протестировать и класс `Validator`. Ниже приведена сокращенная версия класса `ValidatorTest`, в котором тестируется метод `Validator::validateUser()`:

```
// Листинг 18.9

namespace popp\ch18\batch02;
use PHPUnit\Framework\TestCase;
class ValidatorTest extends TestCase
{
    private Validator $validator;
    protected function setUp(): void
    {
        $store = new UserStore();
        $store->addUser("bob williams", "bob@example.com", "12345");
        $this->validator = new Validator($store);
    }
    public function testValidateCorrectPass(): void
    {
        $this->assertTrue(
            $this->validator->validateUser("bob@example.com",
                                           "12345"),
            "Ожидается успешная проверка"
        );
    }
}
```

Теперь, когда в нашем распоряжении уже имеется несколько контрольных примеров, возникает вопрос, как же выполнить их вместе? Все тестовые классы лучше всего разместить в общем корневом каталоге. Затем достаточно указать имя этого каталога в командной строке, как показано ниже, и PHPUnit запустит все тесты, которые в нем находятся:

```
$ phpunit src/ch18/batch02/

PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
........              8 / 8 (100%)
Time: 00:00.026, Memory: 6.00 MB
OK (8 tests, 11 assertions)
```

Ограничения

В большинстве случаев в тестах используются готовые утверждения. На самом деле с помощью одного только метода `AssertTrue()` можно сразу получить очень много информации. Но в версии PHPUnit 3.0 в класс `PHPUnit\Framework\TestCase` был включен набор фабричных методов, возвращающих объекты `PHPUnit\Framework\Constraint`. Их можно комбинировать и передавать методу `TestCase::AssertThat()`, чтобы строить собственные утверждения.

Обратимся к небольшому примеру. Объект `UserStore` не должен разрешать дублировать вводимые адреса электронной почты. Ниже приведен тест, который позволяет это проверить:

```
// Листинг 18.10
// UserStoreTest
public function testAddUserDuplicate()
{
    try
    {
        $ret = $this->store->addUser("bob williams", "a@b.com",
                                     "123456");
        $ret = $this->store->addUser("bob stevens", "a@b.com",
                                     "123456");
        $this->fail("Должно быть вызвано исключение");
    }
    catch (\Exception $e)
    {
        $const = $this->logicalAnd(
                    $this->logicalNot(
                           $this->containsEqual("bob stevens")),
```

```
                    $this->isType('array'),
                );
        $this->AssertThat($this->store->getUser("a@b.com"), $const);
    }
}
```

В этом тесте в объект типа UserStore добавляется пользователь, после чего добавляется второй пользователь с таким же адресом электронной почты. Тем самым этот тест подтверждает, что при втором вызове метода addUser() будет сгенерировано исключение. В блоке catch создается объект ограничения с использованием доступных подходящих методов. Они возвращают соответствующие экземпляры класса PHPUnit\Framework\Constraint. Рассмотрим следующее сложное ограничение из предыдущего примера:

```
$this->logicalNot($this->containsEqual("bob stevens"))
```

Здесь возвращается объект типа PHPUnit\Framework\Constraint\Traversable\TraversableContainsEqual. Когда он передается методу AssertThat, генерируется ошибка, если тестируемый объект не содержит элемент, соответствующий заданному значению ("bob stevens"). Далее благодаря передаче данного ограничения объекту PHPUnit\Framework\Constraint\Not выполняется отрицание. И вновь используется подходящий метод, доступный посредством класса TestCase (по сути, посредством суперкласса Assert):

```
$this->logicalNot( $this->contains("bob stevens"))
```

Теперь метод утверждения AssertThat() завершится неудачно, если тестируемое значение (которое должно пройти тест) содержит элемент, совпадающий с символьной строкой "bob stevens". Подобным образом можно построить достаточно сложные логические структуры с ограничениями. В конечном счете мое ограничение можно сформулировать следующим образом: “Не завершаться неудачно, если тестируемое значение является массивом и не содержит строку "bob stevens"”. Ограничение, накладываемое на тестируемое значение, передается вместе с ним методу AssertThat().

Безусловно, того же самого результата можно достичь и с помощью стандартных методов утверждений, но у ограничений имеется ряд преимуществ. Во-первых, они формируют логические блоки с четкими и ясными отношениями между компонентами (хотя для большей ясности иногда

имеет смысл использовать форматирование). Во-вторых, что еще важнее, ограничение можно использовать повторно. Из сложных ограничений можно создать целую библиотеку, чтобы применять их в различных тестах. Сложные ограничения можно даже объединять:

```
$const = $this->logicalAnd(
            $a_complex_constraint,
            $another_complex_constraint
        );
```

В табл. 18.2 приведены некоторые методы ограничений, имеющиеся в классе TestCase.

Таблица 18.2. *Некоторые методы ограничений*

Метод `TestCase`	Завершается неудачно при несоблюдении условия
`greaterThan($num)`	Тестовое значение больше, чем у аргумента `$num`
`containsEqual($val)`	Тестовое (перебираемое) значение содержит элемент, совпадающий со значением аргумента `$val`
`identicalTo($val)`	Тестовое значение является ссылкой на тот же объект, что и `$val`, а если это не объект, то на такие же тип и значение
`greaterThanOrEqual($num)`	Тестовое значение больше или равно значению аргумента `$num`
`lessThan($num)`	Тестовое значение меньше значения аргумента `$num`
`lessThanOrEqual($num)`	Тестовое значение меньше или равно значению аргумента `$num`
`equalTo($value)`	Тестовое значение равно значению аргумента `$value`
`equalTo($value, $delta=0)`	Тестовое значение равно значению аргумента `$value`. Аргумент `$delta` определяет погрешность для числовых сравнений
`stringContains($str, $casesensitive=true)`	Тестовое значение содержит символьную строку `$str`. По умолчанию учитывает регистр букв
`matchesRegularExpression ($pattern)`	Тестовое значение соответствует регулярному выражению, указанному в аргументе `$pattern`

Окончание табл. 18.2

Метод `TestCase`	Завершается неудачно при несоблюдении условия
`logicalAnd(PHPUnit\Framework\Constraint $const [, $const..])`	Все заданные ограничения должны пройти тест
`logicalOr( PHPUnit\Framework\Constraint $const [, $const..])`	По крайней мере одно из заданных ограничений должно пройти тест
`logicalNot(PHPUnit\Framework\Constraint $const)`	Заданное ограничение не проходит тест

Имитации и заглушки

Цель модульных тестов — в максимально возможной степени проверить компонент отдельно от содержащей его системы. Но только немногие компоненты существуют в изоляции. Даже тщательно разделенным классам требуется доступ к другим объектам с помощью аргументов методов. Многие классы также непосредственно работают с базами данных или файловой системой.

Мы уже рассматривали один из способов решения подобных задач. Методы `setUp()` и `tearDown()` можно использовать для управления средой тестирования, т.е. общим набором ресурсов для проведения тестов, который может включать в себя подключение к базе данных, конфигурируемые объекты, временные файлы в файловой системе и т.д.

Еще один способ состоит в том, чтобы сымитировать контекст тестируемого класса. Для этого нужно создать объекты, подражающие объектам, выполняющим настоящие функции. Например, поддельный объект преобразователя базы данных можно передать конструктору тестового объекта. А поскольку этот поддельный объект имеет общий тип с реальным классом преобразователя (расширяется из общей абстрактной базы или даже замещает настоящий класс), субъект тестирования не заметит подмены. Поддельный объект можно заполнить вполне допустимыми данными. Объекты, образующие своего рода “песочницу” для модульных тестов, называются *заглушками* (*stubs*). Они могут быть полезны, потому что позволяют сосредоточиться на классе, который требуется протестировать, не тестируя при этом всю систему.

Применение поддельных объектов этим не ограничивается. Тестируемые объекты могут каким-то образом вызывать методы поддельного объекта, и поэтому всегда имеется возможность убедиться в том, что выполняется тот вызов, который вы ожидаете. Такое применение поддельного объекта в качестве "шпиона" называется *проверкой поведения*, и именно этим имитирующий объект отличается от заглушки.

Имитирующие объекты можно строить самостоятельно, создавая классы, в которых жестко закодированы возвращаемые значения и выдаются сообщения о вызовах методов. И хотя это несложный процесс, он все же отнимает время.

В PHPUnit доступно более простое решение, позволяющее автоматически генерировать имитирующие объекты в динамическом режиме. С этой целью в PHPUnit сначала исследуется класс, который требуется имитировать, а затем создается дочерний класс, переопределяющий его методы. Получив экземпляр имитирующего объекта, можно вызвать для него методы, чтобы заполнить его данными и определить условия прохождения теста.

Обратимся к конкретному примеру класса `UserStore`, содержащего метод `notifyPasswordFailure()`, в котором устанавливается значение поля для заданного пользователя. Этот метод должен вызываться объектом типа `Validator` при вводе пользователем неправильного пароля. В приведенном ниже тесте мы сымитировали класс `UserStore` таким образом, чтобы он предоставлял данные объекту типа `Validator` и подтверждал, что его метод `notifyPasswordFailure()` вызван, как и предполагалось:

```
// Листинг 18.11
// ValidatorTest
public function testValidateFalsePass(): void
{
    $store = $this->createMock(UserStore::class);
    $this->validator = new Validator($store);
    $store->expects($this->once())
          ->method('notifyPasswordFailure')
          ->with($this->equalTo('bob@example.com'));
    $store->expects($this->any())
          ->method("getUser")
          ->will($this->returnValue([
                                "name" => "bob williams",
                                "mail" => "bob@example.com",
                                "pass" => "right"
                            ]));
    $this->validator->validateUser("bob@example.com", "wrong");
}
```

Имитирующие объекты используют *текучий интерфейс* (*fluent interface*), т.е. языкоподобную структуру. Их намного легче применять, чем описывать. Такие конструкции действуют слева направо, и в результате каждого вызова возвращается ссылка на объект, обратиться к которому можно, вызвав затем следующий модифицирующий метод, в свою очередь возвращающий объект. Это облегчает применение имитирующих объектов, но затрудняет отладку проверяемого кода.

В предыдущем примере мы вызвали метод `TestCase::createMock()`, передав ему имя класса (`"UserStore"`), который требуется сымитировать. Этот метод динамически сформирует класс и создаст экземпляр его объекта. Имитирующий объект затем сохраняется в свойстве `$store` и передается объекту `Validator`. Это не приводит к ошибке, потому что класс вновь созданного объекта расширяет класс `UserStore`. Тем самым мы обманули класс `Validator`, заставив его принять "шпиона" за настоящий объект.

У имитирующих объектов, сформированных средствами PHPUnit, имеется метод `expects()`. Ему требуется передать объект сопоставителя (на самом деле он относится к типу `PHPUnit\Framework\MockObject\Matcher\Invocation`, но не беспокойтесь; для генерации объекта сопоставителя можно воспользоваться подходящими методами из класса `TestCase`). Объект определяет мощность ожидания, т.е. сколько раз метод должен быть вызван.

В табл. 18.3 перечислены методы сопоставителя, доступные в классе `TestCase`.

Таблица 18.3. *Некоторые методы сопоставителя*

Метод класса	Завершается неудачно при несоблюдении условия
`any()`	Сделано нуль или более вызовов соответствующего метода (полезно для объектов заглушек, возвращающих значения, но не тестирующих вызовы)
`never()`	Не сделано ни одного вызова соответствующего метода
`atLeastOnce()`	Сделан один или несколько вызовов соответствующего метода
`once()`	Сделан единственный вызов соответствующего метода
`exactly($num)`	Сделано количество вызовов соответствующего метода, равное `$num`
`at($num)`	Вызов соответствующего метода сделан по индексу `$num` (каждый вызов метода для имитирующего объекта регистрируется и индексируется)

Задав требование для сопоставления, мы должны указать метод, к которому оно применяется. Например, метод expects() возвращает объект (типа PHPUnit\Framework\MockObject\Builder\InvocationMocker), у которого имеется метод method(). Его можно вызвать просто по имени метода. Этого окажется достаточно для небольшой имитации, как показано ниже:

```
// Листинг 18.12
$store->expects($this->once())
    -> method('notifyPasswordFailure');
```

Но мы должны пойти еще дальше, проверив параметры, переданные методу notifyPasswordFailure(). Метод InvocationMocker::method() возвращает экземпляр объекта, для которого он был вызван. В состав класса InvocationMocker входят метод with(), которому передается переменный список параметров для сопоставления, а также объекты ограничений, что дает возможность тестировать диапазоны и т.д. Вооружившись всем этим, мы можем завершить приведенный выше оператор, гарантировав передачу ожидаемого параметра методу notifyPasswordFailure():

```
// Листинг 18.13
$store->expects($this->once())
      ->method('notifyPasswordFailure')
      ->with($this->equalTo('bob@example.com'));
```

Теперь должно быть понятно, почему такая конструкция называется "текучим интерфейсом". Она похожа на предложение. Назначение приведенной выше инструкции можно сформулировать следующим образом: "Объект $store *ожидает* единственный вызов метода notifyPasswordFailure() с параметром bob@example.com".

Обратите внимание на то, что накладываемое ограничение было передано методу with(). На самом деле это излишне, поскольку любые явные аргументы преобразуются в ограничения внутренним образом. Поэтому рассматриваемый здесь оператор можно упростить следующим образом:

```
// Листинг 18.14
$store->expects($this->once())
      -> method('notifyPasswordFailure')
      ->with('bob@example.com');
```

Иногда имитирующие объекты применяются в PHPUnit только как заглушки, т.е. как объекты, возвращающие значения, которые позволяют выполнять тесты. В подобных случаях можно сделать вызов

InvocationMocker::will() из метода method(). Методу will() требуется значение (или несколько значений, если метод вызывается неоднократно), которое должен возвратить связанный с ним метод. Это возвращаемое значение можно передать, сделав вызов TestCase::returnValue() или TestCase::onConsecutiveCalls(). И сделать это намного проще, чем описать. Ниже приведен фрагмент кода одного из предыдущих примеров, в которых объект типа UserStore был заполнен для возврата значения:

```
// Листинг 18.15
$store->expects($this->any())
      ->method("getUser")
      ->will($this->returnValue([
             "name" => "bob@example.com",
             "pass" => "right"
     ]));
```

На заметку TestCase::returnValue() и TestCase::onConsecutiveCalls() — не единственные методы, которые вы можете использовать для установки возвращаемых значений с помощью заглушек. Имеются также методы returnValueMap(), returnArguments(), returnCallback() и returnSelf().

Сначала мы заполняем имитирующий объект типа UserStore данными, ожидая произвольного количества вызовов метода getUser(), хотя нас теперь больше волнует вопрос предоставления данных, а не тестирования вызовов. Затем мы вызываем метод will() с результатом вызова метода TestCase::returnValue(), в котором указаны данные, которые требуется возвратить (в данном случае это объект типа PHPUnit\Framework\MockObject\Stub\ReturnStub, хотя с таким же успехом можно было бы просто вспомнить удобный метод, который использовался для его получения).

С другой стороны, можно передать методу will() результат вызова метода TestCase::onConsecutiveCalls(). Последнему можно передать произвольное количество параметров, каждый из которых будет возвращен сымитированным методом при последующих вызовах.

Тесты достигают своей цели, когда завершаются неудачно

Хотя большинство согласится, что тестирование — дело благое, по достоинству его можно оценить лишь после того, как оно хотя бы несколько раз спасет положение. Представьте ситуацию, когда изменение в одной части системы приводит к неожиданному результату в другой.

Созданный ранее класс `UserStore` уже работал какое-то время, когда в ходе анализа кода было установлено, что в нем лучше формировать объекты типа `User`, а не ассоциативные массивы. Ниже приведена новая версия класса `UserStore`:

```
// Листинг 18.16
namespace popp\ch18\batch03;
class UserStore
{
    private array $users = [];
    public function addUser(string $name, string $mail,
                            string $pass): bool
    {
        if (isset($this->users[$mail]))
        {
            throw new \Exception(
                "Пользователь {$mail} уже есть в системе"
            );
        }
        $this->users[$mail] = new User($name, $mail, $pass);
        return true;
    }
    public function notifyPasswordFailure(string $mail): void
    {
        if (isset($this->users[$mail]))
        {
            $this->users[$mail]->failed(time());
        }
    }
    public function getUser(string $mail): ? User
    {
        if (isset($this->users[$mail]))
        {
            return ($this->users[$mail]);
        }
        return null;
    }
}
```

А вот как выглядит простой класс `User`:

```
// Листинг 18.17
namespace popp\ch18\batch03;
class User
{
    private string $pass;
    private ? string $failed;
    public function __construct(private string $name,
                                private string $mail,
                                string $pass)
    {
        if (strlen($pass) < 5)
        {
            throw new \Exception(
                "Пароль должен быть не короче 5 символов"
            );
        }
        $this->pass = $pass;
    }
    public function getMail(): string
    {
        return $this->mail;
    }
    public function getPass(): string
    {
        return $this->pass;
    }
    public function failed(string $time): void
    {
        $this->failed = $time;
    }
}
```

Безусловно, нам придется внести соответствующие поправки в класс `UserStoreTest`, чтобы учесть эти изменения. В частности, рассмотрим следующий фрагмент кода, предназначенный для обработки массива:

```
// Листинг 18.18
public function testGetUser()
{
    $this->store->addUser("bob williams", "a@b.com", "12345");
    $user = $this->store->getUser("a@b.com");
    $this->assertEquals($user['mail'], "a@b.com");
    $this->assertEquals($user['name'], "bob williams");
    $this->assertEquals($user['pass'], "12345");
}
```

Теперь он преобразуется в код, предназначенный для обработки объекта, следующим образом:

```
// Листинг 18.19
public function testGetUser(): void
{
    $this->store->addUser("bob williams", "a@b.com", "12345");
    $user = $this->store->getUser("a@b.com");
    $this->assertEquals($user->getMail(), "a@b.com");
}
```

Но когда мы запускаем на выполнение набор тестов по следующей команде, то получаем в награду предупреждение о том, что наша работа еще не завершена:

```
$ phpunit src/ch18/batch03/

PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
....F                5 / 5 (100%)
Time: 00:00.019, Memory: 6.00 MB

There was 1 failure:
1) popp\ch18\batch03\ValidatorTest::testValidateCorrectPass
Expecting successful validation
Failed asserting that false is true.
/var/popp/src/ch18/batch03/ValidatorTest.php:26

FAILURES!
Tests: 5, Assertions: 5, Failures: 1.
```

Хотя мои тесты связаны с `User`, мой класс `ValidatorTest` обнаружил тот факт, что я не обновил `Validator` для учета нового возвращаемого значения. Вот тест, который не проходит:

```
// Листинг 18.20
public function testValidateCorrectPass(): void
{
    $this->assertTrue(
        $this->validator->validateUser("bob@example.com", "12345"),
        "Ожидается успешная проверка"
    );
}
```

А вот метод `Validator::validateUser()`, который меня подвел:

```
// Листинг 18.21
public function validateUser($mail, $pass): bool
{
```

```
    if (! is_array($user = $this->store->getUser($mail)))
    {
        return false;
    }
    if ($user['pass'] == $pass)
    {
        return true;
    }
    $this->store->notifyPasswordFailure($mail);
    return false;
}
```

Итак, `User::getUser()` теперь возвращает объект, а не массив. `getUser()` изначально в случае успеха возвращал массив, содержащий данные пользователя, или `null` — в случае неудачи. Я проверял пользователей путем проверки массива с использованием функции `is_array()`. Теперь, конечно, это условие никогда не выполняется, и метод `validateUser()` всегда будет возвращать `false`. Без тестового каркаса `Validator` просто отклонил бы всех пользователей как недействительных безо всяких предупреждений.

Вот относительно быстрое решение, позволяющее привести в действие метод `validateUser()`:

```
// Листинг 18.22
public function validateUser($mail, $pass): bool
{
    $user = $this->store->getUser($mail);

    if (is_null($user))
    {
        return false;
    }
    $testpass = $user->getPass();
    if ($testpass == $pass)
    {
        return true;
    }
    $this->store->notifyPasswordFailure($mail);
    return false;
}
```

А теперь представьте, что это небольшое изменение в `User::getUser()` вы сделали в пятницу вечером без соответствующего тестирования. И теперь вас атакуют возмущенными посланиями, которые не дают вам спокойно отдохнуть дома, в баре или ресторане: “Что вы сделали? Все наши клиенты заблокированы!”

Самыми коварными оказываются те ошибки, которые не приводят к сообщению интерпретатора, что в исполняемом коде что-то пошло не так. Они скрываются за вполне допустимым кодом и незаметно нарушают логику работы системы. Более того, многие ошибки проявляются не в том месте, где были внесены изменения. Их причина находится именно там, но ее последствия могут проявиться в каком-нибудь другом месте через несколько дней и даже недель. Среда тестирования помогает обнаружить по крайней мере некоторые из них и тем самым предотвратить, а не обнаружить отказы в системах.

Пишите тесты по мере создания нового кода и выполняйте их как можно чаще. Если кто-нибудь сообщит о программной ошибке, сначала добавьте тест в среду тестирования, чтобы убедиться в ее наличии, а затем исправьте ошибку так, чтобы тест был успешно пройден. Ошибкам присуща странная особенность снова появляться в одном и том же месте. Написание тестов с целью убедиться в наличии ошибок, а затем защитить их исправление от последующих осложнений называется *регрессионным тестированием*. Кстати, если у вас имеется отдельный каталог для регрессионных тестов, не забудьте присвоить содержательные имена находящимся в нем файлам. Как-то раз в одном проекте наша команда разработчиков решила называть регрессионные тесты по номерам ошибок. В результате у нас получился каталог, содержащий 400 файлов тестов, у каждого из которых было имя наподобие `test_973892.php`. Можете себе представить, во что превратился поиск отдельного теста...

Написание веб-тестов

Свое веб-приложение вы должны проектировать таким образом, чтобы его можно было легко вызвать как из командной строки, так и с помощью программного интерфейса API. В главе 12, “Шаблоны корпоративных приложений”, уже рассматривался ряд приемов, которые могут вам в этом помочь. Так, если вы создадите класс `Request` для хранения всех данных HTTP-запроса, то сможете легко заполнить его экземпляр данными, полученными при вызове вашего приложения из командной строки или из списка аргументов, переданных методу. И тогда ваше приложение сможет нормально работать независимо от его контекста.

Если ваше приложение не так просто запустить на выполнение в разных контекстах, это означает, что при его проектировании была допущена

оплошность. Так, если пути к многочисленным файлам жестко закодированы в ваших компонентах, то вы, вероятнее всего, испытываете на себе отрицательные последствия тесной связи. В таком случае вы должны переместить элементы, жестко привязанные к своему контексту, в инкапсулирующие их объекты, которые можно запросить из центрального хранилища данных. В главе 12 был также описан проектный шаблон `Registry`, который может оказать вам в этом помощь.

Как только ваше веб-приложение можно будет запустить на выполнение, непосредственно вызвав один из методов, вы с удивлением обнаружите, что для него совсем не трудно написать высокоуровневые веб-тесты без привлечения дополнительных инструментальных средств.

Тем не менее вы можете обнаружить, что для выполнения таких тестов даже в тщательно продуманных проектах потребуется некоторая реорганизация кода. По своему опыту могу сказать, что подобная мера всегда приводит к улучшению проекта. Продемонстрируем это положение, подогнав один из аспектов примера системы Woo, описанного в главах 12 и 13, под модульное тестирование.

Рефакторинг кода веб-приложения для тестирования

Разработка примера системы Woo остановлена в состоянии, пригодном для тестирования. Поскольку в данном примере был применен единственный шаблон `Front Controller`, в нашем распоряжении имеется простой интерфейс API. Вот простой сценарий из файла `Runner.php`:

```
// Листинг 18.23
require_once("vendor/autoload.php");

use popp\ch18\batch04\woo\controller\Controller;

Controller::run();
```

Ввести здесь модульный тест будет нетрудно, но как быть с аргументами командной строки? Эта задача в какой-то степени уже решена в классе `Request`, как показано ниже:

```
// Листинг 18.24
public function init()
{
    if (isset($_SERVER['REQUEST_METHOD']))
    {
```

```
        if ($_SERVER['REQUEST_METHOD'])
        {
            $this->properties = $_REQUEST;
            return;
        }
    }

    foreach ($_SERVER['argv'] as $arg)
    {
        if (strpos($arg, '='))
        {
            list($key, $val) = explode("=", $arg);
            $this->setProperty($key, $val);
        }
    }
}
```

На заметку Напомним, что если вы собираетесь реализовать свой класс `Request`, то должны перехватывать и сохранять данные, полученные с помощью методов запроса `GET`, `POST` и `PUT`, в отдельных свойствах, а не выводить их в одно свойство `$request`.

В методе `init()` проверяется, в каком именно контексте (сервера или командной строки) работает данное приложение, и соответственно заполняется массив `$properties` (непосредственным присваиванием или через метод `setProperty()`). Это означает, что данное приложение вполне пригодно для вызова из командной строки. Так, после ввода команды

```
$ php src/ch18/batch04/Runner.php cmd=AddVenue venue_name=bob
```

будет получен следующий ответ:

```
<html>
<head>
    <title>Add a Space for venue bob</title>
</head>
<body>
<h1>Add a Space for Venue 'bob'</h1>
<table>
    <tr><td>'bob' added (22)</td></tr>
    <tr><td>please add name for the space</td></tr>
</table> [add space]
<form method="post">
    <input type="text" value="" name="space_name"/>
    <input type="hidden" name="cmd" value="AddSpace" />
    <input type="hidden" name="venue_id" value="22" />
```

```
    <input type="submit" value="submit" />
</form>
</body>
</html>
```

И хотя данное приложение пригодно для вызова из командной строки, ему нелегко передать аргументы через вызов метода. Не очень изящное решение этой задачи состоит, например, в том, чтобы сформировать вручную массив параметров `$argv` перед вызовом метода `run()` контроллера. Не удивительно, что такое решение вряд ли может понравиться. При непосредственной работе с загадочными массивами возникает чувство, что что-то идет не так, а операции со строками, которые нужно выполнять в любом случае, могут привести к ошибке. Но более тщательный анализ класса контроллера `Controller` позволяет найти подходящее проектное решение. Рассмотрим, в частности, исходный код метода `handleRequest()`:

```
// Листинг 18.25
// Controller
public function handleRequest()
{
    $request = ApplicationRegistry::getRequest();
    $app_c = ApplicationRegistry::appController();
    while ($cmd = $app_c->getCommand($request))
    {
        $cmd->execute($request);
    }
    $this->invokeView($app_c->getView($request));
}
```

Этот метод предназначен для вызова из статического метода `run()`. Обратите внимание на то, что экземпляр объекта типа `Request` не получается непосредственно, а запрашивается из реестра `ApplicationRegistry`. Если в реестре хранится единственный экземпляр объекта типа `Request`, мы можем получить ссылку на него и загрузить его данными непосредственно из теста, прежде чем запускать проверяемую систему на выполнение через вызов контроллера. Подобным образом можно сымитировать веб-запрос. А поскольку в данной системе объект типа `Request` служит лишь в качестве интерфейса для веб-запроса, то она отвязана от источника данных. И если объект `Request` вполне работоспособен, то системе безразлично, откуда поступают данные: из теста или от веб-сервера. В качестве общего правила рекомендуется переносить экземпляры объектов обратно в реестр везде, где это только возможно.

Если все мои объекты созданы одним ApplicationRegistry, я могу перегрузить статический метод фабрики реестра (ApplicationRegistry::instance) и получить полный контроль над всеми данными, которые мое приложение использует во время тестирования. Этот подход возвращает имитацию реестра, заполненного при установке флага поддельными компонентами (тем самым создавая полностью имитированную среду выполнения). Я люблю обманывать свои системы.

Здесь, однако, я продемонстрирую первый, более консервативный трюк, предварительно загрузив объект Request с тестовыми данными.

Простое веб-тестирование

Ниже приведен контрольный пример выполнения очень простого теста системы Woo:

```
// Листинг 18.26

namespace popp\ch18\batch04;
use popp\ch18\batch04\woo\base\ApplicationRegistry;
use popp\ch18\batch04\woo\controller\ApplicationHelper;
use PHPUnit\Framework\TestCase;
class AddVenueTest extends TestCase
{
    public function testAddVenueVanilla(): void
    {
        $this->runCommand("AddVenue", ["venue_name" => "bob"]);
    }
    private function runCommand($command = null,
                                array $args = null): void
    {
        $reg = ApplicationRegistry::instance();
        $applicationHelper = ApplicationHelper::instance();
        $applicationHelper->init();
        $request = ApplicationRegistry::getRequest();
        if (! is_null($args))
        {
            foreach ($args as $key => $val)
            {
                $request->setProperty($key, $val);
            }
        }
        if (! is_null($command))
        {
            $request->setProperty('cmd', $command);
```

```
        }
        woo\controller\Controller::run();
    }
}
```

На самом деле этот тест ничего особенного не проверяет, а только позволяет убедиться, что проверяемая система может быть запущена на выполнение через вызов метода. Нечто реальное выполняется в методе `runCommand()`, в котором нет ничего особенного. Сначала из реестра типа `ApplicationRegistry` запрашивается объект типа `Request`, а затем он заполняется данными в виде ключей и значений, переданных методу `runCommand()` в качестве параметров. А поскольку объект типа `Controller` будет обращаться к тому же самому источнику для получения объекта типа `Request`, то можно не сомневаться, что он будет работать с заданными нами значениями.

Выполнение этого теста подтверждает, что все нормально! Получаемый результат отвечает нашим ожиданиям. Но дело в том, что этот результат выводится с помощью представления, и поэтому его нелегко проверить. Это затруднение можно очень просто устранить, буферизировав выводимые данные, как показано ниже:

```
// Листинг 18.27
namespace popp\ch18\batch04;
use popp\ch18\batch04\woo\base\ApplicationRegistry;
use popp\ch18\batch04\woo\controller\ApplicationHelper;
use PHPUnit\Framework\TestCase;
class AddVenueTest2 extends TestCase
{
    public function testAddVenueVanilla(): void
    {
        $output = $this->runCommand("AddVenue",
                                    ["venue_name" => "bob"]);
        self::AssertMatchesRegularExpression("/added/", $output);
    }
    private function runCommand($command = null, array $args = null):
        string
    {
        $applicationHelper = ApplicationHelper::instance();
        $applicationHelper->init();
        ob_start();
        $request = ApplicationRegistry::getRequest();
        if (! is_null($args))
        {
            foreach ($args as $key => $val)
```

```
            {
                $request->setProperty($key, $val);
            }
        }
        if (! is_null($command))
        {
            $request->setProperty('cmd', $command);
        }
        woo\controller\Controller::run();
        $ret = ob_get_contents();
        ob_end_clean();
        return $ret;
    }
}
```

Перенаправив вывод данных из системы в буфер, мы можем возвратить его из метода `runCommand()`. Чтобы проверить полученный результат, к возвращаемым данным мы применяем простое утверждение. Конечно, при таком подходе имеется ряд проблем.

Вот как выглядит тест из командной строки:

```
$ phpunit src/ch18/batch04/AddVenueTest2.php

PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
.                          1 / 1 (100%)
Time: 00:00.029, Memory: 6.00 MB
OK (1 test, 1 assertion)
```

Если вы планируете выполнять в своей системе многочисленные тесты описанным выше способом, то имеет смысл создать суперкласс для пользовательского веб-интерфейса, разместив в нем метод `runCommand()`.

Мы опустили здесь ряд деталей, которые придется учесть, выполняя собственные тесты. В частности, тестируемая система должна работать с изменяемыми в процессе конфигурации источниками данных. Очевидно, что не стоит выполнять тесты с тем же хранилищем данных, которое используется в процессе разработки приложения. И это еще один удобный случай улучшить проектируемую систему. Выявите в прикладном коде жестко закодированные имена и пути к файлам, а также строки, определяющие источники данных DSN, и перенесите их в объект типа `Registry`. Затем задайте все требуемые значения для контрольного примера в методе `setUp()`, чтобы ваши тесты выполнялись только в пределах одной "песочницы". И, наконец, рассмотрите возможность замены объекта типа `ApplicationRegistry` объектом типа `MockRequestRegistry`, способ-

ным снабдить проверяемую систему заглушками, имитирующими объектами и прочими хитрыми подделками.

Описанные выше подходы вполне пригодны для тестирования входных и выходных данных веб-приложения. Но у них имеются и явные ограничения. Вызываемый метод никак не заменит браузер. И если в вашем приложении применяются JavaScript и другие клиентские технологии, то тестирование сформированного в нем кода никак не поможет ответить на вопрос, увидел ли пользователь на экране нечто вразумительное.

К счастью, имеется и другое решение.

Введение в Selenium

Система Selenium (`http://seleniumhq.org/`) состоит из набора команд, которые позволяют автоматизировать тесты для веб-приложения. Кроме того, в ней имеются средства и интерфейсы API для написания и выполнения тестов в среде браузера.

В этом небольшом введении мы создадим быстрый тест системы Woo, рассмотренной в главе 12. Этот тест будет работать вместе с Selenium Server через интерфейс API, называемый веб-драйвером для PHP.

Получение и установка Selenium

Компоненты Selenium можно загрузить по адресу `http://seleniumhq.org/download/`. Для выполнения рассматриваемого здесь примера понадобится сервер Selenium Server. Загрузив установочный пакет, вы должны обнаружить архивный файл `selenium-server-standalone-3.141.59.jar`, хотя к тому моменту, когда вы будете читать эти строки, номер версии уже может измениться. Скопируйте этот файл в какое-нибудь подходящее место. Для дальнейших действий вам придется установить в своей системе Java. Сделав это, вы сможете запустить сервер Selenium Server.

Ниже приведены команды для копирования архивного файла с сервером Selenium Server в подкаталог `/usr/local/lib` и запуска этого сервера на выполнение:

```
$ sudo cp selenium-server-standalone-3.141.59.jar /usr/local/lib/
$ java -jar /usr/local/lib/selenium-server-standalone-3.141.59.jar

17:58:20.098 INFO [GridLauncherV3.parse] - Selenium server version:
    3.141.59, revision: e82be7d358
17:58:20.200 INFO [GridLauncherV3.lambda$buildLaunchers$3] -
    Launching a standalone Selenium Server on port 4444
```

```
2020-09-13 17:58:20.254:INFO::main: Logging initialized @678ms to
    org.seleniumhq.jetty9.util.log.StdErrLog
17:58:20.459 INFO [WebDriverServlet.<init>] -
    Initialising WebDriverServlet
17:58:20.541 INFO [SeleniumServer.boot] -
    Selenium Server is up and running on port 4444
```

Обратите внимание на то, что при запуске сервера сообщается порт, по которому можно посылать ему запросы. Этот порт понадобится нам чуть позже.

Однако мы только на полпути. Чтобы уменьшить вероятность возникновения непонятных ошибок, лучше всего загрузить правильную версию ChromeDriver — библиотеки, которая передает в браузер команды пользовательского интерфейса. В настоящее время представляется, что лучшим выбором браузера для тестирования с Selenium является Chrome. Начните с установки Chrome в вашей локальной системе, если вы еще этого не сделали. Определите версию своего браузера, просмотрев справку Chrome. Затем загрузите версию ChromeDriver, которая соответствует версии вашего браузера, по адресу `https://sites.google.com/a/chromium.org/chromedriver/downloads`. Вооружившись этой библиотекой, вы можете снова запустить Selenium:

```
$ java -jar -Dwebdriver.chrome.driver="./chromedriver"
    /usr/local/lib/selenium-server- standalone-3.141.59.jar
```

Теперь все готово для тестирования.

PHPUnit и Selenium

Если в прошлом для работы с Selenium в PHPUnit предоставлялись интерфейсы API, то теперь такая поддержка разрозненна, не говоря уже о соответствующей документации. Следовательно, чтобы сделать доступными многочисленные средства Selenium, целесообразно пользоваться PHPUnit вместе с инструментальным средством, специально предоставляющим необходимые привязки.

Общее представление о `php-webdriver`

Веб-драйвер (WebDriver) — это механизм, посредством которого можно управлять браузером, начиная с версии Selenium 2. Разработчики Selenium предусмотрели интерфейсы API веб-драйвера для таких языков, как Java, Python и C#. Что же касается языка PHP, то для него доступно несколь-

ко интерфейсов API. Здесь выбран для использования php-webdriver, созданный разработчиками Facebook. В настоящий момент он находится в стадии активной разработки и соответствует официальному API. Это очень удобная методика, поскольку многие примеры ее применения, которые можно найти в Интернете, предлагаются на Java. Это означает, что они вполне пригодны и для применения веб-драйвера, хотя и с незначительным переносом кода на PHP.

Вы можете добавить веб-драйвер для PHP в свой проект с помощью Composer следующим образом:

```
{
    "require-dev": {
        "phpunit/phpunit": "9.*",
        "php-webdriver/webdriver" : "*"
    }
}
```

Внесите эти изменения в свой файл composer.json, а затем запустите composer update. Таким образом вы подготовитесь к проведению веб-тестов.

Создание тестового каркаса

Нам предстоит протестировать экземпляр приложения Woo, доступной в моей системе по адресу http://popp.vagrant.internal/webwoo/.

Начнем со следующего стереотипного кода тестового класса:

```
// Листинг 18.28
namespace popp\ch18\batch04;

use Facebook\WebDriver\Chrome\ChromeOptions;
use Facebook\WebDriver\Remote\DesiredCapabilities;
use Facebook\WebDriver\Remote\RemoteWebDriver;
use Facebook\WebDriver\Remote\WebDriverCapabilityType;
use PHPUnit\Framework\TestCase;

class SeleniumTest1 extends TestCase
{
    protected function setUp(): void
    {
    }
    public function testAddVenue(): void
    {
    }
}
```

В приведенном выше коде сначала указывается ряд классов веб-драйвера для PHP, которыми мы будем пользоваться в дальнейшем, а затем создается каркас тестового класса. А теперь можно попробовать что-нибудь протестировать.

Подключение к серверу Selenium

Вспомним, что при запуске сервера Selenium он выводит URL. Для подключения к этому серверу непосредственно из тестов нам нужно указать этот URL и массив возможностей (capabilities array) при обращении к классу `RemoteWebDriver`:

```
// Листинг 18.29
namespace popp\ch18\batch04;

use Facebook\WebDriver\Chrome\ChromeOptions;
use Facebook\WebDriver\Remote\DesiredCapabilities;
use Facebook\WebDriver\Remote\RemoteWebDriver;
use Facebook\WebDriver\Remote\WebDriverCapabilityType;
use PHPUnit\Framework\TestCase;

class SeleniumTest2 extends TestCase
{
    private $driver;
    protected function setUp(): void
    {
        $options = new ChromeOptions();
        $capabilities = DesiredCapabilities::chrome();
        $capabilities->setCapability(ChromeOptions::CAPABILITY,
                                     $options);
        $this->driver = RemoteWebDriver::create(
            "http://127.0.0.1:4444/wd/hub",
            $capabilities
        );
    }
    public function testAddVenue(): void
    {
    }
}
```

Если вы установили веб-драйвер для PHP с помощью Composer, то можете обнаружить полный список возможностей в файле класса `vendor/php-webdriver/webdriver/lib/Remote/WebDriverCapabilityType.php`. В целях демонстрации нам достаточно указать лишь имя браузера. Поэтому имя хоста, задаваемое в строковой переменной `$host`, и массив

возможностей `$capabilities` передаются в качестве параметров статическому методу `RemoteWebDriver::create()`, а полученная в итоге ссылка на объект сохраняется в свойстве `$driver`. Выполнив этот тест, мы должны увидеть, как Selenium открывает новое окно в браузере, подготавливая среду для выполнения последующих веб-тестов.

Написание теста

Итак, мы собираемся протестировать простую последовательность действий, выполняемых пользователями в проверяемом приложении. Для этого нам нужно перейти на страницу `AddVenue`, ввести заведение для проведения культурного мероприятия и место его проведения. В ходе выполнения этой последовательности действий нам придется взаимодействовать с тремя веб-страницами.

Ниже приведен соответствующий тест:

```
// Листинг 18.30
namespace popp\ch18\batch04;

use Facebook\WebDriver\Chrome\ChromeOptions;
use Facebook\WebDriver\Remote\DesiredCapabilities;
use Facebook\WebDriver\Remote\RemoteWebDriver;
use Facebook\WebDriver\Remote\WebDriverCapabilityType;
use Facebook\WebDriver\WebDriverBy;
use PHPUnit\Framework\TestCase;

class SeleniumTest3 extends TestCase
{
    protected function setUp(): void
    {
        $options = new ChromeOptions();
        // Раскомментируйте строку для работы в режиме без заголовков
        // $options->addArguments(['--headless', '--no-sandbox']);
        $capabilities = DesiredCapabilities::chrome();
        $capabilities->setCapability(ChromeOptions::CAPABILITY,
                                     $options);
        $this->driver = RemoteWebDriver::create(
            "http://127.0.0.1:4444/wd/hub",
            $capabilities
        );
    }
    public function testAddVenue(): void
    {
        $this->driver->get(
            "http://popp.vagrant.internal/webwoo/AddVenue.php");
```

```
        $venel = $this->driver->findElement(
            WebDriverBy::name("venue_name"));
        $venel->sendKeys("my_test_venue");
        $venel->submit();
        $tdel = $this->driver->findElement(
            WebDriverBy::xpath("//td[1]"));
        $this->assertMatchesRegularExpression(
            "/'my_test_venue' added/",
            $tdel->getText());
        $spacel = $this->driver->findElement(
            WebDriverBy::name("space_name"));
        $spacel->sendKeys("my_test_space");
        $spacel->submit();
        $el = $this->driver->findElement(
            WebDriverBy::xpath("//td[1]"));
        $this->assertMatchesRegularExpression(
            "/'my_test_space' added/", $el->getText());
    }
}
```

Вот что произойдет, если выполнить этот тест из командной строки:

```
$ phpunitsrc/ch18/batch04/SeleniumTest3.php

PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
..                        1 / 1 (100%)
Time: 00:00.029, Memory: 6.00 MB
OK (1 test, 2 assertion)
```

Разумеется, это не все, что происходит на самом деле. Сервер Selenium откроет также окно в браузере и выполнит в нем указанные нами действия. Следует, однако, признаться, что полученный результат несколько пугает!

Проанализируем код данного теста. Сначала в нем вызывается метод `WebDriver::get()`, в котором запрашивается начальная страница проверяемого приложения. Обратите внимание на то, что в качестве параметра этого метода следует указывать полный URL тестируемого приложения. Это позволяет запускать его не только на том же самом физическом сервере, на котором установлен Selenium Server. В данном случае мы настроили веб-сервер Apache на обслуживание имитируемого сценария из файла `AddVenue.php` на виртуальной машине Vagrant. Сервер Selenium Server загрузит указанную страницу в открытое окно запущенного им браузера, как показано на рис. 18.1.

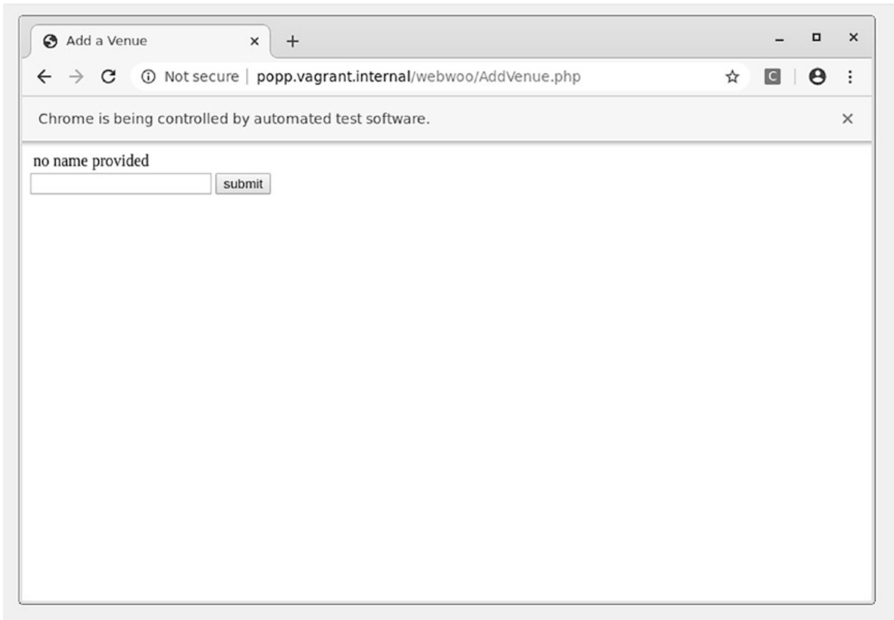


Рис. 18.1. *Страница `AddVenue`, загруженная сервером Selenium Server*

После загрузки страницы доступ к ней можно получить через интерфейс WebDriver API, а ссылку на элемент страницы — вызвав метод `RemoteWebDriver::findElement()` и передав ему объект типа `WebDriverBy`. В классе `WebDriverBy` предусмотрен набор фабричных методов, каждый из которых возвращает объект типа `WebDriverBy`, сконфигурированный согласно выбранному нами способу поиска конкретного элемента. В элементе разметки формы на данной веб-странице установлено значение `"venue_name"` атрибута `name`. Поэтому для поиска этого элемента разметки формы необходимо вызвать метод `WebDriverBy::name()`, а возвращаемый им объект передать в качестве параметра методу `findElement()`. В табл. 18.4 перечислены все фабричные методы, доступные в классе `WebDriverBy`.

Получив ссылку на объект типа `RemoteWebElement`, соответствующий элементу разметки формы под именем `"venue_name"`, мы можем воспользоваться его методом `sendKeys()` для установки значения. Следует, однако, иметь в виду, что метод `sendKeys()` не только присваивает значение элементу разметки формы, но и имитирует процесс ввода текста в поле элемента разметки формы. Этой полезной особенностью можно восполь-

зоваться для тестирования кода из сценария JavaScript, в котором перехватываются события от клавиатуры.

Таблица 18.4. *Фабричные методы класса* `WebDriverBy`

Метод	Описание
`className()`	Ищет элементы разметки по Css-имени класса
`cssSelector()`	Ищет элементы разметки по Css-селектору
`id()`	Ищет элементы разметки по их идентификатору
`name()`	Ищет элементы разметки по значению атрибута `name`
`linkText()`	Ищет элементы разметки по тексту, указанному в гиперссылке
`partialLinkText()`	Ищет элементы разметки по фрагменту текста, указанному в их гиперссылке
`tagName()`	Ищет элементы разметки по их дескриптору
`xpath()`	Ищет элементы, соответствующие выражению Xpath

После установки значения в элементе разметки формы она отправляется на сервер для обработки. Эту обязанность берет на себя интерфейс API. Когда метод `submit()` вызывается для элемента разметки формы, сервер Selenium Server находит нужную форму и отправляет ее на веб-сервер для обработки.

Разумеется, отправка формы на веб-сервер приведет к загрузке новой веб-страницы. Поэтому мы проверяем, все ли прошло так, как предполагалось. И в этом случае может быть вызван метод `WebDriver::findElement()`, хотя на этот раз в качестве параметра ему был передан объект типа `WebDriverBy`, настроенный на выражение Xpath. Если поиск завершится удачно, метод `findElement()` возвратит новый объект типа `RemoteWebElement`. Если же поиск не увенчается успехом, будет сгенерировано исключение и тестирование прервется. Если все пройдет нормально, мы получим значение элемента разметки с помощью метода `RemoteWebElement::getText()`.

На данной стадии мы отправили форму на веб-сервер для обработки и проанализировали данные на веб-странице, возвращенной сервером в ответ (рис. 18.2).

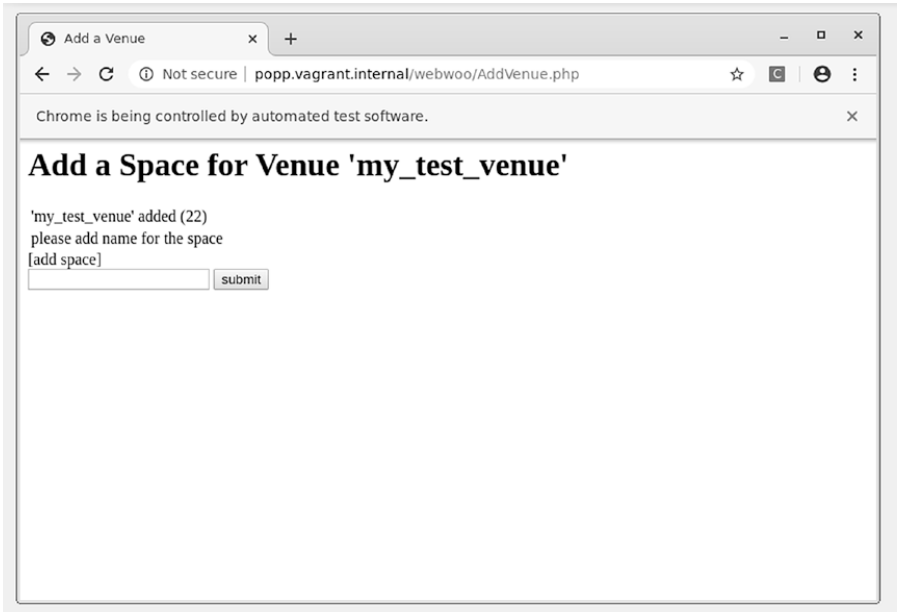


Рис. 18.2. *Страница* `AddSpace`

Теперь осталось лишь вновь заполнить форму, отправить ее на сервер и проанализировать новую страницу, следуя описанной выше методике.

Разумеется, в рамках одного раздела главы невозможно полностью описать систему Selenium, но я надеюсь, что даже такого поверхностного ее рассмотрения достаточно, чтобы понять принцип ее действия и возможности. За более подробными сведениями обращайтесь к руководству по Selenium, доступному по адресу `https://www.selenium.dev/documentation/en/`.

Предупреждения относительно тестирования

Преимуществами автоматических тестов легко увлечься. Я ввожу модульные тесты в свои проекты, а также применяю PHPUnit для функциональных тестов. Одним словом, я тестирую свой код как на уровне системы, так и на уровне класса. При таком подходе к тестированию я обнаружил реальные и заметные преимущества, но считаю, что ничего не дается даром.

Тесты повышают стоимость разработки. Так, если вы встраиваете в проект систему безопасности, то должны предусмотреть и дополнительное время, затрачиваемое на ее построение, что может повлиять на выпуск новых версий. К этому следует также отнести время, затрачиваемое на написание и выполнение тестов. В одной системе могут быть наборы функциональных тестов, выполняемых для нескольких баз данных и нескольких систем контроля версий. Если прибавить к этому еще несколько контекстных переменных, то мы столкнемся с реальной преградой для выполнения набора тестов. Вполне понятно, что тесты, которые не выполняются, не приносят никакой пользы. В качестве выхода из этого затруднительного положения можно, с одной стороны, полностью автоматизировать тесты, чтобы они автоматически запускались по расписанию с помощью такой программы, как `cron`. С другой стороны, можно поддерживать подмножество тестов, которые разработчики могут легко запускать, когда фиксируют свой код в хранилище. Такое подмножество тестов может сосуществовать с более длинным и медленным тестом.

Особое внимание следует обратить на непрочный характер многих средств тестирования. Тесты могут придать вам уверенность при внесении изменений, но если вместе со сложностью системы постепенно увеличится и ее охват тестами, то некоторые тесты могут легко завершиться неудачно. И зачастую это даже хорошо. Ведь вам нужно знать, когда не возникает ожидаемого поведения и когда имеет место поведение, которого вы не ожидаете.

Но нередко тестовая программа может завершиться неудачно из-за относительно простого изменения, например, в строке с адресом обратной связи. Каждый завершившийся неудачно тест требует принятия срочных мер. Но не так-то просто изменить 30 контрольных примеров в связи с незначительным изменением архитектуры или вывода. Модульные тесты менее подвержены подобного рода осложнениям, поскольку они, как правило, сосредоточены на каждом компоненте в отдельности.

Усилия по поддержанию тестов на одном уровне с развивающейся системой — это та цена, которую приходится платить за тестирование. Но я считаю, что преимущества тестирования оправдывают эти затраты.

Можно также принять некоторые меры для повышения прочности среды тестирования. Имеет смысл писать тесты, предусматривающие возможность некоторых изменений. Например, для тестирования вывода я обычно использую регулярные выражения, а не прямую проверку полной эквивалентности. И тогда при тестировании нескольких ключевых

слов маловероятно, что тест завершится неудачно, если я удалю знак новой строки из выводимой символьной строки. Безусловно, делать тесты слишком "снисходительными" тоже опасно, так что окончательное решение остается за вами.

Еще один вопрос — это степень, до которой следует использовать имитирующие объекты и заглушки, чтобы смоделировать систему за пределами компонента, который нужно протестировать. Некоторые считают, что необходимо изолировать компонент, насколько это возможно, и сымитировать все его окружение. В одних из моих проектов такой подход оказался действенным, но в других я обнаружил, что на поддержание имитации системы тратится очень много времени. Ведь приходится не только обеспечивать развитие тестов вместе с системой, но и обновлять имитирующие объекты. А теперь представьте, что изменился тип, возвращаемый из метода. Если вы не обновите метод соответствующего объекта заглушки, чтобы он возвращал новый тип, то клиентские тесты могут стать источником ошибки. Если же имитируется сложная система, то существует реальная опасность, что в имитирующие объекты вкрадутся ошибки. Отладка тестов — утомительное занятие, особенно если в самой системе ошибок нет.

Здесь я полагаюсь на свою интуицию. Обычно я пользуюсь имитациями и заглушками, но, если их применение становится слишком затратным, без колебаний перехожу к реальным компонентам. Сосредоточенность на предмете тестирования уменьшится, но в тоже время это даст преимущество выяснить, что ошибки, возникающие в контексте компонента, по меньшей мере выявляют реальные недостатки системы. Безусловно, реальные и поддельные элементы можно применять в определенном сочетании.

Как вы, вероятно, уже догадались, я совсем не идеолог тестирования. Я регулярно жульничаю, сочетая реальные и поддельные компоненты, а поскольку наполнение данными происходит неоднократно, я сосредоточиваю тестовые конфигурации в том, что Мартин Фаулер называет "материнскими объектами" (Object Mothers). Эти классы являются обычными фабриками, формирующими заполненные объекты для тестирования. Но некоторые разработчики предают такие совместно используемые средства тестирования настоящей анафеме.

Итак, рассмотрев ряд трудностей, с которыми вы можете столкнуться при тестировании, приведем еще раз преимущества тестирования, превосходящие все его недостатки. В частности, тестирование может оказать помощь в следующем:

- предотвратить ошибки (их можно обнаружить на стадии разработки и реорганизации кода);
- выявить ошибки (по мере увеличения количества тестов);
- побудить сосредоточиться на архитектуре системы;
- улучшить структуру кода, уменьшив опасения, что изменения вызовут больше затруднений, чем решат;
- приобрести бо́льшую уверенность при выпуске прикладного кода.

В каждом проекте, для которого я писал тесты, у меня были поводы раньше или позже возблагодарить себя за такую предусмотрительность.

Резюме

В этой главе были рассмотрены разновидности тестов, которые приходится писать всем разработчикам, но от которых они так же часто легкомысленно отказываются. Сначала мы представили инструментальное средство PHPUnit, позволяющее писать аналогичные разновидности тестов на стадии разработки, но в то же время сохранять их, постоянно извлекая из них выгоду! Затем мы реализовали контрольный пример и описали доступные методы с утверждениями. Далее мы рассмотрели группирование тестов в наборы, изучили накладываемые на них ограничения и исследовали поддельный характер имитирующих объектов и заглушек. Мы, кроме того, показали, каким образом реорганизация исходного кода системы для целей тестирования может положительно сказаться на ее архитектуре в целом, а также описали несколько методик тестирования веб-приложений, сначала с использованием только PHPUnit, а затем — и Selenium. И наконец мы рискнули предупредить о затратах, которые влекут за собой тесты, а также обсудили возможные компромиссы.

ГЛАВА 19

Автоматическое построение средствами Phing

Если контроль версий — это одна сторона медали, то автоматическое построение — другая. Контроль версий дает нескольким разработчикам возможность совместно работать над одним проектом. И когда каждому из них приходится разворачивать проект в своей среде разработки, особое значение приобретает автоматическое построение. Например, у одного разработчика каталог для веб-страниц находится в папке `/usr/local/apache/htdocs`, а у другого — в папке `/home/bibble/public_html`. Разработчики могут пользоваться разными паролями для доступа к базе данных, библиотечными каталогами или почтовыми программами. Гибкая кодовая база может легко приспосабливаться ко всем этим изменениям. Но усилия по изменению настроек и копированию каталогов вручную в файловой системе быстро превратятся в утомительное занятие, особенно если прикладной код приходится неоднократно устанавливать каждый день или даже час.

Как пояснялось ранее, диспетчер зависимостей Composer автоматизирует процесс установки пакетов. Доставлять готовый проект конечному пользователю, скорее всего, придется через пакет Composer или PEAR, поскольку это простой для пользователя механизм, а системы управления пакетами легко разрешают зависимости. Но еще остается немало операций, которые необходимо автоматизировать, прежде чем создавать пакет. В частности, необходимо сгенерировать код на основе шаблонов, выполнить тесты и предоставить механизмы для создания и обновления таблиц базы данных. И, наконец, может возникнуть потребность автоматизировать создание пакета, готового к эксплуатации. В этой главе представлено инструментальное средство Phing, позволяющее решать подобные задачи. В ней рассматриваются следующие вопросы.

- *Получение и установка Phing.* Кто строит построитель?
- *Свойства.* Установка и получение данных.
- *Типы.* Описание сложных частей проекта.

- *Целевые задания.* Разбиение сборки на вызываемые независимые наборы функциональных средств.
- *Задания.* Выполняемые задания.

Назначение Phing

Phing — это инструментальное средство PHP для построения проектов. Оно очень похоже на весьма распространенное (и очень эффективное) инструментальное средство Java под названием "Ant" ("муравей"). Такое название объясняется тем, что это небольшое инструментальное средство способно создавать очень крупные проекты. Как в Phing, так и в Ant используется XML-файл, обычно называемый `build.xml`, в котором определяется, что нужно сделать, чтобы установить проект или выполнить над ним любые другие операции.

Сообщество разработчиков приложений на PHP *действительно* нуждается в хорошем средстве для построения проектов. В прошлом у серьезных разработчиков для этой цели было несколько вариантов выбора. Они могли воспользоваться утилитой `make` — вездесущим средством построения для систем Unix, которое по-прежнему применяется в большинстве проектов на C, C++ и Perl. Но утилита `make` весьма требовательна к синтаксису и требует серьезных знаний командной оболочки, вплоть до написания сценариев, что может быть затруднительно для тех программистов на PHP, которые не умеют программировать на языке командной оболочки Unix или Linux. Более того, утилита `make` предоставляет очень мало встроенных инструментальных средств для таких распространенных операций построения, как преобразование имен файлов и содержимого. По существу, это обычное связующее звено для команд оболочки Unix. Утилита `make` служит лишь связующим звеном для выполняемых в оболочке команд, что затрудняет написание программ, которые должны устанавливаться на разных платформах. Кроме того, не во всех средах разработки имеется одна и та же версия утилиты `make`, если вообще имеется. Но даже если она и присутствует, то в вашей системе может и не быть всех команд, указанных в `makefile` (файл конфигурации, который управляет работой `make`).

Отличие инструментального средства Phing от утилиты `make` особо подчеркнуто в его наименовании. В частности, "Phing" означает "Phing Is Not GNU make" (Phing — это не GNU `make`). Это забавная рекурсивная

шутка программистов, например само название "GNU" означает "GNU is Not Unix" (GNU — это не Unix).

Phing — это "родное" приложение, написанное на PHP и интерпретирующее созданный пользователем XML-файл, чтобы выполнить операции над проектом. К таким операциям обычно относится копирование файлов из дистрибутивного каталога в различные каталоги назначения. Но инструментальное средство Phing способно на большее. Его можно использовать для формирования документации, выполнения тестов, вызова команд, выполнения произвольного кода PHP, создания пакетов, замены ключевых слов в исходном файле, удаления комментариев и формирования архивных выпусков пакетов в формате `tar/gzip`. И даже если инструментальное средство Phing пока еще не выполняет нужные вам операции, вы можете легко расширить его функциональные возможности.

Поскольку Phing — приложение, написанное на PHP, все, что вам потребуется для его запуска, — это последняя версия интерпретатора PHP. Так как Phing — это приложение для инсталляции PHP-приложений, наличие выполняемого PHP-файла — это, по сути, все, что требуется для его работы.

Получение и установка Phing

Если установить само инструментальное средство установки трудно, значит, здесь что-то не так! Впрочем, установить Phing совсем не трудно, если у вас установлена версия PHP 5 или более поздняя (а иначе эта книга вообще не для вас!).

Получить и установить Phing можно с помощью Composer, добавив в `composer.json` следующие строки:

```
{
    "require-dev": {
        "phing/phing": "2.*"
    }
}
```

На заметку На момент написания этих строк Phing версии 3 все еще находится в стадии альфа-тестирования. Все примеры в данной главе (и главе 21, "Непрерывная интеграция") работают с ним очень хорошо, но для установки требуется какой-то неортодоксальный хакинг. Надеюсь, когда вы будете это читать, все будет несколько проще. Ознакомьтесь с инструкциями по установке на домашней странице Phing: `www.phing.info/#install`.

Создание документа построения

Теперь все должно быть готово для работы с Phing! Проверим это, выполнив следующую команду:

```
$ phing vendor/bin/phing -v

Phing 2.16.3
```

Параметр `-v`, указанный в команде `phing`, предписывает сценарию возвратить сведения о номере версии. К тому времени, когда вы будете читать эти строки, номер версии может измениться. Но, выполнив данную команду в своей системе, вы должны получить аналогичное сообщение.

На заметку Если вы установили Phing с помощью Composer, файл исполняемого сценария будет установлен в вашем локальном каталоге `vendor/bin/`. Чтобы запустить Phing на выполнение, добавьте этот каталог в переменную окружения `$PATH` или укажите полный путь к исполняемому файлу. В приведенных далее примерах путь опущен.

Теперь выполним команду `phing` без аргументов:

```
$ phing

Buildfile: build.xml does not exist!
```

Как видите, Phing не может работать без инструкций. По умолчанию это инструментальное средство ищет в текущем каталоге файл `build.xml`. Создадим минимальный XML-документ, чтобы по крайней мере избавиться от приведенного выше сообщения об ошибке:

```
// Листинг 19.1
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz" default="main"
        description="my project" basedir="/tmp">
    <target name="main"/>
</project>
```

Это тот минимум, которым можно обойтись в файле построения. Если мы сохраним приведенный выше пример XML-документа в файле `build.xml` и снова выполним команду `phing`, то получим следующий результат:

```
$ phing

Buildfile: /var/popp/src/ch19/build.xml
Warning: target 'main' has no tasks or dependencies
megaquiz > main:

BUILD FINISHED

Total time: 0.0976 seconds
```

Подумать только: сколько усилий, чтобы практически ничего не добиться! Как видите, инструментальное средство Phing снова подсказало нам, что ничего особо полезного в нашем файле построения нет. Но ведь мы должны с чего-то начать! Рассмотрим снова файл построения. Включим в него объявление XML, поскольку это XML-документ. Как вам, должно быть, известно, комментарии в XML-документе выглядят следующим образом:

```
<!-- Это комментарий к XML-документу, ясно? -->
```

Итак, вторая строка в файле построения игнорируется, поскольку это комментарий. В файл построения можно ввести столько комментариев, сколько потребуется, и когда он станет достаточно крупным, этим обстоятельством можно воспользоваться в полной мере. Ведь сопровождать файлы построения крупных проектов нелегко без подробных комментариев.

Любой файл построения по-настоящему начинается с элемента разметки `project`, который может включать в себя до пяти атрибутов, среди которых обязательными являются атрибуты `name` и `default`. В частности, атрибут `name` определяет имя проекта, тогда как атрибут `default` — целевое задание, которое необходимо выполнить по умолчанию, если в командной строке ничего не указано. Необязательный атрибут `description` может предоставлять краткие сведения о проекте, а атрибут `basedir` позволяет указать контекстный каталог для построения. Если этот атрибут опущен, то предполагается текущий рабочий каталог. Последний атрибут, `phingVersion`, позволяет указать минимальную версию Phing, которая требуется для построения проектов. Эти атрибуты вкратце описаны в табл. 19.1.

Определив элемент разметки `project`, необходимо создать как минимум одно целевое задание. Именно на него мы и будем ссылаться в атрибуте `default`.

Таблица 19.1. Атрибуты элемента разметки `project`

Атрибут	Является ли обязательным	Описание	Значение по умолчанию
`name`	Да	Имя проекта	Отсутствует
`description`	Нет	Краткое описание проекта	Отсутствует
`default`	Да	Целевое задание для выполнения по умолчанию	Отсутствует
`phingVersion`	Нет	Минимальная версия Phing, необходимая для построения проекта	Отсутствует
`basedir`	Нет	Контекстный каталог файловой системы, в котором должно выполняться построение проекта	Текущий каталог (`.`)
`strict`	Нет	Строгий режим работы: предупреждения рассматриваются как ошибки	`false`

Целевые задания

В каком-то смысле целевые задания подобны функциям. Целевое задание — это набор действий, сгруппированных для достижения поставленной цели, например для копирования каталога из одного места в другое или формирования документации.

В предыдущем примере мы включили в документ построения минимальную реализацию целевого задания:

```
// Листинг 19.2
< target name="main" />
```

Как видите, в целевом задании должен быть определен как минимум атрибут `name`, который и был использован в элементе разметки `project`. А поскольку элемент разметки по умолчанию указывает на целевое задание `main`, именно оно и будет вызываться всякий раз, когда Phing запускается на выполнение по команде `phing` без аргументов командной строки. Это подтверждает полученный вывод:

```
megaquiz > main:
```

Целевые задания можно организовать таким образом, чтобы они зависели друг от друга. Установление зависимости между целевыми заданиями означает для Phing, что целевое задание не будет выполнено, прежде чем не завершатся все целевые задания, от которых оно зависит. Добавим подобную зависимость в файл построения:

```
// Листинг 19.3
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main">
    <target name="runfirst" />
    <target name="runsecond" depends="runfirst"/>
    <target name="main" depends="runsecond"/>
</project>
```

Как видите, мы добавили новый атрибут depends в элемент разметки target. В атрибуте depends предписывается, что указанное целевое задание должно выполняться в Phing прежде текущего целевого задания. Поэтому можно сделать так, чтобы целевое задание, которое копирует определенные файлы в каталог, вызывалось до целевого задания, выполняющего преобразование всех файлов в этом каталоге. В данном примере были добавлены два новых целевых задания: runsecond, от которого зависит основное задание main, а также runfirst, от которого зависит задание runsecond. А теперь выясним, что произойдет, если выполнить Phing с данным файлом построения:

```
$ phing

Buildfile: /var/popp/src/ch19/build.xml
Warning: target 'runfirst' has no tasks or dependencies

megaquiz > runfirst:
megaquiz > runsecond:
megaquiz > main:

BUILD FINISHED
Total time: 0.1250 seconds
```

Итак, зависимости соблюдаются. Phing обнаруживает целевое задание main, выясняет его зависимость и переходит к выполнению целевого задания runsecond. У целевого задания runsecond имеется своя зависимость, поэтому Phing вызывает задание runfirst.

Удовлетворив его зависимость, Phing снова вызывает задание runsecond. И наконец вызывается основное задание main. В атрибуте depends можно указать сразу несколько целевых заданий через запятую. Все эти зависимости будут выполняться по очереди.

Теперь, когда у нас есть несколько заданий, явно укажем задание из командной строки и тем самым изменим действие атрибута default:

```
$ phing runsecond

Buildfile: /var/popp/src/ch19/build.xml
Warning: target 'runfirst' has no tasks or dependencies

megaquiz > runfirst:
megaquiz > runsecond:

BUILD FINISHED
Total time: 0.1043 seconds
```

Атрибут default игнорируется, поскольку в данном случае в командной строке указано имя целевого задания. Вместо этого вызывается целевое задание, соответствующее указанному аргументу, а также целевое задание, от которого оно зависит. Это полезно для вызова таких специализированных заданий, как очистка каталога построения или выполнение сценариев после установки.

У элемента разметки target имеется также необязательный атрибут description, в котором можно описать назначение целевого задания, как показано ниже:

```
// Листинг 19.4
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main"
         description="Программа quiz">
    <target name="runfirst"
            description="Первая цель" />
    <target name="runsecond"
            depends="runfirst, housekeeping"
            description="Вторая цель" />
    <target name="main"
            depends="runsecond"
            description="Основная цель" />
</project>
```

Дополнение целевых заданий описаниями ничего не меняет в нормальном процессе построения. Но если пользователь запустит Phing на выполнение с параметром `-projecthelp`, как показано ниже, то такие описания послужат для получения кратких сведений о проекте:

```
$ phing -projecthelp

Buildfile: /var/popp/src/ch19/build.xml
Warning: target 'runfirst' has no tasks or dependencies
Программа quiz
Default target:
-------------------------------------------------------------------
Main         Основная цель
Main targets:
-------------------------------------------------------------------
Main         Основная цель
Runfirst     Первая цель
Runsecond    Вторая цель
```

Обратите внимание на то, что мы добавили атрибут `description` и в элемент разметки `project`. Если же требуется скрыть какое-нибудь целевое задание в приведенном выше листинге, добавьте атрибут `hidden` в соответствующий элемент разметки. Такая возможность может понадобиться для тех целевых заданий, в которых выполняются служебные операции, но они не должны вызываться непосредственно из командной строки:

```
// Листинг 19.5
<target name="housekeeping" hidden="true" />
   <!-- Вспомогательные операции, не вызываемые непосредственно -->
</target>
```

Свойства

Phing позволяет устанавливать значения свойств в элементе разметки `property`. Свойства аналогичны глобальным переменным в сценарии. Как правило, они объявляются в начале файла, чтобы разработчикам легче было выяснять, что к чему в файле построения. Ниже приведен файл построения, в котором указаны сведения о базе данных:

```
// Листинг 19.6
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
```

```
        default="main">

    <property name="dbname" value="megaquiz" />
    <property name="dbpass" value="default" />
    <property name="dbhost" value="localhost" />

    <target name="main">
        <echo>database: ${dbname}</echo>
        <echo>pass:     ${dbpass}</echo>
        <echo>host:     ${dbhost}</echo>
    </target>
</project>
```

В данном файле построения мы добавили новый элемент разметки `property` с атрибутами `name` и `value`. В определение целевого задания `main` добавлено три таких экземпляра `property`.

Появился также новый элемент разметки `echo`, который представляет собой пример определения задания. Более подробно задания рассматриваются в следующем разделе, а до тех пор достаточно знать, что элемент разметки `echo` делает именно то, чего и следовало от него ожидать, — выводит свое содержимое. Обратите также внимание на синтаксис, который применяется для ссылки на значение свойства. В частности, указав знак доллара и заключив имя свойства в фигурные скобки, можно предписать Phing заменить заданную символьную строку значением свойства:

```
${имя_свойства}
```

В рассматриваемом здесь файле построения подобным образом объявляются три свойства, значения которых направляются в стандартный поток вывода. Рассмотрим этот файл построения в действии:

```
$ phing

Buildfile: /var/popp/src/ch19/build.xml
megaquiz > main:

    [echo] database: megaquiz
    [echo] pass:     default
    [echo] host:     localhost

BUILD FINISHED
Total time: 0.0989 seconds
```

Рассмотрев свойства, можно завершить обсуждение целевых заданий. В элементе разметки `target` можно указать еще два дополнительных атри-

бута: if и unless, причем со своими свойствами. Так, если указать атрибут if с именем свойства, целевое задание будет выполнено только в том случае, если заданное свойство определено. Если же свойство не определено, то выполнение целевого задания завершится без вывода результатов. В приведенном ниже примере свойство dbpass было закомментировано, но при этом его значение мы затребовали в целевом задании main с помощью атрибута if:

```
// Листинг 19.7

<project name="megaquiz"
         default="main">
    <property name="dbname" value="megaquiz" />
    <!--<property name="dbpass" value="default" />-->
    <property name="dbhost" value="localhost" />

    <target name="main" if="dbpass">
        <echo>database: ${dbname}</echo>
        <echo>pass:     ${dbpass}</echo>
        <echo>host:     ${dbhost}</echo>
    </target>
</project>
```

Выполним команду phing снова:

```
$ phing

Buildfile: /var/popp/src/ch19/build.xml
megaquiz > main:

BUILD FINISHED
Total time: 0.0957 seconds
```

Как видите, это не привело к ошибке, тем не менее целевое задание main так и не было выполнено. Зачем вообще нужна такая возможность? Существует еще один способ определения свойств в проекте. В частности, их можно определить в командной строке. О передаче свойства инструментальному средству Phing сообщает параметр -D, вслед за которым указываются имя и значение свойства. Следовательно, аргумент командной строки должен быть указан в следующей форме:

```
-Dимя=значение
```

В данном примере нам нужно, чтобы значение свойства dbpass можно было определить из командной строки. Попробуем это сделать:

```
$ phing -Ddbpass=userset

Buildfile: /var/popp/src/ch19/build.xml
megaquiz > main:

    [echo] database: megaquiz
    [echo] pass:     userset
    [echo] host:     localhost

BUILD FINISHED
Total time: 0.0978 seconds
```

Атрибут `if` целевого задания `main` удовлетворяется тем, что свойство `dbpass` определено и выполнение целевого задания разрешено.

Как и следует ожидать, атрибут `unless` является полной противоположностью `if`. Если свойство определено и на него имеется ссылка в атрибуте `unless` целевого задания, то оно не будет выполнено. Это удобно в том случае, если необходима возможность подавить выполнение конкретного целевого задания из командной строки. Поэтому целевое задание `main` может быть дополнено следующим образом:

```
// Листинг 19.8

<target name="main" unless="suppressmain">
    <echo>database: ${dbname}</echo>
    <echo>pass:     ${dbpass}</echo>
    <echo>host:     ${dbhost}</echo>
</target>
```

Это задание будет выполнено только в том случае, если свойство `suppressmain` не определено:

```
$ phing -Dsuppressmain=yes
```

Завершив рассмотрение элемента разметки `target`, подытожим в табл. 19.2 все, что касается его атрибутов.

Если значение свойства указано в командной строке, оно переопределяет все объявления свойств в файле построения. Существует еще одно условие, при котором значение свойства может быть заменено. Если свойство объявляется дважды, то по умолчанию приоритет будет иметь первоначальное значение. Но это положение можно изменить, определив атрибут `override` во втором элементе `property`. Приведем конкретный пример:

```
// Листинг 19.9

<?xml version="1.0"?>
<!-- build xml -->
<project name="megaquiz"
        default="main">

    <property name="dbpass" value="default" />

    <target name="main">
        <property name="dbpass" override="yes" value="specific" />
        <echo>pass: ${dbpass}</echo>
    </target>
</project>
```

Таблица 19.2. *Атрибуты элемента разметки* `target`

Атрибут	Является ли обязательным	Описание
`name`	Да	Имя целевого задания
`depends`	Нет	Целевые задания, от которых зависит текущее целевое задание
`if`	Нет	Выполнить целевое задание только в том случае, если указанное свойство определено
`unless`	Нет	Выполнить целевое задание только в том случае, если указанное свойство не определено
`logskipped`	Нет	Если целевое задание пропущено, например, из-за наличия атрибутов `if`/`unless`, добавить уведомление в выводимый результат
`hidden`	Нет	Скрыть целевое задание в выводимых листингах и сводках
`description`	Нет	Краткое описание назначения задания

В данном случае мы определяем свойство `dbpass`, присваивая ему начальное значение `"default"`. В целевом задании `main` мы снова определяем это же свойство, добавляя атрибут `override`, для которого установлено значение `"yes"`, и указываем для него новое значение. Это новое значение отражено в выводимом результате, как показано ниже:

```
$ phing

Buildfile: /var/popp/src/ch19/build.xml

megaquiz > main:
    [echo] pass: specific

BUILDFINISHED
Total time: 0.0978 seconds
```

Если бы мы не определили атрибут `override` во втором элементе разметки `property`, то первоначальное значение `"default"` свойства `dbpass` не изменилось бы. Важно отметить, что целевые задания — это не функции, поскольку для них отсутствует понятие локального контекста. Поэтому если переопределить свойство для некоторого целевого задания, оно останется переопределенным для всех других заданий в файле построения. Но это ограничение можно, конечно, обойти, сохранив значение свойства во временном свойстве до переопределения, а затем восстановив его по завершении локальной обработки.

До сих пор речь шла о свойствах, которые мы определили сами. Но в Phing предусмотрены также встроенные свойства. Обращение к ним происходит таким же образом, как и к самостоятельно объявляемым свойствам. Ниже приведен соответствующий пример:

```
// Листинг 19.10

<?xml version="1.0"?>
<!-- build xml -->
<project name="megaquiz"
         default="main">
    <target name="main">
        <echo>name: ${phing.project.name}</echo>
        <echo>base: ${project.basedir}</echo>
        <echo>home: ${user.home}</echo>
        <echo>pass: ${env.DBPASS}</echo>
    </target>
</project>
```

В данном примере применяются лишь некоторые встроенные свойства Phing. Так, вместо свойства `phing.project.name` подставляется имя проекта, как определено в атрибуте `name` элемента разметки `project`; свойство `project.basedir` задает базовый каталог; а свойство `user.home` — начальный каталог пользователя (эти свойства служат для определения стандартных мест установки проекта).

И, наконец, префикс env в ссылке на свойство указывает на переменную окружения операционной системы. Так, синтаксическая конструкция ${env.DBPASS} указывает на необходимость поиска переменной окружения DBPASS. Запустим Phing на выполнение приведенного выше файла построения, получив представленный ниже результат:

```
$ phing

Buildfile: /var/popp/src/ch19/build.xml

megaquiz > main:
    [echo] name: megaquiz
    [echo] base: /var/popp/src/ch19
    [echo] home: /home/vagrant
    [echo] pass: ${env.DBPASS}

BUILD FINISHED
Total time: 0.1056 seconds
```

Обратите внимание на то, что последнее свойство не было преобразовано в соответствующее значение. Это поведение по умолчанию, когда символьная строка, ссылающаяся на свойство, остается без изменений, если свойство не найдено. Если же определить переменную окружения DBPASS и снова выполнить команду phing, эта переменная окажется в выводимом результате, как показано ниже:

```
$ export DBPASS=wooshpoppow
$ phing

Buildfile: /var/popp/src/ch19/build.xml

megaquiz > main:
    [echo] name: megaquiz
    [echo] base: /var/popp/src/ch19
    [echo] home: /home/vagrant
    [echo] pass: wooshpoppow

BUILD FINISHED
Total time: 0.1044 seconds
```

Итак, мы рассмотрели три способа определения свойств: с помощью элемента разметки property, аргумента командной строки и переменной окружения. Существует и четвертый способ, дополняющий три предыдущих. Значения свойств можно определить в отдельном файле. По мере увеличения сложности проекта я предпочитаю пользоваться имен-

но таким способом. Рассмотрим еще раз содержимое основного файла построения:

```
// Листинг 19.11
<?xml version="1.0"?>
<!-- build xml -->
<project name="megaquiz"
         default="main">
    <target name="main">
        <echo>database: ${dbname}</echo>
        <echo>pass:     ${dbpass}</echo>
        <echo>host:     ${dbhost}</echo>
    </target>
</project>
```

Как видите, в этом файле построения значения свойств просто выводятся на экран без предварительного объявления или проверки наличия присвоенных им значений. Ниже показано, что произойдет, если выполнить команду `phing` без параметров:

```
$ phing

...
   [echo] database: ${dbname}
   [echo] pass:     ${dbpass}
   [echo] host:     ${dbhost}
...
```

А теперь определим значения этих свойств в отдельном файле `megaquiz.properties` следующим образом:

```
dbname=filedb
dbpass=filepass
dbhost=filehost
```

Этот файл можно далее подключить к процессу построения, указав его с помощью параметра `-propertyfile` в команде `phing`, как показано ниже:

```
$ phing -propertyfile megaquiz.properties

...
   [echo] database: filedb
   [echo] pass:     filepass
   [echo] host:     filehost
...
```

Такой способ намного удобнее указания длинного списка значений свойств в командной строке. Но в этом случае нужно проявить осмотрительность, чтобы не поместить файл свойств в глобальное хранилище системы контроля версий!

Целевые задания можно использовать для того, чтобы гарантировать, что нужные свойства определены. Допустим, в проекте требуется свойство dbpass, которое пользователь мог бы устанавливать в командной строке (у такого способа всегда имеется приоритет над другими методами присваивания значений свойствам). Но если пользователь не укажет это свойство в командной строке, то придется обращаться к переменной окружения. Если и она не будет найдена, то будет использовано значение, принятое по умолчанию:

```
// Листинг 19.12
<?xml version="1.0"?>
<!-- build xml -->
<project name="megaquiz"
         default="main" basedir=".">
  <target name="setenvpass" if="env.DBPASS" unless="dbpass">
    <property name="dbpass" override="yes" value="${env.DBPASS}" />
  </target>
  <target name="setpass" unless="dbpass" depends="setenvpass">
    <property name="dbpass" override="yes" value="default" />
  </target>
  <target name="main" depends="setpass">
    <echo>pass: ${dbpass}</echo>
  </target>
</project>
```

Как обычно, первым вызывается целевое задание по умолчанию main. У него имеется ряд зависимостей, и поэтому Phing возвращается к целевому заданию setpass. Но целевое задание setpass зависит, в свою очередь, от задания setenvpass, и поэтому процесс построения начинается именно с него. Целевое задание setenvpass сконфигурировано таким образом, чтобы оно выполнялось лишь в том случае, если свойство dbpass не было определено и если присутствует переменная окружения DBPASS (доступ к ней осуществляется через ссылку env.DBPASS). Если оба эти условия удовлетворяются, то свойство dbpass определяется с помощью элемента разметки property, а его значение берется из аргумента командной строки или из переменной окружения. Если же и то, и другое отсутствует, то данное свойство остается неопределенным. Далее выполняется целевое

задание setpass, но только если свойство dbpass все еще не определено. В таком случае этому свойству присваивается символьная строка по умолчанию "default".

Установка значений свойств по условию в задании condition

В предыдущем примере была продемонстрирована довольно сложная логика присваивания значений свойствам. Но зачастую требуется просто установить значение, принятое по умолчанию. Для этой цели предусмотрено задание condition, которое позволяет устанавливать значения свойств в соответствии с настраиваемыми условиями. Ниже приведен соответствующий пример:

```
// Листинг 19.13
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main">
    <condition property="dbpass" value="default">
       <not>
           <isset property="dbpass" />
       </not>
    </condition> `
    <target name="main">
        <echo>pass: ${dbpass}</echo>
    </target>
</project>
```

В задании condition требуется задать атрибут property. Кроме того, можно указать необязательный атрибут value, значение которого определяет значение свойства в том случае, если проверка вложенного условия окажется истинной. Если атрибут value не задан, то свойству будет присвоено логическое значение true.

Условие проверки задается с помощью одного из дескрипторов, часть из которых, как, например, используемый в данном примере дескриптор not, допускает вложенные дескрипторы. В данном примере используется элемент разметки isset, возвращающий логическое значение true, если указанное в нем свойство определено. А поскольку значение требуется присвоить свойству dbpass только в том случае, если последнее еще *не* определено, то значение элемента разметки isset придется инверти-

ровать, поместив его внутрь дескриптора not. Этот дескриптор инвертирует значение содержащегося в нем дескриптора. Таким образом, задание condition в данном примере можно определить, используя синтаксис PHP, следующим образом:

```
if (!isset($dbname)) {
    $dbname = "default";
}
```

Типы

Можно решить, что, рассмотрев свойства, мы покончили с данными. Но на самом деле в Phing поддерживается ряд специальных элементов, которые называются *типами* и инкапсулируют разные виды информации, полезной для процесса построения.

Тип fileset

Допустим, что в файле построения требуется определить каталог. Для этой цели можно, конечно, воспользоваться свойством, но если разработчики работают на разных платформах, на которых знаки разделения каталогов различаются, то организация процесса построения сразу же вызовет трудности. В качестве выхода из положения можно воспользоваться типом данных fileset, который не зависит от конкретной платформы. Так, если определить каталог, указав знаки косой черты в пути к нему, то они автоматически будут преобразованы в знаки обратной косой черты, когда процесс построения будет запущен в системе Windows. Минимальный элемент разметки fileset можно определить следующим образом:

```
<fileset dir="src/lib" />
```

Как видите, мы используем атрибут dir, чтобы определить каталог, который требуется представить. Можно также добавить необязательный атрибут id, чтобы в дальнейшем ссылаться на элемент разметки fileset, как показано ниже:

```
<fileset dir="src/lib" id="srclib">
```

Тип данных fileset особенно полезен при определении типов документов, которые нужно включать или исключать. Например, при установке набора файлов в него нежелательно включать те файлы, которые

совпадают с определенным шаблоном. Такие условия можно определить в атрибуте excludes следующим образом:

```
<fileset dir="src/lib" id="srclib"
    excludes="**/*_test.php **/*Test.php" />
```

Обратите внимание на синтаксис, который мы использовали в атрибуте excludes, где две звездочки обозначают любой каталог или подкаталог, входящий в каталог src/lib, а одна звездочка — нуль или больше символов. В данном случае мы указываем, что требуется исключить файлы, оканчивающиеся на _test.php или Test.php, во всех подкаталогах, определенных в атрибуте dir. В атрибуте excludes можно также указать несколько шаблонов, разделив их пробелами.

Такой же синтаксис можно употребить и в атрибуте includes. Возможно, каталоги src/lib содержат много файлов, не относящихся непосредственно к PHP. И хотя эти файлы полезны для разработчиков, они не должны быть установлены. Такие файлы можно, конечно, исключить, но, вероятно, проще определить виды файлов, которые требуется включить в установку. В таком случае файл не должен быть установлен, если он не оканчивается на .php, как показано ниже:

```
<fileset dir="src/lib" id="srclib"
    excludes="**/*_test.php **/*Test.php"
    includes="**/*.php" />
```

По мере создания правил include и exclude элемент разметки fileset, вероятно, станет слишком длинным. Правда, отдельные правила exclude можно извлечь из него, разместив каждое из них в отдельном элементе разметки exclude. То же самое можно сделать и для правил include. Таким образом, элемент разметки fileset можно переписать следующим образом:

```
<fileset dir="src/lib" id="srclib">
    <exclude name="**/*_test.php" />
    <exclude name="**/*Test.php" />
    <include name="**/*.php" />
</fileset>
```

Некоторые атрибуты элемента разметки fileset перечислены в табл. 19.3.

Таблица 19.3. Атрибуты элемента разметки `fileset`

Атрибут	Является ли обязательным	Описание
`refid`	Нет	Текущий элемент разметки `fileset` содержит ссылку на элемент разметки `fileset` с заданным идентификатором
`dir`	Нет	Начальный каталог
`expandsymboliclinks`	Нет	Если в этом атрибуте задано логическое значение `true`, то обращение следует символическим ссылкам
`includes`	Нет	Список разделенных запятыми шаблонов. Совпадающие с ними файлы включаются в процесс построения
`excludes`	Нет	Список разделенных запятыми шаблонов. Совпадающие с ними файлы исключаются из процесса построения

Тип `patternset`

При создании шаблонов в элементе разметки `fileset` (и в других элементах) существует вероятность повторения групп элементов `exclude` и `include`. В предыдущем примере мы определили шаблоны для тестовых и обычных кодовых файлов. Со временем мы можем что-нибудь к ним добавить (например, включить расширения `.conf` и `.inc` в определение кодовых файлов). Если мы определили другие элементы разметки `fileset`, в которых используются эти же шаблоны, то нам придется вносить любые необходимые изменения во все соответствующие элементы разметки `fileset`.

Чтобы разрешить подобное затруднение, сгруппируем шаблоны в элементах разметки `patternset`. Элемент разметки `patternset` группирует элементы разметки `exclude` и `include` таким образом, чтобы на них можно было в дальнейшем ссылаться из других типов данных. В приведенном ниже примере показано, как извлечь элементы разметки `exclude` и `include` из элемента разметки `fileset`, показанного в предыдущем примере, и добавить их в элемент разметки `patternset`:

```
// Листинг 19.14
<patternset id="inc_code">
    <include name="**/*.php" />
    <include name="**/*.inc" />
    <include name="**/*.conf" />
</patternset>

<patternset id="exc_test">
    <exclude name="**/*_test.php" />
    <exclude name="**/*Test.php" />
</patternset>
```

В данном примере мы создаем два элемента разметки `patternset`, задавая в их атрибутах `id` значения `inc_code` и `exc_test` соответственно. Элемент разметки `inc_code` содержит элементы разметки `include` для включения кодовых файлов, а элемент разметки `exc_test` — элементы разметки `exclude` для исключения тестовых файлов. Теперь мы можем сослаться на эти элементы разметки `patternset` в самом элементе разметки `fileset` следующим образом:

```
// Листинг 19.15
<fileset dir="src/lib" id="srclib">
    <patternset refid="inc_code" />
    <patternset refid="exc_test" />
</fileset>
```

Чтобы сослаться на существующий элемент разметки `patternset`, необходимо воспользоваться другим элементом разметки, `patternset`, в котором должен быть определен единственный атрибут — `refid`. Он должен содержать ссылку на атрибут `id` того элемента разметки `patternset`, который требуется использовать в текущем контексте. Таким образом, элементы разметки `patternset` можно повторно использовать в разных элементах разметки `fileset`, как показано ниже:

```
<fileset dir="src/views" id="srcviews">
    <patternset refid="inc_code" />
</fileset>
```

Любые изменения, вносимые в элементы разметки `inc_code` `patternset`, будут автоматически отражаться на всех типах, в которых они используются. Как и в типе `fileset`, правила `exclude` можно задать в атрибуте `excludes` или в ряде подэлементов разметки `exclude`. То же самое относится и к правилам `include`. Некоторые атрибуты элемента разметки `patternset` перечислены в табл. 19.4.

Таблица 19.4. Атрибуты элемента разметки `patternset`

Атрибут	Является ли обязательным	Описание
`id`	Нет	Уникальный идентификатор для ссылки на элемент
`excludes`	Нет	Список шаблонов для исключения
`includes`	Нет	Список шаблонов для включения
`refid`	Нет	Текущий элемент разметки `patternset` является ссылкой на `patternset` с указанным идентификатором

Тип `filterchain`

Типы, с которыми мы имели дело до сих пор, обеспечивали механизмы выбора наборов файлов. А тип `filterchain` обеспечивает гибкий механизм преобразования содержимого текстовых файлов.

Как и для остальных типов, определение элемента разметки `filterchain` само по себе не вызывает никаких изменений. Этот элемент разметки и его дочерние элементы должны сначала быть связаны с конкретным заданием, т.е. элемент разметки `filterchain` сообщает Phing, какие именно действия следует выполнить. Мы еще вернемся к заданиям далее в этой главе.

Элемент разметки `filterchain` группирует любое количество фильтров. Фильтры применяются к файлам по принципу конвейера: первый фильтр изменяет свой файл и передает результат второму фильтру, который вносит свои изменения, и т.д. Объединив несколько фильтров в элемент разметки `filterchain`, можно гибко выполнять преобразования.

В качестве примера создадим фильтр, удаляющий комментарии из любого переданного ему исходного текста на PHP:

```
// Листинг 19.16
<filterchain>
    <stripphpcomments />
</filterchain>
```

Задание `stripphpcomments` делает именно то, что подразумевает его наименование, удаляя комментарии из исходного текста на PHP. Если вы включили подробную документацию в исходный код, то вы облегчили жизнь другим разработчикам и в то же время добавили много лишнего в свой проект. А поскольку все значительные операции выполняются

в исходных каталогах, то ничто не мешает удалить комментарии при установке.

На заметку Если для развертывания своих проектов вы пользуетесь инструментальным средством построения, то убедитесь, что никто не вносил изменения в установленный код. Установщик скопирует файлы поверх любых измененных файлов, и поэтому все изменения будут потеряны. Подобные случаи мне приходилось не раз наблюдать в своей практике.

Забегая немного вперед, к следующему разделу, добавим элемент разметки `filterchain` в задание:

```
// Листинг 19.17
<target name="main">
    <copy todir="build/lib">
        <fileset refid="srclib"/>
        <filterchain>
            <stripphpcomments />
        </filterchain>
    </copy>
</target>
```

Заданием `copy` вам, вероятно, придется пользоваться чаще всего. Оно копирует файлы из одного места в другое. Как видите, мы определяем целевой каталог в атрибуте `todir`, а источник файлов определяется в элементе разметки `fileset`, созданном нами в предыдущем разделе. Затем следует элемент разметки `filterchain`, в котором указано преобразование, применяемое к любому файлу, скопированному заданием `Copy`.

В Phing поддерживаются фильтры для многих операций, включая удаление разрывов строк (`striplinebreaks`) и замену знаков табуляции пробелами (`tabtospaces`). Существует даже фильтр `xsltfilter` для применения XSLT-преобразований к исходным файлам! Но самым широко применяемым, вероятно, является фильтр `replacetokens`. Он позволяет заменить маркеры в исходном коде значениями свойств, определенных в файле построения, извлеченных из переменных окружения или переданных в командной строке. Это очень удобно для настройки процесса установки. Все маркеры имеет смысл разместить в центральном файле конфигурации, чтобы было легче следить за меняющимися аспектами проекта.

У фильтра `replacetokens` имеются два необязательных атрибута, `begintoken` и `endtoken`, с помощью которых можно задавать знаки, определяющие границы маркеров. Если опустить эти атрибуты, то в Phing

будет использован стандартный знак @. Чтобы распознавать и заменять маркеры, необходимо добавить элементы разметки token в элемент разметки replacetokens. Итак, добавим элемент replacetokens в разметку из рассматриваемого здесь примера:

```
// Листинг 19.18
<copy todir="build/lib">
    <fileset refid="srclib"/>
    <filterchain>
        <stripphpcomments />
        <replacetokens>
            <token key="dbname" value="${dbname}" />
            <token key="dbhost" value="${dbhost}" />
            <token key="dbpass" value="${dbpass}" />
        </replacetokens>
    </filterchain>
</copy>
```

Как видите, в элементах разметки token следует указать атрибуты key и value. А теперь рассмотрим результат выполнения этого задания по преобразованию файлов в рассматриваемом здесь проекте. Первоначальный файл находится в исходном каталоге src/lib/Config.php:

```
// Листинг 19.19
/**
 * Черновой вариант класса Conf
 **/
class Config {
    public $dbname ="@dbname@";
    public $dbpass ="@dbpass@";
    public $dbhost ="@dbhost@";
}
```

Выполнение основного целевого задания, содержащего определенное ранее задание copy, с помощью команды

```
$ phing
```

дает результат

```
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:
    [copy] Copying 8 files to /home/bob/working/megaquiz/build/lib

BUILD FINISHED
Total time: 0.1413 seconds
```

Первоначальный файл, конечно, остался нетронутым, но благодаря заданию copy он был скопирован в каталог build/lib/Config.php. Теперь в нем не только удалены комментарии, но и маркеры заменены равнозначными им значениями свойств:

```
class Config {
    public $dbname ="megaquiz";
    public $dbpass ="default";
    public $dbhost ="localhost";
}
```

Задания

Задания — это элементы файла построения, которые выполняют необходимые действия. Без заданий многого не добьешься, поэтому, опережая ход событий, мы уже использовали некоторые из них в предыдущих примерах. Рассмотрим их подробнее.

Задание echo

Задание echo идеально подходит для традиционного примера приветствия "Hello World". А на практике оно позволяет сообщить пользователю, что уже сделано или что еще предстоит сделать. Можно также провести санитарную проверку процесса построения, отобразив значения свойств. Как было показано ранее, любой текст, размещаемый между открывающим и закрывающим дескрипторами элемента разметки echo, будет выведен в окне браузера:

```
<echo>Пароль '${dbpass}', тссс!</echo>
```

С другой стороны, можно указать выводимое сообщение в атрибуте msg:

```
<echo msg="Пароль '${dbpass}', тссс!" />
```

Результат окажется тем же самым: указанное сообщение будет направлено в стандартный вывод:

```
[echo] Пароль 'default', тссс!
```

Задание copy

Копирование на самом деле составляет сущность процесса установки. Обычно создается одно целевое задание для копирования файлов из исходных каталогов и последующего их объединения во временном каталоге

построения. После этого следует другое целевое задание для копирования объединенных (и преобразованных) файлов по месту назначения. Разбиение процесса установки на отдельные стадии построения и установки не является абсолютно необходимым, но означает, что результаты первоначального построения можно проверить, прежде чем перезаписывать рабочий код. Можно также изменить значение свойства и установить файлы в другом месте, не выполняя снова потенциально затратную стадию копирования/замены.

Самое простое, что позволяет сделать задание `copy`, — указать исходный файл и целевой каталог или файл, как показано ниже:

```
<copy file="src/lib/Config.php" todir="build/conf" />
```

Как видите, исходный файл указан в атрибуте `file`, тогда как атрибут `todir` служит, как вам, должно быть, уже известно, для указания целевого каталога. Если же целевого каталога не существует, то Phing создаст его автоматически.

Если требуется указать целевой файл, а не содержащий его каталог, то вместо атрибута `todir` можно воспользоваться атрибутом `tofile`, как показано ниже:

```
<copy file="src/lib/Config.php" tofile="build/conf/myConfig.php" />
```

И в этом случае каталог `build/conf` создается автоматически, если его не существует. Но на этот раз файл `Config.php` переименовывается в `myConfig.php`. Чтобы скопировать сразу несколько файлов, в задание `copy` достаточно добавить упоминавшийся ранее элемент разметки `fileset`. В приведенном ниже примере исходные файлы определяются в элементе `srclib fileset`, поэтому в задании `copy` остается лишь определить значение атрибута `todir`:

```
// Листинг 19.20
<copy todir="build/lib">
    <fileset refid="srclib"/>
</copy>
```

Phing обладает достаточно развитой логикой, чтобы проверить, изменился ли исходный файл со времени создания целевого файла. Если никаких изменений в файле не произошло, то Phing не будет его копировать. Это означает, что проект можно построить многократно, но установлены будут только измененные файлы. И этого достаточно до тех пор, пока не предвидится других изменений. Если же файл преобразуется в соот-

ветствии с конфигурацией, заданной, например, в элементе разметки replacetokens, то необходимо обеспечить, чтобы его преобразование происходило при каждом вызове задания copy. Это можно сделать, установив атрибут overwrite, как показано ниже:

```
// Листинг 19.21
<copy todir="build/lib" overwrite="yes">
    <fileset refid="srclib"/>
    <filterchain>
        <stripphpcomments />
        <replacetokens>
            <token key="dbpass" value="${dbpass}" />
        </replacetokens>
    </filterchain>
</copy>
```

Теперь при каждом выполнении задания copy файлы, совпадающие с критерием, заданным в элементе разметки fileset, заменяются независимо от того, обновлялся ли исходный файл. Атрибуты элемента разметки copy перечислены в табл. 19.5.

Таблица 19.5. *Атрибуты элемента разметки copy*

Атрибут	Является ли обязательным	Описание	Значение по умолчанию
file	Нет	Копируемый файл	Отсутствует
todir	Да (если tofile отсутствует)	Каталог, в который выполняется копирование	Отсутствует
tofile	Да (если todir отсутствует)	Файл, в который выполняется копирование	Отсутствует
tstamp или preservelastmodified	Нет	Сравнивает время создания/модификации перезаписываемого файла (оно остается неизменным)	false
preservemode или preservepermissions	Нет	Сравнивает права доступа перезаписываемого файла	false
includeemptydirs	Нет	Копирует пустые каталоги	false

Окончание табл. 19.5

Атрибут	Является ли обязательным	Описание	Значение по умолчанию
`mode`	Нет	Задает права доступа в виде восьмеричного числа	`755`
`haltonerror`	Нет	Останавливает процесс построения при возникновении ошибки	`true`
`overwrite`	Нет	Перезаписывает целевой файл, если он существует	`no`

Задание `input`

Как было показано выше, элемент разметки `echo` служит для вывода сообщений пользователю. Чтобы получить входные данные *от* пользователя, мы применяли разные средства, включая командную строку и переменную окружения. Но эти механизмы не являются ни структурированными, ни интерактивными в достаточной степени.

На заметку Одна из причин, по которым пользователям разрешается определять значения в процессе построения, состоит в том, чтобы обеспечить достаточную гибкость в разных средах построения. Если для доступа к базе данных применяются пароли, то еще одно преимущество такого подхода заключается в том, что эти конфиденциальные данные не сохраняются в самом файле построения. Но как только процесс построения будет запущен, пароль будет сохранен в исходном файле, и поэтому обеспечение безопасности системы возлагается на ее разработчика!

Задание `input`, определяемое в элементе разметки `input`, позволяет выводить приглашение пользователю ввести данные. В этом случае Phing ожидает ввода данных пользователем и присваивает их указанному свойству. В приведенном ниже примере показано, как это осуществляется на практике:

```
// Листинг 19.22
<?xml version="1.0"?>
<!-- build xml -->
```

```
<project name="megaquiz"
        default="main" >
    <target name="setpass" unless="dbpass">
        <input message="Укажите пароль доступа к базе данных"
               propertyName="dbpass"
               defaultValue="default"
               promptChar=" >" />
    </target>
    <target name="main" depends="setpass">
        <echo>pass: ${dbpass}</echo>
    </target>
</project>
```

И в этом случае имеется целевое задание по умолчанию `main`, зависящее от другого целевого задания, `setpass`, которое отвечает за то, чтобы было задано значение свойства `dbpass`. Для этой цели служит атрибут `unless` элемента разметки `target`, который гарантирует, что задание не будет запущено, если свойство `dbpass` уже определено.

Целевое задание `setpass` состоит из единственного элемента разметки `input` одноименного задания. У элемента разметки `input` может быть атрибут `message`, который должен содержать текст приглашения для пользователя. Атрибут `propertyName` является обязательным и определяет свойство, которое заполняется введенными пользователем данными. Если в ответ на приглашение пользователь нажимает клавишу <Enter>, не задавая значения, этому свойству присваивается значение по умолчанию, указанное в атрибуте `defaultValue`. И, наконец, с помощью атрибута `promptChar` можно изменить сам символ приглашения, дающий пользователю визуальный ориентир для ввода данных. Итак, запустим Phing на выполнение, используя предыдущие целевые задания:

```
$ phing

Buildfile: /var/popp/src/ch19/build.xml
megaquiz > setpass:
You don't seem to have set a db password [default] > mypass

megaquiz > main:
    [echo] pass: mypass

BUILD FINISHED
Total time: 3.8878 seconds
```

Атрибуты элемента разметки `input` перечислены в табл. 19.6.

Таблица 19.6. Атрибуты элемента разметки `input`

Атрибут	Является ли обязательным	Описание
`propertyName`	Да	Свойство, которому присваиваются данные, введенные пользователем
`message`	Нет	Текст приглашения для пользователя
`defaultValue`	Нет	Значение, которое следует присвоить свойству, если пользователь не введет данные
`validArgs`	Нет	Список приемлемых входных значений, разделяемых запятыми. Если пользователь ввел значение, которого нет в этом списке, то Phing снова выдаст приглашение
`promptChar`	Нет	Символ приглашения
`hidden`	Нет	Если установлено, скрывает ввод пользователя

Задание `delete`

Процесс установки обычно предусматривает создание, копирование и преобразование файлов. Но иногда необходимо и удаление, особенно если требуется установка заново и начисто. Как упоминалось ранее, из исходного места в место назначения обычно копируются только исходные файлы, которые изменились со времени последнего построения. Удалив каталог построения, можно гарантировать, что процесс построения будет полностью выполнен.

В приведенном ниже примере удаляется каталог построения:

```
// Листинг 19.23
<target name="clean">
    <delete dir="build" />
</target>
```

Если выполнить команду `phing` с аргументом `clean` (имя целевого задания), как показано ниже, то вызывается задание `delete`, указанное в элементе разметки `delete`:

```
$ phing clean

Buildfile: /var/popp/src/ch19/build.xml

megaquiz > clean:
```

```
    [delete] Deleting directory /var/popp/src/ch19/build

BUILD FINISHED
Total time: 0.1000 seconds
```

У элемента разметки `delete` имеется также атрибут `file`, в котором можно указать имя конкретного файла. С другой стороны, операции удаления можно точно настроить, добавляя вложенные элементы разметки `fileset` в элемент разметки `delete`.

Резюме

Серьезный процесс разработки редко происходит в каком-то одном месте. Кодовая база должна быть отделена от своей установки, чтобы продолжающаяся работа не нарушала рабочий код, который всегда должен оставаться действующим. Контроль версий позволяет разработчикам извлекать проект и работать с ним в собственном локальном окружении. Для этого необходимо, чтобы они могли легко сконфигурировать проект в своей среде разработки. И, наконец, самое, вероятно, главное: клиент (даже если это сам разработчик, возвращающийся к своему коду через год, когда уже забыты подробности его написания) должен суметь установить проект, просмотрев файл README.

В этой главе были представлены основы Phing — замечательного инструментального средства, привнесшего значительную часть функциональных возможностей Ant в среду PHP. Возможности Phing были затронуты здесь лишь поверхностно. Тем не менее, после того как вы разберетесь в рассмотренных в этой главе целевых заданиях, заданиях, типах и свойствах, вам будет легче пользоваться новыми элементами в качестве прочного основания для освоения таких дополнительных возможностей, как создание архивных дистрибутивов в формате `tar/gzip`, автоматическая генерация инсталляций пакета PEAR и запуск кода PHP на выполнение непосредственно из файла построения.

Если по какой-то причине инструментальное средство Phing не удовлетворяет все ваши потребности в построении проекта, то оно, как, впрочем, и Ant, допускает расширение. Это позволит вам продвинуться дальше в решении своих задач! И даже если вы не собираетесь внедрять новые функциональные возможности в инструментальное средство Phing, не пожалейте времени, чтобы изучить его исходный код. Оно полностью написано на объектно-ориентированном языке PHP, и его код изобилует примерами применения стандартных проектных шаблонов.

ГЛАВА 20

Виртуальная машина Vagrant

Где вы выполняете свой код? Возможно, в среде разработки, доведенной до совершенства и дополненной самым лучшим текстовым редактором и другими полезными инструментальными средствами. Разумеется, ваша идеальная среда для написания кода заметно отличается от самой лучшей производственной системы, в которой предполагается его выполнять. Преодолеть именно это препятствие и помогает инструментальное средство Vagrant. С его помощью вы, работая на своей локальной машине, сможете выполнить написанный вами код в системе, очень похожей на рабочий сервер, и в этой главе поясняется, как добиться подобной цели. В ней рассматриваются следующие вопросы.

- *Простейшая установка.* От установки до выбора первого готового образа системы (Vagrant box).
- *Подключение и регистрация.* Исследование виртуальной машины средствами `ssh`.
- *Монтирование каталогов хоста.* Редактирование кода на хост-машине и обеспечение его прозрачной доступности на виртуальной машине Vagrant.
- *Подготовка.* Написание сценария для установки пакетов и конфигурирования веб-сервера Apache и сервера баз данных MySQL.
- *Определение имени хоста.* Конфигурирование виртуальной машины Vagrant для доступа к ней по специальному имени хоста.

Задача

Уделим, как всегда, немного внимания определению области решения стоящей задачи. В настоящее время на большинстве настольных и переносных компьютеров можно относительно просто сконфигурировать стек LAMP. Но несмотря на это ваш персональный компьютер вряд ли будет соответствовать рабочей среде. В частности, установлена ли на нем та же версия PHP? А что насчет веб-сервера Apache и базы данных MySQL? Если

вы пользуетесь поисковым механизмом Elasticsearch, то вам, возможно, потребуется и Java. Этот список можно продолжить до бесконечности. Разработка с помощью одного набора инструментальных средства на конкретной платформе может иногда оказаться затруднительной, если рабочий стек значительно отличается.

Можно, конечно, уступить и перенести процесс разработки на удаленную машину, поскольку имеется немало поставщиков облачных решений, которые позволят быстро развернуть среду разработки. Но такое решение не бесплатное, и в зависимости от ваших предпочтений, в удаленной системе, возможно, будут поддерживаться далеко не все необходимые вам инструментальные средства разработки.

В таком случае, возможно, стоит как можно точнее согласовать пакеты на своем компьютере с теми пакетами, которые устанавливаются в рабочей системе. Такое согласование, безусловно, не идеально, но оно может быть достаточным для устранения большинства недостатков на сервере, предназначенном для обкатки разработанного программного обеспечения.

Но что произойдет, когда вы начнете работать над вторым проектом с совершенно другими требованиями? Как было показано ранее, диспетчер зависимостей Composer отлично справляется с разделением зависимостей, но ведь еще остаются глобальные пакеты, подобные PHP, MySQL и Apache, версии которых требуется согласовать.

На заметку Если вы решите выполнять разработку на удаленных системах, то в таком случае рекомендуем вам рассмотреть возможность применения текстового редактора `vim`. Несмотря на некоторые особенности и неудобства работы в этом текстовом редакторе, он довольно эффективен, и в 99 случаях из 100 он и его ближайший предшественник `vi` уже установлен в любой Unix-подобной системе[1].

Потенциально неплохим решением является виртуализация, хотя установка операционной системы и конфигурирование рабочей среды могут доставить немало хлопот.

---

[1] Лучшим решением все-таки является использование протокола FTP для доступа к удаленной системе. Тогда весь процесс редактирования файлов можно будет перенести на свой рабочий компьютер и использовать привычные средства разработки. После внесения изменений в файлы они будут автоматически обновлены на удаленном сервере. Вам останется только запустить тесты в рабочей среде заказчика. — *Примеч. ред.*

Вот если бы имелось инструментальное средство, которое могло бы упростить организацию похожей на рабочую среды разработки на локальной машине! И следует признать, что такое инструментальное средство действительно существует. Оно называется "Vagrant" и обладает поистине замечательными возможностями.

Простейшая установка

Соблазнительно сказать, что Vagrant предоставляет среду разработки единственной командой. Это *может* оказаться правдой, но прежде придется установить необходимое программное обеспечение. Если после этого извлечь файл конфигурации из хранилища системы контроля версий конкретного проекта, то новую среду действительно можно будет запустить единственной командой.

Начнем с установки Vagrant. Для этого инструментального средства требуется платформа виртуализации. В нем поддерживается несколько таких платформ, но мы воспользуемся платформой VirtualBox, которую можно установить в любой разновидности ОС Linux, Mac OS X или Windows. Загрузить и получить соответствующие инструкции по установке платформы VirtualBox можно на веб-странице, доступной по адресу `https://www.virtualbox.org/wiki/Downloads`.

После VirtualBox, безусловно, потребуется установить Vagrant. Загрузить Vagrant можно по адресу `https://www.vagrantup.com/downloads.html`. Как только платформа VirtualBox и инструментальное средство Vagrant будут установлены, необходимо выбрать один из готовых образов операционной системы для выполнения кода.

Выбор и установка образа операционной системы в Vagrant

Чтобы получить один из готовых образов операционной системы для Vagrant, проще всего воспользоваться поисковым механизмом, доступным по адресу `https://app.vagrantup.com/boxes/search`. Во многих производственных системах используется CentOS, поэтому лучше искать именно ее образ. Результаты такого поиска приведены на рис. 20.1.

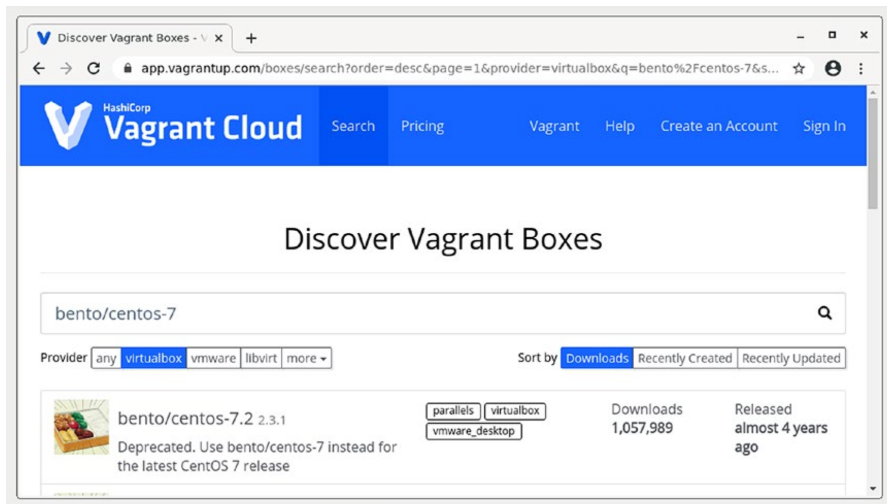


Рис. 20.1. *Поиск образа операционной системы для Vagrant*

CentOS 7 выглядит идеально подходящей для моих нужд. Я могу щелкнуть на списке для получения инструкций по установке. Я получу достаточное количество информации, чтобы получить работоспособную среду Vagrant. Когда Vagrant запускается на выполнение, как правило, выполняется чтение содержимого файла конфигурации `Vagrantfile`. Но поскольку мы все начали с "чистого листа", нам придется сформировать такой файл с помощью следующей команды:

```
$ vagrant init bento/centos-7

A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

(В приведенном выводе говорится буквально следующее: файл `'Vagrantfile'` размещен в указанном каталоге, и теперь вы готовы запустить свою первую виртуальную среду Vagrant. Пожалуйста, прочитайте комментарии в файле `Vagrantfile`, а также дополнительные сведения о применении Vagrant в документации на сайте `vagrantup.com`.)

Как видите, в приведенной выше команде указано имя готового образа операционной системы для Vagrant, в котором нам требуется работать. Эти сведения используются в Vagrant для создания минимального файла кон-

фигурации. Если открыть сформированный в итоге файл конфигурации `Vagrantfile`, то, среди прочего, в нем можно обнаружить следующую информацию о том, что нужную среду разработки можно найти по адресу `https://vagrantcloud.com/search`:

```
# Every Vagrant development environment requires a box.
# You can search for boxes at
# https://vagrantcloud.com/search.
config.vm.box = "bento/centos-7"
```

Мы всего лишь получили минимальную сгенерированную конфигурацию Vagrant. Далее нужно выполнить крайне важную команду `vagrant up`, которая должна быть хорошо известна тем, кто привык часто работать с Vagrant. По этой команде сеанс работы в Vagrant начинается с загрузки и подготовки нового образа, если в этом есть потребность, и последующего его запуска:

```
$ vagrant up
```

Поскольку мы выполняем эту команду с виртуальной машиной `bento/centos-7` в первый раз, Vagrant начинает свое выполнение с загрузки соответствующего образа операционной системы:

```
==> default: Box 'bento/centos-7' could not be found. Attempting to
    find and install...
    default: Box Provider: virtualbox
    default: Box Version: >= 0
==> default: Loading metadata for box 'bento/centos-7'
    default: URL: https://vagrantcloud.com/bento/centos-7
==> default: Adding box 'bento/centos-7' (v202008.16.0) for provider:
    virtualbox
    default: Downloading:
https://vagrantcloud.com/bento/boxes/centos-7/versions/202008.16.0/
providers/ virtualbox.box
    default: Download redirected to host: vagrantcloud-
    files-production.s3.amazonaws.com
==> default: Successfully added box 'bento/centos-7' (v202008.16.0)
    for 'virtualbox'!
```

Vagrant сохраняет образ операционной системы (в каталоге `~/.vagrant.d/boxes/`, если выполняется в Linux), чтобы вам не пришлось загружать его снова в своей системе, даже если в ней выполняется несколько виртуальных машин. Затем производятся конфигурирование и начальная загрузка виртуальной машины с подробным описанием про-

исходящих действий. По завершении данной команды можно проверить работу новой виртуальной машины, зарегистрировавшись на ней, как показано ниже:

```
$ vagrant ssh
$ pwd

/home/vagrant

$ cat /etc/redhat-release

CentOS Linux release 7.8.2009 (Core)
```

Итак, мы находимся в виртуальной среде! Чего же мы добились? Прежде всего, у нас имеется доступ к виртуальной машине, которая в какой-то степени напоминает рабочую среду. На самом деле этого не так уж мало. Как упоминалось ранее, нам хотелось бы редактировать файлы на своей локальной машине, но выполнять их в среде, похожей на рабочую. Организуем же свою среду разработки должным образом.

Пора снова вернуться к хост-машине:

```
$ exit
```

Монтирование локальных каталогов на виртуальной машине Vagrant

А теперь давайте соберем вместе ряд файлов-примеров. Мы выполнили первые свои команды `vagrant init` и `vagrant up` в каталоге, который я назвал `infrastructure`. Обратимся снова к проекту системы `webwoo`, упоминавшейся в главе 18, “Тестирование средствами PHPUnit” (разработка упрощенной версии этой системы была представлена в главе 12, “Шаблоны корпоративных приложений”). Если собрать все это вместе, то наша среда разработки примет следующий вид:

```
ch20/
    infrastructure/
        Vagrantfile
    webwoo/
        AddVenue.php
        index.php
        Main.php
        AddSpace.php
```

Наша задача — настроить среду таким образом, чтобы можно было работать с файлами из каталога webwoo локально, но выполнять их прозрачно, используя стек образа CentOS. В зависимости от конкретной конфигурации Vagrant попытается смонтировать каталоги машины хоста в гостевой операционной системе. На самом деле один каталог *уже* был автоматически смонтирован в Vagrant. Это можно проверить следующим образом:

```
$ vagrant ssh

Last login: Wed Sep 23 16:46:53 2020 from 10.0.2.2

$ ls -a /vagrant

           .vagrant    Vagrantfile
```

Таким образом, каталог infrastructure был смонтирован в среде Vagrant как каталог под именем /vagrant. Это пригодится нам, когда мы приступим к написанию сценария для подготовки и настройки образа операционной системы. А до тех пор уделим основное внимание каталогу webwoo, отредактировав файл конфигурации Vagrantfile. Однако сейчас может быть подходящее время, чтобы снова выйти из виртуальной машины. Сделав это, я открою Vagrantfile и добавляю следующую строку:

```
config.vm.synced_folder "../webwoo", "/var/www/poppch20"
```

Я могу найти лучшее место для размещения этой строки, выполнив в закомментированном шаблоне поиск строки synced_folder. Я нашел образец строки конфигурации, очень похожей на мою. В этой директиве Vagrant предписывается смонтировать каталог webwoo в гостевой операционной системе по пути /var/www/poppch20. Чтобы убедиться в этом, нам придется перезагрузить гостевую операционную систему, выполнив следующую команду (в нашей операционной системе, а не на виртуальной машине):

```
$ vagrant reload
```

В итоге виртуальная машина остановится и корректно перезагрузится, а Vagrant смонтирует каталоги infrastructure (/vagrant) и webwoo (/var/www/poppch20). Ниже приведен фрагмент результата выполнения данной команды:

```
==> default: Mounting shared folders...
    default: /vagrant => /home/mattz/localwork/popp/ch20-vagrant/
    infrastructure
    default: /var/www/poppch20 => /home/mattz/localwork/popp/
    ch20-vagrant/webwoo
```

Чтобы быстро убедиться, что каталог `/var/www/poppch20` находится на своем месте, достаточно зарегистрироваться с помощью следующей команды:

```
$ vagrant ssh
$ ls /var/www/poppch20/

AddSpace.php    AddVenue.php    index.php    Main.php
```

Теперь мы можем работать в удобной для нас интегрированной среде разработки на своей локальной машине. При этом все внесенные в файлы изменения будут автоматически переноситься в среду гостевой ОС!

На заметку Вот примечание технического обозревателя и пользователя Windows Пола Трегоинга: "Не используйте общую файловую систему VirtualBox (которая лежит в основе синхронизированной папки Vagrant в этом примере), если в качестве хоста используется Windows. Если вы это сделаете, у вас могут возникнуть проблемы, связанные с чувствительностью файловой системы к регистру и отсутствием поддержки символических ссылок. В этом случае в гостевой операционной системе лучше запустить Samba (большинство дистрибутивов устанавливают этот пакет как `smbd`) и сопоставить сетевой диск на хосте для беспроблемной работы. Имеется много онлайн-руководств, рассказывающих, как это делается".

Безусловно, размещение файлов на виртуальной машине CentOS не равнозначно их выполнению в рабочей системе. В типичном образе операционной системы для Vagrant сделано не так уж и много предустановок, поскольку считается, что разработчику потребуется специально настроить виртуальную среду для своих потребностей и обстоятельств.

Поэтому следующая стадия состоит в подготовке образа операционной системы и его настройке.

Подготовка

Следует еще раз подчеркнуть, что вся подготовка образа ОС выполняется согласно директивам файла конфигурации `Vagrantfile`. Для подготовки виртуальных машин в Vagrant можно применять несколько инструментальных средств, в том числе Chef (`https://www.chef.io/chef/`),

Puppet (https://puppet.com) и Ansible (https://www.ansible.com). Все они заслуживают специального изучения, но для рассматриваемых здесь целей мы воспользуемся старым добрым сценарием командной оболочки. И начнем снова с файла конфигурации Vagrantfile:

```
config.vm.provision "shell", path: "setup.sh"
```

Назначение этой директивы должно быть совершенно ясно! В ней для подготовки образа среде Vagrant предписывается воспользоваться сценарием командной оболочки, а также указан файл setup.sh с исполняемым сценарием.

Содержимое такого сценария, разумеется, зависит от конкретных требований разработчика. В данном случае мы начнем свой сценарий с определения значения двух переменных окружения и установки некоторых пакетов, как показано ниже:

```
#!/bin/bash

VAGRANTDIR=/vagrant
SERVERDIR=/var/www/poppch20/

sudo yum -q -y install epel-release yum-utils
sudo yum -q -y install
Ä    http://rpms.remirepo.net/enterprise/remi-release-7.rpm

yum-config-manager --enable remi-php80

sudo yum -q -y install mysql-server
sudo yum -q -y install httpd;
sudo yum -q -y install php
sudo yum -q -y install php-common
sudo yum -q -y install php-cli
sudo yum -q -y install php-mbstring
sudo yum -q -y install php-dom
sudo yum -q -y install php-mysql
sudo yum -q -y install php-xml
sudo yum -q -y install php-dom
```

По умолчанию версия PHP 8 недоступна в CentOS 7, но если установить пакет remi-release-7.rpm, то можно будет установить и новейшие версии PHP. Я записываю свой сценарий в файл с именем setup.sh, который помещаю в каталог инфраструктуры вместе с Vagrantfile.

Но как запустить процесс подготовки? Если бы директива config.vm.provision и файл сценария setup.sh были уже на месте, когда мы

выполняли команду vagrant up, то подготовка прошла бы автоматически. Поэтому нам придется провести ее вручную, выполнив следующую команду:

```
$ vagrant provision
```

Выполнение сценария из файла setup.sh на виртуальной машине Vagrant приведет к выводу довольно большого количества информации на терминал. А теперь выясним, насколько успешно прошла подготовка образа ОС, выполнив приведенные ниже команды:

```
$ vagrant ssh
$ php -v

PHP 8.0.0 (cli) (built: Nov 24 2020 17:04:03) ( NTS gcc x86_64 )
Copyright (c) The PHP Group
Zend Engine v4.0.0-dev, Copyright (c) Zend Technologies
```

Настройка веб-сервера

Безусловно, систему еще нельзя считать готовой к работе, даже если установлен веб-сервер Apache. Прежде всего веб-сервер Apache необходимо сконфигурировать. Чтобы сделать это, проще всего создать приведенный ниже файл конфигурации, который можно затем скопировать в каталог conf.d веб-сервера Apache. Присвоим этому файлу имя poppch20.conf и перенесем его в каталог infrastructure:

```
<VirtualHost *:80>
    ServerAdmin matt@getinstance.com
    DocumentRoot /var/www/poppch20
    ServerName poppch20.vagrant.internal
    ErrorLog logs/poppch20-error_log
    CustomLog logs/poppch20-access_log common
</VirtualHost>

<Directory /var/popp/wwwch20>
AllowOverride all
</Directory>
```

Мы еще вернемся к этому имени хоста в дальнейшем. А до тех пор, особо не вдаваясь в подробности, достаточно сообщить веб-серверу Apache о каталоге /var/www/poppch20 и установить режим протоколирования. Необходимо также обновить сценарий из файла setup.sh, чтобы скопировать файл конфигурации во время подготовки, как показано ниже:

```
sudo cp $VAGRANTDIR/poppch20.conf /etc/httpd/conf.d/
systemctl start httpd
systemctl enable httpd
```

Сначала файл конфигурации копируется в нужное место, а затем перезапускается веб-сервер, чтобы произвести его конфигурирование. Кроме того, выполняется команда `systemctl enable`, чтобы обеспечить запуск веб-сервера в момент начальной загрузки виртуальной машины. Внеся эти изменения в сценарий, можно снова запустить его на выполнение следующей командой:

```
$ vagrant provision
```

Следует, однако, иметь в виду, что при повторном выполнении сценария подготовки снова вступят в действие и те его части, которые были рассмотрены ранее. Поэтому разрабатывать сценарий подготовки следует таким образом, чтобы его можно было выполнять повторно без серьезных последствий. Правда, диспетчер пакетов Yum обнаружит, что указанные нами пакеты уже установлены, и не выдаст особых предупреждений отчасти потому, чтобы мы приняли к этому меры, указав в командной строке параметр `-q`, подавляющий вывод информационных сообщений.

Настройка сервера баз данных MariaDB

Для работы большинства приложений требуется обеспечить доступ к базе данных. Вот простое добавление в рассматриваемый здесь сценарий подготовки:

```
sudo yum -q -y install mariadb-server
systemctl start mariadb
systemctl enable mariadb

/usr/bin/mysqladmin -s -u root password 'vagrant' ||
Ä echo " -- задать пароль не удалось - вероятно, это уже сделано"
domysqldb vagrant poppch20_vagrant vagrant vagrant

ROOTPASS=vagrant
DBNAME=poppch20_vagrant
DBUSER=vagrant
DBPASS=vagrant
MYSQL=mysql
MYSQLROOTCMD="mysql -uroot -p$ROOTPASS"
```

```
echo "Создание базы данных $DBNAME..."
echo "CREATE DATABASE IF NOT EXISTS $DBNAME" | $MYSQLROOTCMD ||
Ä die "Невозможно создать базу данных";
echo "grant all on $DBNAME.* to $DBUSER@'localhost' identified
Ä by \"$DBPASS\"" | $MYSQLROOTCMD ||
Ä die "Невозможно настроить привилегии пользователя $DBUSER"
echo "FLUSH PRIVILEGES" | $MYSQL -uroot -p"$ROOTPASS" ||
Ä die "Невозможно установить привилегии"
```

Я устанавливаю MariaDB — современную замену MySQL (созданную разработчиками MySQL и полностью совместимую с MySQL, с реализацией знакомых инструментов и команд MySQL). Сначала выполняется команда `mysqladmin` для определения пароля суперпользователя. Если эта команда запускается не в первый раз, то, скорее всего, она завершится неудачно, поскольку пароль уже был нами задан. Поэтому в ней указан параметр `-s` для подавления сообщений об ошибках. На тот случай, если данная команда завершится неудачно, мы предусмотрели соответствующее предупреждающее сообщение. Далее создаются база данных, пользователь и пароль.

Внеся все эти изменения, можно произвести подготовку снова, а затем проверить работоспособность сервера баз данных, выполнив приведенные ниже команды:

```
$ vagrant provision

# Выводится много разной информации

$ vagrant ssh
$ mysql -uvagrant -pvagrant poppch20_vagrant

Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 8
Server version: 5.5.65-MariaDB MariaDB Server
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current
input statement.
MariaDB [poppch20_vagrant]>
```

Итак, мы привели в рабочее состояние веб-сервер и сервер баз данных. Теперь можно перейти к выполнению прикладного кода.

Настройка имени хоста

Мы уже несколько раз подключались к нашей новой среде разработки, поэтому вопросы настройки сети нами более или менее разрешены. Но, несмотря на то что мы сконфигурировали веб-сервер, мы пока еще не можем им воспользоваться, потому что нам по-прежнему требуется поддержка имени хоста для виртуальной машины. Поэтому введем его в файл конфигурации `Vagrantfile` следующим образом:

```
config.vm.hostname = "poppch20.vagrant.internal"
config.vm.network :private_network, ip: "192.168.33.148"
```

Мы придумали имя хоста и задали его с помощью директивы `config.vm.hostname`. Кроме того, мы сконфигурировали приватную сеть с помощью директивы `config.vm.network`, присвоив виртуальному хосту статический IP-адрес. Для этой цели следует использовать любой свободный IP-адрес, который начинается с префикса `192.168`.

Поскольку наше имя хоста вымышленное, придется настроить гостевую операционную систему на его правильное преобразование. В Unix-подобной системе это означает, что необходимо внести изменения в системный файл `/etc/hosts`. В данном случае в него необходимо добавить следующую строку:

```
192.168.33.148    poppch20.vagrant.internal
```

На заметку Файл `hosts` в Windows — `%windir%\system32\drivers\etc\hosts`.

Мы постепенно продвигаемся в направлении установки Vagrant для всей команды разработчиков с помощью единственной команды, так что было бы неплохо автоматизировать эту стадию процесса. К счастью, в Vagrant поддерживаются подключаемые модули, а среди них — модуль `hostmanager`, который делает именно то, что нам требуется. Для добавления подключаемого модуля следует выполнить следующую команду:

```
$ vagrant plugin install vagrant-hostmanager

Installing the 'vagrant-hostmanager' plugin.
This can take a few minutes...
Installed the plugin 'vagrant-hostmanager (1.8.9)'!
```

Затем можно явно указать установленному подключаемому модулю обновить файл /etc/hosts следующим образом:

```
$ vagrant hostmanager
[default] Updating /etc/hosts file...
```

Чтобы автоматизировать данный процесс для других членов команды разработчиков, мы должны явно активизировать подключаемый модуль hostmanager в файле конфигурации Vagrantfile, как показано ниже:

```
config.hostmanager.enabled = true
```

Внеся все необходимые изменения в файл конфигурации, мы должны выполнить команду vagrant reload, чтобы применить их. И здесь наступает момент истины! Будет ли наша система работать в браузере? Как показано на рис. 20.2, система должна работать исправно.

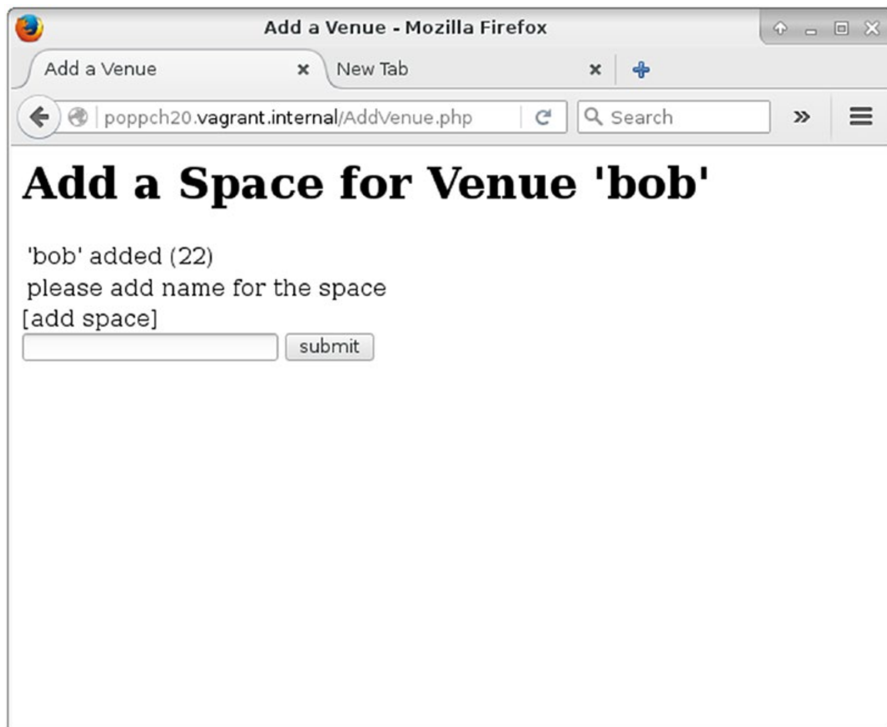


Рис. 20.2. *Доступ к сконфигурированной системе на виртуальной машине Vagrant*

Краткие итоги

Итак, мы организовали практически из ничего полностью функционирующую среду разработки. Если учесть, что для рассмотрения этого процесса потребовалась отдельная глава, то было бы нечестно сказать, что пользоваться Vagrant легко и просто. Подобная простота объясняется двумя обстоятельствами. Во-первых, выполнив описанные выше действия несколько раз, вы сможете в конечном итоге довольно быстро развернуть очередную среду Vagrant. Это, конечно, намного проще, чем манипулировать стеками зависимостей вручную.

И во-вторых, что важнее, настоящий выигрыш в скорости и эффективности зависит не от того, кто устанавливает Vagrant. Представьте, что к вашему проекту присоединился новый разработчик, который заранее рассчитывает, что ему потребуется не один день, чтобы настроить свою среду разработки. А вы говорите, что ему достаточно установить Vagrant и VirtualBox, извлечь исходный код и из каталога `infrastructure` выполнить команду `vagrant up`. Сравните это с трудоемкими процессами вхождения в проект, которые вам приходилось испытывать на своем опыте или слышать от других.

В этой главе мы лишь поверхностно коснулись возможностей Vagrant. Если вам потребуется настроить Vagrant на более сложные задачи, примите к сведению, что на официальном веб-сайте этого инструментального средства по адресу `https://www.vagrantup.com` представлена вся необходимая для этого информация. В табл. 20.1 перечислены команды Vagrant, упоминавшиеся в этой главе.

***Таблица 20.1.** Некоторые команды Vagrant*

Команда	Описание
`vagrant up`	Загружает виртуальную машину и проводит подготовку, если она еще не проведена
`vagrant reload`	Останавливает систему и перезапускает ее снова (подготовка не запускается, если не указан флаг `--provision`)
`vagrant plugin list`	Выводит список установленных подключаемых модулей

Окончание табл. 20.1

Команда	Описание
`vagrant plugin install` <*имя_модуля*>	Устанавливает указанный подключаемый модуль
`vagrant provision`	Выполняет снова стадию подготовки, что удобно после обновления сценариев подготовки
`vagrant halt`	Корректно останавливает виртуальную машину
`vagrant suspend`	Приостанавливает виртуальную машину и сохраняет состояние
`vagrant resume`	Восстанавливает работу приостановленной машины
`vagrant init`	Создает новый файл настроек `Vagrantfile`
`vagrant destroy`	Уничтожает виртуальную машину. Ее можно восстановить по команде `vagrant up`, так что не волнуйтесь!

Резюме

В этой главе было представлено инструментальное средство Vagrant, позволяющее работать в похожей на рабочую среде разработки без ущерба для авторских инструментальных средств. В ней были рассмотрены стадии установки, выбора дистрибутивной версии и первоначальной настройки, включая установку каталогов для разработки. После установки виртуальной машины в этой главе был описан процесс подготовки, охватывающий установку пакетов, веб-сервера и сервера баз данных, а также их конфигурирование. В конце была рассмотрена стадия определения имени хоста, а также продемонстрирована работоспособность системы в браузере!

ГЛАВА 21

Непрерывная интеграция

В предыдущих главах был представлен ряд инструментальных средств, предназначенных для поддержки хорошо управляемых программных проектов. Модульное тестирование, автоматическое составление документации, построение проектов и контроль их версий — все это чрезвычайно полезно! Правда, некоторые из них, особенно инструментальные средства тестирования, могут доставить немало хлопот.

Даже если на выполнение тестов необходимо всего лишь несколько минут, вы нередко оказываетесь всецело поглощенными программированием, не уделяя тестам должного внимания. Помимо этого, ваши коллеги и клиенты часто ждут от вас новых выпусков кода и новых функциональных возможностей. Искушение полностью погрузиться в процесс программирования существует всегда, но ошибки намного легче выявить и локализовать, если после написания фрагмента кода сразу же протестировать его, не откладывая дело в долгий ящик. Благодаря этому можно быстро понять, какие именно изменения, внесенные в исходный код, привели к ошибкам, а следовательно, быстро их устранить.

Здесь представлена *непрерывная интеграция* — методика автоматизации процесса тестирования и построения проекта, а также объединения упоминавшихся ранее инструментальных средств и методик.

В этой главе будут рассмотрены следующие вопросы.

- Определение непрерывной интеграции.
- Подготовка проекта к непрерывной интеграции.
- Представление о Jenkins — сервере непрерывной интеграции.
- Настройка сервера Jenkins на проекты PHP с помощью специализированного подключаемого модуля.

Что такое непрерывная интеграция

В старые добрые времена интеграция была чем-то таким, что вам приходилось выполнять после завершения приятной части работы. Это была именно та стадия, на которой разработчики начинали отчетливо понимать,

сколько еще работы им предстоит выполнить. Интеграция — это процесс, в ходе которого все части проекта собирались в единый пакет, чтобы затем доставить его конечному потребителю и развернуть в условиях эксплуатации. В этом процессе не было ничего захватывающего, и на самом деле он был весьма трудоемким.

Интеграция тесно связана с процессом контроля качества. Нельзя же в самом деле отправить продукт конечному потребителю, если он не соответствует его требованиям. А это означает тестирование, причем интенсивное. И если до этого не было проведено никаких тестов, то на стадии интеграции возникнет немало (и даже очень много!) неприятных сюрпризов.

Как отмечалось в главе 18, "Тестирование средствами PHPUnit", раннее и частое тестирование считается передовой нормой практики. А в главах 15, "Стандарты PHP", и 19, "Автоматическое построение средствами Phing", упоминалось, что, проектируя систему, следует всегда иметь в виду процесс ее развертывания. Многие согласятся, что так должно быть в идеале, но как часто это случается на практике?

Если вы практикуете методику *разработки посредством тестирования*[1] (Test-Oriented Development — TOD), то должны знать, что писать тесты не так утомительно, как может показаться на первый взгляд. Ведь в любом случае в процессе программирования приходится писать тесты. И всякий раз, когда разрабатывается очередной компонент, требуется создать фрагмент кода (возможно, в конце того же самого файла класса), в котором получаются экземпляры объектов и вызываются их методы. Если теперь собрать вместе все разрозненные фрагменты кода, написанные для тестирования компонента в процессе его разработки, то в конечном итоге получится контрольный пример. Остается лишь оформить его в виде класса и добавить в набор тестов.

Как ни странно, в процессе написания кода разработчики нередко избегают *выполнения* тестов. Но впоследствии для выполнения тестов им понадобится уже намного больше времени. Отказы, связанные с известными недостатками, накапливаются как снежный ком, что затрудняет выявление новых недостатков. Кроме того, возникает подозрение, что кто-то зафиксировал в хранилище исходный код, который не прошел тестирование, а времени на то, чтобы прервать свою работу и заняться исправлением

---

[1] Этот термин мне нравится больше, чем *предварительное тестирование*, поскольку он лучше отражает реалии большинства удачных проектов, в которых мне довелось принимать участие.

ошибок, внесенных другими разработчиками, просто нет. Поэтому гораздо лучше сразу выполнить несколько тестов, непосредственно относящихся к текущей работе, чем потом тестировать все вместе.

Если не выполнить тесты вовремя, то вряд ли удастся устранить недостатки, которые выявляют эти тесты, а это, в свою очередь, еще больше затруднит выявление причин возникающих ошибок. Основные усилия при поиске ошибок обычно тратятся на диагностику неполадок, а не на их устранение. Зачастую на устранение неполадок приходится тратить лишь несколько минут, тогда как на их выявление — несколько часов. Если же тест не прошел сразу или по прошествии нескольких часов после внесения исправлений в исходный код, определить место, где следует искать причину неполадки, не составит большого труда.

Аналогичные затруднения возникают и при сборке программного обеспечения. Если вы редко делаете тестовые инсталляции своего проекта, то, вероятнее всего, обнаружите, что на вашем рабочем компьютере все работает замечательно, но при попытке инсталлировать проект на другой компьютер процесс завершается с маловразумительным сообщением об ошибке. И чем больше проходит времени между последовательными стадиями развертывания проекта, тем сложнее обнаружить причину этой скрытой ошибки.

Зачастую причина такой ошибки очень проста: необъявленная зависимость от какой-нибудь библиотеки, установленной в вашей системе, или один или несколько классов, которые вы забыли включить в пакет. Этот недостаток, конечно, легко устранить, если исходный код у вас под рукой. Но что делать, если процесс построения завершился неудачно, а вы в это время уехали в отпуск? Тот несчастный член вашей команды, которому поручена работа по построению и выпуску продукта, может и не знать о сделанных вами настройках и местах, где хранятся пропущенные файлы.

Количество ошибок интеграции растет по мере увеличения числа сотрудников, задействованных в проекте. Вы можете очень хорошо относиться к членам вашего коллектива, но ведь все знают, что это именно они, а не вы забыли выполнить вовремя тесты. И это они могут в конце рабочего дня в пятницу зафиксировать в хранилище результаты работы за неделю только потому, что до этого вы заявили на совещании, что проект почти готов и его уже можно готовить к выпуску.

Непрерывная интеграция (НИ) как раз и призвана избавить вас от описанных выше проблем (или хотя бы свести их к минимуму) благодаря автоматизации процессов построения и тестирования. НИ — это ряд норм

практики и набор инструментальных средств. В качестве нормы практики требуется частая фиксация кода в хранилище (по крайней мере один раз в день). Перед каждой фиксацией должны быть выполнены соответствующие тесты и построение всех необходимых пакетов. Что касается инструментальных средств, требующихся для НИ, то некоторые из них (в частности, с PHPUnit и Phing) упоминались в предыдущих главах, хотя отдельных средств зачастую оказывается недостаточно. Для координации и автоматизации всех процессов требуется система более высокого уровня, а именно — сервер НИ. Без него очень велика вероятность, что все методики НИ будут сведены на нет и подчинены нашему естественному желанию обходить трудности. Ведь мы всегда предпочитаем программирование другим видам работ.

Поддерживая такую систему, вы получаете три очевидных преимущества. Главное из них — построение и тестирование ваших проектов будет выполняться часто. Собственно говоря, это и является конечной целью НИ. А сама автоматизация процессов дает два других преимущества. Тестирование и построение проектов будут выполняться в другой среде, отличающейся от среды разработки. Причем все это будет происходить в фоновом режиме, и вам не придется отвлекаться от своей основной работы на выполнение тестов. Подобно тестированию, НИ способствует улучшению структуры проекта в целом. Чтобы автоматизировать процесс установки в удаленном месте, вам придется с самого начала упростить его настолько, насколько это возможно.

Я не могу точно сказать, сколько раз мне приходилось иметь дело с проектами, в которых процедура установки была тайной, покрытой мраком и известной только нескольким разработчикам. "Ты говоришь, что не можешь задать правила замены URL? — спросил меня один из опытных разработчиков с едва скрываемым пренебрежением. — Ты же знаешь, что они описаны в Википедии. Просто скопируй их оттуда и вставь в файл конфигурации Apache". Разработка проекта с учетом НИ облегчает тестирование и установку системы. Это означает, что нам придется потратить немного времени на предварительную подготовку к проекту, но впоследствии эти усилия окупятся сторицей и сделают нашу жизнь легче, причем намного!

Сначала мы рассмотрим некоторые из затратных подготовительных работ. Но на самом деле они еще не раз будут упоминаться в этой главе.

Подготовка проекта к непрерывной интеграции

Прежде всего нам потребуется проект, который должен быть непрерывно интегрирован. Для этой цели вполне подойдет проект, для которого уже написаны тесты. Самым очевидным тому примером служит проект, упоминавшийся в главе 18, "Тестирование средствами PHPUnit", в качестве иллюстрации возможностей инструментального средства PHPUnit. Назовем его `userthing`, поскольку это нечто (*thing*), содержащее пользовательский объект типа `User`.

На заметку Некоторые из инструментов, описанных в этой главе, либо выпущены совсем недавно, либо находились в стадии бета-тестирования на момент написания. Это привело к нескольким нестыковкам и несовместимости. Вполне вероятно, что эти проблемы уже будут устранены к тому времени, когда вы все это будете читать. Однако для создания надежной системы НИ для этих примеров мне пришлось откатиться к более ранним версиям PHP и PHPUnit. Это должно не иметь значения для показанного кода и конфигурации.

Прежде всего выясним структуру каталогов данного проекта, выполнив следующую команду:

```
$ find src/ test/

src/
src/persist
src/persist/UserStore.php
src/util
src/util/Validator.php
src/domain
src/domain/User.php
test/
test/persist
test/persist/UserStoreTest.php
test/util
test/util/ValidatorTest.php
```

Как видите, структура каталогов данного проекта приведена в некоторый порядок и дополнена несколькими каталогами для пакетов. В исходном коде структура пакетов поддерживается благодаря пространствам имен. Теперь, когда у нас имеется проект, его необходимо перенести в хранилище системы контроля версий.

Непрерывная интеграция и контроль версий

Для НИ система контроля версий играет важную роль. Система НИ должна уметь запрашивать последнюю версию файлов проекта без вмешательства человека — по крайней мере после того, как все будет настроено.

Для данного примера воспользуемся хранилищем, установленным на сервере Bitbucket. Ниже приведены команды для конфигурирования исходного кода в локальной среде разработки, его фиксации и переноса на удаленный сервер:

```
$ cd path/to/userthing
$ git init
$ git remote add origin git@bitbucket.org:getinstance/userthing.git
$ git add build.xml composer.json src/ test/
$ git commit -m 'initial commit'
$ git push -u origin master
```

На заметку В этих примерах я создал и использовал репозиторий Bitbucket. Но я мог бы так же легко использовали GitHub, который теперь поддерживает бесплатные частные репозитории.

Сначала мы осуществили переход в каталог разработки и произвели инициализацию локального хранилища данного проекта. Затем ввели ветку `origin` на удаленном сервере, в которую необходимо перенести код. Чтобы убедиться, что все работает как следует, выполним тестовое клонирование кода следующей командой:

```
$ git clone git@bitbucket.org:getinstance/userthing.git

Cloning into 'userthing'...
X11 forwarding request failed on channel 0
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 16 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (16/16), done.
Checking connectivity… done.
```

Проверка связи прошла успешно, и теперь у нас есть хранилище проекта `userthing` на удаленном сервере и локальная копия клона этого кода. Настало время автоматизировать процесс построения и тестирования.

Применение Phing

С Phing вы уже встречались в главе 19, "Автоматическое построение средствами Phing". В этой главе я использую версию, установленную с помощью Composer:

```
"require-dev": {
    "phing/phing": "3.*"
}
```

На заметку На момент написания этой главы Phing 3 все еще находился на стадии альфа-тестирования, и установка с помощью Composer не удалась из-за проблем с зависимостями. Были также проблемы с версией phar и PHP 8. В результате для этой главы мы воспользовались ветвью Phing с исправлениями для решения различных проблем с кодом и совместимостью. Надеюсь, когда вы будете читать эти строки, Phing снова станет стабильным! Вы можете найти информацию о версии и инструкции по установке Phing по адресу `www.phing.info/#install`.

Воспользуемся этим замечательным инструментальным средством для построения проектов в качестве связующего звена для среды НИ наших проектов. Для этого выполним приведенную выше команду для установки утилиты Phing на сервере, предназначенном для тестирования (хотя вы можете попробовать сделать то же самое на виртуальном сервере с помощью Vagrant и VirtualBox). Нам нужно указать целевые задания для построения и тестирования кода, а также для запуска всех остальных средств проверки качества, о которых пойдет речь в этой главе.

Создадим следующее целевое задание:

```
// Листинг 21.1
<project name="userthing" default="build" basedir=".">
    <property name="build" value="./build" />
    <property name="test" value="./test" />
    <property name="src" value="./src" />
    <property name="version" value="1.1.1" />

    <target name="build">
        <mkdir dir="${build}" />
        <copy todir="${build}/src">
            <fileset dir="${src}">
            </fileset>
        </copy>
        <copy todir="${build}/test">
            <fileset dir="${test}">
```

```
            </fileset>
        </copy>
        </target>
        <target name="clean">
            <delete dir="${build}" />
    </target>
</project>
```

Здесь мы задали значения четырех свойств проекта. В свойстве `build` указан каталог, в котором можно будет выполнить предварительную сборку файлов проекта перед формированием пакета. В свойстве `test` указан каталог для тестирования, в свойстве `src` — каталог с исходным кодом, а в свойстве `version` — номер версии пакета.

В целевом задании `build` каталоги `src` и `test` копируются в среду построения. В более сложных проектах на этой стадии пришлось бы выполнять различные преобразования данных, оперативно создавать файлы конфигурации и собирать различные двоичные ресурсы. Это целевое задание выполняется по умолчанию при построении рассматриваемого здесь проекта с помощью вызова команды `phing` без параметров.

В целевом задании `clean` удаляется каталог построения и все его содержимое. Итак, запустим процесс построения проекта следующей командой:

```
$ ./vendor/bin/phing

Buildfile: /var/popp/src/ch21/build.xml
userthing > build:
 [mkdir] Created dir: /var/popp/src/ch21/build
  [copy] Created 4 empty directories in /var/popp/src/ch21/build/src
  [copy] Copying 3 files to /var/popp/src/ch21/build/src
  [copy] Created 3 empty directories in /var/popp/src/ch21/build/test
  [copy] Copying 2 files to /var/popp/src/ch21/build/test
BUILD FINISHED
Total time: 1.8206 second
```

Модульное тестирование

Основой процесса непрерывной интеграции является модульное тестирование. Дело в том, что не имеет смысла проводить процесс построения проекта до конца, если в нем содержится неработоспособный код. Модульное тестирование средствами PHPUnit было описано в главе 18, “Тестирование средствами PHPUnit”. И если вы еще не читали эту главу, то установите данное замечательное средство, прежде чем двигаться дальше. Ниже приведен один из способов установить PHPUnit глобально:

```
$ wget https://phar.phpunit.de/phpunit.phar
$ chmod 755 phpunit.phar
$ sudo mv phpunit.phar /usr/local/bin/phpunit
```

Установить PHPUnit можно также с помощью диспетчера зависимостей Composer:

```
"require-dev": {
    "phpunit/phpunit": "^9"
}
```

Именно такой способ и выбран для рассматриваемого здесь примера. Это никак не повлияет на содержимое каталога разработки, поскольку инструментальное средство будет установлено в каталоге `vendor/`.

Каталог тестирования отделен от остальной части исходного кода, поэтому нам придется установить правила автозагрузки, чтобы интерпретатор PHP смог найти все классы системы во время тестирования. Ниже приведен завершенный вариант файла `composer.json`, предназначенного для этой цели:

```
"require-dev": {
    "phpunit/phpunit": "^9"
},
"autoload": {
    "psr-4": {
        "userthing\\": ["src/", "test/"]
    }
}
```

Не забывайте запускать `composer update` после внесения любых изменений в файл `composer.json`.

Кроме того, в главе 18, "Тестирование средствами PHPUnit", мы написали тесты для одной из версий кода проекта `userthing`, с которым собираемся работать в этой главе. Нам нужно выполнить эти тесты снова (из каталога `src`), чтобы убедиться, что в процессе реорганизации ничего не было нарушено:

```
$ vendor/bin/phpunit test/

PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
.......               7 / 7 (100%)
Time: 00:00.002, Memory: 18.00 MB
OK (7 tests, 7 assertions)
```

Приведенный выше результат подтверждает, что все тесты проходят, хотя их лучше было бы выполнять средствами Phing.

Для этой цели в Phing предусмотрено задание exec, из которого можно вызвать команду phpunit. Но всегда лучше воспользоваться специализированным средством, если оно доступно. Ниже приведено определение встроенного задания для выполнения этой работы:

```
// Листинг 21.2
<target name="test" depends="build">
    <phpunit bootstrap="${phing.dir}/vendor/autoload.php"
            printsummary="true">
    <formatter type="plain" usefile="false"/>
    <batchtest>
        <fileset dir="${test}">
            <include name="**/*Test.php"/>
        </fileset>
    </batchtest>
    </phpunit>
</target>
```

Эти тесты являются модульными, а не функциональными, поэтому их можно выполнить прямо в каталоге src/ и не создавать отдельный устанавливаемый экземпляр вместе с функционирующей базой данных и веб-сервером. В задании phpunit среди прочих атрибутов можно определить атрибут printsummary, обеспечивающий выдачу сводного отчета о результатах выполнения тестирования.

Большая часть функциональных возможностей данного задания сконфигурирована с помощью вложенных элементов разметки. Так, элемент разметки formatter определяет способ формирования тестовых данных. В данном случае выбран простой удобочитаемый формат вывода. А в элементе batchtest разметки можно определить несколько тестовых файлов с помощью вложенного элемента разметки fileset.

На заметку У задания phpunit имеется большое количество параметров настройки. Все они описаны на странице справочного руководства по Phing, доступной по адресу http://www.phing.info/guide/chunkhtml/PHPUnitTask.html.

Выполнить тесты средствами Phing можно с помощью следующей команды:

```
$ ./vendor/bin/phing test

Buildfile: /vagrant/poppch21/build.xml
```

```
userthing > build:

userthing > test:

[phpunit] Testsuite: userthing\persist\UserStoreTest
[phpunit] Tests run: 4, Warnings: 0, Failures: 0, Errors: 0,
    Incomplete: 0, Skipped: 0, Time elapsed: 0.00684 s
[phpunit] Testsuite: userthing\util\ValidatorTest
[phpunit] Tests run: 3, Warnings: 0, Failures: 0, Errors: 0,
    Incomplete: 0, Skipped: 0, Time elapsed: 0.01188 s
[phpunit] Total tests run: 7, Warnings: 0, Failures: 0, Errors: 0,
    Incomplete: 0, Skipped: 0, Time elapsed: 0.02169 s
```

Документация

Одним из принципов НИ является прозрачность. Именно по этой причине очень важно поддерживать в актуальном состоянии документацию, в которой будут описаны все недавно введенные классы и методы, особенно если вы собираетесь создавать проект в среде НИ.

Поскольку последняя версия PHPDocumentor находится в стадии активной разработки, его установка с помощью Composer в настоящее время не приветствуется. По той же причине страница PHPDocumentor на сайте GitHub может быть наилучшим местом для получения последней информации:

```
$ wget -O phpDocumentor
Ä    https://github.com/phpDocumentor/phpDocumentor/

releases/download/v3.0.0-rc/
phpDocumentor.phar

$ chmod 755 ./phpDocumentor
$ mv phpDocumentor /usr/local/bin/phpDocumentor
```

Работая от имени суперпользователя, я загружаю `phpDocumentor v3`, делаю его исполняемым и перемещаю в центральное положение в моей системе.

На этот раз для безопасности запустим установленную выше систему документирования из каталога `build` следующей командой:

```
$ cd ./build
$ phpDocumentor --directory=src --target=docs --title=userthing
```

В итоге будет сформирована довольно приличная документация. Перед тем как опубликовать ее на сервере НИ, мне пришлось попотеть над тем, чтобы написать несколько реальных блоков документации.

Как и прежде, нам нужно поместить эту команду в файл build.xml, хотя для интеграции с PHPDocumentor в Phing было предусмотрено специальное встроенное задание phpdoc2. Однако, поскольку этот способ не очень хорошо интегрируется с phar-версией PHPDocumentor, я буду вызывать этот инструмент с помощью более грубого задания exec:

```
// Листинг 21.3
<target name="doc" depends="build">
    <mkdir dir="reports/docs" />
    <exec executable="/usr/local/bin/phpDocumentor"
          dir="${phing.dir}">
        <arg line=" --directory=build/src --target=reports/docs
                    --title=userthing" />
    </exec>
</target>
```

В этом случае задание doc вновь зависит от задания build. Сначала создается выходной каталог reports/docs, а затем вызывается PHPDocumentor с помощью задания exec. Оно анализирует вложенный элемент разметки arg, который используется для указания аргументов команды.

Покрытие исходного кода тестами

Полагаться на тесты, если они не применяются к написанному вами коду, не имеет никакого смысла. В PHPUnit имеется возможность выводить отчеты по покрытию кода тестами. Ниже приведена выдержка из инструкции по применению PHPUnit:

```
--coverage-html   <dir>   Сформировать отчет по покрытию
                          кода тестами в формате HTML.
--coverage-clover <file>  Записать данные по покрытию
                          кода тестами в формате Clover XML.
```

Чтобы воспользоваться этой возможностью, следует установить расширение, такое как Xdebug или pcov. Я продемонстрирую, как создавать отчеты о покрытии с помощью Xdebug (использование pcov — очень похожий процесс). Вы можете узнать больше об этих расширениях по адресам http://pecl.php.net/package/Xdebug и https://pecl.php.net/package/pcov (информацию об установке можно найти по адресам

http://xdebug.org/docs/install и https://github.com/krakjoe/pcov/blob/develop/INSTALL.md). Вы также можете установить эти пакеты непосредственно с помощью системы управления пакетами вашего дистрибутива Linux. Вот как это должно работать, например, в Fedora:

```
$ sudo yum install php-xdebug
```

Ниже приведена команда с параметром, активизирующим режим покрытия исходного кода тестами при запуске PHPUnit из каталога src:

```
$ export XDEBUG_MODE=coverage
$ ./vendor/bin/phpunit --whitelist src/ --coverage-html coverage test

PHPUnit 9.5.0 by Sebastian Bergmann and contributors.
.......                    7 / 7 (100%)
Time: 00:00.121, Memory: 12.00 MB
OK (7 tests, 7 assertions)
```

Посмотреть полученный в итоге отчет о покрытии исходного кода тестами можно, открыв его в окне браузера, как показано на рис. 21.1.

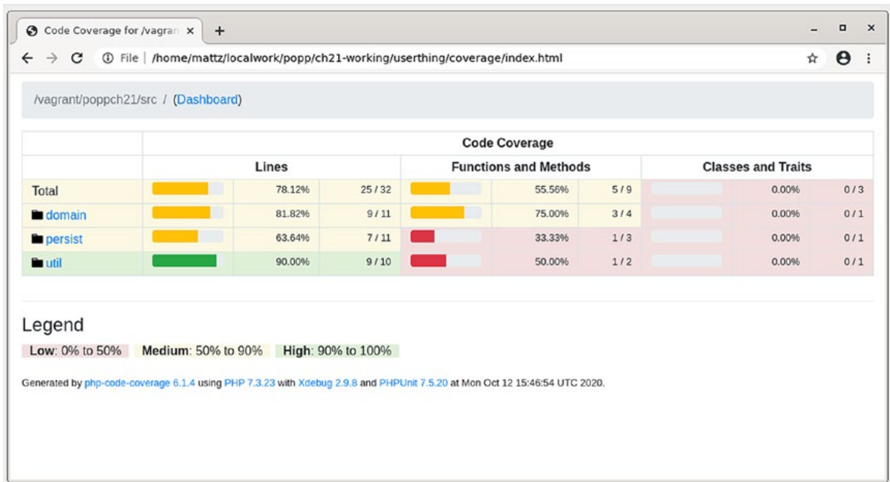


Рис. 21.1. *Отчет о покрытии исходного кода тестами*

Следует, однако, отметить, что достижение полного покрытия исходного кода тестами еще не означает, что система была протестирована в достаточной степени. С другой стороны, всегда полезно знать обо всех недочетах, выявленных в написанных тестах. Как следует из отчета, приведенного на рис. 21.1, нам еще придется потрудиться над этим.

Убедившись в том, что отчет по покрытию кода работает из командной строки, внесем эту функциональную возможность в файл построения, как показано ниже:

```
// Листинг 21.4
<target name="citest" depends="build">
    <mkdir dir="reports/coverage" />
    <coverage-setup database="reports/coverage.db">
        <fileset dir="${src}">
            <include name="**/*.php"/>
        </fileset>
    </coverage-setup>
    <phpunit haltonfailure="true" codecoverage="true"
             bootstrap="${phing.dir}/vendor/autoload.php"
             printsummary="true">
        <formatter type="plain" usefile="false"/>
        <formatter type="xml" outfile="testreport.xml"
                              todir="reports" />
        <formatter type="clover" outfile="cloverreport.xml"
                                 todir="reports" />
        <batchtest>
            <fileset dir="${test}">
                <include name="**/*Test.php"/>
            </fileset>
        </batchtest>
    </phpunit>
    <coverage-report outfile="reports/coverage.xml">
        <report todir="reports/coverage" />
    </coverage-report>
</target>
```

Здесь мы создали новое задание с именем `citest`. В бóльшей части файла построения для этого задания попросту воспроизводится рассмотренное ранее задание `test`.

Сначала создается каталог `reports`, а в нем — подкаталог `coverage`. Для создания параметров конфигурации элемента `coverage` используется задание `coverage-setup`. С помощью атрибута `database` в ней указано место хранения необработанных данных по покрытию исходного кода тестами. А во вложенном элементе разметки `fileset` определены файлы, содержимое которых должно быть подвергнуто анализу по покрытию исходного кода тестами.

В определение задания `phpunit` введены три элемента разметки `formatter`. Элемент разметки `formatter`, относящийся к типу `xml`, предназначен для формирования файла `testreport.xml`, содержащего

результаты тестирования. Элемент разметки `formatter`, относящийся к типу `clover`, также формирует данные о покрытии исходного кода тестами в формате XML. И в самом конце определения задания `citest` приводится в действие задание `coverage-report`. При этом берется готовая информация о покрытии кода, генерируется новый XML-файл и создается отчет в формате HTML.

На заметку Полная документация задания `coverage-report` (под названием `CoverageReportTask`) находится по адресу `http://www.phing.info/guide/chunkhtml/CoverageReportTask.html`.

Стандарты программирования

Стандарты программирования подробно обсуждались в главе 15, “Стандарты PHP”. И хотя приспосабливать свой индивидуальный стиль программирования под общий стандарт — не самая приятная задача, это упрощает организацию коллективного труда над проектом. Именно по этой причине многие команды разработчиков соблюдают общий стандарт кодирования. А поскольку соблюдать его на глаз нелегко, то имеет смысл автоматизировать данный процесс.

С этой целью воспользуемся снова диспетчером зависимостей Composer. На этот раз сконфигурируем его для установки утилиты PHP_CodeSniffer следующим образом:

```
"require-dev": {
    "phpunit/phpunit": "^9",
    "squizlabs/php_codesniffer": "3.*"
}
```

Теперь применим рекомендацию стандарта PSR-12 к нашему исходному коду, выполнив следующую команду:

```
$ vendor/bin/phpcs --standard=PSR12 src/util/Validator.php

FILE: /vagrant/poppch21/src/util/Validator.php
-----------------------------------------------------------------
FOUND 8 ERRORS AFFECTING 2 LINES
-----------------------------------------------------------------
 7|ERROR|[x] Header blocks must be separated by a single blank line
22|ERROR|[ ] Visibility must be declared on method "validateUser"
22|ERROR|[ ] Expected "function abc(...)"; found "function abc (...)"
22|ERROR|[x] Expected 1 space after FUNCTION keyword; 4 found
22|ERROR|[x] Expected 0 spaces after opening parenthesis; 1 found
```

```
22|ERROR|[x] Expected 0 spaces before opening parenthesis; 3 found
22|ERROR|[x] Expected 1 space between comma and argument "$pass";
             0 found
22|ERROR|[x] Opening brace should be on a new line
-----------------------------------------------------------------
PHPCBF CAN FIX THE 6 MARKED SNIFF VIOLATIONS AUTOMATICALLY
------------------------------------------------------------------
```

Очевидно, что исходный код требует небольшой правки! К числу преимуществ, которые дает инструментальное средство автоматического исправления ошибок, относится его беспристрастный характер. Если в команде разработчиков решено установить определенный стиль программирования, то лучше, конечно, предпочесть беспристрастный сценарий лишенному чувства юмора коллеге, делающему то же самое.

Как и следовало ожидать, теперь мы должны добавить задание `CodeSniffer` в свой файл построения:

```
// Листинг 21.5
<target name="sniff" depends="build">
    <exec executable="vendor/bin/phpcs" passthru="true"
          dir="${phing.dir}">
    <arg line="--report-checkstyle=reports/checkstyle.xml
              --standard=PSR12 build/src" />
    </exec>
</target>
```

Хотя Phing предоставляет задание `phpcodesniffer`, оно несовместимо с последними версиями PHP_CodeSniffer. Поэтому я снова использую `exec` для запуска этого инструмента. Я вызываю `phpcs` с флагом `--report-checkstyle`, чтобы создать XML-файл в каталоге `reports`.

Таким образом, в нашем распоряжении имеется немало полезных инструментальных средств, с помощью которых мы можем осуществлять текущий контроль своего проекта. Безусловно, если разработчик будет предоставлен самому себе, он вскоре потеряет всякий интерес к применению этих средств — даже к Phing, столь удобному для построения проекта. На самом деле он может даже вернуться к прежней мысли о выполнении стадии интеграции, прибегнув к соответствующим инструментальным средствам только в том случае, если дело приближается к выпуску. Но ведь к этому моменту их эффективность как средств раннего предупреждения окажется весьма низкой. Следовательно, для эффективного применения описанных выше инструментальных средств потребуется сервер НИ, который будет автоматически все это запускать.

Jenkins — это сервер непрерывной интеграции с открытым исходным кодом, ранее называвшийся "Hudson". Несмотря на то что он написан на языке Java, им совсем не трудно воспользоваться для применения инструментальных средств PHP. Дело в том, что сервер НИ никак не связан со строящимися проектами, а просто выполняет различные команды и выводит получаемые результаты. Сервер Jenkins очень хорошо интегрируется с проектами на PHP, поскольку в нем предусмотрена поддержка подключаемых модулей. Кроме того, существует весьма активное сообщество разработчиков, работающих над расширением основных функциональных возможностей этого сервера.

На заметку Почему, собственно, сервер Jenkins? Дело в том, что им просто пользоваться и его легко расширять. К тому же он имеет солидную репутацию и активное сообщество пользователей. Сервер Jenkins совершенно бесплатный, его исходный код открыт для общего доступа, а подключаемые к нему модули поддерживают интеграцию с PHP и большинство инструментальных средств, которые могут понадобиться для построения и тестирования проекта. Следует, однако, иметь в виду, что в настоящее время имеется немало других серверов НИ. Так, в одном из предыдущих изданий этой книги был описан сервер CruiseControl (`http://cruisecontrol.sourceforge.net/`), который по-прежнему остается неплохим вариантом выбора.

Установка сервера Jenkins

Сервер Jenkins написан на языке Java, и поэтому для его работы требуется установить платформу Java. Конкретная ее установка зависит от используемой системы. На сайте Jenkins по адресу `http://www.jenkins.io/doc/book/installing/` вы найдете подробные инструкции по установке.

Благодаря Vagrant я выполняю установку в системе CentOS 7. Вот как это выглядит:

```
$ sudo wget -O /etc/yum.repos.d/jenkins.repo \
    https://pkg.jenkins.io/redhat-stable/jenkins.repo
$ sudo rpm --import \
    https://pkg.jenkins.io/redhat-stable/jenkins.io.key
$ sudo yum update
$ sudo yum install jenkins java-1.8.0-openjdk-devel
$ sudo systemctl daemon-reload
```

Запуск Jenkins выполняется следующим образом:

```
$ sudo systemctl start jenkins
```

После запуска сервер Jenkins начинает по умолчанию прослушивать порт `8080`. Чтобы проверить, работает ли сервер Jenkins в вашей системе, посетите в веб-браузере страницу `http://`*`адрес_хоста`*`:8080/`. В итоге вы должны увидеть экран установки, приведенный на рис. 21.2.

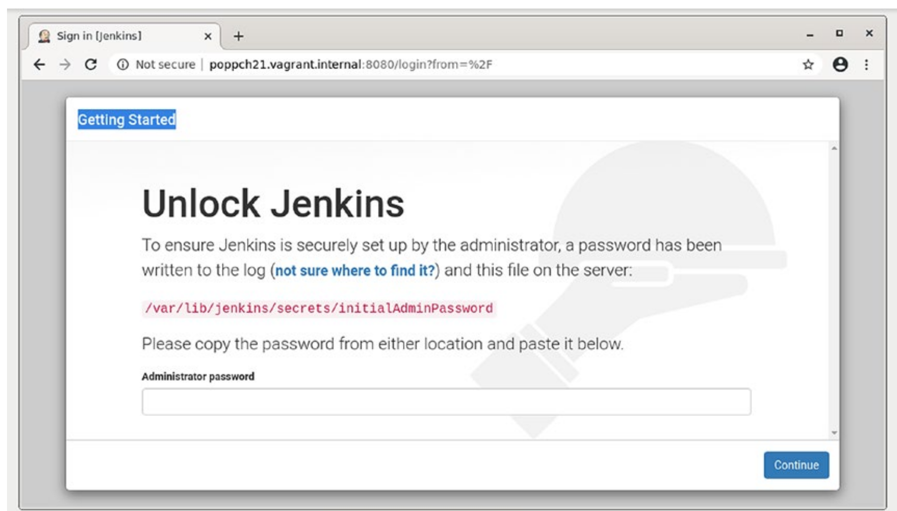


Рис. 21.2. *Экран установки сервера Jenkins*

Инструкции, показанные на рис. 21.2, не требуют особых пояснений. В частности, можно взять пароль из файла `/var/lib/jenkins/secrets/initialAdminPassword` (с использованием `sudo` из-за ограничений прав доступа для чтения файла) и ввести его в указанном поле. А далее предстоит сделать выбор: установить распространенные подключаемые модули или выбрать собственные модули. Чаще всего выбираются распространенные подключаемые модули, поскольку они обеспечивают, среди прочего, поддержку системы Git. Если же требуется более гибкая настройка системы, можно выбрать только нужные подключаемые модули. И прежде чем завершить установку сервера Jenkins, необходимо задать имя пользователя и пароль.

Установка модулей, подключаемых к серверу Jenkins

Сервер Jenkins является полностью настраиваемым программным продуктом. Поэтому для поддержки описанных выше возможностей интеграции с проектами на PHP нам понадобится установить несколько подключаемых модулей. Для этого в окне веб-интерфейса Jenkins щелкните сначала на ссылке Manage Jenkins (Управлять сервером Jenkins), а затем — на ссылке Manage Plugins (Управлять подключаемыми модулями). Далее перейдите на вкладку Available (Доступные модули), и вы увидите длинный список подключаемых к серверу Jenkins модулей, которые можно установить. В столбце Install (Установить) отметьте флажком нужные модули, которые должны быть установлены. Их список приведен в табл. 21.1.

Таблица 21.1. *Некоторые модули, подключаемые к серверу Jenkins*

Подключаемый модуль	Описание
Git	Поддерживает интеграцию с хранилищами Git
jUnit	Поддерживает интеграцию с семейством инструментальных средств xUnit, включая PHPUnit
Phing	Позволяет использовать Phing
Clover PHP	Обеспечивает доступ к XML- и HTML-файлам, генерируемым PHPUnit, и формирует отчеты
HTML Publisher	Средство интеграции отчетов в формате HTML. Используется для вывода PHPDocumentor
Warnings Next Generation	Обеспечивает доступ к XML-файлу, сгенерированному PHPCodeSniffer, и формирует отчет

Список установленных дополнительных модулей приведен на рис. 21.3. После установки перечисленных выше подключаемых модулей почти все готово для построения и настройки рассматриваемого здесь проекта.

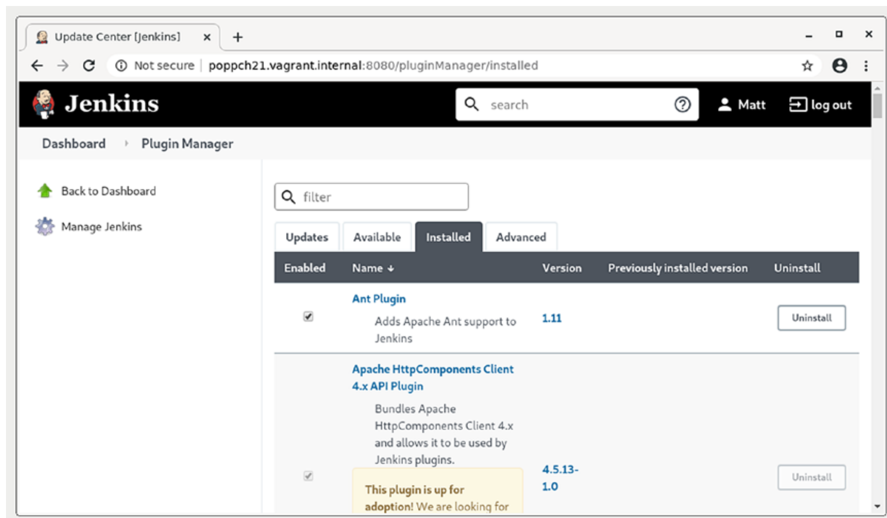


Рис. 21.3. *Экран с установленными модулями, подключаемыми к серверу Jenkins*

Установка открытого ключа доступа к Git

Прежде чем воспользоваться модулем, подключаемым к системе Git, необходимо убедиться, что сервер Jenkins имеет доступ к хранилищу Git. В главе 17, "Контроль версий средствами Git", был описан процесс генерации открытого ключа для доступа к этому удаленному хранилищу. Теперь нам нужно повторить данный процесс в начальном каталоге сервера Jenkins. Но где именно он находится?

Расположение начального каталога можно изменить, но обычно его легко обнаружить в настройках самого сервера Jenkins. Для этого щелкните сначала на ссылке Manage Jenkins, а затем — на ссылке Configure System (Сконфигурировать систему). В открывшемся окне появится путь к начальному каталогу сервера Jenkins. Кроме того, можно просмотреть содержимое файла `/etc/passwd` и выяснить путь к начальному каталогу пользователя `jenkins`. В данном случае это каталог `/usr/local/jenkins`.

Теперь нам нужно сконфигурировать ключи и разместить их в подкаталоге `.ssh`, как показано ниже:

```
$ sudo su jenkins -s /bin/bash
$ cd ~
$ mkdir .ssh
$ chmod 0700 .ssh
$ ssh-keygen
```

Сначала мы переключились на учетную запись пользователя `jenkins`, указав в командной строке путь к оболочке, поскольку по умолчанию доступ к системной оболочке может быть отключен. Далее мы перешли в начальный каталог этого пользователя, создали в нем подкаталог `.ssh` и установили права доступа к нему, запрещающие доступ других пользователей и групп. По команде `ssh-keygen` мы сгенерировали ключи SSH. Когда на экране появилось приглашение ввести пароль, мы просто нажали клавишу <Enter>. В итоге доступ к серверу Jenkins будет осуществляться только по его ключу. Необходимо также убедиться, что созданный файл `.ssh/id_rsa` недоступен для чтения всем остальным пользователям, кроме текущего. Для этого мы выполнили приведенную ниже команду:

```
$ chmod 0600 .ssh/id_rsa
```

Теперь открытый ключ можно получить из файла `.ssh/id_rsa.pub` и ввести его в удаленное хранилище системы Git. Как это сделать, подробно описано в главе 17. Но это еще не все. Необходимо также убедиться, что сервер Git находится в списке разрешенных хостов для связи по протоколу SSH. Чтобы добавить открытый ключ в список и заодно проверить конфигурацию сервера Git, достаточно ввести приведенные ниже команды, не забыв только предварительно войти в систему под именем пользователя `jenkins`:

```
$ cd /tmp
$ git clone git@bitbucket.org:getinstance/userthing.git
```

В итоге будет выдан запрос на согласие добавить сервер Git в список разрешенных хостов для связи по протоколу SSH. Его адрес будет записан в файл `.ssh/known_hosts`, находящийся в начальном каталоге пользователя `jenkins`. Это позволит впоследствии исключить отключение сервера Jenkins при попытке установить соединение с сервером Git.

На заметку Имеются также различные плагины, которые вы можете использовать для управления учетными данными Git, включая *SSH Agent*, *OAuth Credentials* и *Kubernetes Credentials*.

Установка проекта

Перейдите на страницу с панелью управления сервером Jenkins и щелкните на ссылке New Item (Создать новый элемент). На появившемся вновь экране введите имя проекта, `userthing`, как показано на рис. 21.4.

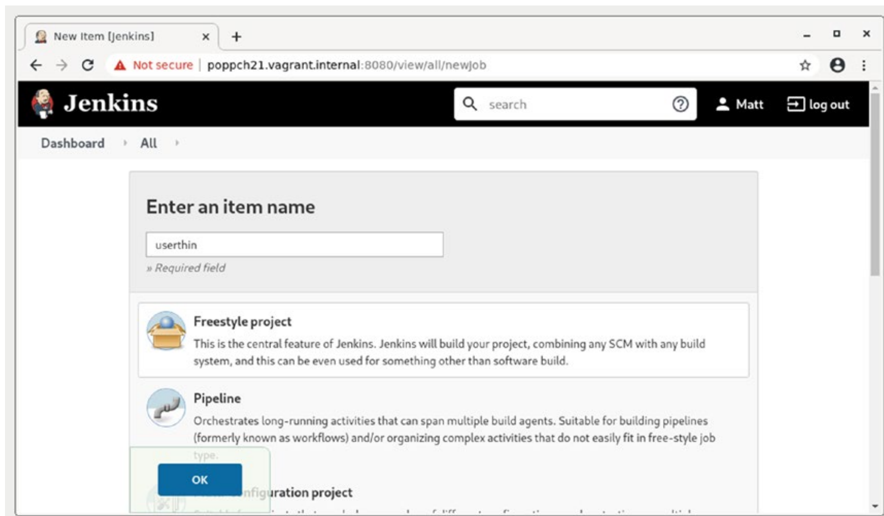


Рис. 21.4. *Экран установки проекта*

В данном случае выбран проект типа Freestyle. Для подтверждения выбора мы щелкнули на кнопке OK. Это приведет нас к экрану конфигурирования проекта. Прежде всего мы должны подключиться к удаленному хранилищу Git. Для этого достаточно выбрать кнопку-переключатель Git в разделе Source Code Manager (Диспетчер исходного кода) и ввести имя хранилища, как показано на рис. 21.5.

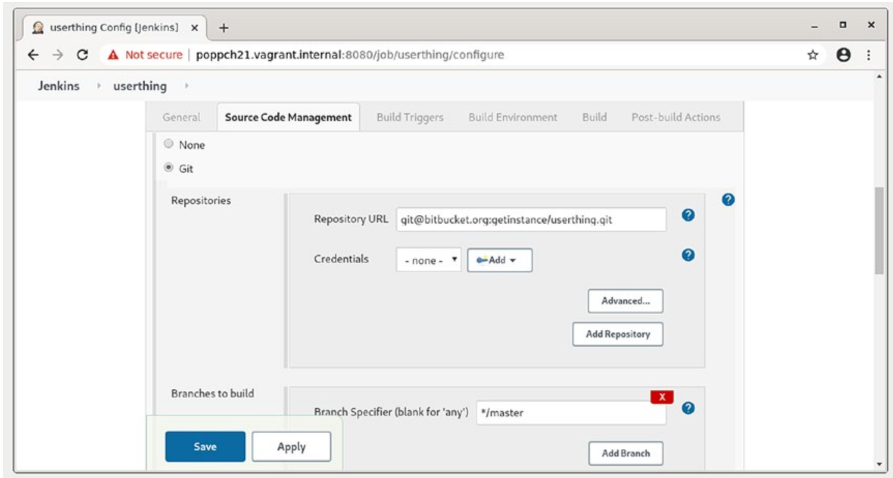


Рис. 21.5. *Настройка параметров хранилища системы контроля версий*

Если все пройдет нормально, мы должны получить доступ к своему исходному коду. В этом можно убедиться, сохранив проект, щелкнув на кнопке Save и выбрав вкладку Build (Построить). Но для того чтобы увидеть сколько-нибудь значимое действие, следует также настроить Phing. Это нетрудно сделать, если инструментальное средство Phing установлено глобально. Но если применяется диспетчер зависимостей Composer, то дело несколько усложняется, поскольку нужно каким-то образом сообщить серверу Jenkins, где искать исполняемый файл Phing. Для этого достаточно выбрать сначала пункт Manage Jenkins из главного меню, а затем — пункт Global Tool Configuration (Конфигурирование глобального инструментального средства).

Поскольку я установил плагин Phing, мы получим область конфигурации данного инструментария. Я щелкаю на Add Phing, чтобы получить доступ к форме. На рис. 21.6 я показываю область конфигурации, которую следует использовать для обращения к локальной версии Phing.

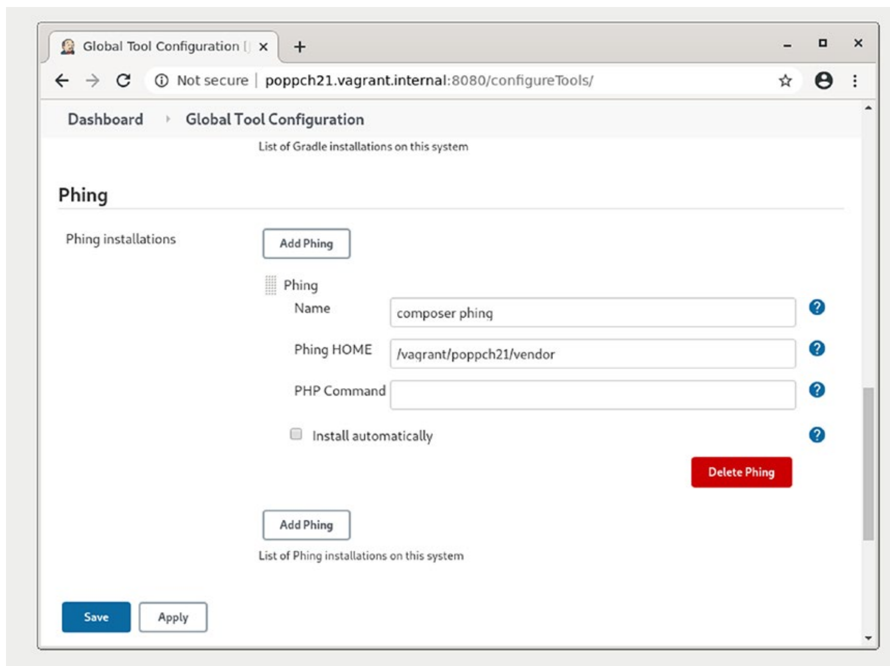


Рис. 21.6. *Указание местоположения Phing*

Я даю конфигурации имя и добавляю в свой проект путь к каталогу поставщика, в котором можно найти Phing.

Убедившись, что Jenkins может найти Phing, выполним настройки для своего проекта. С этой целью вернемся к области конфигурирования проекта `userthing` и меню Configure. Перейдя к разделу Build, выберем два варианта из раскрывающегося списка Add build step (Ввести стадию построения). Прежде всего выберем вариант Execute shell (Выполнить командную оболочку), ведь нам нужно обеспечить установку Composer на сервере Jenkins, иначе не будет установлено ни одно из инструментальных средств, от которых может зависеть рассматриваемый здесь проект. Результаты такого конфигурирования приведены на рис. 21.7.

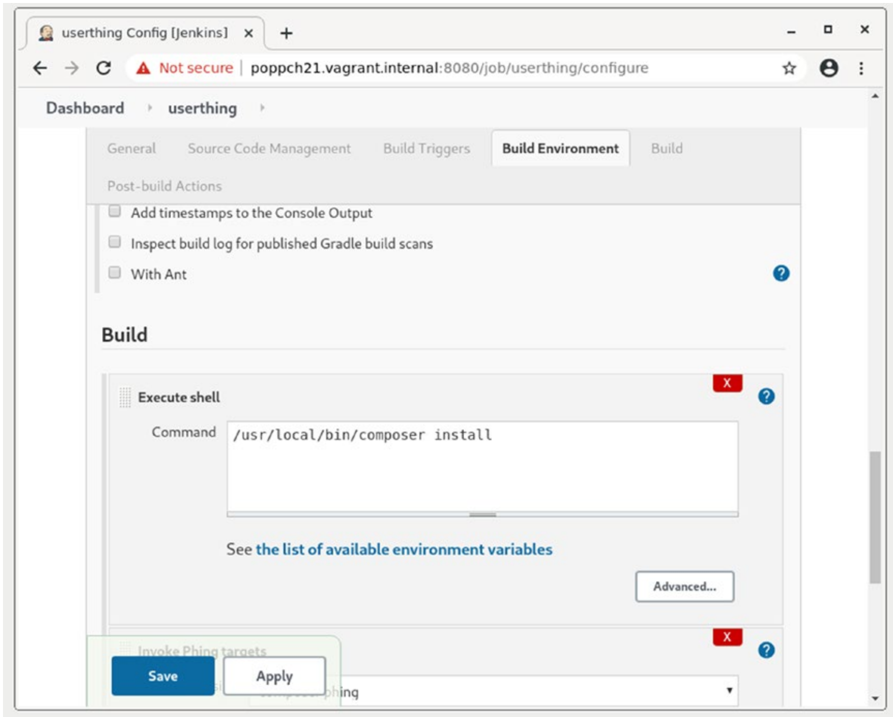


Рис. 21.7. Установка режима выполнения командной оболочки

Далее из раскрывающегося списка Add build step следует выбрать вариант Invoke Phing targets (Вызвать целевые задания Phing). Я выбираю экземпляр Phing, который я настроил ранее (composer phing), из раскрывающегося списка и добавляю мои целевые задания в текстовое поле, как показано на рис. 21.8.

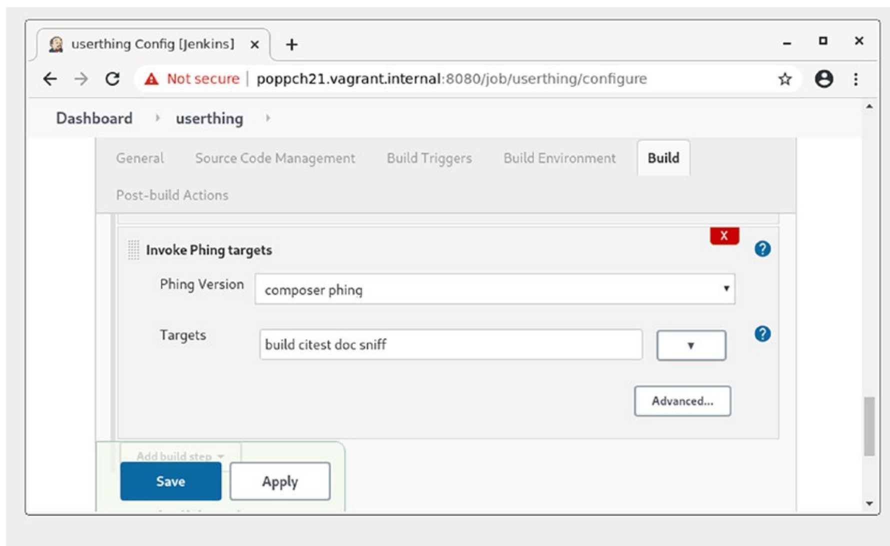


Рис. 21.8. *Конфигурирование Phing*

Первое построение проекта

Чтобы приступить к построению проекта и процессу тестирования, следует сохранить настройки, сделанные на экране конфигурирования, а затем щелкнуть на кнопке Build Now (Построить сейчас). В итоге будет запущен процесс построения и тестирования проекта. И здесь наступает момент истины! В области Build History (Предыстория построения) должна появиться ссылка на построение проекта. Если щелкнуть на ней, появится экран Console Output (Вывод на консоль), на котором должны отобразиться сообщения, подтверждающие, что процесс построения проекта начался и идет так, как и предполагалось (рис. 21.9). Как видите, сервер Jenkins загрузил исходный код проекта `userthing` из удаленного хранилища Git и запустил все целевые задания для построения и тестирования данного проекта.

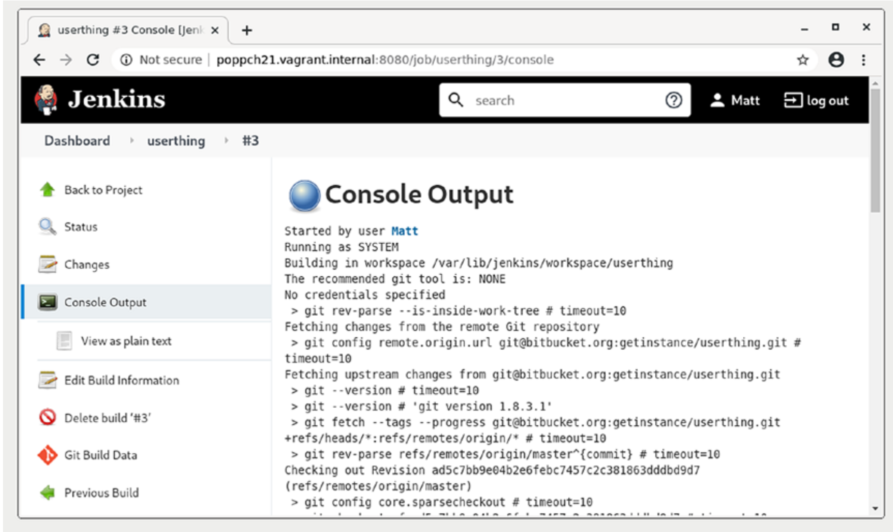


Рис. 21.9. Вывод сообщений на консоль

Настройка отчетов

Благодаря установкам, сделанным в моем файле построения, Phing сохраняет файлы отчетов в каталоге `build/reports`, а документацию — в каталоге `build/docs`. Подключаемые модули для составления отчетов можно настроить из раскрывающегося списка Add post-build action (Ввести действие после построения) на экране конфигурирования проекта. Некоторые параметры их настройки приведены на рис. 21.10.

Чтобы не приводить здесь большое количество копий экрана, все варианты настройки отчетов были сведены в одну таблицу с кратким их описанием. В табл. 21.2 приведены некоторые поля для настройки действий после построения проекта и соответствующие им задания из файла построения Phing.

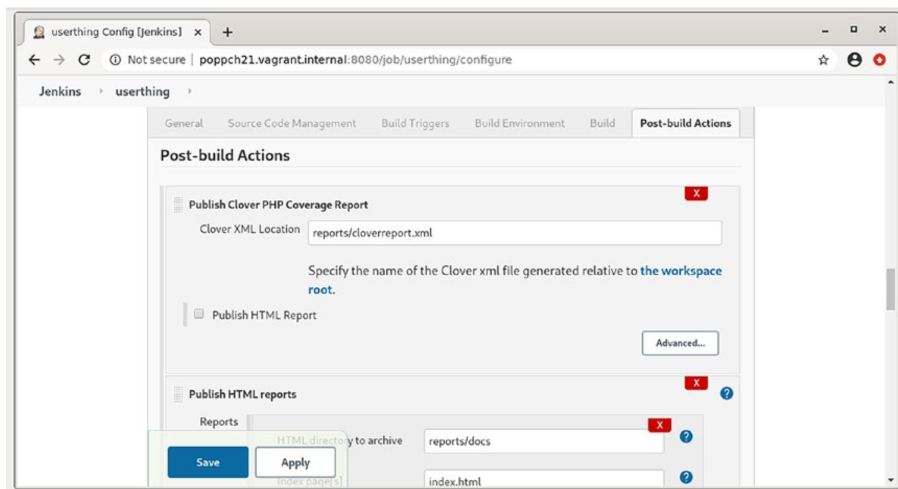


Рис. 21.10. Настройка параметров подключаемых модулей для составления отчетов

Таблица 21.2. *Настройка параметров отчета*

Параметр	Задание Phing	Поле	Значение
Record compiler warning and static analysis results (запись предупреждений компилятора и результатов статического анализа)	`sniff`	Tool (инструменты)	`PHP_CodeSniffer`
		Report File Pattern (шаблон файла отчета)	`reports/checkstyle.xml`
Publish Clover PHP Coverage Report (опубликовать отчет о покрытии кода PHP тестами в формате Clover)	`citest`	Clover XML Location (местоположение файла формата Clover XML)	`reports/cloverreport.xml`
		Clover HTML report directory (каталог отчетов в формате Clover HTML)	`reports/clovercoverage/`

Окончание табл. 21.2

Параметр	Задание Phing	Поле	Значение
Publish HTML reports (опубликовать отчеты в формате HTML)	`doc`	HTML directory to archive (HTML-каталог для архива)	`reports/docs`
		Index page[s] (страницы индексов)	`index.html`
Publish JUnit test result report (опубликовать отчет с результатами тестирования в JUnit)	`citest`	Test report XMLs (XML-файлы с отчетами о тестировании)	`reports/testreport.xml`
E-mail Notification (уведомление по электронной почте)		Recipients (получатели)	`someone@somemail.com`

Все параметры конфигурации, представленные в табл. 21.2, соответствуют тем параметрам, которые мы указали в файле построения проекта, за исключением последнего. В поле E-mail Notification можно указать список разработчиков, которым по электронной почте будут приходить уведомления о неудачных исходах построения проекта.

Прежде чем я смогу проверить покрытие, я должен создать переменную среды с именем `XDEBUG_MODE`, со значением `coverage`. Я могу сделать это, сначала перейдя в раздел Manage Jenkins, а затем — на экран Configure System (Конфигурация системы). Как вы можете увидеть на рис. 21.11, я могу установить переменные среды в разделе `Global Properties` (Глобальные свойства).

После всех этих настроек я могу вернуться к экрану проекта и запустить новую сборку. На рис. 21.12 показан новый результат.

Через некоторое время на экране управления проектом появятся графики, демонстрирующие тенденции, проявляющиеся в ходе тестирования на производительность, покрытие исходного кода тестами и анализ стиля программирования. Там же предоставляются ссылки на последнюю документацию по интерфейсу API, подробные результаты тестирования и полные сведения о покрытии исходного кода тестами.

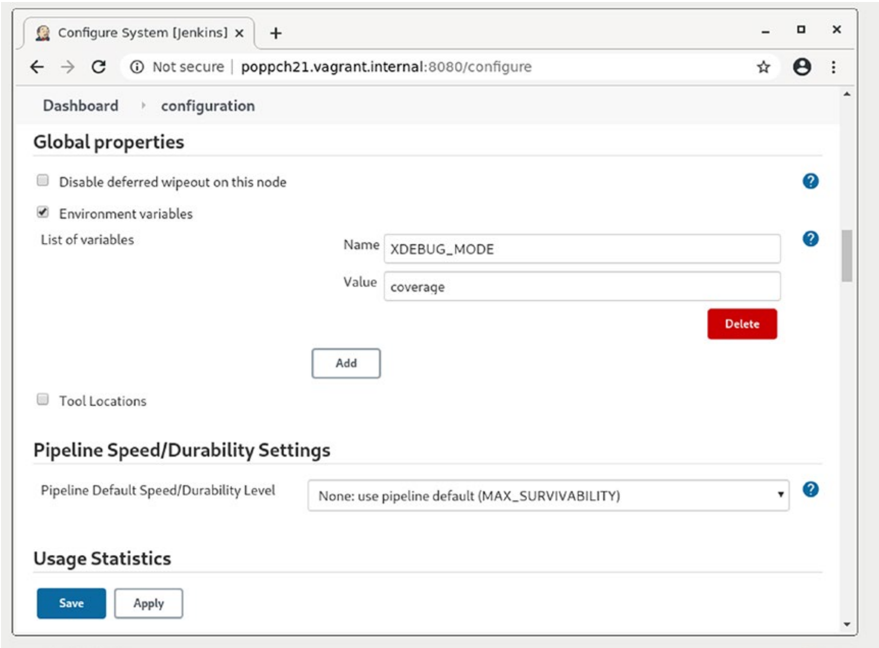


Рис. 21.11. Настройка переменной среды

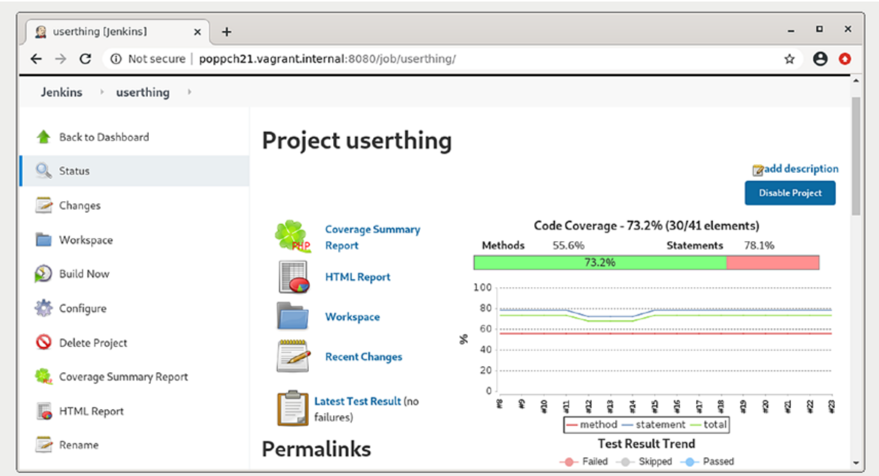


Рис. 21.12. Экран проекта, демонстрирующий проявляющиеся тенденции

Автоматический запуск процессов построения проектов

Все приведенное выше обилие информации практически не представляет никакой пользы, если обязанность по запуску процессов построения проектов вручную возложена на одного из забывчивых членов вашей команды разработчиков. Именно по этой причине на сервере Jenkins предусмотрены механизмы автоматического запуска процессов построения проектов.

Сервер Jenkins можно настроить на автоматический запуск процессов построения проектов или отслеживание изменений, внесенных в хранилище системы контроля версий, через определенные промежутки времени. Эти промежутки задаются в формате утилиты `cron` операционной системы Unix, что позволяет составить желательное расписание подобных запусков, хотя оно и может выглядеть несколько запутанным. Впрочем, для тех, кому не требуется составлять сложные расписания запусков, на сервере Jenkins предусмотрено несколько простых псевдонимов, включая `@hourly`, `@midnight`, `@daily`, `@weekly` и `@monthly`. В качестве примера на рис. 21.13 показано, как настроить запуск процесса построения проекта один раз в сутки или каждый раз после того, как будут внесены изменения в хранилище, которое будет опрашиваться каждый час на предмет подобных изменений.

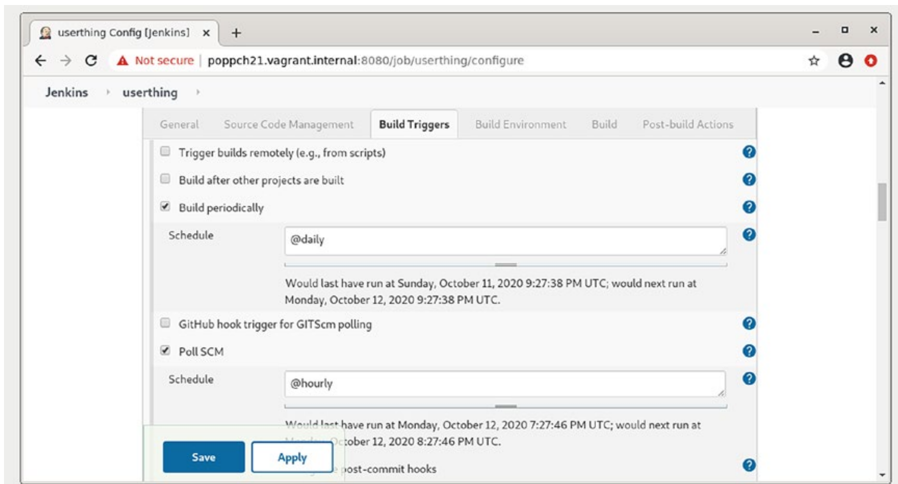


Рис. 21.13. Составление расписания автоматического запуска процесса построения проекта

Сбои тестов

До сих пор все складывалось удачно, даже несмотря на то, что проект userthing пока еще не заслуживает того, чтобы быть как-то отмеченным за совместимость кода. Но ведь тесты достигают успеха лишь в том случае, если они не проходят. Поэтому нам придется внести в исходный код данного проекта какой-нибудь изъян, чтобы убедиться, что сервер Jenkins о нем сообщит.

В качестве примера ниже приведен фрагмент кода из класса Validate в пространстве имен userthing\util:

```
// Листинг 21.6
public function validateUser(string $mail, string $pass): bool
{
    // Сделать тест сбойным:
    // return false;
    $user = $this->store->getUser($mail);
    if (is_null($user))
    {
        return false;
    }
    $testpass = $user->getPass();
    if ($testpass == $pass)
    {
        return true;
    }
    $this->store->notifyPasswordFailure($mail);
    return false;
}
```

Видите, где я могу саботировать этот метод? Если я раскомментирую вторую строку в метод, validateUser() всегда будет возвращать false.

Вот тест, который должен поймать этот сбой. Он находится в файле test/util/ValidatorTest.php:

```
// Листинг 21.7
public function testValidateCorrectPass(): void
{
    $this->assertTrue(
        $this->validator->validateUser("bob@example.com", "12345"),
        "Ожидается успешная проверка"
    );
}
```

После внесения изменений в код нам остается лишь зафиксировать их в хранилище и ждать! Очень скоро на странице состояния рассматриваемого здесь проекта `userthing` появится желтый значок, означающий, что что-то пошло не так. Если щелкнуть на ссылке построения, то можно выяснить подробности произошедшего. На рис. 21.14 показан экран с отчетом об ошибках.

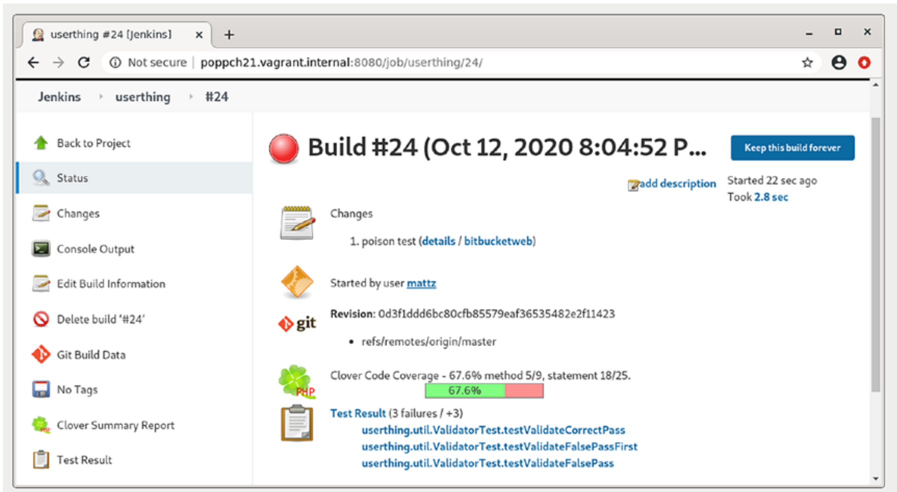


Рис. 21.14. *Сбой построения проекта*

Помните, что если вам нужна дополнительная информация о том, в чем именно сборка пошла не так, всегда можно перейти по ссылке `Console Output` на экране построения. Зачастую там имеется больше полезной информации, чем на сводном экране построения.

Резюме

В этой главе мы рассмотрели немало инструментальных средств, которые уже упоминались в предыдущих главах, и собрали их в единое целое с Jenkins. С этой целью мы подготовили небольшой проект для непрерывной интеграции, применили в нем целый ряд инструментальных средств, включая PHPUnit (и для тестирования, и для покрытия исходного кода тестами), PHP_CodeSniffer, PHP_CodeBrowser, phpDocumentor и Git. После этого мы установили сервер Jenkins и показали, как вводить проекты в систему непрерывной интеграции. Далее мы запустили систему непрерыв-

ной интеграции, продемонстрировали, как расширить функциональные возможности сервера Jenkins для рассылки уведомлений об ошибках по электронной почте, и с его помощью выполнили тестирование процессов построения и установки проекта.

ГЛАВА 22

Объекты, проектные шаблоны и практика

Цель этой книги — помочь читателю в создании успешного проекта на PHP. Чтобы достичь ее, мы рассмотрели обширный ряд тем, начиная с основ объектно-ориентированного программирования и принципов проектирования с помощью шаблонов и заканчивая инструментальными средствами и нормами практики проектирования.

В этой главе мы подытожим некоторые основные темы и вопросы, которые были рассмотрены в данной книге.

- *PHP и объекты.* Как в языке PHP продолжает усиливаться поддержка объектно-ориентированного программирования и как выгодно пользоваться его средствами.
- *Объекты и проектирование.* Краткое описание некоторых принципов объектно-ориентированного подхода к проектированию.
- *Проектные шаблоны.* Их преимущества.
- *Принципы организации проектных шаблонов.* Краткий перечень главных принципов объектно-ориентированного проектирования, которые лежат в основе многих шаблонов.
- *Рабочие инструментальные средства.* Напоминание о средствах, описанных в предыдущих главах, а также рассмотрение тех из них, которые еще не были описаны.

Объекты

Как упоминалось в главе 2, в течение долгого времени объекты были чем-то вроде дополнения к языку PHP. Поддержка объектов в версии PHP 3 была, мягко говоря, незначительной, причем объекты представляли собой всего лишь ассоциативные массивы в причудливом одеянии. И хотя в версии PHP 4, к радости приверженцев объектно-ориентированного программирования, ситуация существенно изменилась к лучшему, все еще

остались серьезные трудности. Одна из них состояла в том, что объекты присваивались и передавались по значению.

С появлением версии PHP 5 объекты, наконец-то, вышли из тени. Но программировать на PHP можно было по-прежнему, даже не объявляя класс, хотя язык был оптимизирован для объектно-ориентированного программирования. Развитие объектно-ориентированных средств было завершено в версии PHP 7 внедрением долгожданных объявлений скалярного и возвращаемого типов. Вероятно, по причинам обратной совместимости некоторые из распространенных каркасов по-прежнему остаются, по существу, процедурными по своей природе (в первую очередь WordPress). Однако в общем и целом большинство новых проектов на PHP оказываются объектно-ориентированными.

В главах 3–5 мы подробно рассмотрели поддержку объектно-ориентированного программирования в языке PHP. В версии PHP 5 появились следующие новые возможности: рефлексия, исключения, закрытые и защищенные методы и свойства, метод `__toString()`, модификатор доступа `static`, абстрактные классы и методы, завершенные методы и свойства, интерфейсы, итераторы, методы перехвата, объявления типов данных, модификатор доступа `const`, передача объектов по ссылке, методы `__clone()`, `__construct()`, позднее статическое связывание, пространства имен и анонимные классы. Даже этот неполный список явно свидетельствует, в какой степени будущее языка PHP связано с объектно-ориентированным программированием.

В версии PHP 5 и Zend Engine 2 объектно-ориентированное программирование было поставлено во главу угла каждого проекта на языке PHP, открывая тем самым этот язык для новых разработчиков и новые возможности — для уже имеющихся его приверженцев. В главе 6 мы рассмотрели преимущества, которые могут дать объекты при разработке проектов. А поскольку объекты и проектирование относятся к одной из главных тем данной книги, стоит еще раз сделать более основательно некоторые выводы.

Выбор

Нет никакого закона, который требует, чтобы проект разрабатывался только с помощью классов и объектов. Грамотно спроектированный объектно-ориентированный код обеспечивает понятный интерфейс, доступный из любого клиентского кода: как процедурного, так и объектно-ориентированного. Даже если вам не интересно создавать объекты (что маловеро-

ятно, раз вы все еще читаете эту книгу), скорее всего, вы все равно будете ими пользоваться, хотя бы как клиент пакетов Composer.

Инкапсуляция и делегирование

Объекты выполняют свои обязанности как бы за закрытыми дверями. Они обеспечивают интерфейс, через который передаются запросы и принимаются результаты их обработки. За этим интерфейсом скрываются любые данные, которые не должны быть раскрыты, а также конкретные подробности реализации.

Это придает разные формы объектно-ориентированным и процедурным проектам. В объектно-ориентированном проекте, как правило, получается создать на удивление простой и компактный контроллер, в котором используется небольшое количество экземпляров объектов. Они генерируют другие объекты и вызывают их методы, передавая данные одного уровня другому.

С другой стороны, для процедурного проекта характерна бо́льшая степень вмешательства. Логика управления в большей степени внедряется в реализацию, ссылаясь на переменные, оценивая возвращаемые значения и направляясь по различным путям действия в зависимости от обстоятельств.

Развязка

Развязать (разъединить) компоненты означает удалить взаимную их зависимость таким образом, чтобы изменение в одном компоненте не влекло за собой необходимость изменения других компонентов. Грамотно спроектированные объекты самодостаточны, им не нужно ссылаться на нечто внешнее, чтобы восстановить детали, полученные в предыдущем вызове.

Поддерживая внутреннее представление состояния, объекты снижают потребность в глобальных переменных, которые являются общеизвестной причиной возникновения тесной связи. С помощью глобальной переменной одна часть системы связывается с другой. Если в компоненте (функции, классе или блоке кода) применяется глобальная переменная, то существует риск, что другой компонент случайно воспользуется тем же именем переменной и испортит ее значение, используемое первым компонентом. Возможно, третий компонент зависит от значения переменной, установленной первым компонентом. И если изменить режим работы первого компонента, то третий компонент может перестать нормально функцио-

нировать. Цель объектно-ориентированного проекта — уменьшить такую взаимозависимость, сделав каждый компонент настолько самодостаточным, насколько это возможно.

Еще одной причиной тесной связи служит дублирование кода. Тесная связь обнаруживается, когда реализация алгоритма повторяется в разных частях проекта. А что произойдет в том случае, если придется изменить сам алгоритм? Очевидно, что при этом нужно не забыть изменить его исходный код везде, где он встречается. И если забыть это сделать, то в проектируемой системе возникнут проблемы.

Распространенной причиной дублирования кода служат параллельные условные выражения. Если проект должен действовать некоторым способом в одной ситуации (например, при выполнении в среде Linux) и иным — в другой (при выполнении под управлением Windows), то условные операторы `if/else` будут часто обнаруживаться в разных частях проектируемой системы. Если же добавить в проект новую ситуацию вместе со стратегией ее обработки (например, в системе Mac OS X), то придется обновить все условные операторы.

В объектно-ориентированном программировании предусмотрена методика разрешения данной проблемы. В частности, условные операторы можно заменить *полиморфизмом*. Полиморфизм означает прозрачное использование различных подклассов в зависимости от ситуации. В каждом подклассе поддерживается такой же интерфейс, как и в общем суперклассе, и поэтому в клиентском коде вообще неизвестно (да это и неважно), какой именно реализацией он пользуется.

Это совсем не значит, что код, содержащий условные операторы, должен быть полностью исключен из объектно-ориентированных систем; его просто нужно свести к минимуму и централизовать. Какой-то тип условного кода необходимо использовать, чтобы определить, какие конкретно подтипы должны быть предоставлены клиентам. Но эта проверка обычно выполняется один раз и в одном месте, что сводит тесную связь к минимуму.

Повторное использование кода

Инкапсуляция способствует развязке, которая, в свою очередь, способствует повторному использованию кода. Самодостаточные компоненты, устанавливающие связь с более обширными системами только через их общедоступные интерфейсы, можно, как правило, перемещать из одной системы в другую, чтобы использовать их без изменений.

На самом деле такие компоненты встречаются реже, чем вы думаете. Даже идеально ортогональный код может быть зависим от проекта. Например, при создании ряда классов для управления содержимым определенного веб-сайта имеет смысл потратить некоторое время на этапе планирования, чтобы определить возможности, характерные для клиентов, а также те возможности, которые послужат основанием для будущих проектов по управлению содержимым.

Относительно повторного использования кода можно дать еще один совет: централизовать классы, которые можно использовать в нескольких проектах. Иными словами, повторно используемый класс не следует копировать в новый проект. Это может стать причиной возникновения тесной связи на макроуровне, поскольку, внося неизбежные изменения в класс из одного проекта, можно забыть сделать это в другом проекте. Намного лучше работать с общими классами из центрального хранилища, общедоступного для разных проектов.

Эстетика

Я не собираюсь никого убеждать, но считаю, что объектно-ориентированный код вызывает эстетическое удовлетворение. Беспорядочность реализации скрыта за аккуратными интерфейсами, а следовательно, объект кажется клиенту простым.

Я люблю аккуратность и изящество полиморфизма, потому что API позволяет выполнять операции над разными объектами, которые, тем не менее, работают взаимозависимо и прозрачно, а значит, эти объекты можно складывать, как детские кубики, или вставлять один в другой, как детали детского конструктора.

Конечно, найдутся и такие люди, которые будут утверждать совершенно противоположное. Объектно-ориентированный характер кода может проявиться в бурном развитии классов, развязанных друг от друга до такой степени, что их взаимосвязи могут стать причиной серьезных затруднений, а это само по себе может привести к недоброкачественному коду. И зачастую возникает искушение строить одни фабрики, производящие другие фабрики до тех пор, пока исходный код не станет напоминать зеркальный зал. Иногда имеет смысл сделать самое простое, что только может работать, а затем реорганизовать код, чтобы повысить его изящество для тестирования и гибкость — для выполнения. Определять конкретное проектное решение должна предметная область, а не перечень норм передовой практики.

На заметку Строгое применение так называемых норм передовой практики также вызывает трудности управления проектом. Всякий раз, когда применение методики или процесса начинает напоминать ритуал, выполняемый автоматически и негибко, имеет смысл уделить время исследованию причин, стоящих за текущим подходом, потому что иначе это может привести к переходу от области применения инструментальных средств к карго-культу.

Стоит также упомянуть, что изящное решение не всегда оказывается самым лучшим и эффективным. Иногда возникает искушение использовать полномасштабное объектно-ориентированное решение там, где достаточно небольшого сценария или нескольких системных вызовов.

Проектные шаблоны

Недавно один разработчик, программирующий на Java, поступая на работу в компанию, с которой я сотрудничал, извинялся за то, что использовал шаблоны всего пару лет. Почему-то считается, что проектные шаблоны — это революционное достижение технического прогресса. Именно этим объясняется тот энтузиазм, который они вызвали. На самом деле этот опытный разработчик наверняка пользовался проектными шаблонами намного дольше, чем он думал.

С помощью проектных шаблонов описываются общие задачи и проверенные решения, присваиваются названия, кодируются и систематизируются нормы передовой практики решения конкретных задач. Они не являются элементами изобретения или голой теории. Проектный шаблон не будет эффективным, если он не описывает нормы практики, которые уже распространены на момент подготовки к разработке.

Напомним, что понятие языка шаблонов происходит из области строительства и архитектуры. Люди в течение тысяч лет строили внутренние дворики и арки, прежде чем шаблоны были предложены в качестве способа описания решений отдельных задач рационального использования пространства и функционирования.

Учитывая сказанное, легко понять, почему проектные шаблоны вызывают эмоции, сравнимые с религиозными и политическими дискуссиями. Ярые приверженцы проектных шаблонов бродят по коридорам с фанатичным блеском в глазах и экземпляром книги "Банды четырех" под мышкой. Они обращаются к непосвященным и зачитывают имена шаблонов как

символ веры. Неудивительно, что многие критики считают, что проектные шаблоны — это просто надувательство.

В таких языках, как Perl и PHP, применение шаблонов также вызывает споры из-за устойчивой их связи с объектно-ориентированным программированием. В контексте, в котором объекты являются обоснованным проектным решением, а не самоцелью, ориентация на проектные шаблоны говорит только о предпочтениях. И это вовсе не означает, что одни шаблоны должны порождать другие шаблоны, а одни объекты — другие объекты.

Преимущества проектных шаблонов

Проектные шаблоны были представлены в главе 7, “Назначение и применение проектных шаблонов”. Напомним еще раз преимущества, которые они дают.

Испытанные и проверенные

Прежде всего, как отмечалось ранее, шаблоны — это испытанные решения конкретных задач. Но проводить аналогию между шаблонами и рецептами опасно: рецептам можно следовать слепо, в то время как шаблоны по своему характеру являются “полуфабрикатами” (по Мартину Фаулеру) и требуют более вдумчивого обращения с ними. Тем не менее у рецептов и шаблонов имеется одна важная общая особенность: перед описанием они были тщательно проверены и испытаны.

Одни шаблоны предполагают другие шаблоны

У шаблонов имеются совпадающие друг с другом пазы и выступы. Это означает, что некоторые шаблоны вставляются один в другой, как собираемые с приятным щелчком детали. Решение задачи с помощью проектного шаблона неизбежно будет иметь последствия, которые могут привести к условиям, предполагающим применение дополнительных проектных шаблонов. Безусловно, выбирая связанные вместе шаблоны, очень важно решать реальные задачи, а не просто нагромождать изящные, но бесполезные башни взаимосвязанного кода. Это очень заманчиво — создать программный эквивалент архитектурной причуды.

Общий словарь

Проектные шаблоны — это средство выработки общего словаря для описания задач и их решений. Имена очень важны — они заменяют описания и позволяют очень быстро осветить множество тем. Но, конечно, имена делают также неясным смысл для тех, кто еще не разобрался в используемом словаре, и это одна из причин, по которым шаблоны иногда очень сильно раздражают.

Шаблоны способствуют проектированию

Как упоминалось ранее, при правильном применении шаблоны способствуют качественному проектированию. Но здесь необходимо сделать следующее важное замечание: шаблоны не являются панацеей.

Шаблоны и принципы проектирования

По своему характеру проектные шаблоны связаны с качественным проектом. При правильном применении они помогут создать слабосвязанный и гибкий код. Но критики шаблонов в чем-то правы, когда говорят, что шаблонами часто злоупотребляют, особенно начинающие. Реализации шаблонов формируют привлекательные и изящные структуры, и поэтому очень легко забыть, что хороший проект всегда должен отвечать своей цели. Не следует забывать, что шаблоны существуют для решения конкретных задач.

Впервые работая с шаблонами, я обнаружил, что использовал шаблон Abstract Factory по всему коду. Мне нужно было формировать объекты, и шаблон Abstract Factory помогал мне в этом.

Но на самом деле я просто ленился подумать, и поэтому делал лишнюю работу. Совокупности объектов, которые мне нужно было создавать, в действительности были связаны, но у них не было альтернативных реализаций. Классический шаблон Abstract Factory идеально подходит в том случае, когда требуется создавать альтернативные совокупности объектов в соответствии с конкретной ситуацией. Чтобы привести шаблон Abstract Factory в действие, необходимо создавать фабричные классы для каждого типа объектов, а также класс для обслуживания фабричного класса. Даже описать этот процесс не так-то просто.

Код стал бы намного более аккуратным, если бы я создал базовый фабричный класс, проводя рефакторинг только для реализации шаблона Abstract Factory лишь в том случае, когда требуется генерировать параллельную совокупность объектов.

Применение шаблонов еще не гарантирует качество проекта. При разработке стоит помнить о двух формулировках одного и того же принципа: KISS ("Keep it simple, stupid" — "Будь проще, дурачок!") и "Do the simplest thing that works" ("Делать самое простое, что только может работать"). В экстремальном программировании используется похожее правило — YAGNI ("You aren't going to need it" — "Вам это не понадобится"). Это означает, что реализовывать функциональное средство, если в нем нет крайней необходимости, не следует.

Сделав столь категоричные замечания, вернемся к тому, что вызывает больше энтузиазма. Как утверждалось в главе 9, "Генерация объектов", в проектных шаблонах обычно воплощается ряд принципов, которые можно обобщить и применить ко всему коду.

Предпочитайте композицию наследованию

Отношения наследования очень эффективны. Наследование служит для поддержки переключения классов во время выполнения (т.е. полиморфизма) и поэтому положено в основу многих проектных шаблонов и методов, рассматриваемых в данной книге. Но, полагаясь в проекте исключительно на наследование, можно создать негибкие структуры, подверженные дублированию.

Избегайте тесных связей

О данном принципе мы уже упоминали в этой главе, но повторим еще раз ради полноты изложения. Вам следует считаться с тем фактом, что изменение в одном компоненте может потребовать изменений в других частях проекта. Но такую необходимость можно свести к минимуму, избегая дублирования, типичной причиной которого являются параллельные условные операторы, а также злоупотребления глобальными переменными или синглтонами. Можно также свести к минимуму применение конкретных подклассов, в которых абстрактные типы служат для поддержки полиморфизма. Последнее приводит нас к следующему принципу.

Программируйте интерфейсы, а не реализации

Проектируйте программные компоненты с четко определенными общедоступными интерфейсами, которые делают прозрачными области ответственности каждого компонента. Если вы определяете интерфейс в абстрактном суперклассе и у вас имеются клиентские классы, в которых

требуется работать с этим абстрактным типом, вы тем самым отделяете клиентов от конкретных реализаций.

Учитывая сказанное, вспомните о принципе YAGNI. Если сначала вам нужна только одна реализация типа данных, то нет причины немедленно создавать абстрактный суперкласс. С тем же успехом можно определить понятный интерфейс в одном конкретном классе. Как только вы обнаружите, что ваша единственная реализация пытается выполнить больше одной операции одновременно, вы сможете переопределить конкретный класс в качестве абстрактного родителя двух подклассов. Клиентский код не станет ничуть "разумнее", поскольку он продолжит работать с одним типом. Классическим признаком того, что реализацию лучше разделить, а получающиеся в итоге классы — скрыть за абстрактным родительским классом, служит появление в такой реализации условных операторов.

Инкапсулируйте концепции, которые изменяются

Если вы обнаружите, что увязли в подклассах, выясните причину этого, а затем создайте для всех этих подклассов собственный тип данных. Особенно если причина состоит в том, чтобы достичь цели, которая связана с основным назначением этого типа данных.

Рассмотрим в качестве примера тип `UpdatableThing`. На его основании можно создать подтипы `FtpUpdatableThing`, `HttpUpdatableThing` и `FileSystemUpdatableThing`. Но ответственность этого типа состоит в том, чтобы быть *обновляемым*, а для этой цели механизмы сохранения и извлечения вторичны. В рассматриваемом примере классы, подобные `Ftp`, `Http` и `FileSystem` (то, что изменяется), относятся к собственному типу, который можно назвать механизмом обновления `UpdateMechanism`. Для разных реализаций у класса `UpdateMechanism` могут быть отдельные подклассы. В таком случае можно будет ввести столько механизмов обновления, сколько потребуется, не затрагивая тип `UpdatableThing`, который будет сосредоточен на выполнении своей основной обязанности. Кстати, обратите внимание, что `UpdateMechanism` можно также назвать `UpdateStrategy`. По сути, я описал реализацию проектного шаблона Strategy (Стратегия). Подробнее о нем речь идет в главе 11, "Выполнение задач и представление результатов".

Обратите также внимание на то, что мы заменили статическую структуру, создаваемую на стадии компиляции, динамической структурой, создаваемой на стадии выполнения. Она словно случайно возвращает нас к первому принципу — "Предпочитайте композицию наследованию".

Практика

Вопросы, которые мы освещаем в этой части книги (и с которыми познакомились в главе 14), часто игнорируют как авторы книг, так и программисты. Но в своей карьере программиста я обнаружил, что эти инструментальные средства и методики по крайней мере так же необходимы для достижения успеха, как и грамотное проектирование. Но такие вопросы документирования и автоматического построения не настолько очевидны по своему характеру, как шаблон Composite.

На заметку Вспомним о прелестях шаблона Composite, реализующего простое дерево наследования, в котором объекты можно объединить во время выполнения, чтобы сформировать структуры, которые также являются деревьями, но на несколько порядков более гибкими и сложными. Несколько объектов совместно используют один общий интерфейс, который является их представлением во внешнем мире. Взаимодействие простого и сложного, единичного и множественного заставит учащенно биться ваше сердце. Ведь это не просто проектирование программного обеспечения, а поэзия!

Но даже если такие вопросы, как документация и построение, тестирование и контроль версий, более прозаичны, чем проектные шаблоны, то они ничуть не менее важны. На практике даже самый фантастический проект не будет долговечным, если несколько разработчиков не смогут легко вводить в него свой код или у них не будет ясного понимания уже имеющегося исходного кода. Большие системы трудно поддерживать и расширять без автоматического тестирования. Без инструментальных средств построения никто не будет утруждать себя развертыванием вашего проекта. Но поскольку пользовательская база проектов на PHP все время расширяется, их разработчики обязаны обеспечить качество и простоту их развертывания.

Проект существует в двух ипостасях: структуры кода и функциональных возможностей, а также набора файлов и каталогов, которые образуют основание для сотрудничества, набор источников и мест назначения и предмет для преобразования. В этом смысле проект является системой как с внешней точки зрения, так и с внутренней, с точки зрения исходного кода. Механизмы построения, тестирования, документации и контроля версий требуют такого же внимания к деталям, как и в коде, в котором эти механизмы поддерживаются. Уделяйте метасистеме такое же пристальное внимание, как и самой системе.

Тестирование

Хотя тестирование — это часть каркаса, применяемого к проекту извне, оно глубоко интегрировано в сам код. Полная отвязка от тестирования невозможна и даже нежелательна, и поэтому среды тестирования весьма эффективны для отслеживания результатов изменений. Изменение типа, возвращаемого методом, может повлиять на клиентский код в другом месте, что вызовет появление ошибок через недели или месяцы после внесения изменения. Тестовые каркасы дают неплохие шансы обнаружить ошибки подобного рода (чем лучше тесты, тем выше эти шансы).

Тестирование — это также инструментальное средство для улучшения объектно-ориентированного проекта. Изначальное (или хотя бы параллельное с разработкой) тестирование поможет сосредоточиться на интерфейсе класса и тщательно обдумать обязанности и поведение каждого метода. Инструментальное средство PHPUnit, которое применяется для тестирования кода PHP, рассматривалось в главе 18.

Стандарты

По своему характеру я бунтарь и терпеть не могу, когда мне говорят, что я обязан делать. Такие слова, как *соответствие стандарту*, мгновенно вызывают во мне противодействие. Но, как ни странно, стандарты побуждают к нововведениям, поскольку способствуют взаимозаменяемости. Появлению и распространению Интернета отчасти способствовал тот факт, что в его основание были положены открытые стандарты. Веб-сайты могут связываться друг с другом, а веб-серверы — повторно использоваться в любом домене, поскольку сетевые протоколы хорошо известны и строго соблюдаются. Решение использовать изолированное хранилище может оказаться более совершенным, чем повсеместно принятый и соблюдаемый стандарт, но что если такое хранилище сгорит? И что делать, если его новый владелец решит взимать плату за доступ, а некоторые пользователи решат, что расположенное рядом хранилище лучше? Рекомендации стандартов PSR подробно рассматривались в главе 15, в которой особое внимание было уделено стандартам автозагрузки, которые стали намного более ясными с тех пор, как разработчики приложений на PHP стали включать в них классы. В этой главе были также рассмотрены рекомендации стандарта PSR-12 по стилю программирования. Программисты обязаны использовать в исходном коде фигурные скобки и указывать списки аргументов, но

если они согласятся подчиняться ряду общих правил, то написанный ими код станет более удобочитаемым и согласованным, что позволит применять инструментальные средства для проверки и форматирования исходных файлов. В этом духе мне пришлось переформатировать исходный код всех примеров в настоящем издании книги, чтобы они соответствовали рекомендациям стандарта PSR-12.

Контроль версий

Сотрудничество — дело непростое. Откровенно говоря, люди неловки, а программисты — тем более. Распределив роли и задачи в команде, самое последнее, чем вам захочется заниматься, — это разбираться в несоответствиях в самом исходном коде. Как пояснялось в главе 17, Git (и аналогичные инструментальные средства наподобие CVS и Subversion) позволяют объединять работу нескольких программистов в одном хранилище. В тех случаях, когда конфликты (и несоответствия версий) неизбежны, система Git сообщает об этом и указывает, в каком месте исходного кода нужно разрешить возникший конфликт.

Даже если вы программируете в одиночку, система контроля версий вам все равно потребуется. Git допускает ветвление, позволяющее поддерживать в отдельной ветке старую версию программы и одновременно разрабатывать следующую ее версию, перенося код с исправленными ошибками из устойчивой версии в ветку разработки.

Git регистрирует также все фиксации кода, которые когда-либо были сделаны в проекте. Это означает, что вы можете вернуться к прежней версии, найдя нужный вариант программы по дате или метке. Поверьте, в один прекрасный день это спасет ваш проект.

Автоматическое построение

Контроль версий без автоматического построения имеет ограниченную область применения. Для развертывания проекта любой сложности необходимо приложить дополнительные усилия. В частности, отдельные файлы требуется разместить в разных каталогах системы, файлы конфигурации — преобразовать, чтобы они содержали нужные параметры для текущей платформы и базы данных, а таблицы базы данных — создать или преобразовать. В этой книге были описаны два инструментальных средства для установки. Первым из них является диспетчер зависимостей

Composer (см. главу 16), идеально подходящий для автономных пакетов или небольших приложений, а вторым — инструментальное средство Phing (см. главу 19), обладающее достаточной эффективностью и гибкостью для автоматизации процесса установки самых крупных и запутанных проектов.

Автоматическое построение превращает процесс развертывания из утомительной работы в выполнение одной-двух команд из командной строки. Вы легко и непринужденно вызываете тестовую программу и вывод документации из инструментального средства построения. Даже если вам безразличны интересы команды разработчиков, подумайте о том, как вам будут благодарны пользователи, когда обнаружат, что им не нужно больше тратить полдня на копирование файлов и изменение файлов конфигурации при каждом выпуске новой версии вашего проекта.

Непрерывная интеграция

Одной только возможности тестирования и построения проекта явно недостаточно. Это нужно делать *постоянно*! Причем важность этого процесса возрастает по мере расширения и усложнения проекта и поддержки в нем нескольких веток разработки. Вам придется постоянно выполнять построение и тестирование устойчивой ветки, в которую были внесены небольшие исправления замеченных ошибок. Кроме нее, могут существовать несколько экспериментальных веток разработки, а также ваша основная ветка. Пытаясь все делать вручную, даже с помощью соответствующих инструментальных средств построения и тестирования, вы можете увязнуть в этом процессе надолго, а ведь вам еще нужно выполнять свою основную работу! Разумеется, все программисты ненавидят эту работу, так что построение и тестирование проекта неизбежно будет сведено на нет.

В главе 21 мы рассмотрели непрерывную интеграцию как набор методик и средств, предназначенных для максимальной автоматизации процесса построения и тестирования проекта.

Что упущено из виду

Некоторые категории инструментальных средств, описание которых пропущено в данной книге из-за недостатка времени и места, чрезвычайно полезны для любого проекта. Как правило, в распоряжении разработчика имеется целый ряд удобных инструментальных средств, хотя применение

некоторых из них вам, возможно, придется обсудить с другими разработчиками; и вам придется оптимизировать избранное вами средство, прежде чем сделать окончательный выбор.

Если в вашем проекте принимает участие несколько разработчиков или даже просто активный клиент, вам может потребоваться инструментальное средство для отслеживания программных ошибок и выполняемых задач. Аналогично контролю версий, средство отслеживания программных ошибок относится к числу тех, которые повышают производительность труда настолько, что, опробовав его однажды, вам трудно будет себе представить, как это вы не пользовались им раньше. Средства отслеживания программных ошибок сообщают о недостатках в проекте, но они не менее часто применяются и для описания необходимых функциональных средств и распределения их реализации среди членов команды разработчиков.

В любой момент можно получить описание неисправленных ошибок, сузив область поиска по продукту, автору ошибки, номеру версии или приоритету. У каждой программной ошибки имеется своя страница, на которой можно обсудить текущие проблемы. Записи в обсуждении и изменении статуса ошибки можно скопировать и отослать по электронной почте членам команды, что позволяет следить за ходом событий, не заходя постоянно на веб-сайт средства отслеживания ошибок.

Имеется немало подобных инструментальных средств. К их числу относится пользующееся заслуженной популярностью средство Bugzilla (`http://www.bugzilla.org`). Оно свободно доступно, имеет открытый исходный код и предоставляет все возможности, которые только могут понадобиться разработчикам. А поскольку это загружаемое средство, его можно запустить на своем сервере. И хотя оно по-прежнему очень похоже на веб-сайт версии Web 1.0, от этого оно ничуть не становится хуже. Если же вы не желаете размещать на своем сервере отдельное средство отслеживания программных ошибок и предпочитаете пользоваться более удобными интерфейсами (и обладаете толстым кошельком), обратите внимание на решение Jira, предлагаемое компанией Atlassian (`https://www.atlassian.com/software/jira`). Для отслеживания высокоуровневых задач и планирования проектов рекомендуется средство Trello (`http://www.trello.com`), особенно тем, кто интересуется системой Kanban.

Как правило, средство отслеживания программных ошибок является лишь одним из инструментальных средств для организации коллективного труда, с помощью которых требуется обмениваться информацией в рамках проекта. За определенную плату можно приобрести такое интегрирован-

ное решение, как, например, Basecamp (`https://basecamp.com/`) или инструментальные средства компании Atlassian (`https://www.atlassian.com/`). Кроме того, можно составить экосистему из различных инструментальных средств. Так, чтобы упростить общение в команде разработчиков, может потребоваться механизм интерактивной переписки и обмена сообщениями. Едва ли не самым широко применяемым для этой цели инструментальным средством в настоящее время является Slack (`https://www.slack.com`) — веб-ориентированная многокомнатная среда интерактивной переписки. Разработчики старой школы вроде меня могут по-прежнему пользоваться системой IRC (Internet Relay Chat — ретранслируемые через Интернет разговоры) и будут правы, поскольку она мало чем уступает среде Slack, кроме ориентации последней на браузеры, простоты применения и интеграции с другими уже встроенными службами. Среда Slack свободно доступна в базовом варианте.

Кроме того, разработчики старой школы могут пользоваться списками рассылки для работы над своими проектами. Для этой цели лучше всего подойдет свободно доступное, относительно легко устанавливаемое и гибко конфигурируемое программное обеспечение Mailman (`http://www.gnu.org/software/mailman/`). А для совместного редактирования текстовых документов и электронных таблиц, вероятно, самым простым решением окажется служба Google Docs (`https://docs.google.com/`).

Ваш исходный код не настолько ясен, как вам может показаться на первый взгляд. Чужому разработчику, получившему доступ к вашей кодовой базе, поначалу будет нелегко в ней разобраться. И даже вы, автор исходного кода, можете со временем забыть, как он организован. Поэтому для составления встраиваемой документации рекомендуется обратить внимание на инструментальное средство phpDocumentor (`https://www.phpdoc.org/`), позволяющее документировать исходный код по мере его написания, а также автоматически формировать вывод с гиперссылками. Результат, выводимый phpDocumentor, особенно удобен в объектно-ориентированном контексте, поскольку он дает пользователю возможность переходить от одного класса к другому, выбирая гиперссылки щелчком кнопки мыши. А поскольку классы зачастую содержатся в отдельных файлах, для непосредственного чтения исходного кода могут потребоваться переходы из одного исходного файла в другой.

Несмотря на то что внутренняя документация очень важна, к проектам также прилагается немало печатных материалов. В них включаются инструкции по применению, советы по будущим направлениям развития,

активы клиентов, протоколы собраний и уведомления о корпоративных вечеринках. В течение срока действия проекта такие материалы постоянно меняются, и нужен механизм, позволяющий членам команды разрабатывать их совместно.

Вики-страницы (в переводе с гавайского *вики* означает "очень быстро") — идеальное средство для создания совместно используемых систем документов с гиперссылками. Создавать и редактировать страницы можно щелчком кнопки мышьи, а гиперссылки автоматически формируются для слов, соответствующих именам страниц. Вики-страницы — настолько важное и очевидное инструментальное средство, что создается впечатление, будто у вас уже была такая идея, но вы просто не взялись за ее осуществление. Имеются самые разные вики-страницы. У меня есть положительный опыт работы с вики-страницами типа PhpWiki (`http://phpwiki.sourceforge.net`) и DokuWiki (https://www.splitbrain.org/projects/dokuwiki).

Однако я все чаще склоняюсь к тому, что для документации (и для написания программ) ст*о*ит вернуться к простым текстовым документам и системам управления версиями. Для форматирования я использую Markdown, простой облегченный язык разметки. В результате текст легко читается до рендеринга, а после получается красиво отформатированный документ (хотя, как и в случае с любым рендерингом, вы полагаетесь на милость соответствующего программного обеспечения). Лучшая отправная точка для Markdown — `https://commonmark.org/`. После многих лет борьбы с Word и текстовыми редакторами, совместимыми с Word, я очень признателен за то, что Apress позволил мне использовать Markdown для этого издания книги!

На заметку Хотя я и не пропустил этот инструмент (см. главу 17), стоит упомянуть о том, что переход на простой текстовый формат позволил нам при написании этой книги широко использовать Git.

Технический рецензент Пол Трегоинг (Paul Tregoing) хотел бы увидеть в разделе о непрерывной интеграции Docker (`https://docs.docker.com/`), но временн*ы*е рамки этому помешали. Задания сборки Jenkins, выполняемые внутри контейнеров Docker, означают, что у вас есть полная свобода настраивать среду сборки без ограничений, налагаемых системой, на которой размещен Jenkins. Вы можете выполнять построение, используя разные версии пакетов или даже другой дистрибутив Linux.

Резюме

В этой главе мы подвели некоторые итоги, повторив основные темы, которые освещались в данной книге. Мы не затрагивали здесь конкретные вопросы, касающиеся отдельных проектных шаблонов или функций объектов, потому что цель данной главы — обобщить материал книги.

Чтобы охватить все вопросы, никогда не хватает ни места, ни времени. Но я надеюсь, что данная книга послужит еще одним аргументом в пользу того, что язык PHP активно развивается. В настоящее время это один из самых распространенных языков программирования в мире, и, вероятнее всего, он останется излюбленным языком непрофессиональных программистов. А многие начинающие программировать на PHP с удовольствием обнаружат, каких заметных результатов можно достичь с помощью совсем небольшого кода. В то же время следует отметить, что все больше и больше профессиональных команд разработчиков создают крупные и сложные системы на PHP. Такие проекты заслуживают более серьезного подхода. Благодаря своему уровню расширения язык PHP всегда был гибким и разносторонним, предоставляя доступ к многочисленным приложениям и библиотекам. Благодаря поддержке объектно-ориентированного программирования в нем предоставляется доступ к разнообразным наборам инструментов. Как только вы начнете мыслить категориями объектов, вы сможете сразу же воспользоваться опытом других программистов, добытым ценой больших усилий. Вы сможете также находить и внедрять проектные шаблоны, разработанные не только для PHP, но и для Smalltalk, C++, C# или Java. Наша обязанность — решать сложные задачи, тщательно выполняя проектирование и применяя передовые нормы практики программирования. Будущее за многократно используемым кодом!

ПРИЛОЖЕНИЕ А

Дополнительные источники информации

Литература

1. Alexander Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford, UK: Oxford University Press, 1977.

   (Александер, К., Исикава, С., Силверстайн, М. *Язык шаблонов. Города. Здания. Строительство* : Пер. с англ. — Изд-во студии Артемия Лебедева, 2014.)

2. Alur Deepak, John Crupi and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Englewood Cliffs, NJ: Prentice Hall PTR, 2001.

   (Алур, Д., Крупи, Дж., Малкс, Д. *Образцы J2EE. Лучшие решения и стратегии проектирования* : Пер. с англ. — Изд-во "Лори", 2013.)

3. Beck Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley Professional, 1999.

   (Бек, К. *Экстремальное программирование* : Пер. с англ. — Изд-во "Питер", 2002.)

4. Chacon Scott. *Pro Git*. New York, NY: Apress, 2009.

5. Fogel Karl and Moshe Bar. *Open Source Development with CVS, Third Edition*. Scottsdale, AZ: Paraglyph Press, 2003.

6. Fowler Martin and Kendall Scott. *UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language*. Reading, MA: Addison-Wesley Professional, 1999.

   (Фаулер, М. *UML. Основы. Краткое руководство по стандартному языку объектного моделирования* : Пер. с англ. — Изд-во "Символ-Плюс", 2011.)

7. Fowler Martin, Kent Beck, John Brant, William Opdyke and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley Professional, 1999.

   (Фаулер, М., Бек, К., Брант, Дж., Опдайк, У., Робертс, Д. *Рефакторинг: улучшение проекта существующего кода* : Пер. с англ. — Изд-во "Диалектика", 2017.)

8. Fowler Martin. *Patterns of Enterprise Application Architecture*. Reading, MA: Addison-Wesley Professional, 2002.

   (Фаулер, М. Шаблоны корпоративных приложений : Пер. с англ. — И.Д. "Вильямс", 2009.)

9. Gamma Erich, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Professional, 1995.

   (Гамма, Э., Хелм, Р., Джонсон, Р., Влиссидес, Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* : Пер. с англ. — Изд-во "Питер", 2007.)

10. Hunt Andrew and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA: Addison-Wesley Professional, 2000.

    (Хант, Э., Томас, Д. *Программист-прагматик. Путь от подмастерья к мастеру* : Пер. с англ. — Изд-во "Лори", 2012.)

11. Kerievsky Joshua. *Refactoring to Patterns*. Reading, MA: Addison-Wesley, Professional 2004.

    (Кериевски, Д. *Рефакторинг с использованием шаблонов* : Пер. с англ. — И.Д. "Вильямс", 2006.)

12. Nock Clifton. *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Reading, MA: Addison-Wesley Professional, 2003.

13. Shalloway Alan and James R Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Reading, MA: Addison Wesley Professional, 2001.

14. Stelting Stephen and Olav Maasen. *Applied Java Patterns*. Palo Alto, CA: Sun Microsystems Press, 2002.

Статьи

1. Beck Kent and Erich Gamma. "Test Infected: Programmers Love Writing Tests", `http://junit.sourceforge.net/doc/testinfected/testing.htm`.

2. Collins-Sussman, Ben, Brian W. Fitzpatrick, C. Michael Pilato. "Version Control with Subversion", `http://svnbook.red-bean.com/`.

3. Lerdorf Rasmus. "PHP/FI Brief History", `http://www.php.net//manual/phpfi2.php#history`.

4. Suraski Zeev. "The Object-Oriented Evolution of PHP", `http://www.devx.com/webdev/Article/10007/0/page/1`.

5. Wikipedia. "Law of Triviality.", `https://en.wikipedia.org/wiki/Law_of_triviality` (`https://ru.wikipedia.org/wiki/Закон_тривиальности`).

Сайты

Ansible	www.ansible.com
Basecamp	https://basecamp.com/
Bitbucket	https://bitbucket.org
Bugzilla	www.bugzilla.org
CVS	www.nongnu.org/cvs/
Chef	www.chef.io/chef/
ChromeDriver	https://sites.google.com/a/chromium.org/chromedriver
CommonMark	https://commonmark.org/
Composer	https://getcomposer.org/download/
CruiseControl	http://cruisecontrol.sourceforge.net/
Dang It Git	https://dangitgit.com/en
Docker	https://docs.docker.com/
DokuWiki	www.dokuwiki.org/dokuwiki
Foswiki	https://foswiki.org/
GNU	www.gnu.org/
Git	https://git-scm.com/
Git Flow Cheat Sheet	https://danielkummer.github.io/git-flow-cheatsheet/
Git For Windows	https://gitforwindows.org/
GitHub	https://github.com/
GitLab	https://about.gitlab.com/
Google Code	https://code.google.com
Google Docs	https://docs.google.com/
Java	www.java.com
Jenkins	https://jenkins-ci.org/
Jira	www.atlassian.com/software/jira

Magic Methods in PHP	www.php.net/manual/en/language.oop5.magic.php
Mailman	www.gnu.org/software/mailman/
Martin Fowler	www.martinfowler.com/
Memcached	https://memcached.org/
Mercurial	www.mercurial-scm.org/
Openpear	https://openpear.org/
PCOV	https://pecl.php.net/package/pcov
PEAR	https://pear.php.net
PEAR2	https://pear2.php.net/
PECL	https://pecl.php.net/
PHP	www.php.net
PHP Memcached support	www.php.net/memcache
PHPDocumentor	www.phpdoc.org/
PHPUnit	https://phpunit.de
PHP_CodeSniffer	https://github.com/squizlabs/PHP_CodeSniffer
PSR	www.php-fig.org/psr/
Packagist	https://packagist.org
Phing	www.phing.info
PhpWiki	https://phpwiki.sourceforge.net
Pimple	https://pimple.sensiolabs.org/
Portland Pattern Repository's Wiki (Ward Cunningham)	www.c2.com/cgi/wiki
Pro Git	https://git-scm.com/book/en/v2
Puppet	https://puppet.com
QDB	www.bash.org
RapidSVN	https://rapidsvn.org/
SPL	www.php.net/spl

Selenium	www.selenium.dev/
Semantic Versioning	https://semver.org
Slack	www.slack.com
Subversion	https://subversion.apache.org/
Symfony Dependency Injection	https://symfony.com/doc/current/components/dependency_injection/introduction.html
Trello	www.trello.com
Vagrant	www.vagrantup.com/downloads.html
Vagrant Box Search	https://app.vagrantup.com/boxes/search
VirtualBox	www.virtualbox.org/wiki/Downloads
WordPress standards	https://make.wordpress.org/core/handbook/bestpractices/coding-standards/php/
Xdebug	https://xdebug.org/
Zend	www.zend.com

ПРИЛОЖЕНИЕ Б

Простой синтаксический анализатор

Шаблон Interpreter, рассмотренный в главе 11, “Выполнение задач и представление результатов”, не описывает сам процесс синтаксического анализа. Интерпретатор без синтаксического анализатора неполноценен, если, конечно, не принудить пользователей писать специальный код для вызова интерпретатора PHP! Имеется целый ряд качественных синтаксических анализаторов сторонних производителей, которыми можно пользоваться вместе с шаблоном Interpreter. И это, пожалуй, самое лучшее решение в реальных проектах. В этом же приложении представлен простой объектно-ориентированный синтаксический анализатор, предназначенный для встраивания в интерпретатор языка MarkLogic, рассмотренного в главе 11. Следует, однако, иметь в виду, что приведенные далее примеры служат лишь для проверки самой концепции, и поэтому их нельзя использовать в реальных проектах без доработки.

На заметку Интерфейс и основные структуры этого анализатора взяты из книги Стивена Мецкера (Steven Metsker) *Building Parsers with Java* (Addison-Wesley Professional, 2001). Но поскольку я сильно его упростил, все ошибки следует отсылать в мой адрес. Стивен любезно разрешил мне воспользоваться его первоначальной концепцией синтаксического анализатора по своему усмотрению.

Сканер

Прежде чем приступить к синтаксическому анализу любой языковой конструкции, следует разобрать ее по словам и служебным символам, называемым *токенами*. Для определения токенов в приведенном ниже классе `Scanner` используется ряд регулярных выражений. Полученный результат представляется в виде удобного стека, которым мы будем пользоваться далее:

```
// Листинг Б.1
class Scanner
{
    // Типы токенов
    public const WORD = 1;
    public const QUOTE = 2;
    public const APOS = 3;
    public const WHITESPACE = 6;
    public const EOL = 8;
    public const CHAR = 9;
    public const EOF = 0;
    public const SOF = -1;
    protected int $line_no = 1;
    protected int $char_no = 0;
    protected ? string $token = null;
    protected int $token_type = -1;
    // Доступ к исходным данным предоставляется в
    // классе Reader. Получаемые в итоге данные
    // сохраняются в предоставляемом контексте
    public function __construct(private Reader $r,
                                private Context $context)
    {
    }
    public function getContext(): Context
    {
        return $this->context;
    }
    // Пропуск всех пробельных символов
    public function eatWhiteSpace(): int
    {
        $ret = 0;
        if (
            $this->token_type != self::WHITESPACE &&
            $this->token_type != self::EOL
        )
        {
            return $ret;
        }
        while (
            $this->nextToken() == self::WHITESPACE ||
            $this->token_type == self::EOL
        )
        {
            $ret++;
        }
        return $ret;
    }
```

```
// Возвращает строковое представление текущего токена
// или токена, указанного в аргументе $int
public function getTypeString(int $int = -1): ? string
{
    if ($int < 0)
    {
        $int = $this->tokenType();
    }

    if ($int < 0)
    {
        return null;
    }

    $resolve = [
        self::WORD => 'WORD',
        self::QUOTE => 'QUOTE',
        self::APOS => 'APOS',
        self::WHITESPACE => 'WHITESPACE',
        self::EOL => 'EOL',
        self::CHAR => 'CHAR',
        self::EOF => 'EOF'
    ];
    return $resolve[$int];
}
// Текущий тип токена, представленный целым числом
public function tokenType(): int
{
    return $this->token_type;
}
// Содержимое текущего токена
public function token(): ? string
{
    return $this->token;
}
// true, если текущий токен — слово
public function isWord(): bool
{
    return ($this->token_type == self::WORD);
}
// true, если текущий токен — символ кавычек
public function isQuote(): bool
{
    return ($this->token_type == self::APOS ||
            $this->token_type == self::QUOTE);
}
// Номер текущей строки и исходном тексте
```

```
    public function lineNo(): int
    {
        return $this->line_no;
    }
    // Номер текущего символа в исходном тексте
    public function charNo(): int
    {
        return $this->char_no;
    }
    // Клонирует данный объект
    public function __clone(): void
    {
        $this->r = clone($this->r);
    }
    // Переход к следующему токену в исходном файле.
    // Устанавливает текущий токен и отслеживает номер строки
    // и номер символа
    public function nextToken(): int
    {
        $this->token = null;
        $type = -1;

        while (! is_bool($char = $this->getChar()))
        {
            if ($this->isEolChar($char))
            {
                $this->token = $this->manageEolChars($char);
                $this->line_no++;
                $this->char_no = 0;
                return ($this->token_type = self::EOL);
            }

            elseif($this->isWordChar($char))
            {
                $this->token = $this->eatWordChars($char);
                $type = self::WORD;
            }
            elseif($this->isSpaceChar($char))
            {
                $this->token = $char;
                $type = self::WHITESPACE;
            }
            elseif($char == "'")
            {
                $this->token = $char;
                $type = self::APOS;
            }
```

```
        elseif($char == '"')
        {
            $this->token = $char;
            $type = self::QUOTE;
        }
        else
        {
            $type = self::CHAR;
            $this->token = $char;
        }

        $this->char_no += strlen($this->token());
        return ($this->token_type = $type);
    }

    return ($this->token_type = self::EOF);
}
// Возвращает массив, содержащий тип и содержимое
// следующего токена
public function peekToken(): array
{
    $state = $this->getState();
    $type = $this->nextToken();
    $token = $this->token();
    $this->setState($state);
    return [$type, $token];
}
// Получает объект типа ScannerState, содержащий
// текущую позицию сканера в исходном файле
// и сведения о текущем токене
public function getState(): ScannerState
{
    $state = new ScannerState();
    $state->line_no = $this->line_no;
    $state->char_no = $this->char_no;
    $state->token = $this->token;
    $state->token_type = $this->token_type;
    $state->r = clone($this->r);
    $state->context = clone($this->context);
    return $state;
}
// Использует объект ScannerState для восстановления
// состояния сканера
public function setState(ScannerState $state): void
{
    $this->line_no = $state->line_no;
    $this->char_no = $state->char_no;
```

```
        $this->token = $state->token;
        $this->token_type = $state->token_type;
        $this->r = $state->r;
        $this->context = $state->context;
    }
    // Возвращает следующий символ из исходного файла.
    // Если достигнут конец файла, то возвращается
    // логическое значение
    private function getChar(): string | bool
    {
        return $this->r->getChar();
    }
    // Возвращает все символы, составляющие слово
    private function eatWordChars(string $char): string
    {
        $val = $char;

        while ($this->isWordChar($char = $this->getChar()))
        {
            $val .= $char;
        }

        if ($char)
        {
            $this->pushBackChar();
        }

        return $val;
    }
    // Перемещение в исходном тексте на один символ назад
    private function pushBackChar(): void
    {
        $this->r->pushBackChar();
    }
    // Аргумент является символом слова
    private function isWordChar($char): bool
    {
        if (is_bool($char))
        {
            return false;
        }

        return (preg_match("/[A-Za-z0-9_\-]/", $char) === 1);
    }
    // Аргумент является пробельным символом
    private function isSpaceChar($char): bool
    {
```

```
        return (preg_match("/\t| /", $char) === 1);
    }
    // Аргумент является символом конца строки
    private function isEolChar($char): bool
    {
        $check = preg_match("/\n|\r/", $char);
        return ($check === 1);
    }
    // Поглощает символы \n, \r или \r\n
    private function manageEolChars(string $char): string
    {
        if ($char == "\r")
        {
            $next_char = $this->getChar();

            if ($next_char == "\n")
            {
                return "{$char}{$next_char}";
            }
            else
            {
                $this->pushBackChar();
            }
        }

        return $char;
    }
    public function getPos(): int
    {
        return $this->r->getPos();
    }
}
```

```
// Листинг Б.2
class ScannerState
{
    public int $line_no;
    public int $char_no;
    public ? string $token;
    public int $token_type;
    public Context $context;
    public Reader $r;
}
```

Сначала мы определяем типы всех токенов с помощью констант. Мы собираемся обрабатывать текстовые символы, слова, пробелы и кавычки. Для проверки каждого из этих типов токенов созданы специальные методы: `isWordChar()`, `isSpaceChar()` и т.п. Ядром класса `Scanner` служит метод `nextToken()`, в котором из текущей строки выделяется следующий токен и определяется его тип. Все полученные в итоге сведения сохраняются объектом типа `Scanner` в объекте типа `Context`, который используется объектами синтаксического анализатора для обмена данными по мере обработки исходного текста.

Обратите внимание на второй класс, `ScannerState`. Сканер реализован в классе `Scanner` таким образом, чтобы объекты синтаксического анализатора могли сохранять, изменять и восстанавливать свое состояние, если что-нибудь пойдет не так. Метод `getState()` возвращает объект типа `ScannerState`, заполненный данными, а в методе `setState()` предоставляемый объект типа `ScannerState` используется для восстановления состояния, если в этом возникает потребность. Ниже приведено определение класса `Context`:

```
// Листинг Б.3
class Context
{
    public array $resultstack = [];
    public function pushResult($mixed): void
    {
        array_push($this->resultstack, $mixed);
    }
    public function popResult(): mixed
    {
        return array_pop($this->resultstack);
    }
    public function resultCount(): int
    {
        return count($this->resultstack);
    }
    public function peekResult(): mixed
    {
        if (empty($this->resultstack))
        {
            throw new Exception("empty resultstack");
        }
        return $this->resultstack[count($this->resultstack)-1];
    }
}
```

Как видите, это обычная реализация стека, похожая на доску объявлений, с которой удобно работать классам синтаксического анализатора. Этот класс выполняет те же самые обязанности, что и класс контекста в шаблоне Interpreter, но это не один и тот же класс!

Обратите внимание на то, что класс `Scanner` не обрабатывает файлы и символьные строки самостоятельно. Для этого ему требуется объект типа `Reader`. Это позволит нам легко изменить источник обрабатываемых данных. Ниже приведены определение интерфейса `Reader` и его реализация в классе `StringReader`:

```
// Листинг Б.4
interface Reader
{
    public function getChar(): string | bool;
    public function getPos(): int;
    public function pushBackChar(): void;
}

// Листинг Б.5
class StringReader implements Reader
{
    private int $pos;
    private int $len;
    public function __construct(private string $in)
    {
        $this->pos = 0;
        $this->len = strlen($in);
    }
    public function getChar(): string | bool
    {
        if ($this->pos >= $this->len)
        {
            return false;
        }
        $char = substr($this->in, $this->pos, 1);
        $this->pos++;
        return $char;
    }
    public function getPos(): int
    {
        return $this->pos;
    }
    public function pushBackChar(): void
    {
        $this->pos--;
    }
```

```
    public function string(): string
    {
        return $this->in;
    }
}
```

В этом классе символы просто по очереди извлекаются из предоставленной символьной строки. Конечно, достаточно легко реализовать версию класса для извлечения символов из файла.

Вероятно, наилучший способ увидеть, как может использоваться сканер, — это его использовать. Вот небольшой код, разбивающий пример инструкции на токены:

```
// Листинг Б.6
$context = new Context();
$user_in = "\$input equals '4' or \$input equals 'four'";
$reader = new StringReader($user_in);
$scanner = new Scanner($reader, $context);

while ($scanner->nextToken() != Scanner::EOF)
{
    print $scanner->token();
    print "    {$scanner->charNo()}";
    print "    {$scanner->getTypeString()}\n";
}
```

Сначала мы инициализируем объект типа `Scanner`, а затем извлекаем все токены из заданной символьной строки в цикле с помощью повторяющегося вызова метода `nextToken()`. Метод `token()` возвращает текущую порцию данных из входных данных, совпавших с токеном. С помощью метода `charNo()` можно определить текущую позицию в заданной символьной строке. Метод `getTypeString()` возвращает строковую версию типа текущего токена, определенную ранее в виде константы. Ниже приведен пример выводимого результата синтаксического анализа:

```
$       1       CHAR
input   6       WORD
        7       WHITESPACE
equals  13      WORD
        14      WHITESPACE
'       15      APOS
4       16      WORD
'       17      APOS
        18      WHITESPACE
or      20      WORD
```

```
        21      WHITESPACE
$       22      CHAR
input   27      WORD
        28      WHITESPACE
equals  34      WORD
        35      WHITESPACE
'       36      APOS
four    40      WORD
'       41      APOS
```

При желании можно, конечно, придумать и более изощренные токены, но и имеющихся токенов вполне достаточно для демонстрации возможностей синтаксического анализатора. Разбиение символьной строки на токены не представляет особых трудностей. Как же тогда реализовать грамматику в коде?

Синтаксический анализатор

Один из подходов к реализации синтаксического анализатора состоит в создании древовидной структуры объектов типа `Parser`. Ниже приведено определение используемого для этой цели абстрактного класса `Parser`:

```
// Листинг Б.7
abstract class Parser
{
    public const GIP_RESPECTSPACE = 1;
    protected bool $respectSpace = false;
    protected static bool $debug = false;
    protected bool $discard = false;
    protected string $name;
    private static int $count = 0;
    public function __construct(string $name = null,
                                array $options = [])
    {
        if (is_null($name))
        {
            self::$count++;
            $this->name =
                get_class($this) . " (" . self::$count . ")";
        }
        else
        {
            $this->name = $name;
        }
```

```
        if (isset($options[self::GIP_RESPECTSPACE]))
        {
            $this->respectSpace = true;
        }
    }
    protected function next(Scanner $scanner): void
    {
        $scanner->nextToken();
        if (! $this->respectSpace)
        {
            $scanner->eatWhiteSpace();
        }
    }
    public function spaceSignificant(bool $bool): bool
    {
        $this->respectSpace = $bool;
    }
    public static function setDebug(bool $bool): void
    {
        self::$debug = $bool;
    }
    public function setHandler(Handler $handler): void
    {
        $this->handler = $handler;
    }
    final public function scan(Scanner $scanner): bool
    {
        if ($scanner->tokenType() == Scanner::SOF)
        {
            $scanner->nextToken();
        }
        $ret = $this->doScan($scanner);
        if ($ret && ! $this->discard && $this->term())
        {
            $this->push($scanner);
        }
        if ($ret)
        {
            $this->invokeHandler($scanner);
        }
        if ($this->term() && $ret)
        {
            $this->next($scanner);
        }
        $this->report("::scan возвращает $ret");
        return $ret;
    }
```

```
    public function discard(): void
    {
        $this->discard = true;
    }
    abstract public function trigger(Scanner $scanner): bool;
    public function term(): bool
    {
        return true;
    }
    // private/protected
    protected function invokeHandler(Scanner $scanner): void
    {
        if (! empty($this->handler))
        {
            $this->report("Вызов обработчика: " .
                          get_class($this->handler));
            $this->handler->handleMatch($this, $scanner);
        }
    }
    protected function report($msg): void
    {
        if (self::$debug)
        {
            print "<{$this->name}> " . get_class($this) . ": $msg\n";
        }
    }
    protected function push(Scanner $scanner): void
    {
        $context = $scanner->getContext();
        $context->pushResult($scanner->token());
    }
    abstract protected function doScan(Scanner $scanner): bool;
}
```

Основным в данном классе является метод scan(). В нем расположена основная логика синтаксического анализатора. Методу scan() передается объект типа Scanner, который он обрабатывает. Прежде всего, класс Parser делегирует свои функции конкретному дочернему классу, для чего вызывается абстрактный метод doScan(). Этот метод возвращает истинное или ложное логическое значение. Конкретный пример реализации будет рассмотрен ниже.

Если метод doScan() сообщает об успешном завершении и соблюдается ряд других условий, то результат синтаксического анализа размещается в стеке объекта типа Context. Ссылка на объект типа Context, применяемый в объекте типа Parser, сохраняется также в объекте типа Scanner,

и благодаря этому становится возможным обмен результатами между этими объектами. Конкретные результаты синтаксического анализа размещаются в стеке приведенным ниже методом `Parser::push()`:

```
// Листинг Б.8
protected function push(Scanner $scanner): void
{
    $context = $scanner->getContext();
    $context->pushResult($scanner->token());
}
```

Помимо неудачного исхода синтаксического анализа, имеются еще два условия, способные помешать размещению данных в стеке сканера. Во-первых, из клиентского кода может поступить запрос на отброс корректных результатов анализа путем вызова `discard()`. В итоге устанавливается истинное логическое значение в свойстве `$discard`. Во-вторых, значения в стеке могут размещать только терминальные анализаторы (т.е. анализаторы, которые не состоят из других анализаторов). Составные анализаторы (т.е. экземпляры объектов типа `CollectionParser`), напротив, поручают сохранение результатов успешного анализа своим дочерним классам. Чтобы проверить, является ли анализатор терминальным, достаточно вызвать метод `term()`. В анализаторах коллекций этот метод должен быть переопределен так, чтобы он возвращал ложное логическое значение.

Если конкретная реализация синтаксического анализатора успешно справляется с исходным текстом, вызывается еще один метод `invokeHandler()`, которому передается объект типа `Scanner`. Если объект типа `Handler` (т.е. объект, класс которого реализует интерфейс `Handler`) присоединен к объекту типа `Parser` (с помощью вызова метода `setHandler()`), в нем вызывается метод `handleMatch()`. В данном случае обработчики применяются для того, чтобы успешно проанализированная грамматика оказалась полезной для выполнения каких-нибудь действий, как будет показано далее.

Возвращаясь к методу `scan()`, следует заметить, что в нем вызывается объект типа `Scanner` (через метод `next()`), чтобы переместиться в исходном тексте к следующему токену. При этом вызываются методы `nextToken()` и `eatWhiteSpace()` объекта типа `Scanner`. И наконец возвращается значение, полученное из метода `doScan()`.

Помимо метода `doScan()`, обратите внимание на абстрактный метод `trigger()`. Он служит для того, чтобы выяснить, удалось ли анализатору извлечь что-нибудь совпавшее из исходной строки. Если метод `trigger()`

возвращает ложное логическое значение, значит, условия для выполнения синтаксического анализа не выполняются. Рассмотрим конкретную реализацию терминального объекта типа `Parser`. Приведенный ниже класс `CharacterParse` служит для того, чтобы извлекать из исходной строки одиночные совпадающие символы:

```
// Листинг Б.9
class CharacterParse extends Parser
{
    public function __construct(private string $char,
                                string $name = null,
                                array $options = [])
    {
        parent:: construct($name, $options);
    }
    public function trigger(Scanner $scanner): bool
    {
        return ($scanner->token() == $this->char);
    }
    protected function doScan(Scanner $scanner): bool
    {
        return ($this->trigger($scanner));
    }
}
```

Конструктору данного класса передается символ для сравнения и необязательное имя анализатора для отладочных целей. В методе `trigger()` просто проверяется, не достиг ли сканер такого токена в исходной строке, который соответствует переданному символу. Но поскольку никакого сканирования в дальнейшем не потребуется, в методе `doScan()` просто вызывается метод `trigger()`.

Как видите, реализовать терминальный синтаксический анализатор совсем не трудно. А теперь выясним, каким образом действует анализатор коллекций. Определим сначала общий суперкласс, а затем перейдем к конкретному примеру:

```
// Листинг Б.10
abstract class CollectionParse extends Parser
{
    protected array $parsers = [];
    public function add(Parser $p): Parser
    {
        if (is_null($p))
        {
            throw new Exception("аргумент равен null");
```

```
        }
        $this->parsers[] = $p;
        return $p;
    }
    public function term(): bool
    {
        return false;
    }
}

// Листинг Б.11
class SequenceParse extends CollectionParse
{
    public function trigger(Scanner $scanner): bool
    {
        if (empty($this->parsers))
        {
            return false;
        }
        return $this->parsers[0]->trigger($scanner);
    }
    protected function doScan(Scanner $scanner): bool
    {
        $start_state = $scanner->getState();
        foreach ($this->parsers as $parser)
        {
            if (!($parser->trigger($scanner) &&
                  $parser->scan($scanner)))
            {
                $scanner->setState($start_state);
                return false;
            }
        }
        return true;
    }
}
```

В абстрактном классе `CollectionParse` просто реализован метод `add()`, с помощью которого происходит агрегирование объектов типа `Parser` и переопределяется метод `term()`, чтобы он возвращал ложное логическое значение. В методе `SequenceParse::trigger()` проверяется только первый дочерний объект типа `Parser`; при этом вызывается его метод `trigger()`. При обращении к объекту типа `Parser` сначала вызывается метод `SequenceParse::trigger()`, чтобы выяснить, стоит ли вызывать метод `CollectionParse::scan()`. Если ме-

тод CollectionParse::scan() вызван, то затем вызываются методы doScan(), trigger() и scan() всех дочерних объектов типа Parser. Если же при вызове хотя бы одного из этих методов возникает ошибка, то и метод CollectionParse::doScan() завершается с ошибкой.

Одно из затруднений, возникающих при выполнении синтаксического анализа, связано с необходимостью проверять анализируемое содержимое. Объект типа SequenceParse может содержать целое дерево синтаксических анализаторов, каждый из которых, в свою очередь, может агрегировать другие анализаторы. В итоге объект типа Scanner разместит в стеке один или несколько токенов и зарегистрирует полученные данные в объекте типа Context. Но если вдруг последний дочерний объект типа Parser возвратит ложное логическое значение, что тогда должен делать объект типа SequenceParse со всеми результатами, сохраняемыми в объекте типа Context другими дочерними объектами, возвращающими истинное логическое значение? Должна быть получена либо вся последовательность токенов, либо вообще ничего. Поэтому у нас не остается иного выбора, кроме как вернуть объекты типа Context и Scanner в первоначальное состояние. Для этого в начале метода doScan() сохраняется их состояние, а перед тем, как возвратить ложное логическое значение при неудачном исходе, вызывается метод setState(). Естественно, что при удачном исходе восстанавливать первоначальное состояние объектов не нужно.

Ради полноты картины ниже приводятся все оставшиеся классы из реализации синтаксического анализатора:

```
// Листинг Б.12
class RepetitionParse extends CollectionParse
{
    public function __construct(private int $min = 0,
                                private int $max = 0,
                                ? string $name = null,
                                array $options = [])
    {
        parent:: __construct($name, $options);
        if ($max < $min && $max > 0)
        {
            throw new Exception(
                "максимум ( $max ) меньше минимума ( $min )"
            );
        }
    }
```

```
    public function trigger(Scanner $scanner): bool
    {
        return true;
    }
    protected function doScan(Scanner $scanner): bool
    {
        $start_state = $scanner->getState();
        if (empty($this->parsers))
        {
            return true;
        }
        $parser = $this->parsers[0];
        $count = 0;
        while (true)
        {
            if ($this->max > 0 && $count >= $this->max)
            {
                return true;
            }
            if (! $parser->trigger($scanner))
            {
                if ($this->min == 0 || $count >= $this->min)
                {
                    return true;
                }
                else
                {
                    $scanner->setState($start_state);
                    return false;
                }
            }
            if (! $parser->scan($scanner))
            {
                if ($this->min == 0 || $count >= $this->min)
                {
                    return true;
                }
                else
                {
                    $scanner->setState($start_state);
                    return false;
                }
            }
            $count++;
        }
    }
}
```

```
// Листинг Б.13
// В этом анализаторе достигается совпадение,
// если оно происходит в одном или двух других
// подчиненных анализаторах
class AlternationParse extends CollectionParse
{
    public function trigger(Scanner $scanner): bool
    {
        foreach ($this->parsers as $parser)
        {
            if ($parser->trigger($scanner))
            {
                return true;
            }
        }
        return false;
    }
    protected function doScan(Scanner $scanner): bool
    {
        $type = $scanner->tokenType();
        $start_state = $scanner->getState();
        foreach ($this->parsers as $parser)
        {
            if ($type == $parser->trigger($scanner) &&
                         $parser->scan($scanner))
            {
                return true;
            }
        }
        $scanner->setState($start_state);
        return false;
    }
}

// Листинг Б.14
// В этом терминальном анализаторе проверяется
// совпадение со строковым литералом
class StringLiteralParse extends Parser
{
    public function trigger(Scanner $scanner): bool
    {
        return (
            $scanner->tokenType() == Scanner::APOS ||
            $scanner->tokenType() == Scanner::QUOTE
        );
    }
```

```
    protected function push(Scanner $scanner): void
    {
    }
    protected function doScan(Scanner $scanner): bool
    {
        $quotechar = $scanner->tokenType();
        $ret = false;
        $string = "";
        while ($token = $scanner->nextToken())
        {
            if ($token == $quotechar)
            {
                $ret = true;
                break;
            }
            $string .= $scanner->token();
        }
        if ($string && ! $this->discard)
        {
            $scanner->getContext()->pushResult($string);
        }
        return $ret;
    }
}

// Листинг Б.15
// В этом терминальном анализаторе проверяется
// совпадение с токеном слова
class WordParse extends Parser
{
    public function __construct(private $word = null,
                                $name = null,
                                $options = [])
    {
        parent:: construct($name, $options);
    }
    public function trigger(Scanner $scanner): bool
    {
        if ($scanner->tokenType() != Scanner::WORD)
        {
            return false;
        }
        if (is_null($this->word))
        {
            return true;
        }
        return ($this->word == $scanner->token());
    }
```

```
    protected function doScan(Scanner $scanner): bool
    {
        return ($this->trigger($scanner));
    }
}
```

Сочетая терминальные и нетерминальные объекты типа `Parser`, можно построить достаточно развитый логически синтаксический анализатор. Все классы типа `Parser`, применяемые в рассматриваемом здесь примере, представлены на рис. Б.1.

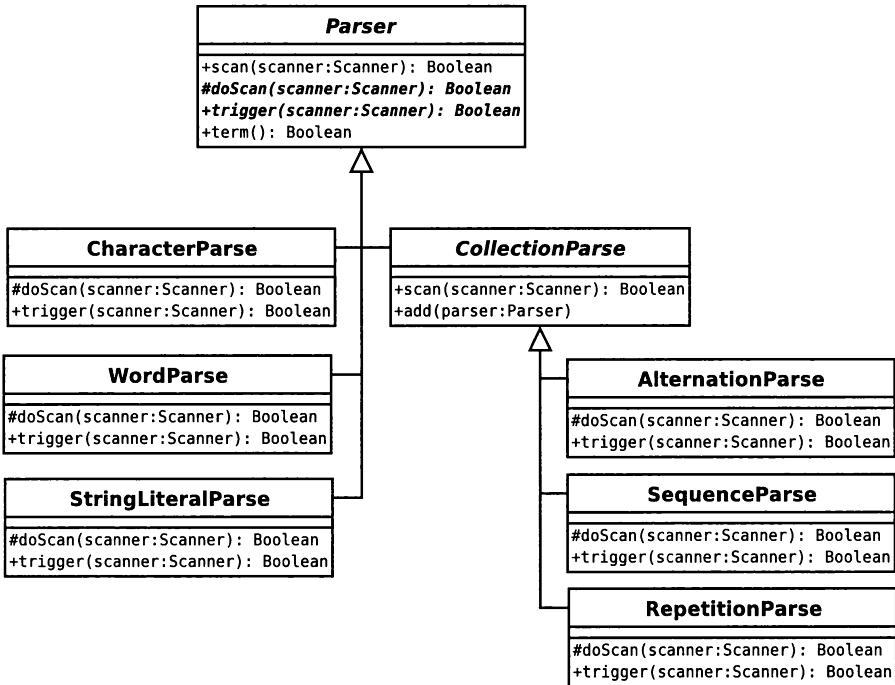


Рис. Б.1. *Классы синтаксического анализатора*

Идея, положенная в основу применения шаблона `Composite`, состоит в том, что в клиентском коде можно реализовать в коде грамматику, формы записи которой очень точно соответствуют расширенной нормальной форме Бэкуса–Наура. Эта параллель между рассмотренными выше классами и фрагментами формы Бэкуса–Наура отслеживается в табл. Б.1.

Таблица Б.1. Составные анализаторы и расширенная нормальная форма Бэкуса–Наура

Класс	Пример формы записи	Описание
AlternationParse	orExpr | andExpr	Одно или другое
SequenceParse	'and' operand	Список (в требуемом порядке)
RepetitionParse	(eqExpr)*	Требуется нуль или больше раз

Теперь напишем клиентский код для реализации мини-языка. В качестве напоминания ниже приводится фрагмент расширенной нормальной формы Бэкуса–Наура, представленный в главе 11, “Выполнение задач и представление результатов”:

```
expr     = operand { orExpr | andExpr }
operand  = ( '(' expr ')' | ? string literal ? | variable ) { eqExpr }
orExpr   = 'or'     operand
andExpr  = 'and'    operand
eqExpr   = 'equals' operand
variable = '$' , ? word ?
```

В приведенном ниже простом классе грамматика реализована на основе приведенного фрагмента и выполняет соответствующие действия:

```
// Листинг Б.16
class MarkParse
{
    private Parser $expression;
    private Parser $operand;
    private Expression $interpreter;
    public function __construct($statement)
    {
        $this->compile($statement);
    }
    public function evaluate($input): mixed
    {
        $icontext = new InterpreterContext();
        $prefab = new VariableExpression('input', $input);
        // Добавление входной переменной в Context
        $prefab->interpret($icontext);
        $this->interpreter->interpret($icontext);
        return $icontext->lookup($this->interpreter);
    }
```

```
public function compile($statementStr): void
{
    // Построение дерева синтаксического анализа
    $context = new Context();
    $scanner = new Scanner(new StringReader($statementStr),
                           $context);
    $statement = $this->expression();
    $scanresult = $statement->scan($scanner);
    if (! $scanresult || $scanner->tokenType() != Scanner::EOF)
    {
        $msg = "";
        $msg .= " line: {$scanner->lineNo()} ";
        $msg .= " char: {$scanner->charNo()}";
        $msg .= " token: {$scanner->token()}\n";
        throw new \Exception($msg);
    }
    $this->interpreter = $scanner->getContext()->popResult();
}
public function expression(): Parser
{
    if (! isset($this->expression))
    {
        $this->expression = new SequenceParse();
        $this->expression->add($this->operand());
        $bools = new RepetitionParse();
        $whichbool = new AlternationParse();
        $whichbool->add($this->orExpr());
        $whichbool->add($this->andExpr());
        $bools->add($whichbool);
        $this->expression->add($bools);
    }

    return $this->expression;
}
public function orExpr(): Parser
{
    $ or = new SequenceParse();
    $ or ->add(new WordParse('or'))->discard();
    $ or ->add($this->operand());
    $ or ->setHandler(new BooleanOrHandler());
    return $ or;
}
public function andExpr(): Parser
{
    $ and = new SequenceParse();
    $ and ->add(new WordParse('and'))->discard();
    $ and ->add($this->operand());
    $ and ->setHandler(new BooleanAndHandler());
```

```
        return $ and;
    }
    public function operand(): Parser
    {
        if (! isset($this->operand))
        {
            $this->operand = new SequenceParse();
            $comp = new AlternationParse();
            $exp = new SequenceParse();
            $exp->add(new CharacterParse('('))->discard();
            $exp->add($this->expression());
            $exp->add(new CharacterParse(')'))->discard();
            $comp->add($exp);
            $comp->add(new StringLiteralParse())
            ->setHandler(new StringLiteralHandler());
            $comp->add($this->variable());
            $this->operand->add($comp);
            $this->operand->add(
                new RepetitionParse())->add($this->eqExpr());
        }
        return $this->operand;
    }
    public function eqExpr(): Parser
    {
        $equals = new SequenceParse();
        $equals->add(new WordParse('equals'))->discard();
        $equals->add($this->operand());
        $equals->setHandler(new EqualsHandler());
        return $equals;
    }
    public function variable(): Parser
    {
        $variable = new SequenceParse();
        $variable->add(new CharacterParse('$'))->discard();
        $variable->add(new WordParse());
        $variable->setHandler(new VariableHandler());
        return $variable;
    }
}
```

Приведенный выше класс на первый взгляд может показаться сложным, но он всего лишь реализует определенную ранее грамматику. Большинство методов аналогично продукциям (т.е. именам, расположенным в начале продукционной строки расширенной нормальной формы Бэкуса–Наура, таким как `eqExpr` или `andExpr`). Если проанализировать метод `expression()`, то можно обнаружить, что в нем составляется правило, аналогичное определенному выше в расширенной нормальной форме Бэкуса–Наура.

```
// Листинг Б.17
// expr = operand { orExpr | andExpr }
public function expression(): Parser
{
    if (! isset($this->expression))
    {
        $this->expression = new SequenceParse();
        $this->expression->add($this->operand());
        $bools = new RepetitionParse();
        $whichbool = new AlternationParse();
        $whichbool->add($this->orExpr());
        $whichbool->add($this->andExpr());
        $bools->add($whichbool);
        $this->expression->add($bools);
    }
    return $this->expression;
}
```

Как в коде, так и в расширенной нормальной форме Бэкуса–Наура определяется последовательность, состоящая из ссылок на операнды, за которыми нуль или большее количество раз следуют альтернативные варианты orExpr или andExpr. Обратите внимание на то, что в переменной свойства сохраняется объект типа Parser, возвращаемый приведенным ниже методом. Это сделано с целью предотвратить зацикливание, поскольку методы, вызываемые из метода expression(), сами вызывают его.

Единственными методами, которые делают еще кое-что, кроме реализации грамматики, являются методы compile() и evaluate(). Метод compile() можно вызвать напрямую или автоматически из конструктора. Ему передается строка выражения, которая используется для создания объекта типа Scanner. Он вызывает метод expression(), который возвращает дерево объектов типа Parser, составляющее грамматику. Затем он вызывает метод Parser::scan(), передавая ему объект типа Scanner. Если исходный код проанализировать нельзя, в методе compile() генерируется исключение; в противном случае в нем извлекается результат компиляции из контекста объекта типа Scanner. Как будет показано далее, это должен быть объект типа Expression. Полученный результат сохраняется в свойстве $interpreter.

Метод evaluate() делает значение доступным дереву объектов типа Expression. Для этого заранее определяется объект типа VariableExpression, который соответствует переменной $input и регистрируется в объекте типа Context. В итоге он передается основному объекту типа Expression. Аналогично глобальным переменным PHP

типа `$_REQUEST` переменная `$input` будет всегда доступна программистам, пользующимся языком MarkLogic.

На заметку Класс `VariableExpression`, являющийся частью шаблона Interpreter, подробно описан в главе 11, "Выполнение задач и представление результатов".

В методе `evaluate()` вызывается метод `Expression::interpret()` для получения окончательного результата. Не стоит, однако, забывать, что результаты работы интерпретатора следует извлечь из объекта типа `Context`.

Выше был описан процесс синтаксического анализа текста и реализации грамматики. В главе 11 было также показано, каким образом шаблон Interpreter применяется для комбинирования объектов типа `Expression` и выполнения запроса. Но все еще непонятно, каким образом эти два процесса связаны вместе. Как, например, передать интерпретатору результаты из дерева синтаксического анализа? Ответ состоит в том, чтобы воспользоваться объектами типа `Handler`, которые связываются с объектами типа `Parser` посредством вызова `Parser::setHandler()`. Рассмотрим один из способов обработки переменных. Свяжем объект типа `VariableHandler` с объектом типа `Parser` в методе `variable()`:

```
// Листинг Б.18
$variable->setHandler(new VariableHandler());
```

Ниже приведено определение интерфейса `Handler`:

```
// Листинг Б.19
interface Handler
{
    public function handleMatch(
        Parser $parser,
        Scanner $scanner
    ): void;
}
```

А вот как выглядит класс `VariableHandler`, реализующий этот интерфейс:

```
// Листинг Б.20
class VariableHandler implements Handler
{
    public function handleMatch(Parser $parser,
                                Scanner $scanner): void
    {
```

```
        $varname = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
                    new VariableExpression($varname));
    }
}
```

Если объект типа `Parser`, с которым связан объект типа `VariableHandler`, обнаруживает совпадение во время операции сканирования, то вызывается метод `handleMatch()`. По определению последний элемент в стеке будет соответствовать имени переменной. Он удаляется из стека и заменяется новым объектом типа `VariableExpression`, содержащим правильное имя. Аналогичным образом создаются объекты типа `EqualsExpression`, `LiteralExpression` и т.п.

Ниже приведены остальные обработчики:

```
// Листинг Б.21
class StringLiteralHandler implements Handler
{
    public function handleMatch(Parser $parser,
                                Scanner $scanner): void
    {
        $value = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
                    new LiteralExpression($value));
    }
}

// Листинг Б.22
class EqualsHandler implements Handler
{
    public function handleMatch(Parser $parser,
                                Scanner $scanner): void
    {
        $comp1 = $scanner->getContext()->popResult();
        $comp2 = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
                    new BooleanEqualsExpression($comp1, $comp2));
    }
}

// Листинг Б.23
class BooleanOrHandler implements Handler
{
    public function handleMatch(Parser $parser,
                                Scanner $scanner): void
```

```
    {
        $comp1 = $scanner->getContext()->popResult();
        $comp2 = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
                    new BooleanOrExpression($comp1, $comp2));
    }
}

// Листинг Б.24
class BooleanAndHandler implements Handler
{
    public function handleMatch(Parser $parser,
                                Scanner $scanner): void
    {
        $comp1 = $scanner->getContext()->popResult();
        $comp2 = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
                    new BooleanAndExpression($comp1, $comp2));
    }
}
```

Не следует, однако, забывать, что нам еще необходим пример применения шаблона Interpreter, описанный в главе 11. Тогда мы сможем работать с классом `MarkParse`, как показано ниже:

```
// Листинг Б.25
$input = 'five';
$statement = "( \$input equals 'five')";
$engine = new MarkParse($statement);
$result = $engine->evaluate($input);
print "Ввод: $input Вычисление: $statement\n";
if ($result)
{
    print "Истинно!\n";
}
else
{
    print "Ложно!\n";
}
```

При выполнении данного фрагмента кода должны быть выведены следующие результаты:

```
Ввод: five Вычисление: ( $input equals 'five')
Истинно!
```

Предметный указатель

$this 57, 107
__construct() 93
__NAMESPACE__ 188

A

abstract 112, 113
API
  Reflection 214
  RESTful 455
  SimpleXML 66
as 125

C

catch 139
class 50, 206
clone 164, 333, 338
Composer 596, 601, 631
const 111

D

DIRECTORY_SEPARATOR 198
DSL 396

E

EBNF 397
extends 89, 118

F

final 149
finally 147
function 55

G

Git 34, 649
  установка 652

I

implements 115, 118
include_path 193
instanceof 86, 205
insteadof 124
interface 115

J

Jenkins 791

L

LAMP 759

M

mixed 76

N

namespace 184
new 50
new class 178

P

Packagist 596
parent 91, 93, 106
PDO 108
PEAR 191, 596
Phing 728
  задания 752
  свойства 735
  типы 745
PHP_CodeSniffer 789
PHPDocumentor 785
PHPUnit 35, 688, 782
private 94, 253
protected 94, 253
public 52, 94

R

Reflection API 214
RESTful 455

S

Selenium 715
self 107
SOAP 455
SPL 423
SSH 658
static 44, 106, 132
strict_types 74

T

throw 138
trait 120

U

UML. См. Язык UML
use 120, 124, 186

V

Vagrant 602, 759

X

XUnit 688

Y

yield 539

А

Автозагрузка 195, 626, 629, 638
Агрегирование 264
Аннотация 46, 232
Аргумент
  именованный 62
  по умолчанию 61
Атрибут 232

Б

База данных
  реляционная 527
Безопасность 248

В

Внешний ключ 527

Г

Генератор 539

Д

Делегирование 157, 159, 294, 383
Деструктор 161
Диаграммы классов 258
Дублирование 256, 812

З

Заглушка 699
Загрузка по требованию 564
Замыкание 174

И

Инкапсуляция 253, 297
Интеграция 776
  непрерывная 604, 777
Интерпретатор 833
Интерфейс 115
  Iterator 534, 562
  Reflection API 214
  текучий 572, 701
Исключение 137

К

Класс 50
  final 149
  абстрактный 112
  агрегирование 264
  анонимный 177
  ассоциация 262
  выбор 248
  диаграмма 258
  дочерний 80
  завершенный 149
  композиция 265
  конструктор 58
  метод 55
  наследование 210
  переменная-член 52
  поиск 203
  представление в UML 258
  родительский 80
  свойство 52, 210
  суперкласс 80
  экземпляр 50
Ковариантность возвращаемого типа 532
Код
  клиентский 53
  некачественный 256
Композиция 265, 292, 296, 383, 426
Конструктор 58, 161
Контроль версий 598, 649, 821
  Mercurial 651
  Subversion 651
Копирование поверхностное 165

М

Метафора закона тривиальности 612
Метод 55
  __clone() 165
  final 149
  __toString() 167
  абстрактный 113
  вызов 212
  доступа 97
  завершенный 149
  магический 58, 163

перехватчик 152
получение информации 207
статический 106
фабричный 131
Модульное тестирование 684

Н

Наследование 79, 80, 90, 210, 288, 380
множественное 118
одиночное 118
Непрерывная интеграция 36, 775, 822
контроль версий 780

О

Область видимости 464
приложения 464
Обработка ошибок 135, 137
catch 139
Error 151
Exception 137, 141
throw 138
исключение 137
Объект 50, 307
передача
по значению 41
по ссылке 41
проверка типа 204
Объектно-ориентированное программирование 25
Объявление типа 70
Ортогональность 248, 415
Ослабление связей 454

П

Пакет 181, 633
автозагрузка 195
версия 635
имитация файловой системой 189
создание 639
Паттерн. См. Проектный шаблон
Переменная глобальная 257, 313, 811
Перехват 152
Повторное использование 812
Подкласс 80
Позднее статическое связывание 132
Полиморфизм 250, 309, 812
Потеря соответствия 527
Представление 465
Программный проект 239
ортогональность 248
тесная связь 247
Продукция 398
Проектирование 239
Проектный шаблон 28, 274, 814
Abstract Factory 275, 326, 568
Application Controller 481
Command 436
Composite 364, 427, 819
Data Access Object 526
Data Mapper 526, 585
Data Transfer Object 569
Decorator 380
Dependency Injection 342
Domain Model 518
Domain Object Factory 564, 565
Facade 389
Factory Method 319, 331
Front Controller 442, 466
Identity Map 548
Identity Object 569
Intercepting Filter 388
Interpreter 395
Layer Supertype 515
Lazy Load 561
Null Object 443
Observer 415
Page Controller 499
Prototype 333
Registry 457
Selection Factory 578
Service Layer 511
Service Locator 339, 343
Session Facade 511
Singleton 313
Strategy 292, 409, 818
Template Method 404
Template View 507
Transaction Script 512
Unit of Work 554
Update Factory 578
View Helper 507
Visitor 426
корпоративного приложения 452
название 277
постановка задачи 278

решение 278
язык 277
Пространство имен 44, 183
use 186
вложенное 184

Р

Расширенная форма Бэкуса–Наура 397
Реестр 459
Рефакторинг 30
Рефлексия 159

С

Свойство 52, 210
константное 111
составное 159
статическое 106
Связность 247
Связь
слабая 248
тесная 247
Синдром неприятия чужой разработки 597
Стандарт 608, 820
PSR 610
PSR-1 613
PSR-4 689
PSR-12 616
Суперкласс 80

Т

Тестирование 343, 455, 603, 604, 684, 820
веб 708
модульное 684, 782
регрессионное 708
ручное 685
Типы данных 63
object 63
объявление 70
примитивные 63
автоматическое приведение 68
скалярные 72
составные 369
Трейт 118, 618
абстрактные методы 128
модификаторы доступа 129

У

Установка программного обеспечения 599
Утверждение 692

Ф

Фабрика 110, 312
Функция
var_dump() 52
анонимная 171
обратного вызова 169
проверки типов 64
со стрелками 172

Ш

Шаблон, проектный. См. Проектный шаблон

Э

Экстремальное программирование 29, 304, 688

Я

Язык
Java 28
PHP
версия 3, особенности 40
версия 4 40
версия 5 43
версия 7 45
история 39
слабая типизация 63
эволюция версий 28
UML 33, 258
агрегирование 264
ассоциация 262
атрибуты 260
использование 265
композиция 265
наследование 261
операции 261
последовательности 267
примечания 266
реализация 261
сообщения 268
предметно-ориентированный 396

У книзі розглядаються методики об'єктно-орієнтованого програмування на PHP і застосування головних принципів проектування програмного забезпечення на основі класичних проектних шаблонів, а також описуються інструментальні засоби та норми практики розробки, тестування, безперервної інтеграції та розгортання надійного прикладного коду. Шосте видання книги повністю оновлено відповідно до версії 8 мови PHP. У цій книзі детально описані нові можливості PHP, такі як атрибути та численні удосконалення в області оголошення типів.

Науково-популярне видання

Зандстра, Метт

PHP 8: об'єкти, шаблони та методики програмування
6-е видання

(Рос. мовою)

Зав. редакцією *С.М. Тригуб*

Із загальних питань звертайтеся до видавництва “Діалектика” за адресою:
info@dialektika.com, http://www.dialektika.com

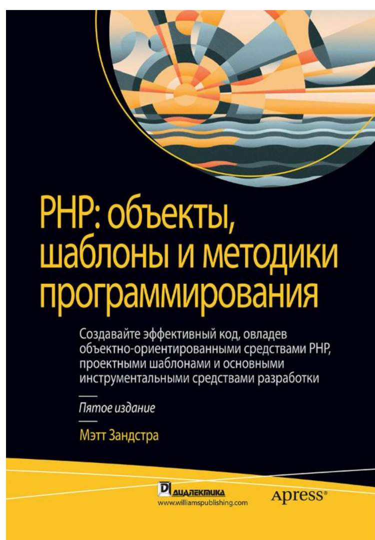
Підписано до друку 01.10.2021. Формат 60x90/16
Ум. друк. арк. 54,0. Обл.-вид. арк. 36,8
Зам. № 21-3261

Видавець ТОВ “Комп’ютерне видавництво “Діалектика”
03164, м. Київ, вул. Генерала Наумова, буд. 23-Б.
Свідоцтво суб’єкта видавничої справи ДК № 6758 від 16.05.2019.

Надруковано ТОВ “АЛЬФА ГРАФІК ”
03067, м. Київ, вул. Машинобудівна, 42
Свідоцтво суб’єкта видавничої справи ДК № 6838 від 09.07.2019.

PHP: ОБЪЕКТЫ, ШАБЛОНЫ И МЕТОДИКИ ПРОГРАММИРОВАНИЯ
ПЯТОЕ ИЗДАНИЕ

Мэтт Зандстра



www.williamspublishing.com

В этой книге рассматриваются методики объектно-ориентированного программирования на PHP, применение главных принципов проектирования программного обеспечения на основе классических проектных шаблонов, а также описываются инструментальные средства и нормы практики разработки, тестирования, непрерывной интеграции и развертывания надежного прикладного кода. Настоящее, пятое издание книги полностью обновлено по версии 7 языка PHP и включает описание диспетчера зависимостей Composer, инструментального средства Vagrant и рекомендаций стандартов по программированию на PHP.

Книга адресована разработчикам, твердо усвоившим основы программирования на PHP и стремящимся развить свои навыки проектирования веб-приложений, применяя нормы передовой практики разработки.

ISBN 978-5-907144-54-5 **в продаже**

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

PHP 8: объекты, шаблоны и методики программирования

Настоящее, шестое, издание этой популярной книги полностью обновлено в соответствии с версией 8 языка PHP и включает описание диспетчера зависимостей Composer, материал, посвященный инструментальным средствам Vagrant, PHPUnit, Jenkins и другим, а также рекомендации стандартов по программированию на PHP. В этой книге подробно описаны новые возможности PHP, такие как атрибуты и многочисленные усовершенствования в области объявления типов.

Основная цель книги — исследовать в контексте PHP некоторые устоявшиеся принципы проектирования и основные проектные шаблоны.

В части I этой книги рассматриваются ключевые объектно-ориентированные средства языка PHP, включая объявления классов и типов, наследование, рефлексию и многое другое. Часть II посвящена проектным шаблонам, в которых поясняются принципы, определяющие их эффективность, а также классическим шаблонам для проектирования корпоративных приложений и баз данных. В части III рассматриваются инструментальные средства и нормы практики, помогающие превратить качественный код в удачный проект. В этой части показано, как организовать коллективный труд разработчиков и выпуски программных продуктов с помощью Git и как управлять процессом построения проектов и зависимостями средствами Composer, а также исследованы стратегии автоматизированного тестирования и непрерывной интеграции. В целом книга посвящена основам объектно-ориентированного программирования, принципам проектирования и нормам передовой практики разработки, которые призваны помочь читателю в разработке изящных, надежных и устойчивых систем.

НА ВЕБ-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с веб-сайта издательства по адресу:

`http://www.williamspublishing.com/Books/978-5-907458-23-9.html`.

Категория: программирование

Предмет рассмотрения: методики и нормы практики программирования на PHP

Уровень: промежуточный/продвинутый

Комп'ютерне видавництво
"ДІАЛЕКТИКА"
www.dialektika.com

Apress®
www.apress.com

ISBN 978-617-7987-35-1