ПРОФЕССИОНАЛЬНО О РАЗРАБОТКЕ ВЕБ-ПРИЛОЖЕНИЙ!

4-е издание

PHP

ОБЪЕКТЫ, ШАБЛОНЫ И МЕТОДИКИ ПРОГРАММИРОВАНИЯ

Создайте высокопрофессиональный код на PHP, изучив его объектно-ориентированные возможности, шаблоны проектных решений и важные средства разработки

Мэтт Зандстра

ВИЛЬЯМС
www.williamspublishing.com

Apress®
www.apress.com

PHP
объекты, шаблоны и методики программирования

4-е издание

PHP
объекты, шаблоны и методики программирования

4-е издание

Мэтт Зандстра

Москва • Санкт-Петербург • Киев
2015

ББК 32.973.26-018.2.75
З-27
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. физ.-мат. наук *С.Г. Тригуб*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Зандстра, Мэтт.

З-27 PHP: объекты, шаблоны и методики программирования, 4-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 576 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1922-9 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Мэтт Зандстра

PHP: объекты, шаблоны и методики программирования

4-е издание

Литературный редактор *Л.Н. Красножон*
Верстка *М.А. Удалов*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 23.07.2015. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 46,44. Уч.-изд. л. 33.
Тираж 1000 экз. Заказ № 3880.

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1922-9 (рус.)
ISBN 978-1-4302-6031-8 (англ.)

Оглавление

Содержание

Луизе, которая для меня важнее всего

Об авторе

Мэтт Зандстра почти 20 лет проработал веб-программистом, консультантом по PHP и составителем технической документации. Он был старшим разработчиком в компании Yahoo! и работал в офисах компании как в Лондоне, так и в Силиконовой долине. Сейчас он зарабатывает себе на жизнь в качестве свободного консультанта и писателя. До этой книги Мэтт написал книгу *Освой самостоятельно PHP за 24 часа* (3-е издание), выпущенную в ИД "Вильямс" в 2007 году, а также был соавтором книги *DHTML Unleashed* (издательство SAMS Publishing). Кроме всего прочего, он писал статьи для *Linux Magazine*, Zend.com, IBM DeveloperWorks и *php|architect Magazine*.

Мэтт также изучает литературу и пишет фантастические рассказы. Он получил степень магистра в области литературного мастерства (creative writing) в университетах Манчестера и Ист-Англии.

В то время, когда не нужно мотаться по всем уголкам Великобритании, изучая литературу или выполняя какую-либо работу, Мэтт живет в Ливерпуле со своей женой Луизой и двумя детьми, Холли и Джейком.

О техническом рецензенте

Вес Хант (Wes Hunt) был разработчиком приложений и руководителем группы по созданию пользовательского интерфейса (UX lead) с конца 90-х годов прошлого века. В настоящее время он является основателем и ведущим разработчиком в компании Arrnigent — стартапе веб-служб в индустрии инвестиций в недвижимость. Хант использует PHP и Scala для быстрой разработки веб-приложений и REST-служб как для небольших интернет-компаний (стартапов), так и для крупных корпораций. В свободное от работы время он обучает программированию детей в Монтане через сайт `CodeMontana.org` (только в этом году их было более тысячи), а также занимается защитой прав программистов в группе пользователей сайта `MontanaProgrammers.org`.

Благодарности

Как всегда, хочу выразить признательность всем, кто работал над этим изданием книги. Также я должен вспомнить всех, 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 PHP 5. Выражаю благодарность организатору конференции Энди Бадду (Andy Budd) и всему животрепещущему сообществу разработчиков Брайтона. Благодарю также Джесси Вайт-Синис (Jessey White-Cinis), которая познакомила меня на конференции с Мартином Штрейхером (Martin Streicher) из издательства Apress.

Как и прежде, сотрудники издательства Apress оказывали мне неоценимую поддержку, высказывали ценные замечания и обеспечивали всяческое содействие. Я необычайно рад работать в такой команде профессионалов.

Выражаю благодарность и свою безграничную любовь своей жене Луизе и детям Холли и Джейку за то, что они периодически отвлекали меня от сложной работы и обеспечивали разрядку.

Спасибо Стивену Мецкеру (Steven Metsker) за любезное разрешение реализовать на PHP чрезвычайно упрощенную версию API синтаксического анализатора, который он представил в своей книге *Building Parsers in Java*.

Я работаю под музыку, и в первых трех изданиях этой книги я упомянул о великом диджее Джоне Пиле (John Peel), поборнике всего тайного и эклектичного. Я должен также поблагодарить радиовещателей, принявших эстафету от Джона, особенно тех, кто работает в таинственных уголках радиостанции BBC 6 Music and Dandelion Radio.

Предисловие

Когда меня впервые посетила идея написать эту книгу, объектно-ориентированные средства разработки в PHP использовали лишь избранные программисты. Однако со временем мы увидели не только безусловный рост популярности объектно-ориентированных средств языка PHP, но и развитие всего фреймворка. Безусловно, фреймворки (каркасы приложений) чрезвычайно полезны. Они составляют основу многих (на сегодняшний день, вероятно, большинства) веб-приложений. Более того, часто эти каркасы точно иллюстрируют основные подходы к проектированию программного обеспечения, рассматриваемые в этой книге.

Тем не менее здесь для разработчиков таится определенная опасность, точно так же как и при использовании любого другого полезного API. Речь идет о боязни того, 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 изменения в его функционал. 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 и непознанной инфраструктуры.

Несмотря на то что я отношусь к злобным изобретателям велосипедов, суть моих доводов состоит вовсе не в том, что я призываю вас полностью отказаться от своих любимых каркасов и начать создавать MVC-приложения "с нуля" (по крайней мере, хотя бы иногда). Наоборот, мы как разработчики должны понимать задачи, которые решает тот или иной каркас, а также разбираться в методиках, которые используются для их решения. Мы должны оценивать каждый каркас не только с точки зрения предлагаемых им функциональных возможностей, но и с точки зрения проектных решений, которые использовали их создатели, и оценить, насколько качественно они реализованы. Разумеется, что, когда нам позволяют обстоятельства, мы как разработчики должны развиваться и создавать собственные служебные и специализированные приложения, а со временем — создать собственную библиотеку повторно используемого кода.

Я надеюсь, что эта книга поможет PHP-разработчикам выработать навыки проектно-ориентированного мышления и реализовать их в собственных программных платформах и библиотеках. Здесь описаны также несколько концептуальных средств, которые пригодятся вам, когда наступит время действовать в одиночку и брать всю ответственность на себя.

Совсем недавно я потратил около года на обучение. И это то, что я настоятельно рекомендую сделать всем по целому ряду причин. Одна из них — после обучения вы сможете взглянуть на знакомый мир с другой точки зрения. После возврата к своей привычной консалтинговой деятельности я с удивлением обнаружил, что большинство моих старых клиентов и знакомых сделали решительный шаг вперед и перешли на использование системы контроля версий Git (в этом издании мне также пришлось отдать дань моде). Кроме того, почти все из них стали называть свою методологию разработки *гибкой* (*agile*), хотя трое из четверых моих новых клиентов попросили оценить их наспех созданную и по этой причине совсем не гибкую кодовую базу (codebase). В каждом проекте требуется сначала создать (или подогнать) набор модульных тестов, написать основную документацию и разработать механизм автоматизированного построения проекта. И только после этого можно браться за рефакторинг. В противном случае его результаты будут весьма плачевными. Я потратил немало усилий на то, чтобы освоить на практике все те средства

и правила, о которых говорится в последней части книги. И я надеюсь, что вы также оцените их по достоинству и что они помогут вам создавать надежные и гибкие программные системы.

От издательства

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 интересно услышать и любые другие замечания, которые вам хотелось бы высказать 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.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Адреса для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Украины: 03150, Киев, а/я 152

Часть I

Введение

Глава 1

PHP: проектирование и сопровождение систем

Одним из самых важных нововведений PHP 5 была расширенная поддержка объектно-ориентированного программирования. Это вызвало рост интереса к объектам и их проектированию в сообществе PHP-разработчиков. На самом деле это был новый этап развития процесса, начавшегося благодаря выпуску PHP версии 4, в которой объектно-ориентированное программирование впервые стало реальностью.

В этой главе мы рассмотрим несколько задач, для решения которых необходимо программирование с использованием объектов. Я очень коротко расскажу об эволюции шаблонов проектных решений (или просто, проектных шаблонов) и связанной с ними практикой программирования, используемой в языке Java. Хочется отметить, что аналогичные процессы происходят и в среде PHP-программистов.

Также будут кратко рассмотрены темы, освещаемые в данной книге.

- *Эволюция катастрофы*: проект не удался.
- *Проектирование и PHP*: как методы объектно-ориентированного проектирования пускают корни в PHP-сообществе.
- *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 PHP-кода можно писать прямо в коде веб-страниц, потому что в PHP предусмотрена соответствующая поддержка. Вы добавляете дополнительные функции (например, код доступа к базе данных) к файлам, которые можно скопировать с одной страницы на другую, и не успеваете оглянуться, как у вас уже есть рабочее веб-приложение.

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

Проблема возникает, когда вы возвращаетесь к коду, чтобы начать новый этап. Теперь ваша команда увеличилась, пользователей стало больше, бюджет тоже 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 20 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; при этом без дела сидит третий программист, который тоже работает с этим файлом.

Поскольку приложение популярно, вам нужно перенести код на новый сервер. Инсталляцию проекта нужно проводить вручную, и вы обнаруживаете, что пути к файлам, имена баз данных и пароли жестко закодированы во многих исходных файлах. На время этого перемещения вы останавливаете работу сайта, потому что не хотите затереть изменения в конфигурации, которых требует этот переход. Предполагаемые два часа работы превращаются в восемь, поскольку обнаруживается, что кто-то немного "поумничал", задействовав модуль `ModRewrite` сервера Apache, и теперь для нормальной работы приложения требуется, чтобы этот модуль функционировал на сервере и был правильно настроен.

Вы, наконец, успешно преодолеваете этап 2. 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, однако в результате более тщательного анализа обнаруживается, что это другая ошибка, которая вызывает похожее 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. Вы в страшной спешке вносите изменения, необходимые для исправления ошибок. Вы слишком спешите и не можете протестировать внесенные изменения, но это же просто операции копирования и вставки, что тут может случиться?

На следующее утро, приехав в офис, вы выясняете, что модуль "корзины для покупок" не работал всю ночь. Дело в том, что вчера при внесении изменений в последнюю минуту вы пропустили открывающие кавычки, в результате чего код стал нерабочим. Конечно, 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 для следующего дня "пожарной" работы.

Эта история о ежедневных буднях программистов может показаться преувеличением, но все это мне случалось наблюдать неоднократно. Многие PHP-проекты начинались с малого, а затем превращались в гиганты.

Поскольку в проекте логика приложения содержится также на уровне представления данных, дублирование происходит уже в коде запросов к базе данных, проверке аутентификации, обработке форм, причем этот код копируется с одной страницы на другую. Каждый раз, когда требуется внести изменение в один из этих блоков кода, его необходимо осуществить везде, где есть данный код, иначе неминуемо произойдет ошибка.

Отсутствие документации делает код трудным для чтения, а отсутствие тестирования позволяет скрытым дефектам оставаться необнаруженными до момента развертывания приложения. А изменение основного направления деятельности заказчика часто означает, что в результате модификации код меняет свое первоначальное назначение и в конце концов начинает выполнять задачи, для которых он в корне непригоден. И поскольку такой код, как правило, разрабатывался и развивался в качестве единой массы, в которой было много чего намешано, очень трудно, или даже невозможно, перестроиться и переписать его фрагменты, чтобы он соответствовал новой цели.

Но все это совсем неплохо, если вы — независимый консультант по PHP. Анализ и исправление подобной системы позволят вам покупать дорогие напитки и наборы DVD-дисков в течение полугода или даже дольше. А если серьезно, то проблемы подобного рода как раз и отличают успешный бизнес от неудачного.

PHP и другие языки

Феноменальная популярность PHP означает, что он был основательно протестирован во всех сферах своего применения еще на начальном этапе развития. Как вы увидите в следующей главе, PHP начинался как набор макросов для управления персональными веб-страницами. С появлением PHP 3 и в значительной степени PHP 4 этот язык быстро стал популярным и мощным инструментом создания крупных коммерческих веб-сайтов. Однако во многих случаях сферы его применения ограничивались разработкой сценариев и средств управления проектами, как и первоначальные версии. В глазах некоторых специалистов у PHP осталась несправедливая репутация языка для любителей, который больше приспособлен для задач представления данных.

Примерно в это же время (при вступлении в новое тысячелетие) в других сообществах программистов стали распространяться новые идеи. Интерес к объектно-ориентированному проектированию всколыхнул Java-сообщество. Возможно, вы думаете, что это излишне, ведь Java и так является объектно-ориентированным языком. Java обеспечивает лишь каркас для создания приложений, который нужно научиться правильно использовать, поскольку применение в программах классов и объектов само по себе не определяет конкретный подход к проектированию.

Понятие проектного шаблона как способа описания проблемы вместе с сутью ее решения впервые обсуждалось в 70-х годах прошлого века. Первоначально эта идея возникла в области строительства и архитектуры, а не в информатике. В начале 90-х годов программисты, использующие объектно-ориентированный подход, применяли такие же методы для определения и описания проблем разработки программного обеспечения. Основополагающая книга по проектным шаблонам, *Design Patterns: Elements of Reusable Object-Oriented Software*[1], под авторством "Банды четырех", опубликованная в 1995 году, незаменима и по сей день. Шаблоны, которые в ней содержатся, — необходимый первый шаг для каждого, кто начинает изучать эту тематику. Именно поэтому большинство шаблонов в данной книге взято из этого фундаментального труда.

В API самого языка Java используются многие основные шаблоны, но до конца 90-х годов проектные шаблоны еще не завладели сознанием сообщества программистов в такой степени. Книги о шаблонах быстро заполонили компьютерные раз-

[1] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* (пер. с англ., изд. "Питер", 2007).

делы книжных магазинов, и на форумах появились первые флеймы со словами похвалы или неодобрения.

Возможно, вы думаете, что шаблоны — это мощный способ передачи информации или, наоборот, просто "мыльный пузырь" (какой точки зрения придерживаюсь я, очевидно из названия книги). Каким бы ни было ваше мнение, трудно отрицать, что подход к разработке программного обеспечения, который при этом пропагандируется, полезен сам по себе.

Родственные темы также стали более популярными. Среди них — методология экстремального программирования (eXtreme Programming — XP), одним из авторов которой является Кент Бек (Kent Beck)[2]. XP — это подход к проектам, который направлен на гибкое, проектно-ориентированное, очень тщательное планирование и выполнение.

Один из важных принципов XP — утверждение, что тестирование является ключевым фактором для успеха проекта. Тесты должны быть автоматическими, выполняться часто, и желательно, чтобы они были разработаны до написания самого кода.

Еще одно требование методологии XP — проекты должны быть разбиты на малые (очень малые) итерации. И код, и требования к нему должны постоянно анализироваться. Архитектура и проект должны быть предметом постоянного совместного обсуждения, что ведет к частому пересмотру кода.

Если XP был воинствующим крылом в движении проектирования, то умеренная тенденция превосходно представлена в одной из лучших книг по программированию, которую я когда-либо читал: *The Pragmatic Programmer*[3] (Andrew Hunt, David Thomas, 2000 год).

Некоторые считают, что XP был в какой-то степени культовым методом, но за два десятилетия практики объектно-ориентированного программирования он достиг высочайшего уровня, а его принципы широко использовались и заимствовались. В особенности процесс пересмотра кода, который называется *рефакторинг*, был использован в качестве мощного дополнения к шаблонам. Рефакторинг развивался с 80-х годов, но был классифицирован только в каталоге рефакторинга Мартина Фаулера (Martin Fowler) *Refactoring: Improving the Design of Existing Code*[4], опубликованном в 1999 году, и тем самым определил данную область.

С развитием и ростом популярности методологии XP и шаблонов тестирование также стало ключевым вопросом. Важность автоматических тестов была еще более усилена с появлением мощной тестовой платформы JUnit, которая стала главным инструментом в арсенале Java-программистов. Основополагающая статья по этой теме, *Test Infected: Programmers Love Writing Tests* (Kent Beck, Erich Gamma) (`http://junit.sourceforge.net/doc/testinfected/testing.htm`), была превосходным введением в предмет и до сих пор не потеряла своей актуальности.

Версия PHP 4, более эффективная и с усиленной поддержкой объектов, вышла примерно в то же время. Благодаря этим улучшениям 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 Лердорфу (Rasmus Lerdorf) для разработки PHP. Как вы увидите в следующей главе, поддержка объектов в PHP была далеко не идеальной, но при выпол-

[2] Кент Бек. *Экстремальное программирование* (пер. с англ., изд. "Питер", 2002).

[3] Эндрю Хант, Дэвид Томас. *Программист-прагматик. Путь от подмастерья к мастеру* (пер. с англ., изд. "Лори", 2012).

[4] Мартин Фаулер, Кент Бек, Джон Брант, Уильям Апдайк, Дон Робертс, Эрих Гамма. *Рефакторинг. Улучшение существующего кода* (пер. с англ., изд. "Символ-Плюс", 2013).

нении тщательного контроля синтаксиса на PHP можно было писать объектно-ориентированные программы.

Тем не менее неприятности, подобные описанной в начале данной главы, случались часто. Культура проектирования была еще не развита, и в книгах по PHP о ней почти не упоминалось. Но в среде Интернета интерес к этому был очевиден. Леон Аткинсон (Leon Atkinson) написал статью о PHP и проектных шаблонах для Zend в 2001 году, а Гарри Фойкс (Harry Fuecks) завел журнал по адресу `www.phppatterns.com` (который уже давно не работает) в 2002 году. Стали появляться проекты на основе шаблонов, такие как BinaryCloud, а также инструменты для автоматического тестирования и документации.

Выход первой бета-версии PHP 5 в 2003 году обеспечил будущее PHP как языка для объектно-ориентированного программирования. Движок Zend 2 обеспечил существенно улучшенную поддержку объектов. И что не менее важно, его появление стало сигналом о том, что объекты и объектно-ориентированное проектирование теперь являются основой PHP-проекта.

Со временем язык PHP 5 продолжал улучшаться и совершенствоваться. В нем появились такие важные средства, как поддержка пространств имен и механизма замыканий. Тогда же PHP 5 завоевал репутацию надежного и самого лучшего средства для создания серверных веб-приложений.

Об этой книге

Данная книга не является попыткой открыть что-то новое в области объектно-ориентированного проектирования; в этом отношении она просто "стоит на плечах гигантов". Цель книги — изучить в контексте PHP некоторые установленные принципы проектирования и некоторые основные шаблоны (особенно те, которые упоминаются в *Design Patterns*, классическом труде "Банды четырех"). И в конце книги я выйду за пределы строгих ограничений кода, чтобы рассмотреть инструменты и методы, которые могут помочь успешному выполнению проекта. За исключением этого введения и краткого заключения, данная книга разделена на еще три основные части: объекты, шаблоны и практика.

Объекты

Часть II начинается с краткого рассмотрения истории PHP и объектов, описывающего их трансформацию из дополнительной возможности в PHP 3 в основную в 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. По этой причине я начинаю книгу с самых основных принципов, чтобы объяснить, что такое объекты, классы и наследование. Даже на этом начальном этапе я рассматриваю некоторые усовершенствования объектов, которые появились в PHP 5.

После изложения основ мы углубимся в тему и изучим более сложные объектно-ориентированные возможности PHP. Я также посвящаю целую главу инструментам, которые предусмотрены в PHP для работы с объектами и классами.

Знаний о том, как объявить класс и как использовать его для создания экземпляра объекта, совсем не достаточно. Вы должны сначала правильно выбрать участников системы и определить оптимальные способы их взаимодействия. Эти выборы намного труднее описать и изучить, чем простые факты об инструментах и синтаксисе объектов. Часть II заканчивается введением в объектно-ориентированное проектирование с помощью PHP.

Шаблоны

Шаблон описывает проблему в проекте программного обеспечения и предоставляет 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 проблемы. При этом может прилагаться пример реализации, но он менее важен, чем концепция, которую он призван иллюстрировать.

Часть III начинается с определения проектных шаблонов и описания их структуры. Я рассмотрю также некоторые причины их популярности.

При создании шаблонов обычно выдвигаются некоторые основополагающие принципы проектирования, которых стараются придерживаться в процессе разработки всего приложения. Понимание этого факта поможет понять причины создания шаблона, который затем можно применять при создании любой программы. Мы обсудим некоторые из этих принципов. Я рассмотрю также унифицированный язык моделирования (Unified Modeling Language — UML), который представляет собой платформенно-независимый способ описания классов и способа их взаимодействия.

Хотя данная книга не является справочником по шаблонам, я рассмотрю некоторые из самых известных и полезных шаблонов. Я опишу проблему, для которой предназначен каждый шаблон, проанализирую решение и представлю пример реализации на PHP.

Практика

Даже самое гармоничное архитектурное сооружение разрушится, если не обращаться с ним должным образом. В части IV я рассмотрю имеющиеся инструменты, с помощью которых можно создать структуру, которая даст гарантию успеха проекта. Во всей книге речь идет о практике проектирования и программирования, и только в части IV — о практике контроля за кодом. Инструменты, которые я рассматриваю, позволяют создать поддерживающую структуру проекта, помогая обнаруживать ошибки по мере их появления, способствуя сотрудничеству программистов и обеспечивая простоту установки и понятность кода.

Я уже говорил об эффективности автоматических тестов. Часть IV я начинаю со вступительной главы, в которой приводится обзор проблем и решений в этой области.

Многие программисты стремятся все делать самостоятельно. Но PHP-сообщество поддерживает PEAR, хранилище качественных пакетов, которые можно легко внедрить в свой проект. Я рассмотрю преимущества и недостатки написания собственной реализации кода и использования PEAR-пакета.

Обсуждая тему PEAR, я рассматриваю механизм инсталляции, благодаря которому развертывание пакета осуществляется с помощью одной команды. Но этот механизм, предназначенный для автономных пакетов, можно использовать для автоматизации инсталляции собственного кода. Я покажу, как это сделать.

Создание документации — скучная работа, и, когда сроки поджимают, именно от этой части проекта, наряду с тестированием, обычно отказываются. Я объясню, почему это ошибка, и покажу, как пользоваться `PHPDocumentor` — утилитой, позволяющей превратить комментарии в коде в набор гипертекстовых HTML-документов, описывающих все элементы вашего API.

Почти каждая утилита или метод, обсуждаемый в данной книге, имеет непосредственное отношение к PHP или развертывается с его помощью. Единственное исключение из этого правила — система контроля версий Git, которая позволяет группе программистов работать совместно над одним и тем же кодом, не рискуя затереть результаты работы друг друга. Она позволяет получить копии файлов с кодом на любом этапе разработки, посмотреть, кто какие изменения внес, и разбить проект на независимые ветки, которые затем можно объединить. В Git сохраняется ежедневное состояние проекта.

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 дежавю. Во-вторых, часто улучшения ломают столько же или даже больше, чем чинят. Автоматическое тестирование помогает решить обе эти проблемы, обеспечивая систему раннего предупреждения о проблемах в коде. Я представляю PHPUnit, мощную реализацию так называемой тестовой платформы xUnit, первоначально предназначенной для Smalltalk, но теперь приспособленной для многих языков, особенно для Java. В частности, я рассматриваю функции PHPUnit, а в целом — преимущества тестирования и ту цену, которую нужно за него заплатить.

PEAR предоставляет средства построения, которые идеально подходят для инсталляции пакетов независимых разработчиков. Но для полного, законченного, приложения необходима большая гибкость. Для приложений характерен беспорядок. Им могут понадобиться файлы, которые нужно устанавливать в нестандартных местах, базы данных или изменение конфигурации сервера. Короче говоря, во время установки приложений нужно *много чего* сделать. Утилита Phing представляет собой точный порт утилиты Ant, используемой в Java-проектах. Phing и Ant интерпретируют файл построения (build file) и обрабатывают исходные файлы определенным вами способом. Чаще всего их нужно скопировать из исходного каталога в различные системные каталоги. Но если ваши потребности выше, то Phing поможет легко удовлетворить и их.

Тестирование и построение проекта — это очень хорошо, но для того чтобы начать пожинать плоды своих трудов, вы должны установить и непрерывно выполнять наборы тестов. Если не автоматизировать процесс построения и тестирования проекта, очень быстро все пойдет прахом. Поэтому в книге я рассмотрю несколько средств и методик, которые, будучи объединенными, называются процессом непрерывной интеграции (continuous integration). Они призваны помочь вам справиться с описанной выше проблемой.

Что нового в 4-м издании

PHP — это живой язык, поскольку 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. Я описал новые средства языка, такие как трейты (traits), директиву `finally`, которая используется при обработке исключений, а также генераторы — простое и мощное средство построения итерируемых классов.

Еще в 1-м издании книги я описал систему модульного тестирования на основе PHPUnit. И хотя в эту часть книги не было внесено никаких изменений, я полностью обновил информацию о системе Selenium, а также об API и наборе средств для тестирования веб-интерфейсов с учетом внесенных в них существенных изменений и дополнений.

Я также обновил главу, посвященную системе контроля версий программного обеспечения, описав Git вместо Subversion. Тем самым я отразил общую тенденцию

перехода на новую платформу, которая наметилась в среде разработчиков с момента выхода 3-го издания книги. Также в книгу включена глава, посвященная непрерывной интеграции. В ней рассмотрены как методики, так и набор средств, позволяющих разработчикам автоматизировать и контролировать процесс построения приложения и стратегии его тестирования. В предыдущем издании книги я описал систему под названием "CruiseControl". На этот раз я остановил свой выбор на сервере непрерывной интеграции Jenkins, который на сегодняшний день широко применяется в сообществе пользователей и разработчиков из-за своей простоты.

Резюме

Это книга об объектно-ориентированном проектировании и программировании. В ней описываются средства для работы с PHP-кодом, начиная от приложений для совместной работы программистов и заканчивая утилитами для развертывания кода.

Эти две темы рассматривают одну и ту же проблему под различными, но дополняющими друг друга углами. Цель состоит в том, чтобы создать системы, выполняющие поставленные цели и обеспечивающие возможности совместной работы над проектами.

Второстепенная цель — эстетичность систем программного обеспечения. Как программисты мы создаем продукт, у которого есть форма и который работает. Мы тратим много часов рабочего времени и много дней жизни, воплощая эти формы в жизнь. Мы создаем нужные нам средства — отдельные классы и объекты, компоненты программного обеспечения или окончательные продукты, — чтобы создать изящное единое целое. Процесс контроля версий, тестирования, документирования и построения представляет собой нечто большее, чем достижение этой цели; это часть формы, которую мы хотим создать. Точно так же, как ясный и понятный код, необходима кодовая база, которая предназначена как для разработчиков, так и для пользователей. И механизм совместного распространения, чтения и развертывания проекта должен иметь такое же значение, как и сам код.

Часть II

Объекты

Глава 2

PHP и объекты

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

Но впоследствии эта идея доказала свою жизнеспособность. В данной главе я познакомлю вас с объектами и опишу процесс разработки объектно-ориентированных приложений на PHP.

Мы рассмотрим следующее.

- *PHP/FI 2.0:* это PHP, но не такой, каким мы его знаем.
- *PHP 3:* первое появление объектов.
- *PHP 4:* объектно-ориентированное программирование развивается.
- *PHP 5:* объекты — в самом сердце языка.
- *PHP 6:* отблеск будущего.

Неожиданный успех PHP-объектов

При таком количестве существующих объектно-ориентированных библиотек и приложений на PHP, не говоря уже об улучшенной поддержке объектов в PHP 5, развитие объектов в PHP может показаться просто кульминацией естественного и неизбежного процесса. На самом деле все это очень далеко от истины.

Вначале был PHP/FI

Своим происхождением PHP, каким мы его знаем сегодня, обязан двум инструментам, которые разработал Расмус Лердорф с помощью Perl. Аббревиатура "PHP" обозначала "Personal Homepage Tools" (средства для персональной домашней страницы), а "FI" — "Form Interpreter" (интерпретатор форм). Вместе они составляли набор макросов 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 именем "PHP/FI 2.0". На этом этапе синтаксис языка PHP отличался от того, который мы знаем сегодня, но не *слишком* значительно. Была поддержка переменных, ассоциативных массивов и функций. Но объектов не было еще и в помине.

Изменение синтаксиса в PHP 3

На самом деле, даже когда PHP был еще на этапе планирования, объекты не стояли на повестке дня. Главными архитекторами PHP 3 были Зеэв Сураски и Энди Гутманс. PHP 3 представлял собой полностью переписанный PHP/FI 2.0, но тогда объекты 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. Однако все еще существовали жесткие ограничения в отношении операций с классами. В частности, нельзя было получить доступ к переопределенным методам родительского класса (если вы еще не знаете, что это такое, не волнуйтесь; я объясню позже). Еще одним недостатком, о котором мы поговорим в следующем разделе, был не самый оптимальный способ передачи объектов в PHP-сценариях.

В то время объекты считались второстепенным вопросом. Об этом говорило и то, что им придавалось мало значения в официальной документации. В руководстве были даны одно предложение о них и один пример кода. В примере не иллюстрировалось использование наследования или свойств.

PHP 4 и тихая революция

Хотя PHP 4 был еще одним революционным шагом в развитии языка, большинство основных изменений произошло незаметно для пользователя на нижнем уровне. Для расширения возможностей языка PHP был заново переписан движок Zend, название которого происходит от имен **Ze**ev и A**nd**i. Zend — один из основных компонентов, положенных в основу работы PHP. Любая вызываемая PHP-функция на самом деле является частью яруса высокоуровневых расширений. Эти функции выполняют всю возложенную на них работу, например взаимодействие с API системы управления базами данных или манипуляции строками. А на нижнем уровне движок Zend управляет памятью, передает управление другим компонентам и преобразует знакомый вам PHP-синтаксис, с которым вы работаете каждый день, в выполняемый байт-код. Именно движок Zend мы должны "благодарить" за поддержку ключевых возможностей языка, таких как классы.

С точки зрения рассматриваемых нами *объектов* большим преимуществом PHP 4 стала возможность переопределения родительских методов и получения к ним доступа из дочерних классов.

Но остался и большой недостаток. Присвоение объекта переменной, передача его функции или возвращение его из метода приводило к появлению копий этого объекта. Поэтому присвоение, подобное

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

приводило к появлению двух копий объекта `User`, а не двух ссылок на один и тот же объект `User`. В большинстве объектно-ориентированных языков программирования используется более естественное присвоение по ссылке, а не по значению, как здесь. Это означает, что вы передаете и присваиваете указатели на объекты, а не копируете сами объекты. Принятая по умолчанию стратегия передачи по значению приводила 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 этой книге вы увидите много примеров, в которых будет использоваться несколько ссылок на один и тот же объект.

К счастью, в PHP-программе можно было принудительно осуществить передачу объекта по ссылке, но для этого нужно было использовать довольно "неуклюжую" конструкцию.

Выполним присвоение по ссылке следующим образом.

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

Выполним передачу параметра функции по ссылке следующим образом.

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

И осуществим возврат по ссылке.

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

Хотя эта конструкция работала нормально, программисты легко забывали добавить символ амперсанда, и вероятность появления ошибок в объектно-ориентированном коде была очень высока. Причем такие ошибки очень трудно было обнаружить, потому что они редко вызывали сообщения об ошибках; только программа работала не совсем так, как нужно.

Описание синтаксиса в целом и объектов в частности было расширено в руководстве по PHP, и объектно-ориентированное программирование стало превращаться в основное направление, в главную тенденцию. Объекты в PHP не были приняты сообществом программистов без споров, и сообщения типа "Зачем мне нужны эти объекты?" часто раздували флеймы на форумах и в списках рассылки. На сайте Zend размещались статьи, которые поощряли объектно-ориентированное программирование, наряду со статьями, в которых звучали предостережения.

Но несмотря на проблемы передачи по ссылке и споры, многие программисты просто приняли новые возможности и "усеяли" свои коды символами амперсандов. Популярность объектно-ориентированного PHP стала расти. Вот что Зеэв Сураски написал в статье для сайта DevX.com (`http://www.devx.com/webdev/Article/10007/0/page/1`).

> *Одним из величайших неожиданных поворотов в истории PHP было то, что, несмотря на очень ограниченную функциональность, на множество проблем и ограничений, объектно-ориентированное программирование в PHP процветало и стало самой популярной парадигмой для растущего числа стандартных PHP-приложений. Эта тенденция, которая по большей части была неожиданной, застала PHP в не самой выигрышной ситуации. Стало очевидным, что объекты ведут себя не так, как объекты в других объектно-ориентированных языках, а как [ассоциативные] массивы.*

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 программного обеспечения PHP, PEAR, также была принята концепция объектно-ориентированного программирования. Некоторые из лучших примеров использования шаблонов объектно-

ориентированного проектирования можно найти в пакетах PEAR, созданных для расширения функциональности PHP.

Оглядываясь в прошлое, можно подумать, что введение в PHP поддержки средств объектно-ориентированного программирования стало результатом вынужденной капитуляции перед лицом неизбежности. Но важно помнить, что хотя концепция объектно-ориентированного программирования существует с 60-х годов прошлого века, широкое распространение она получила только в средине 90-х годов. Язык Java, этот "великий популяризатор" методологии объектно-ориентированного программирования, был выпущен только в 1995 году. Язык C++, расширение процедурного языка C, появился в 1979 году. И после длительного этапа эволюции он совершил большой скачок только в 90-х годах. В 1994 году вышел язык Perl 5. Это была еще одна революция для бывшего процедурного языка, что дало возможность пользователям перейти к концепции объектов (хотя некоторые утверждают, что поддержка объектно-ориентированных возможностей в Perl напоминает добавления, 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 PHP 5 уже была явно выражена поддержка объектов и объектно-ориентированного программирования. Это не означает, что теперь объекты — единственное средство работы с PHP (кстати, настоящая книга тоже этого не утверждает). Но объекты теперь считаются мощным и важным средством разработки корпоративных приложений, и в PHP предусмотрена их полная и всесторонняя поддержка.

Возможно, одним из самых важных следствий расширения языка PHP 5 явилось то, что он был принят в качестве основного большими интернет-компаниями. Например, такие компании, как Yahoo! и Facebook, в значительной степени использовали PHP для создания своих программных платформ. С появлением версии 5 язык PHP стал одним из стандартных языков для разработки корпоративных веб-приложений в Интернете.

Объекты из дополнительной возможности превратились в основу языка. И вероятно, самое важное изменение — это принятый по умолчанию тип передачи объектов по ссылке, а не по значению. Но это было только начало. По всей книге, и особенно в этой части, описано гораздо больше изменений, которые расширяют и усиливают поддержку объектов в PHP, включая уточнения типов аргументов (hints), закрытые (private) и защищенные (protected) методы и свойства, ключевое слово `static`, пространства имен, исключения и многое другое.

PHP остается языком, который поддерживает объектно-ориентированную разработку, а не языком объектно-ориентированного программирования. Но поддержка в нем объектов развита достаточно для того, чтобы стало оправданным появление книг, подобных этой и посвященных проектированию исключительно с объектно-ориентированной точки зрения.

Взгляд в будущее

На момент написания этой книги важная с психологической точки зрения версия PHP 6 отошла на второй план, поскольку была выпущена версия PHP 5.5. В своем интервью по поводу плана выпуска PHP 6 в 2012 году Энди Гутманс сказал следующее (http://venturebeat.com/2012/10/24/zends-andi-gutmans-on-php-6-being-a-developer-ceo-and-how-apple-is-the-biggest-barrier-to-the-future-of-mobile/).

Прямо сейчас не существует никакого плана, поскольку PHP-сообщество не всегда придерживается каких-либо графиков. Сейчас мы работаем над версией 5.5, но решение о том, когда наступит время версии 6, зависит от количества реализованных в ней нами новых возможностей.

Несмотря на то что мы все ожидаем выхода версии PHP 6, должен сказать, что как минимум в трех из четырех последних изданий этой книги были описаны предполагаемые новинки версии 6, которые затем, как правило, появлялись в основных выпусках новых версий PHP 5. Например, в PHP 5.3 были реализованы пространства имен. Они позволили ограничить область видимости имен классов и функций, чтобы уменьшить вероятность дублирования имен при включении библиотек и расширении системы с помощью пакетов. Благодаря пространствам имен также удалось избавиться от уродливых, но необходимых правил именования объектов, подобных следующему.

```
class megaquiz_util_Conf {
}
```

Подобные имена классов — средство предотвращения конфликтов между пакетами, но следование этому правилу еще больше запутывало код.

Мы также говорили о поддержке замыканий (closures), генераторов, трейтов и позднего статического связывания (late static binding).

Когда я писал эти строки, 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.

Сторонники и противники: дебаты об объектах

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

В данной книге повсюду демонстрируется пристрастие к объектно-ориентированному программированию, поскольку это является отражением моего мировоззрения, "зараженного" любовью к объектам. Поскольку эта книга посвящена объектам и является введением в объектно-ориентированное проектирование, нет ничего удивительного в том, что главное внимание в основном уделяется объектно-ориентированным методам. Но в то же время в книге нигде не утверждается, что объекты — это единственно правильный путь к успеху в программировании на PHP.

Во время чтения книги стоит иметь в виду 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 создать структуру, которая сможет эффективно и безболезненно вырасти в большую систему (в мире экстремального программирования наспех разрабатываемые проекты такого рода называют "пробными решениями").

Код — это гибкая среда. Самое главное — понять, когда быстрое испытание идеи станет основанием для дальнейшего развития, и вовремя остановиться, прежде чем решения по вопросам проектирования вам будет диктовать громадный объем кода. И если вы примете решение использовать для развивающегося проекта процедурно-ориентированный подход, то найдете множество книг, в которых приведены примеры процедурного проектирования для разнообразных проектов. В данной книге предлагаются некоторые идеи проектирования с помощью объектов. Надеюсь, она станет для вас хорошей отправной точкой.

Резюме

В этой короткой главе объекты рассмотрены в контексте языка PHP. Будущее PHP во многом связано с объектно-ориентированным проектированием. В следующих нескольких главах мы рассмотрим текущее состояние поддержки объектов в PHP и обсудим некоторые вопросы проектирования.

Глава 3

Основные сведения об объектах

В данной книге основное внимание уделяется объектам и классам, поскольку с появлением PHP 5 они стали основными элементами языка. В этой главе мы создадим основу для дальнейшей работы с объектами и проектами, изучив основные объектно-ориентированные возможности PHP.

В PHP 5 радикально изменилась поддержка объектно-ориентированного программирования, и, если вы уже знакомы с PHP 4, то, вероятно, найдете в этой главе нечто новое. Если объектно-ориентированное программирование — новая для вас область, постарайтесь очень внимательно прочитать эту главу.

В данной главе рассматриваются следующие темы.

- *Классы и объекты*: объявление классов и создание экземпляров объектов.
- *Методы конструктора*: автоматизация установок начальных значений объектов.
- *Элементарные типы и классы*: почему тип имеет значение.
- *Наследование*: зачем нужно наследование и как его использовать.
- *Видимость*: упрощение интерфейсов объектов и защита методов и свойств от вмешательства извне.

Классы и объекты

Первое препятствие для понимания объектно-ориентированного программирования — это 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 основам.

Первый класс

Классы часто описывают с помощью объектов. Это очень интересно, потому что объекты часто описывают с помощью классов. Это хождение по кругу может сделать очень трудными первые шаги в объектно-ориентированном программировании. Поскольку именно классы определяют объекты, мы должны начать с определения классов.

Если говорить кратко, то *класс* — это шаблон кода, который используется для создания объектов. Класс объявляется с помощью ключевого слова `class` и произвольного имени класса. В именах классов может использоваться любое сочетание букв и цифр, но они не должны начинаться с цифры. Код, связанный с классом, должен быть заключен в фигурные скобки. Давайте объединим эти элементы, чтобы построить класс.

```
class ShopProduct {
   // Тело класса
}
```

Класс `ShopProduct` из этого примера — уже полноправный класс, хотя пока и не слишком полезный. Тем не менее мы сделали нечто очень важное. Мы определили тип, т.е. создали категорию данных, которые можем использовать в своих сценариях. Важность этого станет для вас очевидной по мере дальнейшего чтения главы.

Первые несколько объектов

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

Мы будем использовать класс `ShopProduct` как форму для создания объектов типа `ShopProduct`. Для этого нам нужен оператор `new`. Он используется совместно с именем класса следующим образом.

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

После оператора `new` указывается имя класса в качестве его единственного операнда. В результате он создает экземпляр этого класса; в нашем примере создается объект типа `ShopProduct`.

Итак, мы использовали класс `ShopProduct` как шаблон для создания двух объектов типа `ShopProduct`. Хотя функционально они идентичны (т.е. пусты), `$product1` и `$product2` — это два разных объекта одного типа, созданные с помощью одного класса.

Если вам все еще не понятно, давайте приведем такую аналогию. Представьте, что класс — это форма для отливки, с помощью которой изготавливают пластмассовые утки. Объекты — это и есть утки. Тип создаваемых объектов определяется формой отливки. Утки выглядят одинаковыми во всех отношениях, но все-таки это разные предметы. Другими словами, это разные экземпляры одного и того же типа. У уток могут быть даже разные серийные номера, подтверждающие их индивидуальность. Каждому объекту, создаваемому в PHP-сценарии, также присваивается уникальный идентификатор (он уникален на время жизни объекта, т.е. в PHP повторно используются идентификаторы, даже в пределах одного и того же процесса (т.е. запуска сценария)). Это можно продемонстрировать, выведя на печать объекты `$product1` и `$product2`.

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

После выполнения этих функций будут выведены следующие данные.

```
object(ShopProduct)#1 (0) {
}
object(ShopProduct)#2 (0) {
}
```

На заметку. В PHP 4 и PHP 5 (до версии 5.1 включительно) можно выводить объекты на печать непосредственно. В результате объект будет преобразован в строку, содержащую идентификатор объекта. Начиная с версии PHP 5.2 и выше такая возможность больше не поддерживается, и любая попытка рассматривать объект как 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, "Расширенные средства".

Передав наши объекты функции `var_dump()`, мы можем узнать о них полезную информацию, включая внутренний идентификатор каждого объекта, указанный после символа `'#'`.

Чтобы сделать эти объекты более интересными, мы должны немного изменить определение класса `ShopProduct`, добавив в него специальные поля данных, называемые *свойствами* (properties).

Определение свойств в классе

В классах можно определять специальные переменные, которые называются свойствами. Свойство, которое называется также *переменной-членом* (member variable), содержит данные, которые могут изменяться от одного объекта к другому. В случае объектов `ShopProduct` нам нужно иметь возможность изменять, например, 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 нужно поместить одно из ключевых слов, характеризующих область его видимости: `public`, `protected` или `private`.

На заметку. Область видимости (scope) определяет контекст функции или класса, в котором можно пользоваться данной переменной (или методом, о чем мы поговорим далее в этой главе). Так, переменная, определенная внутри тела функции, имеет локальную область видимости, а переменная, определенная за пределами функции, — глобальную область видимости. Как правило, нельзя получить доступ к данным, находящимся в локализованных областях видимости по отношению к текущей области. Поэтому, определив переменную внутри функции, вы впоследствии не сможете получить к ней доступ извне этой функции. Объекты в этом смысле более "проницаемы", и к некоторым объектным переменным можно иногда получать доступ из другого контекста. К каким переменным можно получать доступ и из какого контекста, и определяют ключевые слова `public`, `protected` и `private`, как мы вскоре увидим.

К этим ключевым словам и вопросу видимости мы вернемся несколько позже в этой главе. А сейчас давайте определим некоторые свойства с помощью ключевого слова `public`.

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

Как видите, мы определили четыре свойства, присвоив каждому из них стандартное значение. Теперь любым объектам, экземпляры которых мы будем создавать с помощью класса `ShopProduct`, будут присвоены стандартные данные. А ключевое слово `public`, присутствующее в объявлении каждого свойства, обеспечит доступ к этому свойству извне контекста объекта.

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

На заметку. Ключевые слова `public`, `protected` и `private`, определяющие область видимости свойств, появились в PHP 5. В версии PHP 4 приведенные выше примеры работать не будут. В PHP 4 все свойства должны быть объявлены с помощью ключевого слова `var`, что, по сути, идентично использованию ключевого слова `public`. Исходя из принципа обратной совместимости, в PHP 5 допускается использование для свойств ключевого слова `var` вместо `public`.

К переменным свойств можно обращаться с помощью символов `'->'`, указав имя объектной переменной и имя свойства.

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

В результате будет выведено следующее.

```
Стандартный товар
```

Поскольку свойства объектов были определены как `public`, мы можем считывать их значения, а также присваивать им новые значения, заменяя тем самым набор стандартных значений, определенный в классе.

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

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

Объявляя и определяя свойство `$title` в классе `ShopProduct`, мы гарантируем, что при создании любого объекта типа `ShopProduct` это свойство будет присутствовать и его значение будет заранее определено. Это означает, что при таком предположении в коде, где используется данный класс, можно будет работать с любыми объектами типа `ShopProduct`. Но поскольку мы можем легко переопределить это свойство, значение `$title` может изменяться от одного объекта к другому.

На заметку. Код, в котором используется класс, функция или метод, обычно называют клиентом класса, функции или метода либо просто клиентским кодом. В последующих главах вы будете часто встречать этот термин.

На самом деле в PHP необязательно объявлять все свойства в классе. Свойства можно динамически добавлять к объекту следующим образом.

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

Но нужно отметить, что этот способ присвоения свойств объектам считается дурным тоном в объектно-ориентированном программировании и почти никогда не используется.

Но почему динамическое определение свойств — это плохо? Создавая класс, вы определяете тип. Вы сообщаете всем о том, что ваш класс (и любой объект, который является его экземпляром) содержит определенный набор полей и функций. Если в классе `ShopProduct` определяется свойство `$title`, то любой код, который работает с объектами типа `ShopProduct`, может исходить из предположения, что свойство `$title` определено. Но подобной гарантии относительно свойств, определенных динамически, не существует.

На данном этапе наши объекты пока производят довольно тягостное впечатление. Когда нам понадобится работать со свойствами объекта, придется делать это извне объектов. Мы пришли к тому, что нужно как-то задавать и получать значения свойств объекта. Определение нескольких свойств для нескольких объектов часто становится довольно неприятной задачей.

```
$product1                    = new ShopProduct();

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

Здесь мы снова используем класс `ShopProduct`, переопределяя один за другим все стандартные значения его свойств, пока не определим всю информацию о товаре. А теперь, когда мы определили некоторые данные, можно к ним обратиться.

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

В результате на выходе получим следующее.

```
Автор: Михаил Булгаков
```

У этого подхода к определению значений свойств есть несколько недостатков. Поскольку PHP позволяет определять свойства динамически, вы не получите предупреждения, если забудете, как называется имя свойства, или сделаете в нем опечатку. Например, можно ошибочно записать строку

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

как

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

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

Еще одна проблема — наши объекты, в целом, слишком "нестрогие". Мы не обязаны определять название книги, цену или имя автора. Клиентский код может быть уверен, что эти свойства существуют, но, вполне вероятно, очень часто их стандартные значения не будут вас устраивать. В идеале следовало бы заставлять всякого, кто создает экземпляры объекта `ShopProduct`, определять осмысленные значения его свойств.

И наконец мы должны потратить уйму сил, чтобы сделать то, что, вероятно, придется делать очень часто. Вывести полное имя автора — это довольно трудоемкий и нудный процесс.

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

И было бы прекрасно, если бы объект делал это вместо нас.

Все эти проблемы можно решить, если снабдить объект `ShopProduct` собственным набором функций, которые можно использовать для выполнения операций над данными внутри контекста объекта.

Работа с методами

Так же, как свойства позволяют объектам сохранять данные, методы позволяют объектам выполнять задачи. *Методы* (methods) — это специальные функции, которые объявляются внутри класса. Как и можно было ожидать, объявление метода напоминает объявление функции. За ключевым словом `function` следует имя метода, а за ним — необязательный список переменных-аргументов в круглых скобках. Тело метода заключается в фигурные скобки.

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

В отличие от функций, методы необходимо объявлять в теле класса. При этом можно также указывать 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 извне текущего объекта. Если в определении метода вы опустите ключевое слово, определяющее видимость, то метод будет объявлен неявно как `public`. К модификаторам методов мы вернемся позже в этой главе.

На заметку. PHP 4 не распознает ключевые слова, определяющие видимость, для методов и свойств. Добавление к объявлению метода ключевых слов `public`, `protected` и `private` приведет к неустранимой ошибке. Все методы в PHP 4 неявно относятся к типу `public`.

В большинстве случаев метод вызывают с помощью объектной переменной, за которой указываются символы `'->'` и имя метода. При вызове метода нужно использовать круглые скобки, так же как при вызове функции (даже если методу не передаются никакие аргументы).

```
$myObj = new MyClass();
$myObj->myMethod( "Михаил", "Булгаков" );
```

Давайте добавим методы к определенному ранее классу `ShopProduct`.

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

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

}

$product1 = new ShopProduct();
 $product1->title             = "Собачье сердце";
 $product1->producerMainName  = "Булгаков";
 $product1->producerFirstName = "Михаил";
 $product1->price             = 5.99;

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

В результате на выходе получим следующее.

```
Автор: Михаил Булгаков
```

Мы добавили метод `getProducer()` к классу `ShopProduct`. Обратите внимание на то, что при определении метода мы не использовали ключевое слово, определяющее его видимость. Это означает, что метод `getProducer()` относится к типу `public` и его можно вызвать из-за пределов класса.

При определении метода `getProducer()` мы воспользовались новой возможностью — псевдопеременной `$this`. Она представляет собой механизм, посредством которого из кода класса можно обратиться к экземпляру объекта. Если вы считаете,

что это трудно для понимания, попробуйте заменить `$this` "текущим экземпляром объекта". Тогда оператор

```
$this->producerFirstName
```

превратится в

```
Свойство $producerFirstName текущего экземпляра объекта
```

Так, метод `getProducer()` объединяет и возвращает значения свойств `$producer FirstName` и `$producerMainName`, избавляя нас от неприятной работы всякий раз, когда нужно вывести полное имя автора.

Итак, нам удалось немного улучшить наш класс. Но для него по-прежнему характерна слишком большая "гибкость". Мы полагаемся на то, что программист будет изменять стандартные значения свойств объекта `ShopProduct`. Но это проблематично в двух отношениях. Во-первых, нужно пять строк кода, чтобы должным образом инициализировать объект типа `ShopProduct`, и ни один программист вам не скажет за это "спасибо". Во-вторых, у нас нет способа гарантировать, что какое-либо свойство будет определено при инициализации объекта `ShopProduct`. Поэтому нам нужен метод, который будет вызываться автоматически при создании экземпляра объекта на основе класса.

Создание метода конструктора

Метод конструктора вызывается при создании объекта. Его можно использовать, чтобы все настроить, обеспечить определенные значения необходимых свойств и выполнить всю предварительную работу. До PHP 5 имя метода конструктора совпадало с именем класса, к которому оно относилось. Так, класс `ShopProduct` мог использовать метод `ShopProduct()` в качестве своего конструктора. В PHP 5 вы должны назвать метод конструктора `__construct()`. Обратите внимание на то, что имя метода начинается с двух символов подчеркивания. Это правило наименования действует для многих других специальных методов в PHP-классах. Давайте определим конструктор для класса `ShopProduct`.

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

   function __construct( $title,
                         $firstName, $mainName, $price ) {

      $this->title             = $title;
      $this->producerFirstName = $firstName;
      $this->producerMainName  = $mainName;
      $this->price             = $price;
   }

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

И снова мы добавляем к классу функциональность, стараясь сэкономить время и силы программиста и избавить его от необходимости дублирования кода, работаю-

щего с этим классом. Метод `__construct()` вызывается, когда создается объект с помощью оператора `new`.

```
$product1 = new ShopProduct( "Собачье сердце",
                             "Михаил", "Булгаков", 5.99 );
print "Автор: {$product1->getProducer()}\n";
```

В результате получаем следующее.

```
Автор: Михаил Булгаков
```

Значения всех перечисленных аргументов передаются конструктору. Так, в нашем примере мы передаем конструктору название произведения, имя и фамилию автора, а также цену. В методе конструктора используется псевдопеременная `$this` для присвоения значений соответствующим свойствам объекта.

На заметку. В PHP 4 метод `__construct()` не распознается в качестве конструктора. Если вы используете PHP 4, то для создания конструктора объявите метод, имя которого совпадает с именем содержащего его класса. Поэтому для класса с именем `ShopProduct` можно объявить конструктор с помощью метода под названием `ShopProduct()`.

В PHP по-прежнему поддерживается эта схема именования конструктора. Но если вам не нужна совместимость со старыми версиями PHP, то методы конструктора лучше называть `__construct()`.

Теперь стало безопаснее использовать объект `ShopProduct` и легче создавать экземпляры на основе его класса. Создание экземпляров и определение значений свойств выполняются в одном операторе. При написании любого кода, в котором используется объект `ShopProduct`, можно быть уверенным, что все свойства этого объекта будут инициализированы.

Очень важным аспектом объектно-ориентированного программирования является предсказуемость. Вы должны так разрабатывать классы, чтобы пользователи объектов могли легко догадаться об их функциональных возможностях. Кроме того, при использовании объекта вы должны быть уверены в его типе. В следующем разделе мы изучим механизм, который можно использовать для явного определения типов объектов при объявлении методов.

Аргументы и типы

Типы определяют, каким способом можно оперировать 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 используются в математических выражениях, булевы числа — в логических выражениях и т.д. Эти категории называются элементарными типами данных. Класс также определяет тип имени себя, но на более высоком уровне. Поэтому объект `ShopProduct` относится к элементарному типу `object`, а также к типу класса `ShopProduct`. В данном разделе мы рассмотрим обе разновидности типов в отношении методов класса.

При определении метода и функции не требуется, чтобы аргумент был определенного типа. Но это одновременно и преимущество, и недостаток. То, что аргумент может быть любого типа, дает широкое поле действий. Благодаря этому можно создавать методы, которые будут гибко обрабатывать данные различных типов и приспосабливать свою функциональность к меняющимся обстоятельствам. Но эта гибкость, с другой стороны, может стать причиной неопределенности в коде, когда тело метода ожидает один тип аргумента, а получает — другой.

Элементарные типы

PHP является слабо типизированным языком. Это означает, что нет необходимости объявлять тип данных, который должна хранить переменная. Так, в пределах одной и той же области видимости переменная `$number` может содержать как значение `2`, так и строку `"two"` ("два"). В строго типизированных языках программирования, таких как C или Java, вы обязаны определить тип переменной, прежде чем присваивать ей значение, и, конечно, это значение должно быть указанного типа.

Но это не означает, что в PHP нет понятия типа. Каждое значение, которое можно присвоить переменной, имеет свой тип. Вы можете определить тип значения переменной с помощью одной из функций проверки типов языка PHP. В табл. 3.1 перечислены элементарные типы данных, используемые в PHP, и соответствующие им функции проверки. Каждой функции передается переменная или значение, а она возвращает значение `true` ("истина"), если аргумент относится к соответствующему типу.

Таблица 3.1. Элементарные типы данных и функции проверки в PHP

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

Проверка типа переменной особенно важна, когда вы работаете с аргументами в методе или функции.

Пример использования элементарных типов данных

Имейте в виду, что вы должны внимательно следить за типами данных в коде. Давайте рассмотрим пример одной из многих проблем, связанных с типами, с которой вы можете столкнуться.

Предположим, вам нужно извлечь параметры конфигурации из XML-файла. XML-элемент `<resolvedomains>` говорит приложению о том, следует ли пытаться преобразовывать IP-адреса в доменные имена. Надо отметить, что такое преобразование полезно, однако отнимает много времени. Вот фрагмент XML-кода.

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

Приложение извлекает строку `"false"` и передает ее в качестве параметра методу `outputAddresses()`, который выводит данные IP-адресов. Вот определение метода `outputAddresses()`.

```
class AddressManager {
    private $addresses = array( "209.131.36.159", "74.125.19.106" );

    function outputAddresses( $resolve ) {
```

```
        foreach ( $this->addresses as $address ) {
           print $address;
           if ( $resolve ) {
              print " (" . gethostbyaddr( $address ) . ")";
           }
           print "\n";
        }
    }
}
```

Разумеется, в код класса `AddressManager` можно внести ряд улучшений. Например, очень неудобно кодировать IP-адреса в виде массива внутри кода класса. Несмотря на это метод `outputAddresses()` циклически проходит по массиву IP-адресов и выводит его элементы один за другим. Если значение аргумента `$resolve` равно `true`, то метод, кроме IP-адресов, выводит также доменные имена.

Ниже приведен один из возможных вариантов использования класса `AddressManager` совместно с тегом `settings` XML-файла конфигурации. Сможете ли вы сходу определить ошибку в приведенном ниже коде?

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

В этом фрагменте кода для получения значения элемента `resolvedomains` используется SimpleXML API, который появился в PHP 5. В нашем примере мы знаем, что это значение — текстовый элемент `"false"`, и преобразуем его в строку, поскольку так советуют в документации по SimpleXML.

Но этот код будет работать не так, как вы того ожидаете. Передавая строку `"false"` методу `outputAddresses()`, мы точно не знаем, какой тип аргумента используется в методе по умолчанию. Данный метод ожидает булеву величину, которая может принимать значение `true` (истина) или `false` (ложь). При выполнении условного оператора строка `"false"` на самом деле превратится в значение `true`. Все дело в том, что при выполнении проверки интерпретатор PHP "заботливо" преобразует непустое строковое значение в булево значение `true`. Поэтому

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

эквивалентно следующему.

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

Исправить эту ситуацию можно по-разному. Во-первых, можно сделать метод `outputAddresses()` менее требовательным, чтобы он распознавал строку и применял некоторые основные правила для ее преобразования в булев эквивалент.

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

```
    // ...
  }
```

Тем не менее с точки зрения проектирования существуют достаточно веские причины избегать приведенного выше решения. В сущности, лучше всего при проектировании метода или функции предусматривать для них строгий интерфейс, избегающий двусмысленности, и не заниматься реализацией мутного и всепрощающего функционала. Подобные решения рано или поздно вызывают путаницу и становятся источником ошибок.

Во-вторых, можно оставить метод `outputAddresses()` таким, как есть, и включить комментарий с четкими инструкциями о том, что аргумент `$resolve` должен содержать булево значение. Этот подход, по сути, говорит программисту о том, что нужно внимательно читать инструкции или пенять на себя.

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

Это вполне разумное решение, если вы точно уверены в том, что программисты, использующие ваш класс, добросовестно прочтут документацию к нему.

И наконец, в-третьих, можно сделать метод `outputAddresses()` строгим в отношении типа данных аргумента `$resolve`.

```
function outputAddresses( $resolve ) {
  if ( ! is_bool( $resolve ) ) {
    die( "Методу outputAddress() требуется булев аргумент\n" );
  }
 //...
}
```

Этот подход заставляет клиентский код обеспечить корректный тип данных для аргумента `$resolve`. Преобразование строкового аргумента в клиентском коде — более дружественный подход, но, вероятно, он может вызвать другие проблемы. Обеспечивая механизм преобразования в методе, мы предугадываем контекст его использования и намерение клиента. С другой стороны, навязывая булев тип данных, мы предоставляем клиенту право выбора — преобразовывать ли строки в булев тип и какое слово какому значению будет соответствовать. А между тем метод `outputAddresses()` разрабатывался с целью решить конкретную задачу. Этот акцент *на выполнении конкретной задачи при намеренном игнорировании более широкого контекста* — важный принцип объектно-ориентированного программирования, к которому мы будем часто возвращаться в этой книге.

На самом деле стратегии работы с типами аргументов зависят от степени серьезности возможных ошибок. PHP автоматически преобразовывает большинство элементарных значений в зависимости от контекста. Например, если числа в строках используются в математических выражениях, то они преобразуются в эквиваленты целых значений или значений с плавающей точкой. В результате код будет "снисходителен" к некорректному типу аргумента. Но если вы рассчитываете, что один из аргументов метода будет массивом, то нужно проявить большую осторожность.

Передача "немассивного" значения одной из функций массивов PHP не приведет к хорошим результатам и вызовет ряд ошибок в методе.

Поэтому, вполне вероятно, вы примените свой подход и найдете нечто среднее между проверкой типа, преобразованием из одного типа в другой и надеждой на хорошую, понятную документацию (причем документацию вы должны предоставить независимо от того, какой способ предпочтете).

Как бы вы ни решали проблемы подобного рода, можете быть уверены в одном — тип аргумента всегда имеет значение! То, что язык PHP не является строго типизированным, делает эту проблему еще более важной. Нельзя полагаться на компилятор в вопросе предотвращения ошибок, связанных с типом. Поэтому вы должны предусмотреть возможные последствия того, что типы аргументов окажутся не такими, как ожидалось. Не стоит надеяться, что программисты клиентского кода будут читать ваши мысли. И вы всегда должны учитывать, как в создаваемых вами методах будут обрабатываться входные данные, имеющие некорректный тип.

Уточнения типов объектов

Мы уже говорили, что переменная-аргумент может содержать любой элементарный тип данных, однако по умолчанию ее тип не оговаривается, поэтому она может содержать объект любого типа. Такая гибкость, с одной стороны, полезна, но, с другой, может стать причиной проблем при определении метода.

Рассмотрим метод, предназначенный для работы с объектом типа `ShopProduct`.

```
class ShopProductWriter {
   public function write( $shopProduct ) {
      $str = "{$shopProduct->title}: "
              .$shopProduct->getProducer()
          ." ({$shopProduct->price})\n";
      print $str;
   }
}
```

Мы можем протестировать этот класс следующим образом.

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

Тогда на выходе получим следующее.

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

Класс `ShopProductWriter` содержит единственный метод, `write()`. Методу `write()` передается объект типа `ShopProduct`. В нем используются свойства и методы последнего для построения и вывода результирующей строки описания товара. Мы используем имя переменной-аргумента, `$shopProduct`, как напоминание программисту о том, что методу `$write()` нужно передать объект типа `ShopProduct`. Но это требование не является обязательным. Это значит, что я могу передать некорректный объект или элементарный тип методу `$write()` и ничего об этом не узнать до момента обращения к аргументу `$shopProduct`. К тому времени в нашем коде уже могут быть выполнены какие-либо действия так, как если бы мы передали методу настоящий объект типа `ShopProduct`.

На заметку. Возможно, вас заинтересует, почему мы не добавили метод `write()` непосредственно в класс `ShopProduct`. Вообще говоря, проектным решениям подобного рода, в числе прочего, посвящена и эта глава, и целая книга. А ответом на поставленный вопрос будет следующее: все дело в ответственности. Класс `ShopProduct` ответственен за хранение данных о товаре, а класс `ShopProductWriter` — за вывод этих данных. По мере чтения этой главы вы начнете понимать, в чем польза такого разделения ответственности.

Для решения описанной проблемы в PHP 5 появилась новая возможность — уточнение типов данных класса. Чтобы добавить уточнение типа к аргументу метода, просто поместите перед ним имя класса. Поэтому метод `write()` можно изменить следующим образом.

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

Теперь методу `write()` можно передавать аргумент `$shopProduct`, содержащий только объект типа `ShopProduct`. Давайте попробуем вызвать метод `write()` для такого "хитрого" объекта.

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

Поскольку метод `write()` содержит уточнение типа класса, передача ему объекта `Wrong` приведет к неустранимой ошибке.

```
PHP Catchable fatal error: Argument 1 passed to ShopProductWriter::write()
 must be an instance of ShopProduct, instance of Wrong given, ...[2]
```

Теперь нам не нужно каждый раз при вызове метода проверять тип передаваемого ему аргумента. Кроме того, использование уточнений делает запись метода намного понятнее для программиста клиентского кода. Он сразу же увидит требования метода `write()`. Программисту не нужно будет волноваться по поводу незаметных ошибок, возникающих в результате несоответствия типов аргументов, поскольку благодаря уточнению они определяются принудительно и строго.

Хотя автоматическая проверка типов — это превосходный способ предотвращения ошибок, важно понимать, что уточнения проверяются во время выполнения программы. Это означает, что уточнение класса сообщит об ошибке только тогда, когда нежелательный объект будет передан методу. И если вызов метода `write()` находится где-то глубоко в условном операторе, который запускается только на Рождество, то, если вы тщательно не проверили код, вам придется работать на праздники.

Уточнения нельзя использовать для принудительного определения аргументов элементарных типов, таких как строки и целые значения. Для этой цели в теле методов следует использовать функции проверки типов, такие как `is_int()`. Но можно принудительно определить, что аргумент является массивом.

```
function setArray( array $storearray ) {
   $this->array = $storearray;
}
```

Поддержка уточнения для массивов была добавлена в PHP начиная с версии 5.1. В дальнейшем также была добавлена поддержка нулевых стандартных значений в

[2] Неустранимая и обрабатываемая ошибка PHP: аргумент 1, переданный методу `ShopProductWriter::write()` должен быть экземпляром класса `ShopProduct`, переданный экземпляр класса `Wrong` некорректен...

аргументах с уточнениями. Это означает, что можно требовать, чтобы аргумент был либо определенного типа, либо нулевым значением. Вот как это сделать.

```
function setWriter( ObjectWriter $objwriter=null ) {
   $this->writer = $objwriter;
}
```

До сих пор мы обсуждали типы и классы так, как будто это синонимы. Но между ними есть коренное различие. При определении класса вы определяете также и тип, но тип может описывать целое семейство классов. Механизм, посредством которого различные классы можно группировать под одним типом, называется наследованием. О наследовании мы и поговорим в следующем разделе.

Наследование

Наследование — это механизм, посредством которого один или несколько классов можно получить из некоторого базового класса.

Класс, который получается в результате наследования от другого, называется его подклассом. Эту связь обычно описывают с помощью терминов "родительский" и "дочерний". Дочерний класс происходит от родительского и наследует его характеристики. Эти характеристики состоят из свойств и методов. Обычно в дочернем классе к функциональности родительского класса (который также называют *суперклассом*) добавляются новые функциональные возможности. Поэтому говорят, что дочерний класс расширяет родительский.

Прежде чем приступить к изучению синтаксиса наследования, давайте рассмотрим проблемы, которые оно поможет нам решить.

Проблема наследования

Давайте вернемся к классу `ShopProduct`. В настоящее время он является достаточно обобщенным. С его помощью можно оперировать самыми разными товарами.

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

$product2 = new ShopProduct( "Пропавший без вести",
                             "Группа", "ДДТ", 10.99 );

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

На выходе получаем следующее.

```
Автор: Михаил Булгаков
Исполнитель: Группа ДДТ
```

Как видим, разделение имени автора на две части пригодилось нам при работе и с книгами, и с компакт-дисками. В этом случае мы можем сортировать товары по фамилии автора (т.е. по полю, содержащему `"Булгаков"` и `"ДДТ"`), а не по имени, в котором содержатся малозначимые `"Михаил"` и `"Группа"`. Лень — это отличная стратегия проектирования, поэтому на данном этапе вам не следует заботиться об использовании класса `ShopProduct` для более чем одного типа товара.

Но если в нашем примере добавить несколько новых требований, то все сразу усложнится. Представьте, например, что вам нужно отобразить данные, специфич-

ные для книг и компакт-дисков. Скажем, для CD желательно вывести общее время звучания, а для книг — количество страниц. Конечно, могут быть и другие отличия, но эти хорошо иллюстрируют суть проблемы.

Как расширить наш пример, чтобы учесть все эти изменения? На ум сразу приходят два варианта. Во-первых, можно поместить все данные в класс `ShopProduct`. Во-вторых, можно разбить `ShopProduct` на два отдельных класса.

Давайте рассмотрим первый подход. Итак, мы объединяем данные о книгах и компакт-дисках в одном классе.

```
class ShopProduct {
   public $numPages;
   public $playLength;
   public $title;
   public $producerMainName;
   public $producerFirstName;
   public $price;

   function __construct( $title     , $firstName,
                         $mainName  , $price,
                         $numPages=0, $playLength=0 ) {
      $this->title             = $title;
      $this->producerFirstName = $firstName;
      $this->producerMainName  = $mainName;
      $this->price             = $price;
      $this->numPages          = $numPages;
      $this->playLength        = $playLength;
   }

   function getNumberOfPages() {
      return $this->numPages;
   }

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

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

Чтобы продемонстрировать большой объем выполняемой работы по кодированию, в данном примере были использованы методы доступа к свойствам `$numPages` и `$playLength`. В результате объект, экземпляр которого создается с помощью такого класса, будет всегда содержать избыточные методы. Кроме того, для CD экземпляр объекта нужно создавать с помощью бессмысленного аргумента конструктора. Таким образом, для CD будут сохраняться информация и функциональные возможности класса, относящиеся к книгам (количество страниц), а для книг — данные о времени звучания CD. Вероятно, пока вы можете с этим смириться. Но что будет, если мы добавим больше типов товаров, причем каждый — с собственными методами, а затем добавим больше методов для каждого типа? Наш класс будет становиться все более сложным и трудным для использования.

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

Но этим проблемы не ограничиваются. С функциональностью тоже возникнут трудности. Представьте метод, который выводит краткую информацию о товаре. Скажем, отделу продаж нужна информация о товаре в виде одной строки для использования в счете-фактуре. Они хотят, чтобы мы включили в нее время звучания для компакт-диска и количество страниц для книги. Таким образом, при реализации этого метода нам придется учитывать тип каждого товара. Для отслеживания формата объекта можно использовать специальный флаг. Приведем пример.

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

Как видите, чтобы правильно установить значение свойства `$type`, нам нужно в конструкторе проверить значение аргумента `$numPages`. И снова класс `ShopProduct` стал более сложным, чем нужно. По мере добавления дополнительных отличий в форматы или новых форматов нам будет трудно справляться с реализацией этого метода. Поэтому, видимо, для решения данной задачи необходимо применить второй подход.

Поскольку `ShopProduct` начинает напоминать "два класса в одном", мы должны это признать и создать два типа вместо одного. Вот как это можно сделать.

```
class CDProduct {
   public $playLength;
   public $title;
   public $producerMainName;
   public $producerFirstName;
   public $price;

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

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

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

```
      $base .= ": Время звучания - {$this->playLength}";
   return $base;
   }

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

class BookProduct {
   public $numPages;
   public $title;
   public $producerMainName;
   public $producerFirstName;
   public $price;

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

   function getNumberOfPages() {
      return $this->numPages;
   }

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

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

Мы постарались справиться с этой сложностью, хотя пришлось кое-чем пожертвовать. Теперь мы можем создать метод `getSummaryLine()` для каждого типа товара, причем нам даже не нужно проверять значение специального флага. И больше ни один класс не содержит поля или методы, которые не имеют к нему отношения.

А жертва состояла в дублировании. Методы `getProducer()` абсолютно одинаковы для каждого класса. Каждый конструктор одинаково устанавливает ряд идентичных свойств. Это еще один признак дурного тона, и вы должны этого избегать.

Если нужно, чтобы методы `getProducer()` работали одинаково для каждого класса, любые изменения, внесенные в одну реализацию, должны быть внесены и в другую. Но так мы довольно скоро нарушим синхронизацию классов.

Даже если мы уверены, что можем поддерживать дублирование, на этом наши проблемы не закончатся. Ведь у нас теперь есть два типа, а не один.

Помните класс `ShopProductWriter`? Его метод `write()` предназначен для работы с одним типом: `ShopProduct`. Как выйти из сложившейся ситуации, чтобы все работало, как раньше? Мы можем удалить уточнение типа класса из объявления метода, но тогда мы должны надеяться, что методу `write()` будет передан объект правильного типа. Мы можем добавить в тело метода собственный код для проверки типа.

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

Обратите внимание на оператор `instanceof`, использованный в этом примере. Вместо него подставляется значение `true` (истина), если объект в операнде слева относится к типу, представляемому операндом справа.

И снова мы были вынуждены добавить новый уровень сложности. 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, иметь только один тип было бы гораздо лучше. Тогда мы могли бы использовать уточнение типа класса для аргумента метода `write()`, и мы были бы уверены в том, что класс `ShopProduct` поддерживает нужный нам интерфейс.

Особенности класса `ShopProduct`, связанные с книгами и CD, плохо работают вместе, но, похоже, не могут существовать по отдельности. Нам нужно работать с книгами и CD как с одним типом, но в то же время обеспечить отдельную реализацию метода для каждого формата вывода. Нам нужно создать общую функциональность в одном месте, чтобы избежать дублирования, но в то же время сделать так, чтобы при вызове метода, выводящего краткую информацию о товаре, учитывались особенности этого товара. Одним словом, нам необходимо использовать наследование.

Работа с наследованием

Первый шаг в построении дерева наследования — найти элементы базового класса, которые не соответствуют друг другу или которыми нужно оперировать иначе.

Мы знаем, что методы `getPlayLength()` и `getNumberOfPages()` противоречат один другому. Нам также известно, что нужно создать разные реализации метода `getSummaryLine()`. Давайте используем эти различия как основу для создания двух производных классов.

```
class ShopProduct {
   public $numPages;
   public $playLength;
   public $title;
```

```
    public $producerMainName;
    public $producerFirstName;
    public $price;

    function __construct( $title, $firstName,
                          $mainName, $price,
                        $numPages=0, $playLength=0 ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }

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

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

class CDProduct extends ShopProduct {
    function getPlayLength() {
        return $this->playLength;
    }

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

class BookProduct extends ShopProduct {
    function getNumberOfPages() {
        return $this->numPages;
    }

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

Чтобы создать дочерний класс, необходимо использовать в объявлении класса 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 создании экземпляров объектов этих классов будет автоматически вызываться конструктор родительского класса. Дочерние классы наследуют доступ ко всем методам типа `public` и `protected` родительского класса (но не к методам и свойствам типа `private`). Это означает, что мы можем вызвать метод `getProducer()` для экземпляра объекта класса `CDProduct`, хотя метод `getProducer()` определен в классе `ShopProduct`.

```
$product2 = new CDProduct( "Пропавший без вести",
                           "Группа", "ДДТ",
                          10.99, null, 60.33 );
print "Исполнитель: {$product2->getProducer()}\n";
```

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

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

Реализация метода `getSummaryLine()` в суперклассе может показаться избыточной, поскольку 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 также гарантирует для клиентского кода, что во всех объектах типа `ShopProduct` будет присутствовать метод `getSummaryLine()`. Позже вы увидите, как можно выполнить это требование в базовом классе, не предоставляя никакой его реализации. Каждый дочерний объект класса `ShopProduct` унаследует все свойства своего родителя. В собственных реализациях метода `getSummaryLine()` для обоих классов, `CDProduct` и `BookProduct`, обеспечивается доступ к свойству `$title`.

С понятием наследования сразу разобраться непросто. Определяя класс, который расширяет другой класс, мы гарантируем, что экземпляр его объекта определяется характеристиками сначала дочернего, а затем — родительского класса. Чтобы понять это, нужно размышлять с точки зрения поиска. При вызове `$product2->getProducer()` интерпретатор PHP не может найти такой метод в классе `CDProduct`. Поиск заканчивается неудачей, и поэтому используется стандартная реализация этого метода, заданная в классе `ShopProduct`. С другой стороны, когда мы вызываем `$product2->getSummaryLine()`, интерпретатор PHP находит реализацию метода `getSummaryLine()` в классе `CDProduct` и вызывает его.

То же самое верно и в отношении доступа к свойствам. При обращении к свойству `$title` в методе `getSummaryLine()` из класса `BookProduct` интерпретатор PHP не находит определение этого свойства в классе `BookProduct`. Поэтому он использует определение данного свойства, заданное в родительском классе `ShopProduct`. Поскольку свойство `$title` используется в обоих подклассах, оно должно определяться в суперклассе.

Даже поверхностного взгляда на конструктор `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 `$numPages`, значение которого заносится в одноименное свойство, а конструктор класса `CDProduct` должен обрабатывать аргумент и свойство `$playLength`. Чтобы добиться этого, мы определим методы конструктора в каждом дочернем классе.

Конструкторы и наследование

При определении конструктора в дочернем классе вы берете на себя ответственность за передачу требуемых аргументов родительскому классу. Если же вы этого не сделаете, то у вас получится частично сконструированный объект.

Чтобы вызвать нужный метод родительского класса, вам понадобится обратиться к самому 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 "`->`". Поэтому конструкция `parent::__construct()` означает следующее: "Вызвать метод `__construct()` родительского класса". Давайте изменим наш пример так, чтобы каждый класс оперировал только теми данными, которые имеют к нему отношение.

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

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

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

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

class CDProduct extends ShopProduct {
   public $playLength;

   function __construct( $title, $firstName,
                         $mainName,     $price, $playLength ) {
      parent::__construct( $title, $firstName,
                           $mainName, $price );
      $this->playLength = $playLength;
   }

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

   function getSummaryLine() {
```

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

class BookProduct extends ShopProduct {
    public $numPages;

    function __construct( $title, $firstName,
                          $mainName,     $price, $numPages ) {
        parent::__construct( $title, $firstName,
                             $mainName, $price );
        $this->numPages = $numPages;
    }

    function getNumberOfPages() {
        return $this->numPages;
    }

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

Каждый дочерний класс вызывает конструктор своего родительского класса, прежде чем определять собственные свойства. Базовый класс теперь "знает" только о собственных данных. Дочерние классы — это обычно "специализации" родительских классов. Как правило, следует избегать того, чтобы давать родительским классам какие-либо особые "знания" о дочерних классах.

На заметку. До появления PHP 5 имя функции конструктора совпадало с именем класса, к которому она относилась. В новой версии PHP для унификации используется имя функции конструктора `__construct()`. При использовании старого синтаксиса вызов конструктора родительского класса был привязан к имени конкретного класса, например `parent::ShopProduct();`.

В случае изменения иерархии классов это часто приводило к проблемам. Большинство ошибок возникало из-за того, что программисты, после непосредственного изменения "родителя" класса, забывали обновить сам конструктор. А при использовании унифицированного конструктора вызов родительского конструктора `parent::__construct()` означает обращение непосредственно к родительскому классу, независимо от того, какие изменения произошли в иерархии классов. Но, конечно, нужно позаботиться о том, чтобы этому родительскому классу были переданы правильные аргументы!

Вызов переопределенного метода

Ключевое слово `parent` можно использовать в любом методе, который переопределяет свой эквивалент в родительском классе. Когда мы переопределяем метод, то, возможно, хотим не удалить функции "родителя", а, скорее, расширить их. Достичь этого можно, вызвав метод родительского класса в контексте текущего объекта. Если вы снова посмотрите на реализации метода `getSummaryLine()`, то увидите, что

значительная часть кода в них дублируется. И лучше этим воспользоваться, чем повторять функциональность, уже разработанную в классе ShopProduct.

```
// Класс ShopProduct...
   function getSummaryLine() {
      $base  = "{$this->title} ( {$this->producerMainName}, ";
      $base .= "{$this->producerFirstName} )";
      return $base;
   }

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

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 добавлять дополнительные данные к итоговой строке.

Теперь, когда мы познакомились с основами наследования, можно, наконец, рассмотреть вопрос видимости свойств и методов в свете полной картины происходящего.

Public, Private и Protected: управление доступом к классам

До сих пор мы явно или неявно объявляли все свойства как public (общедоступные). Такой тип доступа задан по умолчанию для всех методов, а также свойств, объявленных с использованием устаревшего ключевого слова var.

Элементы класса можно объявить как public (общедоступные), private (закрытые) или protected (защищенные).

- К общедоступным 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 это может быть нам полезно? Ключевые слова, определяющие область видимости, позволяют показывать только те аспекты класса, которые требуются клиенту. Это позволяет создать ясный и понятный интерфейс для объекта.

Контроль доступа, позволяющий запрещать клиенту доступ к некоторым свойствам, поможет также избежать ошибок в коде. Предположим, мы хотим сделать так, чтобы в объектах типа ShopProduct поддерживались скидки. Для этого можно добавить свойство $discount и метод setDiscount().

```
// Класс ShopProduct...
   public $discount = 0;
// ...
```

```
function setDiscount( $num ) {
   $this->discount=$num;
}
```

Вооруженные механизмом определения скидки, мы можем создать метод `getPrice()`, который принимает во внимание установленную скидку.

```
// Класс ShopProduct...
   function getPrice() {
      return ($this->price - $this->discount);
   }
```

Но тут у нас есть проблема. Мы хотим показать всем только скорректированную цену, но клиент может легко обойти метод `getPrice()` и получить доступ к свойству `$price`.

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

В результате будет выведена исходная цена, а не цена со скидкой, которую мы хотим представить. Чтобы предотвратить это, можно просто закрыть свойство `$price`. Это позволит запретить клиентам прямой доступ к нему, заставляя использовать метод `getPrice()`. Любая попытка получить доступ к свойству `$price` из-за пределов класса `ShopProduct` закончится неудачей. В результате для внешнего мира это свойство прекратит существование.

Но определение свойств как `private` — не всегда удачная стратегия, поскольку тогда дочерний класс не сможет получить доступ к закрытым свойствам. А теперь представьте, что правила вашего бизнеса таковы: при покупке только книг скидку на них делать нельзя. Мы можем переопределить метод `getPrice()`, чтобы он возвращал свойство `$price` без применения скидки.

```
// Класс BookProduct
   function getPrice() {
      return $this->price;
   }
```

Поскольку свойство `$price` объявлено в классе `ShopProduct`, а не в `BookProduct`, попытка в приведенном выше коде получить к нему доступ закончится неудачей. Чтобы решить эту проблему, нужно объявить свойство `$price` защищенным (`protected`) и тем самым предоставить доступ к нему дочерним классам. Помните, что к защищенному свойству или методу нельзя получить доступ из-за пределов иерархии того класса, в котором это свойство или метод был объявлен. Доступ к ним можно получить только из исходного класса или его дочерних классов.

Как правило, появление ошибок при доступе к свойствам или методам способствует созданию хорошо защищенного кода. Сначала сделайте свойства закрытыми или защищенными, а затем ослабляйте ограничения по мере необходимости. Многие (если не все) методы в ваших классах будут общедоступными, но, повторяю еще раз, если у вас есть сомнения, ограничьте доступ. Метод, предоставляющий локальные функции другим методам в классе, не нужен пользователям класса. Поэтому сделайте его закрытым или защищенным.

Методы как средство доступа к свойствам

Даже если в клиентской программе нужно будет работать со значениями, хранящимися в экземпляре вашего класса, как правило, стоит запретить прямой доступ к свойствам этого объекта. Вместо этого создайте методы, которые возвращают или

устанавливают нужные значения. Такие методы называют *методами доступа* (accessors) или *получателями* (getter) и *установщиками* (setter).

Вы уже видели одно преимущество, которое дают методы доступа: их можно использовать для фильтрации значений свойств в зависимости от обстоятельств, как было проиллюстрировано выше с помощью метода `getPrice()`.

Метод-установщик может также использоваться для принудительного определения типа свойства. Мы уже видели, что для ограничения типа аргументов метода можно использовать уточнения, но у нас нет непосредственного контроля над типами свойств. Помните определение класса `ShopProductWriter`, с помощью которого выводилась информация об объектах типа `ShopProduct`? Давайте попробуем пойти дальше и сделать так, чтобы класс `ShopProductWriter` мог выводить информацию о любом количестве объектов типа `ShopProduct` одновременно.

```
class ShopProductWriter {
   public $products = array();

   public function addProduct( ShopProduct $shopProduct ) {
      $this->products[] = $shopProduct;
   }

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

Теперь класс `ShopProductWriter` стал намного полезнее. Он может содержать много объектов типа `ShopProduct` и сразу выводить информацию обо всех них. Но мы все еще должны полагаться на то, что программисты клиентского кода будут строго придерживаться правил работы с классом. Хотя мы предоставили метод `addProduct()`, мы не запретили программистам непосредственно выполнять операции над свойством `$products`. В результате можно не только добавить объект неправильного типа к массиву свойств `$products`, но и затереть весь массив и заменить его значением элементарного типа. Чтобы не допустить этого, нужно сделать свойство `$products` закрытым.

```
class ShopProductWriter {
   private $products = array();
//...
```

Теперь внешний код не сможет повредить массив свойств `$products`. Весь доступ к нему должен осуществляться через метод `addProduct()`, а уточнения типа класса, которые используются в объявлении этого метода, гарантируют, что к массиву свойств могут быть добавлены только объекты типа `ShopProduct`.

Семейство классов `ShopProduct`

И в заключение данной главы давайте изменим класс `ShopProduct` и его дочерние классы так, чтобы ограничить доступ к свойствам.

```
class ShopProduct {
   private $title;
   private $producerMainName;
   private $producerFirstName;
   protected $price;
   private $discount = 0;

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

   public function getProducerFirstName() {
      return $this->producerFirstName;
   }

   public function getProducerMainName() {
      return $this->producerMainName;
   }

   public function setDiscount( $num ) {
      $this->discount=$num;
   }

   public function getDiscount() {
      return $this->discount;
   }

   public function getTitle() {
      return $this->title;
   }

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

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

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

class CDProduct extends ShopProduct {
   private $playLength = 0;

   public function __construct( $title, $firstName,
                                $mainName,     $price, $playLength ) {
```

```
        parent::__construct( $title, $firstName,
                             $mainName, $price );
        $this->playLength = $playLength;
    }

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

    public function getSummaryLine() {
        $base  = parent::getSummaryLine();
        $base .= ": Время звучания - {$this->playLength}";
        return $base;
    }
}

class BookProduct extends ShopProduct {
    private $numPages = 0;

    public function __construct( $title, $firstName,
                                 $mainName,     $price, $numPages ) {
        parent::__construct( $title, $firstName,
                             $mainName, $price );
        $this->numPages = $numPages;
    }

    public function getNumberOfPages() {
        return $this->numPages;
    }

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

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

В этой версии семейства классов `ShopProduct` нет ничего существенно нового. Все методы были явно сделаны общедоступными, а все свойства теперь стали либо закрытыми, либо защищенными. И для завершенности мы добавили ряд методов доступа.

Резюме

В этой главе мы подробно рассмотрели основы объектно-ориентированного программирования в PHP, превратив первоначально пустой класс в полностью функциональную иерархию наследования. Мы разобрались в некоторых вопросах проектирования, особенно касающихся типа и наследования. Вы также узнали о поддержке видимости в PHP и познакомились с некоторыми примерами ее применения. В следующей главе вы узнаете о других объектно-ориентированных возможностях PHP.

Глава 4

Расширенные средства

В предыдущей главе вы уже познакомились с уточнением типов аргументов методов класса и управлением доступом к свойствам и методам. Все это позволяет довольно гибко управлять интерфейсом класса. В данной главе мы подробнее изучим более сложные объектно-ориентированные возможности PHP.

В этой главе рассматриваются следующие темы.

- *Статические методы и свойства:* доступ к данным и функциям с помощью классов, а не объектов.
- *Постоянные свойства:* неизменяемая часть класса, или константы.
- *Абстрактные классы и интерфейсы:* отделение проекта от реализации.
- *Трейты:* совместное использование реализации разными классами.
- *Позднее статическое связывание:* новая возможность в PHP 5.3.
- *Обработка ошибок:* знакомство с исключениями.
- *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 строки:* создание резюмирующего метода.
- *Функции обратного вызова:* добавление функциональных возможностей компонентам с помощью анонимных функций.

Статические методы и свойства

Во всех примерах предыдущей главы мы работали с объектами. Я охарактеризовал классы как шаблоны, с помощью которых создаются объекты, а объекты — как активные компоненты, методы которых мы вызываем и к свойствам которых получаем доступ. Отсюда следовал вывод, что в объектно-ориентированном программировании реальная работа выполняется с помощью экземпляров классов. А классы в конечном счете — это просто шаблоны для создания объектов.

Но на самом деле не все так просто. Мы можем получать доступ и к методам, и к свойствам в контексте класса, а не объекта. Такие методы и свойства являются "статическими" и должны быть объявлены с помощью ключевого слова `static`.

```
class StaticExample {
   static public $aNum = 0;

   static public function sayHello() {
      print "Привет!";
   }
}
```

На заметку. Ключевое слово `static` появилось в PHP 5. Его нельзя использовать в сценариях PHP 4.

Статические методы — это функции, используемые в контексте класса. Они сами не могут получать доступ ни к каким обычным свойствам класса, потому что такие свойства принадлежат объектам. Однако из статических методов можно обращаться к *статическим свойствам*. И если вы измените статическое свойство, то все экземпляры этого класса смогут получать доступ к новому значению.

Поскольку доступ к статическому элементу осуществляется через класс, а не экземпляр объекта, вам не нужна переменная, которая ссылается на объект. Вместо этого используется имя класса, после которого указывается два двоеточия `"::"`.

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

С этим синтаксисом вы познакомились в предыдущей главе. Мы использовали конструкцию `"::"` в сочетании с ключевым словом `parent`, чтобы получить доступ к переопределенному методу родительского класса. Однако теперь мы будем обращаться к классу, а не к данным, содержащимся в объекте. В коде класса можно использовать ключевое слово `parent`, чтобы получить доступ к суперклассу, не используя имя класса. Чтобы получить доступ к статическому методу или свойству из того же самого класса (а не из дочернего класса), мы будем использовать ключевое слово `self`. Ключевое слово `self` используется для обращения к текущему классу, точно так же, как псевдопеременная `$this` — к текущему объекту. Поэтому из-за пределов класса `StaticExample` мы должны обращаться к свойству `$aNum` с помощью имени его класса.

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

А внутри класса `StaticExample` можно использовать ключевое слово `self`.

```
class StaticExample {
   static public $aNum = 0;

   static public function sayHello() {
      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.

Кроме случаев обращения к переопределенному методу родительского класса, конструкция `"::"` должна всегда использоваться только для доступа к статическим методам или свойствам.

Однако в документации часто можно увидеть использование конструкции `"::"` для ссылок на методы или

свойства. Это не означает, что рассматриваемый элемент — обязательно статический; это всего лишь значит, что он принадлежит к указанному классу. Например, ссылку на метод `write()` класса `ShopProduct Writer` можно записать так: `ShopProductWriter::write()`, несмотря на то что метод `write()` не является статическим. Мы будем использовать данный синтаксис в книге там, где это уместно.

По определению к статическим методам и свойствам происходит обращение в контексте класса, а не объекта. По этой причине статические свойства и методы часто называют переменными и свойствами класса. Как следствие привязки к классу внутри статического метода нельзя использовать псевдопеременную `$this` для доступа к статическим элементам класса.

А зачем вообще нужны статические методы или свойства? Статические элементы имеют ряд полезных характеристик. Во-первых, они доступны из любой точки сценария (при условии, что у вас есть доступ к классу). Это означает, что вы можете вызывать функции, не передавая экземпляр класса от одного объекта другому или, что еще хуже, сохраняя экземпляр объекта в глобальной переменной. Во-вторых, статическое свойство доступно каждому экземпляру объекта этого класса. Поэтому можно определить значения, которые должны быть доступны всем объектам данного типа. И наконец, в-третьих, сам факт, что не нужно иметь экземпляр класса для доступа к его статическому свойству или методу, позволит избежать создания экземпляров объектов исключительно ради вызова простой функции.

Чтобы продемонстрировать это, давайте создадим статический метод для класса `ShopProduct`, который будет автоматически создавать экземпляры объектов `ShopProduct` на основе информации, хранящейся в базе данных. С помощью `SQLite` определим таблицу `products` следующим образом.

```
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 программу. Мы можем добавить эти методы к классу `ShopProduct`, который был создан в предыдущей главе. Как вы, наверное, знаете, *PDO* расшифровывается как *PHP Data Object* (объекты данных PHP). Класс `PDO` обеспечивает универсальный интерфейс для различных приложений баз данных.

```
// Класс ShopProduct class...
   private $id = 0;
// ...
   public function setID( $id ) {
      $this->id = $id;
   }
// ...
   public static function getInstance( $id, PDO $pdo ) {
```

```
        $stmt = $pdo->prepare("select * from products where id=?");
        $result = $stmt->execute( array( $id ) );


        $row    = $stmt->fetch( );

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

        if ( $row['type'] === "book" ) {
           $product = new BookProduct(
                                    $row['title'],
                                    $row['firstname'],
                                    $row['mainname'],
                                    $row['price'],
                                    $row['numpages']
                                    );
        } else if ( $row['type'] === "cd" ) {
           $product = new CDProduct(
                                    $row['title'],
                                    $row['firstname'],
                                    $row['mainname'],
                                    $row['price'],
                                    $row['playlength'] );
        } else {
           $product = new ShopProduct(
                                    $row['title'],
                                    $row['firstname'],
                                    $row['mainname'],
                                    $row['price'] );
        }
        $product->setId( $row['id'] );
        $product->setDiscount( $row['discount'] );
        return $product;
    }
//...
```

Как видите, метод `getInstance()` возвращает объект типа `ShopProduct`, причем он достаточно "умен" для того, чтобы на основании значения поля `type` создать объект с нужными характеристиками. Я специально опустил код обработки ошибок, чтобы пример был по возможности лаконичным. Например, в реально работающей версии этого кода нам нельзя быть слишком доверчивыми и предполагать, что переданный PDO-объект был корректно проинициализирован и подключен к требуемой базе данных. На самом деле нам, вероятно, следует заключить PDO-объект в класс-оболочку, который гарантирует такое поведение. Дополнительную информацию об объектно-ориентированном программировании и базах данных вы найдете в главе 13, "Шаблоны баз данных".

Метод `getInstance()` более полезен в контексте класса, чем в контексте объекта. Он позволяет легко преобразовать данные, находящиеся в базе данных, в объект, причем для этого нам не нужно иметь отдельный экземпляр объекта типа `ShopProduct`. В этом методе не используются никакие методы или свойства, требующие отдельного экземпляра объекта, поэтому нет никакой причины, чтобы не объявить его статическим. Тогда, имея корректный PDO-объект, мы можем вызвать данный метод из любого места приложения.

```
$dsn = "sqlite://home/bob/projects/products.db";
$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 они берут "сырые" материалы (например, данные, полученные из строки базы данных или файла конфигурации) и используют их для создания объектов. Термин *фабрика* относится к коду, предназначенному для создания экземпляров объектов. С примерами подобных "фабрик" мы еще встретимся в последующих главах.

Разумеется на основе рассмотренного выше примера мы в какой-то степени показали далеко не все проблемы. И хотя я сделал метод `ShopProduct::getInstance()` доступным из любой части программы без необходимости создавать экземпляр объекта `ShopProduct`, я также потребовал, чтобы объект PDO был передан из клиентского кода. Откуда мы должны его взять? В объектно-ориентированном программировании существует еще ряд подобных широко распространенных проблем, например где взять набор ключевых объектов программы и их значений. Различные аспекты создания объектов будут рассмотрены в главе 9, "Генерация объектов".

Постоянные свойства

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

В PHP 5 можно определять постоянные свойства внутри класса. Как и глобальные константы, константы класса нельзя изменять после того, как они были определены. Постоянное свойство объявляют с помощью ключевого слова `const`. В отличие от обычных свойств, перед именем постоянного свойства не ставится знак доллара. По принятому соглашению для них часто выбираются имена, состоящие только из прописных букв, как показано в следующем примере.

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

Постоянные свойства могут содержать только значения, относящиеся к элементарному типу. Константе нельзя присвоить объект. Как и к статическим свойствам, доступ к постоянным свойствам осуществляется через класс, а не через экземпляр объекта. Поскольку константа определяется без знака доллара, при обращении к ней также не требуется использовать никакой символ впереди.

```
print ShopProduct::AVAILABLE;
```

Попытка присвоить константе значение после того, как она была объявлена, приведет к ошибке на этапе синтаксического анализа.

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

Абстрактные классы

Введение абстрактных классов стало одним из главных изменений в PHP5. А включение этой функции в список новых возможностей стало еще одним подтверждением растущей приверженности PHP объектно-ориентированному проектированию.

Экземпляр абстрактного класса нельзя создать. 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`. Давайте переопределим класс `ShopProductWriter`, который мы создали в предыдущей главе, в виде абстрактного класса.

```
abstract class ShopProductWriter {
   protected $products = array();

   public function addProduct( ShopProduct $shopProduct ) {
      $this->products[]=$shopProduct;
   }
}
```

В абстрактном классе вы можете создавать методы и свойства, как обычно, но любая попытка создать его экземпляр приведет к ошибке. Например, в результате выполнения кода

```
$writer = new ShopProductWriter();
```

будет выведено:

```
PHP Fatal error: Cannot instantiate abstract class ShopProductWriter ...[1]
```

В большинстве случаев абстрактный класс будет содержать по меньшей мере один абстрактный метод. Как и класс, он описывается с помощью ключевого слова `abstract`. Абстрактный метод не может иметь реализацию в абстрактном классе. Он объявляется, как обычный метод, но объявление заканчивается точкой с запятой, а не телом метода. Давайте добавим абстрактный метод `write()` к классу `ShopProductWriter`.

```
abstract class ShopProductWriter {
   protected $products = array();

   public function addProduct( ShopProduct $shopProduct ) {
      $this->products[]=$shopProduct;
   }

   abstract public function write();
}
```

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

Если бы мы, как показано ниже, создали класс, производный от `ShopProductWriter`, в котором метод `write()` не был бы реализован,

```
class ErroredWriter extends ShopProductWriter{},
```

то столкнулись бы со следующей ошибкой.

[1] Нельзя создать экземпляр абстрактного класса ShopProductWriter. — *Примеч. ред.*

```
PHP Fatal error: Class ErroredWriter contains 1 abstract method and
must therefore be declared abstract or implement the remaining methods
(ShopProductWriter::write) in...[2]
```

Итак, в любом классе, который расширяет абстрактный класс, должны быть реализованы все абстрактные методы либо сам класс должен быть объявлен абстрактным. При этом в расширяющем классе должны быть не просто реализованы все абстрактные методы, но должна быть воспроизведена сигнатура этих методов. Это означает, что уровень доступа в реализующем методе не может быть более строгим, чем в абстрактном методе. Реализующему методу также должно передаваться такое же количество аргументов, как и абстрактному методу, а также в нем должны воспроизводиться все уточнения типов класса.

Ниже приведены две реализации класса `ShopProductWriter`.

```
class XmlProductWriter extends ShopProductWriter {

   public function write() {
      $writer =new XMLWriter();
      $writer->openMemory();
      $writer->startDocument('1.0','UTF-8');
         $writer->startElement("products");

         foreach ( $this->products as $shopProduct ) {
            $writer->startElement("product");
            $writer->writeAttribute( "title", $shopProduct->getTitle() );
               $writer->startElement("summary");
               $writer->text( $shopProduct->getSummaryLine() );
               $writer->endElement(); // summary
            $writer->endElement(); // product
         }
         $writer->endElement(); // products
      $writer->endDocument();
      print $writer->flush();
   }
}

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

Я создал два класса, каждый с собственной реализацией метода `write()`. Первый выводит данные о товаре в формате XML, а второй — в текстовом виде. Теперь методу, которому требуется передать объект типа `ShopProductWriter`, не нужно точно знать, какой из этих двух классов он получает, поскольку ему достоверно известно, что в обоих классах реализован метод `write()`. Обратите внимание на то, что мы не проверяем тип переменной `$products`, прежде чем использовать ее как массив.

[2] Класс `ErroredWriter` содержит один абстрактный метод и поэтому должен быть объявлен как абстрактный или реализовать перечисленные методы... — *Примеч. ред.*

Причина в том, что это свойство инициализируется как пустой массив в классе `ShopProductWriter`.

В PHP 4 работу абстрактных классов моделировали с помощью методов, которые выводили предупреждающие сообщения или даже содержали операторы `die()`. Это заставляло программиста реализовывать абстрактные методы в производном классе, поскольку в противном случае сценарий переставал работать.

```
class AbstractClass {
   function abstractFunction() {
      die( "AbstractClass::abstractFunction() - абстрактная функция! \n" );
   }
}
```

Проблема в таком подходе состоит в том, что абстрактная природа базового класса проверяется только в случае вызова абстрактного метода. В PHP 5 абстрактные классы проверяются еще на этапе синтаксического анализа, что намного безопаснее.

Интерфейсы

Как известно, в абстрактном классе допускается реализация некоторых методов, не объявленных абстрактными. В отличие от них, *интерфейсы* — это чистой воды 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 методов, но не тела этих методов.

Давайте определим интерфейс.

```
interface Chargeable {
   public function getPrice();
}
```

Как видите, интерфейс очень похож на класс. В любом классе, поддерживающем этот интерфейс, нужно реализовать все методы, определенные в интерфейсе; 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 расширение абстрактного класса, который содержит только абстрактные методы. Давайте сделаем так, чтобы в классе `ShopProduct` был реализован интерфейс `Chargeable`.

```
class ShopProduct implements Chargeable {
   // ...
   public function getPrice() {
      return ( $this->price - $this->discount );
   }
   // ...
```

В классе `ShopProduct` уже есть метод `getPrice()`, что же может быть полезного в реализации интерфейса `Chargeable`? И снова ответ связан с типами. Дело в том, что реализующий класс принимает тип класса и интерфейса, который он расширяет. Это означает, что класс `CDProduct` относится к следующим типам.

```
CDProduct
ShopProduct
Chargeable
```

Эту особенность можно использовать в клиентском коде. Как известно, тип объекта определяет его функциональные возможности. Поэтому метод

```
public function CDInfo( CDProduct $prod ) {
   // ...
}
```

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

Если тот же самый объект `CDProduct` передается методу

```
public function addProduct( ShopProduct $prod ) {
   // ..
}
```

то известно, что объект `$prod` поддерживает все методы, определенные в классе `ShopProduct`. Однако без дальнейшей проверки данный метод ничего не будет знать о методе `getPlayLength()`.

И снова, если передать тот же объект `CDProduct` методу

```
public function addChargeableItem( Chargeable $item ) {
   //...
}
```

данному методу ничего не будет известно обо всех методах, определенных в классах `ShopProduct` или `CDProduct`. При этом интерпретатор только проверит, содержится ли в аргументе `$item` метод `getPrice()`.

Поскольку интерфейс можно реализовать в любом классе (на самом деле в классе можно реализовать любое количество интерфейсов), с помощью интерфейсов можно эффективно объединить типы данных, не связанных никакими другими отношениями. В результате мы можем определить совершенно новый класс, в котором реализуется интерфейс `Chargeable`.

```
class Shipping implements Chargeable {
   public function getPrice() {
     //...
   }
}
```

Затем объект типа `Shipping` мы можем передать методу `addChargeableItem()`, точно так же, как мы передавали ему объект типа `ShopProduct`.

Для клиента, работающего с объектом типа `Chargeable`, очень важно то, что он может вызвать метод `getPrice()`. Любые другие имеющиеся методы связаны с другими типами — через собственный класс объекта, суперкласс или другой интерфейс. Но они не имеют никакого отношения к нашему клиенту.

В классе можно как расширить суперкласс, так и реализовать любое количество интерфейсов. При этом ключевое слово `extends` должно предшествовать ключевому слову `implements`, как показано ниже.

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

Обратите внимание на то, что в классе `Consultancy` реализуется более одного интерфейса. После ключевого слова `implements` можно перечислить через запятую несколько интерфейсов.

В PHP поддерживается только наследование от одного родителя (так называемое одиночное 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. Однако эту проблему можно частично решить с помощью интерфейсов, как было показано в предыдущем разделе. Другими 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 соответствовать типам всех тех интерфейсов, которые в нем реализованы.

Как видите, с помощью интерфейсов создаются новые типы объектов без их реализации. Но что делать, если вам нужно реализовать ряд общих методов для всей иерархии наследования классов? Для этой цели в PHP 5.4 было введено понятие *трейтов*[3].

По сути, трейты напоминают классы, для которых нельзя создать экземпляр объекта, но которые можно включить в другие классы. Поэтому любое свойство (или метод), определенное в трейте, становится частью того класса, в который этот трейт включен. При этом трейт изменяет структуру этого класса, но не меняет его тип. Можно считать трейты своего рода оператором `include`, действие которого распространяется только на конкретный класс.

Давайте рассмотрим на примерах, насколько могут быть полезны трейты.

Проблемы, которые можно решить с помощью трейтов

Ниже приведена версия класса `ShopProduct`, в который был включен метод `calculateTax()`.

```
class ShopProduct {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
}

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

Методу `calculateTax()` в качестве параметра `$price` передается цена товара, а он вычисляет налог с продаж на основе значения ставки, сохраненной во внутреннем свойстве `$taxrate`.

Разумеется, доступ к методу `calculateTax()` будет у всех подклассов данного класса. Но что нам делать, если речь заходит о совершенно другой иерархии классов? Представьте себе класс `UtilityService`, который унаследован от другого класса `Service`. И если для класса `UtilityService` понадобится определить величину нало-

[3] От англ. *trait* (особенность, характерная черта), что можно перевести как *характеристический класс*. Однако для краткости будем использовать кальку с английского языка, тем более что к моменту перевода книги термин *трейт* уже был официально принят в русскоязычном сообществе PHP-разработчиков. — *Примеч. ред.*

га по точно такой же формуле, то нам ничего не остается другого, как просто целиком скопировать тело метода `calculateTax()`, как показано ниже.

```
abstract class Service {
   // Базовый класс для службы сервиса
}

class UtilityService extends Service {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
}

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

Определение и использование 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 стратегические классы (strategy class). Трейты также позволяют решить данную проблему, хотя, возможно, и менее элегантно, но, вне всякого сомнения, эффективно.

В приведенном ниже примере я объявил простой трейт, содержащий метод `calculateTax()`, а затем включил его сразу в оба класса: `ShopProduct` и `UtilityService`.

```
trait PriceUtilities {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
 ...
   // Другие общие методы
}

class ShopProduct {
   use PriceUtilities;
}

abstract class Service {
   // Базовый класс для службы сервиса
}

class UtilityService extends Service {
   use PriceUtilities;
}

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

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

Как видите, трейт `PriceUtilities` объявляется с помощью ключевого слова `trait`. Тело трейта сильно напоминает тело обычного класса. В нем в фигурных скобках просто указывается набор методов (или, как вы увидите ниже, свойств). После объявления трейта `PriceUtilities` я могу его использовать при создании собственных классов. Для этого используется ключевое слово `use`, после которого указывает имя трейта. В результате, объявив и реализовав в одном месте метод `calculateTax()`, я могу его использовать в обоих классах: и в `ShopProduct`, и в `UtilityService`.

Использование нескольких трейтов

В класс можно включить несколько трейтов. Для этого их нужно перечислить через запятую после ключевого слова `use`. В приведенном ниже примере я определил и реализовал новый трейт `IdentityTrait`, а затем использовал его в своем классе наряду с трейтом `PriceUtilities`.

```
trait IdentityTrait {
   public function generateId() {
      return uniqid();
   }
}

trait PriceUtilities {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
...
// Другие методы
}

class ShopProduct {
   use PriceUtilities, IdentityTrait;
}

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

Перечислив оба трейта, `PriceUtilities` и `IdentityTrait`, после ключевого слова `use`, я сделал доступными методы `calculateTax()` и `generateId()` для класса `ShopProduct`. Это означает, что эти методы становятся членами класса `ShopProduct`.

На заметку. В трейте `IdentityTrait` реализован метод `generateId()`. По сути, уникальные 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 уникальных идентификаторах мы поговорим в главе 13, "Шаблоны баз данных", где описан шаблон `Identity Map`. О процессе тестирования и имитации функционала объектов (mocking) речь пойдет в главе 18, "Тестирование с помощью PHPUnit".

Совместное использование трейтов и интерфейсов

Несмотря на то что полезность применения трейтов не вызывает особых сомнений, они не позволяют изменить тип класса, в который были включены. Поэтому, если трейт `IdentityTrait` используется сразу в нескольких классах, у вас не будет общего типа, который можно было бы указать в уточнениях для сигнатур методов.

К счастью, трейты можно успешно использовать вместе с интерфейсами. Мы можем определить интерфейс с сигнатурой метода `generateId()`, а затем указать, что в классе `ShopProduct` реализуются методы этого интерфейса.

```
interface IdentityObject {
   public function generateId();
}

trait IdentityTrait {
   public function generateId() {
      return uniqid();
   }
}
trait PriceUtilities {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
   // Другие методы
}

class ShopProduct implements IdentityObject {
   use PriceUtilities, IdentityTrait;
}
```

Здесь, как и в предыдущем примере, в классе `ShopProduct` используется трейт `IdentityTrait`. Однако импортируемый с его помощью метод `generateId()` теперь также удовлетворяет требованиям интерфейса `IdentityObject`. А это означает, что мы можем передавать объекты `ShopProduct` тем методам и функциям, в описании аргументов которых используются уточнения типа объекта `IdentityObject`, как показано ниже.

```
function storeIdentityObject( IdentityObject $idobj ) {
   // Работа с объектом типа IdentityObject
}

$p = new ShopProduct();
storeIdentityObject( $p );
```

Устранение конфликтов имен с помощью ключевого слова `insteadof`

Возможность комбинирования трейтов является просто замечательной! Однако рано или поздно вы можете столкнуться с конфликтом имен. Например, что произойдет, если в обоих включаемых трейтах будет реализован метод `calculateTax()`, как показано ниже?

```
trait TaxTools {
   function calculateTax( $price ) {
      return 222;
   }
}

trait PriceUtilities {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
   // Другие методы
}

abstract class Service {
   // Базовый класс для службы сервиса
}

class UtilityService extends Service {
   use PriceUtilities, TaxTools;
}

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

Поскольку в один классы мы включили два трейта, содержащие методы `calculateTax()`, интерпретатор PHP не сможет продолжить работу, так как он не знает, какой из методов нужно использовать. В результате выводится сообщение о неустранимой ошибке, как показано ниже.

```
Fatal error: Trait method calculateTax has not been applied, because there
are collisions with other trait methods on UtilityService in...[4]
```

Для устранения этой проблемы используется ключевое слово `insteadof`, как показано ниже.

```
trait TaxTools {
   function calculateTax( $price ) {
      return 222;
   }
}

trait PriceUtilities {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
   // Другие методы
}

abstract class Service {
   // Базовый класс для службы сервиса
```

[4] Метод трейта `calculateTax` не может быть использован из-за конфликта имен в другом трейте `UtilityService` в... — *Примеч. ред.*

```
}

class UtilityService extends Service {
   use PriceUtilities, TaxTools {
   TaxTools::calculateTax insteadof PriceUtilities;
   }
}

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

Для того чтобы можно было применять директивы оператора `use`, нам нужно добавить к нему тело, которое помещается в фигурные скобки. Внутри этого блока используется конструкция с ключевым словом `insteadof`. Слева от него указывается полностью определенное имя метода, состоящее из имени трейта и имени метода. Они разделяются двумя двоеточиями, играющими в данном случае роль оператора определения зоны видимости. В правой части конструкции `insteadof` указывается имя трейта, метод которого с аналогичным именем должен быть заменен. Таким образом, запись

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

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

Поэтому при запуске приведенного выше кода будет выведено число `222`, которое я ввел в код метода `TaxTools::calculateTax()`.

Псевдонимы для переопределенных методов трейта

Выше мы уже убедились в том, что с помощью ключевого слова `insteadof` можно устранить конфликт имен методов, принадлежащих разным трейтам. Однако что делать, если вдруг понадобится вызвать в коде переопределенный метод трейта? Для этого используется ключевое слово `as`, которое позволяет назначить этому методу псевдоним. Как и в конструкции с ключевым словом `insteadof`, при использовании `as` нужно слева от него указать полностью определенное имя метода, а справа — псевдоним имени метода. В приведенном ниже примере метод `calculateTax()` трейта `PriceUtilities` был восстановлен под новым именем `basicTax()`.

```
trait TaxTools {
   function calculateTax( $price ) {
      return 222;
   }
}

trait PriceUtilities {
   private $taxrate = 17;

   function calculateTax( $price ) {
      return ( ( $this->taxrate/100 ) * $price );
   }
   // Другие методы
}

abstract class Service {
   // Базовый класс для службы сервиса
}
```

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

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

В результате выполнения этого фрагмента кода будет выведено

```
222
17
```

Как видите, метод 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.

Кстати, здесь уместно отметить, что псевдонимы имен методов можно использовать даже тогда, когда нет никакого конфликта имен. Так, например, с помощью метода трейта вы можете реализовать абстрактный метод, сигнатура которого была объявлена в родительском классе или интерфейсе.

Использование статических методов в трейте

В большинстве примеров, которые мы рассматривали до сих пор, использовались статические методы, поскольку для их вызова не требуются экземпляры класса. Поэтому нам ничто не мешает поместить статический метод в трейт. В приведенном ниже примере я изменил описание свойства `PriceUtilities::$taxrate` и метода `PriceUtilities::calculateTax()` так, чтобы они стали статическими.

```
trait PriceUtilities {
   private static $taxrate = 17;

   static function calculateTax( $price ) {
      return ( ( self::$taxrate/100 ) * $price );
   }

// Другие методы
}

abstract class Service {
// Базовый класс для службы сервиса
}

class UtilityService extends Service {
   use PriceUtilities;
}

$u = new UtilityService();
print $u::calculateTax( 100 )."\n";
```

Как и следовало ожидать, в результате выполнения этого кода будет выведено число 17.

Доступ к свойствам базового класса

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, являются по своей природе статическими, ведь так? Ну что же, вас ввели в заблуждение — к свойствам и методам базового класса также можно получить доступ.

```
trait PriceUtilities {
   function calculateTax( $price ) {
      // Хорош ли такой подход?
         return ( ( $this->taxrate/100 ) * $price );
   }

// Другие методы
}
abstract class Service {
// Базовый класс для службы сервиса
}

class UtilityService extends Service {
   public $taxrate = 17;
   use PriceUtilities;
}

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

Здесь я усовершенствовал трейт `PriceUtilities` так, чтобы из него можно было обращаться к свойству базового класса. И если вам кажется, что такой подход плох, то вы правы. Скажу больше — он вызывающе плох! Несмотря на то что обращение из трейтов к данным, расположенным в базовом классе, является обычной практикой, у нас не было весомых причин объявлять свойство `$taxrate` в классе `UtilityService`. Здесь не стоит забывать, что трейты могут использоваться во многих совершенно разных классах. И кто может дать гарантию (или даже обещание!), что в каждом базовом классе будет объявлено свойство `$taxrate`?

С другой стороны, будет просто замечательно, если вам удастся заключить договор с пользователем, в котором в частности говорится: "При использовании данного трейта вы обязаны предоставить в его распоряжение определенные ресурсы". По сути, здесь нам удалось достичь точно такого же эффекта. Дело в том, что в трейтах поддерживаются абстрактные методы.

Определение абстрактных методов в трейтах

В трейтах можно объявлять абстрактные методы точно так же, как и в обычных классах. При использовании такого трейта в классе в нем должны быть реализованы все объявленные в трейте абстрактные методы.

Имея это в виду, я могу переписать предыдущий пример так, чтобы трейт заставлял использующий его класс предоставлять информацию о ставке налога.

```
trait PriceUtilities {
   function calculateTax( $price ) {
```

```
      // Гораздо лучший подход, поскольку нам точно известно,
      // что метод getTaxRate() будет реализован
         return ( ( $this->getTaxRate()/100 ) * $price );
   }

   abstract function getTaxRate();

// Другие методы
}

abstract class Service {
// Базовый класс для службы сервиса

}

class UtilityService extends Service {
   use PriceUtilities;
   function getTaxRate() {
      return 17;
   }
}

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

Объявив абстрактный метод `getTaxRate()` в трейте `PriceUtilities`, я вынудил программиста обеспечить его реализацию в классе `UtilityService`. Разумеется, поскольку в PHP не накладывается каких-либо жестких ограничений на тип возвращаемого значения, нельзя быть точно уверенным в том, что в методе `UtilityService::calculateTax()` мы получим от метода `getTaxRate()` корректное значение. Этот недостаток можно преодолеть, поместив в код операторы, выполняющие все возможные виды проверок, но тем самым мы не достигнем поставленной цели. Здесь, вероятнее всего, будет вполне достаточно указать программисту клиентского кода, что при реализации затребованных нами методов нужно возвратить значение заданного типа.

Изменение прав доступа к методам трейта

Разумеется, ничто не может вам помешать объявить методы трейта открытыми (`public`), защищенными (`private`) или закрытыми (`protected`). Тем не менее вы можете также изменить эти атрибуты доступа к методам прямо внутри класса, в котором используется трейт. Выше было показано, как с помощью ключевого слова `as` можно назначить методу псевдоним. Если справа от этого ключевого слова указать новый модификатор доступа, то вместо назначения методу псевдонима будет изменен его атрибут доступа.

Давайте в качестве примера представим, что вы хотите использовать метод `calculateTax()` только внутри класса `UtilityService` и вам не нужно, чтобы этот метод можно было вызвать из клиентского кода. Внесите изменения в оператор `use`, как показано ниже.

```
trait PriceUtilities {
   function calculateTax( $price ) {
      return ( ( $this->getTaxRate()/100 ) * $price );
   }
   abstract function getTaxRate();
```

```
// Другие методы
}

abstract class Service {
// Базовый класс для службы сервиса
}

class UtilityService extends Service {
   use PriceUtilities {
   PriceUtilities::calculateTax as private;
   }
   private $price;
   function __construct( $price ) {
      $this->price = $price;
   }

   function getTaxRate() {
      return 17;
   }

   function getFinalPrice() {
      return ( $this->price + $this->calculateTax( $this->price ) );
   }
}

$u = new UtilityService( 100 );
print $u->getFinalPrice()."\n";
```

Для того чтобы закрыть доступ к методу `calculateTax()` извне класса `UtilityService`, после ключевого слова `as` в операторе `use` был указан модификатор `private`. В результате доступ к этому методу стал возможен только из метода `getFinalPrice()`. Теперь при попытке обращения к методу `calculateTax()` извне класса, например так:

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

выводится сообщение об ошибке:

```
Fatal error: Call to private method UtilityService::calculateTax() from
context '' in...[5]
```

Позднее статическое связывание: ключевое слово `static`

После того как мы рассмотрели абстрактные классы, трейты и интерфейсы, самое время снова ненадолго обратиться к статическим методам. Вы уже знаете, что статический метод можно использовать в качестве фабрики, создающей экземпляры объектов того класса, в котором содержится этот метод. Если вы такой же ленивый программист, как и я, то вас должны раздражать повторы кода наподобие приведенных ниже.

[5] Вызов закрытого метода `UtilityService::calculateTax()` из контекста `''` в ...— *Примеч. ред.*

```
abstract class DomainObject {
}

class User extends DomainObject {
   public static function create() {
      return new User();
   }
}

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

Сначала я создал суперкласс под именем `DomainObject`. Само собой разумеется, что в реальном проекте в нем будет находиться функциональность, общая для всех дочерних классов. После этого я создал два дочерних класса: `User` и `Document`. Я хотел, чтобы в каждом из моих конкретных классов находился метод `create()`.

На заметку. Почему для создания конкретного объекта я воспользовался статическим методом-фабрикой, а не оператором `new` и конструктором объекта? В главе 13, "Шаблоны баз данных", я описал шаблон Identity Map. Компонент Identity Map создает и инициализирует новый объект только в том случае, если объект с аналогичными отличительными особенностями еще не создан. Если таковой объект существует, то возвращается просто ссылка на него. Статический метод-фабрика наподобие рассмотренного выше метода `create()` является отличным кандидатом для реализации подобной функциональности.

Созданный мною выше код прекрасно работает, но в нем есть досадный недостаток — дублирование. Мне совсем не нравится повторять однотипный код наподобие того, который приведен выше, для каждого создаваемого дочернего объекта, расширяющего класс `DomainObject`. Как насчет того, чтобы переместить метод `create()` в суперкласс?

```
abstract class DomainObject {

   public static function create() {
      return new self();
   }
}

class User extends DomainObject {
}

class Document extends DomainObject {
}

Document::create();
```

Ну что ж, все это *выглядит* круто! Теперь весь общий код сосредоточен в одном месте, и, чтобы обратиться к текущему классу, я воспользовался ключевым словом `self`. Однако насчет ключевого слова `self` я сделал допущение, что оно *должно* так работать. На самом деле оно *не* работает для классов так же, как псевдопеременная `$this` для объектов. С помощью ключевого слова `self` *нельзя* сослаться на вызывающий контекст. Оно используется только для разрешения ссылок на содержащий класс, в контексте которого вызывается метод. Поэтому при попытке запуска приведенного выше примера получим следующее сообщение об ошибке.

```
PHP Fatal error: Cannot instantiate abstract class DomainObject[6] in....
```

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

Итак, теперь я смогу воспользоваться всеми преимуществами наследования в статическом контексте.

```
abstract class DomainObject {
   public static function create() {
      return new static();
   }
}

class User extends DomainObject {
}

class Document extends DomainObject {
}

print_r(Document::create());
```

В результате будет выведено следующее.

```
Document Object
(
)
```

Ключевое слово `static` можно использовать не только для создания объектов. Так же, как и `self` и `parent`, его можно использовать как идентификатор для вызова статических методов даже из нестатического контекста. Например, я хочу реализовать идею группировки моих классов типа `DomainObject`. По умолчанию все классы попадают в категорию `'default'`. Но для некоторых веток иерархии наследования моих классов мне нужно это переопределить.

```
abstract class DomainObject {
   private $group;

   public function __construct() {
      $this->group = static::getGroup();
   }

   public static function create() {
      return new static();
   }

   static function getGroup() {
```

[6] Нельзя создать экземпляр абстрактного класса. — *Примеч. ред.*

```
        return "default";
    }
}

class User extends DomainObject {
}

class Document extends DomainObject {
    static function getGroup() {
        return "document";
    }
}
class SpreadSheet extends Document {
}

print_r(User::create());
print_r(SpreadSheet::create());
```

Здесь в класс `DomainObject` я ввел конструктор, в котором используется ключевое слово `static` для вызова метода `getGroup()`. Стандартное значение группы сосредоточено в классе `DomainObject`, но оно переопределяется в классе `Document`. Я также создал новый класс `SpreadSheet`, расширяющий класс `Document`. Вот что получим в результате.

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

Все происходящее с классом `User` не настолько очевидно и поэтому требует объяснений. В конструкторе класса `DomainObject` вызывается метод `getGroup()`, который интерпретатор находит в текущем классе. Несмотря на это, в случае с классом `SpreadSheet` поиск метода `getGroup()` начинается не с класса `DomainObject`, а с класса `SpreadSheet`, для которого из метода `create()` был вызван стандартный конструктор. Поскольку в классе `SpreadSheet` реализация метода `getGroup()` не предусмотрена, интерпретатор вызывает аналогичный метод класса `Document` (т.е. идет вверх по иерархии объектов). До появления PHP 5.3 и позднего статического связывания здесь у меня возникала проблема из-за использования ключевого слова `self`, которое находило метод `getGroup()` 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-адреса изменились, XML-файлы были повреждены, права доступа установлены неправильно, лимиты на дисковую память превышены. Этот список можно продолжать до бесконечности. В стремлении предусмотреть любую проблему простой метод может иногда утонуть под тяжестью собственного кода обработки ошибок.

Ниже приведено определение простого класса `Conf`, который сохраняет, извлекает и определяет данные в XML-файле конфигурации.

```
class Conf {
   private $file;
   private $xml;
   private $lastmatch;

   function __construct( $file ) {
      $this->file = $file;
      $this->xml = simplexml_load_file($file);
   }

   function write() {
      file_put_contents( $this->file, $this->xml->asXML() );
   }

   function get( $str ) {
      $matches = $this->xml->xpath("/conf/item[@name=\"$str\"]");
      if ( count( $matches ) ) {
         $this->lastmatch = $matches[0];
         return (string)$matches[0];
      }
      return null;
   }

   function set( $key, $value ) {
      if ( ! is_null( $this->get( $key ) ) ) {
         $this->lastmatch[0]=$value;
         return;
      }
      $conf = $this->xml->conf;
      $this->xml->addChild('item', $value)->addAttribute( 'name', $key );
   }
}
```

В классе `Conf` для доступа к парам "имя-значение" используется расширение PHP SimpleXml. Ниже приведен фрагмент файла конфигурации в формате XML, с которым работает наш класс.

```
<?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()`. Полученный от функции объект типа `SimpleXmlElement` сохраняется в свойстве `$xml`. В методе `get()` для нахождения элемента `item` с заданным атрибутом `name` используется метод `xpath` объекта `SimpleXmlElement`. Значение найденного элемента возвращается в вызывающий код. Метод `set()` либо меняет значение существующего элемента, либо создает новый. И наконец метод `write()` сохраняет данные о новой конфигурации в исходном файле на диске.

Как и многие коды, приведенные в качестве примеров, код класса `Conf` крайне упрощен. В частности, в нем не предусмотрена обработка ситуаций, когда файл 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.

Провести тестирование подобных ошибок достаточно просто, но мы должны решить, как нужно на них реагировать, если они возникнут. В целом у нас есть две возможности.

1. Мы можем завершить выполнение программы. Это простой, но радикальный выход. В результате наш скромный класс будет виноват в том, что из-за него потерпела неудачу вся программа. Хотя такие методы, как `__construct()` и `write()`, удачно расположены в коде для обнаружения ошибок, у них нет информации, позволяющей решить, как обрабатывать эти ошибки.
2. Вместо обработки ошибки в классе мы можем вернуть признак ошибки в том или ином виде. Это может быть булево или целое 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.

Во многих PEAR-пакетах сочетаются эти два подхода и возвращается объект ошибок (экземпляр класса `PEAR_Error`). Наличие этого объекта говорит о том, что произошла ошибка, а подробная информация о ней содержится в самом объекте. Этот подход в настоящее время не рекомендуется использовать, но многие классы до сих пор не были обновлены в значительной степени из-за того, что клиентский код зачастую рассчитан на старые стандарты.

Проблема также заключается еще и в том, что возвращаемое значение может быть затерто. В PHP нет средств, заставляющих возвращать унифицированное значение. На момент написания данной книги в PHP не поддерживались уточнения типа для возвращаемого класса, поэтому ничто не может нам помешать вернуть признак ошибки вместо ожидаемого объекта или значения элементарного типа. Делая так, мы должны полагаться на то, что клиентский код будет проверять тип возвращаемого объекта после каждого вызова нашего метода, подверженного ошибкам. А это довольно рискованно. Никому нельзя доверять!

Когда в вызывающий код возвращается ошибочное значение, нет никакой гарантии, что клиентский код будет "вооружен" лучше нашего метода и сможет решить, как обрабатывать ошибку. А если не сможет, то проблемы будут появляться снова и снова. В клиентском методе нужно будет определить, как реагировать на ошибочную ситуацию, и, возможно, даже реализовать другую стратегию сообщения об ошибке.

Исключения

В PHP 5 было введено понятие исключения, представляющее собой совершенно другой способ обработки ошибок. Я хочу сказать — совершенно другой для PHP. Но если у вас есть опыт работы с Java или C++, то исключения покажутся вам знакомыми и близкими. Использование исключений позволяет решить все проблемы, которые я описывал в данном разделе.

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 классе существуют также некоторые полезные методы для анализа ошибочной ситуации (табл. 4.1).

Таблица 4.1. Общедоступные методы класса `Exception`

Метод	Описание
`getMessage()`	Получить строку сообщения, переданную конструктору
`getCode()`	Получить код ошибки (целое число), 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 типа `Exception`
`getTrace()`	Получить многомерный массив, отслеживающий вызовы метода, которые привели к исключению, в том числе имя метода, класса, файла и значение аргумента
`getTraceAsString()`	Получить строковую версию данных, возвращенных методом `getTrace()`
`__toString()`	Вызывается автоматически, когда объект `Exception` используется в контексте строки. Возвращает строку, описывающую подробности исключения

Класс `Exception` крайне полезен для поиска сообщения об ошибке и информации для отладки (в этом отношении особенно полезны методы `getTrace()` и `getTraceAsString()`). На самом деле класс `Exception` почти идентичен классу `PEAR_Error`, который мы обсуждали выше. Но в нем сохраняется меньше информации об исключениях, чем есть на самом деле.

Генерация исключений

Совместно с объектом `Exception` используется ключевое слово `throw`. Оно останавливает выполнение текущего метода и передает ответственность за обработку ошибок назад в вызывающий код. Давайте подкорректируем метод `__construct()`, чтобы использовать оператор `throw`.

```
function __construct( $file ) {
   $this->file = $file;
   if ( ! file_exists( $file ) ) {
      throw new Exception( "Файл '$file' не найден" );
   }
   $this->xml = simplexml_load_file($file);
}
```

Аналогичная конструкция может использоваться и в методе `write()`.

```
function write() {
   if ( ! is_writeable( $this->file ) ) {
      throw new Exception("Файл '{$this->file}' недоступен для записи.");
   }
   file_put_contents( $this->file, $this->xml->asXML() );
}
```

Теперь наши методы `__construct()` и `write()` могут тщательно проверять ошибки, связанные с доступом к файлу, по мере выполнения своей работы. Однако при этом решение о том, как реагировать на любые обнаруженные ошибки, будет приниматься в клиентском коде.

Так как же тогда клиентский код узнает, что возникло исключение и настала пора его обрабатывать? Для этого при вызове метода, в котором может возникнуть исключение, следует использовать оператор `try`. Оператор `try` состоит из ключевого слова `try` и следующих за ним фигурных скобок. За оператором `try` должен сле-

довать по меньшей мере один оператор catch, в котором можно обработать любую ошибку, как показано ниже.

```
try {
    $conf = new Conf( dirname(__FILE__) . "/conf01.xml" );
    print "user: " . $conf->get('user') . "\n";
    print "host: " . $conf->get('host') . "\n";
    $conf->set("pass", "newpass");
    $conf->write();
} catch ( Exception $e ) {
    die( $e->__toString() );
}
```

Как видите, оператор catch внешне напоминает объявление метода. Когда генерируется исключение, управление передается оператору catch в контексте вызывающего метода, которому автоматически передается в качестве переменной-аргумента объект типа Exception.

Теперь, если при выполнении метода возникнет исключение и вызов этого метода находится внутри оператора try, работа сценария останавливается и управление передается непосредственно оператору catch.

Создание подклассов класса Exception

Вы можете создать классы, расширяющие класс Exception, точно так же, как это делается для любого другого определенного пользователем класса. Существуют две причины, по которым возникает необходимость это сделать. Во-первых, можно расширить функциональность класса. Во-вторых, производный класс определяет новый тип класса, который поможет при обработке ошибок.

В сущности, для одного оператора try вы можете определить столько операторов catch, сколько нужно. То, какой конкретно оператор catch будет вызван, будет зависеть от типа сгенерированного исключения и указанного уточнения типа класса в списке аргументов. Давайте определим некоторые простые классы, расширяющие класс Exception.

```
class XmlException extends Exception {
   private $error;

   function __construct( LibXmlError $error ) {
      $shortfile = basename( $error->file );
      $msg  = "[{$shortfile}, строка {$error->line}, ";
      $msg .= "колонка {$error->column}] {$error->message}";
      $this->error = $error;
      parent::__construct( $msg, $error->code );
   }

   function getLibXmlError() {
      return $this->error;
   }
}
class FileException extends Exception { }
class ConfException extends Exception { }
```

Объект типа LibXmlError создается автоматически, когда средства SimpleXml обнаруживают поврежденный XML-файл. У него есть свойства message и code, и он напоминает класс Exception. Мы пользуемся преимуществом этого подобия и используем объект LibXmlError в классе XmlException. У классов FileException

и ConfException не больше функциональных возможностей, чем у подкласса Exception. Теперь мы можем использовать эти классы в коде и подкорректировать оба метода, __construct() и write().

```
// Класс Conf...
   function __construct( $file ) {
      $this->file = $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() );
      }
      print gettype( $this->xml );
      $matches = $this->xml->xpath("/conf");
      if ( ! count( $matches ) ) {
         throw new ConfException( "Корневой элемент conf не найден." );
      }
   }

   function write() {
      if ( ! is_writeable( $this->file ) ) {
         throw new FileException("Файл '{$this->file}' недоступен для записи");
      }
      file_put_contents( $this->file, $this->xml->asXML() );
   }
```

Метод __construct() генерирует исключения типа XmlException, FileException или ConfException в зависимости от вида ошибки, которую он обнаружит. Обратите внимание на то, что методу simplexml_load_file() передается флажок LIBXML_NOERROR. Это блокирует выдачу предупреждений внутри класса и оставляет программисту свободу действий для их последующей обработки с помощью класса XmlException. Если обнаружится поврежденный XML-файл, то метод simplexml_load_file() уже не возвратит объект типа SimpleXMLElement. Благодаря классу XmlException в клиентском коде можно будет легко узнать причину ошибки, а с помощью метода libxml_get_last_error() — все подробности этой ошибки.

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

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

```
class Runner {
   static function init() {
      try {
          $conf = new Conf( dirname(__FILE__) . "/conf01.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 либо недоступен для записи

      } catch ( XmlException $e ) {
```

```
        // Поврежденный XML-файл

      } catch ( ConfException $e ) {
        // Некорректный формат XML-файла

      } catch ( Exception $e ) {
        // Ловушка: этот код не должен никогда вызываться
      }
    }
}
```

В этом примере мы предусмотрели оператор `catch` для каждого типа класса ошибки. То, какой оператор будет вызван, зависит от типа сгенерированного исключения. При этом будет выполнен первый подходящий оператор. Поэтому помните: самый общий тип нужно размещать в конце, а самый специализированный — в начале списка операторов `catch`. Например, если бы вы разместили оператор `catch` для обработки исключения типа `Exception` перед операторами для обработки исключений типа `XmlException` и `ConfException`, ни один из них никогда не был бы вызван. Причина в том, что оба исключения относятся к типу `Exception` и поэтому будут соответствовать первому оператору.

Первый оператор `catch (FileException)` вызывается, если есть проблема с файлом конфигурации (если этот файл не существует или в него нельзя ничего записать). Второй оператор `catch (XmlException)` вызывается, если происходит ошибка при синтаксическом анализе XML-файла (например, если какой-то элемент не закрыт). Третий оператор `catch (ConfException)` вызывается, если корректный в плане формата XML-файл не содержит ожидаемый корневой элемент `conf`. Последний оператор `catch (Exception)` не должен вызываться, потому что наши методы генерируют только три исключения, которые обрабатываются явным образом. Вообще, неплохо иметь такой оператор-ловушку на случай, если в процессе разработки понадобится добавить в код новые исключения.

Преимущество этих уточненных операторов `catch` в том, что они позволяют применить к разным ошибкам различные механизмы восстановления или неудачного завершения. Например, вы можете прекратить выполнение программы, записать в журнал информацию об ошибке и продолжить выполнение или повторно сгенерировать исключение, как показано ниже.

```
try {
    //...
} catch ( FileException $e ) {
    throw $e;
}
```

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 ваша программа. Более подробно об этом методе читайте в главе 15.

Но что произойдет, если исключение не будет обработано в клиентском коде? Тогда оно автоматически сгенерируется повторно, и его сможет обработать вызывающий код клиентской программы. Этот процесс будет продолжаться до тех пор, пока исключение не будет обработано либо его уже нельзя будет снова сгенерировать. Тогда произойдет неустранимая ошибка. Вот что произойдет, если в примере нашего кода не будет обработано ни одно исключение.

```
PHP Fatal error: Uncaught exception 'FileException' with message
'file 'nonexistent/not_there.xml' does not exist' in ...[7]
```

Итак, генерируя исключение, вы заставляете 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. Метод `write()` в нашем примере знает, когда попытка сделать запись заканчивается неудачей и почему, но не знает, что с этим делать. Именно так и должно быть. Если бы мы сделали класс `Conf` более сведущим, чем он есть в настоящее время, он бы потерял свою универсальность и перестал бы быть повторно используемым.

Уборка за собой с помощью оператора `finally`

Сам факт, что в ход выполнения программы могут вмешаться внешние факторы в виде исключений, может привести к неожиданным проблемам. Например, после возникновения исключения в блоке `try` может не выполниться код очистки или любые другие важные действия. Как было сказано выше, при возникновении исключительной ситуации в блоке `try` управление программы передается первому подходящему блоку `catch`. В результате может не выполниться код, в котором закрывается подключение к базе данных или файлу, обновляется текущая информация о состоянии и т.п.

В качестве примера предположим, что в методе `Runner::init()` регистрируются все действия приложения. При этом в системный журнал записываются факт начала процесса инициализации, все ошибки, возникающие при работе приложения, а в самом конце — факт окончания процесса инициализации. Ниже приведен типичный упрощенный фрагмент кода, выполняющий эти действия.

```
class Runner {

   static function init() {
      try {
          $fh = fopen("./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 ) {
          fputs( $fh, "Файловая ошибка\n" );
          //...
```

Здесь сначала открывается файл `log.txt`, после чего в него записываются данные, а затем вызывается код для конфигурирования приложения. В случае возникновения ошибки на данном этапе информация об этом записывается в файл журнала в блоке `catch`. Блок `try` завершается оператором записи в файл и закрытием этого файла. Разумеется, в случае возникновения исключительной ситуации эти действия выполнены не будут. При этом управление передается в соответствующий

[7] Необработанное исключение типа `FileException`, содержащее сообщение с указанием ошибки. — *Примеч. ред.*

блок `catch`, и оставшаяся часть кода блока `try` выполнена не будет. Ниже показан фрагмент системного журнала при возникновении исключительной ситуации, связанной с файлами.

```
Начало
Файловая ошибка
```

Как видите, в журнале зафиксированы факт начала процесса инициализации приложения и файловая ошибка. Поскольку фрагмент кода, в котором выводится информация в журнал об окончании процесса инициализации, не был выполнен, соответственно, в файл журнала ничего записано не было.

На первый взгляд может показаться, что код последнего этапа записи в журнал нужно вынести за пределы блока `try/catch`. Однако такое решение нельзя назвать надежным. Дело в том, что при обработке исключительной ситуации в блоке `catch` может быть принято решение о возобновлении выполнения программы. При этом 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 либо вовсе завершена работа программы.

Поэтому, чтобы помочь программистам корректно выйти из описанной выше ситуации, в PHP 5.5 был введен новый оператор — `finally`. Если вы знакомы с языком Java, наверняка вы уже с ним сталкивались. Несмотря на то что блок кода `catch` вызывается только при возникновении исключительной ситуации заданного типа, блок кода `finally` вызывается всегда, независимо от того, возникло ли исключение при выполнении блока `try`.

Итак, для решения описанной выше проблемы мне нужно переместить код последнего этапа записи в журнал и закрытия файла в блок `finally`.

```
class Runner {

   static function init() {
      $fh = fopen("./log.txt","w");
      try {
          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 недоступен для записи или не найден
          fputs( $fh, "Файловая ошибка\n" );
          throw $e;
      } catch ( XmlException $e ) {
          fputs( $fh, "Ошибка в коде xml\n" );
       // Некорректный код xml
      } catch ( ConfException $e ) {
          fputs( $fh, "Ошибка в файле конфигурации\n" );
       // Некорректный тип XML-файла
      } catch ( Exception $e ) {
          fputs( $fh, "Другая ошибка\n" );
       // Ловушка: этот код не должен никогда вызываться
      } finally {
          fputs( $fh, "Конец\n" );
          fclose( $fh );
```

```
        }
    }
}
```

Поскольку код последней записи в журнал и вызов функции `fclose()` помещены в блок `finally`, они будут выполняться всегда, даже в случае возникновения исключения `FileException` и повторной его генерации в блоке `catch`. Ниже приведен фрагмент журнала при возникновении исключения `FileException`.

```
Начало
Файловая ошибка
Конец
```

На заметку. Блок кода `finally` запускается, если в блоке `catch` будет повторно сгенерировано исключение либо будет выполнен оператор `return`, возвращающий значение в вызывающую программу. Если же в блоке `try` или `catch` будет вызвана функция `die()` или `exit()` для завершения программы, блок кода `finally` не запустится.

Завершенные классы и методы

Наследование открывает большие возможности для широкого поля действий в пределах иерархии класса. Вы можете переопределить класс или метод, чтобы вызов в клиентском методе приводил к совершенно разным результатам, в зависимости от типа объекта, переданного 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, используйте ключевое слово `final`.

Ключевое слово `final` позволяет положить конец наследованию. Для завершенного класса нельзя создать подкласс. А завершенный метод нельзя переопределить.

Давайте объявим класс завершенным.

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

А теперь попытаемся создать подкласс класса `Checkout`.

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

Это приведет к ошибке.

```
PHP Fatal error: Class IllegalCheckout may not inherit from
final class (Checkout) in...[8]
```

Мы можем несколько смягчить ситуацию, объявив завершенным только метод в классе `Checkout`, а не весь класс. Ключевое слово `final` должно стоять перед любыми другими модификаторами, такими как `protected` или `static`.

[8] Класс `IllegalCheckout` не может быть унаследован от завершенного класса `Checkout`. — *Примеч. ред.*

```
class Checkout {
   final function totalize() {
      // Вычисление итоговых данных
   }
}
```

Теперь мы можем создать подкласс класса `Checkout`, но любая попытка переопределить метод `totalize()` приведет к неустранимой ошибке.

```
class IllegalCheckout extends Checkout {
   final function totalize() {
      // Зафиксируем алгоритм расчета итоговых данных
   }
}
```

```
Fatal error: Cannot override final method checkout::totalize() in...[9]
```

В хорошем объектно-ориентированном коде обычно во главу угла ставится строго определенный интерфейс. Но за этим интерфейсом могут скрываться разные реализации. Разные классы или сочетания классов могут соответствовать общим интерфейсам, но при этом вести себя по-разному в разных ситуациях. Объявляя класс или метод завершенным, вы тем самым ограничиваете эту гибкость. В некоторых случаях это выгодно, и мы рассмотрим такие случаи далее в книге. Но прежде чем объявлять что-либо завершенным, следует серьезно подумать. Действительно ли нет таких ситуаций, в которых переопределение было бы полезным? Конечно, вы всегда можете передумать, но внести изменения впоследствии может быть нелегко, например, если вы распространяете библиотеку для совместного использования. Поэтому будьте осторожны, используя ключевое слово `final`.

Работа с методами-перехватчиками

В PHP предусмотрены встроенные методы-перехватчики, которые могут перехватывать сообщения, посланные неопределенным (т.е. несуществующим) методам или свойствам. Это свойство называется также *перегрузкой* (overloading), но поскольку этот термин в Java и C++ означает нечто совершенно другое, я думаю, будет лучше использовать термин "перехват" (interception).

В PHP 5 поддерживается три встроенных метода-перехватчика. Как и в случае метода `__construct()`, вызов этих методов происходит неявно, когда удовлетворяются соответствующие условия. Эти методы описаны в табл. 4.2.

Методы `__get()` и `__set()` предназначены для работы со свойствами, которые не были объявлены в классе (или его родителе).

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 вернет метод `__get()`, будет отослано обратно клиенту, как будто искомое свойство существует с этим значением. Рассмотрим короткий пример.

```
class Person {
   function __get( $property ) {
      $method = "get{$property}";
      if ( method_exists( $this, $method ) ) {
         return $this->$method();
```

[9] Нельзя переопределять завершенный метод `checkout::totalize()`. — *Примеч. ред.*

```
    }
  }

  function getName() {
    return "Иван";
  }

  function getAge() {
    return 44;
  }
}
```

Таблица 4.2. Методы-перехватчики[10]

Метод	Описание
`__get( $property )`	Вызывается при обращении к неопределенному свойству
`__set( $property, $value )`	Вызывается, когда неопределенному свойству присваивается значение
`__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

Когда клиентский код пытается получить доступ к неопределенному свойству, вызывается метод `__get()`, в котором перед именем переданного ему свойства добавляется строка `"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, мы вызываем его и передаем возвращенное им значение клиентскому коду. Поэтому, если клиентский код запрашивает свойство `$name`

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

метод `getName()` вызывается неявно и выводится следующая строка.

```
Иван
```

Если же метод не существует, то ничего не происходит. Свойству, к которому пользователь пытается обратиться, присваивается значение **NULL**.

Метод `__isset()` работает аналогично методу `__get()`. Он вызывается после того, как в клиентском коде вызывается функция `isset()` и ей в качестве параметра передается имя неопределенного свойства. Рассмотрим пример расширения класса `Person`.

```
function __isset( $property ) {
  $method = "get{$property}";
```

[10] Обратите внимание: названия этих методов начинаются с двух символов подчеркивания. — *Примеч. ред*

```
        return ( method_exists( $this, $method ) );
}
```

А теперь предусмотрительный пользователь может проверить свойство, прежде чем работать с ним.

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

Метод `__set()` вызывается, когда клиентский код пытается присвоить значение неопределенному свойству. При этом передаются два аргумента: имя свойства и значение, которое клиентский код пытается присвоить. Затем вы можете решить, как работать с этими аргументами. Давайте продолжим расширение класса `Person`.

```
class Person {
   private $_name;
   private $_age;

   function __set( $property, $value ) {
      $method = "set{$property}";
      if ( method_exists( $this, $method ) ) {
         return $this->$method( $value );
      }
   }

   function setName( $name ) {
      $this->_name = $name;
      if ( ! is_null( $name ) ) {
         $this->_name = strtoupper($this->_name);
      }
   }

   function setAge( $age ) {
      $this->_age = strtoupper($age);
   }
}
```

В этом примере мы работаем с методами-установщиками (setter), а не с методами-получателями (getter). 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 результате мы можем отфильтровать присваиваемое свойству значение.

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

Итак, если мы создаем объект типа `Person`, а затем пытаемся установить свойство `Person::$name`, то вызывается метод `__set()`, потому что в этом классе не определено свойство `$name`. Методу передаются две строки, содержащие имя свойства и значение, которое мы хотим установить. И уже от нас зависит, что мы сделаем с этой информацией. В данном примере мы создаем новое имя метода, добавив перед именем свойства строку `"set"`. Программа находит метод `setName()`

и запускает его должным образом. Он преобразует входящую строку в символы верхнего регистра и сохраняет ее в реальном свойстве.

```
setLocale(LC_ALL, "ru_RU.CP1251");
$p = new Person();
$p->name = "Иван";
// Реальному свойству $_name присваивается строка 'ИВАН'
```

Как и можно было ожидать, метод `__unset()` является зеркальным отражением метода `__set()`. Он вызывается в случае, когда функции `unset()` передается имя неопределенного свойства. Имя этого свойства и передается методу `__unset()`. С полученной информацией можно делать все что угодно. В приведенном ниже примере значение **null** передается найденному результирующему методу тем же самым способом, который мы использовали при рассмотрении метода `__set()`.

```
function __unset( $property ) {
   $method = "set{$property}";
   if ( method_exists( $this, $method ) ) {
      $this->$method( null );
   }
}
```

Метод `__call()`, вероятно, — самый полезный из всех методов-перехватчиков. Он вызывается, когда 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 клиенту так, как будто оно было возвращено вызванным несуществующим методом.

Метод `__call()` может использоваться для делегирования. *Делегирование* — это механизм, посредством которого один объект может вызвать метод другого объекта. Это чем-то напоминает наследование, когда дочерний класс вызывает метод, реализованный в родительском классе. В случае наследования взаимосвязь между родительским и дочерним классами фиксирована. Поэтому возможность изменить объект-получатель во время выполнения программы означает, что делегирование является более гибким, чем наследование. Чтобы лучше это понять, давайте проиллюстрируем все на примере. Рассмотрим простой класс, предназначенный для форматирования информации, полученной от класса `Person`.

```
class PersonWriter {

   function writeName( Person $p ) {
      print $p->getName() . "\n";
   }

   function writeAge( Person $p ) {
      print $p->getAge() . "\n";
   }
}
```

Конечно, мы можем создать подкласс от этого класса, чтобы выводить данные о классе `Person` различными способами. Вот реализация класса `Person`, в которой используются и объект `PersonWriter`, и метод `__call()`.

```
class Person {
   private $writer;
```

```
   function __construct( PersonWriter $writer ) {
      $this->writer = $writer;
   }

   function __call( $methodname, $args ) {
      if ( method_exists( $this->writer, $methodname ) ) {
         return $this->writer->$methodname( $this );
      }
   }

   function getName() { return "Иван"; }
   function getAge() { return 44; }
}
```

Здесь конструктору класса `Person` в качестве аргумента передается объект типа `PersonWriter`, ссылка на который сохраняется в переменной свойства `$writer`. В методе `__call()` используется значение аргумента `$methodname` 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 методу передается ссылка на текущий экземпляр объекта типа `Person`, которая хранится в псевдопеременной `$this`. Поэтому, если клиент вызовет несуществующий в классе `Person` метод

```
$person = new Person( new PersonWriter() );
$person->writeName();
```

будет вызван метод `__call()`. В нем определяется, что в объекте типа `PersonWriter` существует метод с именем `writeName()`, который и вызывается. Это позволяет избежать вызова делегированного метода вручную, как показано ниже.

```
function writeName() {
   $this->writer->writeName( $this );
}
```

Таким образом, класс `Person`, как по волшебству, получил два новых метода класса `PersonWriter`. Хотя автоматическое делегирование избавляет вас от рутинной работы по однотипному кодированию 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. Причина в том, что логика, лежащая в основе взаимодействия между делегирующим классом и целевым объектом, может быть непонятной. Ведь она скрыта в таких методах, как `__call()`, а не явно задана отношениями наследования или уточнениями типа аргументов методов. Методы-перехватчики имеют свою область применения, но использовать их следует с осторожностью. И в классах, где используются эти методы, факт такого применения необходимо очень четко и ясно документировать.

К вопросам делегирования и рефлексии мы вернемся позже в данной книге.

Методы-перехватчики `__get()` и `__set()` можно также использовать для поддержки составных свойств (composite properties). Они могут создать определенные удобства для программиста клиентского кода. В качестве примера давайте представим, что в классе `Address` сохраняется информация о номере дома и названии улицы. Поскольку в конечном итоге данные из этого класса будут сохранены в соответствующих полях базы данных, то такое разделение адреса на улицу и номер дома имеет определенный смысл. Однако в случае, если часто требуется неразделенная

информация об адресе, содержащая в одной строке и номер дома, и название улицы, вам необходимо позаботиться о том, кто будет использовать ваш класс в своих программах. Ниже приведен пример класса, поддерживающий составное свойство `Address::$streetaddress`.

```
class Address {
   private $number;
   private $street;

   function __construct( $maybenumber, $maybestreet=null ) {
      if ( is_null( $maybestreet ) ) {
         $this->streetaddress = $maybenumber;
      } else {
         $this->number = $maybenumber;
         $this->street = $maybestreet;
      }
   }

   function __set( $property, $value ) {
      if ( $property === "streetaddress" ) {
         if ( preg_match( "/^(\d+.*?)[\s,]+(.+)$/", $value, $matches ) ) {
            $this->number = $matches[1];
            $this->street = $matches[2];
         } else {
            throw new Exception("Ошибка в адресе: '{$value}'");
         }
      }
   }

   function __get( $property ) {
      if ( $property === "streetaddress" ) {
         return $this->number." ".$this->street;
      }
   }
}

$address = new Address( "441b Bakers Street" );
print "Адрес: {$address->streetaddress}\n";

$address = new Address( 15, "Albert Mews" );
print "Адрес: {$address->streetaddress}\n";

$address->streetaddress = "34, West 24th Avenue";
print "Адрес: {$address->streetaddress}\n";
```

При попытке обращения к несуществующему свойству `Address::$streetaddress` будет вызван метод `__call()`. В нем проверяется имя составного свойства `"street address"`, которое мы должны поддерживать в классе. Перед тем как установить значения свойств `$number` и `$street`, мы должны убедиться, что переданный нам адрес соответствует ожидаемому шаблону и может быть корректно проанализирован. После этого нужно извлечь значения свойств из строки адреса, которую передал пользователь. Для нашего примера я задал простое правило. Адрес может быть корректно проанализирован, если он начинается с цифры, после которой следует пробел или запятая, а затем — название улицы. Благодаря использованию обратных ссылок в регулярном выражении, после удачного анализа строки в массиве

$matches будут находиться нужные мне данные, которые можно присвоить свойствам $number и $street. Если же строка адреса проверку не прошла, в программе генерируется исключение. Поэтому, когда строка адреса наподобие "441b Bakers Street" присваивается свойству Address::$streetaddress, на самом деле будут присвоены значения свойствам $number и $street, которые были выделены из этой строки. В этом можно убедиться с помощью функции print_r().

```
$address = new Address( "441b Bakers Street" );
print_r( $address );
```

```
Address Object
(
    [number:Address:private] => 441b
    [street:Address:private] => Bakers Street
)
```

По сравнению с методом __set() метод __get() очень простой. При попытке доступа к свойству Address::$streetaddress будет вызван метод __get(). В моей реализации в нем сначала проверяется имя свойства "streetaddress", и, если оно совпадает, в вызывающую программу возвращается строка, составленная из двух значений свойств $number и $street.

Определение методов деструктора

Как мы уже видели, при создании экземпляра объекта автоматически вызывается метод __construct(). В PHP 5 также существует метод __destruct(). Он вызывается непосредственно перед тем, как объект отправляется на "свалку", т.е., я хочу сказать, перед тем как он удаляется из памяти. Вы можете использовать этот метод для выполнения завершающей очистки объекта, если это необходимо.

Например, предположим, что класс после получения специальной команды сохраняет данные объекта в базе данных. Тогда метод __destruct() можно использовать для того, чтобы гарантированно сохранить данные объекта перед его удалением из памяти. Добавим деструктор в класс Person, как показано ниже.

```
class Person {
   private $name;
   private $age;
   private $id;

   function __construct( $name, $age ) {
      $this->name = $name;
      $this->age = $age;
   }

   function setId( $id ) {
      $this->id = $id;
   }

   function __destruct() {
      if ( ! empty( $this->id ) ) {
      // Сохраним данные объекта Person
      print "Сохранение объекта person\n";
```

```
        }
    }
}
```

Метод `__destruct()` вызывается каждый раз, когда объект `Person` удаляется из памяти. Это происходит при вызове функции `__unset()`, которой передается ссылка на удаляемый объект, а также в случае, когда в текущем процессе не остается никаких ссылок на наш объект. Поэтому, если мы создадим объект `Person`, а затем решим его уничтожить, то увидим, как на самом деле работает метод `__destruct()`.

```
$person = new Person( "Иван", 44 );
$person->setId( 343 );
unset( $person );
// Выводится: "Сохранение объекта person"
```

Хотя такие приемы выглядят очень занимательно, следует добавить предостережение. Методы-перехватчики, такие как `__call()`, `__destruct()` и им подобные, иногда называют *магическими*. Если вы когда-либо читали произведения жанра "фэнтези", то, наверное, знаете, что магия — это не всегда хорошо. Магия случайна и непредсказуема. Магия нарушает правила. Магия приносит скрытые расходы.

Например, в случае 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 использовать класс `Person`, не разобравшись, что к чему. Он не заметил метод `__destruct()` и собирается создать ряд экземпляров объекта `Person`. Передавая значения конструктору, он назначает тайную и обидную кличку генерального директора свойству `$name` и устанавливает для свойства `$age` значение 150. Разработчик прогоняет тестовый сценарий несколько раз, используя красочные сочетания имени и возраста.

А на следующее утро начальник вызывает его к себе в кабинет и просит объяснить, почему в базе данных содержатся оскорбительные данные в адрес служащих компании. Мораль сей басни такова: не доверяйте магии.

Копирование объектов с помощью метода `__clone()`

В PHP 4 копирование объекта выполнялось очень просто — достаточно было присвоить значение одной объектной переменной другой.

```
class CopyMe {}

$first = new CopyMe();
$second = $first;
// В PHP 4 переменные $second и $first ссылаются на два разных объекта
// Начиная с PHP 5 переменные $second и $first ссылаются на один объект
```

Но эта простота часто служила источником множества ошибок, поскольку копии объекта случайно "размножались" при присвоении значений переменным, вызове методов и возврате объектов. Еще больше ухудшало ситуацию то, что не существовало способа протестировать две переменные, чтобы понять, относятся ли они к одному и тому же объекту. С помощью операторов эквивалентности можно было проверить идентичность значений всех полей у двух объектов (==) либо являются ли

обе переменные объектами (===). Но в результате этих проверок нельзя было понять, указывают ли обе переменные на один и тот же объект.

В PHP 5 объекты всегда присваиваются и передаются по ссылке. Это означает, что если предыдущий пример запустить в PHP 5, переменные `$first` и `$second` будут содержать ссылки на один и тот же объект, а не на две его копии. И хотя в большинстве случаев при работе с объектами это именно то, что нам нужно, будут ситуации, когда нам понадобится получить копию объекта, а не ссылку на объект.

В PHP 5 для этой цели предусмотрено ключевое слово `clone`. Оно применяется к экземпляру объекта и создает дополнительную копию.

```
class CopyMe {}
$first = new CopyMe();
$second = clone $first;
// В PHP 5 и более поздних версиях переменные $second и $first
//  ссылаются на два разных объекта
```

И здесь вопросы, касающиеся копирования объектов, только начинаются. Рассмотрим класс `Person`, который мы создали в предыдущем разделе. Стандартная копия объекта `Person` содержит идентификатор (свойство `$id`), который при реализации полноценного приложения будет использоваться для нахождения нужной строки в базе данных. Если мы разрешим копировать это свойство, то получим два различных объекта, ссылающихся на один и тот же источник данных, причем, вероятно, не тот, который мы хотели, когда создавали копию. Изменение в одном объекте повлияет на другой и наоборот.

К счастью, при использовании перед объектом ключевого слова `clone` мы можем контролировать то, что копируется. Для этого следует реализовать специальный метод `__clone()` (обратите внимание на два символа подчеркивания, с которых начинаются имена всех встроенных методов). Метод `__clone()` вызывается автоматически, когда для копирования объекта используется ключевое слово `clone`.

При реализации метода `__clone()` важно понимать контекст, в котором работает данный метод. Метод `__clone()` работает в контексте *скопированного* объекта, а не исходного. Давайте добавим метод `__clone()` к одной из версий нашего класса `Person`.

```
class Person {
   private $name;
   private $age;
   private $id;

   function __construct( $name, $age ) {
      $this->name = $name;
      $this->age = $age;
   }

   function setId( $id ) {
      $this->id = $id;
   }

   function __clone() {
      $this->id = 0;
   }
}
```

Когда оператор `clone` вызывается для объекта типа `Person`, создается его новая поверхностная (shallow) копия, и для *нее* вызывается метод `__clone()`. Это означает,

что все изменения значений свойств, выполняемые в методе `__clone()`, отразятся только на новой копии объекта. Старые значения, полученные из исходного объекта, будут затерты. В данном случае мы гарантируем, что свойство `$id` скопированного объекта устанавливается равным нулю.

```
$person = new Person( "Петр", 44 );
$person->setId( 343 );
$person2 = clone $person;
// $person2 :
// name: "Петр"
// age: 44
// id: 0.
```

Поверхностное копирование гарантирует, что значения элементарных свойств будут скопированы из старого объекта в новый. Однако при этом будут скопированы и ссылки на другие объекты, которые были присвоены свойствам. Возможно, это не совсем то, что вы хотите, когда клонируете объект. Предположим, мы добавим к объекту `Person` свойство-объект `Account`. В этом объекте хранятся данные о состоянии счета, которые мы тоже хотим скопировать в клонированный объект. Однако при этом мы не хотим, чтобы в обоих объектах `Person` сохранялись ссылки на *один и тот же* счет.

```
class Account {
   public $balance;

   function __construct( $balance ) {
      $this->balance = $balance;
   }
}

class Person {
   private $name;
   private $age;
   private $id;
   public $account;

   function __construct( $name, $age, Account $account ) {
      $this->name = $name;
      $this->age = $age;
      $this->account = $account;
   }

   function setId( $id ) {
      $this->id = $id;
   }

   function __clone() {
      $this->id = 0;
   }
}

$person = new Person( "Иван", 44, new Account( 200 ) );
$person->setId( 343 );
$person2 = clone $person;
// Добавим $person немного денег
$person->account->balance += 10;
```

```
// Это изменение увидит и $person2
print $person2->account->balance;
```

В результате будет выведено следующее.

```
210
```

Объектная переменная `$person`, которую мы сделали общедоступной ради компактности, содержит ссылку на объект типа `Account`. Как вы знаете, мы обычно ограничиваем доступ к свойству, создавая в случае необходимости метод доступа. Когда создается клон, он содержит ссылку на тот же самый объект `Account`, на который ссылается и `$person`. Мы демонстрируем это, добавляя немного денег к балансу объекта `Account` переменной `$person`, а затем выводя ее значение через переменную `$person2`.

Если мы не хотим, чтобы после выполнения операции клонирования в новом объекте осталась ссылка на старое свойство-объект, последнее нужно клонировать явно в методе `__clone()`.

```
function __clone() {
   $this->id = 0;
   $this->account = clone $this->account;
}
```

Определение строковых значений для объектов

Еще одна функция, введенная в PHP 5 явно под влиянием Java, — метод `__toString()`. До выхода версии PHP 5.2 при печати объекта с помощью кода

```
class StringThing {}

$st = new StringThing();
print $st;
```

выводилась следующая строка.

```
Object id #1
```

А начиная с версии PHP 5.2 этот код вызовет следующую ошибку.

```
PHP Catchable fatal error: Object of class StringThing could not be
converted to string in...[11]
```

Реализовав метод `__toString()`, вы можете контролировать то, какую информацию будут выводить объекты при печати. 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. Давайте добавим версию метода `__toString()` к минимальной реализации класса `Person`.

```
class Person {

   function getName() { return "Иван"; }
   function getAge()  { return 44; }

   function __toString() {
```

[11] Неустранимая обрабатываемая ошибка: объект класса `StringThing` не может быть преобразован в строку в... — *Примеч. ред.*

```
        $desc  = $this->getName();
        $desc .= " (возраст " . $this->getAge() . " лет)";
        return $desc;
    }
}
```

Теперь при печати объекта `Person`

```
$person = new Person();
print $person;
```

получим следующее.

```
Иван (возраст 44 лет)
```

Метод `__toString()` особенно полезен для записи информации в журнальные файлы и для выдачи сообщений об ошибках, а также для классов, основная задача которых — передавать информацию. Например, при выводе на печать объекта типа `Exception` можно выдавать краткий отчет о происшедших исключениях, реализовав в нем метод `__toString()`.

Функции обратного вызова, анонимные функции и механизм замыканий

Несмотря на то что анонимные функции относятся не только к области объектно-ориентированного программирования, они настолько полезны, что я не мог их здесь не упомянуть. Все дело в том, что они активно используются в объектно-ориентированных приложениях наряду с функциями обратного вызова. Более того, в этой области за последнее время было сделано несколько довольно интересных открытий.

Для начала давайте определим несколько классов.

```
class Product {
    public $name;
    public $price;

    function __construct( $name, $price ) {
        $this->name = $name;
        $this->price = $price;
    }
}

class ProcessSale {
    private $callbacks;

    function registerCallback( $callback ) {
        if ( ! is_callable( $callback ) ) {
            throw new Exception( "Функция обратного вызова — невызываемая!" );
        }

        $this->callbacks[] = $callback;
    }

    function sale( $product ) {
        print "{$product->name}: обрабатывается... \n";
```

```
        foreach ( $this->callbacks as $callback ) {
            call_user_func( $callback, $product );
        }
    }
}
```

Этот код предназначен для запуска нескольких функций обратного вызова. В нем определены два класса. В классе `Product` просто сохраняются значения свойств `$name` и `$price`. Для компактности я объявил их открытыми. Не забудьте в реальном проекте сделать их закрытыми или защищенными и создать для этих свойств методы доступа. В классе `ProcessSale` определены два метода. Методу `registerCallback()` передается обычная скалярная переменная безо всяких уточнений. После ее проверки она добавляется в массив функций обратного вызова `$callbacks`. Процесс тестирования выполняется с помощью встроенной функции `is_callable()`. В результате гарантируется, что методу `registerCallback()` будет передано имя функции, которую можно вызвать с помощью таких функций, как `call_user_func()` или `array_walk()`.

Методу `sale()` передается объект типа `Product`. Метод выводит об этом информацию и затем в цикле выполняет перебор элементов массива `$callback`. Каждый элемент вместе с объектом типа `Product` передается функции `call_user_func()`, которая, собственно, и вызывает код, написанный пользователем. Все приведенные ниже примеры будут работать в контексте одной инфраструктуры.

Чем же так полезны функции обратного вызова? Они позволяют в процессе выполнения программы добавлять компоненту новые функциональные возможности, которые напрямую не были связаны с его первоначальной основной функциональностью. Предусмотрев в объекте работу с функциями обратного вызова, вы тем самым позволите другим программистам расширять функциональность вашего кода, причем при таких обстоятельствах, о которых раньше вы даже и не догадывались.

Представьте себе, например, что через какое-то время пользователь класса `ProcessSale` захочет создать журнал продаж. Если у пользователя будет доступ к исходному коду класса, он может добавить код фиксации сделок прямо в метод `sale()`. Однако это не всегда удачное решение. Если этот пользователь не является владельцем пакета, в котором определен класс `ProcessSale`, то все его исправления будут затерты, когда выйдет его обновленная версия. И даже если он является владельцем пакета, все равно, добавлять в метод `sale()` дополнительные ветки кода, решающие случайные задачи, неразумно. В конечном итоге это приведет к изменению зоны ответственности класса и потенциально может снизить возможность его повторного использования в других проектах. Я еще вернусь к этой теме в других главах книги.

Несмотря на это, по счастливой случайности, я заложил в класс `ProcessSale` возможность обратного вызова. Ниже я создал функцию обратного вызова, которая фиксирует все сделки в журнале продаж.

```
$logger = create_function( '$product',
                           'print " Записываем...  ({$product->name})\n";' );
$processor = new ProcessSale();
$processor->registerCallback( $logger );

$processor->sale( new Product( "Туфли", 6 ) );
print "\n";
$processor->sale( new Product( "Кофе", 6 ) );
```

Здесь для создания функции обратного вызова я воспользовался функцией `create_function()`. Как видите, ей передаются два параметра. Сначала указывается список параметров функции, а затем — тело самой функции. В результате у

нас получилась конструкция, которую часто называют анонимной, поскольку при создании мы не присвоили ей имя, как в обычных функциях. Вместо этого ссылка на вновь созданную функцию сохраняется в переменной, которую затем можно передать в качестве параметра другим функциям и методам. Собственно, это я и сделал в приведенном выше фрагменте кода. Я сохранил ссылку на анонимную функцию в переменной `$logger`, которую затем передал в качестве параметра методу `ProcessSale::registerCallback()`. В конце листинга я создал пару объектов, описывающих товары, и передал их по очереди методу `sale()`. О том, что произойдет дальше, вы уже, наверное, догадались. Каждая сделка по продаже будет обработана (на самом деле будет выведено обычное сообщение о товаре), после чего будут вызваны все установленные функции обратного вызова, как показано ниже.

```
Туфли: обрабатывается
Записываем (Туфли)

Кофе: обрабатывается
Записываем (Кофе)
```

Давайте еще раз проанализируем пример кода с функцией `create_function()`. Вы обратили внимание на то, насколько уродливо он выглядит? При помещении выполняемого кода в строку всегда возникает головная боль. Для начала вам нужно выполнить экранирование всех символов `'$'` и `'?'`, которые встречаются в тексте программы. Более того, по мере роста тела функции обратного вызова ее и в самом деле будет все труднее и труднее проанализировать и понять. Было бы просто замечательно, если бы существовал какой-нибудь более элегантный способ создания таких функций. Начиная с PHP 5.3 такой способ существует! Теперь вы можете просто объявить функцию, как обычно, а затем присвоить ссылку на нее переменной. И все это — в одном операторе! Ниже приведен предыдущий пример, в котором использован новый синтаксис.

```
$logger2 = function( $product ) {
              print " Записываем ({$product->name})\n";
           };
$processor = new ProcessSale();
$processor->registerCallback( $logger2 );

$processor->sale( new Product( "Туфли", 6 ) );
print "\n";
$processor->sale( new Product( "Кофе", 6 ) );
```

Единственное отличие здесь заключается в способе создания переменной, ссылающейся на анонимную функцию. Как видите, этот код намного понятнее. Я указал в операторе присваивания ключевое слово `function` и не задал имя функции. Обратите внимание на то, что, поскольку в операторе присваивания используется встроенная функция, в конце блока нужно обязательно поместить точку с запятой. Конечно, если ваш код должен работать в одной из предыдущих версий PHP, вам придется продолжать использовать уродливый синтаксис с функцией `create_function()`. Результат работы нового фрагмента кода ничем не будет отличаться от предыдущего.

Само собой разумеется, что функции обратного вызова не обязательно должны быть анонимными. В качестве такой функции вы смело можете использовать имя обычной функции или даже ссылку на метод какого-либо объекта. Ниже приведен пример.

```
class Mailer {
   function doMail( $product ) {
      print " Упаковываем ({$product->name})\n";
   }
}

$processor = new ProcessSale();
$processor->registerCallback( array( new Mailer(), "doMail" ) );

$processor->sale( new Product( "Туфли", 6 ) );
print "\n";
$processor->sale( new Product( "Кофе", 6 ) );
```

Здесь я создал новый класс `Mailer`, содержащий единственный метод `doMail()`. Этому методу передается объект типа `Product`, о котором метод выводит сообщение. При вызове метода `registerCallback()` я передал ему в качестве параметра массив, а не ссылку на функцию обратного вызова, как это было раньше. Первым элементом этого массива является объект типа `Mailer`, а вторым — строка, содержащая имя метода, который мы хотим вызвать. Помните, что в методе `registerCallback()` с помощью функции `is_callable()` выполняется проверка аргумента на предмет того, можно ли его вызвать? Данная функция достаточно интеллектуальна и распознает массивы подобного вида. Поэтому при указании функции обратного вызова в виде массива в первом элементе такого массива должен находиться объект, содержащий вызываемый метод, а имя этого метода помещается в виде строки во второй элемент массива. Таким образом, мы успешно прошли проверку типа аргумента, и вот что выведет программа в результате.

```
Туфли: обрабатывается
Упаковываем (Туфли)

Кофе: обрабатывается
Упаковываем (Кофе)
```

Разумеется, что анонимную функцию можно вернуть из метода, как показано ниже.

```
class Totalizer {
   static function warnAmount() {
      return function( $product ) {
                if ( $product->price > 5 ) {
                   print " покупается дорогой товар: {$product->price}\n";
                }
             };
   }
}

$processor = new ProcessSale();
$processor->registerCallback( Totalizer::warnAmount() );
...
```

В этом примере нет ничего интересного, кроме того, что метод `warnAmount()` используется в качестве фабрики анонимной функции. Тем не менее подобная структура позволяет мне делать нечто гораздо большее, чем просто генерировать анонимную функцию. Она позволяет мне воспользоваться преимуществами *механизма замыкания* (closures). В анонимной функции нового типа могут использоваться переменные, объявленные в другой анонимной функции, находящейся в родитель-

ской области видимости. Данная концепция очень сложна, чтобы понять ее сразу. Вкратце это можно объяснить так, как будто анонимная функция запоминает контекст, в котором она была создана. Предположим, я хочу сделать так, чтобы метод `Totalizer::warnAmount()` выполнял следующее. Во-первых, я хочу, чтобы ему можно было передавать пороговое значение стоимости товаров. Во-вторых, я хочу, чтобы он подсчитывал стоимость (т.е. сумму цен) проданного товара. И когда стоимость проданного товара превысит установленный порог, функция должна выполнить некоторые действия (в нашем случае, как вы уже можете догадаться, она просто выведет соответствующее сообщение).

Чтобы анонимная функция могла воспользоваться переменными, определенными в родительской области видимости, используется ключевое слово `use`, как показано в примере ниже.

```
class Totalizer {
   static function warnAmount( $amt ) {
      $count=0;
      return function( $product ) use ( $amt, &$count ) {
                     $count += $product->price;
                     print " сумма: $count\n";

                     if ( $count > $amt ) {
                        print " Продано товаров на сумму: {$count}\n";
                     }
             };
   }
}

$processor = new ProcessSale();
$processor->registerCallback( Totalizer::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()`, указаны две переменные. Первая из них — это `$amt`, которая является аргументом, переданным методу `warnAmount()`. Вторая — замкнутая переменная `$count`. Она объявлена в теле метода `warnAmount()`, и начальное ее состояние равно нулю. Обратите внимание на то, что перед именем переменной `$count` в директиве `use` я указал символ амперсанда `'&'`. Это означает, что данная переменная будет передаваться в анонимную функцию по ссылке, а не по значению. Дело в том, что в теле анонимной функции я добавляю к ней цену товара и затем сравниваю новую сумму со значением переменной `$amt`. Если будет достигнуто пороговое значение, выводится соответствующее сообщение, как показано ниже.

```
Туфли: обрабатывается
сумма: 6

Кофе: обрабатывается
сумма: 12

Продано товаров на сумму: 12
```

В этом примере было показано, что значение переменной `$count` сохраняется между вызовами функции обратного вызова. Обе переменные, и `$count` и `$amt`, оста-

ются связанными с этой функцией, поскольку они указаны в контексте ее объявления и перечислены в директиве `use`.

Резюме

В этой главе мы попытались осветить тему более сложных объектно-ориентированных возможностей PHP. С некоторыми из них вы еще будете встречаться по мере чтения книги. В частности, мы будем часто возвращаться к абстрактным классам, исключениям и статическим методам.

В следующей главе мы немного отойдем от описания встроенных возможностей объектов и рассмотрим классы и функции, предназначенные для облегчения работы с объектами.

Глава 5

Средства для работы с объектами

Как мы уже видели, в PHP поддержка объектно-ориентированного программирования осуществляется через конструкции языка, такие как классы и методы. Кроме того, в PHP предусмотрен большой набор функций и классов, предназначенных для облегчения работы программиста с объектами.

В данной главе мы рассмотрим некоторые средства и методики, которые можно использовать для систематизации, тестирования объектов и классов и выполнения операций над ними.

В этой главе будут рассмотрены следующие темы.

- *Пакеты:* организация кода в логические структуры.
- *Пространства имен:* начиная с PHP версии 5.3 вы можете инкапсулировать элементы кода в самостоятельные логические модули.
- *Пути включения файлов:* позволяют указать места централизованного хранения библиотечного кода.
- *Функции для работы с классами и объектами:* предназначены для тестирования объектов, классов, свойств и методов.
- *Reflection API:* многофункциональный набор встроенных классов, который обеспечивает беспрецедентный доступ к информации о классе во время выполнения программы.

PHP и пакеты

Пакет — это набор связанных классов, обычно сгруппированных вместе некоторым способом. Пакеты можно использовать для разделения частей системы на логические компоненты. В некоторых языках программирования пакеты распознаются формально, и для них создаются различные пространства имен. В PHP такого понятия, как пакет, никогда не существовало, однако начиная с PHP 5.3 в нем уже поддерживаются пространства имен. Данная возможность будет рассмотрена в следующем разделе.

Поскольку всем нам все еще приходится иметь дело с устаревшим кодом, я должен хотя бы вкратце описать устаревший способ организации классов в пакетоподобные структуры.

Пакеты и пространства имен в PHP

Несмотря на то что, по сути, в PHP концепция пакетов не поддерживалась, разработчики традиционно использовали различные схемы именования и файловую систему для организации своего кода в пакетоподобные структуры. Чуть ниже я опишу способ использования файлов и каталогов операционной системы для организации кода. А пока что я познакомлю вас со схемами именования, а также с новой, но связанной с этим, возможностью — поддержкой пространств имен.

До появления PHP 5.3 разработчики были вынуждены подбирать для файлов и классов своего проекта уникальные имена, поскольку с точки зрения интерпретатора все они находились в одном глобальном контексте. Другими словами, если вы, например, определяли класс `ShoppingBasket`, он сразу же становился доступным всему вашему приложению. Это порождало две большие проблемы. Первая и самая неприятная проблема была связана с потенциальным конфликтом имен в проекте. Наверняка вы считали эту проблему несущественной, правда? В конце концов, все, что вам нужно было сделать, — так это запомнить, что всем классам нужно было назначать уникальные имена! Однако проблема состояла в том, что все чаще и чаще в своих проектах нам приходилось использовать библиотечный код. Это очень хороший подход, поскольку он способствует повторному использованию кода. Но что происходило, если в своем проекте вы использовали конструкцию

```
// my.php
require_once "useful/Outputter1.php"
class Outputter {
   // Вывод данных
}
```

а во включаемом файле было следующее?

```
// useful/Outputter1.php
class Outputter {
//
}
```

Ну, наверное, вы уже догадались, правда? Это приводило к неустранимой ошибке.

```
Fatal error: Cannot redeclare class Outputter in ../useful/Outputter1.php
on line 3[1]
```

Естественно, существовал традиционный обходной маневр, о котором я сейчас расскажу. Он заключался в том, что перед именем класса нужно было указать имя пакета, чтобы в результате получились уникальные имена классов.

```
// my.php
require_once "useful/Outputter2.php";
class my_Outputter {
// Вывод данных
}

// useful/Outputter2.php
class useful_Outputter {
//
}
```

Описанная выше проблема приводила к тому, что по мере роста проекта имена классов становились все длиннее и длиннее. И хотя ничего страшного в этом не

[1] Нельзя переопределять класс `Outputter`. — *Примеч. ред.*

было, но в результате страдала читабельность кода и вам было все труднее и труднее держать в голове гигантские имена классов. Кроме того, огромное количество времени уходило на исправление опечаток в таких именах.

С этой проблемой вам предстоит столкнуться и сейчас, поскольку большинству из нас часто приходится сопровождать в той или иной форме устаревший код на протяжении длительного периода времени. По этой причине я еще вернусь к устаревшему способу организации пакетов в PHP чуть ниже в этой главе.

Спасительные пространства имен

Пространства имен появились в PHP 5.3. По существу они представляют собой корзину, в которую вы можете поместить классы, функции и переменные. В пределах одного пространства имен вы можете обращаться к ним безо всякого уточнения. Чтобы обратиться к этим элементам за пределами пространства имен, вам придется либо импортировать в код целое пространство имен, либо использовать ссылку на него.

Запутались? Поясним все на примере. Ниже я переписал предыдущий пример с использованием пространств имен.

```
namespace my;
require_once "useful/Outputter3.php";
class Outputter {
// Вывод данных
}

// useful/Outputter3.php
namespace useful;
class Outputter {
//
}
```

Обратите внимание на ключевое слово `namespace`. Как вы уже, наверное, догадались, оно устанавливает указанное пространство имен. При использовании пространств имен в коде их объявление должно быть выполнено в первом операторе в файле. В приведенном выше примере я создал два пространства имен: `my` и `useful`. Однако обычно в программах используются более длинные пространства имен. Как правило, они начинаются с названия организации или проекта, а далее указывается название пакета. В PHP допускаются вложенные пространства имен. Для их разделения на уровни используется символ обратной косой черты `'\'`, как показано ниже.

```
namespace com\getinstance\util;
   class Debug {
      static function helloWorld() {
         print "Привет из класса Debug!\n";
      }
   }
```

При помещении этого кода в хранилище от меня требовалось указать название одного из моих доменов: `getinstance.com`. Поэтому я решил воспользоваться именем этого домена как собственным пространством имен. Этот прием хорошо знаком разработчикам на Java, которые обычно присваивают своим пакетам подобные имена. При этом они изменяют порядок следования поддоменов от самого общего к частному. После создания собственного хранилища кода я должен определить по-

мещаемые в него пакеты. В данном случае после имени домена я использовал ключевое слово `util`.

Как же теперь я могу вызвать метод? На самом деле все зависит от того, из какого контекста я собираюсь это сделать. Если я вызываю метод из текущего пространства имен, то все происходит, как и раньше, т.е. метод вызывается напрямую, как показано ниже.

```
Debug::helloWorld();
```

При этом имя класса остается не полностью определенным. В самом деле, поскольку мы и так находимся в пространстве имен `com\getinstance\util`, нам не нужно указывать перед ним какой-либо путь. Другое дело, если нам нужно вызвать метод нашего класса за пределами текущего пространства имен. Тогда нужно указать следующее.

```
com\getinstance\util\Debug::helloWorld();
```

Как вы думаете, что выведет приведенная ниже программа?

```
namespace main;

com\getinstance\util\Debug::helloWorld();
```

Это хитрый вопрос. На самом деле мы получим следующее.

```
PHP Fatal error: Class 'main\com\getinstance\util\Debug' not found in.../
listing5.04.php on line 12[2]
```

Причина в том, что я использовал в своем коде относительное имя пространства имен. Интерпретатор PHP будет искать в текущем пространстве имен `main` имя `com\getinstance\util` и не сможет его найти. Чтобы решить проблему, мы должны указать абсолютное имя пространства имен. Все происходит примерно так же, как и при указании абсолютных путей к файлам и URL. Вам нужно поместить перед названием пространства имен символ-разделитель, как показано ниже.

```
namespace main;

\com\getinstance\util\Debug::helloWorld();
```

Вот этот начальный символ обратной косой черты и говорит интерпретатору о том, что поиск классов нужно начинать не с текущего пространства имен, а с корня.

Но разве введение пространств имен не должно было сократить имена классов и упростить для вас ввод кода программы? Само собой, имя класса `Debug` очень краткое и понятное, но такой "многословный" вызов метода этого класса получился потому, что мы работаем в рамках старого пространства имен. Все можно упростить, воспользовавшись ключевым словом `use`. Оно позволяет создать псевдонимы других пространств имен в текущем пространстве, как показано ниже.

```
namespace main;
use com\getinstance\util;

util\Debug::helloWorld();
```

При этом импортируется пространство имен `com\getinstance\util`, и ему неявно назначается псевдоним `util`. Обратите внимание на то, что перед его названием я не использую символ обратной косой черты. Все дело в том, что поиск указанного после ключевого слова `use` пространства имен выполняется из глобального, а не те-

[2] Класс `'main\com\getinstance\util\Debug'` не найден в... — *Примеч. ред.*

кущего контекста. Если из нового пространства имен мне нужен только класс Debug, я могу импортировать только его, а не все пространство, как показано ниже.

```
namespace main;
use com\getinstance\util\Debug;

Debug::helloWorld();
```

Но что произойдет, если в пространстве имен main уже определен другой класс Debug? Я думаю, вы уже догадались. Ниже приведен фрагмент кода и то, что выведет программа.

```
namespace main;
use com\getinstance\util\Debug;

class Debug {
   static function helloWorld() {
      print "Привет из класса main\Debug";
   }
}

Debug::helloWorld();
```

```
PHP Fatal error: Cannot declare class main\Debug because the name is
already in use in.../listing5.08.php on line 13[3]
```

Итак, я снова наступил на те же грабли, столкнувшись с конфликтом имен классов. К счастью, из подобной ситуации очень легко выбраться, явно указав псевдоним класса, как показано ниже.

```
namespace main;
use com\getinstance\util\Debug as uDebug;

class Debug {
   static function helloWorld() {
      print "Привет, из класса main\Debug";
   }
}

uDebug::helloWorld();
```

Воспользовавшись директивой as в операторе use, я явно изменил псевдоним класса Debug на uDebug.

Если вы пишете код в текущем пространстве имен и хотите обратиться к классу, находящемуся в глобальном (без имени) пространстве, просто укажите перед его именем обратную косую черту. Ниже приведен пример метода helloWorld() класса Lister, определенного в глобальном пространстве.

```
// global.php: пространство имен не используется
class Lister {
   public static function helloWorld() {
      print "Привет из глобального пространства\n";

   }
}
```

[3] Нельзя определить класс main\Debug, потому что это имя уже используется в... — *Примеч. ред.*

А вот пример кода, написанного в локальном пространстве имен, в котором используется этот класс.

```
namespace com\getinstance\util;
require_once 'global.php';

class Lister {
   public static function helloWorld() {
      print "Привет из " . __NAMESPACE__ . "\n";
   }
}

Lister::helloWorld();  // Доступ к локальному классу
\Lister::helloWorld(); // Доступ к глобальному классу
```

В локальном пространстве имен определен собственный класс `Lister`. При использовании не полностью определенного имени класса обращение происходит к его локальной версии. Если имя класса полностью определено (т.е. начинается с обратной косой черты), обращение происходит к классу, находящемуся в глобальном пространстве. Вот что выведет предыдущий фрагмент программы.

```
Привет из com\getinstance\util
Привет из глобального пространства
```

Это очень ценный пример, поскольку в нем показано, как пользоваться константой `__NAMESPACE__`. Она содержит имя текущего пространства имен, что может очень пригодиться при отладке кода.

С помощью описанного выше синтаксиса можно определить в одном файле несколько пространств имен. Можно также использовать альтернативный синтаксис, в котором используются фигурные скобки после названия пространства имен.

```
namespace com\getinstance\util {
   class Debug {
      static function helloWorld() {
         print "Привет из Debug\n";
      }
   }
}

namespace main {
   \com\getinstance\util\Debug::helloWorld();
}
```

Если вам приходится объединять несколько пространств имен в одном файле, то второй синтаксис предпочтительнее, хотя обычно рекомендуется в одном файле определять только одно пространство имен.

Синтаксис с фигурными скобками позволяет сделать еще одну полезную вещь — переключиться к глобальному пространству имен в том же файле. Выше я использовал оператор `require_once` для включения в текущий проект кода из глобального пространства. На самом деле я мог бы сделать все то же самое и сохранить весь код в одном файле, если бы воспользовался альтернативным синтаксисом определения пространств имен, как показано ниже.

```
namespace {
   class Lister {
   //...
   }
```

```
}

namespace com\getinstance\util {
   class Lister {
   //...
   }

Lister::helloWorld(); // Доступ к локальному классу
\Lister::helloWorld(); // Доступ к глобальному классу

}
```

Для перехода в глобальное пространство имен я открыл блок `namespace` и не указал его имя.

На заметку. В одном файле нельзя использовать оба синтаксиса определения пространств имен. Вы должны выбрать какой-нибудь один и пользоваться им.

Использование файловой системы для имитации пакетов

Независимо от используемой версии PHP вы можете упорядочивать классы с помощью файловой системы, которая отдаленно напоминает пакетную структуру. Например, вы можете создать каталоги `util` и `business` и включать находящиеся в них файлы классов с помощью оператора `require_once()` следующим образом.

```
require_once('business/Customer.php');
require_once('util/WebTools.php');
```

С тем же успехом вы можете также использовать оператор `include_once()`. Единственное различие между операторами `require_once()` и `include_once()` заключается в обработке ошибок. Файл, к которому обратились с помощью оператора `require_once()`, приведет к остановке всего процесса выполнения программы, если в нем окажется ошибка. Такая же ошибка, обнаруженная в результате вызова оператора `include_once()`, приведет просто к выдаче предупреждения и продолжению выполнения включенного файла. Поэтому операторы `require()` и `require_once()` будут самым безопасным выбором для включения библиотечных файлов, а `include()` и `include_once()` пригодятся при создании шаблонов кода.

На заметку. Конструкции `require()` и `require_once()` — на самом деле операторы, а не функции. 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 к тому, что вам будут надоедать педанты, стремящиеся объяснить вашу ошибку.

На рис. 5.1 показано, как выглядят пакеты `util` и `business` в диспетчере файлов Nautilus.

На заметку. Оператору `require_once()` передается путь к файлу, содержащему PHP-код, который будет включен в текущий сценарий и выполнен. Причем включение указанного файла в текущий сценарий выполняется только один раз, т.е. если ранее этот файл был включен в сценарий, то повторное включение не выполняется. Использование подобного однократного подхода особенно полезно при доступе к библиотечному коду, поскольку тем самым предотвращается случайное переопределение классов и функций. Подобная ситуация может возникнуть, если один и тот же файл включается в разные части сценария, выполняющегося в одном процессе с помощью оператора `require()` или `include()`.

Обычно программисты предпочитают пользоваться операторами `require()` и `require_once()`, а не аналогичными им `include()` и `include_once()`. Причина в том, что неустранимая ошибка, обнаружен-

ная в файле, к которому обратились с помощью оператора `require()`, приведет к остановке всего сценария. А такая же ошибка, обнаруженная в файле, к которому обратились с помощью оператора `include()`, приведет к прекращению выполнения включенного файла, но в вызывающем сценарии будет всего лишь сгенерировано предупреждение. Первый, более радикальный, вариант безопаснее.

Существуют некоторые издержки, связанные с использованием оператора `require_once()`, по сравнению с `require()`. Если вам нужно сократить до минимума время выполнения сценария, то лучше использовать оператор `require()`. Как это часто бывает, приходится выбирать между эффективностью и удобством.

Рис. 5.1. Пакеты PHP, организованные с помощью файловой системы

В том, что касается PHP, в такой организации файлов нет ничего особенного. Мы просто помещаем библиотечные сценарии в разные каталоги, что придает нашему проекту четкую организацию. При этом и пространства имен и файловую систему можно использовать параллельно.

Именование в стиле PEAR

Скорее всего, вам не придется использовать пространства имен везде, где только можно. Очень часто работодатели, клиенты или компании-хостеры медленно переходят к новым версиям программного обеспечения. Отчасти это связано с принципом "работает — не ломай!" И даже если вы используете в своем проекте самую последнюю версию PHP, рано или поздно вам придется столкнуться с устаревшим кодом. Если у вас будет время на то, чтобы переписать его и использовать в нем пространства имен, это просто замечательно. Но большинству из нас такая роскошь недоступна.

Как же без такой новой возможности, как пространства имен, можно решить проблему конфликта имен? Выше я уже давал ответ на этот вопрос. Он заключается в использовании общего соглашения об именовании, принятого для пакетов PEAR.

На заметку. PEAR расшифровывается как *PHP Extension and Application Repository* (хранилище расширений и приложений PHP). Это официально поддерживаемый архив пакетов и средств, которые расширяют функциональные возможности PHP. Основные пакеты PEAR включены в дистрибутивный пакет PHP, а другие можно добавить с помощью простой утилиты командной строки. Просмотреть список пакетов PEAR можно по адресу `http://pear.php.net`. А некоторые другие аспекты PEAR будут рассмотрены в главе 15, "Введение в PEAR и Pyrus".

В PEAR для определения пакетов используется файловая система, как уже было описано выше. Затем каждый класс получает имя в соответствии со своим путем, причем имена каталогов отделяются символом подчеркивания.

Например, в PEAR включен пакет с именем XML, в который вложен пакет RPC. В пакете RPC есть файл, который называется `Server.php`. Класс, определенный внутри `Server.php`, не называется `Server`, как можно было ожидать. В противном случае рано или поздно произошел бы конфликт с другим классом `Server`, находящим-

ся в каком-нибудь другом пакете PEAR или в пользовательском коде. Чтобы этого не произошло, класс назван `XML_RPC_Server`. Конечно, в результате имена классов становятся громоздкими и непривлекательными. Но, с другой стороны, код становится более читабельным, потому что имя класса всегда описывает его контекст.

Пути включения файлов

При упорядочении компонентов кода вы можете воспользоваться одним из двух способов. Первый из них я уже описал. Речь идет о размещении файлов проекта по разным каталогам файловой системы. Но за рамками описания остался вопрос о том, как компоненты должны взаимодействовать друг с другом. О включении файлов в проект я уже писал чуть выше. При этом вы должны указать либо относительный (относительно текущего рабочего каталога), либо полный путь к файлу в рамках файловой системы вашего ПК.

В примерах, которые рассматривались до сих пор, мы использовали относительные пути.

```
require_once('business/User.php');
```

При этом требуется, чтобы каталог `business` находился в текущем рабочем каталоге, что довольно скоро станет неудобно. Используя относительные пути для библиотечных включений, мы, вероятно, увидим запутанные операторы `require_once()`.

```
require_once('../../projectlib/business/User.php');
```

Конечно, мы можем использовать абсолютный путь.

```
require_once('/home/john/projectlib/business/User.php');
```

Ни одно из этих решений не будет идеальным. Чем детальнее мы определяем пути, тем сильнее мы привязываем библиотечный файл к конкретному системному каталогу.

При использовании абсолютного пути мы "привяжем" библиотеку к конкретной файловой системе. И каждый раз при установке проекта на новом сервере нужно будет менять все операторы `require` в соответствии с новыми путями к файлам.

При использовании относительного пути мы фиксируем связь между рабочим каталогом проекта и библиотекой. В результате при перемещении библиотек в файловой системе нужно будет отредактировать операторы `require()`. При использовании библиотеки в нескольких проектах нужно будет делать ее копии. В любом случае при использовании дополнительных каталогов теряется сама идея пакета. Мы уже не будем знать, с каким пакетом имеем дело. Это пакет `business` или `projectlib/business`?

Для того чтобы включаемые библиотеки хорошо работали в нашем коде, нужно отделить вызывающий код от библиотеки. Поэтому ссылка

```
business/User.php
```

должна работать из любой точки файловой системы. Это можно сделать, поместив пакет в один из каталогов, перечисленных в директиве `include_path`. Директива `include_path` обычно определяется в главном файле конфигурации PHP, `php.ini`. В нем определяется список каталогов, разделенных двоеточием (в UNIX-подобных системах) и точкой с запятой (в системе Windows).

```
include_path = ".:/usr/local/lib/php-libraries"
```

Если вы используете сервер Apache, то директиву `include_path` можете указать в глобальном файле конфигурации сервера (который обычно называется `httpd.conf`)

или в файле конфигурации для конкретного каталога (который обычно называется `.htaccess`), воспользовавшись следующим синтаксисом.

```
php_value include_path ".:/usr/local/lib/php-libraries"
```

На заметку. Файлы `.htaccess` обычно используются на веб-серверах ряда хостинг-компаний, которые сильно ограничивают права доступа пользователей к своим серверам.

При указании неабсолютного имени файла в функциях, обращающихся к файловой системе, таких как `fopen()` или `require()`, если файл не находится в иерархии текущего рабочего каталога, автоматически выполняется его поиск в каталогах, включенных в путь поиска начиная с первого в списке. В случае функции `fopen()` вы должны включить в список ее аргументов специальный флаг, чтобы активизировать эту возможность. Когда искомый файл найден, поиск заканчивается и файловая функция завершает свою работу.

Поэтому, помещая пакет во включаемый каталог, в операторах `require()` достаточно будет указать имя файла и название пакета.

Чтобы можно было поддерживать собственный библиотечный каталог, добавьте его имя в директиве `include_path`. Для этого отредактируйте файл `php.ini`. Если вы пользуетесь модулем PHP, не забудьте перезагрузить сервер Apache, чтобы изменения вступили в силу.

Если у вас нет прав доступа, необходимых для редактирования файла `php.ini`, можете задать пути включения файлов прямо из сценария с помощью функции `set_include_path()`. Этой функции передается список каталогов (в таком виде, как он выглядел бы в файле `php.ini`), который заменяет тот, который указан в директиве `include_path` для текущего процесса. В файле `php.ini` уже, вероятно, определено полезное значение для `include_path`. Поэтому, вместо того чтобы затирать его, вы можете получить его с помощью функции `get_include_path()` и добавить к нему собственный каталог. Вот как можно добавить каталог к текущему пути включения файлов.

```
set_include_path( get_include_path() . PATH_SEPARATOR . "/home/john/phplib/");
```

Вместо константы `PATH_SEPARATOR` будет подставлен символ-разделитель списка каталогов, принятый в текущей операционной системе. Напомним, что в Unix — это двоеточие (`:`), а в Windows — точка с запятой (`;`). Поэтому, если вас волнует переносимость кода, использование константы `PATH_SEPARATOR` является хорошим стилем программирования.

Автозагрузка

В некоторых ситуациях вам может понадобиться так организовать классы, чтобы определение каждого из них находилось в отдельном файле. У этого подхода есть свои недостатки, поскольку процесс включения файла влечет за собой дополнительные издержки. Однако подобный вид организации может быть очень полезным, особенно при расширении системы, когда новые классы должны включаться во время выполнения программы (подробнее об этой стратегии читайте в главах 11 и 12). В таких случаях можно связать имя класса с именем файла, содержащего определение этого класса. Например, мы можем определить класс `ShopProduct` в файле с именем `ShopProduct.php`. Более того, вы можете даже воспользоваться соглашением для поддержки пакетов, принятым в PEAR. В этом случае, если вам понадобится определить класс `ShopProduct` в пакете `business`, определение класса нужно поместить в файл `ShopProduct.php`, а сам этот файл — в каталог `business`. После этого вы долж-

ны присвоить имя классу с учетом имени пакета, чтобы избежать конфликта имен: `business_ShopProduct`. В случае использования пространств имен вы также можете воспользоваться подходом по организации классов в файловой системе, принятым в PEAR. Для этого нужно просто объявить пространство имен `business`, в котором содержится определение класса `ShopProduct`, файл которого `ShopProduct.php` должен находиться в папке `business`.

Чтобы автоматизировать процесс включения в программу файлов, содержащих определения классов, в PHP 5 предусмотрена возможность их автозагрузки. Ее стандартная реализация очень проста, но от этого она не становится менее полезной. Для этого нужно просто вызвать функцию `spl_autoload_register()` без параметров. После этого активизируется средство автозагрузки классов, которое автоматически вызывает функцию `spl_autoload()` при попытке создать в программе экземпляр неизвестного класса. Функции `spl_autoload()` в качестве параметра передается имя неизвестного класса, которое затем преобразуется в имя файла. Для этого имя класса преобразуется в нижний регистр и к нему по очереди добавляются все зарегистрированные стандартные расширения (сначала `.inc`, а затем `.php`). Далее функция пытается загрузить содержимое этого файла в программу, полагая, что в нем должно содержаться определение неизвестного класса. Пример приведен ниже.

```
spl_autoload_register();
$writer = new Writer();
```

Предположим, что в программу еще не включен файл с определением класса `Writer`, поэтому при попытке создания экземпляра этого класса возникнет исключительная ситуация. Однако, поскольку ранее я активизировал в программе автозагрузку классов, 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.

В стандартной реализации автозагрузки классов поддерживаются пространства имен. При этом имя пакета считается именем каталога. Поэтому при выполнении приведенного ниже кода интерпретатор PHP будет искать файл `writer.php` (обратите внимание: его имя задано в нижнем регистре) в каталоге `util`.

```
spl_autoload_register();
$writer = new util\Writer();
```

Но что произойдет в случае, если мне понадобится использовать чувствительные к регистру символов имена файлов классов (т.е. когда в именах файлов могут присутствовать как прописные, так и строчные буквы)? Так, если определение класса `Writer` будет помещено в файл `Writer.php`, то стандартная реализация автозагрузки классов работать уже не будет.

К счастью, я могу зарегистрировать собственную функцию автозагрузки классов, в которой реализовано другое соглашение об именовании файлов. Для этого нужно передать функции `spl_autoload_register()` в качестве параметра имя моей функции либо анонимную функцию. Моей функции автозагрузки должен передаваться один параметр. Тогда при попытке создания в программе экземпляра неизвестного класса интерпретатор PHP вызовет мою функцию автозагрузки, передав ей в качестве строкового параметра имя этого класса. При этом в функции автозагрузки мы должны сами реализовать стратегию поиска и включения в программу нужных файлов классов. После вызова пользовательской функции автозагрузки интерпретатор PHP снова попытается создать экземпляр нашего класса.

Ниже приведен пример простой пользовательской функции, выполняющей автозагрузку файлов.

```
function straightIncludeWithCase( $classname ) {
   $file = "{$classname}.php";
   if ( file_exists( $file ) ) {
     require_once( $file );
   }
}

spl_autoload_register( 'straightIncludeWithCase' );
$product = new ShopProduct( 'Черные врата', 'Гарри', 'Хантер', 12.99 );
```

После первоначальной неудачной попытки создать экземпляр класса `ShopProduct` интерпретатор PHP обнаружит, что я зарегистрировал собственную функцию автозагрузки, передав ее имя функции `spl_autoload_register()`. Тогда при вызове моей функции автозагрузки ей в качестве параметра передается строка `"ShopProduct"`, содержащая имя ненайденного класса. В нашей реализации мы просто пытаемся включить в программу файл `ShopProduct.php`. Конечно, это получится, только если файл находится в текущем рабочем каталоге или в одном из каталогов, включенных в путь поиска. В этой ситуации не существует простого способа работы с пакетами, кроме как воспользоваться устаревшим соглашением о присвоении имен, принятым в PEAR, либо использовать пространства имен. Хотя методику включения пакетов, принятую в PEAR, можно достаточно легко реализовать, как показано ниже.

```
function replaceUnderscores( $classname ) {
   $path = str_replace('_', DIRECTORY_SEPARATOR, $classname );
   if ( file_exists( "{$path}.php" ) ) {
     require_once( "{$path}.php" );
   }
}

spl_autoload_register( 'replaceUnderscores' );

$x = new ShopProduct();
$y = new business_ShopProduct();
```

Как видите, функция `replaceUnderscores()` преобразует символы подчеркивания, содержащиеся в переданном аргументе `$classname`, в символ разделения каталогов, заданный константой `DIRECTORY_SEPARATOR` (`'/'` для систем Unix и `'\'` для Windows). Затем мы пытаемся включить в программу файл класса `business/ShopProduct.php`. Если такой файл существует и класс, который в нем содержится, был назван корректно, то экземпляр объекта будет создан без ошибок. Разумеется, для этого необходимо, чтобы программист соблюдал соглашение об именовании, которое запрещает использование символа подчеркивания в именах классов, за исключением случаев, когда он используется для разделения пакетов.

А как быть в случае использования пространств имен? Выше мы уже упоминали о том, что в стандартной реализации автозагрузки классов пространства имен поддерживаются. Однако, если мы заменяем стандартную реализацию собственной, проблема поддержки пространств имен полностью перекладывается на нас. Для этого нам нужно просто проверить наличие в длинном имени класса символов обратной косой черты и заменить их символом-разделителем каталогов, принятым в используемой операционной системе, как показано ниже.

```
function myNamespaceAutoload( $path ) {
   if ( preg_match( '/\\\\/', $path ) ) {
```

```
        $path = str_replace('\\', DIRECTORY_SEPARATOR, $path );
    }

    if ( file_exists( "{$path}.php" ) ) {
        require_once( "{$path}.php" );
    }
}
```

Значение, которое передается пользовательской функции автозагрузки, всегда нормализовано и содержит полностью определенное имя класса без начального символа обратной косой черты. Поэтому в момент создания экземпляра класса вам не нужно заботиться о его псевдониме или относительной ссылке на пространство имен.

Но что произойдет, если в моей программе нужно использовать оба стиля автозагрузки файлов классов, принятых как в PEAR, так и в пространствах имен? Для этого можно просто объединить реализацию двух приведенных выше функций автозагрузки в одной функции либо воспользоваться стеком функций автозагрузки, который поддерживается в функции `spl_autoload_register()`, как показано ниже.

```
spl_autoload_register( 'replaceUnderscores' );
spl_autoload_register( 'myNamespaceAutoload' );

$x = new ShopProduct();
$y = new business_ShopProduct();
$z = new business\ShopProduct2();
$a = new \business\ShopProduct3();
```

Когда интерпретатор PHP обнаружит неизвестный класс, он сначала вызовет функцию `replaceUnderscores()`, а затем — `myNamespaceAutoload()`, если после вызова первой функции определение нужного класса все еще не будет найдено.

Очевидно, что при использовании описанного выше стека функций автозагрузки производительность приложения снижается из-за высоких накладных расходов. Так почему же в PHP он все-таки поддерживается? Все дело в том, что в реальных проектах довольно часто одновременно используются как пространства имен, так и пакеты PEAR, в которых имена классов разделяются символом подчеркивания. Кроме того, в некоторых компонентах больших систем, а также библиотеках независимых разработчиков, требуется реализовать собственный механизм автозагрузки классов. Поддержка стека функций автозагрузки как раз позволяет зарегистрировать независимые стратегии загрузки классов, которые не мешают друг другу. После того как работа с библиотекой закончена и отпала необходимость в автоматической загрузке ее классов, имя собственной функции автозагрузки можно передать функции `spl_autoload_unregister()` в качестве параметра, чтобы удалить ее из стека и таким образом "убрать все за собой".

На заметку. В PHP поддерживается также устаревшая и менее гибкая функция для автозагрузки файлов классов `__autoload()`. Если в вашем проекте реализована эта функция, интерпретатор PHP вызовет ее в случае, когда определение класса не будет найдено. Поскольку при использовании данной функции отключается стек автозагрузки, описанный выше, поддержка функции `__autoload()` в будущих версиях PHP не гарантируется. Чтобы выйти из ситуации, когда обнаружится, что функция `__autoload()` перестала вызываться интерпретатором PHP, или если нужно обеспечить работоспособность стека автозагрузки, просто вызовите функцию `spl_autoload_register()` и передайте ей в качестве параметра строку `'__autoload'`, как показано ниже.

```
spl_autoload_register('__autoload');
```

Функции для исследования классов и объектов

В PHP предусмотрен широкий набор функций для исследования классов и объектов. Для чего это нужно? Вы ведь наверняка сами создали большинство классов, которые используются в программе.

На самом деле во время выполнения программы не всегда известно, какие именно классы используются. Например, вы могли создать систему для "прозрачной" работы со сложными классами, разработанными сторонними производителями. В подобном случае экземпляр объекта обычно создается только на основании имени класса. В PHP разрешается использовать строки, чтобы ссылаться на классы динамически, как показано ниже.

```
// Task.php

namespace tasks;

class Task {
   function doSpeak() {
      print "Привет!\n";
   }
}

// TaskRunner.php
$classname = "Task";

require_once( "tasks/{$classname}.php" );
$classname = "tasks\\$classname";

$myObj = new $classname();
$myObj->doSpeak();
```

Строку, которую мы присвоили переменной `$classname`, можно получить из файла конфигурации или путем отображения данных, полученных в веб-запросе, на содержимое каталога. Затем можно использовать эту строку для загрузки файла класса и создания экземпляра объекта. Обратите внимание на то, что в приведенном выше фрагменте кода я указал перед именем класса пространство имен.

На заметку. Даже предприняв все меры предосторожности, вы должны быть предельно внимательны при динамической установке дополнительных модулей кода, созданных сторонними разработчиками. Нельзя автоматически запускать код, полученный от неизвестных пользователей. Дело в том, что любой установленный вами дополнительный модуль, как правило, запускается с теми же правами доступа, что и основной код вашего приложения. В результате злоумышленник, написавший такой модуль, может полностью вывести из строя ваше приложение и даже сервер, на котором оно выполняется.

Однако сказанное выше вовсе не означает, что стоит отказываться от использования дополнений. Сам факт, что независимые разработчики смогут расширять функциональные возможности вашего приложения, позволяет достичь большей гибкости. Для повышения уровня безопасности вам следует создать специальный каталог, в котором будут размещаться дополнительные модули. Доступ к этому каталогу нужно предоставить только системному администратору, который и должен заниматься установкой дополнительных модулей, либо напрямую, либо через специальное приложение, защищенное паролем. При этом перед установкой администратор должен лично проконтролировать исходный код дополнения либо получить его из источника, заслуживающего доверия. Именно так реализована работа с дополнительными модулями в популярной платформе для разработки веб-сайтов и блогов WordPress.

Обычно подобные операции выполняют, когда нужно, чтобы система могла запускать созданные пользователем подключаемые дополнительные модули (plug-ins).

Но прежде чем делать такие рискованные вещи в реальном проекте, вы должны убедиться, что указанный класс существует, у него есть нужные вам методы и т.д.

В PHP 5 часть функций для работы с классами была заменена более мощным Reflection API, который мы будем изучать далее в этой главе. Однако простота и удобство использования в некоторых случаях делают эти функции более предпочтительными. По этой причине мы сейчас и приступим к их рассмотрению.

Поиск классов

Функции `class_exists()` передается строка, содержащая имя класса. Она возвращает значение `true`, если класс существует, и `false` — в противном случае.

С помощью этой функции можно сделать предыдущий фрагмент кода более безопасным.

```
// TaskRunner.php
$classname = "Task";

$path = "tasks/{$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();
```

Конечно, мы не можем быть уверенными, что для конструктора рассматриваемого класса не потребуются аргументы. Для достижения такого уровня безопасности программирования необходимо обратиться к Reflection API, который мы будем изучать далее в этой главе. Тем не менее перед его использованием с помощью функции `class_exists()` мы должны убедиться в том, что нужные нам классы существуют.

На заметку. Не забывайте, что всегда следует с осторожностью относиться к данным, полученным из внешних источников. Об этом мы говорили выше. Перед использованием всегда проверяйте полученные данные и выполняйте их предварительную обработку. В случае получения от пользователя пути и имени файла следует экранировать или удалить в нем точки и разделители каталогов. Тем самым предотвращается возможность взлома сайта, когда недобросовестный пользователь может заменить имена каталогов и включить в них нежелательные файлы. Тем не менее при описании легко расширяемой с помощью дополнительных модулей системы я имел в виду, что установкой дополнительных модулей должен заниматься специальный человек, которому предоставлены соответствующие права, но никак не посторонний пользователь.

Чтобы получить массив всех классов, определенных в сценарии, можно воспользоваться функцией `get_declared_classes()`.

```
print_r( get_declared_classes() );
```

В результате будет выведен список встроенных и определенных пользователем классов. Помните, что данная функция возвращает только те классы, которые были объявлены к моменту ее вызова. Впоследствии вы можете использовать директиву `require()` или `require_once()` и тем самым добавить классы к сценарию.

Получение информации об объекте или классе

Как вы знаете, с помощью уточнений типов классов можно ограничить тип аргументов метода некоторого объекта. Но даже используя эту возможность, не всегда можно быть уверенным в отношении типа объекта. Например, на момент написания данной книги в PHP не разрешалось ограничивать тип класса, возвращаемого из метода или функции, хотя такая возможность должна быть реализована в следующих выпусках PHP.

Существует ряд основных средств для проверки типа объекта. Прежде всего, мы можем узнать класс объекта с помощью функции `get_class()`. В качестве аргумента ей передается объект любого типа, а она возвращает в виде строки его имя класса.

```
$product = getProduct();
if ( get_class( $product ) == 'CDProduct' ) {
   print "\$product -- объект класса CDProduct\n";
}
```

В данном фрагменте кода мы получаем *что-то* от функции `getProduct()`. Чтобы быть абсолютно уверенными, что это объект типа `CDProduct`, мы используем функцию `get_class()`.

На заметку. Классы `CdProduct` и `BookProduct` были описаны в главе 3.

Вот определение функции `getProduct()`.

```
function getProduct() {
   return new CDProduct( "Пропавший без вести",
                         "Группа", "ДДТ",
                        10.99, 60.33 );
}
```

Функция `getProduct()` просто создает экземпляр объекта `CDProduct` и возвращает его. Мы воспользуемся данной функцией в этом разделе.

Функция `get_class()` выдает очень специфическую информацию. Обычно же нужна более общая информация о принадлежности к семейству классов. Предположим, нам нужно знать, что объект принадлежит семейству `ShopProduct`, но при этом не имеет значения, к какому классу конкретно: `BookProduct` или `CDProduct`. Для этой цели в PHP предусмотрен оператор `instanceof`.

На заметку. В PHP 4 оператор `instanceof` не поддерживается. Вместо этого в PHP 4 была предусмотрена функция `is_a()`, которую в PHP 5.0 не рекомендовалось использовать. Начиная с PHP 5.3 эту функцию снова можно использовать.

Оператор `instanceof` работает с двумя операндами: объектом, который нужно протестировать (указывается слева от ключевого слова `instanceof`), и именем класса или интерфейса справа. Оператор возвращает значение `true`, если объект является экземпляром класса указанного типа.

```
$product = getProduct();
if ( $product instanceof ShopProduct ) {
   print "\$product -- объект типа ShopProduct\n";
}
```

Получение полностью определенной строковой ссылки на класс

Благодаря введению пространств имен удалось избежать множества проблем, которые возникали при использовании объектно-ориентированных средств языка PHP. Теперь мы навсегда избавились от необходимости создавать длинные и уродливые имена классов, а также от опасности возникновения конфликта имен (устаревший код не в счет!). С другой стороны, при использовании псевдонимов классов или относительных ссылок на пространства имен иногда довольно трудно выяснить, являются ли пути к некоторым классам полностью определенными.

Ниже приведено несколько примеров, когда сложно определить имя класса.

```
namespace mypackage;

use util as u;
use util\db\Querier as q;
class Local {}

// Определите следующее:

// Чему соответствует псевдоним пространства имен
// u\Writer;

// Чему соответствует псевдоним класса
// q;

// Чему соответствует ссылка на класс в локальном контексте
// Local
```

На самом деле не так и сложно определить, во что превращаются приведенные выше ссылки на класс. Гораздо труднее написать код, в котором бы обрабатывались все возможные случаи. В качестве примера рассмотрим ссылку `u\Writer`. При этом в программе автоматического распознавания нужно определить, что `u` — это псевдоним `util` и что `util` не является пространством имен, заслуживающим отдельного исследования. К счастью, в PHP 5.5 введена конструкция *ИмяКласса*`::class`. Другими словами, к существующему имени класса вы можете добавить оператор определения зоны видимости и с помощью ключевого слова `class` найти полностью определенное имя этого класса, как показано в приведенных ниже примерах.

```
print u\Writer::class."\n";
print q::class."\n";
print Local::class."\n";
```

В результате будет выведено следующее.

```
util\Writer
util\db\Querier
mypackage\Local
```

Получение информации о методах

Чтобы получить список всех методов класса, можно воспользоваться функцией `get_class_methods()`. В качестве аргумента ей передается имя класса, а она возвращает массив, содержащий имена всех методов класса.

```
print_r( get_class_methods( 'CDProduct' ) );
```

Предполагая, что класс `CDProduct` существует, получим следующий результат.

```
Array
(
    [0] => __construct
    [1] => getPlayLength
    [2] => getSummaryLine
    [3] => getProducerFirstName
    [4] => getProducerMainName
    [5] => setDiscount
    [6] => getDiscount
    [7] => getTitle
    [8] => getPrice
    [9] => getProducer
)
```

В этом примере мы передаем имя класса функции `get_class_methods()` и выводим возвращенный ею массив с помощью функции `print_r()`. Мы могли бы также передать функции `get_class_methods()` *объект* и получить такой же результат.

Если вы используете не самую старую версию PHP, то в возвращенный список будут включены только имена общедоступных методов.

Как мы видели, можно сохранить имя метода в строковой переменной и вызвать его динамически вместе с объектом следующим образом.

```
$product = getProduct();    // Получим объект
$method = "getTitle";       // Определим имя вызываемого метода
print $product->$method(); // Вызовем метод
```

Конечно, такая методика таит в себе опасность. Что произойдет, если метода не существует? Как и следовало ожидать, сценарий завершится ошибкой. У нас уже есть опыт проверки того, существует ли метод.

```
if ( in_array( $method, get_class_methods( $product ) ) ) {
   print $product->$method(); // Вызовем метод
}
```

Прежде чем вызывать метод, мы проверяем, присутствует ли его имя в массиве, возвращенном функцией `get_class_methods()`. Однако в PHP для этой цели предусмотрены более специализированные инструменты. Имена методов в той или иной степени можно проверить с помощью двух функций: `is_callable()` и `method_exists()`. Из этих двух функций `is_callable()` — более сложная. В качестве первого аргумента ей передается строковая переменная, определяющая имя функции. Если заданная функция существует и ее можно вызвать, функция `is_callable()` возвращает значение `true`. Чтобы применить такую же проверку к методу, вместо имени функции нужно передать ей массив. Этот массив должен содержать ссылку на объект или имя класса в качестве первого элемента и имя метода для проверки — в качестве второго. Функция вернет значение `true`, если указанный метод существует в классе.

```
if ( is_callable( array( $product, $method) ) ) {
   print $product->$method(); // Вызовем метод
}
```

У функции `is_callable()` есть и второй необязательный аргумент — булево значение. Если вы установите для него значение `true`, то функция проверит, существует ли заданное имя функции или метода, но не будет проверять возможность его вызова.

Функции method_exists() передаются ссылка на объект (или имя класса) и имя метода, а она возвращает значение true, если заданный метод существует в классе объекта.

```
if ( method_exists( $product, $method ) ) {
   print $product->$method(); // Вызовем метод
}
```

Внимание!

То, что метод существует, еще не означает, что его можно вызвать. Функция method_exists() возвращает значение true для закрытых (private), защищенных (protected) и общедоступных (public) методов.

Получение информации о свойствах

Точно так же, как можно запросить список методов класса, можно запросить и список его свойств. Функции get_class_vars() передается имя класса, а она возвращает ассоциативный массив. Имена свойств сохраняются в виде ключей этого массива, а значения полей — в виде значений. Давайте выполним проверку объекта CDProduct. Для наглядности добавим к классу общедоступное свойство: CDProduct::$coverUrl. В результате вызова

```
print_r( get_class_vars( 'CDProduct' ) );
```

будет показано только общедоступное свойство.

```
Array
(
    [coverUrl] =>
)
```

Получение информации о наследовании

С помощью функций для работы с классами можно также выявлять отношения наследования. Например, с помощью функции get_parent_class() можно узнать 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 нет, т.е. если у проверяемого класса нет родительского класса, то функция возвращает значение false. В результате вызова

```
print get_parent_class( 'CDProduct' );
```

мы получим имя родительского класса ShopProduct, как и можно было 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. Функция возвращает значение true, если второй класс является суперклассом первого аргумента.

```
$product = getProduct(); // Получим объект
if ( is_subclass_of( $product, 'ShopProduct' ) ) {
   print "CDProduct является подклассом класса ShopProduct\n";
}
```

Функция is_subclass_of() сообщит информацию только об отношениях наследования в классе, но она не поможет вам узнать, реализует ли класс интерфейс. Для

этой цели следует использовать оператор `instanceof`. Кроме того, можно воспользоваться функцией `class_implements()`, которая является частью SPL (Standard PHP Library — стандартная библиотека PHP). Этой функции передается имя класса или ссылка на объект, а она возвращает массив имен интерфейсов.

```
if ( in_array( 'someInterface', class_implements( $product )) ) {
   print "CdProduct реализует интерфейс someInterface\n";
}
```

Вызов метода

Мы уже рассматривали пример, в котором использовалась строковая переменная для динамического вызова метода.

```
$product = getProduct();     // Получим объект
$method = "getTitle";        // Определим имя метода
print $product->$method();   // Вызовем метод
```

Для этой цели в PHP предусмотрена также функция `call_user_func()`. Она может вызывать либо методы, либо функции. Чтобы вызвать функцию, в качестве первого аргумента нужно 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()`.

```
$product = getProduct(); // Получим объект
call_user_func( array( $product, 'setDiscount' ), 20 );
```

Этот динамический вызов эквивалентен следующему.

```
$product->setDiscount( 20 );
```

Поскольку мы можем в равной степени использовать строковую переменную вместо имени метода

```
$method = "setDiscount";
$product->$method(20);
```

пользы от функции `call_user_func()` мало. Намного большее впечатление производит родственная ей функция `call_user_func_array()`. Она работает аналогично `call_user_func()`, пока дело касается выбора нужного метода либо функции. Но чрезвычайно важно то, что требуемые для вызова метода аргументы передаются функции `call_user_func_array()` в виде массива.

Почему это так полезно? Время от времени вы будете получать аргументы в виде массива. Если вы не будете знать заранее количество аргументов, с которыми придется иметь дело, то передать их функции будет очень трудно. В главе 4, "Расширенные средства", мы рассматривали методы-перехватчики, которые можно использовать для создания делегирующих классов. Вот простой пример метода `__call()`.

```
function __call( $method, $args ) {
   if ( method_exists( $this->thirdpartyShop, $method ) ) {
      return $this->thirdpartyShop->$method();
   }
}
```

Как вы уже знаете, метод `__call()` вызывается, когда из клиентского кода пытаются вызвать неопределенный метод. В данном примере мы используем объект, сохраненный в свойстве `$thirdpartyShop`. Если в данном объекте существует метод, имя которого указано в первом аргументе `$method`, то он вызывается. Здесь мы легкомысленно предположили, что метод, который нужно вызвать, не требует никаких аргументов. Это и является источником всех проблем! При создании метода `__call()` заранее не существует способа выяснить, насколько большим может быть массив `$args` при разных вызовах этого метода. Если мы передадим массив `$args` непосредственно делегирующему методу, то передадим единственный аргумент-массив, а не отдельные аргументы, как того, возможно, ожидает делегирующий метод. Поэтому функция `call_user_func_array()` решает эту проблему идеальным образом.

```
function __call( $method, $args ) {
   if ( method_exists( $this->thirdpartyShop, $method ) ) {
      return call_user_func_array(
                array( $this->thirdpartyShop, $method ), $args );
   }
}
```

Интерфейс Reflection API

Программный интерфейс Reflection API для PHP версии 5 — то же самое, что и пакет `java.lang.reflect` для Java. Он состоит из встроенных классов для анализа свойств, методов и классов. В некоторых отношениях он напоминает рассмотренные выше функции для работы с объектами, такие как `get_class_vars()`, но является более гибким и позволяет учитывать больше нюансов. Он также предназначен для более эффективной работы с объектно-ориентированными средствами PHP, такими как управление доступом, интерфейсы и абстрактные классы. Старые, более ограниченные, функции классов так работать не могут.

Основные сведения

Интерфейс Reflection API можно использовать для исследования не только классов. Например, класс `ReflectionFunction` предоставляет информацию о заданной функции, а `ReflectionExtension` — информацию о скомпилированных расширениях языка PHP. В табл. 5.1 перечислены некоторые классы интерфейса Reflection API.

Некоторые классы из Reflection API позволяют во время выполнения программы получить просто беспрецедентную информацию об объектах, функциях и расширениях языка, содержащихся в сценарии, которую раньше невозможно было добыть.

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. И вскоре вы поймете, что это незаменимый инструмент для исследования классов. Например, с его помощью можно создавать диаграммы классов или документацию, сохранять информацию об объекте в базе данных, исследовать методы доступа объекта (установщики или получатели), которые используются для извлечения значений полей. Создание каркаса, который вызывает методы в классах модуля в соответствии с принятой схемой именования, — это еще один пример применения интерфейса Reflection.

Таблица 5.1. Некоторые классы интерфейса Reflection API

Класс	Описание
`Reflection`	Содержит статический метод `export()`, предоставляющий итоговую информацию о классе
`ReflectionClass`	Позволяет получить информацию о классе и содержит средства для работы с ним
`ReflectionMethod`	Позволяет получить информацию о методах класса и содержит средства для работы с ними
`ReflectionParameter`	Позволяет получить информацию об аргументах метода
`ReflectionProperty`	Позволяет получить информацию о свойствах класса
`ReflectionFunction`	Позволяет получить информацию о функциях и содержит средства для работы с ними
`ReflectionExtension`	Позволяет получить информацию о расширениях PHP
`ReflectionException`	Предназначен для обработки ошибок
`ReflectionZendExtension`	Информация о расширениях PHP Zend

Время закатать рукава

Мы уже встречались с некоторыми функциями для изучения атрибутов классов. Они полезны, но обычно довольно ограничены. А теперь давайте рассмотрим инструмент, который *готов* к выполнению данной работы. Класс `ReflectionClass` предоставляет методы, которые собирают информацию обо всех аспектах заданного класса, независимо от того, внутренний это класс или определенный пользователем. Конструктору класса `ReflectionClass` в качестве единственного аргумента передается имя класса, как показано ниже.

```
$prod_class = new ReflectionClass( 'CDProduct' );
Reflection::export( $prod_class );
```

Создав объект типа `ReflectionClass`, вы можете использовать вспомогательный класс `Reflection` для создания итоговой информации о классе `CDProduct`. У класса `Reflection` есть статический метод `export()`, который форматирует и создает дамп данных, содержащихся в объекте типа `Reflection` (точнее, в любом экземпляре класса, который реализует интерфейс `Reflector`). Вот фрагмент вывода, полученного в результате вызова метода `Reflection::export()`.

```
Class [ <user> class CdProduct extends ShopProduct ] {
 @@ fullshop.php 53-73

- Constants [0] {
}

- Static properties [0] {
}

- Static methods [0] {
}
- Properties [2] {
Property [ <default> private $playLength ]
Property [ <default> protected $price ]
}
- Methods [10] {
Method [ <user, overwrites ShopProduct, ctor> public method __construct ] {
```

```
@@ fullshop.php 56 - 61
- Parameters [5] {
Parameter #0 [ <required> $title ]
Parameter #1 [ <required> $firstName ]
Parameter #2 [ <required> $mainName ]
Parameter #3 [ <required> $price ]
Parameter #4 [ <required> $playLength ]
 }
}
Method [ <user> public method getPlayLength ] {
@@ fullshop.php 63 - 65
}
Method [ <user, overwrites ShopProduct, prototype ShopProduct> public method
getSummaryLine ] {
@@ fullshop.php 67 - 71
   }
 }
}
```

Как видите, метод `Reflection::export()` предоставляет отличный доступ к информации о классе. Этот метод дает обобщенную информацию почти обо всех аспектах класса `CDProduct`, включая состояние доступа к свойствам и методам, аргументы, необходимые каждому методу, и расположение каждого метода внутри файла сценария. Сравните это с традиционной функцией отладки. Функция `var_dump()` — это инструмент общего назначения для выдачи итоговой информации о данных. Прежде чем получить итоговую информацию, вы должны создать экземпляр объекта, но даже после этого функция `var_dump()` не даст настолько подробной информации, как метод `Reflection::export()`.

```
$cd = new CDProduct( "Пропавший без вести",
                            "Группа", "ДДТ",
                           10.99, null, 60.33 );
var_dump( $cd );
```

На выходе получим следующее.

```
object(CDProduct)#1 (8) {
  ["playLength:private"]=>
  float(60.33)
  ["coverUrl"]=>
  NULL
  ["title:private"]=>
  string(19) "Пропавший без вести"
  ["producerMainName:private"]=>
  string(3) "ДДТ"
  ["producerFirstName:private"]=>
  string(6) "Группа"
  ["price:protected"]=>
  float(10.99)
  ["discount:private"]=>
  int(0)
  ["id:private"]=>
  int(0)
}
```

Функция `var_dump()` и ее "сестра" функция `print_r()` — невероятно удобные инструменты для отображения данных в сценариях. Но для классов и функций интерфейс Reflection API позволяет выйти на совершенно новый уровень отладки.

Исследование класса

Метод `Reflection::export()` предоставляет много полезной информации для отладки, но интерфейс Reflection API можно использовать особым образом. Давайте будем работать непосредственно с классами `Reflection`.

Вы уже знаете, как создать экземпляр объекта `ReflectionClass`.

```
$prod_class = new ReflectionClass( 'CDProduct' );
```

А теперь давайте используем объект `ReflectionClass`, чтобы исследовать класс `CDProduct` в процессе выполнения сценария. Мы хотим узнать, к какому типу класса он относится и можно ли создать его экземпляр. Вот функция, которая поможет ответить на эти вопросы.

```
function classData( ReflectionClass $class ) {
   $details = "";
   $name = $class->getName();
   if ( $class->isUserDefined() ) {
      $details .= "$name -- класс определен пользователем\n";
   }
   if ( $class->isInternal() ) {
      $details .= "$name -- встроенный класс\n";
   }
   if ( $class->isInterface() ) {
      $details .= "$name -- это интерфейс\n";
   }
   if ( $class->isAbstract() ) {
      $details .= "$name -- это абстрактный класс\n";
   }
   if ( $class->isFinal() ) {
      $details .= "$name -- это завершенный класс\n";
   }
   if ( $class->isInstantiable() ) {
      $details .= "$name -- можно создать экземпляр класса\n";
   } else {
      $details .= "$name -- нельзя создать экземпляр класса\n";
   }
   if ( $class->isCloneable() ) {

      $details .= "$name -- нельзя клонировать\n";

   } else {

      $details .= "$name -- нельзя клонировать \n";

}
   return $details;
}
$prod_class = new ReflectionClass( 'CDProduct' );
print classData( $prod_class );
```

Мы создаем объект типа `ReflectionClass` и присваиваем его переменной `$prod_class`. При этом конструктору класса `ReflectionClass` передается строка, содержащая имя исследуемого класса `'CDProduct'`. Переменная `$prod_class` затем передается функции `classData()`, демонстрирующей ряд методов, которые можно использовать для запросов к классу.

Эти методы, вероятно, не требуют объяснений, но мы все-таки приведем их краткие описания.

- Метод `ReflectionClass::getName()` возвращает имя исследуемого класса.
- Метод `ReflectionClass::isUserDefined()` возвращает истинное значение, если класс был объявлен в PHP-коде.
- Метод `ReflectionClass::isInternal()` возвращает истинное значение, если класс является встроенным.
- С помощью метода `ReflectionClass::isAbstract()` можно проверить, является ли класс абстрактным.
- Метод `ReflectionClass::isInterface()` позволяет узнать, является ли класс интерфейсом.
- Если вы хотите создать экземпляр класса, то с помощью метода `ReflectionClass::isInstantiable()` можно проверить, осуществимо ли это.

Можно исследовать даже исходный код класса, определенного пользователем. Объект `ReflectionClass` позволяет получить имя файла класса и номера первой и последней строк в этом файле, в которых этот класс определяется.

Вот простой метод, в котором объект типа `ReflectionClass` используется для доступа к исходному коду класса.

```
class ReflectionUtil {
   static function getClassSource( ReflectionClass $class ) {
      $path  = $class->getFileName();
      $lines = @file( $path );
      $from  = $class->getStartLine();
      $to    = $class->getEndLine();
      $len   = $to-$from+1;
      return implode( array_slice( $lines, $from-1, $len ));
   }
}
print ReflectionUtil::getClassSource(
         new ReflectionClass( 'CDProduct' ) );
```

Как видите, класс `ReflectionUtil` очень прост и содержит единственный статический метод `ReflectionUtil::getClassSource()`. Этому методу в качестве единственного аргумента передается объект типа `ReflectionClass`, а он возвращает исходный код класса. Метод `ReflectionClass::getFileName()` возвращает абсолютное имя файла класса, поэтому код должен работать без проблем и открыть данный исходный файл. Функция `file()` возвращает массив всех строк в файле. Метод `ReflectionClass::getStartLine()` возвращает номер начальной строки в исходном файле, а метод `ReflectionClass::getEndLine()` — номер последней строки, в которых содержится определение класса. И теперь для получения нужных строк достаточно воспользоваться функцией `array_slice()`.

Для компактности в этом коде опущена обработка ошибок. Но в реальном приложении нужно будет проверять аргументы и коды возврата.

Исследование методов

Так же как `ReflectionClass` используется для исследования класса, объект типа `ReflectionMethod` применяется для исследования метода.

Ссылку на объект `ReflectionMethod` можно получить двумя способами. Во-первых, можно получить массив объектов типа `ReflectionMethod`, вызвав метод `ReflectionClass::getMethods()`. Во-вторых, если вас интересует конкретный метод, передайте его имя методу `ReflectionClass::getMethod()`, который и вернет соответствующий объект типа `ReflectionMethod`.

Ниже мы воспользуемся методом `ReflectionClass::getMethods()`, чтобы проверить возможности класса `ReflectionMethod`.

```
$prod_class = new ReflectionClass( 'CDProduct' );
$methods = $prod_class->getMethods();

foreach ( $methods as $method ) {
   print methodData( $method );
   print "\n----\n";
}

function methodData( ReflectionMethod $method ) {
   $details = "";
   $name = $method->getName();
   if ( $method->isUserDefined() ) {
      $details .= "$name -- метод определен пользователем\n";
   }
   if ( $method->isInternal() ) {
      $details .= "$name -- внутренний метод\n";
   }
   if ( $method->isAbstract() ) {
      $details .= "$name -- абстрактный метод\n";
   }
   if ( $method->isPublic() ) {
      $details .= "$name -- общедоступный метод\n";
   }
   if ( $method->isProtected() ) {
      $details .= "$name -- защищенный метод\n";
  }
   if ( $method->isPrivate() ) {
      $details .= "$name -- закрытый метод\n";
   }
   if ( $method->isStatic() ) {
      $details .= "$name -- статический метод\n";
   }
   if ( $method->isFinal() ) {
      $details .= "$name -- завершенный метод\n";
   }
   if ( $method->isConstructor() ) {
      $details .= "$name -- метод конструктора\n";
   }
   if ( $method->returnsReference() ) {
      $details .= "$name -- метод возвращает ссылку, а не значение\n";
   }
```

```
    return $details;
}
```

В этом коде для получения массива объектов типа `ReflectionMethod` используется вызов метода `ReflectionClass::getMethods()`. Затем в цикле выполняется обращение к каждому элементу массива и вызывается функция `methodData()`, которой передается полученный объект типа `ReflectionMethod`.

Имена методов, вызываемых в функции `methodData()`, отражают их назначение: код проверяет, является ли метод определенным пользователем, встроенным, абстрактным, общедоступным, защищенным, статическим или завершенным. Можно также проверить, является ли метод конструктором для своего класса и что он возвращает — ссылку или значение.

Однако здесь нужно сделать одно замечание: метод `ReflectionMethod::returnsReference()` не возвращает истинное значение, даже если проверяемый метод возвращает объект целиком (а не ссылку на него), хотя в PHP 5 объекты передаются и присваиваются по ссылке. Этот метод возвращает истинное значение, только если исследуемый метод был явно объявлен таким, который возвращает ссылку (путем помещения символа амперсанда перед именем метода).

Как и можно было ожидать, доступ к исходному коду метода можно получить тем же способом, который использовался ранее с `ReflectionClass` для доступа к классу.

```
class ReflectionUtil {
    static function getMethodSource( ReflectionMethod $method ) {
        $path = $method->getFileName();
        $lines = @file( $path );
        $from = $method->getStartLine();
        $to = $method->getEndLine();
        $len = $to-$from+1;
        return implode( array_slice( $lines, $from-1, $len ));
    }
}
$class = new ReflectionClass( 'CDProduct' );
$method = $class->getMethod( 'getSummaryLine' );
print ReflectionUtil::getMethodSource( $method );
```

Поскольку в классе `ReflectionMethod` есть методы `getFileName()`, `getStartLine()` и `getEndLine()`, получить исходный код исследуемого метода очень просто.

Исследование аргументов методов

Теперь, когда стало возможным ограничивать типы аргументов с помощью сигнатур методов, чрезвычайно полезной кажется возможность исследования аргументов, объявленных в сигнатуре метода. В интерфейсе Reflection API именно для этой цели предусмотрен класс `ReflectionParameter`. Чтобы получить объект типа `ReflectionParameter`, нам понадобится помощь объекта `ReflectionMethod`. Метод `ReflectionMethod::getParameters()` возвращает массив объектов типа `ReflectionParameter`.

Имея объект `ReflectionParameter`, можно узнать следующее: имя аргумента; была ли переменная передана по ссылке (т.е. при объявлении метода перед его именем был помещен символ амперсанда); информацию о классе, который используется для уточнения типа аргумента; и будет ли метод по умолчанию назначать нулевое значение для аргумента.

Ниже приведен пример использования некоторых методов класса `ReflectionParameter`.

```
$prod_class = new ReflectionClass( 'CDProduct' );
$method = $prod_class->getMethod( "__construct" );
$params = $method->getParameters();

foreach ( $params as $param ) {
   print argData( $param );
}

function argData( ReflectionParameter $arg ) {
   $details = "";
   $declaringclass = $arg->getDeclaringClass();
   $name  = $arg->getName();
   $class = $arg->getClass();
   $position = $arg->getPosition();
   $details .= "\$$name находится в позиции $position\n";
   if ( ! empty( $class ) ) {
      $classname = $class->getName();
      $details .= "\$$name должен быть объектом типа $classname\n";
   }
   if ( $arg->isPassedByReference() ) {
      $details .= "\$$name передан по ссылке\n";
   }
   if ( $arg->isDefaultValueAvailable() ) {
      $def = $arg->getDefaultValue();
      $details .= "\$$name по умолчанию равно: $def\n";
   }
   return $details;
}
```

Метод `ReflectionClass::getMethod()` возвращает объект типа `ReflectionMethod` для интересующего нас метода (в рассмотренном примере это конструктор). Затем вызывается метод `ReflectionClass::getParameters()` для получения массива объектов типа `ReflectionParameter`, соответствующих данному методу. В цикле функции `argData()` передается объект типа `ReflectionParameter`, а она возвращает информацию об аргументе.

Сначала мы узнаем имя переменной-аргумента с помощью метода `ReflectionParameter::getName()`. Метод `ReflectionParameter::getClass()` возвращает объект типа `ReflectionClass`, если в сигнатуре метода было использовано уточнение. Затем с помощью метода `isPassedByReference()` проверяется, является ли аргумент ссылкой. И наконец с помощью метода `isDefaultValueAvailable()` проверяется, было ли аргументу присвоено значение по умолчанию.

Использование интерфейса Reflection API

Зная основы интерфейса Reflection API, теперь вы сможете использовать его для работы.

Предположим, вы создаете класс, который динамически вызывает объекты типа `Module`. Это означает, что ваш класс должен уметь работать с дополнительными модулями, написанными сторонними разработчиками, которые можно автоматически вставлять в приложение, не занимаясь трудоемким кодированием. Чтобы достичь этого, можно определить метод `execute()` в интерфейсе `Module` или абстрактном базовом классе, заставляя все дочерние классы определять его реализацию. Вы можете разрешить пользователям своей системы перечислять классы типа `Module` во внешнем XML-файле конфигурации. Ваша система может использовать данную

информацию для группирования ряда объектов типа Module, перед тем как вызывать метод execute() для каждого из них.

Но что будет, если каждому объекту типа Module для выполнения работы потребуются *свои* (отличные от других!) данные? В этом случае в XML-файле конфигурации могут находиться ключи и значения свойств для каждого объекта типа Module, а создатель объекта типа Module должен предоставить методы-установщики для каждого имени свойства. А на этой основе в своем коде вы уже сможете обеспечить, чтобы для нужного имени свойства вызывался нужный метод-установщик.

Ниже объявлен интерфейс Module и приведена пара реализующих его классов.

```
class Person {
   public $name;

   function __construct( $name ) {
      $this->name = $name;
   }
}

interface Module {
   function execute();
}

class FtpModule implements Module {
   function setHost( $host ) {
      print "FtpModule::setHost(): $host\n";
   }

   function setUser( $user ) {
      print "FtpModule::setUser(): $user\n";
   }

   function execute() {
   // Выполнение работы
   }
}

class PersonModule implements Module {
   function setPerson( Person $person ) {
      print "PersonModule::setPerson(): {$person->name}\n";
   }

   function execute() {
   // Выполнение работы
   }
}
```

В этом коде в классах PersonModule и FtpModule предусмотрены пустые реализации метода execute(). В каждом классе также реализованы методы-установщики, которые ничего не делают, но сообщают, что они были вызваны. В нашей системе принято соглашение, что всем методам-установщикам должен передаваться только один аргумент: либо строка, либо объект, экземпляр которого можно создать с помощью одного строкового аргумента. Методу PersonModule::setPerson() должен передаваться объект типа Person, поэтому мы включили в наш пример простенький класс Person.

Следующим шагом для работы с классами PersonModule и FtpModule является создание класса ModuleRunner. В нем используется многомерный массив, проиндексированный по имени модуля, в котором содержится информация о параметрах конфигурации, полученная из XML-файла. Вот как выглядит этот код.

```
class ModuleRunner {
   private $configData = array(
            "PersonModule" => array( 'person' => 'bob' ),
               "FtpModule" => array(   'host' => 'example.com',
                                       'user' => 'anon' )
                     );
   private $modules = array();
// ...
}
```

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() класса ModuleRunner отвечает за создание корректных объектов типа Module, как показано в следующем фрагменте кода.

```
class ModuleRunner {
   // ...

   function init() {
      $interface = new ReflectionClass('Module');
      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 описан в следующем листинге!
         }
         array_push( $this->modules, $module );
      }
   }
//...
}
$test = new ModuleRunner();
$test->init();
```

Метод init() проходит в цикле по массиву ModuleRunner::$configData и для каждого указанного в нем имени класса типа Module пытается создать объект типа ReflectionClass. Когда конструктору класса ReflectionClass передается имя несуществующего класса типа Module, генерируется исключение. Поэтому в реальное приложение мы должны включить также код обработки ошибок. Для проверки того, что класс модуля принадлежит типу Module, используется вызов метода ReflectionClass::isSubclassOf().

Прежде чем вызвать метод execute() для каждого объекта типа Module, необходимо создать экземпляр этого объекта. Этим занимается метод method::ReflectionClass::newInstance(). Этому методу можно передать произвольное количество аргументов, которые будут переданы конструктору соответствующего класса. Если

все прошло успешно, то метод возвращает экземпляр нашего класса. В реальном приложении обязательно убедитесь, что методу конструктора каждого объекта типа `Module` не нужно передавать никаких аргументов, которые используются для создания экземпляра объекта.

Метод `ReflectionClass::getMethods()` возвращает массив всех объектов типа `ReflectionMethod`, имеющихся для текущего класса модуля. Для каждого элемента в массиве код вызывает метод `ModuleRunner::handleMethod()`, которому передаются экземпляр объекта типа `Module`, объект типа `ReflectionMethod` и массив свойств, ассоциированных с текущим объектом типа `Module`, полученный из файла конфигурации. Переданные данные проверяются в методе `handleMethod()`, после чего вызываются методы-установщики текущего объекта типа `Module`.

```
class ModuleRunner {
   // ...
   function handleMethod( Module $module, ReflectionMethod $method, $params ) {
      $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;
      }

      $arg_class = $args[0]->getClass();
      if ( empty( $arg_class ) ) {
         $method->invoke( $module, $params[$property] );
      } else {
         $method->invoke( $module,
                  $arg_class->newInstance( $params[$property] ) );
      }
   }
}
```

В методе `handleMethod()` сначала проверяется, является ли исследуемый метод допустимым установщиком. В коде допустимый метод-установщик должен называться `setXXXX()` и содержать только один аргумент.

Если предположить, что аргумент проверен, в коде затем извлекается имя свойства из имени метода-установщика. Для этого из начала имени метода удаляется слово `"set"` и полученная подстрока преобразуется в строчные символы. Эта строка используется для проверки элемента массива `$params`. Напомним, что в этом массиве содержатся предоставленные пользователем значения свойств, ассоциированных с объектом типа `Module`. Если в массиве `$params` нужное нам свойство не было обнаружено, то код возвращает значение `false`.

Если имя свойства, извлеченное из имени метода, содержащегося в объекте типа `Module`, соответствует элементу массива `$params`, то мы можем вызвать для него нужный корректный метод-установщик. Однако перед этим нам нужно проверить тип первого (и только первого!) аргумента метода-установщика. Эту информацию предоставляет метод `ReflectionParameter::getClass()`. Если этот метод возвраща-

ет пустое значение, то методу-установщику должно передаваться значение элементарного типа (т.е. строковое значение), в противном случае — объект.

Чтобы вызвать метод-установщик, воспользуемся новым методом интерфейса Reflection API `ReflectionMethod::invoke()`. Этому методу передаются объект (в нашем случае типа `Module`) и произвольное количество аргументов, которые будут переданы методу-установщику объекта типа `Module`. Метод `ReflectionMethod::invoke()` генерирует исключение, если переданный объект не соответствует методу, определенному в объекте типа `ReflectionMethod`. Метод `ReflectionMethod::invoke()` можно вызвать одним из двух способов. Если методу-установщику требуется передать значение элементарного типа, мы вызываем `ReflectionMethod::invoke()` и передаем ему строковое значение свойства, предоставленное пользователем. Если методу-установщику требуется объект, мы используем строковое значение свойства для создания экземпляра объекта нужного типа, который затем передается методу-установщику.

В данном примере предполагается, что для создания экземпляра требуемого объекта его конструктору нужно передать только один строковый аргумент. Но, конечно, лучше всего это проверить, прежде чем вызывать метод `ReflectionClass::newInstance()`.

После того как метод `ModuleRunner::init()` завершит свою работу, наш объект типа `ModuleRunner` будет содержать набор объектов типа `Module` (в массиве свойств `$modules`), причем все они будут наполнены данными. Теперь остается создать в классе `ModuleRunner` специальный метод, выполняющий перебор в цикле всех объектов типа `Module` и вызывающий для каждого из них метод `execute()`.

Резюме

В данной главе были рассмотрены некоторые способы и средства, предназначенные для управления библиотеками и классами. Я описал новую возможность PHP 5.3 — пространства имен. Вы убедились, что их успешно можно использовать для обеспечения гибкой организации классов и пакетов наряду с путями включения файлов, соглашением об именовании классов PEAR и каталогами файловой системы. Мы изучили функции для работы с объектами и классами в PHP, а затем перешли на более высокий уровень с помощью эффективного и многофункционального интерфейса Reflection API. И наконец мы использовали классы `Reflection`, чтобы создать простой пример, иллюстрирующий одно из возможных применений интерфейса Reflection.

Глава 6

Объекты и методология проектирования

Теперь, когда мы достаточно подробно рассмотрели механизм поддержки объектов в PHP, абстрагируемся от деталей и выясним, почему лучше всего использовать те средства, с которыми мы познакомились. В этой главе я остановлюсь на некоторых вопросах, касающихся объектов и методологии проектирования. Будет также рассмотрен UML — мощный графический язык описания объектно-ориентированных систем.

В этой главе рассмотрены такие вопросы.

- *Основы проектирования:* что я понимаю под проектированием и чем объектно-ориентированный проект отличается от процедурного кода.
- *Контекст класса:* как решить, что включить в класс.
- *Инкапсуляция:* сокрытие реализации и данных за интерфейсом класса.
- *Полиморфизм:* использование общего супертипа для того, чтобы разрешить прозрачную подстановку специализированных подтипов во время выполнения программы.
- *UML:* использование диаграмм для описания структуры объектно-ориентированной системы.

Определение программного проекта

Один из аспектов программного проекта касается определения системы: выяснение требований к системе, контекста и целей. Что система должна делать? Для кого она должна это делать? Какие выходные данные системы? Отвечают ли они поставленным требованиям? На нижнем уровне *проектирование* можно понимать как процесс, посредством которого вы определяете участников системы и устанавливаете связи между ними. Данная глава посвящена второму аспекту: определению и расположению классов и объектов.

Что такое участник? Объектно-ориентированная система состоит из классов. И очень важно решить, какой будет природа этих классов в вашей системе. Классы состоят отчасти из методов. Поэтому при определении классов вы должны решить, какие методы нужно объединить, чтобы они составляли одно целое. Но, как вы увидите, классы часто объединяются в отношения наследования, чтобы подчиняться

общим интерфейсам. Именно эти интерфейсы, или типы, должны быть первым пунктом в проектировании системы.

Существуют и другие отношения, которые можно определить для классов. Можно создавать классы, которые состоят из других типов или управляют списками экземпляров других типов. Вы можете спроектировать классы, в которых просто используются другие объекты. Возможность для таких отношений или использования встроена в классы (например, через использование уточнения типа класса в сигнатуре методов). Однако реальные связи объектов вступают в действие во время выполнения программы, что позволяет сделать процесс проектирования более гибким. Из данной главы вы узнаете, как моделировать эти отношения, а в остальной части книги мы будем изучать их более подробно.

В процессе проектирования вы должны решить, когда операция должна принадлежать одному типу и когда она должна принадлежать другому классу, используемому этим типом. Куда бы вы ни бросили взгляд, везде нужно делать выбор, принимать решения, которые либо приведут к ясности и изяществу, либо затянут в болото компромисса.

В этой главе мы займемся исследованием вопросов, которые могут повлиять на эти решения.

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 есть объекты". Но это неверно. В PHP часто бывает, что в процедурном коде используются объекты. С другой стороны, вы можете столкнуться с классами, в которых содержатся фрагменты процедурного кода. Наличие классов — еще не гарантия объектно-ориентированного проекта, даже в таком языке, как Java, который заставляет большую часть работы выполнять внутри класса.

Коренное различие между объектно-ориентированным и процедурным кодом заключается в том, как распределяется ответственность. 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 минимизировать эти зависимости путем передачи ответственности за управление задачами от клиентского кода объектам в системе.

В этом разделе я опишу простую задачу и проанализирую ее с точки зрения объектно-ориентированного и процедурного кодов, чтобы проиллюстрировать эти моменты. Итак, наша задача — построить простое средство для считывания информации из конфигурационных файлов и записи в них. Чтобы сосредоточить внимание на структуре кода, в этих примерах я опускаю код их реализации.

Начнем с процедурного подхода к этой проблеме. Для начала прочитаем и запишем текст в приведенном ниже формате.

ключ: *значение*

Для этой цели нам нужны только две функции.

```
function readParams( $sourceFile ) {
   $params = array();
   // Читаем текстовый параметр из $sourceFile
```

```
    return $params;
}

function writeParams( $params, $sourceFile ) {
    // Записываем текстовый параметр в $sourceFile
}
```

Функции `readParams()` нужно передать имя конфигурационного файла. Она пытается открыть его и читает каждую строку, выискивая в ней пары "ключ–значение". В ходе работы эта функция создает ассоциативный массив. И наконец она возвращает этот массив управляющему коду. Функции `writeParams()` передаются ассоциативный массив и имя конфигурационного файла. Она проходит в цикле по этому ассоциативному массиву, записывая каждую пару "ключ–значение" в файл. Вот пример клиентского кода, который работает с этими функциями.

```
$file = "./param.txt";
$array['key1'] = "val1";
$array['key2'] = "val2";
$array['key3'] = "val3";
writeParams( $array, $file );  // Запишем массив параметров в файл
$output = readParams( $file ); // Прочитаем массив параметров из файла
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
```

Но вдруг нас проинформировали, что программа должна уметь работать с простыми конфигурационными файлами, заданными в формате XML, как показано ниже.

```
<params>
    <param>
        <key>Ключ</key>
        <val>Значение</val>
    </param>
</params>
```

Если данный конфигурационный файл имеет расширение `.xml`, то для его обработки необходимо задействовать средства обработки XML, встроенные в PHP. И хотя такую задачу совсем нетрудно решить, существует угроза, что наш код станет намного труднее сопровождать. На этом этапе у нас есть два варианта. Мы можем проверить расширение файла в управляющем коде или сделать проверку внутри функций чтения и записи. Давайте пока остановимся на втором варианте.

```
function readParams( $source ) {
    $params = array();
    if ( preg_match( "/\.xml$/i", $source )) {
        // Читаем параметры в формате XML из $source
    } else {
        // Читаем текстовые параметры из $source
    }
    return $params;
```

```
}

function writeParams( $params, $source ) {
   if ( preg_match( "/\.xml$/i", $source )) {
      // Запишем параметры в формате XML в $source
   } else {
      // Запишем текстовые параметры в $source
  }
}
```

На заметку. Представленный в книге код всегда несет в себе элемент трудного компромисса. Он должен быть достаточно понятным, чтобы объяснять суть, но это часто означает, что нужно пожертвовать проверкой ошибок и пригодностью кода для той цели, ради которой он был написан. Другими словами, приведенный здесь пример предназначен для иллюстрации вопросов проектирования и дублирования, а не лучшего способа синтаксического анализа и записи данных в файл. По этой причине я буду опускать реализацию там, где это не имеет отношения к рассматриваемому вопросу.

Как видите, расширение `.xml` мы проверяли в каждой функции. Именно это повторение в итоге может стать причиной проблем. Если нас попросят включить в систему поддержку еще одного формата параметров, то мы должны будем помнить о том, что изменения нужно внести в обе функции: `readParams()` и `writeParams()`.

А теперь давайте попробуем решить ту же задачу с помощью простых классов. Сначала создадим абстрактный базовый класс, который будет определять интерфейс для типа.

```
abstract class ParamHandler {
   protected $source;
   protected $params = array();

   function __construct( $source ) {
      $this->source = $source;
   }

   function addParam( $key, $val ) {
      $this->params[$key] = $val;
   }

   function getAllParams() {
      return $this->params;
   }

   static function getInstance( $filename ) {
      if ( preg_match( "/\.xml$/i", $filename )) {
         return new XmlParamHandler( $filename );
      }
      return new TextParamHandler( $filename );
}

   abstract function write();
   abstract function read();
}
```

Я определил метод `addParam()`, чтобы позволить пользователю добавлять параметры к защищенному свойству `$params`, и метод `getAllParams()`, чтобы предоставить доступ к копии массива параметров.

Я также создал статический метод `getInstance()`, который проверяет расширение файла и возвращает объект конкретного подкласса в соответствии с результатами проверки. Решающим является то, что мы определили два абстрактных метода, `read()` и `write()`, тем самым гарантировав, что любые подклассы будут поддерживать этот интерфейс.

На заметку. Поместить статический метод для генерации дочерних объектов в родительский класс — это удобно. Но такое проектное решение будет иметь последствия, поскольку теперь вы существенно ограничили возможности класса `ParamHandler` в плане работы с его конкретными подклассами, возвращаемыми рассматриваемым статическим методом из центрального условного оператора. Что произойдет, если вам понадобится работать с еще одним форматом конфигурационных файлов? Конечно, 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 применять корректирующий код (патч) при каждой повторной инсталляции пакета, в котором этот класс используется). Вопросы создания объектов мы обсудим в главе 9.

А теперь давайте определим подклассы, снова опуская детали реализации с целью сохранения понятности кода.

```
class XmlParamHandler extends ParamHandler {

   function write() {
      // Запись в формате XML
      // массива параметров $this->params
   }

   function read() {
      // Чтение из XML-файла и
      // запись значений в массив $this->params
   }
}

class TextParamHandler extends ParamHandler {

   function write() {
      // Запись в текстовый файл
      // массива параметров $this->params
   }

   function read() {
      // Чтение из текстового файла и
      // запись значений в массив $this->params
   }
}
```

Эти классы просто обеспечивают реализации методов `read()` и `write()`. Каждый класс будет считывать и записывать данные согласно соответствующему формату.

В результате из клиентского кода можно будет записывать параметры конфигурации в текстовый и XML-файлы совершенно ясно и прозрачно, в зависимости от расширения имени файла.

```
$test = ParamHandler::getInstance( "./params.xml" );
$test->addParam("key1", "val1" );
$test->addParam("key2", "val2" );
$test->addParam("key3", "val3" );
$test->write(); // Запись файла в XML-формате
```

Мы также можем прочитать параметры конфигурации из файла любого поддерживаемого формата.

```
$test = ParamHandler::getInstance( "./params.txt" );
$test->read(); // Читаем данные из текстового файла
```

Итак, какие выводы можно сделать из этих двух подходов?

Ответственность

Управляющий код в "процедурном" примере несет ответственность за выбор формата, но принимает это решение не один раз, а дважды. Условный код, конечно, помещен в функции, но это просто скрывает факт принятия решений в одном потоке. Вызов функции `readParams()` всегда имеет место в контексте, отличном от контекста вызова функции `writeParams()`, поэтому мы вынуждены повторять проверку расширения файла в каждой функции (или выполнять вариации этой проверки).

В объектно-ориентированной версии выбор формата файла делает статический метод `getInstance()`, который проверяет расширение файла только один раз и создает нужный подкласс. Клиентский код не несет ответственности за реализацию. Он использует предоставленный объект, не зная и даже не интересуясь, какому конкретному подклассу он принадлежит. Он только знает, что работает с объектом `ParamHandler` и что он поддерживает методы `read()` и `write()`. В то время как процедурный код занимается деталями, объектно-ориентированный код работает только с интерфейсом, не заботясь о деталях реализации. Поскольку ответственность за реализацию лежит на объектах, а не на клиентском коде, будет легко включить поддержку новых форматов явным и прозрачным способом.

Связность

Связность (cohesion) — это степень, в которой соседние процедуры связаны одна с другой. В идеальном случае вы должны создавать компоненты, которые разделяют ответственность явным образом. Если по всему коду разбросаны связанные процедуры, то вы вскоре обнаружите, что их трудно сопровождать, потому что придется прилагать усилия для поиска мест, куда нужно внести изменения.

В наших классах типа `ParamHandler` связанные процедуры собраны в общем контексте. Методы для работы с XML-форматом совместно используют один контекст, в котором они так же совместно используют данные и где, в случае необходимости, изменения в одном методе могут быть легко отражены в другом (например, если нужно изменить имя XML-элемента). И тогда можно сказать, что классы `ParamHandler` имеют высокую связность.

С другой стороны, в процедурном примере связанные процедуры разделены. Код для работы с XML-форматом разбросан по нескольким функциям.

Тесная связь

Тесная связь (coupling) происходит, когда отдельные части кода системы тесно связаны одна с другой, так что изменение в одной части влечет необходимость изменений в других частях. Тесная связь не обязательно возникает в процедурном коде, но последовательная природа такого кода приводит к определенным проблемам.

Такой вид тесной связи можно увидеть в примере процедурного кода. Функции `readParams()` и `writeParams()` осуществляют одну и ту же проверку расширения файла, чтобы определить, как они должны работать с данными. Любое изменение

в логике, которое осуществляется для одной функции, должно быть реализовано и для другой. Например, в случае добавления нового формата нужно было бы привести все функции в соответствие, чтобы новое расширение файла обрабатывалось в них одинаково. По мере добавления новых функций, обрабатывающих другие форматы файлов, эта проблема будет только усугубляться.

А в примере объектно-ориентированного кода классы отделяются один от другого и от клиентского кода. Если бы нам потребовалось добавить поддержку нового формата файла, то мы могли бы просто создать новый подкласс, изменив единственную проверку в статическом методе `getInstance()`.

Ортогональность

Потрясающее сочетание компонентов с четко определенными обязанностями наряду с независимостью от более широкого контекста иногда называют *ортогональностью (orthogonality)*, как, например, в книге Эндрю Ханта (Andrew Hunt) и Дэвида Томаса (David Thomas) *The Pragmatic Programmer* (Addison-Wesley Professional, 1999).

Как утверждают авторы, ортогональность способствует повторному использованию кода, поскольку готовые компоненты можно включать в новые системы, не делая никакой их специальной настройки. Такие компоненты должны иметь четко определенные входные и выходные данные, независимые от какого-либо более широкого контекста. В ортогональный код легче вносить изменения, поскольку изменение реализации будет локализовано тем компонентом, в который 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 "ударить" по более широкой системе.

Но нежесткая связь и высокая связность не являются автоматическими признаками в контексте класса. В конце концов, мы можем включить целый пример процедурного кода в один "неправильный" класс. Как же нам достичь необходимого баланса в коде? Обычно я начинаю с рассмотрения классов, которые должны присутствовать в моей системе.

Выбор классов

Вы можете столкнуться с тем, что определить границы классов на удивление трудно, особенно по мере того, как они будут развиваться вместе с создаваемой системой.

При моделировании реальной ситуации это может показаться простым. Обычно объектно-ориентированные системы — это программное представление реальных вещей, поскольку часто используются классы `Person`, `Invoice` и `Shop`. Отсюда можно предположить, что определение классов — это вопрос нахождения некоторых *объектов* в системе и придания им функциональности с помощью методов. Это неплохая отправная точка, но здесь таятся некоторые опасности. 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 операций.

Давайте рассмотрим пример с классом `ShopProduct`, который мы создали в главе 3. Наша система предназначена для того, чтобы продавать товары покупателям, поэтому определение класса `ShopProduct` — это очевидное решение. Но будет ли это решение единственным? Для доступа к данным о товаре мы предоставляем методы `getTitle()` и `getPrice()`. Если нас попросят обеспечить механизм для вывода кра-

ткой информации о товаре для счетов-фактур и уведомлений о доставке товаров, то, наверное, имеет смысл определить метод `write()`. Когда клиент попросит нас обеспечить механизм выдачи краткой информации в различных форматах, мы снова обратимся к нашему классу и надлежащим образом создадим методы `writeXML()` и `writeXHTML()` в дополнение к методу `write()`. Либо добавим в код метода `write()` условные операторы, чтобы выводить данные в различных форматах в соответствии со значением входного аргумента.

В любом случае проблема заключается в том, что класс `ShopProduct` теперь пытается делать слишком много. Кроме хранения информации о самом товаре, наш класс еще должен управлять стратегиями представления этих данных.

Так как же мы *должны* определять классы? Наилучший подход — представлять, что класс имеет основную обязанность, и сделать эту обязанность как можно более единичной и специализированной. Выразите эту обязанность словами. Считается, что обязанность класса должна описываться не более чем 25 словами с редкими включениями союзов "и" и "или". И если предложение становится слишком длинным или "тонет" в дополнительных формулировках, значит, пришло время подумать об определении новых классов с новыми обязанностями.

Например, обязанность класса `ShopProduct` — хранить информацию о товаре. И если мы добавляем методы для вывода данных в различных форматах, то тем самым вводим новую сферу ответственности (т.е. обязанностей) — представление информации о продукте. Как вы видели в главе 3, на самом деле на основе этих отдельных обязанностей мы определили не один, а два класса. Класс `ShopProduct` остался отвечать за хранение информации о товарах, а `ShopProductWriter` берет на себя ответственность за представление этой информации. А уточнение этих обязанностей осуществляется с помощью отдельных подклассов.

На заметку. Далеко не все правила проектирования являются очень жесткими. Например, иногда вам будет встречаться код, предназначенный для сохранения объектных данных, который находится в другом, совершенно не связанном с текущим, классе. Очевидно, что самое удобное место для такой функциональности — это текущий класс, хотя на первый взгляд может показаться, что такой подход нарушает правило о том, что у класса должна быть только одна обязанность. Такое удобство связано с тем, что локальный метод будет иметь полный доступ к полям экземпляра класса. Использование локальных методов для сохранения объектных данных (персистентности) также позволит нам не создавать параллельную иерархию персистентных классов, которые являются зеркальным отображением сохраняемых классов, что ведет к неизбежному увеличению уровня связи. Другие стратегии обеспечения сохраняемости объектов мы рассмотрим в главе 12. Но старайтесь не воспринимать правила проектирования как религиозную догму; правила не могут заменить анализ задачи, стоящей перед вами. Следуйте в большей степени смыслу правила, а не самому правилу.

Полиморфизм

Полиморфизм (polymorphism), или замена классов, — это общее свойство объектно-ориентированных систем. Вы уже несколько раз встречались с ним в этой книге.

Полиморфизм — это поддержка нескольких реализаций на основе общего интерфейса. И хотя это звучит непривычно, на самом деле вы с этим понятием уже знакомы. О необходимости полиморфизма обычно говорит наличие в коде большого количества условных операторов.

Когда в главе 3 был впервые создан класс `ShopProduct`, мы проводили эксперименты с одним классом и использовали его для хранения информации о книгах и

компакт-дисках, а также о других товарах общего назначения. Чтобы вывести краткую справку о товаре, мы использовали набор условных операторов.

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

Эти операторы определили функциональность двух подклассов: `CDProduct` и `BookProduct`.

Кроме того, в нашем примере процедурного кода в условных операторах, проверяющих значения параметров, заложено "зерно" объектно-ориентированной структуры, к которой мы в конце концов пришли. Одни и те же условия мы проверяем в двух частях сценария.

```
function readParams( $source ) {
   $params = array();
   if ( preg_match( "/\.xml$/i", $source )) {
      // Читаем параметры в формате XML из $source
   } else {
      // Читаем текстовые параметры из $source
   }
   return $params;
}

function writeParams( $params, $source ) {
   if ( preg_match( "/\.xml$/i", $source )) {
      // Записываем параметры в формате XML в $source
   } else {
      // Записываем текстовые параметры в $source
   }
}
```

В каждой части напрашивается выделить один из подклассов, которые мы в конце концов и создаем: `XmlParamHandler` и `TextParamHandler`. В этих классах реализуются методы `write()` и `read()` абстрактного базового класса `ParamHandler`.

```
// Оператор возвращает объект типа XmlParamHandler или TextParamHandler
$test = ParamHandler::getInstance( $file );

$test->read(); // Вызывается XmlParamHandler::read()
               ..   или TextParamHandler::read()
$test->addParam("key1", "val1" );
$test->write(); // Вызывается XmlParamHandler::write()
                //  или TextParamHandler::write()
```

Важно отметить, что полиморфизм не отрицает использование условных операторов. Обычно в методах наподобие `ParamHandler::getInstance()` с помощью операторов `switch` или `if` определяется, какие объекты нужно возвращать. Но это ведет к сосредоточению кода с условными операторами в одном месте.

Как мы видели, в PHP 5 программиста вынуждают создавать интерфейсы, определяемые абстрактными классами. Это удобно, потому что мы можем быть уверенными, что конкретный дочерний класс будет поддерживать в точности те же сигнатуры методов, которые были определены в абстрактном родительском классе. В них будут включены все уточнения типов классов и элементы контроля доступа. Поэтому в клиентском коде можно рассматривать все дочерние классы общего суперкласса как взаимозаменяемые (если в нем используются только функции, определенные в родительском классе). Но у этого правила есть важное исключение: пока что не существует способа ограничить тип, возвращаемый методом.

На заметку. На момент написания книги планы по включению в PHP функции уточнения возвращаемых типов были объявлены. Но как скоро это будет сделано — тайна, покрытая мраком.

Поскольку нельзя указать тип возвращаемого значения, методы из различных подклассов могут возвращать различные типы объектов или значения элементарного типа. А это уже угроза взаимозаменяемости типов, ведь вы должны стремиться к тому, чтобы поддерживать согласованность возвращаемых значений. В некоторых случаях из такого нежесткого определения типов можно извлечь определенные преимущества, когда нужно в зависимости от ситуации вернуть значение соответствующего типа. Однако чаще всего существует неявное соглашение между клиентским кодом и используемыми в нем объектами, согласно которому методы должны всегда возвращать значения заранее оговоренного типа. И если такое соглашение установлено в абстрактном суперклассе, то оно должно соблюдаться и в конкретных дочерних классах. Тогда программист клиентского кода может быть уверен в том, что все используемые им объекты работают согласованно. Если вас обязывают вернуть объект определенного типа, то вы, конечно, можете вернуть экземпляр объекта, относящегося к его подтипу. Хотя интерпретатор не обязывает возвращать значения определенного типа, вы можете создать соглашение в проекте о том, что для определенных методов требуется согласовать типы возвращаемых значений. Для этого используйте комментарии в исходном коде.

Инкапсуляция

Инкапсуляция (encapsulation) — это просто сокрытие данных и функциональности от клиентского кода. И опять-таки, это ключевое понятие объектно-ориентированного программирования.

На самом простейшем уровне мы инкапсулируем данные, объявляя свойства как `private` или `protected`. Скрывая свойство от клиентского кода, мы вводим в действие интерфейс и тем самым предотвращаем случайное повреждение данных объекта.

Полиморфизм иллюстрирует другой вид инкапсуляции. Размещая различные реализации за общим интерфейсом, мы скрываем работающий в их основе механизм от клиентского кода. Это означает, что любые изменения, внесенные за этим интерфейсом, являются прозрачными для более широкой системы. Мы можем добавлять новые классы или менять код в классе, что не приведет к возникновению ошибок. Значение имеет интерфейс, а не механизм, работающий в его основе. Чем более независимы эти механизмы, тем меньше вероятность того, что внесенные изменения или поправки будут иметь "эффект домино" для ваших проектов.

Инкапсуляция — это в некотором отношении ключ к объектно-ориентированному программированию. Наша цель — сделать каждую часть как можно более независимой от других. Классы и методы должны получать столько информации,

сколько необходимо для выполнения назначенных им задач, которые должны быть ограничены по контексту и четко определены.

Введение ключевых слов `private`, `protected` и `public` облегчает инкапсуляцию. Но инкапсуляция — это также и образ мыслей. В 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;
```

Конечно, код должен быть внимательно проверен, потому что конфиденциальность не была строго введена. Но, что интересно, ошибки случались редко, потому что структура и стиль кода довольно четко определяли, какие свойства трогать нельзя.

Кроме того, даже в PHP 5 мы можем нарушить правила и выяснить точный подтип объекта, который мы используем в контексте замены классов, просто используя оператор `instanceof`.

```
function workWithProducts( ShopProduct $prod ) {
   if ( $prod instanceof CDProduct ) {
      // Обработка данных CD
   } else if ($prod instanceof BookProduct ) {
      // Обработка данных книги
   }
}
```

Для выполнения подобных действий должна быть серьезная причина, поскольку в целом это вносит определенный элемент сумбура. Запрашивая конкретный подтип, как в данном примере, мы устанавливаем жесткую зависимость. Если же специфика подтипа скрыта полиморфизмом, то можно совершенно безболезненно изменить иерархию наследования класса `ShopProduct`, не опасаясь негативных последствий. Но в нашем коде этому положен предел. Теперь, если нам нужно усовершенствовать классы `CDProduct` и `BookProduct`, мы должны помнить о возможности нежелательных побочных эффектов в методе `workWithProducts()`.

Из этого примера мы должны вынести два урока. Во-первых, инкапсуляция помогает создать ортогональный код. Во-вторых, степень инкапсуляции, которую можно ввести в силу, к делу не относится. Инкапсуляция — это методика, которую должны соблюдать в равной степени и классы, и их клиенты.

Забудьте, как это делается

Если вы похожи на меня, то упоминание о некоторой задаче заставит ваш ум интенсивно работать в поисках механизма решения. Вы можете выбрать функции, которые могут решить данную задачу, поискать регулярные выражения, исследовать пакеты PEAR. Возможно, у вас есть фрагмент кода из старого проекта, который выполняет подобное, и его можно вставить в новый код. На этапе разработки вы выиграете, если на некоторое время отставите все это в сторону. Освободите голову от процедур и механизмов.

Думайте только о ключевых участниках системы: типах, которые ей нужны, и интерфейсах. Конечно, знание всего процесса поможет в ваших размышлениях. Классу, который открывает файл, нужно передать имя этого файла; код базы данных должен оперировать именами таблиц, паролями и т.д. Но старайтесь, чтобы

структуры и отношения в коде вели и направляли вас. Вы обнаружите, что реализация легко встанет на место, если будет хорошо определен интерфейс. И затем вы получите широкие возможности заменить, улучшить или расширить реализацию, если вам это понадобится, не затрагивая более широкую систему.

Чтобы сделать упор на интерфейсе, размышляйте в терминах абстрактных базовых классов, а не конкретных дочерних классов. Например, в нашем коде извлечения параметров, интерфейс — это самый важный аспект проектирования. Нам нужен тип, который считывает и записывает пары "имя–значение". Для типа важна именно эта обязанность, а не реальный носитель, на котором будут сохраняться данные или способы их хранения и извлечения. Мы проектируем систему вокруг абстрактного класса `ParamHandler` и добавляем только конкретные стратегии для того, чтобы в дальнейшем в реальном приложении можно было считывать и записывать параметры. Таким образом, мы встраиваем в нашу систему полиморфизм и инкапсуляцию с самого начала. Такая структура уже тяготеет к использованию замены классов.

Но, конечно, говоря это, мы знали с самого начала, что должны существовать текстовая и XML-реализации класса `ParamHandler`, которые, безусловно, оказывают влияние на наш интерфейс. При проектировании интерфейсов всегда есть несколько вариантов принятия решения.

Банда четырех (в книге *Design Patterns*) сформулировала этот принцип с помощью следующей фразы: "*Программируйте на основе интерфейса, а не его реализации*". Это высказывание заслуживает того, чтобы вы добавили его в свою записную книжку.

Четыре столпа

Очень немногие люди принимают абсолютно правильные решения еще на стадии проектирования. Большинство программистов исправляют код по мере изменения требований к нему либо в результате более полного понимания природы решаемой задачи.

Но по мере изменения кода он может, в конце концов, выйти из-под нашего контроля. Здесь мы добавили метод, здесь — класс, и в результате система постепенно начинает усложняться. Но, как мы уже видели, сам код может указывать пути для его совершенствования. Такие места в коде иногда называют "запахами". Речь идет о тех моментах в коде, в которые *нужно будет* внести конкретные исправления или которые по меньшей мере заставят вас еще раз проанализировать проект. В данном разделе я выделю некоторые моменты и представлю их в виде четырех аксиом, которые необходимо всегда иметь в виду при написании кода.

Дублирование кода

Дублирование — один из самых больших недостатков кода. Если во время написания процедуры у вас появляется странное чувство "дежавю", то, вероятнее всего, есть проблемы.

Посмотрите на повторяющиеся элементы в системе. Возможно, они связаны один с другим. Дублирование обычно говорит о наличии тесной связи. Если вы изменяете что-то фундаментальное в одной процедуре, то понадобится ли вносить исправления в схожие процедуры? Если ситуация именно такова, то, вероятно, они принадлежат одному и тому же классу.

Класс, который слишком много знал

Это может быть очень утомительно — пересылать параметры туда-сюда от одного метода к другому. Почему бы не упростить себе жизнь путем использования глобальной переменной? С помощью глобальной переменной каждый может получить доступ к данным.

Глобальные переменные очень важны, но на них нужно смотреть с некоторым подозрением. Причем с большим подозрением. Используя глобальную переменную или давая классу любые виды знаний о более широком контексте, вы привязываете его к контексту, делая менее пригодным для повторного использования и зависимым от кода, который находится за пределами его контроля. Помните о том, что вам следует разъединять классы и процедуры и не создавать взаимозависимости. Постарайтесь ограничить знание класса о его контексте. Некоторые стратегии осуществления этого мы рассмотрим далее в книге.

На все руки мастер

А если класс пытается делать слишком много сразу? В этом случае попробуйте составить список обязанностей класса. Может оказаться, что одна из них может легко создать основу для отдельного класса.

Оставить "слишком работящий класс" без изменений — значит создать определенные проблемы в случае создания подклассов. Какую обязанность вы расширяете с помощью подкласса? Что будете делать, если вам понадобится подкласс для более чем одной обязанности? Очень вероятно, что у вас получится слишком много подклассов или слишком сильная зависимость от условного кода.

Условные операторы

Конечно, у вас будут серьезные причины для использования в коде операторов `if` и `switch`. Но иногда наличие подобных структур является сигналом к полиморфизму.

Если вы обнаружили, что проверяете некоторые условия в классе слишком часто, особенно если эти проверки повторяются в нескольких методах, то, возможно, это сигнал о том, что из одного класса нужно сделать два или больше. Выясните, не предполагает ли структура условного кода обязанности, которые можно выразить в классах. Эти новые классы реализуют совместно используемый абстрактный базовый класс. Вполне вероятно, что затем вам понадобится найти способ передать нужный класс клиентскому коду. О некоторых шаблонах создания объектов читайте в главе 9.

UML

До сих пор в данной книге я позволял коду говорить самому за себя и использовал краткие примеры для иллюстрации таких понятий, как наследование и полиморфизм.

Это полезно, потому что речь идет о PHP, который находится в центре нашего внимания. Но по мере увеличения размеров и сложности примеров использование только одного кода для иллюстрации больших изменений в проекте станет несколько абсурдным. Согласитесь, трудно увидеть общую картину в нескольких строках кода.

UML расшифровывается как "Unified Modeling Language" (унифицированный язык моделирования). История его создания довольно интересна. По словам Мартина Фаулера (Martin Fowler) (*UML Distilled*, Addison-Wesley Professional, 1999), UML возник как стандарт после многолетних интеллектуальных и бюрократических споров в сообществе приверженцев объектно-ориентированного проектирования.

Результатом этой борьбы стал мощный графический синтаксис для описания объектно-ориентированных систем. В данном разделе мы рассмотрим его только в общих чертах, но вы вскоре поймете, что "малыш" UML прошел долгий путь.

Диаграммы классов позволяют описывать структуры и шаблоны, так что их смысл становится ясным и понятным. Такой кристальной ясности трудно достичь с помощью фрагментов кода или маркированных списков.

Диаграммы классов

Диаграммы классов — это только один аспект UML, но, вероятно, они чаще всего используются. И поскольку они чрезвычайно полезны для описания объектно-ориентированных связей, я тоже буду использовать их в книге.

Представление классов

Как и можно было ожидать, классы — это главные составные элементы диаграмм классов. Класс представляется в виде прямоугольника с именем, как показано на рис. 6.1.

Прямоугольник, представляющий класс, делится на три раздела, в первом из которых отображается имя. Разделительные линии необязательны, если, помимо имени класса, больше никакой дополнительной информации не предоставляется. Создавая диаграммы, вы увидите, что для некоторых классов вполне достаточно того уровня детализации, который представлен на рис. 6.1. Мы вовсе не обязаны представлять все поля и методы, и даже все классы, на диаграмме классов.

Абстрактные классы представляют, либо выделяя имя класса курсивом, как показано на рис. 6.2, либо добавляя к имени класса уточнение `{abstract}`, как показано на рис. 6.3. Первый способ более распространенный, а второй более удобный, если вы делаете заметки в своем блокноте.

На заметку. Надпись `{abstract}` — это пример уточнения. Уточнения в диаграммах классов служат для описания того, каким способом должны использоваться конкретные элементы. Для текста в фигурных скобках не существует специальных правил; он должен просто дать понятное объяснение каких-либо условий, которые применяются к элементу.

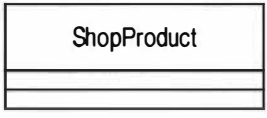


Рис. 6.1. Представление класса в UML

Рис. 6.2. Пример представления абстрактного класса

Рис. 6.3. Представление абстрактного класса, определенного с помощью уточнения

Интерфейсы определяются таким же образом, как классы, за исключением того, что они должны включать стереотип (т.е. расширение словаря UML), как показано на рис. 6.4.

Атрибуты

Если говорить в общем, то атрибуты описывают свойства класса. Атрибуты перечисляются в разделе, который расположен непосредственно под именем класса, как показано на рис. 6.5.

Давайте рассмотрим атрибут из этого примера повнимательнее. Первый символ (#) обозначает уровень видимости, или контроль доступа, для атрибута. В табл. 6.1 перечислены три возможных варианта символа видимости.

Таблица 6.1. Символы видимости

Символ	Видимость	Описание
+	Общедоступный	Доступный для всего кода
-	Закрытый	Доступный только для текущего класса
#	Защищенный	Доступный только для текущего класса и его подклассов

За символом видимости следует имя атрибута. В данном случае мы описываем свойство `ShopProduct::$price`. Двоеточие используется для отделения имени атрибута от его типа (и, возможно, от его стандартного значения).

И снова повторяю, что вы можете включать ровно столько деталей, сколько это необходимо для ясности.

Операции

Операции описывают методы или, точнее, вызовы, которые могут быть сделаны по отношению к экземпляру класса. На рис. 6.6 показаны две операции класса `ShopProduct`.

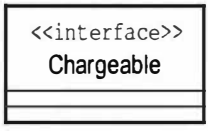


Рис. 6.4. Представление интерфейса

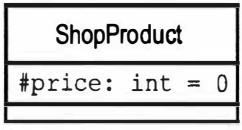


Рис. 6.5. Представление атрибута

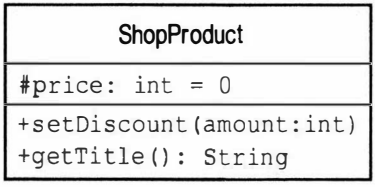


Рис. 6.6. Представление операций

Как видите, для операций используется такой же синтаксис, как и для атрибутов. Символ видимости предшествует имени метода. Список параметров заключается в круглые скобки. Тип, возвращаемый методом, если он есть, отделен двоеточием. Параметры разделяются запятыми, и для них используется такой же синтаксис, как и для атрибута: имя атрибута отделяется от его типа двоеточием.

Как и можно было ожидать, это достаточно гибкий синтаксис. Вы можете опустить признак видимости и возвращаемый тип. Параметры часто представляются только их типом, поскольку имя аргумента обычно не имеет особого значения.

Описание наследования и реализации

В UML отношения наследования описываются в виде обобщений. Это отношение обозначается линией, ведущей от подкласса к родительскому классу. Эта линия заканчивается незакрашенной замкнутой стрелкой.

На рис. 6.7 показана связь между классом `ShopProduct` и его дочерними классами.

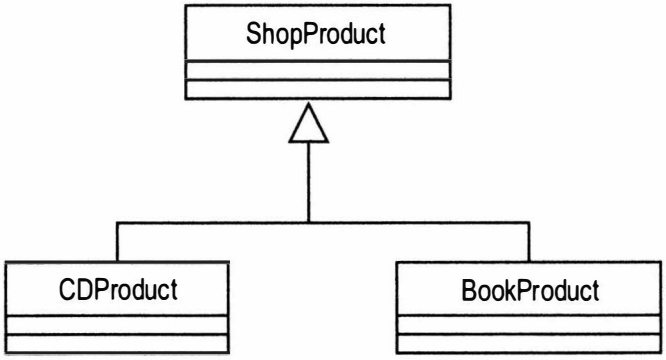


Рис. 6.7. Описание наследования

С помощью UML описывается также связь между интерфейсом и классами, которые его реализуют. Так, если бы в классе `ShopProduct` был реализован интерфейс `Chargeable`, мы бы добавили его к диаграмме класса, как показано на рис. 6.8.

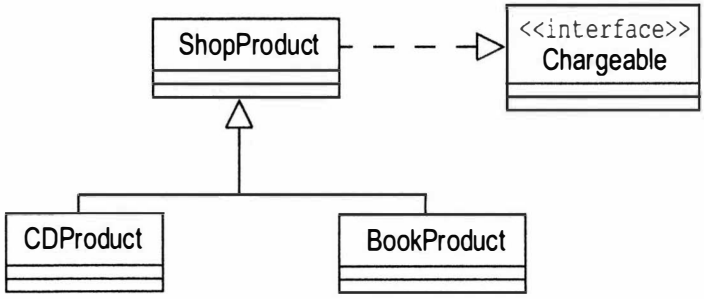


Рис. 6.8. Описание реализации интерфейса

Ассоциации

Наследование — это только одно из отношений в объектно-ориентированной системе. Ассоциация происходит, когда объявляется, что в свойстве класса содержится ссылка на экземпляр (или экземпляры) другого класса.

На рис. 6.9 мы моделируем два класса и создаем ассоциацию между ними.

Рис. 6.9. Ассоциация класса

На данном этапе нам неизвестна природа этих отношений. Мы только указали, что у объекта `Teacher` будет ссылка на один или более объектов `Pupil` или наоборот. Это отношение может быть или не быть взаимным.

Стрелки используются для описания направления ассоциации. Если в классе `Teacher` существует ссылка на экземпляр класса `Pupil`, но не наоборот, то мы должны определить ассоциацию с помощью стрелки, направленной от класса `Teacher` к классу `Pupil`. Эта ассоциация, которая называется однонаправленной, показана на рис. 6.10.

Рис. 6.10. Однонаправленная ассоциация

Рис. 6.11. Двунаправленная ассоциация

Если в каждом классе существует ссылка на другой класс, то для описания двунаправленного отношения используется двунаправленная стрелка, как показано на рис. 6.11.

Можно указать также количество экземпляров класса, на которые ссылается другой класс в ассоциации. Для этого нужно поместить число или диапазон чисел рядом с каждым классом. Можно также использовать звездочку (*), обозначающую любое число. На рис. 6.12 показано, что может существовать только один объект `Teacher` и нуль или больше объектов `Pupil`.

На рис. 6.13 показано, что в ассоциации может быть один объект `Teacher` и от пяти до десяти объектов `Pupil`.

Рис. 6.12. Определение множественности для ассоциации

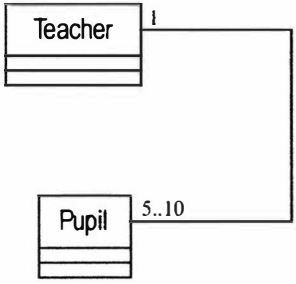


Рис. 6.13. Еще один вариант определения множественности для ассоциации

Агрегирование и композиция

Агрегирование и композиция похожи на ассоциацию. Все эти термины служат для описания ситуации, когда в классе содержится постоянная ссылка на один или более экземпляров другого класса. Но с помощью агрегирования и композиции экземпляры, на которые ссылаются, формируют внутреннюю часть ссылающегося объекта.

В случае агрегирования содержащиеся объекты составляют основную часть объекта-контейнера (который их содержит), но они могут также одновременно содержаться и в других объектах. Отношение агрегирования обозначается линией, которая начинается с незакрашенного ромба.

На рис. 6.14 мы определяем два класса: `SchoolClass` и `Pupil`. Класс `SchoolClass` агрегирует класс `Pupil`.

Класс состоит из учеников, но на один и тот же объект `Pupil` могут одновременно ссылаться различные экземпляры класса `SchoolClass`. Если бы нам понадобилось распустить школьный класс, для этого необязательно удалять ученика, который может посещать другие классы.

Композиция представляет собой еще более сильное отношение. В композиции на содержащийся объект может ссылаться только его объект-контейнер. И он должен быть удален при удалении объекта-контейнера. Отношения композиции изображаются так же, как и отношения агрегирования, только с закрашенным ромбом. Отношение композиции проиллюстрировано на рис. 6.15.

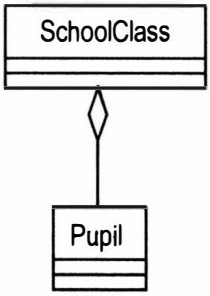


Рис. 6.14. Агрегирование

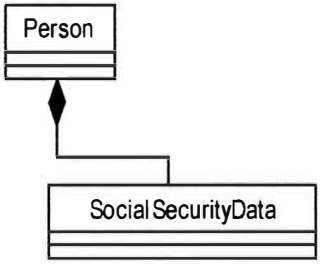


Рис. 6.15. Композиция

В классе `Person` существует постоянная ссылка на объект `SocialSecurityData`. Экземпляр этого объекта может принадлежать только содержащему его объекту `Person`.

Описание отношений использования

В 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` на рис. 6.16 используется объект `ShopProductWriter`. Отношение использования изображается с помощью пунктирной линии со стрелкой с незамкнутым контуром на конце; эта линия соединяет рассматриваемые класс и объект. Тем не менее при этом отношении ссылка на объект не хранится в виде свойства, как, например, в объекте `ShopProductWriter`, где хранится массив ссылок на объекты типа `ShopProduct`.

Использование примечаний

На диаграммах классов отображается структура системы, но они не дают представления о самом процессе. На рис. 6.16 показаны классы нашей системы. Мы знаем, что в объекте `Report` используется объект `ShopProductWriter`, но не знаем механизма этого процесса. На рис. 6.17 используются примечания, которые помогают немного прояснить ситуацию.

Как видите, примечание представляет собой прямоугольник с загнутым уголком. Обычно в нем содержатся фрагменты псевдокода.

Примечание вносит некоторую ясность в диаграмму; теперь мы видим, что в объекте `Report` используется объект `ShopProductWriter` для вывода данных о продукте. На самом деле отношения использования не всегда так очевидны. В некоторых случаях даже примечание может не дать достаточной информации. Но, к счастью, мы можем моделировать взаимодействие объектов в нашей системе, а также структуру классов.

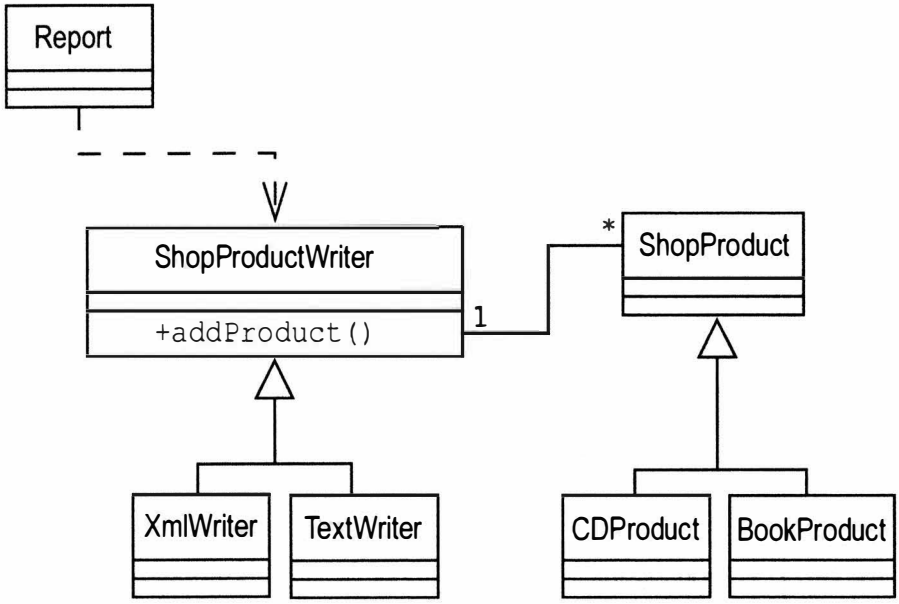


Рис. 6.16. Отношение использования

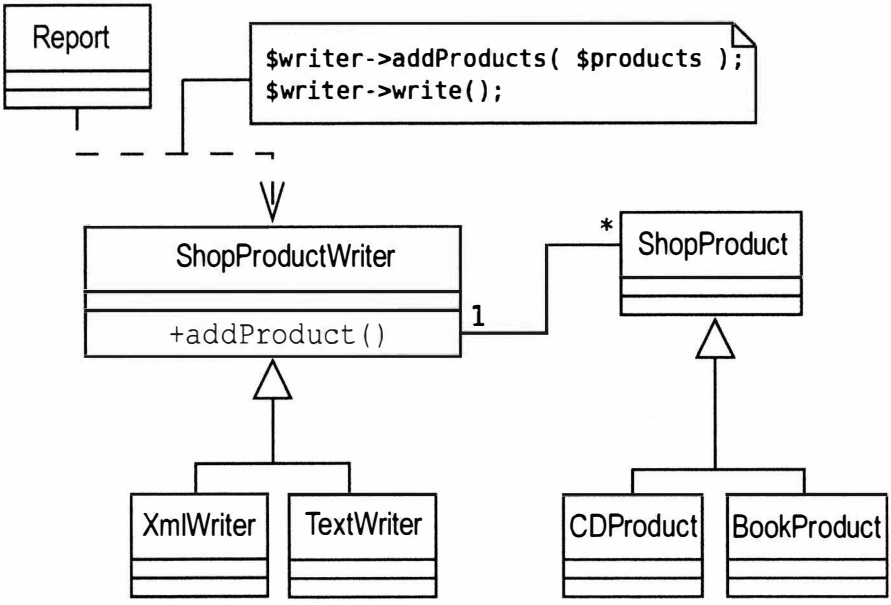


Рис. 6.17. Использование примечаний для прояснения ситуации

Диаграмма последовательности

В основе *диаграммы последовательности* лежит объект, а не класс. Она используется для поэтапного моделирования процесса в системе.

Давайте составим простую диаграмму, моделируя способ, с помощью которого объект `Report` выводит данные о продукте. На диаграмме последовательности объекты системы отображаются справа налево, как показано на рис. 6.18.

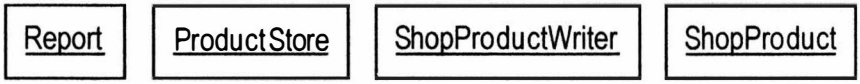


Рис. 6.18. Объекты системы на диаграмме последовательности

Мы пометили объекты только именем класса. Если бы у нас было несколько экземпляров одного класса, работающих независимо, мы бы включили в надписи имя объекта в формате `метка:класс` (например, `product1:ShopProduct`).

Мы показываем время жизни моделируемого процесса сверху вниз, как на рис. 6.19.

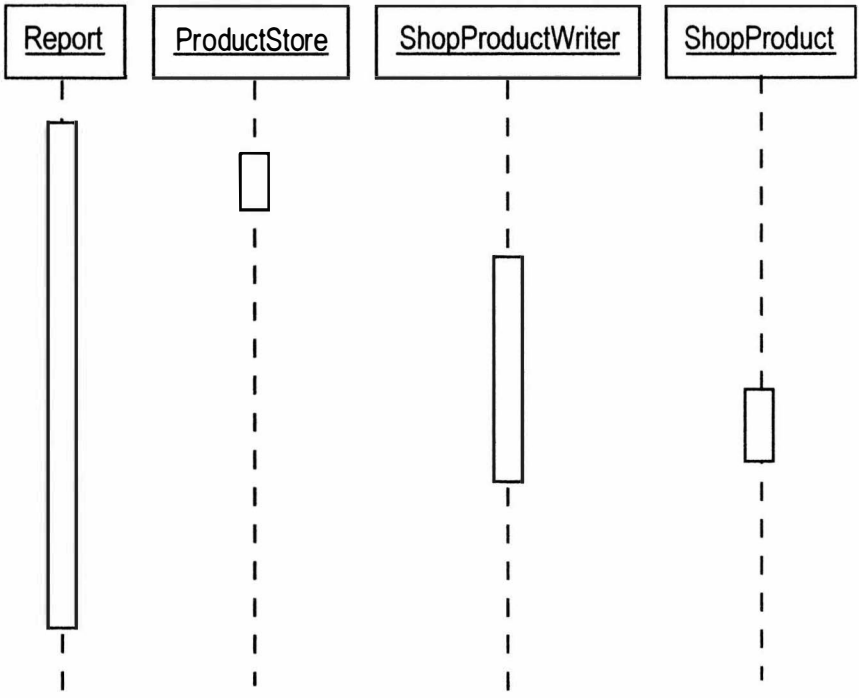


Рис. 6.19. Линии жизни объектов на диаграмме последовательности

Вертикальные пунктирные линии представляют время жизни объектов в системе. Прямоугольники на линиях жизни представляют направленность процесса. Если читать рис. 6.19 сверху вниз, можно увидеть, как процесс движется по объектам в системе. Но это трудно понять без сообщений, которые передаются между объектами. Сообщения добавлены на рис. 6.20.

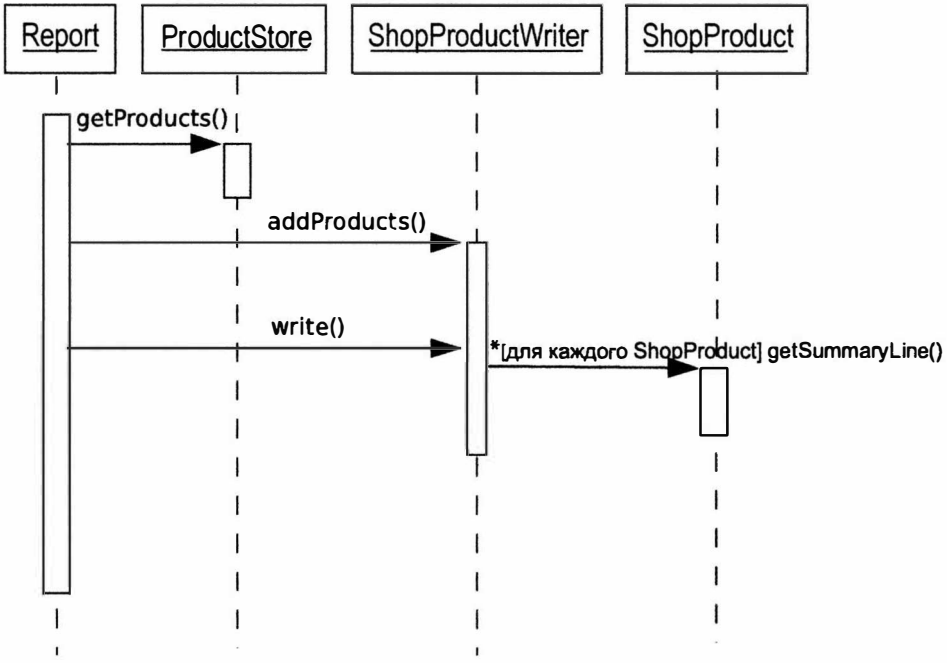


Рис. 6.20. Завершенная диаграмма последовательности

Стрелки представляют собой сообщения, отсылаемые от одного объекта другому. Возвращенные значения часто остаются неявными (хотя они могут быть представлены пунктирной линией, идущей от вызванного объекта к объекту — отправителю сообщения). Каждое сообщение помечается вызовом соответствующего метода. Существует синтаксис для меток. Квадратные скобки обозначают условие. Поэтому запись

```
[okToPrint]
write()
```

означает, что вызов метода `write()` может произойти только при выполнении указанного условия. Звездочка используется для указания повторения и, возможно, с дальнейшим объяснением в квадратных скобках (но это необязательно).

```
*[для каждого ShopProduct]
write()
```

Теперь мы можем интерпретировать рис. 6.20 сверху вниз. Объект `Report` получает список объектов типа `ShopProduct` от объекта `ProductStore`. Он передает этот список объекту `ShopProductWriter`, в котором сохраняются ссылки на эти объекты (хотя из диаграммы мы можем об этом только догадываться). Объект `ShopProduct Writer` вызывает метод `ShopProduct::getSummaryLine()` для каждого объекта типа `ShopProduct`, ссылка на который хранится в массиве, добавляя полученный результат к выходным данным.

Как видите, диаграммы последовательности могут моделировать процессы, фиксируя "срезы" динамического взаимодействия и представляя их с удивительной ясностью.

На заметку. Посмотрите на рис. 6.16 и 6.20. Обратите внимание на то, как на диаграмме класса проиллюстрирован полиморфизм (показаны классы, которые происходят от `ShopProductWriter` и `ShopProduct`). А теперь посмотрите, как этот факт становится очевидным, когда мы моделируем связь между объектами. Мы хотим, чтобы по возможности объекты работали с самыми общими имеющимися типами и чтобы можно было скрыть детали их реализации.

Резюме

В этой главе мы немного отошли от конкретики объектно-ориентированного программирования и рассмотрели некоторые важные вопросы проектирования. Мы исследовали свойства ООП, такие как инкапсуляция, слабая связность и тесная связь, которые являются важными аспектами гибкой и повторно используемой объектно-ориентированной системы. Мы рассмотрели UML как основу, очень важную для работы с шаблонами, о которых речь пойдет далее в этой книге.

Часть III

Шаблоны

Глава 7

Что такое проектные шаблоны и зачем они нужны

Большинство задач, с которыми часто приходится сталкиваться программистам, уже давным-давно решены другими членами нашего сообщества. Проектные шаблоны 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 поделиться с другими новыми идеями проектирования и их результатами. С помощью проектных шаблонов просто выделяют общие задачи, определяют проверенные решения и описывают вероятные результаты. Во многих книгах и статьях описываются особенности конкретных языков программирования, имеющиеся функции, классы и методы. А в каталогах шаблонов, наоборот, акцент сделан на том, как перейти в своих проектах от этих основ ("что именно") к пониманию задач и возможных решений ("почему" и "как").

В этой главе мы познакомимся с проектными шаблонами и рассмотрим некоторые причины их популярности.

Здесь будут рассмотрены следующие темы.

- *Основы шаблонов:* что такое проектные шаблоны?
- *Структура шаблона:* основные элементы проектного шаблона.
- *Преимущества шаблонов:* почему шаблоны стоят того, чтобы их изучить?

Что такое проектные шаблоны

В мире программного обеспечения, шаблон — это реальное проявление генетической памяти организации.

— Гради Буч (Grady Booch), из книги *Core J2EE Patterns*[1]

Шаблон — это решение задачи в некотором контексте.

— "Банда четырех" (The Gang of Four), из книги *Design Patterns: Elements of Reusable Object-Oriented Software*[2]

[1] Дипак Алур, Джон Крупи, Дэн Малкс. *Образцы J2EE. Лучшие решения и стратегии проектирования* (пер. с англ., изд. "Лори", 2013).

[2] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* (пер. с англ., изд. "Питер", 2007).

Как следует из приведенных выше цитат, проектный шаблон — это задача, взятая из практики передовых программистов, решение которой проанализировано и объяснено.

Задачи имеют свойство повторяться, и веб-программистам приходится решать их снова и снова. Как обработать входящий запрос? Как преобразовать данные в команды для нашей системы? Как ввести данные? Как представить результаты? Со временем мы находим более или менее изящные ответы на эти вопросы и создаем неформальный набор методик, которые затем снова и снова используем 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. Их корень — в практике, а не в теории. Но это совсем не означает, что в проектных шаблонах отсутствует элемент теории (как мы увидим в следующей главе). Шаблоны основаны на реальных методах, используемых реальными программистами. Знаменитый приверженец шаблонов Мартин Фаулер говорит, что он открывает шаблоны, а не создает их. Поэтому многие шаблоны будут вызывать у вас чувство "дежавю", ведь вы будете узнавать методы, которые используете сами.

Каталог шаблонов — это не книга кулинарных рецептов. Рецептам можно следовать буквально, а код можно скопировать и вставить в проект с незначительными изменениями. Вам не всегда нужно даже понимать весь код, используемый в этом "рецепте". Проектные шаблоны описывают *подходы* к решению конкретных задач. Детали реализации могут существенно изменяться в зависимости от более широкого контекста. От этого контекста зависят выбор используемого языка программирования, природа приложения, размер проекта и специфика задачи.

Предположим, что в проекте требуется создать систему обработки шаблонов. На основании имени файла шаблона вы должны синтаксически проанализировать его содержимое и построить дерево объектов, представляющих найденные теги.

Сначала синтаксический анализатор сканирует текст на предмет поиска триггерных лексем (trigger tokens). Найдя соответствие, он передает лексему другому объекту-анализатору, который специализируется на чтении содержимого, расположенного внутри тегов. В результате данные шаблона продолжают анализироваться до тех пор, пока не произойдет синтаксическая ошибка, не будет достигнут их конец либо не будет найдена другая триггерная лексема. В случае нахождения такой лексемы, объект-анализатор также должен передать ее на обработку соответствующей программе — скорее всего, анализатору аргументов. Все вместе эти компоненты образуют то, что называется рекурсивным нисходящим синтаксическим анализатором.

Итак, вот наши участники: `MainParser`, `TagParser` и `ArgumentParser`. Мы также определили класс `ParserFactory`, который создает и возвращает эти объекты.

Но, конечно, все идет не так гладко, как хотелось бы, и позже, на одном из совещаний, вы узнаете, что в шаблонах нужно поддерживать несколько синтаксисов. И теперь вам нужно создать параллельный набор объектов-анализаторов в соответствии с синтаксисом: `OtherTagParser`, `OtherArgumentParser` и т.д.

Вам поставлена такая задача: генерировать различные наборы объектов в зависимости от конкретной ситуации, и эти наборы объектов должны быть более или менее "прозрачными" для других компонентов системы. Случилось так, что "Банда четырех" в своей книге определила следующую задачу для шаблона Abstract Factory (Абстрактная фабрика): "Предусмотреть интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов".

Этот шаблон нам прекрасно подходит! По сути нашей задачи мы как раз должны определить и очертить рамки использования данного шаблона. Но решение задачи не имеет ничего общего с операциями вырезания и вставки, как вы увидите в главе 9, где я буду рассказывать о шаблоне Abstract Factory.

Наименование шаблона уже само по себе очень ценно; таким образом создается что-то вроде общего словаря, который годами накапливается и используется в среде профессионалов. Такие условные обозначения помогают в совместных разработках, когда оцениваются и тестируются альтернативные подходы и их различные результаты. Например, при обсуждении семейств альтернативных синтаксических анализаторов вы можете просто сказать коллегам, что система создает каждый набор с помощью шаблона Abstract Factory. Они покивают с умным видом; кто-то сразу поймет, о чем идет речь, а кто-то отметит про себя, что нужно будет узнать об этом позже. Но суть в том, что у этого набора идей и различных результатов есть абстрактный описатель, который способствует созданию более кратких обозначений. Я проиллюстрирую это далее в главе.

И наконец, согласно международному законодательству, некорректно писать о шаблонах, не процитировав Кристофера Александера (Christopher Alexander), профессора архитектуры, работы которого оказали огромное влияние на первых сторонников объектно-ориентированных шаблонов. Вот что он пишет в книге *A Pattern Language*[3] (Oxford University Press, 1977).

> *Каждый шаблон описывает задачу, которая возникает снова и снова, а затем описывает суть решения данной задачи, так что вы можете использовать это решение миллион раз, каждый раз делая это по-разному.*

Важно, что это определение (которое относится к архитектурным задачам и решениям) начинается с формулировки задачи и ее более широкого контекста и движется к решению. В последние годы звучала критика, что проектными шаблонами злоупотребляют, особенно неопытные программисты. Причина в том, что решения применялись там, где не было сформулировано соответствующей задачи и контекста. Шаблоны — это нечто большее, чем конкретная организация классов и объектов, совместно работающих определенным образом. Шаблоны создаются для определения условий, в которых должны применяться решения, и для обсуждения результатов этих решений.

В данной книге основной акцент будет сделан на особенно влиятельном направлении в сфере шаблонов: форме, описанной в книге Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1995). Она посвящена использованию шаблонов при разработке объектно-ориентированного программного обеспечения, и в ней описываются некоторые классические шаблоны, которые присутствуют в большинстве современных объектно-ориентированных проектов.

На заметку. Шаблоны, описанные "Бандой четырех", а также в данной книге, — это настоящие примеры 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 других сфер задач, таких как визуальное проектирование и менеджмент проектов (и, конечно, архитектура). Но когда в этой книге я обсуждаю проектные шаблоны, имею в виду задачи и решения в области разработки объектно-ориентированного программного обеспечения.

---

[3] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* (пер. с англ., изд. "Питер", 2007).

Книга "Банды четырех" очень важна не только потому, что в ней описываются основные 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.

Обзор проектных шаблонов

По сути, проектный шаблон состоит из четырех частей: имени, формулировки задачи, описания решения и результатов.

Имя

Выбор имени для шаблона очень важен. Имена обогащают лексикон программистов; несколько коротких слов могут служить для обозначения довольно сложных задач и решений. Имена должны сочетать в себе краткость и описательность. "Банда четырех" утверждает: "Поиск хороших имен был одной из самых трудных задач при разработке нашего каталога".

Мартин Фаулер согласен с этим утверждением: "Имена шаблонов чрезвычайно важны, потому что одна из целей шаблонов — создать словарь, позволяющий разработчикам общаться более эффективно" (*Patterns of Enterprise Application Architecture*[4], Addison-Wesley Professional, 2002).

В книге *Patterns of Enterprise Application Architecture* Мартин Фаулер совершенствует шаблон доступа к базе данных, который я впервые встретил в книге Дипака Алура (Deepak Alur), Дэна Малкса (Dan Malks) и Джона Крупи (John Crupi) *Core J2EE Patterns* (Prentice Hall, 2003). Фаулер определяет два шаблона, которые описывают специализацию старого шаблона. Логика этого подхода полностью правильна. Один из новых шаблонов моделирует объекты предметной области, в то время как другой — таблицы базы данных (а в предыдущей работе это различие было нечетким). Было трудно заставить себя мыслить в категориях новых шаблонов. Я использовал имя первоначального шаблона при проектировании и в документации так долго, что оно прочно вошло в мой лексикон.

Формулировка задачи

Независимо от изящества решений (а некоторые из них действительно очень изящны), формулировка задачи и ее контекста — это основа шаблона. Сформулировать задачу гораздо труднее, чем применить какое-либо решение из каталога шаблонов. Это одна из причин, по которым некоторые шаблоны могут применяться неправильно.

В шаблонах очень тщательно описываются условия задачи. Сначала кратко описывается сама задача, затем — ее контекст; обычно приводятся типичный пример и одна или несколько диаграмм. Анализируется специфика задачи, ее различные проявления. Также описываются все признаки, которые могут помочь при идентификации задачи.

Решение

Решение сначала кратко описывается вместе с задачей. Оно также описывается подробно, как правило, с использованием UML-класса и диаграмм взаимодействия. В шаблон обычно включается пример кода.

---

[4] Мартин Фаулер. *Шаблоны корпоративных приложений* (пер. с англ., ИД "Вильямс", 2009).

Но хотя код может присутствовать, в качестве решения никогда нельзя использовать метод "вырезать и вставить". Помните, что шаблон описывает подход к решению задачи, поскольку в реализации могут быть сотни нюансов. Представьте, что перед вами — инструкции о том, как сеять хлеб. Если вы просто слепо выполните все указания, то, скорее всего, будете голодать после сбора урожая. Гораздо полезнее подход, основанный на шаблоне, в котором описываются различные условия его применения. Основное решение задачи (заставить хлеб расти) всегда будет одним и тем же (посеять семена, поливать, собрать урожай), но реальные шаги, которые надо будет предпринять, зависят от всевозможных факторов, таких как тип почвы, местность, местные вредные насекомые и т.д.

Мартин Фаулер называет решения, описанные в шаблонах, "полуфабрикатами", т.е. программист должен взять идею решения и закончить ее самостоятельно.

Выводы

Каждое решение по проектированию, которое вы принимаете, будет иметь широкий набор результатов. Конечно, в него должно быть включено удовлетворительное решение поставленной задачи. Решение, однажды примененное, может идеально подходить для работы с другими шаблонами, но его нужно применять с особой осторожностью.

Формат "Банды четырех"

Сейчас, когда я пишу эти строки, передо мной на рабочем столе находятся пять каталогов шаблонов. Даже поверхностный взгляд на шаблоны в каждом каталоге говорит о том, что ни в одном из них не используется такая же структура, как в других. Одни более формализованы, другие очень детализированы (содержат множество подразделов), а третьи более беспорядочны.

Существует ряд четко определенных структур шаблонов, включая первоначальную форму, разработанную Кристофером Александером (александрийская форма), и описательный подход, который используют в Портлендском хранилище шаблонов (Portland Pattern Repository) (портлендская форма). Поскольку "Банда четырех" очень влиятельна и мы будем рассматривать многие из описанных ими шаблонов, давайте изучим несколько разделов, которые они включили в свои шаблоны.

- *Предназначение.* Краткая формулировка цели шаблона. Вы должны 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 UML-класс и диаграммы взаимодействия, описывающие отношения между классами и объекты в решении.
- *Реализация.* В данном разделе рассматриваются детали решения. В нем исследуются любые вопросы, которые могут возникнуть во время применения данного метода, и предоставляются советы по его применению.

- *Пример кода.* Я всегда сразу перехожу к этому разделу. Я считаю, что простой пример кода помогает разобраться в шаблоне. Этот пример часто сведен до минимума с целью демонстрации самой сути решения. Пример может быть написан на любом объектно-ориентированном языке. Разумеется, что в этой книге всегда будет использоваться PHP.
- *Примеры применения.* Реальные системы, в которых встречается шаблон (задача, контекст и решение). Некоторые люди считают, что, для того чтобы шаблон был настоящий, он должен присутствовать по меньшей мере в трех широко известных контекстах. Это иногда называется "правило трех".
- *Связанные шаблоны.* Одни шаблоны порождают другие. Применяя одно решение, вы можете создать контекст, в котором станет полезным другое решение. Эти связи и исследуются в данном разделе. Здесь также могут обсуждаться шаблоны, имеющие что-то общее в задаче и решении, а также любые "предшественники": шаблоны, определенные где-то еще, на основе которых строится текущий шаблон.

Зачем используются проектные шаблоны

Так в чем преимущества шаблонов? Учитывая, что шаблон — это поставленная задача и описанное решение, ответ, казалось бы, очевиден. Шаблоны помогают решать распространенные задачи. Но, конечно, шаблон — это нечто большее.

Шаблоны определяют задачи

Сколько раз вы доходили до какого-то этапа в проекте и обнаруживали, что дальше идти некуда? Вполне вероятно, вам нужно вернуться немного назад, прежде чем начать снова.

Определяя распространенные задачи, шаблоны помогают улучшить проект. И иногда первый шаг к решению — это осознание того, что есть проблема.

Шаблоны определяют решения

Определив и осознав проблему (и убедившись, что это именно та проблема), с помощью шаблона вы получаете доступ к решению, а также к анализу результатов его использования. Хотя шаблон не избавляет вас от необходимости рассмотреть последствия выбранного решения, вы по крайней мере будете уверены, что используете проверенный метод.

Шаблоны не зависят от языка программирования

Шаблоны определяют объекты и решения с помощью "объектно-ориентированных" терминов. Это означает, что многие шаблоны одинаково применимы ко многим языкам программирования. Начав использовать шаблоны, я изучал примеры кода на C++ и Smalltalk и писал 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 шаблоны помогают при переходе от одного языка к другому. Точно так же приложение, построенное на четких принципах объектно-ориентированного проектирования, легко перенести с одного языка на другой (хотя при этом всегда остаются проблемы, которые нужно решать).

Шаблоны определяют словарь

Предоставляя разработчикам имена методик, шаблоны обогащают процесс общения и передачи информации. Давайте 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.

Я: Я собираюсь использовать шаблон Composite.

Боб: Мне кажется, ты не продумал это как следует.

Что ж, Боб не согласен со мной. Он никогда со мной не согласен. Но он знает, о чем я говорю и почему моя идея плоха. А теперь проиграем эту сцену еще раз без использования словаря.

Я: Я собираюсь использовать три объекта, в которых используется один и тот же тип данных. В интерфейсе типа будут определены методы для добавления дочерних объектов собственного типа. Таким образом, мы можем построить сложную комбинацию реализованных объектов во время выполнения программы.

Боб: Да?

Шаблоны или методики, которые в них описаны, имеют тенденцию к взаимодействию. Так, шаблон Composite взаимодействует с шаблоном Visitor.

Я: А затем можно использовать шаблон Visitor для получения итоговых данных.

Боб: Ты упустил суть.

Не будем обращать внимание на Боба. Я не буду долго и мучительно описывать версию этого разговора без использования терминов шаблонов. В главе 10 я расскажу о шаблоне Composite, а в главе 11 — о шаблоне Visitor.

Суть в том, что эти методики можно использовать и без языка шаблонов. Сами методики всегда появляются *до* того, как им будет назначено имя и они будут организованы в виде шаблона. Если шаблона нет, его могут разработать. А любой инструмент, который используется достаточно часто, в конце концов получит имя.

Шаблоны проверяются и тестируются

Итак, если с помощью шаблонов, по сути, описываются лучшие методики решения задачи, то будет ли выбор имени самым важным элементом при создании каталогов шаблонов? В некотором смысле это так. Шаблоны представляют собой лучшие методики в сфере объектно-ориентированного программирования. Некоторым очень опытным программистам это может показаться упражнением по перекомпоновке очевидных вещей. Но для всех остальных шаблоны — это доступ к задачам и решениям, которые в противном случае приходилось бы находить нелегким путем.

Шаблоны делают проект доступным. Каталоги шаблонов появляются для все большего количества специализированных областей, поэтому даже очень опытный специалист может извлечь из них пользу. Например, программист графического пользовательского интерфейса (GUI) может быстро получить доступ к задачам и решениям при создании корпоративного приложения. Веб-программист сможет быстро наметить стратегию того, как избежать ошибок и подводных камней, которые могут возникнуть в проектах для PDA и смартфонов.

Шаблоны предназначены для совместной работы

По своей природе шаблоны должны быть наиболее общими и компонуемыми. Это означает, что у вас должна быть возможность применить один шаблон и тем самым создавать условия, подходящие для применения другого. Иначе говоря, используя шаблон, вы можете обнаружить, что для вас открываются другие двери.

Каталоги шаблонов обычно разрабатываются с расчетом на такого рода совместную работу, и возможность компоновки шаблонов всегда документируется в самом шаблоне.

Шаблоны способствуют хорошим проектам

В проектных шаблонах демонстрируются и применяются принципы объектно-ориентированного проектирования. Поэтому изучение проектных шаблонов может дать больше, чем конкретное решение в некотором контексте. В результате у вас может появиться новое видение того, как можно объединять объекты и классы для достижения поставленной цели.

Шаблоны используются в популярных каркасах

В этой книге процесс проектирования описан практически "с нуля". Принципы и проектные шаблоны, рассмотренные в ней, должны побудить вас создать ряд собственных базовых каркасов, которые лягут в основу всех ваших проектов. Хотя, как известно, лень — двигатель прогресса. Поэтому вы вполне можете использовать такие стандартные каркасы, как Zend, Code Igniter или Symfony, либо скопировать код из готовых проектов. Хорошее понимание основ проектных шаблонов поможет вам быстро освоить API этих каркасов.

PHP и проектные шаблоны

В этой главе мало что относится конкретно к PHP, что в некоторой степени характерно для данной темы. Большинство шаблонов применимо ко многим языкам программирования, в которых есть возможности работы с объектами, достаточно только решить некоторые вопросы реализации (если они вообще возникают).

Но, конечно, это не всегда так. Некоторые шаблоны корпоративных приложений прекрасно используются в тех языках, в которых прикладной процесс не останавливает свою работу в перерывах между запросами к серверу. PHP работает иначе. Для обработки каждого поступившего запроса сценарий запускается заново. Это означает, что с некоторыми шаблонами нужно обращаться более аккуратно. 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. Это не значит, что нельзя использовать данный шаблон; я применял его в своей практике с очень хорошими результатами. Просто при обсуждении шаблона мы должны обязательно принять во внимание вопросы, связанные с PHP. PHP формирует контекст для всех шаблонов, которые изучаются в данной книге.

Выше в данном разделе я уже упоминал о языках программирования, в которых есть возможность работы с объектами. В PHP можно программировать, вообще не определяя никаких классов (хотя, если вы пользуетесь любой библиотекой или каркасом, созданным сторонними разработчиками, вероятно, в некоторой степени вы столкнетесь с объектами). Несмотря на то что в данной книге мы почти полностью концентрируемся на объектно-ориентированных решениях задач программирования, не считайте это залпом из всех орудий в войне между приверженцами разных стилей программирования. Шаблоны и PHP могут создать мощный союз, поэтому они и составляют основу данной книги. Тем не менее они могут прекрасно сосуществовать с другими, более традиционными, подходами. Убедительное доказательство тому — PEAR. В пакетах PEAR очень изящно используются проектные шабло-

ны. Они склонны быть объектно-ориентированными по своей природе. И это делает их более, а не менее, полезными в процедурных проектах. Поскольку пакеты PEAR автономны и их сложность скрыта за четко описанным интерфейсом, их легко включить в проект любого типа.

Резюме

В этой главе мы познакомились с проектными шаблонами, рассмотрели их структуру (с помощью формата "Банды четырех") и изучили несколько причин, по которым у вас может возникнуть необходимость использовать проектные шаблоны в своих сценариях.

Важно помнить, что проектные шаблоны — это не набор готовых решений, которые можно объединять, как компоненты, при построении проекта. Шаблоны — это предлагаемые подходы к решению распространенных задач. В этих решениях воплощены некоторые основные принципы проектирования. Именно их мы и будем изучать в следующей главе.

Глава 8

Некоторые принципы шаблонов

Хотя проектные шаблоны просто описывают решения задач, у них есть тенденция делать акцент на решениях, которые способствуют повторному использованию, гибкости и универсальности. Для достижения этого в шаблонах реализуется ряд основных принципов объектно-ориентированного проектирования. С некоторыми из них мы познакомимся в данной главе, а более подробно будем изучать на протяжении остальной части книги.

В этой главе мы рассмотрим следующие принципы.

- *Композиция:* использование агрегирования объектов для достижения большей гибкости по сравнению с применением одного наследования.
- *Разделение:* как уменьшить зависимость между элементами в системе.
- *Сила интерфейса:* шаблоны и полиморфизм.
- *Категории шаблонов:* типы шаблонов, которые будут описываться в данной книге.

Открытие шаблонов

Я начал работать с объектами в языке Java. Как и можно было ожидать, прошло некоторое время, прежде чем кое-какие идеи пришлись кстати и сработали. Причем произошло это очень быстро, почти как откровение. Изящество принципов наследования и инкапсуляции просто поразило меня. Я почувствовал, что это другой способ определения и построения систем. Я *использовал* полиморфизм, работая с типом и изменяя реализации во время выполнения программы. Мне казалось, что эти знания и понимание помогут решить большинство моих проектных задач и позволят создавать красивые и элегантные системы.

В то время все книги на моем рабочем столе были посвящены возможностям языков программирования и многочисленным API, доступным программисту на Java. Если не считать краткого определения полиморфизма, в этих книгах почти не было попыток изучить стратегии проектирования.

Одни только возможности языка не порождают объектно-ориентированных проектов. Хотя мои проекты удовлетворяли их 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 широкими и глубокими, поскольку я пытался создавать новые классы по каждому поводу. Структура моих систем затрудняла передачу сообщений от одного уровня другому таким образом, чтобы не давать промежуточным классам слишком много информации об их окружении, не привязывать их к приложению и не делать их непригодными в новом контексте.

И только открыв для себя книгу *Design Patterns*[1], которую еще называют книгой "Банды четырех" (Gang of Four), я понял, что упустил целый аспект проектирования. К тому времени я уже самостоятельно открыл несколько основных шаблонов, но знакомство с другими шаблонами изменило мой способ мышления.

Я выяснил, что в своих проектах дал наследованию слишком много привилегий, пытаясь добавить классам слишком много функций. Но где еще нужна функциональность в объектно-ориентированной системе?

Я нашел ответ 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 с целью достижения уровня гибкости, который невозможен при применении только одного дерева наследования.

Композиция и наследование

Наследование (inheritance) — это эффективный способ описания меняющихся обстоятельств или контекста. Но за счет этого можно потерять в гибкости, особенно если на классы возложено несколько функциональных обязанностей.

Проблема

Как вы знаете, дочерние классы наследуют методы и свойства родительских классов (если они относятся к открытым или защищенным элементам). Мы будем использовать этот факт для создания дочерних классов, обладающих особой функциональностью.

На рис. 8.1 приведен простой пример с использованием диаграммы UML.

Абстрактный класс `Lesson` на рис. 8.1 моделирует занятие в колледже. Он определяет абстрактные методы `cost()` и `chargeType()`. На диаграмме показаны два

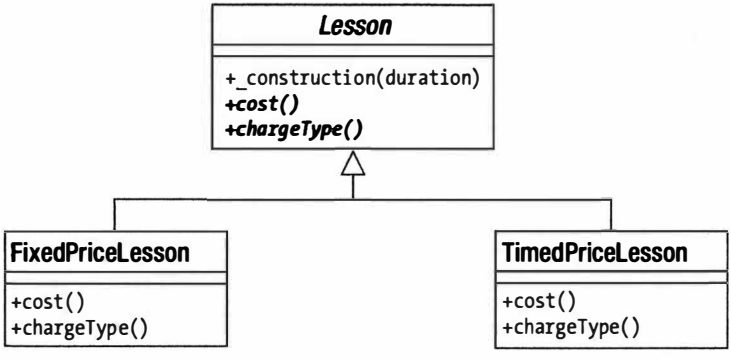


Рис. 8.1. Родительский класс и два дочерних

[1] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* (пер. с англ., изд. "Питер", 2007).

реализующих их класса, FixedPriceLesson и TimedPriceLesson, которые обеспечивают разные механизмы оплаты занятий.

С помощью этой схемы наследования я могу легко изменить реализацию занятия. Клиентский код будет знать только, что он имеет дело с объектом типа Lesson, поэтому детали механизма оплаты будут прозрачными.

Но что произойдет, если нужно будет ввести новый набор специализаций? Предположим, нам нужно работать с такими элементами, как лекции и семинары. Поскольку они подразумевают разные способы регистрации учащихся и создания рабочих материалов к занятиям, для них нужны отдельные классы. Поэтому теперь у нас есть две движущие силы проекта: нам нужно работать со стратегиями оплаты и разделить лекции и семинары.

На рис. 8.2 показано решение проблемы "в лоб".

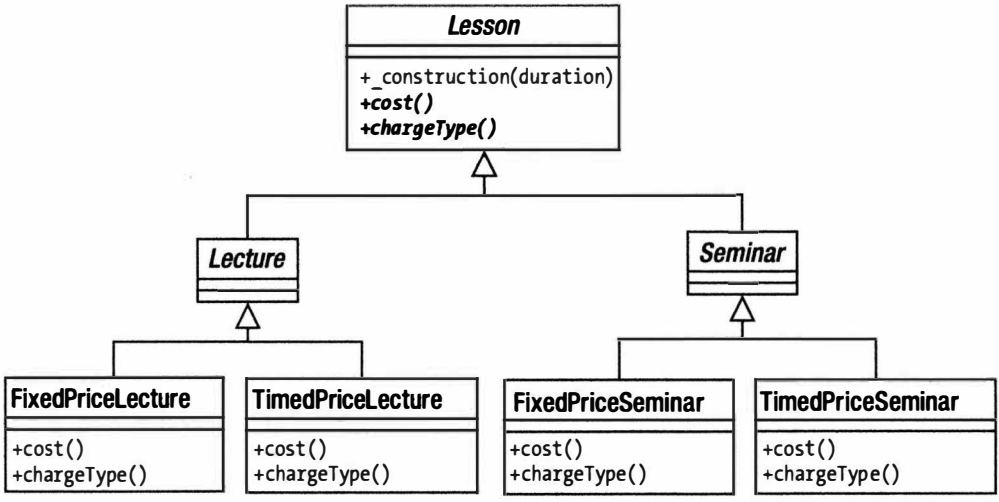


Рис. 8.2. Неудачная структура наследования

На рис. 8.2 показана явно неудачная иерархия. Мы не сможем в дальнейшем использовать это дерево наследования, чтобы управлять механизмами оплаты, не дублируя большие блоки функциональности. Эти стратегии оплаты повторяются в семействах классов Lecture и Seminar.

На данном этапе мы должны рассмотреть использование условных операторов в суперклассе Lesson, чтобы избавиться от дублирования. В сущности, мы удаляем логику оплаты из дерева наследования вообще, перемещая ее вверх, в суперкласс. Это полная противоположность традиционному рефакторингу, когда условные операторы заменяются полиморфизмом. Вот как выглядит исправленный класс Lesson.

```
abstract class Lesson {
   protected $duration;
   const     FIXED = 1;
   const     TIMED = 2;
   private   $costtype;

   function __construct( $duration, $costtype=1 ) {
      $this->duration = $duration;
      $this->costtype = $costtype;
   }
```

```
   function cost() {
      switch ( $this->costtype ) {
         CASE self::TIMED :
            return (5 * $this->duration);
            break;
         CASE self::FIXED :
            return 30;
            break;
        default:
            $this->costtype = self::FIXED;
            return 30;
      }
   }

   function chargeType() {
      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 lesson...
}

class Lecture extends Lesson {
   // Специфичные для Lecture реализации...
}

class Seminar extends Lesson {
   // Специфичные для Seminar реализации...
}
```

А вот как я должен работать с этими классами.

```
$lecture = new Lecture( 5, Lesson::FIXED );
print "{$lecture->cost()} ({$lecture->chargeType()})\n";

$seminar= new Seminar( 3, Lesson::TIMED );
print "{$seminar->cost()} ({$seminar->chargeType()})\n";
```

На выходе получим следующее.

```
30 (Фиксированная ставка)
15 (Почасовая оплата)
```

Диаграмма нового класса показана на рис. 8.3.

Я сделал структуру класса намного более управляемой, но дорогой ценой. Использование условных операторов в данном коде — это шаг назад. Обычно мы стараемся заменить условный оператор полиморфизмом. А здесь я сделал противопо-

ложное. Как видите, это вынудило меня продублировать условный оператор в методах `chargeType()` и `cost()`.

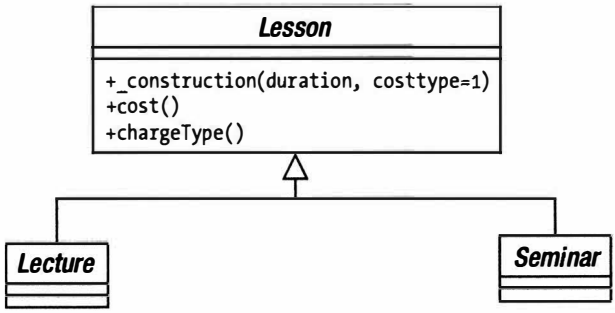


Рис. 8.3. Иерархия наследования улучшена в результате удаления расчетов стоимости занятий из подклассов

Похоже, я обречен на дублирование кода.

Использование композиции

Для решения данной проблемы я могу воспользоваться шаблоном Strategy. Этот шаблон используется для перемещения набора алгоритмов в отдельный тип. Перемещая код для вычисления стоимости, я могу упростить тип `Lesson` (рис. 8.4).

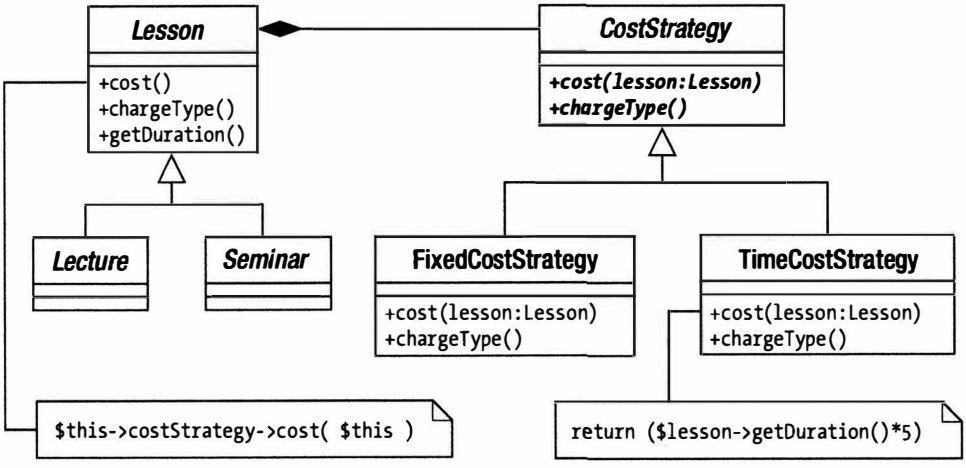


Рис. 8.4. Перемещение алгоритмов в отдельный тип

Я создал еще один абстрактный класс `CostStrategy`, в котором определены абстрактные методы `cost()` и `chargeType()`. Методу `cost()` нужно передать экземпляр класса `Lesson`, который он будет использовать для расчета стоимости занятия. Мы обеспечиваем две реализации класса `CostStrategy`. Объекты `Lesson` работают только с типом `CostStrategy`, а не с конкретной реализацией, поэтому мы в любое время можем добавить новые алгоритмы расчета стоимости, создавая подклассы на основе `CostStrategy`. При этом не понадобится вносить вообще никаких изменений в классы `Lesson`.

Приведем упрощенную версию нового класса `Lesson`, показанного на рис. 8.4.

```
abstract class Lesson {
   private       $duration;
   private   $costStrategy;

   function __construct( $duration, CostStrategy $strategy ) {
      $this->duration     = $duration;
      $this->costStrategy = $strategy;
   }

   function cost() {
      return $this->costStrategy->cost( $this );
   }

   function chargeType() {
      return $this->costStrategy->chargeType();
   }

   function getDuration() {
      return $this->duration;
   }
// Другие методы класса lesson...
}

class Lecture extends Lesson {
   // Специфичные для Lecture реализации...
}

class Seminar extends Lesson {
   // Специфичные для Seminar реализации...
}
```

Конструктору класса `Lesson` передается объект типа `CostStrategy`, который он сохраняет в виде свойства. Метод `Lesson::cost()` просто вызывает `CostStrategy::cost()`. Точно так же `Lesson::chargeType()` вызывает `CostStrategy::chargeType()`. Такой явный вызов метода другого объекта для выполнения запроса называется делегированием. В нашем примере объект типа `CostStrategy` — делегат класса `Lesson`. Класс `Lesson` снимает с себя ответственность за расчет стоимости занятия и возлагает эту задачу на реализацию класса `CostStrategy`. Вот как осуществляется делегирование.

```
function cost() {
   return $this->costStrategy->cost( $this );
}
```

Ниже приведено определение класса `CostStrategy` вместе с реализующими его дочерними классами.

```
abstract class CostStrategy {
   abstract function cost( Lesson $lesson );
   abstract function chargeType();
}

class TimedCostStrategy extends CostStrategy {
   function cost( Lesson $lesson ) {
      return ( $lesson->getDuration() * 5 );
```

```
    }

    function chargeType() {
        return "Почасовая оплата";
    }
}

class FixedCostStrategy extends CostStrategy {
    function cost( Lesson $lesson ) {
        return 30;
    }

    function chargeType() {
        return "Фиксированная ставка";
    }
}
```

Во время выполнения программы я легко могу изменить способ расчета стоимости занятий, выполняемый любым объектом типа `Lesson`, передав ему другой объект типа `CostStrategy`. Этот подход способствует созданию очень гибкого кода. Вместо того чтобы статично встраивать функциональность в структуры кода, я могу комбинировать объекты и менять их сочетания динамически.

```
$lessons[] = new Seminar( 4, new TimedCostStrategy() );
$lessons[] = new Lecture( 4, new FixedCostStrategy() );

foreach ( $lessons as $lesson ) {
    print "Плата за занятие {$lesson->cost()}. ";
    print "Тип оплаты: {$lesson->chargeType()}\n";
}
```

В результате на выходе получим следующее.

```
Плата за занятие 20. Тип оплаты: Почасовая оплата
Плата за занятие 30. Тип оплаты: Фиксированная ставка
```

Как видите, одно из следствий принятия этой структуры состоит в том, что мы рассредоточили обязанности наших классов. Объекты `CostStrategy` ответственны только за расчет стоимости занятия, а объекты `Lesson` управляют данными занятия.

Итак, композиция позволяет сделать код намного более гибким, поскольку можно комбинировать объекты и решать задачи динамически намного большим количеством способом, чем при использовании одной лишь иерархии наследования. Однако при этом могут возникнуть проблемы с читабельностью кода. В результате композиции, как правило, создается больше типов, причем с отношениями, которые не настолько предсказуемы, как в отношениях наследования. Поэтому понять отношения в такой системе немного труднее.

Разделение

Из главы 6, "Объекты и методология проектировани", мы узнали, что имеет смысл создавать независимые компоненты, поскольку систему, состоящую из зависимых классов, как правило, гораздо труднее сопровождать. Дело в том, что внесение изменения 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, и тесная связь — враг этой цели. Тесная связь имеет место, когда изменение в одном компоненте системы ведет к необходимости вносить множество изменений повсюду. Вы должны стремиться создавать независимые компоненты, чтобы можно было вносить изменения, не опасаясь "эффекта домино" непредвиденных последствий. Когда вы меняете компонент, степень его независимости влияет на вероятность того, что эти изменения вызовут ошибки в других частях системы.

На рис. 8.2 мы видели пример тесной связи. Поскольку логика схем оплаты повторяется в типах `Lecture` и `Seminar`, изменения в компоненте `TimedPriceLecture` приведут к необходимости внесения параллельных изменений в ту же логику в компоненте `TimedPriceSeminar`. Обновляя один класс и не обновляя другой, я нарушаю работу системы. При этом от интерпретатора PHP я не получу никакого предупреждения. Мое первое решение, с использованием условного оператора, породило аналогичную зависимость между методами `cost()` и `chargeType()`.

Применяя шаблон Strategy, я преобразовал алгоритмы оплаты в тип `CostStrategy`, разместил их за общим интерфейсом и реализовал каждый из них только один раз.

Тесная связь другого рода может иметь место, когда много классов в системе внедрены явным образом в платформу или среду. Предположим, вы создаете систему, которая, например, работает с базой данных MySQL. Для запросов к серверу базы данных вы можете использовать метод `mysqli::query()`.

Но если вам понадобится развернуть систему на сервере, который не поддерживает MySQL, вы *должны* будете внести изменения в весь проект, чтобы использовать SQLite. При этом вы будете вынуждены вносить изменения по всему коду, и вас ожидает перспектива поддерживать две параллельные версии приложения.

И проблема здесь не в зависимости системы от внешней платформы. Такая зависимость неизбежна, поскольку мы должны работать с кодом, который связывается с базой данных. Проблема возникает, когда такой код разбросан по всему проекту. Коммуникации с базами данных — это не главная обязанность большинства классов в системе, поэтому наилучшая стратегия — выделить такой код и сгруппировать его за общим интерфейсом. Таким образом вы будете способствовать независимости классов. В то же время, собирая код шлюза в одном месте, вы создаете условия для более легкого перехода на новую платформу, при котором не потребуется вносить изменения в остальную часть системы. Этот процесс — сокрытие реализации за ясным интерфейсом — называется *инкапсуляцией*.

В PEAR эта проблема решена с помощью пакета `PEAR::MDB2` (который пришел на смену пакету `PEAR::DB`). Это обеспечивает одну точку доступа для разных систем баз данных. А недавно, благодаря встроенному расширению PDO, эта модель была включена в сам язык PHP.

В классе `MDB2` существует статический метод `connect()`, которому передается строка, содержащая имя *источника данных* (DSN). В зависимости от состава этой строки, метод возвращает экземпляр объекта, реализующего класс `MDB2_Driver_Common`. Поэтому для строки `"mysql://"` метод `connect()` возвращает объект типа `MDB2_Driver_mysql`, в то время как для строки, которая начинается с `"sqlite://"`, он возвращает объект типа `MDB2_Driver_sqlite`. Структура этих классов показана на рис. 8.5.

Кроме того, пакет `PEAR::MDB2` позволяет отделить код приложения от специфики платформы базы данных. При условии, что вы используете совместимый SQL-код, ваше приложение будет работать со многими СУБД, включая MySQL, SQLite, MSSQL и др. При этом вам не нужно будет вносить изменения в свой код, кроме,

конечно, настройки DSN. Пожалуй, это единственное место в программе, где необходимо сконфигурировать контекст базы данных. На самом деле пакет `PEAR::MDB2` до некоторой степени помогает также работать с различными "диалектами" SQL — и это одна из причин, по которым вы можете решить использовать его, несмотря на скорость и удобство PDO.

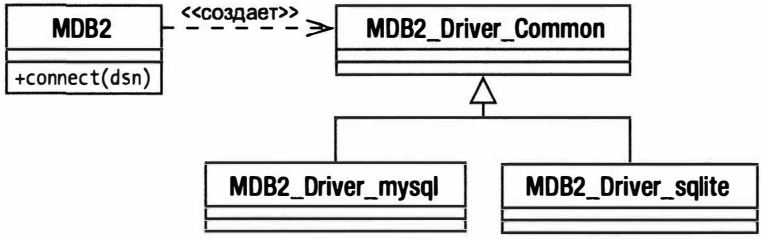


Рис. 8.5. В пакете `PEAR::MDB2` клиентский код отделен от объектов базы данных

Структура, показанная на рис. 8.5, имеет некоторое сходство с шаблоном Abstract Factory, который был описан в книге "Банды четырех" и будет также рассмотрен далее в этой книге. Более простой по своей природе, он, тем не менее, имеет такое же назначение: генерировать объект, в котором реализован абстрактный интерфейс, и не требовать от клиента непосредственного создания экземпляра объекта.

Несмотря на то что в пакете `MDB2` или в расширении PDO клиентский код отделен от специфики реализации платформы СУБД, свою часть работы вы все равно должны сделать. Если ваш (теперь уже универсальный) SQL-код рассредоточен по всему проекту, вы вскоре обнаружите, что единственное изменение какого-либо аспекта проекта может повлечь за собой каскад изменений во многих местах кода. И здесь самым типичным примером было бы изменение структуры базы данных, при котором добавление дополнительного столбца в таблице может повлечь за собой изменение SQL-кода многих повторяющихся запросов к базе данных. Поэтому вам следует подумать о том, чтобы извлечь этот код и поместить его в один пакет, тем самым отделив логику приложения от специфики реляционной базы данных.

Ослабление связи

Чтобы сделать код связи с базой данных гибким и управляемым, вы должны отделить логику приложения от специфики платформы системы управления базами данных. От такого разделения компонентов вы должны получить кучу преимуществ в своих проектах.

В качестве примера представьте себе, что в нашу систему автоматизации учебного процесса нужно включить регистрационный компонент, в задачи которого входит добавление к системе новых занятий. Процедура регистрации должна предусматривать рассылку уведомлений администратору после добавления нового занятия. При этом пользователи вашей системы никак не могут решить, в каком виде эти уведомления должны рассылаться — по электронной почте или в виде коротких текстовых сообщений (SMS). По сути, они так любят спорить, что вы вполне можете ожидать от них изменения формы коммуникаций в недалеком 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 программы.

Если вы в коде будете использовать явные ссылки на класс `Mailer` или `Texter`, то ваша система станет наглухо привязанной к конкретному типу рассылки уведомле-

ний. Это примерно то же самое, что привязаться к конкретному типу СУБД, используя в коде вызовы ее специализированных функций API.

Ниже приведен фрагмент кода, в котором детали реализации конкретной системы уведомления скрыты от кода, который ее использует.

```
class RegistrationMgr {

   function register( Lesson $lesson ) {
      // Что-то делаем с объектом типа Lesson
 . . .
      // Рассылаем уведомления
      $notifier = Notifier::getNotifier();
      $notifier->inform( "Новое занятие: стоимость - ({$lesson->cost()})" );
   }
}

abstract class Notifier {

   static function getNotifier() {
      // Создадим конкретный класс согласно
      // файлу конфигурации системы или другой логики

      if ( rand(1,2) === 1 ) {
         return new MailNotifier();
      } else {
         return new TextNotifier();
      }
   }
   abstract function inform( $message );
}

class MailNotifier extends Notifier {

   function inform( $message ) {
      print "Уведомление по e-MAIL: {$message}\n";
   }
}

class TextNotifier extends Notifier {

   function inform( $message ) {
      print "Текстовое уведомление: {$message}\n";
   }
}
```

Здесь я создал класс `RegistrationMgr`, который является простым клиентским классом для классов типа `Notifier`. Несмотря на то что класс `Notifier` 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 класса типа `Notifier` должен определяться каким-нибудь гибким способом, например параметром в файле конфигурации. Здесь же я немного сжульничал и выбрал тип объекта случайным образом. Метод `inform()` классов `MailNotifier` и `TextNotifier` практически ничего не делает. Он просто выводит информацию, полученную в качестве параметра, а также дополняет ее типом сообщения, чтобы было видно, к какому классу он относится.

Обратите внимание на то, что только в методе `Notifier::getNotifier()` сосредоточена информация о том, какой конкретно объект типа `Notifier` должен быть создан. В результате я могу посылать уведомления из сотен разных участков программы, а при изменении типа уведомлений мне потребуется внести изменения только в один метод в классе `Notifier`.

Ниже приведен пример кода, вызывающий метод `register()` класса `RegistrationMgr`.

```
$lessons1 = new Seminar( 4, new TimedCostStrategy() );
$lessons2 = new Lecture( 4, new FixedCostStrategy() );

$mgr = new RegistrationMgr();
$mgr->register( $lessons1 );
$mgr->register( $lessons2 );
```

Вот что будет выведено в результате.

```
Текстовое уведомление: Новое занятие: стоимость - (20)
Уведомление по e-MAIL: Новое занятие: стоимость - (30)
```

На рис. 8.6 приведена диаграмма классов нашего проекта.

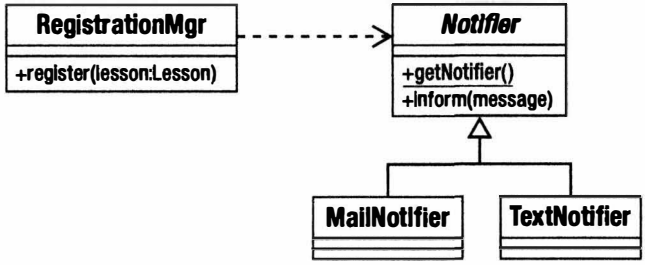


Рис. 8.6. В классе `Notifier` клиентский код отделен от кода реализации различных уведомителей

Обратите внимание на то, что структура, показанная на рис. 8.6, очень сильно напоминает структуру, которая используется в пакете `PEAR::MDB2` (см. рис. 8.5).

Программируйте на основе интерфейса, а не его реализации

Этот принцип — один из главных лейтмотивов данной книги, который пронизывает весь ее материал. Вы видели в главе 6 (и в предыдущем разделе), что можно скрыть различные реализации за общим интерфейсом, который определен в суперклассе. В результате появляется возможность оперировать в клиентском коде объектом, относящимся к суперклассу, а не к реализующему его классу, не заботясь при этом о его конкретной реализации.

"Параллельные" условные операторы, подобные тем, которые мы встроили в методы `Lesson::cost()` и `Lesson::chargeType()`, — это сигнал к тому, что полиморфизм необходим. Условные операторы усложняют сопровождение кода, потому что изменение в одном условном выражении ведет к необходимости изменения во всех его "двойниках". Об условных операторах иногда говорят, что они реализуют "сымитированное наследование".

Размещая алгоритмы оплаты в отдельных классах, в которых реализованы методы класса `CostStrategy`, мы избавляемся от дублирования. Мы также намного упрощаем процесс добавления новых стратегий оплаты, если возникнет такая необходимость.

С точки зрения клиентского кода иногда полезно потребовать, чтобы в параметрах метода задавались абстрактные или общие типы данных. Требуя более специфические типы, можно ограничить гибкость кода во время выполнения программы.

Но, конечно, уровень универсальности, который вы выбираете с помощью уточнений аргументов, — это спорный вопрос. Если выбор будет слишком "общим", то метод может стать менее безопасным. Если вы требуете специфическую функциональность от подтипа, то передача методу совершенно другого родственного элемента может быть небезопасной.

Но если сделать выбор уточнений аргументов слишком ограниченным, вы потеряете преимущества полиморфизма. Посмотрите на этот модифицированный фрагмент из класса `Lesson`.

```
function __construct( $duration, FixedPriceStrategy $strategy ) {
   $this->duration     = $duration;
   $this->costStrategy = $strategy;
}
```

Проектное решение, приведенное в данном примере, ставит два вопроса. Во-первых, объект `Lesson` теперь привязан к определенной стратегии стоимости, в результате чего мы больше не можем формировать динамические компоненты. Во-вторых, явная ссылка на класс `FixedPriceStrategy` заставляет нас поддерживать эту конкретную реализацию.

Требуя общий интерфейс, вы можете комбинировать объект `Lesson` с любой реализацией объекта `CostStrategy`.

```
function __construct( $duration, CostStrategy $strategy ) {
   $this->duration     = $duration;
   $this->costStrategy = $strategy;
}
```

Другими словами, мы отделили класс `Lesson` от специфики расчетов стоимости занятия. Самое главное здесь — это интерфейс и гарантия, что предоставленный объект будет ему соответствовать.

Конечно, кодирование на основе интерфейса во многих случаях может просто отложить вопрос о том, как создавать экземпляры объектов. Когда мы говорим, что во время выполнения программы объект `Lesson` можно комбинировать с любым интерфейсом `CostStrategy`, мы уклоняемся от вопроса "Откуда возьмется объект `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 формируют отдельную категорию в каталоге шаблонов "Банды четырех", и мы подробнее изучим ее в следующей главе.

Меняющаяся концепция

После того как проектное решение создано, его легко интерпретировать. Но как узнать, с чего начать?

"Банда четырех" рекомендует "инкапсулировать меняющуюся концепцию". Для нашего примера с классом `Lesson` меняющаяся концепция — это алгоритм оплаты.

Но это может быть не только вычисление стоимости для одной из двух возможных стратегий оплаты, как в нашем примере. Очевидно, данный пример можно расширить, используя специальные предложения (акции), тарифы для иностранных студентов, вступительные скидки, а также другие возможности.

Мы быстро определили, что создание подклассов в условиях постоянно изменяющейся ситуации неприемлемо, и обратились к условным операторам. Поместив в один класс все варианты оплаты, мы подчеркнули его пригодность для инкапсуляции.

"Банда четырех" рекомендует активно искать меняющиеся элементы в классах и оценить их пригодность для инкапсуляции в новом типе. Каждую альтернативу в рассматриваемом условном операторе можно извлечь и можно сформировать класс, расширяющий общий абстрактный родительский класс. Затем этот новый тип можно использовать в классах, из которых он был извлечен. В результате получим следующее:

- концентрация обязанностей;
- поддержка гибкости благодаря композиции;
- иерархии наследования будут более компактными и сосредоточенными;
- сокращение дублирования.

Но как определить вариацию? Один из признаков — злоупотребление наследованием, например использование наследования для различных ситуаций одновременно (лекция/семинар, фиксированная/повременная оплата), или создание подклассов на основе алгоритма, который является второстепенным по отношению к основной обязанности типа. Другой признак вариации, пригодной для инкапсуляции, — это, как было показано выше, условное выражение.

Проблемы применения шаблонов

Одна из задач, для которых не существует шаблонов, заключается в необязательном или неподходящем использовании шаблонов. Такое положение вещей создало шаблонам плохую репутацию в некоторых кругах. Поскольку решения на основе шаблонов изящны и ясны, возникает искушение применять их везде, где только можно, независимо от того, есть ли в этом реальная необходимость.

Методология экстремального программирования (eXtreme Programming — XP) предлагает ряд принципов, которые применимы в данной ситуации. Первый принцип — "Вам необязательно это нужно" (обычно для него используют аббревиатуру YAGNI от *You aren't going to need it*). Как правило, данный принцип применяется для функций приложения, но он имеет смысл и для шаблонов.

Когда я создаю большие проекты на PHP, то обычно разбиваю приложение на уровни, отделяя логику приложения от представления данных и уровня сохранения данных. Я использую всевозможные виды основных и промышленных шаблонов, а также их комбинации.

Но если меня попросят создать простую форму для обратной связи небольшого веб-сайта, я могу запросто использовать процедурный код, поместив его на одну страницу с HTML-кодом. В этом случае мне не нужна гибкость в огромных масштабах, потому что я не буду что-либо строить на этой первоначальной основе. Мне не нужно использовать шаблоны, которые решают соответствующие задачи в более широких системах. Поэтому я применяю второй принцип экстремального программирования: "Сделайте самый простой работающий вариант".

Работая с каталогом шаблонов, думайте о структуре и процессе решения, обращая 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 я рассмотрю несколько основных шаблонов, которые используются в настоящее время, предоставив PHP-реализации и обсудив их в широком контексте PHP-программирования.

Описанные шаблоны будут взяты из основных каталогов, включая книги Мартина Фаулера (Martin Fowler) *Design Patterns* и *Patterns of Enterprise Application Architecture*[2] (Addison-Wesley, 2003) и Дипака Алура (Deepak Alur) *Core J2EE Patterns*[3] (Prentice Hall PTR, 2001).

В качестве отправной точки я использую деление на категории, принятое "Бандой четырех", которое распределяет шаблоны следующим образом.

Шаблоны для генерации объектов

Эти шаблоны предназначены для создания экземпляров объектов. Это важная категория, если учитывать принцип кодирования на основе интерфейса. Если в нашем 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 мы объединяем 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. Взятые в основном из книг *Patterns of Enterprise Application Architecture* и *Core J2EE Patterns*, эти шаблоны имеют дело с представлением данных и логикой приложения.

Шаблоны баз данных

Обзор шаблонов, которые помогают сохранять и извлекать данные из баз данных и устанавливать соответствие между объектами базы данных и приложения.

---

[2] Мартин Фаулер. *Шаблоны корпоративных приложений* (пер. с англ., ИД "Вильямс", 2009).

[3] Дипак Алур, Джон Крупи, Дэн Малкс. *Образцы J2EE. Лучшие решения и стратегии проектирования.* (пер. с англ., изд. "Лори", 2013).

Резюме

В этой главе мы рассмотрели некоторые принципы, лежащие в основе многих проектных шаблонов. Мы обсудили использование композиции, дающей возможность комбинировать и рекомбинировать объекты во время выполнения. Это позволяет создавать более гибкие структуры, чем те, в которых используется только наследование. Вы познакомились с разделением — практикой выделения компонентов программного обеспечения из их контекста, чтобы их можно было применять более широко. Мы проанализировали важность интерфейса как средства отделения клиентов от деталей реализации.

В последующих главах мы подробнее изучим некоторые проектные шаблоны.

Глава 9

Генерация объектов

Создание объектов — это рутинная работа. Поэтому во многих объектно-ориентированных проектах используются изящные и четко определенные абстрактные классы. Это позволяет извлечь преимущество из впечатляющей гибкости, которую обеспечивает полиморфизм (переключение конкретных реализаций во время выполнения). Но, чтобы достичь этой гибкости, мы должны разработать стратегии генерации объектов. Вот этой темой мы сейчас и займемся.

Итак, в данной главе мы рассмотрим следующие темы.

- *Шаблон Singleton:* специальный класс, который генерирует один и только один экземпляр объекта.
- *Шаблон Factory Method:* построение иерархии наследования классов создателя.
- *Шаблон Abstract Factory:* создание групп функционально связанных продуктов.
- *Шаблон Prototype:* использование директивы `clone` для генерации объектов.

Генерация объектов: задачи и решения

Создание объекта — это слабое место в объектно-ориентированном проектировании. В предыдущей главе мы описали принцип "кодирования на основе интерфейса, а не реализации". Поэтому в наших классах мы будем работать с абстрактными супертипами. Это делает код более гибким, позволяя использовать объекты, экземпляры которых создаются на основе различных конкретных подклассов во время выполнения программы. Побочный эффект от такого подхода состоит в том, что создание экземпляров объектов отложено.

Вот класс, конструктору которого передается строка с именем сотрудника, для которого и создается экземпляр конкретного объекта.

```
abstract class Employee {
   protected $name;

   function __construct( $name ) {
      $this->name = $name;
   }

   abstract function fire();
```

```
}

class Minion extends Employee {
   function fire() {
      print "{$this->name}: убери со стола \n";
   }
}

class NastyBoss {
   private $employees = array();

   function addEmployee( $employeeName ) {
      $this->employees[] = new Minion( $employeeName );
   }

   function projectFails() {
      if ( count( $this->employees ) > 0 ) {
         $emp = array_pop( $this->employees );
         $emp->fire();
      }
   }
}

$boss = new NastyBoss();
$boss->addEmployee( "Игорь" );
$boss->addEmployee( "Владимир" );
$boss->addEmployee( "Мария" );
$boss->projectFails();
// Выводится:
// Мария: убери со стола
```

Как видите, мы определяем абстрактный базовый класс `Employee` и реализующий его класс `Minion`. Получив строку с именем, метод `NastyBoss::addEmployee()` создает экземпляр нового объекта типа `Minion`. Каждый раз, когда объект `NastyBoss` попадает в неприятную ситуацию (из-за вызова метода `NastyBoss::projectFails()`, т.е. провала проекта), он ищет подходящий объект типа `Minion`, чтобы уволить его.

Создавая экземпляр объекта `Minion` непосредственно в классе `NastyBoss`, мы ограничиваем гибкость последнего. Вот если бы объект `NastyBoss` мог работать с *любым* экземпляром объекта типа `Employee`, мы могли бы сделать код доступным для изменения прямо во время выполнения программы, просто добавив дополнительные специализации класса `Employee`. Полиморфизм, показанный на рис. 9.1, должен показаться вам знакомым.

Если класс `NastyBoss` не создаст экземпляр объекта `Minion`, то откуда он возьмется? Авторы кода обычно уклоняются от решения данной проблемы, ограничивая тип аргумента в объявлении метода, а затем с легкостью опуская демонстрацию создания экземпляров везде, за исключением тестового контекста.

```
class NastyBoss {
   private $employees = array();

   function addEmployee( Employee $employee ) {
      $this->employees[] = $employee;
   }

   function projectFails() {
```

```
        if ( count( $this->employees ) > 0 ) {
            $emp = array_pop( $this->employees );
            $emp->fire();
        }
    }
}

// Новый класс типа Employee...
class CluedUp extends Employee {
    function fire() {
        print "{$this->name}: вызови адвоката\n";
    }
}

$boss = new NastyBoss();
$boss->addEmployee( new Minion(  "Игорь"    ) );
$boss->addEmployee( new CluedUp( "Владимир" ) );
$boss->addEmployee( new Minion(  "Мария"    ) );
$boss->projectFails();
$boss->projectFails();
$boss->projectFails();

// Выводится:
// Мария: убери со стола
// Владимир: вызови адвоката
// Игорь: убери со стола
```

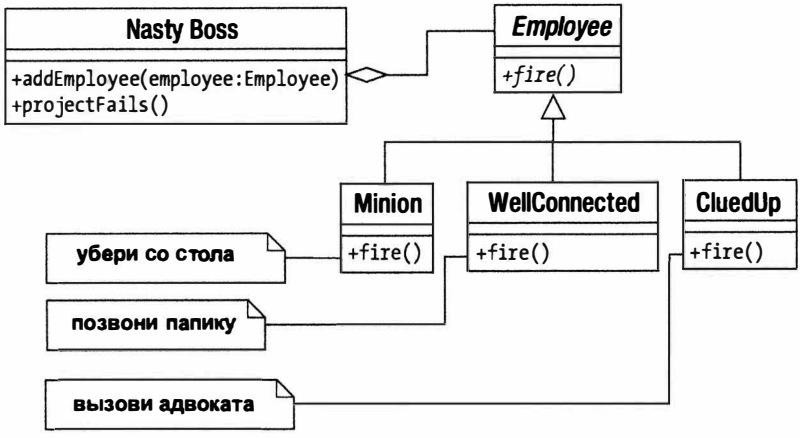


Рис. 9.1. Работа с абстрактным типом активизирует полиморфизм

Хотя эта версия класса `NastyBoss` работает с типом `Employee` и поэтому получает преимущества от полиморфизма, мы все еще не определили стратегию создания объекта. Создание экземпляров объектов — это скучная работа, но ее нужно сделать. В этой главе речь пойдет о классах и объектах, которые работают с конкретными классами, так что остальным классам этого делать не нужно.

И если здесь нужно сформулировать принцип, то такой: "делегировать создание 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 генерировать объекты типа `Employee`. Давайте добавим статический метод к классу `Employee`, в котором и реализована стратегия создания объекта.

```
abstract class Employee {
   protected $name;
   private static $types = array( 'Minion', 'CluedUp', 'WellConnected' );

   static function recruit( $name ) {
      $num = rand( 1, count( self::$types ) )-1;
      $class = self::$types[$num];
      return new $class( $name );
}

   function __construct( $name ) {
      $this->name = $name;
   }

   abstract function fire();
}
// Новый класс типа Employee...
class WellConnected extends Employee {
   function fire() {
      print "{$this->name}: позвони папику \n";
   }
}
```

Как видите, этому методу передается строка с именем сотрудника, которая используется для создания экземпляра конкретного подтипа `Employee`, выбранного случайным образом. Теперь мы можем делегировать детали создания экземпляра объекта методу `recruit()` класса `Employee`.

```
$boss = new NastyBoss();
$boss->addEmployee( Employee::recruit( "Игорь"    ) );
$boss->addEmployee( Employee::recruit( "Владимир" ) );
$boss->addEmployee( Employee::recruit( "Мария"    ) );
```

Простой пример подобного класса мы уже видели в главе 4. Если помните, тогда мы поместили статический метод под названием `getInstance()` в класс `ShopProduct`. Метод `getInstance()` отвечал за генерацию правильного подкласса `ShopProduct` на основании данных, полученных в результате запроса к базе данных. Поэтому класс `ShopProduct` выполняет двоякую роль. Он определяет тип `ShopProduct` и действует как "фабрика" по созданию конкретных объектов `ShopProduct`.

На заметку. В этой главе я часто использую термин "фабрика". Фабрика — это класс или метод, отвечающий за генерацию объектов.

```
// Класс ShopProduct

   public static function getInstance( $id, PDO $dbh ) {
      $query = "select * from products where id = ?";
      $stmt  = $dbh->prepare( $query );
      if (  ! $stmt->execute( array( $id ) ) ) {
         $error=$dbh->errorInfo();
         die( "Ошибка: " . $error[1] );
```

```
        }
        $row = $stmt->fetch( );
        if ( empty( $row ) ) { return null; }
        if ( $row['type'] == "book" ) {
            // Создадим объект типа BookProduct
            $product=new BookProduct()
        } else if ( $row['type'] == "cd" ) {
            // Создадим объект типа CDProduct
            $product=new CDProduct()
        } else {
            // Создадим объект типа ShopProduct
            $product=new ShopProduct()
        }
        $product->setId( $row['id'] );
        $product->setDiscount( $row['discount'] );
        return $product;
    }
```

В методе `getInstance()` используется большой условный оператор для определения того, экземпляры каких подклассов создавать. Подобные условные операторы распространены в коде фабрик. Обычно мы стараемся избавляться от больших условных операторов в проектах, но такие действия часто приводят к тому, что условные операторы смещаются к моменту генерации объектов. Как правило, это не представляет серьезной проблемы, потому что мы удаляем параллельные условные операторы из кода и смещаем момент принятия решения в данную точку.

Итак, в данной главе мы изучим некоторые шаблоны "Банды четырех" для генерации объектов.

Шаблон Singleton

Глобальная переменная — это один из самых больших источников проблем для программиста, использующего ООП. Причины этого к настоящему моменту уже должны быть вам понятны. Глобальные переменные привязывают классы к их контексту, подрывая основы инкапсуляции (подробнее об этом говорится в главах 6, "Объекты и методология проектирования", и 8, "Некоторые принципы шаблонов"). Если в классе используется 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.

Незащищенная природа глобальных переменных может стать причиной серьезных проблем, хотя их и удобно использовать. Как скоро вы начнете зависеть от глобальных переменных, это только вопрос времени, например, когда в одной из библиотек будет объявлена глобальная переменная, которая окажется в конфликте с другой глобальной переменной, объявленной где-то в другом месте. Мы уже видели, что PHP уязвим к конфликтам между именами классов, но это гораздо хуже. PHP не предупредит вас, когда произойдет конфликт глобальных переменных. Вы узнаете об этом только тогда, когда сценарий начнет вести себя не так, как обычно. А еще хуже, если вы вообще не заметите никаких проблем при разработке кода. Но, используя глобальные переменные, вы потенциально оставляете пользователей наедине с угрозой новых конфликтов, когда они попытаются использовать вашу библиотеку наряду с другими.

Но искушение использовать глобальные переменные все равно остается. Причина в том, что бывают случаи, когда недостатки глобальных переменных — это цена, которую стоит заплатить за то, чтобы предоставить всем классам доступ к объекту.

Как я говорил выше, избежать проблем подобного рода частично помогают пространства имен. По крайней мере они позволяют собрать переменные в отдельном пакете, а это значит, что библиотеки сторонних разработчиков уже не смогут конфликтовать с кодом вашей системы. Однако даже в таком случае существует риск возникновения конфликтов внутри самого пространства имен.

Проблема

Обычно в хорошо спроектированных системах экземпляры объектов передаются в виде параметров при вызове методов. При этом каждый класс сохраняет свою независимость от более широкого контекста и взаимодействует с другими частями системы через очевидные каналы коммуникации. Но иногда вы обнаружите, что должны использовать некоторые классы как каналы передачи информации для объектов, которые не имеют к ним отношения, создавая зависимости во имя хорошего стиля проектирования.

Рассмотрим в качестве примера класс `Preferences`, в котором хранятся данные, используемые в процессе выполнения приложения. Мы можем использовать объект `Preferences` для хранения таких параметров, как DSN-строки (т.е. строки, содержащие имена источников данных, в которых хранится информация о базе данных), URL сайта, пути к файлам и т.д. Очевидно, что подобная информация может меняться в зависимости от конкретной инсталляции. Данный объект может также использоваться как доска объявлений (или центр для сообщений), которые размещаются или извлекаются объектами системы, в других отношениях не связанными с ним.

Но передавать объект `Preferences` от одного объекта к другому — это не всегда хорошо. Многие классы, которые в других отношениях не используют этот объект, будут вынуждены принимать его просто для того, чтобы передать объектам, с которыми они работают. В результате получаем просто другой вид тесной связи.

Мы также должны быть уверены, что все объекты в нашей системе работают с одним и тем же объектом `Preferences`. Нам не нужна ситуация, когда одни объекты устанавливают значения в каком-то объекте, в то время как другие считывают данные с совершенно другого объекта.

Давайте выделим действующие факторы данной проблемы.

- Объект `Preferences` должен быть доступен для любого объекта в системе.
- Объект `Preferences` не должен сохраняться в глобальной переменной, значение которой может быть случайно затерто.
- В системе не должно быть больше одного объекта `Preferences`. Это означает, что *объект* Y устанавливает свойство в объекте `Preferences`, а *объект* Z извлекает то же самое значение этого свойства, причем они не связываются один с другим непосредственно (мы предполагаем, что оба объекта имеют доступ к объекту `Preferences`).

Реализация

Чтобы решить эту проблему, начнем с установления контроля над созданием экземпляров объектов. Мы создадим класс, экземпляр которого нельзя создать за его пределами. Может показаться, что это трудно сделать, но на самом деле это просто вопрос определения закрытого конструктора.

```
class Preferences {
   private $props = array();

   private function __construct() { }

   public function setProperty( $key, $val ) {
      $this->props[$key] = $val;
   }

   public function getProperty( $key ) {
      return $this->props[$key];
   }
}
```

Конечно, на данном этапе класс `Preferences` совершенно бесполезен. Мы довели ограничение доступа до уровня абсурда. Поскольку конструктор объявлен как `private`, никакой клиентский код не может создать на его основе экземпляр объекта. Поэтому методы `setProperty()` и `getProperty()` оказываются лишними.

Мы можем использовать статический метод и статическое свойство, чтобы создавать экземпляры объекта через посредника.

```
class Preferences {
   private $props = array();
   private static $instance;

   private function __construct() { }

   public static function getInstance() {
      if ( empty( self::$instance ) ) {
         self::$instance = new Preferences();
      }
      return self::$instance;
   }

   public function setProperty( $key, $val ) {
      $this->props[$key] = $val;
   }

   public function getProperty( $key ) {
      return $this->props[$key];
   }
}
```

Свойство `$instance` — закрытое и статическое, поэтому к нему нельзя получить доступ из-за пределов класса. Но у метода `getInstance()` есть доступ к нему. Поскольку метод `getInstance()` — общедоступный и статический, его можно вызвать через класс из какого-либо места сценария.

```
$pref = Preferences::getInstance();
$pref->setProperty( "name", "Иван" );

unset( $pref ); // Удалим ссылку

$pref2 = Preferences::getInstance();
// Убедимся, что ранее установленное значение сохранено
print $pref2->getProperty( "name" ) . "\n";
```

В результате будет выведено единственное значение параметра 'name', которое мы ранее добавили к объекту Preferences, воспользовавшись разными объектными переменными.

Иван

Статический метод не может получить доступ к свойствам объектов, потому что он по определению вызывается через класс, а не в контексте объекта. Но он может получить доступ к статическому свойству. Когда вызывается метод getInstance(), мы проверяем свойство Preferences::$instance. Если оно пусто, то создаем экземпляр класса Preferences и сохраняем его в свойстве. Затем мы возвращаем этот экземпляр вызывающему коду. Поскольку статический метод getInstance() — это часть класса Preferences, у нас нет проблем с созданием экземпляра объекта Preferences, даже несмотря на то что конструктор закрытый.

На рис. 9.2 показан шаблон Singleton в действии.

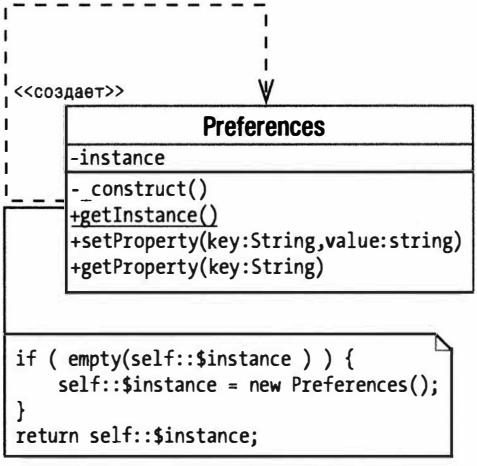


Рис. 9.2. Пример шаблона Singleton

Выводы

Итак, насколько хорош подход с использованием шаблона Singleton по сравнению с глобальными переменными? Начнем с плохого. И шаблоном Singleton, и глобальными переменными часто злоупотребляют. Поскольку доступ к объектам Singleton можно получить из любого места системы, они могут способствовать созданию зависимостей, которые затрудняют отладку приложения. А в случае изменения шаблона Singleton это повлияет на классы, которые его используют. Зависимости не представляют проблемы сами по себе. В конце концов, мы создаем зависимость каждый раз, когда объявляем, что методу требуется передать аргумент определенного типа. Проблема в том, что глобальная природа шаблона Singleton позволяет программисту обойти каналы коммуникации, определенные интерфейсами класса. Когда используется Singleton, зависимость скрыта внутри метода и не объявляется в его сигнатуре. Это затрудняет отслеживание связей внутри системы. Поэтому классы Singleton должны использоваться редко и очень осторожно.

Тем не менее я считаю, что умеренное использование шаблона Singleton может улучшить проект системы, избавив ее от излишнего загромождения при передаче ненужных объектов в системе.

Шаблоны Singleton — это шаг вперед по сравнению с использованием глобальных переменных в объектно-ориентированном контексте. Вы не сможете затереть объекты Singleton неправильными данными. Такой вид защиты особенно важен в версиях PHP, в которых нет поддержки пространства имен. Любой конфликт имен будет обнаружен на стадии компиляции, что приведет к завершению выполнения программы.

Шаблон Factory Method

В объектно-ориентированном проекте акцент делается на абстрактном классе, а не на его реализации, т.е. мы работаем с обобщениями, а не с частностями. Шаблон Factory Method решает проблему создания экземпляров объектов, когда в коде используются абстрактные типы. И в чем же состоит решение? Пусть созданием экземпляров объектов занимаются специальные классы.

Проблема

Давайте рассмотрим проект личного дневника-календаря. В числе прочих мы будем иметь дело с объектами `Appointment` (встреча). Наша бизнес-группа установила взаимоотношения с другой компанией, и мы должны передать им информацию о назначенной встрече с помощью формата, который называется "BloggsCal". Но нас предупредили, что со временем могут понадобиться и другие форматы.

Рассуждая только на уровне интерфейса, мы сразу можем определить двух участников. Нам нужен кодировщик данных, который преобразовывает объекты типа `Appointment` во внутренний формат BloggsCal. Давайте назовем этот класс `Appt Encoder`. Нам нужен управляющий класс, который будет выполнять поиск кодировщика и, возможно, работать с ним, чтобы осуществлять коммуникации с третьей стороной. Назовем этот класс `CommsManager`. Используя терминологию шаблона, можем сказать, что `CommsManager` — это создатель, а `ApptEncoder` — продукт. Эта структура показана на рис. 9.3.

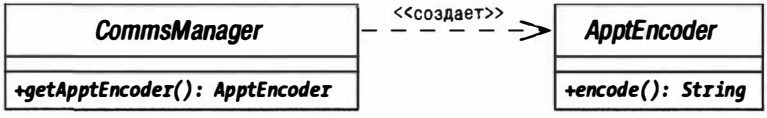


Рис. 9.3. Абстрактные классы — создатель и продукт

Но как нам получить реальный конкретный объект типа `ApptEncoder`?

Мы можем потребовать, чтобы объект `ApptEncoder` передавался `CommsManager`, но это просто откладывает решение проблемы, а мы хотим решить ее сейчас. Давайте создадим экземпляр объекта `BloggsApptEncoder` непосредственно внутри класса `CommsManager`.

```
abstract class ApptEncoder {
   abstract function encode();
}

class BloggsApptEncoder extends ApptEncoder {

   function encode() {
      return "Данные о встрече закодированы в формате BloggsCal \n";
   }
```

```
}

class MegaApptEncoder extends ApptEncoder {

   function encode() {
      return "Данные о встрече закодированы в формате MegaCal \n";
   }
}

class CommsManager {

   function getApptEncoder() {
      return new BloggsApptEncoder();
   }
}
```

Класс `CommsManager` отвечает за генерацию объектов `BloggsApptEncoder`. Теперь, когда нас попросят преобразовать нашу систему для работы с новым форматом MegaCal, мы просто добавим условный оператор в метод `CommsManager::getAppt Encoder()`. В конце концов, это стратегия, которую мы использовали в прошлом. Давайте создадим новую реализацию `CommsManager`, которая будет работать с обоими форматами: BloggsCal и MegaCal.

```
class CommsManager {
   const  BLOGGS = 1;
   const    MEGA = 2;
   private $mode = 1;

   function __construct( $mode ) {
      $this->mode = $mode;
   }

   function getApptEncoder() {
      switch ( $this->mode ) {
         case ( self::MEGA ):
            return new MegaApptEncoder();
         default:
            return new BloggsApptEncoder();
      }
   }
}
$comms = new CommsManager( CommsManager::MEGA );
$apptEncoder = $comms->getApptEncoder();
print $apptEncoder->encode();
```

В качестве флажков, определяющих два режима, в которых может работать сценарий, мы используем константы класса `MEGA` и `BLOGGS`. В методе `getApptEncoder()` используется оператор `switch`, чтобы протестировать свойство `$mode` и создать экземпляр соответствующей реализации класса `ApptEncoder`.

Однако описанный подход не совсем удачен. Использование условных операторов иногда считается дурным тоном, но при создании объектов часто приходится их применять в определенный момент. Но не стоит слишком оптимистично относиться к тому, что в коде один за другим дублируются условные операторы. Класс `CommsManager` обеспечивает функции для передачи календарных данных. Предположим, что в используемом нами протоколе нужно вывести данные в верхний и

нижний колонтитулы, чтобы очертить границы каждой встречи. Давайте расширим предыдущий пример и создадим метод `getHeaderText()`.

```
class CommsManager {
   const  BLOGGS = 1;
   const    MEGA = 2;
   private $mode = 1;

   function __construct( $mode ) {
      $this->mode = $mode;
   }

   function getHeaderText() {
      switch ( $this->mode ) {
         case ( self::MEGA ):
            return "MegaCal верхний колонтитул\n";
         default:
            return "BloggsCal верхний колонтитул \n";
      }
   }

   function getApptEncoder() {
      switch ( $this->mode ) {
         case ( self::MEGA ):
            return new MegaApptEncoder();
         default:
            return new BloggsApptEncoder();
         }
      }
}
```

Как видите, необходимость в поддержке выходных данных верхнего колонтитула заставила нас продублировать проверку типа протокола с помощью оператора `switch`. Но по мере добавления новых протоколов код становится громоздким, особенно если мы еще добавим метод `getFooterText()`.

Итак, подытожим основные моменты проблемы.

- До момента выполнения программы мы не знаем, какой вид объекта нам понадобится создать (`BloggsApptEncoder` или `MegaApptEncoder`).
- Мы должны иметь возможность достаточно просто добавлять новые типы объектов (например, следующее требование бизнеса — поддержка протокола SyncML).
- Каждый тип продукта связан с контекстом, который требует других специализированных операций (`getHeaderText()`, `getFooterText()`).

Кроме того, нужно отметить, что мы используем условные операторы, и мы уже видели, что их можно заменить полиморфизмом. Шаблон Factory Method позволяет использовать наследование и полиморфизм, чтобы инкапсулировать создание конкретных продуктов. Другими словами, для каждого протокола создается свой подкласс типа `CommsManager`, в котором реализован свой метод `getApptEncoder()`.

Реализация

В шаблоне Factory Method классы создателей отделены от продуктов, которые они должны генерировать. Создатель — это класс фабрики, в котором определен

метод для генерации объекта-продукта. Если стандартной реализации этого метода не предусмотрено, то создание экземпляров объектов оставляют дочерним классам создателя. Обычно в каждом подклассе создателя создается экземпляр параллельного дочернего класса продукта.

Давайте переопределим класс `CommsManager` в виде абстрактного класса. Таким образом мы сохраним гибкий суперкласс и поместим весь наш код, связанный с конкретным протоколом, в отдельные подклассы. Это изменение продемонстрировано на рис. 9.4.

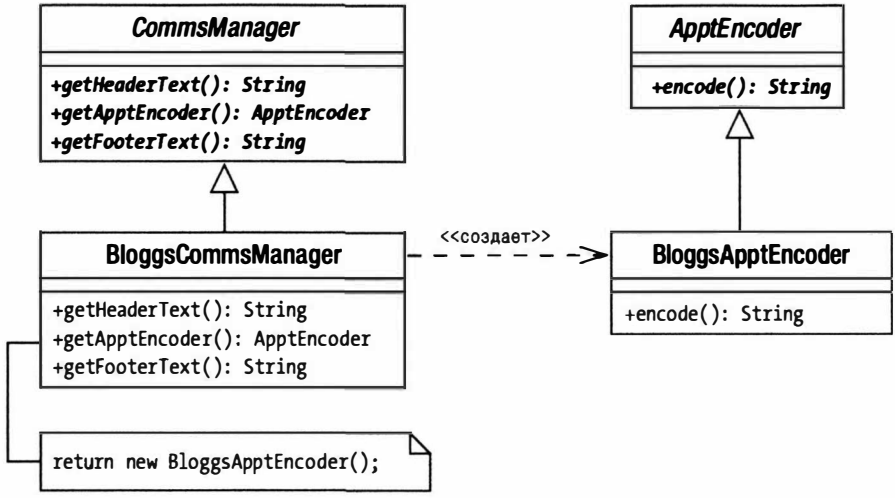


Рис. 9.4. Конкретные классы создателя и продукта

Вот пример упрощенного кода.

```
abstract class ApptEncoder {

   abstract function encode();
}

class BloggsApptEncoder extends ApptEncoder {

   function encode() {
      return "Данные о встрече закодированы в формате BloggsCal \n";
   }
}

abstract class CommsManager {

   abstract function getHeaderText();
   abstract function getApptEncoder();
   abstract function getFooterText();
}

class BloggsCommsManager extends CommsManager {

   function getHeaderText() {
      return "BloggsCal верхний колонтитул\n";
   }
```

```
    function getApptEncoder() {
        return new BloggsApptEncoder();
    }

    function getFooterText() {
        return "BloggsCal нижний колонтитул\n";
    }
}

$mgr = new BloggsCommsManager();
print $mgr->getHeaderText();
print $mgr->getApptEncoder()->encode();
print $mgr->getFooterText();
```

```
BloggsCal верхний колонтитул
Данные о встрече закодированы в формате BloggsCal
BloggsCal нижний колонтитул
```

Метод `BloggsCommsManager::getApptEncoder()` возвращает объект типа `BloggsApptEncoder`. Клиентский код, вызывающий `getApptEncoder()`, ожидает получить объект типа `ApptEncoder` и необязательно должен знать что-либо о конкретном классе продукта, который он получил. В некоторых языках программирования жестко оговариваются типы, возвращаемые методом. Поэтому клиентский код, вызывающий такой метод, как `getApptEncoder()`, может быть абсолютно уверен, что он получит объект типа `ApptEncoder`. В PHP 5 это вопрос соглашения. Поэтому очень важно документировать возвращаемые типы либо сообщать о них с помощью соглашений о наименовании.

На заметку. На момент написания данной книги в будущих версиях PHP планировалась реализация уточнения для возвращаемых типов.

Когда от нас потребуют реализовать формат MegaCal, его поддержка станет просто вопросом времени написания новой реализации для наших абстрактных классов. Классы MegaCal показаны на рис. 9.5.

Выводы

Обратите внимание на то, что наши классы-создатели отражают иерархию продуктов. Это обычный результат, получаемый в результате использования шаблона Factory Method. Некоторые программисты считают этот шаблон особым видом дублирования кода и поэтому часто испытывают к нему антипатию. Другая проблема — шаблон Factory Method часто способствует ненужному созданию подклассов. Поэтому, если для генерации подклассов создателя вы планируете применить шаблон Factory Method (и других причин для использования этого шаблона у вас нет!), рекомендуем сначала хорошо подумать. Именно поэтому мы продемонстрировали в приведенном выше примере ограничения, вызванные поддержкой верхнего и нижнего колонтитулов.

В нашем примере мы сосредоточились только на поддержке данных для встреч. Если же мы немного расширим пример и включим в него элементы типа "Что сделать" и "Контакты", то столкнемся с новой проблемой. Нам понадобится структура, которая будет работать с множеством связанных реализаций одновременно. Поэтому шаблон Factory Method часто используется вместе с шаблоном Abstract Factory, как мы увидим в следующем разделе.

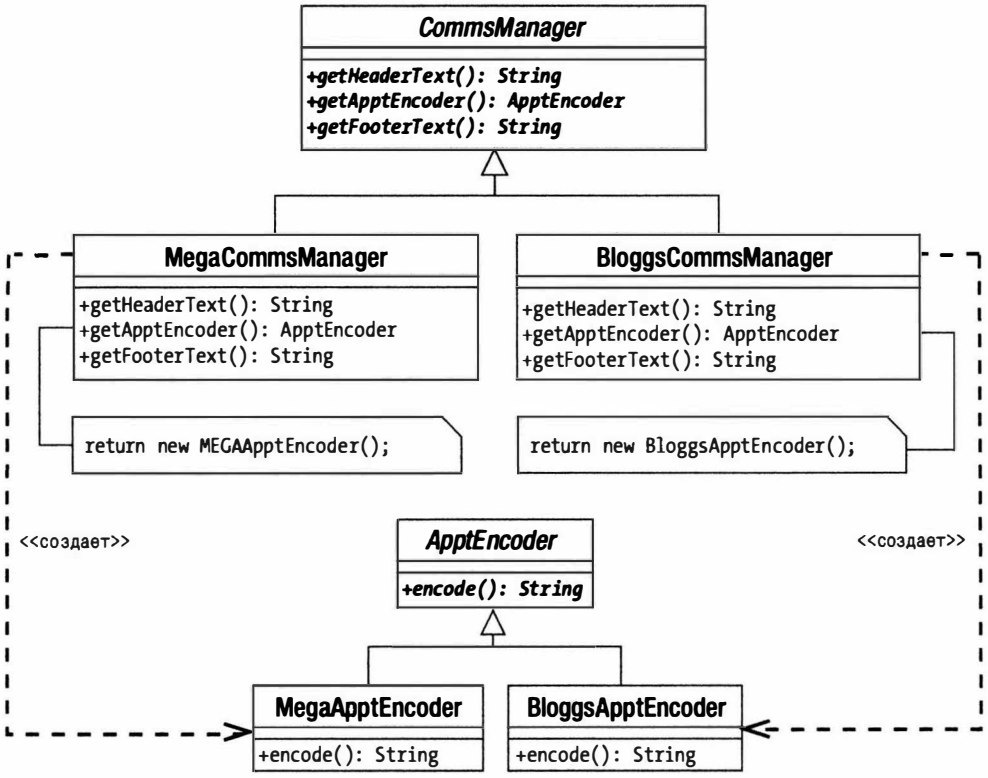


Рис. 9.5. Расширение проекта для поддержки нового протокола

Шаблон Abstract Factory

В больших приложениях вам, возможно, понадобятся фабрики, которые генерируют связанные наборы классов. Эту проблему решает шаблон Abstract Factory.

Проблема

Давайте еще раз рассмотрим пример с реализацией личного дневника. Мы написали код для двух форматов, BloggsCal и MegaCal. Мы можем легко нарастить эту структуру в горизонтальном направлении, добавив дополнительные форматы для кодирования. Но как нам нарастить ее вертикально, добавив кодировщики для различных типов объектов дневника? На самом деле мы уже работаем по этому шаблону.

На рис. 9.6 показаны параллельные семейства продуктов, с которыми нам предстоит работать. Это — встречи (`Appt`), "что сделать" (`Ttd`) и контакты (`Contact`).

Как видим, классы BloggsCal не связаны один с другим с помощью наследования (хотя они могут реализовывать общий интерфейс), но они выполняют параллельные функции. Если ваша система в настоящее время работает с классом `Bloggs TtdEncoder`, то она должна работать и с `BloggsContactEncoder`.

Чтобы понять, как это осуществить, начнем с интерфейса, как мы это делали в случае шаблона Factory Method (рис. 9.7).

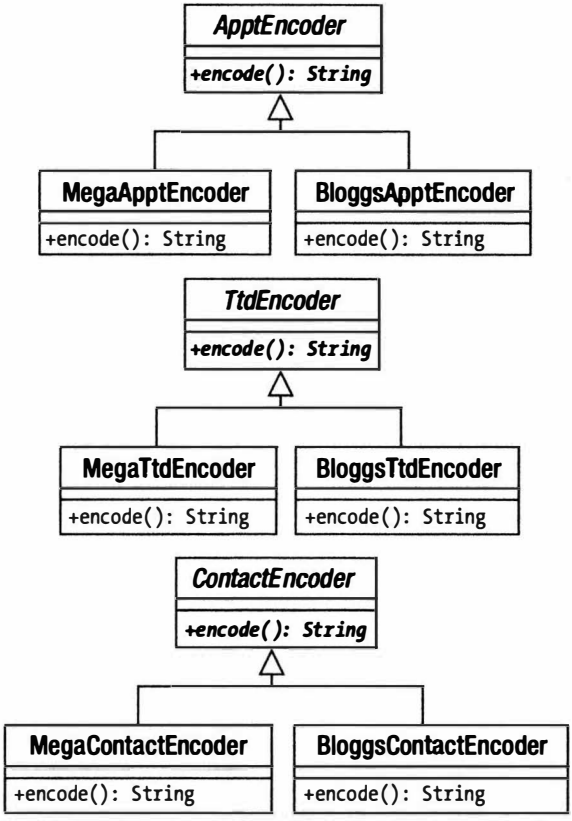


Рис. 9.6. Три семейства продуктов

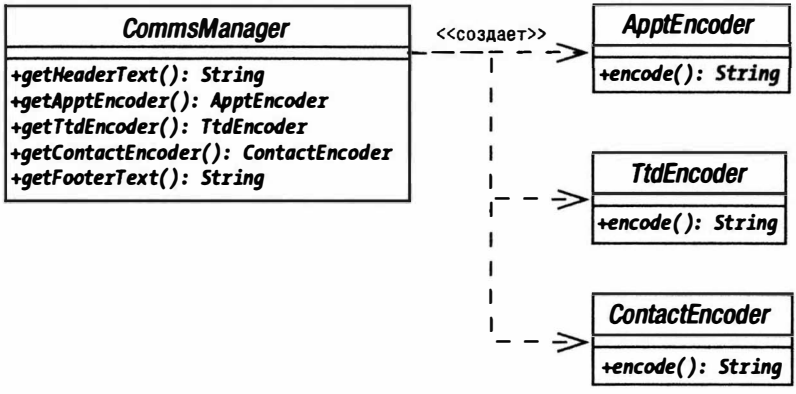


Рис. 9.7. Абстрактный создатель и его абстрактные продукты

Реализация

В абстрактном классе `CommsManager` определяется интерфейс для генерации каждого из трех продуктов (объекты типа `ApptEncoder`, `TtdEncoder` и `ContactEncoder`).

Поэтому нам нужно реализовать конкретный объект создателя, чтобы можно было генерировать конкретные объекты продуктов для определенного семейства. Как это можно сделать для формата BloggsCal, показано на рис. 9.8.

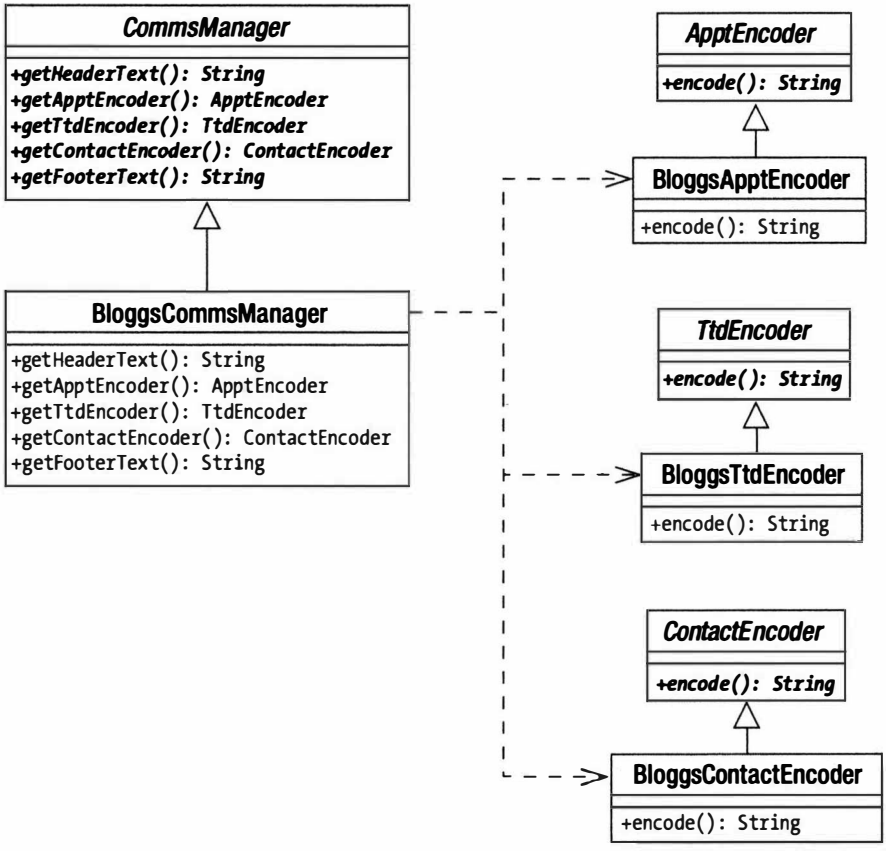


Рис. 9.8. Добавление конкретного создателя и некоторых конкретных продуктов

Вот вариант кода для `CommsManager` и `BloggsCommsManager`.

```
abstract class CommsManager {

   abstract function getHeaderText();
   abstract function getApptEncoder();
   abstract function getTtdEncoder();
   abstract function getContactEncoder();
   abstract function getFooterText();
}

class BloggsCommsManager extends CommsManager {

   function getHeaderText() {
      return "BloggsCal верхний колонтитул\n";
   }

   function getApptEncoder() {
```

```
        return new BloggsApptEncoder();
    }

    function getTtdEncoder() {
        return new BloggsTtdEncoder();
    }

    function getContactEncoder() {
        return new BloggsContactEncoder();
    }

    function getFooterText() {
        return "BloggsCal нижний колонтитул\n";
    }
}
```

Обратите внимание на то, что в этом примере мы используем шаблон Factory Method. Метод `getContactEncoder()` объявлен абстрактным в классе `CommsManager` и реализован в классе `BloggsCommsManager`. В результате проектные шаблоны работают совместно, причем один шаблон создает контекст, который служит для другого. На рис. 9.9 показано, что мы добавили поддержку формата MegaCal.

Выводы

Так что дает нам шаблон Abstract Factory?

- Во-первых, мы отделили нашу систему от деталей реализации. Мы можем добавлять или удалять любое количество кодирующих форматов в нашем примере, не опасаясь каких-либо проблем.
- Во-вторых, мы ввели в действие группировку функционально связанных элементов нашей системы. Поэтому при использовании `BloggsCommsManager` есть гарантия, что мы будем работать только с классами, связанными с BloggsCal.
- В-третьих, добавление новых продуктов может представлять проблему. Мы должны будем не только создать конкретные реализации новых продуктов, но и внести изменения в абстрактный класс создателя, а также создать каждый конкретный реализатор, чтобы его поддержать.

Во многих реализациях шаблона Abstract Factory используется шаблон Factory Method. Возможно, причина в том, что большинство примеров написано на Java или C++. Но в PHP не накладываются обязательные ограничения на тип, возвращаемый из метода, что дает нам больше гибкости, которой мы можем воспользоваться.

Вместо того чтобы создавать отдельные методы для каждого объекта шаблона Factory Method, мы можем создать один метод `make()` и передать ему в качестве аргумента флаг, определяющий тип возвращаемого объекта.

```
abstract class CommsManager {
    const    APPT = 1;
    const     TTD = 2;
    const CONTACT = 3;

    abstract function getHeaderText();
    abstract function make( $flag_int );
    abstract function getFooterText();
}
```

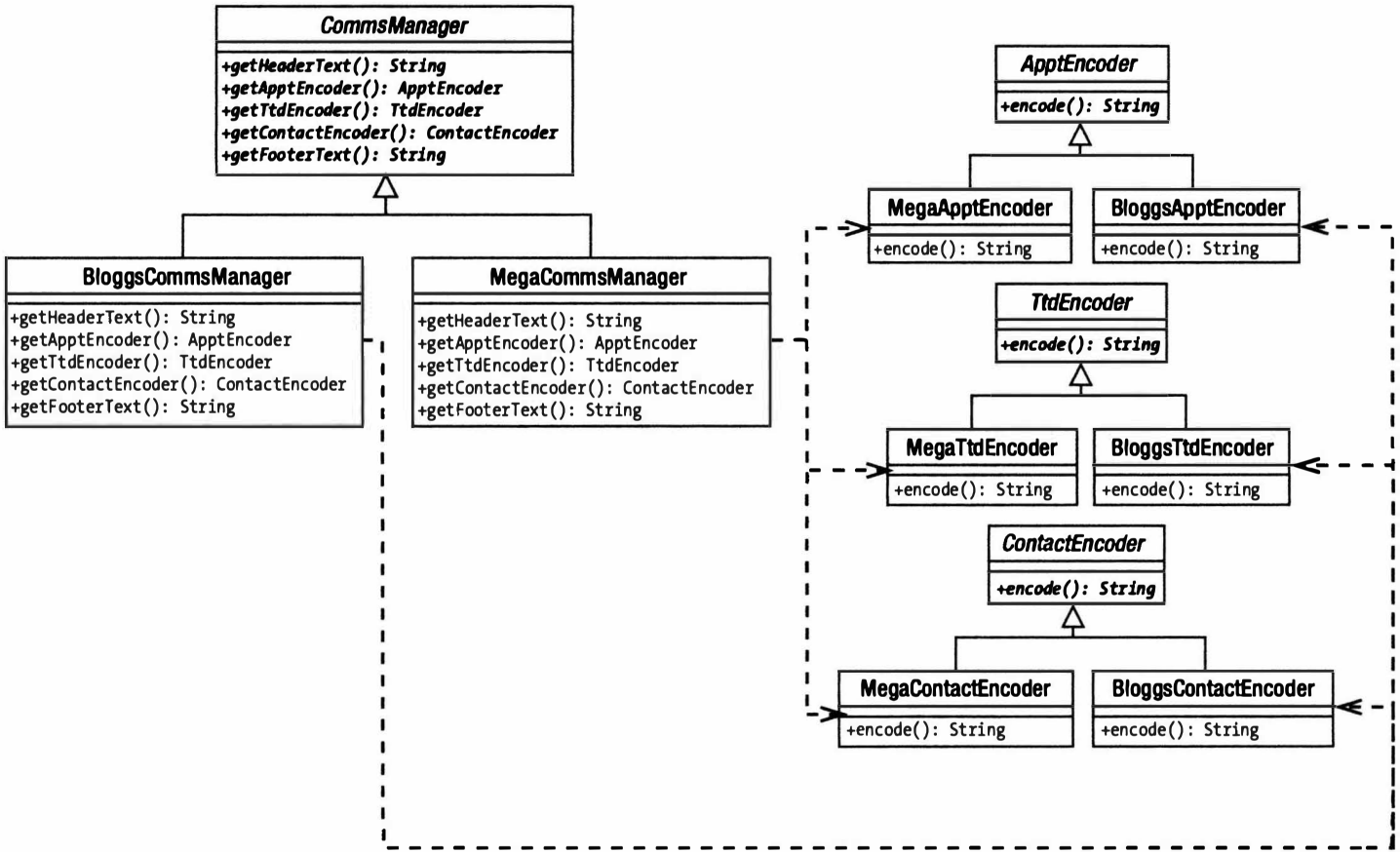


Рис. 9.9. Добавление конкретных создателей и некоторых конкретных продуктов

```
class BloggsCommsManager extends CommsManager {

   function getHeaderText() {
      return "BloggsCal верхний колонтитул\n";
   }

   function make( $flag_int ) {
      switch ( $flag_int ) {
         case self::APPT:
            return new BloggsApptEncoder();
         case self::CONTACT:
            return new BloggsContactEncoder();
         case self::TTD:
            return new BloggsTtdEncoder();
   }
}

   function getFooterText() {
      return "BloggsCal нижний колонтитул\n";
   }
}
```

Как видите, мы сделали интерфейс класса более компактным. Но мы достаточно дорого за это заплатили. Используя шаблон Factory Method, мы определяем четкий интерфейс и заставляем все конкретные объекты фабрики подчиняться ему. Используя единственный метод `make()`, мы должны помнить о поддержке всех объектов-продуктов во всех конкретных создателях. Мы также используем параллельные условные операторы, поскольку в каждом конкретном создателе должны быть реализованы одинаковые проверки флага-аргумента. Клиентский класс не может быть уверен, что конкретные создатели генерируют весь набор продуктов, поскольку внутренняя организация метода `make()` может отличаться в каждом случае.

С другой стороны, мы можем построить более гибкие создатели. В базовом классе создателя можно предусмотреть метод `make()`, который будет гарантировать стандартную реализацию для каждого семейства продуктов. И тогда конкретные дочерние классы могут избирательно модифицировать это поведение. Реализующим создателя классам будет предоставлено право выбора — вызывать стандартный метод `make()` после собственной реализации или нет.

Еще один вариант шаблона Abstract Factory мы рассмотрим в следующем разделе.

Шаблон Prototype

При использовании шаблона Factory Method проблемой может стать появление параллельных иерархий наследования. Этот вид тесной связи вызывает у некоторых программистов чувство дискомфорта. Каждый раз при добавлении нового семейства продуктов вы вынуждены создавать связанного с ним конкретного создателя (например, кодировщикам `BloggsCal` соответствует `BloggsCommsManager`). В системе, которая быстро расширяется и включает в себя много продуктов, поддержание связи такого рода может стать очень утомительным.

Один из способов избежать этой зависимости — использовать ключевое слово PHP `clone` для дублирования существующих конкретных продуктов. Затем конкретные классы продуктов сами станут основой для их собственной генерации. Ниже мы описали шаблон Prototype, который позволяет заменить наследование компози-

цией. Такой подход, в свою очередь, способствует гибкости во время выполнения программы и сокращает количество классов, которые мы должны создать.

Проблема

Представьте веб-игру типа "Цивилизация", в которой элементы действуют на клеточном поле. Каждая клетка может представлять море, равнину или лес. Тип местности может ограничивать движение и возможности элементов занять клетку. Предположим, у нас есть объект типа `TerrainFactory` (Фабрика местности), который позволяет создавать объекты `Sea` (море), `Forest` (лес) и `Plains` (равнины). Мы решили, что позволим пользователю выбирать совершенно разные типы окружающей среды, так что объект `Sea` — это абстрактный суперкласс, реализуемый в классах `MarsSea` и `EarthSea`. Объекты `Forest` и `Plains` реализуются аналогичным образом. В этой ситуации применим 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 класса, показывающая, как можно применить шаблоны Abstract Factory и Factory Method для работы с этими продуктами.

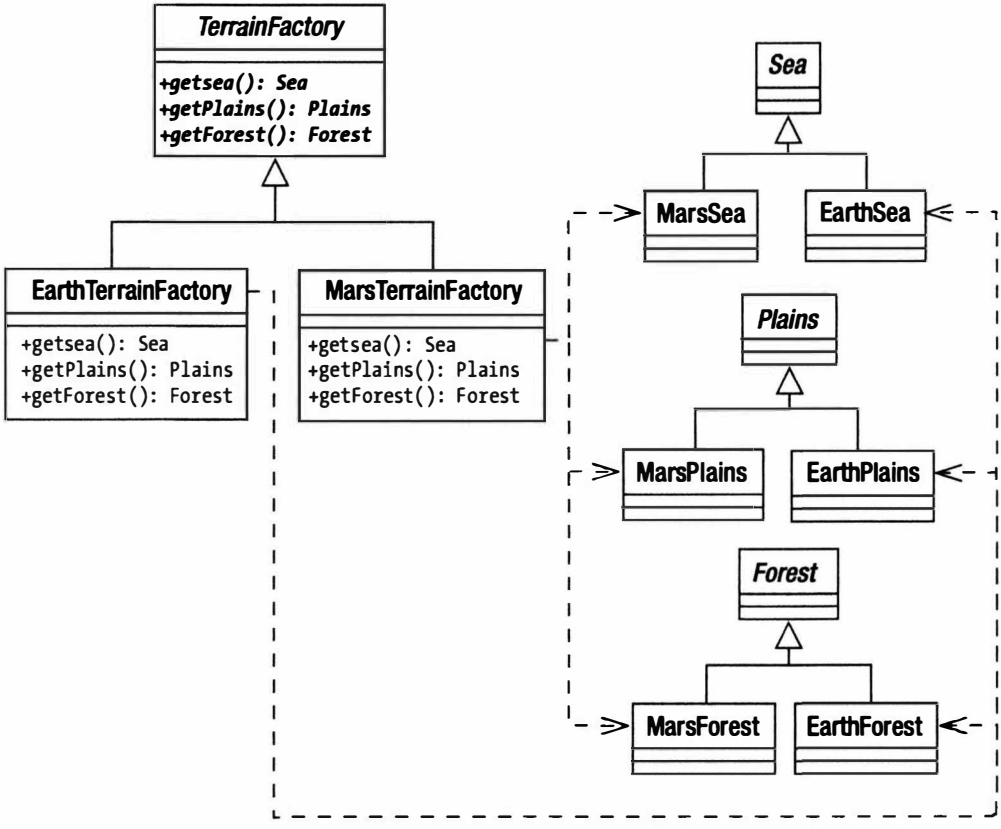


Рис. 9.10. Управление различными типами местности с помощью методов шаблона Abstract Factory

Как видите, мы используем наследование, чтобы сгруппировать семейство типов местности для продуктов, которые будет генерировать шаблон Factory. Это подходящее решение, но оно требует большой иерархии наследования и не обладает достаточной гибкостью. Если вам не нужны параллельные иерархии наследования и нужна максимальная гибкость во время выполнения программы, можно использовать шаблон Prototype в более мощном варианте шаблона Abstract Factory.

Реализация

Работая с шаблонами Abstract Factory и Factory Method, мы должны решить в определенный момент, с каким конкретно создателем хотим работать. Вероятно, это можно осуществить путем анализа значения некоторого флага. Поскольку так или иначе мы должны это сделать, почему бы просто не создать класс фабрики, хранящий конкретные продукты и размножающий их во время инициализации? Таким образом мы можем избавиться от нескольких классов и, как мы вскоре увидим, воспользоваться другими преимуществами. Приведем пример простого кода, в котором в качестве фабрики используется шаблон Prototype.

```
class Sea {}
class EarthSea extends Sea {}
class MarsSea extends Sea {}

class Plains {}
class EarthPlains extends Plains {}
class MarsPlains extends Plains {}

class Forest {}
class EarthForest extends Forest {}
class MarsForest extends Forest {}

class TerrainFactory {
   private $sea;
   private $forest;
   private $plains;

   function __construct( Sea $sea, Plains $plains, Forest $forest ) {
      $this->sea    = $sea;
      $this->plains = $plains;
      $this->forest = $forest;
   }

   function getSea( ) {
      return clone $this->sea;
   }

   function getPlains( ) {
      return clone $this->plains;
   }

   function getForest( ) {
      return clone $this->forest;
   }
}

$factory = new TerrainFactory( new EarthSea(),
```

```
                                new EarthPlains(),
                                new EarthForest() );
print_r( $factory->getSea() );
print_r( $factory->getPlains() );
print_r( $factory->getForest() );
```

```
EarthSea Object
(
)
EarthPlains Object
(
)
EarthForest Object
(
)
```

Как видите, здесь мы загружаем в экземпляр конкретной фабрики типа `Terrain Factory` экземпляры объектов наших продуктов. Когда в клиентском коде вызывается метод `getSea()`, ему возвращается клон объекта `Sea`, который мы поместили в кеш во время инициализации. В результате мы не только сократили пару классов, но и достигли 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 писать новый класс создателя — достаточно просто изменить набор классов, который мы добавляем в `TerrainFactory`.

```
$factory = new TerrainFactory( new EarthSea(),
                               new MarsPlains(),
                               new EarthForest() );
```

Итак, шаблон Prototype позволяет пользоваться преимуществами гибкости, которые предоставляет композиция. Но мы получили нечто большее. Поскольку мы сохраняем и клонируем объекты во время выполнения, воспроизводим состояние объектов, когда генерируем новые продукты. Предположим, у объектов `Sea` есть свойство `$navigability` (судоходность). От него зависит количество энергии движения, которое клетка моря отнимает у судна. С помощью этого свойства можно регулировать уровень сложности игры.

```
class Sea {
   private $navigability = 0;

   function __construct( $navigability ) {
      $this->navigability = $navigability;
   }
}
```

Теперь, во время инициализации объекта `TerrainFactory`, можно добавить объект `Sea` с модификатором судоходности. Затем это будет иметь силу для всех объектов `Sea`, создаваемых с помощью `TerrainFactory`.

```
$factory = new TerrainFactory( new EarthSea( -1 ),
                               new EarthPlains(),
                               new EarthForest() );
```

Эта гибкость также становится очевидной, когда объект, который нужно сгенерировать, состоит из других объектов. Например, все объекты типа `Sea` могут содержать объекты типа `Resource` (`FishResource`, `OilResource` и т.д.). В соответствии со значением свойства мы можем определить, что по умолчанию все объекты `Sea`

содержат объекты типа `FishResource`. Но помните, что если в ваших продуктах имеются ссылки на другие объекты, вы можете реализовать метод `__clone()`, чтобы корректно можно было сделать глубокую копию этого продукта.

```
class Contained { }

class Container {
   public $contained;

   function __construct() {
      $this->contained = new Contained();
   }

   function __clone() {
      // Нужно сделать так, чтобы в клонируемом объекте был
      // именно клон объекта, содержащегося в свойстве self::$contained,
      // а не ссылка на него
         $this->contained = clone $this->contained;
   }
}
```

На заметку. О клонировании объектов говорилось в главе 4. Ключевое слово `clone` позволяет создать поверхностную копию объекта, к которому оно применяется. Это означает, что объект-продукт будет иметь те же значения свойств, что и исходный объект. Если какие-либо свойства исходного объекта содержат ссылки на другие объекты, то значения последних не будут скопированы в новый продукт. Вместо этого свойства нового продукта будут содержать ссылки на *старые* объекты. Однако в наших силах изменить эту ситуацию и настроить копирование объектов по-другому, реализовав метод `__clone()`. Он вызывается автоматически при использовании ключевого слова `clone`.

Но это обман!

Я обещал, что в этой главе будет говориться о логике создания объектов, а не о конкретных "объектно-ориентированных" примерах. Но в некоторых рассмотренных шаблонах в скрытом виде присутствует принятие решений о создании объектов, если не само создание объектов.

Шаблон Singleton в этом обвинить нельзя. Логика создания объектов в нем встроена и совершенно однозначна. В шаблоне Abstract Factory создание семейств продуктов группируется в различных конкретных классах-создателях. Но как решить, какой конкретно создатель использовать? С аналогичной проблемой вы столкнетесь при использовании шаблона Prototype. Оба эти шаблона управляют созданием объектов, но они откладывают решение о том, какой объект (или группа объектов) должен быть создан.

Решение о том, какой конкретно создатель будет выбран, обычно принимается в соответствии со значением параметра, заданного при конфигурации приложения. Он может находиться в базе данных, в файле конфигурации или в файле на сервере (таком, как файл конфигурации уровня каталогов сервера Apache, который обычно называется `.htaccess`) или может быть даже жестко закодирован в виде PHP-переменной или свойства. Поскольку PHP-приложения необходимо реконфигурировать для каждого запроса, инициализация сценария должна быть как можно менее "болезненной". Поэтому я во многих случаях жестко кодирую значения флагов конфигурации в PHP-коде. Можно сделать это вручную или написать сценарий,

который автоматически генерирует файл класса. Вот пример класса, в который включен признак типа протокола календаря.

```
class Settings {
   static $COMMSTYPE = 'Bloggs';
}
```

Теперь, когда у нас есть значение параметра (хотя и сделанного неэлегантно), можно создать класс, который использует его для принятия решения о том, какой объект типа `CommsManager` нужно предоставить по запросу. В таких ситуациях часто используют шаблон Singleton в сочетании с шаблоном Abstract Factory, поэтому давайте так и поступим.

```
require_once( 'Settings.php' );

class AppConfig {
   private static $instance;
   private $commsManager;

   private function __construct() {
   // Выполняется только один раз
      $this->init();
   }

   private function init() {
      switch ( Settings::$COMMSTYPE ) {
         case 'Mega':
            $this->commsManager = new MegaCommsManager();
            break;
         default:
            $this->commsManager = new BloggsCommsManager();
      }
   }

   public static function getInstance() {
      if ( empty( self::$instance ) ) {
         self::$instance = new self();
      }
   return self::$instance;
   }

   public function getCommsManager() {
      return $this->commsManager;
   }
}
```

Класс `AppConfig` — это стандартный Singleton. Поэтому мы можем легко получить ссылку на экземпляр `AppConfig` в любом месте системы, причем всегда на один и тот же экземпляр. Метод `init()` вызывается конструктором класса и поэтому запускается только один раз в процессе выполнения программы. В нем проверяется свойство `Settings::$COMMSTYPE`, и в соответствии с его значением создается экземпляр конкретного объекта типа `CommsManager`. Теперь наш сценарий может получить объект типа `CommsManager` и работать с ним, даже не зная о его конкретных реализациях или конкретных классах, которые он генерирует.

```
$commsMgr = AppConfig::getInstance()->getCommsManager();
print $commsMgr->getApptEncoder()->encode();
```

Резюме

В этой главе мы рассмотрели ряд приемов, которые вы можете использовать при генерации объектов. Мы изучили шаблон Singleton, который предоставляет глобальный доступ к единственному экземпляру объекта, и рассмотрели также шаблон Factory Method, в котором принцип полиморфизма применяется к генерации объектов. Мы использовали сочетание шаблонов Factory Method и Abstract Factory для генерации классов-создателей, которые создают экземпляры наборов связанных объектов. И наконец мы рассмотрели шаблон Prototype и увидели, как клонирование объектов позволяет использовать композицию при генерации объектов.

Глава 10

Шаблоны для программирования гибких объектов

После изучения стратегии генерации объектов мы можем рассмотреть некоторые стратегии структурирования классов и объектов. В особенности мы сосредоточимся на том принципе, что композиция предоставляет бо́льшую гибкость, чем наследование. Шаблоны, которые мы изучим в этой главе, снова взяты из каталога "Банды четырех".

В этой главе мы рассмотрим следующее.

- *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 с целью расширения функциональности.
- *Шаблон Facade:* создание простого интерфейса для сложных или изменчивых систем.

Как структурировать классы, чтобы достичь гибкости

В главе 4 я уже говорил о том, что начинающие программисты часто путают объекты и классы. Но это еще не все. Остальные время от времени в недоумении чешут затылки, разглядывая UML-диаграммы классов и пытаясь согласовать статичные структуры наследования с динамическими отношениями, в которые могут вступать объекты.

Помните принцип шаблонов "Предпочитайте композицию наследованию"? В этом принципе подчеркивается напряженность между организацией классов и объектов. Чтобы достичь гибкости в наших проектах, мы структурируем классы так, чтобы их объекты можно было компоновать в удобные структуры во время выполнения программы.

Это общая тема, которая будет звучать при рассмотрении двух первых шаблонов в данной главе. Наследование — важная функция в обоих этих шаблонах, но ее важность отчасти объясняется предоставлением механизма, с помощью которого композицию можно использовать для представления структур и расширения функциональности.

Шаблон Composite

Шаблон Composite — это, наверное, самый экстремальный пример наследования, которое используется для обслуживания композиции. У этого шаблона простая, но поразительно изящная конструкция. Он также удивительно полезен. Но имейте в виду: из-за всех этих преимуществ у вас может возникнуть искушение слишком часто его использовать.

Шаблон Composite — это простой способ объединения, а затем и управления группами схожих объектов. В результате отдельный объект для клиентского кода становится неотличимым от коллекции объектов. В сущности, этот шаблон очень прост, тем не менее он может сбить с толку. Одна из причин этого — сходство структуры классов в шаблоне с организацией его объектов. Иерархии наследования представляют собой деревья, корнем которых является суперкласс, а ветвями — специализированные подклассы. Дерево наследования *классов*, созданных с помощью шаблона Composite, предназначено для того, чтобы разрешить простую генерацию *объектов* и их обход по дереву.

Если вы еще не знакомы с этим шаблоном, то сейчас у вас есть полное право на то, чтобы почувствовать себя сбитым с толку. Чтобы проиллюстрировать, каким образом с отдельными объектами можно обращаться так же, как с наборами объектов, давайте воспользуемся аналогией. Имея такие ингредиенты, как крупа и мясо (или соя, в зависимости от предпочтений), можно произвести пищевой продукт, например колбасу. А затем с полученным результатом мы будем обращаться, как с единым объектом. Точно так же, как мы едим, готовим, покупаем или продаем мясо, мы можем есть, готовить, покупать или продавать колбасу, одним из компонентов которой является мясо. Мы можем взять колбасу и соединить ее с другими составными ингредиентами, чтобы сделать пирог, тем самым включив одну составную часть (композит) в большую составную часть (другой композит). Заметьте, мы обращаемся с наборами точно так же, как с их составными частями. Шаблон Composite помогает моделировать отношение между наборами и компонентами в нашем коде.

Проблема

Управление группами объектов может быть довольно сложной задачей, особенно если эти объекты также содержат собственные объекты. Это очень распространенная проблема в кодировании. Представьте счет-фактуру, в строках которой указаны дополнительные продукты или услуги, или список дел, которые нужно выполнить, элементы которого сами содержат несколько подзадач. При управлении контентом мы можем оперировать деревьями подразделов, страниц, статей и медийных компонентов. Управление этими структурами извне может быстро превратиться в очень сложную задачу.

Вернемся к сценарию, описанному в предыдущей главе. Напомним, что мы разрабатываем веб-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. Давайте определим пару типов боевых единиц (юнитов).

```
abstract class Unit {
   abstract function bombardStrength();
}

class Archer extends Unit {
```

```
    function bombardStrength() {
        return 4;
    }
}

class LaserCannonUnit extends Unit {

    function bombardStrength() {
        return 44;
    }
}
```

В классе `Unit` определен абстрактный метод `bombardStrength()`, который определяет атакующую силу элемента, обстреливающего соседнюю клетку. Мы реализуем этот метод в обоих классах — `Archer` и `LaserCannonUnit`. Эти классы могут также содержать информацию о перемещении и возможностях защиты, но давайте пока остановимся на самом простом варианте. Мы можем определить отдельный класс для группировки юнитов следующим образом.

```
class Army {
    private $units = array();

    function addUnit( Unit $unit ) {
        array_push( $this->units, $unit );
    }

    function bombardStrength() {
        $ret = 0;
        foreach( $this->units as $unit ) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}
```

У класса `Army` есть метод `addUnit()`, которому передается объект типа `Unit`. Объекты типа `Unit` сохраняются в массиве свойств с именем `$units`. Мы вычисляем объединенную силу нашей армии в методе `bombardStrength()`. Для этого просто выполняется итерация по объединенным объектам `Unit` и для каждого объекта вызывается метод `bombardStrength()`.

Это идеальная модель, пока задача остается настолько простой. Но что будет, если мы добавим несколько новых требований? Предположим, армия должна быть способна объединяться с другими армиями. При этом каждая армия должна сохранять свою индивидуальность, чтобы впоследствии она могла выйти из более крупного соединения. Сегодня у храбрых вояк ArchDuke может быть общая причина с генералом Соамсом 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.

Мы можем изменить класс `Army` так, чтобы в нем можно было оперировать как объектами типа `Army`, так и объектами типа `Unit`.

```
function addArmy( Army $army ) {
    array_push( $this->armies, $army );
}
```

Нам нужно изменить метод `bombardStrength()`, чтобы делать итерации по всем армиям и объектам типа `Unit`.

```
function bombardStrength() {
   $ret = 0;
   foreach( $this->units as $unit ) {
      $ret += $unit->bombardStrength();
   }

   foreach( $this->armies as $army ) {
      $ret += $army->bombardStrength();
   }
   return $ret;
}
```

Дополнительная сложность в данный момент не представляет особой проблемы. Но помните, что мы должны сделать нечто подобное и с другими методами, например, определяющими защитную силу `defensiveStrength()`, диапазон перемещения `movementRange()` и т.д. Наша игра обещает быть функционально богатой. Уже деловые круги вызывают транспортно-десантные самолеты, которые могут содержать до десяти юнитов, чтобы выполнить быструю переброску войск и улучшить свои позиции в некоторых районах. Очевидно, что транспортный самолет подобен армии в том, что в нем группируются объекты типа `Unit`. Но у него также могут быть собственные характеристики. Мы можем снова изменить класс `Army`, чтобы управлять транспортными самолетами (объектами `TroopCarrier`), но ведь может появиться необходимость и в других типах группировки объектов `Unit`. Очевидно, что нам нужна более гибкая модель.

Давайте еще раз рассмотрим модель, которую строим. Всем созданным классам нужен метод `bombardStrength()`. В сущности, клиентскому коду не нужно различать армию, юнит или транспортный самолет. Функционально они идентичны. Они должны уметь перемещаться, атаковать и защищаться. А объектам, которые содержат другие объекты, нужны методы добавления и удаления. Эти сходные черты приводят нас к неизбежному выводу. Поскольку объекты-контейнеры совместно используют интерфейс с объектами, которые они содержат, вполне естественно, что они должны принадлежать к одному семейству типов.

Реализация

В шаблоне Composite определяется единственная иерархия наследования, которая устанавливает два различных набора функциональных обязанностей. Мы их уже видели в нашем примере. Классы в шаблоне должны поддерживать общий набор операций как основную обязанность. Для нашего случая — это метод `bombardStrength()`. Классы должны также поддерживать методы для добавления и удаления объектов.

На рис. 10.1 показана диаграмма класса, иллюстрирующая применение шаблона Composite к данной проблеме.

Как видите, все элементы нашей модели являются производными от класса `Unit`. Поэтому клиентский код может быть уверен, что любой объект типа `Unit` будет поддерживать метод `bombardStrength()`. А это значит, что с объектом `Army` можно обращаться точно так же, как с объектом `Archer`.

Классы `Army` и `TroopCarrier` — это *композиты*; они предназначены для того, чтобы содержать объекты типа `Unit`. Классы `Archer` и `LaserCannon` — это *листья*, предназначенные для того, чтобы поддерживать операции с объектами типа `Unit`; в

них не могут содержаться другие объекты типа `Unit`. Тут возникает вопрос "Должны ли «листья» подчиняться тому же интерфейсу, что и композиты, как показано на рис. 10.1?" На этой диаграмме показано, что в классах `TroopCarrier` и `Army` агрегируются другие объекты типа `Unit`, хотя в классах-"листьях" также нужно реализовать метод `addUnit()`. Вскоре я вернусь к этому вопросу, а пока определим абстрактный класс `Unit`.

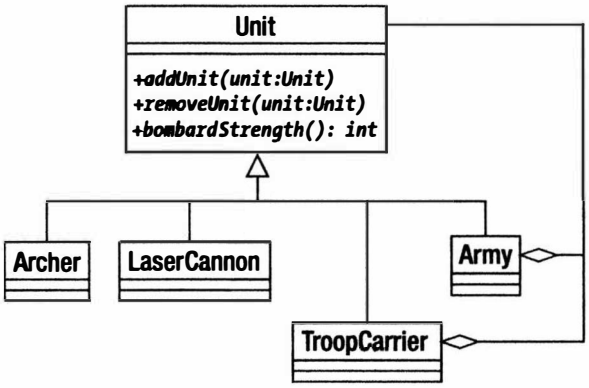


Рис. 10.1. Шаблон Composite в действии

```
abstract class Unit {
   abstract function    addUnit( Unit $unit );
   abstract function removeUnit( Unit $unit );
   abstract function bombardStrength();
}
```

Как видите, мы определили основные функции для всех объектов типа `Unit`. А теперь давайте посмотрим, как в объекте-композите могут быть реализованы эти абстрактные методы.

```
class Army extends Unit {
   private $units = array();

   function addUnit( Unit $unit ) {
      if ( in_array( $unit, $this->units, true ) ) {
         return;
      }
      $this->units[] = $unit;
   }

   function removeUnit( Unit $unit ) {
      $this->units = array_udiff( $this->units, array( $unit ),
               function( $a, $b ) { return ($a === $b)?0:1; } );
   }

   function bombardStrength() {
      $ret = 0;
      foreach( $this->units as $unit ) {
         $ret += $unit->bombardStrength();
      }
      return $ret;
   }
}
```

В методе `addUnit()` переданный в качестве параметра объект типа `Unit` сохраняется в закрытом массиве свойств `$units`. Перед этим проверяется, чтобы в массиве `$units` не было дублей объектов типа `Unit`. В методе `removeUnit()` используется аналогичная проверка, и если объект типа `Unit` найден, он удаляется из закрытого массива свойств `$units`.

На заметку. В методе `removeUnit()` я воспользовался анонимной функцией обратного вызова. Она предназначена для проверки элементов массива, содержащихся в свойстве `$units`, на эквивалентность. Анонимные функции впервые появились в PHP 5.3. Если вы работаете со старой версией PHP, то для достижения такого же эффекта используйте функцию `create_function()`, как показано ниже.

```
$this->units = array_udiff( $this->units, array( $unit ),
               create_function( '$a,$b', 'return ($a === $b)?0:1;' ) );
```

В объектах типа `Army` могут храниться ссылки на любые объекты типа `Unit`, включая другие объекты типа `Army`, а также листья, такие как `Archer` или `LaserCannonUnit`. Поскольку все подразделения гарантированно поддерживают метод `bombardStrength()`, в нашем методе `Army::bombardStrength()` просто делается итерация по всем дочерним объектам типа `Unit`, сохраненным в свойстве `$units`, и для каждого из них вызывается один и тот же метод.

Существует один проблематичный аспект шаблона Composite — это реализация функций добавления и удаления. В классическом шаблоне методы `add()` и `remove()` помещены в абстрактный суперкласс. Это гарантирует, что во всех классах шаблона совместно используется общий интерфейс. Но, как видите, это также означает, что в классах-"листьях" нужно обеспечить их реализацию.

```
class UnitException extends Exception {}

class Archer extends Unit {
   function addUnit( Unit $unit ) {
      throw new UnitException( get_class($this) . " относится к 'листьям'" );
   }

   function removeUnit( Unit $unit ) {
      throw new UnitException( get_class($this) . " относится к 'листьям'");
   }

   function bombardStrength() {
      return 4;
   }
}
```

Нам не нужна возможность добавлять объект типа `Unit` к объекту типа `Archer`. Поэтому при вызове метода `addUnit()` или `removeUnit()` генерируется исключение. Нам понадобится сделать это для всех объектов-"листьев", поэтому, вероятно, мы можем улучшить проект, заменив абстрактные методы `addUnit()`/`removeUnit()` в объекте `Unit` стандартными реализациями, как в предыдущем примере.

```
abstract class Unit {
   abstract function bombardStrength();

   function addUnit( Unit $unit ) {
      throw new UnitException( get_class($this) . " относится к 'листьям'");
   }

   function removeUnit( Unit $unit ) {
```

```
        throw new UnitException( get_class($this) . " относится к 'листьям'");
    }
}

class Archer extends Unit {
    function bombardStrength() {
        return 4;
    }
}
```

Такой подход позволяет избавиться от дублирования кода в классах-"листьях". Однако у него имеется серьезный недостаток — шаблон Composite во время компиляции не обязан обеспечивать реализацию методов `addUnit()` и `removeUnit()`, что в конечном итоге может стать причиной проблем.

Некоторые проблемы шаблона Composite мы подробнее рассмотрим в следующем разделе. А этот раздел давайте закончим тем, что подытожим преимущества данного шаблона.

- *Гибкость.* Поскольку во всех элементах шаблона Composite используется общий супертип, очень просто добавлять к проекту новые объекты-композиты или "листья", не меняя более широкий контекст программы.
- *Простота.* Клиентский код, использующий структуру Composite, имеет простой интерфейс. Клиентскому коду не нужно делать различие между объектом, состоящим из других объектов, и объектом-"листом" (за исключением случая добавления новых компонентов). Вызов метода `Army::bombardStrength()` может стать причиной серии делегированных внутренних вызовов, но для клиентского кода процесс и результат в точности эквивалентны тому, что связано с вызовом `Archer::bombardStrength()`.
- *Неявная досягаемость.* В шаблоне Composite объекты организованы в древовидную структуру. В каждом композите содержатся ссылки на дочерний объект. Поэтому операция над определенной частью дерева может иметь более широкий эффект. Мы можем удалить один объект `Army` из его родительского объекта `Army` и добавить к другому. Это простое действие осуществляется над одним объектом, но в результате изменяется статус объектов `Unit`, на которые ссылается объект `Army`, а также статус их дочерних объектов.
- *Явная досягаемость.* В древовидной структуре можно легко выполнить обход всех ее узлов. Для получения информации нужно последовательно перебрать все ее узлы либо выполнить преобразования. Очень эффективные методы осуществления этих действий мы рассмотрим в следующей главе при изучении шаблона Visitor.

Во многих случаях реально увидеть преимущества шаблона можно только с точки зрения клиентского кода, поэтому давайте создадим пару армий.

```
// Создадим армию
$main_army = new Army();

// Добавим пару боевых единиц
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );

// Создадим еще одну армию
$sub_army = new Army();
```

```
// Добавим несколько боевых единиц
$sub_army->addUnit( new Archer() );
$sub_army->addUnit( new Archer() );
$sub_army->addUnit( new Archer() );

// Добавим вторую армию к первой
$main_army->addUnit( $sub_army );

// Все вычисления выполняются за кулисами
print "Атакующая сила: {$main_army->bombardStrength()}\n";
```

Мы создаем новый объект `Army` и добавляем несколько боевых единиц `Unit`. Повторяем этот процесс для второго объекта `Army`, который затем добавляем к первому. Когда мы вызываем метод `Unit::bombardStrength()` для первого объекта `Army`, вся сложность структуры, которую мы построили, оказывается полностью скрытой.

Промежуточные выводы

Если вы в чем-то похожи на меня, то, увидев фрагмент кода для класса `Archer`, должны были сильно задуматься. Для чего нам нужны эти лишние методы `addUnit()` и `removeUnit()` в классах-"листьях", 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 шаблона Composite, который заключается в том, что у элементарных классов ("листьев") такой же интерфейс, как у композитов. Это не особенно нам помогает, потому что мы по-прежнему не знаем, насколько безопасно вызывать метод `addUnit()` для любого объекта `Unit`, с которым мы можем столкнуться.

Если переместить эти методы добавления/удаления вниз, чтобы они были доступны только классам композитов, то при передаче объекта типа `Unit` методу возникает проблема из-за того, что по умолчанию мы не знаем, поддерживает он метод `addUnit()` или нет. Тем не менее оставлять методы-заглушки в классах-"листьях" мне кажется неправильным. Пользы от этого никакой, а проект усложняется, поскольку интерфейс дает ложные сведения о собственной функциональности.

Мы можем легко разбить классы-композиты на подтипы `CompositeUnit`. Прежде всего, мы исключим функции добавления/удаления боевых единиц из класса `Unit`.

```
abstract class Unit {
   function getComposite() {
      return null;
   }
   abstract function bombardStrength();
}
```

Обратите внимание на новый метод `getComposite()`. Мы вернемся к нему через некоторое время. А теперь нам нужен новый абстрактный класс, который будет поддерживать методы `addUnit()` и `removeUnit()`. Мы можем обеспечить даже его стандартные реализации.

```
abstract class CompositeUnit extends Unit {
   private $units = array();

   function getComposite() {
      return $this;
   }
```

```
    protected function units() {
        return $this->units;
    }

    function removeUnit( Unit $unit ) {
        $this->units = array_udiff( $this->units, array( $unit ),
                 function( $a, $b ) { return ($a === $b)?0:1; } );
    }

    function addUnit( Unit $unit ) {
        if ( in_array( $unit, $this->units, true ) ) {
            return;
        }
        $this->units[] = $unit;
    }
}
```

Класс `CompositeUnit` объявлен абстрактным, несмотря на то что в нем не объявлено никаких абстрактных методов. Однако он расширяет класс `Unit` и в нем не реализован абстрактный метод `bombardStrength()`. Класс `Army` (и любые другие классы-композиты) теперь может расширять класс `CompositeUnit`. Классы в нашем примере теперь организованы, как показано на рис. 10.2.

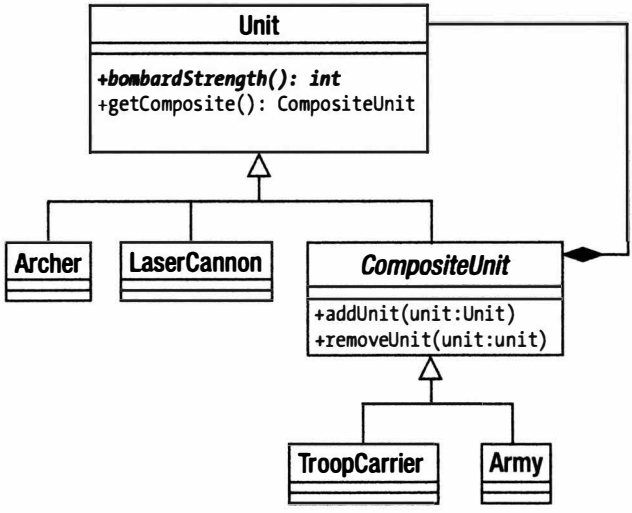


Рис. 10.2. Перемещение методов добавления/удаления боевых единиц из базового класса

Теперь у нас нет бесполезной реализации методов добавления/удаления боевых единиц в классах-"листьях", но в клиентском коде перед вызовом метода `addUnit()` нужно по-прежнему проверить тип объекта (унаследован ли он от класса `CompositeUnit`).

Вот тут-то и вступает в свои права метод `getComposite()`. По умолчанию этот метод возвращает нулевое значение. Только в классе `CompositeUnit` он возвращает ссылку на объект типа `CompositeUnit`. Поэтому если вызов этого метода возвращает объект, то мы должны иметь возможность вызвать для него метод `addUnit()`. Вот как это можно использовать в клиентском коде.

```
class UnitScript {
   private $comp;

   static function joinExisting( Unit $newUnit,
                                 Unit $occupyingUnit ) {

      if ( ! is_null( $comp = $occupyingUnit->getComposite() ) ) {
         $comp->addUnit( $newUnit );
      } else {
         $comp = new Army();
         $comp->addUnit( $occupyingUnit );
         $comp->addUnit( $newUnit );
      }
      return $comp;
   }
}
```

Методу `joinExisting()` передаются два объекта типа `Unit`. Первый — это объект, вновь прибывший на клетку, а второй — объект, который занимал клетку до этого. Если второй объект типа `Unit` принадлежит к классу `CompositeUnit`, то первый объект попытается присоединиться к нему. Если нет, то будет создан новый объект типа `Army`, включающий оба объекта типа `Unit`. А с самого начала у нас нет способа узнать, содержит ли аргумент `$occupyingUnit` объект класса `CompositeUnit`. Но вызов метода `getComposite()` позволит решить проблему. Если метод `getComposite()` возвращает объект, то мы можем непосредственно добавить к нему новый объект типа `Unit`. В противном случае мы создадим новый объект типа `Army` и добавим к нему оба объекта.

Мы можем еще более упростить эту модель, сделав так, чтобы метод `Unit::getComposite()` вернул объект типа `Army`, содержащий текущий объект типа `Unit`. Либо можем вернуться к предыдущей модели (в которой не было структурного разделения между объектами-композитами и "листьями"), где метод `Unit::addUnit()` делает то же самое: создает объект типа `Army` и добавляет к нему оба объекта типа `Unit`. Это ясно и понятно, когда предполагается, что вы знаете заранее тип композита, который хотите использовать для объединения объектов типа `Unit`. Предположения, которые вы сделаете при проектировании таких методов, как `getComposite()` и `addUnit()`, будут определяться логикой вашего приложения.

Эти перекосы говорят о недостатках шаблона Composite. Простота достигается за счет гарантии того, что все классы происходят из общего базового класса. Но за простоту иногда приходится платить безопасностью использования типа. Чем сложнее становится модель, тем больше вероятность, что вам придется вручную делать проверку типов. Предположим, у нас есть объект типа `Cavalry` (кавалерия). Если по правилам нашей игры нельзя помещать лошадь на бронетранспортер, то мы не сможем автоматически сделать это с помощью шаблона Composite.

```
class TroopCarrier extends CompositeUnit {

   function addUnit( Unit $unit ) {
      if ( $unit instanceof Cavalry ) {
         throw new UnitException(
                   "Нельзя помещать лошадь на бронетранспортер");
      }
      super::addUnit( $unit );
   }
```

```
    function bombardStrength() {
        return 0;
    }
}
```

Здесь мы обязаны использовать оператор `instanceof`, чтобы протестировать тип объекта, переданного методу `addUnit()`. Случаев подобного рода существует слишком много, поэтому недостатки шаблона начинают перевешивать его преимущества. Шаблон Composite наиболее эффективен, когда большинство компонентов являются взаимозаменяемыми.

Еще один вопрос, о котором нужно помнить, — это стоимость некоторых операций шаблона Composite. Метод `Army::bombardStrength()` типичен в том, что запускает серию вызовов одного и того же метода вниз по дереву. Для большого дерева с множеством вложенных армий единственный вызов может стать причиной лавины внутренних вызовов. Метод `bombardStrength()` сам по себе не является слишком "дорогим", но что произойдет, если некоторые "листья" должны выполнять сложные вычисления, чтобы получить возвращаемые значения? Один из способов решения этой проблемы — сохранить результат вызова подобного метода в родительском объекте, чтобы последующие вызовы не обходились так дорого. Но вы должны быть внимательны и гарантировать, чтобы сохраненное значение не устарело. Необходимо разработать стратегии удаления всех сохраненных значений, когда на дереве выполняются какие-либо операции. Для этого, возможно, понадобится дать дочерним объектам ссылки на их родительские объекты.

И наконец — замечание по поводу сохраняемости. Шаблон Composite довольно изящен, но не слишком пригоден для сохранения в реляционной базе данных. Причина в том, что по умолчанию вы обращаетесь ко всей структуре только с помощью серии ссылок. Поэтому, чтобы естественным образом построить структуру Composite на основе базы данных, вам придется делать множество "дорогостоящих" запросов. Можно решить эту проблему, присвоив идентификатор ID всему дереву, чтобы все компоненты можно было извлекать из базы данных за один раз. Но, получив все объекты, нам по-прежнему нужно воссоздавать ссылки "родительский объект–дочерний объект", которые, в свою очередь, должны сохраняться в базе данных. Это нетрудно, но достаточно неприятно.

Как уже говорилось, шаблоны типа Composite трудно сохранять в реляционной базе данных, но зато они отлично подходят для сохранения в формате XML. Это объясняется тем, что XML-элементы, как правило, сами состоят из деревьев вложенных элементов.

Выводы о шаблоне Composite

Итак, шаблон Composite полезен, когда нужно обращаться с набором объектов так же, как с отдельным объектом, либо потому, что набор по своей сути такой же, как компонент (например, армии и лучники), либо потому, что контекст придает набору такие же характеристики, как компоненту (например, строки в счете-фактуре). Шаблоны Composite организованы в виде деревьев, поэтому операция на целом дереве может затронуть его части, и данные частей очевидным образом доступны для всего целого. Шаблон Composite делает такие операции и запросы прозрачными для клиентского кода. К тому же деревья легко обходить (как мы увидим в следующей главе). К структурам типа Composite легко добавлять новые типы компонентов.

Недостаток шаблона Composite в том, что он зависит от сходства своих частей. Как только мы введем сложные правила в отношении того, какие объекты-композиты какие наборы компонентов могут содержать, управлять кодом станет трудно.

Шаблоны Composite не слишком пригодны для сохранения в реляционной базе данных, но зато они отлично приспособлены для сохранения в XML-формате.

Шаблон Decorator

В то время как шаблон Composite помогает создать гибкое представление из набора компонентов, шаблон Decorator использует сходную структуру, чтобы помочь модифицировать функции конкретных компонентов. И опять-таки, основа этого шаблона — важность композиции во время выполнения. Наследование — это хороший способ построения характеристик, определяемых родительским классом. Но это может привести к жесткому кодированию вариантов в иерархиях наследования, что, как правило, приводит к потере гибкости.

Проблема

Встраивание всех функций в структуру наследования может привести к бурному росту классов в системе. Хуже того, попытавшись применить аналогичные изменения к разным ветвям дерева наследования, скорее всего, вы увидите, что появляется дублирование.

Давайте вернемся к нашей игре. Определим класс `Tile` и его производный тип.

```
abstract class Tile {
   abstract function getWealthFactor();
}

class Plains extends Tile {
   private $wealthfactor = 2;

   function getWealthFactor() {
      return $this->wealthfactor;
   }
}
```

Мы определили класс `Tile`, представляющий собой квадрат, в котором могут находиться боевые единицы. У каждой клетки есть определенные характеристики. В данном примере мы определили метод `getWealthFactor()`, влияющий на доход, который может генерировать определенная клетка, если ею владеет игрок. Как видите, у объектов типа `Plains` коэффициент богатства равен 2. Но, конечно, в клетках могут быть и другие характеристики. Они также могут хранить ссылку на информацию об изображении так, чтобы игровое поле можно было нарисовать. И снова постараемся все сделать просто.

Нам нужно модифицировать поведение объекта `Plains` (равнины), чтобы работать с информацией о природных ресурсах и губительном воздействии на природу человеческого фактора. Мы хотим смоделировать месторождения алмазов на местности, а также вред, наносимый загрязнением окружающей среды. Один из подходов — воспользоваться наследованием от объекта `Plains`.

```
class DiamondPlains extends Plains {
   function getWealthFactor() {
      return parent::getWealthFactor() + 2;
   }
}

class PollutedPlains extends Plains {
```

```
    function getWealthFactor() {
        return parent::getWealthFactor() - 4;
    }
}
```

Теперь можно легко получить загрязненную клетку.

```
$tile = new PollutedPlains();
print $tile->getWealthFactor();
```

Диаграмма класса для этого примера показана на рис. 10.3.

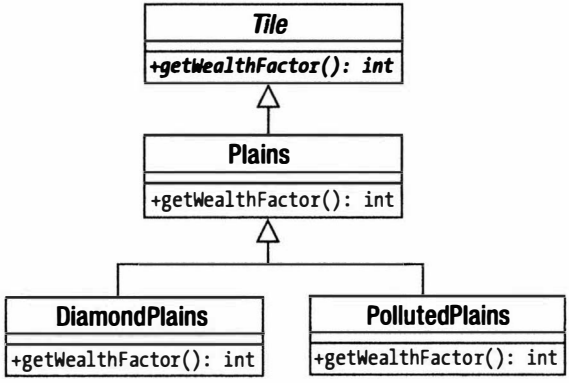


Рис. 10.3. Построение вариации на основе дерева наследования

Очевидно, что этой структуре недостает гибкости. Мы можем получить равнины с алмазами. Мы можем получить 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 тоже могут быть алмазы и загрязнения.

Конечно, это пример самого крайнего случая, но он хорошо иллюстрирует суть дела. Если при определении функций полагаться только на наследование, это приведет к увеличению количества классов и появится тенденция дублирования.

А теперь давайте рассмотрим дежурный пример. Серьезным веб-приложениям обычно нужно выполнить ряд действий после поступления запроса, до того как инициируется задача для формирования ответа на запрос. Например, нужно аутентифицировать пользователя и зарегистрировать запрос в журнале. Вероятно, мы должны как-то обработать запрос, чтобы создать структуру данных на основе необработанных исходных данных. И наконец мы должны осуществить основную обработку данных. Перед нами встала та же проблема.

Мы можем расширить функциональность на основе класса `ProcessRequest` с дополнительной обработкой в производном классе `LogRequest`, в классе `StructureRequest` и в классе `AuthenticateRequest`. Эта иерархия классов показана на рис. 10.4.

Но что произойдет, если нам нужно будет осуществить запись в журнальный файл и аутентификацию, но не подготовку данных? Мы создадим класс `LogAndAuthenticateProcessor`? Очевидно, настало время найти более гибкое решение.

Реализация

Вместо того чтобы для решения проблемы меняющейся функциональности использовать только наследование, в шаблоне Decorator используются композиция и

делегирование. В сущности, в классах Decorator хранится экземпляр другого класса его собственного типа. В классе Decorator реализуется собственно процесс выполнения операции и вызывается аналогичная операция на объекте, на который у него есть ссылка (до или после выполнения собственных действий). Таким образом, во время выполнения программы можно создать конвейер объектов типа Decorator.

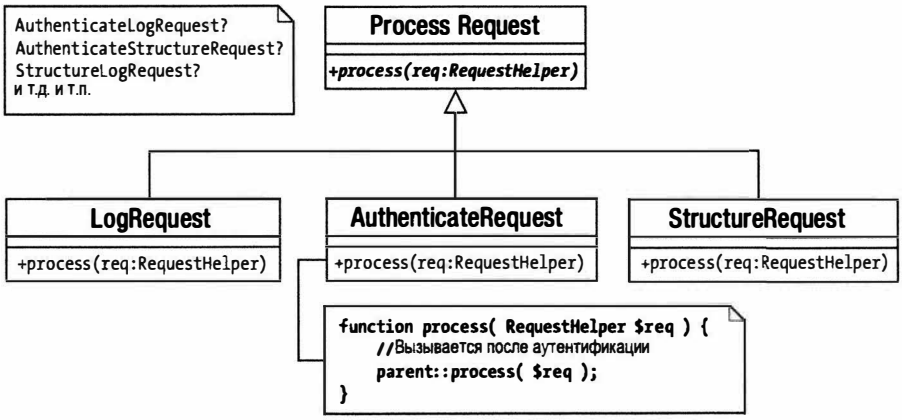


Рис. 10.4. Дополнительные жестко закодированные вариации

Чтобы проиллюстрировать это, давайте перепишем наш пример с игрой.

```
abstract class Tile {
   abstract function getWealthFactor();
}

class Plains extends Tile {
   private $wealthfactor = 2;

   function getWealthFactor() {
      return $this->wealthfactor;
   }
}

abstract class TileDecorator extends Tile {
   protected $tile;

   function __construct( Tile $tile ) {
      $this->tile = $tile;
   }
}
```

Итак, мы объявили классы `Tile` и `Plains`, как раньше, и ввели новый класс `TileDecorator`. В нем не реализуется метод `getWealthFactor()`, поэтому он должен быть объявлен абстрактным. Мы определили конструктор, которому передается объект типа `Tile`, и ссылка на него сохраняется в свойстве `$tile`. Мы сделали это свойство защищенным, т.е. `protected`, чтобы дочерние классы могли получать к нему доступ. Давайте переопределим классы `Pollution` и `Diamond`.

```
class DiamondDecorator extends TileDecorator {
   function getWealthFactor() {
      return $this->tile->getWealthFactor()+2;
   }
```

```
}

class PollutionDecorator extends TileDecorator {
   function getWealthFactor() {
      return $this->tile->getWealthFactor()-4;
   }
}
```

Каждый из этих классов расширяет `TileDecorator`. Это означает, что у них есть ссылка на объект типа `Tile`. Когда вызывается метод `getWealthFactor()`, каждый из этих классов сначала вызывает такой же метод у объекта типа `Tile`, а затем выполняет собственную корректировку значения.

Используя композицию и делегирование подобным образом, мы легко можем комбинировать объекты во время выполнения. Поскольку все объекты в шаблоне расширяют класс `Tile`, клиентскому коду не нужно знать, с какой комбинацией объектов он работает. Можно быть уверенным, что метод `getWealthFactor()` доступен для любого объекта типа `Tile`, независимо от того, перекрыт он "за сценой" другим объектом или нет.

```
$tile = new Plains();
print $tile->getWealthFactor(); // Возвращается 2
```

Поскольку класс типа `Plains` является компонентом системы, он просто возвращает значение `2`.

```
$tile = new DiamondDecorator( new Plains() );
print $tile->getWealthFactor(); // Возвращается 4
```

В объекте типа `DiamondDecorator` хранится ссылка на объект типа `Plains`. Перед прибавлением собственного значения `2` он вызывает метод `getWealthFactor()` объекта типа `Plains`.

```
$tile = new PollutionDecorator(
             new DiamondDecorator( new Plains() ));
print $tile->getWealthFactor(); // Возвращается 0
```

В объекте типа `PollutionDecorator` хранится ссылка на объект типа `Diamond Decorator`, а у того — собственная ссылка на другой объект типа `Tile`.

Диаграмма класса для этого примера показана на рис. 10.5.

Эту модель очень легко расширять. Вы можете очень легко добавлять новые объекты Decorator и компоненты. Имея множество объектов Decorator, можно строить очень гибкие структуры во время выполнения программы. Класс компонентов системы (`Plains` в данном примере) можно существенно модифицировать совершенно разными способами, при этом не встраивая всю совокупность модификаций в иерархию классов. Проще говоря, это означает, что можно создать загрязненную равнину (объект `Plains`) с месторождением алмазов, не создавая при этом объект `PollutedDiamondPlains`.

В шаблоне Decorator строятся конвейеры, которые очень удобны для создания фильтров. В пакете `java.io` очень эффективно используются классы Decorator. Программист, разрабатывающий клиентский код, может комбинировать объекты Decorator с основными компонентами, чтобы добавить фильтрацию, буферизацию, сжатие и так далее к основным методам, таким как `read()`. Наш пример веб-запроса также может быть преобразован в конфигурируемый конвейер. Вот пример простой реализации, в которой используется шаблон Decorator.

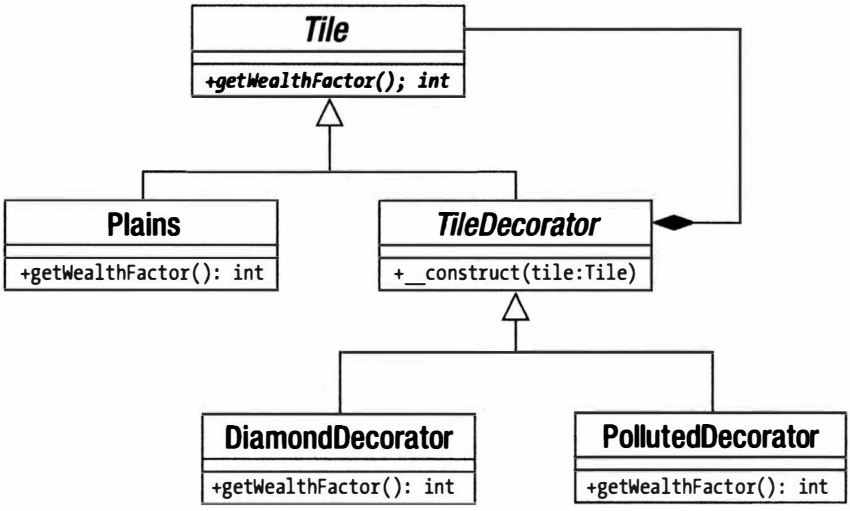


Рис. 10.5. Шаблон Decorator в действии

```
class RequestHelper{}

abstract class ProcessRequest {
   abstract function process( RequestHelper $req );
}

class MainProcess extends ProcessRequest {
   function process( RequestHelper $req ) {
      print __CLASS__ . ": выполнение запроса \n";
   }
}

abstract class DecorateProcess extends ProcessRequest {
   protected $processrequest;

   function __construct( ProcessRequest $pr ) {
      $this->processrequest = $pr;
   }
}
```

Как и раньше, мы определяем абстрактный суперкласс (`ProcessRequest`), конкретный компонент (`MainProcess`) и абстрактный декоратор (`DecorateProcess`). Метод `MainProcess::process()` не делает ничего, только сообщает, что он был вызван. В классе `DecorateProcess` сохраняется ссылка на объект типа `ProcessRequest` от имени его дочерних объектов. Вот примеры простых классов конкретных декораторов.

```
class LogRequest extends DecorateProcess {
   function process( RequestHelper $req ) {
      print __CLASS__.": регистрация запроса \n";
      $this->processrequest->process( $req );
   }
}
```

```
class AuthenticateRequest extends DecorateProcess {
   function process( RequestHelper $req ) {
      print __CLASS__.": аутентификация запроса \n";
      $this->processrequest->process( $req );
   }
}

class StructureRequest extends DecorateProcess {
   function process( RequestHelper $req ) {
      print __CLASS__.": упорядочение данных запроса \n";
      $this->processrequest->process( $req );
   }
}
```

Каждый метод `process()` выводит сообщение, прежде чем вызвать собственный метод `process()` объекта `ProcessRequest`, на который ссылаются. Теперь можно комбинировать объекты-экземпляры этих классов во время выполнения программы, чтобы создать фильтры, выполняющие различные действия по запросу, причем в различном порядке. Вот пример кода комбинирования объектов из всех этих конкретных классов в одном фильтре.

```
$process = new AuthenticateRequest(
              new StructureRequest(
                 new LogRequest (
                    new MainProcess()
           )));
$process->process( new RequestHelper() );
```

В результате получим следующее.

```
AuthenticateRequest: аутентификация запроса
StructureRequest: упорядочение данных запроса
LogRequest: регистрация запроса
MainProcess: выполнение запроса
```

На заметку. По сути, мы привели пример шаблона корпоративных приложений Intercepting Filter. Этот шаблон описан в книге *Core J2EE Patterns* .

Выводы

Как и шаблон Composite, шаблон Decorator может показаться сложным для понимания. Важно помнить, что и композиция, и наследование вступают в действие одновременно. Поэтому `LogRequest` наследует свой интерфейс от `ProcessRequest`, но, в свою очередь, выступает в качестве оболочки для другого объекта типа `ProcessRequest`.

Поскольку объект-декоратор формирует оболочку вокруг дочернего объекта, очень важно поддерживать интерфейс настолько неплотным, насколько это возможно. Если мы создадим базовый класс с множеством функций, то объекты-декораторы вынуждены будут делегировать эти функции всем общедоступным методам в объекте, который они содержат. Это можно сделать в абстрактном классе-декораторе, но в результате получится такая тесная связь, которая может привести к ошибкам.

Некоторые программисты создают декораторы, не разделяющие общий тип с объектами, которые они модифицируют. Пока они работают в рамках того же ин-

терфейса, что и данные объекты, описанная стратегия эффективна. При этом можно извлекать преимущества из того, что есть возможность использовать встроенные методы-перехватчики для автоматизации делегирования (реализуя метод `__call()` для перехвата вызовов несуществующих методов и вызывая этот же метод на дочернем объекте автоматически). Но в результате вы теряете в безопасности, которую обеспечивает проверка типа класса. До сих пор в наших примерах клиентский код в своем списке аргументов мог требовать объект типа `Tile` или `ProcessRequest` и быть уверенным в его интерфейсе независимо от того, является ли этот объект сильно "декорированным".

Шаблон Facade

Возможно, вы уже когда-то встраивали системы сторонних компаний-разработчиков в свои проекты. И независимо от того, какой это код — объектно-ориентированный или нет, он, как правило, очень сложный, длинный и непонятный. А ваш код, в свою очередь, может стать проблемой для программиста клиентского кода, которому нужно всего лишь получить доступ к нескольким функциям. Шаблон Facade — это простой способ предоставить простой и понятный интерфейс для сложных систем.

Проблема

Обычно в процессе проектирования системы длина ее кода, который на самом деле полезен только в пределах самой системы, постепенно увеличивается. В хорошо спроектированных системах разработчики с помощью классов определяют понятный общедоступный интерфейс и прячут поглубже внутреннее содержание системы. Но не всегда очевидно, какие части системы должны использоваться в клиентском коде, а какие лучше спрятать.

Работая с такими подсистемами, как веб-форумы или приложения для создания галерей, вы, возможно, делали вызовы глубоко в логику кода. Если код подсистемы должен со временем меняться, а ваш код взаимодействует с ним в самых разных точках, то по мере развития подсистемы у вас возникнет серьезная проблема с поддержкой вашего кода.

Аналогично, когда вы создаете собственные системы, имеет смысл разнести отдельные части на разные уровни. Как правило, первый уровень отвечает за логику приложения, второй — за взаимодействие с базой данных, третий — за представление данных и т.п. 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, то трудно будет достичь этой цели.

Приведем пример специально запутанного процедурного кода, который из простой задачи получения информации из текстовых файлов и преобразования ее в данные объекта делает что-то очень сложное.

```
function getProductFileLines( $file ) {
   return file( $file );
}

function getProductObjectFromId( $id, $productname ) {
// Выполняем запрос к базе данных
   return new Product( $id, $productname );
}
```

```
function getNameFromLine( $line ) {
   if ( preg_match( "/.*-(.*)\s\d+/", $line, $array ) ) {
      return str_replace( '_',' ', $array[1] );
   }
   return '';
}

function getIDFromLine( $line ) {
   if ( preg_match( "/^(\d{1,3})-/", $line, $array ) ) {
      return $array[1];
   }
   return -1;
}

class Product {
   public $id;
   public $name;

   function __construct( $id, $name ) {
      $this->id = $id;
      $this->name = $name;
   }
}
```

Давайте представим, что внутреннее содержимое этого кода сложнее, чем оно есть на самом деле, и нам приходится использовать его, вместо того чтобы переписать заново. Чтобы превратить файл

```
234-женский_свитер 55
532-мужская_шляпа 44
```

в массив объектов, мы должны вызвать все эти функции (ради краткости мы не извлекаем последнее число, представляющее собой цену).

```
$lines = getProductFileLines( 'test.txt' );
$objects = array();
foreach ( $lines as $line ) {
   $id   = getIDFromLine  ( $line );
   $name = getNameFromLine( $line );
   $objects[$id] = getProductObjectFromID( $id, $name );
}
```

Если мы будем вызывать эти функции непосредственно, как здесь, во всем проекте, то наш код будет плотно "вплетен" в подсистему, которую он использует. Это может стать причиной проблем, если подсистема изменится или если мы решим полностью от нее отказаться. Поэтому нам обязательно нужно создать шлюз между системой и остальным кодом.

Реализация

Приведем пример простого класса, который предоставляет интерфейс для процедурного кода, с которым мы уже встречались в предыдущем разделе.

```
class ProductFacade {
   private $products = array();

   function __construct( $file ) {
```

```
        $this->file = $file;
        $this->compile();
    }

    private function compile() {
        $lines = getProductFileLines( $this->file );
        foreach ( $lines as $line ) {
            $id   = getIDFromLine  ( $line );
            $name = getNameFromLine( $line );
            $this->products[$id] = getProductObjectFromID( $id, $name );
        }
    }

    function getProducts() {
        return $this->products;
    }


    function getProduct( $id ) {
        if ( isset( $this->products[$id] ) ) {
            return $this->products[$id];
        }
        return null;
    }
}
```

С точки зрения клиентского кода теперь доступ к объектам `Product` из текстового файла намного упрощен.

```
$facade = new ProductFacade( 'test.txt' );
$facade->getProduct( 234 );
```

Выводы

В основе шаблона Facade на самом деле лежит очень простая идея. Это всего лишь вопрос создания одной точки входа для уровня или подсистемы в целом. В результате 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 вещи. Это позволяет сократить количество ошибок, сосредоточив обращение к подсистеме в одном месте, так что изменения в этой подсистеме вызовут сбой в предсказуемом месте. Классы Facade также минимизируют ошибки в комплексных подсистемах, где клиентский код, в противном случае, мог бы некорректно использовать внутренние функции.

Несмотря на простоту шаблона Facade, очень легко забыть воспользоваться им, особенно если вы знакомы с подсистемой, с которой работаете. Но, конечно, тут необходимо найти нужный баланс. С одной стороны, преимущества создания простых интерфейсов для сложных систем очевидны. С другой стороны, можно необдуманно разделить системы, а затем разделить разделения. Если вы осуществляете значительные упрощения для пользы клиентского кода и/или экранируете его от систем, которые могут изменяться, то, вероятно, есть 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 используется для поддержки композиции и определения общего интерфейса, который обеспечивает гарантии для клиентского кода.

Мы также видели, что в этих шаблонах эффективно используется делегирование. И наконец мы рассмотрели простой, но очень эффективный шаблон Facade. Это один из шаблонов, которыми программисты пользуются годами, не зная, что он так называется. Facade позволяет создать удобную и понятную точку входа для уровня или подсистемы. В PHP шаблон Facade также используется для создания объектов-оболочек, в которые инкапсулированы блоки процедурного кода.

Глава 11

Выполнение задач и представление результатов

В этой главе мы начнем действовать активно. Рассмотрим шаблоны, которые помогут выполнить поставленные задачи, такие как интерпретация мини-языка или инкапсулирование алгоритма.

В главе будут рассмотрены следующие темы.

- *Шаблон Interpreter:* построение интерпретатора мини-языка, используемого для создания приложений, которые можно записать на языке сценариев.
- *Шаблон Strategy:* определение алгоритмов в системе и их инкапсулирование в собственные типы.
- *Шаблон Observer:* создание перехватчиков для оповещения несовместимых объектов о событиях в системе.
- *Шаблон Visitor:* применение операции ко всем узлам на дереве объектов.
- *Шаблон Command:* создание командных объектов, которые можно сохранять и передавать.

Шаблон Interpreter

Компиляторы и интерпретаторы языков программирования пишут на других языках программирования (по крайней мере, так было сначала). Например, PHP написан на языке C. Точно так же, как это ни покажется странным, с помощью PHP можно определить и создать компилятор собственного языка. Конечно, любой язык, который мы создадим, будет работать медленно и окажется в некоторой степени ограниченным. Тем не менее мини-языки могут быть очень полезны, как мы увидим в этой главе.

Проблема

Создавая на PHP веб-приложения или приложения командной строки, мы, по сути, предоставляем пользователю доступ к функциям. При разработке интерфейса приходится искать компромисс между функциональностью и простотой использования. Как правило, чем больше функций и возможностей вы предоставляете пользователю, тем более усложненным и запутанным становится интерфейс. Конечно, если хорошо спроектировать интерфейс, это поможет делу. Но если 90%

пользователей используют один и тот же набор функций (примерно треть от общего состава), то вряд ли есть смысл увеличивать функциональность нашего приложения. Возможно, вы даже решите упростить систему в расчете на большинство пользователей. Но как быть с 10% пользователей, которые используют более сложные возможности вашей системы? Вероятно, вы можете помочь им как-то иначе. Предложив таким пользователям предметно-ориентированный язык программирования (который обычно называют DSL, или Domain Specific Language), вы можете реально расширить функциональность приложения.

На самом деле у нас уже есть язык программирования под названием "PHP". Вот как можно позволить пользователям создавать на нем сценарии в нашей системе.

```
$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 не очевидно, почему, то на это есть две причины: безопасность и сложность. Проблема безопасности хорошо проиллюстрирована в нашем примере. Позволяя пользователям выполнять PHP-код через наш сценарий, мы, по сути, даем им доступ к серверу, на котором запущен сценарий. Проблема сложности настолько же серьезна. Независимо от того, насколько понятен ваш код, маловероятно, что среднестатистический пользователь легко сможет его расширить, особенно из окна браузера.

Но мини-язык может помочь в решении обеих этих проблем. Вы можете сделать его гибким, сократить возможности пользователя что-либо нарушить и сфокусироваться на самом необходимом.

Давайте представим себе приложение для разработки викторин. Создатели придумывают вопросы и устанавливают правила оценки ответов конкурсантов. Существует требование, что ответы на вопросы викторины должны оцениваться без вмешательства человека, хотя некоторые ответы пользователи могут вводить в текстовые поля.

Вот пример вопроса.

```
Сколько членов в группе разработки проектных шаблонов?
```

Мы можем принимать "четыре" или "4" в качестве правильных ответов. Нам потребуется создать веб-интерфейс, позволяющий создателям использовать регулярное выражение для оценки ответов.

```
^4|четыре$
```

Но, к сожалению, не все создатели владеют регулярными выражениями. Поэтому, чтобы облегчить всем жизнь, мы можем реализовать более дружественный пользователю механизм оценки ответов.

```
$input equals "4" or $input equals "четыре"
```

Мы предлагаем язык, который поддерживает переменные, оператор `equals` и булеву логику (операторы `or` и `and`). Программисты любят все называть, поэтому давайте назовем наш язык MarkLogic. Этот язык должен легко расширяться, поскольку мы предвидим множество запросов на более сложные функции. Давайте пока оставим в стороне вопрос анализа входных данных и сосредоточимся на механизме подключения этих элементов вместе во время выполнения программы для генерации ответа. Вот тут, как и можно было ожидать, нам и пригодится шаблон Interpreter.

Реализация

Наш язык состоит из выражений, вместо которых подставляются некоторые значения. Как видно из табл. 11.1, даже в таком небольшом языке, как MarkLogic, необходимо отслеживать множество элементов.

Таблица 11.1. Элементы грамматики языка MarkLogic

Описание	Имя EBNF	Имя класса	Пример
Переменная	`variable`	`VariableExpression`	`$input`
Строковый литерал	`<stringLiteral>`	`LiteralExpression`	`"четыре"`
Булево "И"	`andExpr`	`BooleanAndExpression`	`$input equals '4' and $other equals '6'`
Булево "Или"	`orExpr`	`BooleanOrExpression`	`$input equals '4' or $other equals '6'`
Проверка равенства	`equalsExpr`	`EqualsExpression`	`$input equals '4'`

В табл. 11.1 приведены имена EBNF. А что такое EBNF? Это — обозначение, которое можно использовать для описания грамматики языка. EBNF расшифровывается как "Extended Backus-Naur Form" — расширенная форма Бэкуса-Наура. Она состоит из набора строк, которые называются *продукциями (productions)*. 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). Вот как можно описать нашу грамматику с помощью EBNF.

```
expr ::= operand (orExpr | andExpr )*
operand ::= ( '(' expr ')' | <stringLiteral> | variable ) ( eqExpr )*
orExpr ::= 'or' operand
andExpr ::= 'and' operand
eqExpr ::= 'equals' operand
variable ::= '$' <word>
```

Некоторые символы несут специальный смысл (они должны быть вам знакомы по регулярным выражениям), например `'*'` означает "нуль или больше", а `'|'` — "или". Группировать элементы можно с помощью скобок. Так, в приведенном примере выражение (`expr`) состоит из лексемы `operand`, за которой следует нуль или более лексем либо `orExpr`, либо `andExpr`. В качестве `operand` может быть выражение в скобках, строковая константа (я опустил продукцию для него) или переменная, за которой следует нуль или больше лексем `eqExpr`. Как только вы поймете, как ссылаться с одной продукции на другую, EBNF станет очень простой формой для прочтения.

На рис. 11.1 элементы нашей грамматики представлены в виде диаграммы классов.

Как видите, класс `BooleanAndExpression` и его "братья" наследуют класс `Operator Expression`. Причина в том, что все эти классы выполняют операции над другими объектами типа `Expression`. Объекты `VariableExpression` и `LiteralExpression` оперируют непосредственно значениями.

Во всех объектах типа `Expression` реализован метод `interpret()`, который определен в абстрактном базовом классе `Expression`. Методу `interpret()` передается объект типа `InterpreterContext`, который используется как совместно используемое хранилище данных. Каждый объект типа `Expression` может сохранять данные в объекте типа `InterpreterContext`. Объект `InterpreterContext` затем передается другим объектам типа `Expression`. Поэтому, чтобы данные было легко извлекать из объекта `InterpreterContext`, в базовом классе `Expression` реализован метод `getKey()`, возвращающий уникальный дескриптор. Давайте посмотрим, как это работает на практике, на примере реализации класса `Expression`.

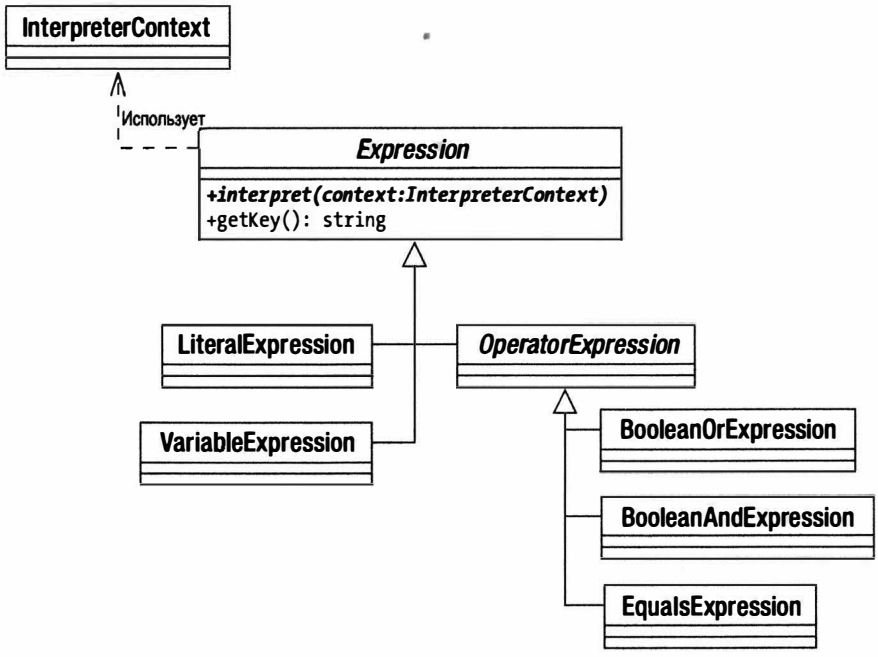


Рис. 11.1. Классы шаблона Interpreter, из которых состоит язык MarkLogic

```
abstract class Expression {
   private static $keycount=0;
   private $key;

   abstract function interpret( InterpreterContext $context );

   function getKey() {
      if ( ! isset( $this->key ) ) {
         self::$keycount++;
         $this->key=self::$keycount;
      }
      return $this->key;
   }
}

class LiteralExpression extends Expression {
   private $value;

   function __construct( $value ) {
      $this->value = $value;
}

   function interpret( InterpreterContext $context ) {
      $context->replace( $this, $this->value );
   }
}

class InterpreterContext {
```

```
    private $expressionstore = array();

    function replace( Expression $exp, $value ) {
        $this->expressionstore[$exp->getKey()] = $value;
    }

    function lookup( Expression $exp ) {
        return $this->expressionstore[$exp->getKey()];
    }
}

$context = new InterpreterContext();
$literal = new LiteralExpression( 'Четыре');
$literal->interpret( $context );
print $context->lookup( $literal ) . "\n";
```

И вот что мы получим в результате на выходе.

```
Четыре
```

Начнем с класса `InterpreterContext`. Как видите, на самом деле он представляет собой только внешний интерфейс для ассоциативного массива `$expressionstore`, который мы используем для хранения данных. Методу `replace()` передаются ключ и значение, которые сохраняются в ассоциативном массиве `$expressionstore`. В качестве ключа используется объект типа `Expression`, а значение может быть любого типа. В классе `InterpreterContext` реализован также метод `lookup()` для извлечения данных.

В классе `Expression` определены абстрактный метод `interpret()` и конкретный метод `getKey()`, оперирующий статическим значением счетчика; 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 аргумент-значение. Методу `interpret()` нужно передать объект типа `InterpreterContext`. В нем просто вызывается метод `replace()` этого объекта, которому передаются ключ (ссылка на сам объект типа `LiteralExpression`) и значение `$value`. В методе `replace()` объекта `InterpreterContext` для определения численного значения ключа используется метод `getKey()`. По мере знакомства с другими классами типа `Expression` подобный шаблонный подход покажется вам знакомым. Метод `interpret()` всегда сохраняет свои результаты в объекте `InterpreterContext`.

В приведенный выше пример включен также фрагмент клиентского кода, в котором создаются экземпляры объектов `InterpreterContext` и `LiteralExpression` (со значением `'Четыре'`). Объект типа `InterpreterContext` передается методу `LiteralExpression::interpret()`. Метод `interpret()` сохраняет пару "ключ–значение" в объекте `InterpreterContext`, из которого мы затем извлекаем значение, вызывая метод `lookup()`.

Давайте определим оставшийся конечный класс. Класс `VariableExpression` немного сложнее.

```
class VariableExpression extends Expression {
    private $name;
    private $val;

    function __construct( $name, $val=null ) {
        $this->name = $name;
```

```
        $this->val = $val;
    }

    function interpret( InterpreterContext $context ) {
        if ( ! is_null( $this->val ) ) {
            $context->replace( $this, $this->val );
            $this->val = null;
        }
    }

    function setValue( $value ) {
        $this->val = $value;
    }

    function getKey() {
        return $this->name;
    }
}
$context = new InterpreterContext();
$myvar = new VariableExpression( 'input', 'Четыре');
$myvar->interpret( $context );
print $context->lookup( $myvar ). "\n";
// Выводится: 'Четыре'

$newvar = new VariableExpression( 'input' );
$newvar->interpret( $context );
print $context->lookup( $newvar ). "\n";
// Выводится: 'Четыре'

$myvar->setValue("Пять");
$myvar->interpret( $context );
print $context->lookup( $myvar ). "\n";
// Выводится: 'Пять'
print $context->lookup( $newvar ) . "\n";
// Выводится: 'Пять'
```

Конструктору класса `VariableExpression` передаются два аргумента (имя и значение), которые сохраняются в свойствах объекта. В классе реализован метод `setValue()`, чтобы клиентский код мог изменить значение переменной в любое время.

Метод `interpret()` проверяет, имеет ли свойство `$val` ненулевое значение. Если у свойства `$val` есть некоторое значение, то его значение сохраняется в объекте `InterpreterContext`. Затем мы устанавливаем для свойства `$val` значение `null`. Это делается для того, чтобы повторный вызов метода `interpret()` не испортил значение переменной с тем же именем, сохраненной в объекте `InterpreterContext` другим экземпляром объекта `VariableExpression`. Возможности нашей переменной достаточно ограничены, так как ей могут быть присвоены только строковые значения. Если бы мы собирались расширить наш язык, то нужно было бы сделать так, чтобы он работал с другими объектами типа `Expression`, содержащими результаты выполнения булевых и других операций. Но пока `VariableExpression` будет делать то, что нам от него нужно. Обратите внимание на то, что мы заменили метод `getKey()`, чтобы значения переменных были связаны с именем переменной, а не с произвольным статическим идентификатором.

В нашем языке все операторные выражения работают с двумя другими объектами типа `Expression`. Поэтому есть смысл, чтобы они расширяли общий суперкласс. Вот определение класса `OperatorExpression`.

```
abstract class OperatorExpression extends Expression {
   protected $l_op;
   protected $r_op;

   function __construct( Expression $l_op, Expression $r_op ) {
      $this->l_op = $l_op;
      $this->r_op = $r_op;
   }

   function interpret( InterpreterContext $context ) {
      $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 );
   }

   protected abstract function doInterpret( InterpreterContext $context,
                                            $result_l, $result_r );
}
```

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, сохраненных в свойствах (если вы читали предыдущую главу, то, наверное, заметили, что здесь мы воспользовались экземпляром шаблона 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 нужно выполнить над полученными значениями операндов.

На заметку. Метод `doInterpret()` представляет собой экземпляр шаблона Template Method. В этом шаблоне в родительском классе и 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 четких и понятных целях.

Вот определение класса `EqualsExpression`, который проверяет равенство двух объектов типа `Expression`.

```
class EqualsExpression extends OperatorExpression {
   protected function doInterpret( InterpreterContext $context,
                                   $result_l, $result_r ) {
      $context->replace( $this, $result_l == $result_r );
   }
}
```

В классе `EqualsExpression` реализован только метод `doInterpret()`. В нем проверяется равенство значений двух операндов, переданных из метода `interpret()` и полученных из объекта `InterpreterContext`.

Чтобы завершить набор классов типа `Expression`, приведем определение классов `BooleanOrExpression` и `BooleanAndExpression`.

```
class BooleanOrExpression extends OperatorExpression {
   protected function doInterpret( InterpreterContext $context,
                                   $result_l, $result_r ) {
      $context->replace( $this, $result_l || $result_r );
   }
}

class BooleanAndExpression extends OperatorExpression {
   protected function doInterpret( InterpreterContext $context,
                                   $result_l, $result_r ) {
      $context->replace( $this, $result_l && $result_r );
   }
}
```

Вместо проверки на равенство в классе `BooleanOrExpression` используется операция логического "ИЛИ", и полученный результат сохраняется в контексте с помощью метода `InterpreterContext::replace()`. В классе `BooleanAndExpression`, естественно, используется операция логического "И".

Итак, мы получили код, с помощью которого можно интерпретировать фрагмент выражения нашего мини-языка, который был приведен выше. Повторим его еще раз.

```
$input equals "4" or $input equals "Четыре"
```

Вот как можно описать этот оператор с помощью классов `Expression`.

```
$context = new InterpreterContext();
$input = new VariableExpression( 'input' );
$statement = new BooleanOrExpression(
   new EqualsExpression($input, new LiteralExpression( 'Четыре' ) ),
   new EqualsExpression($input, new LiteralExpression( '4' ) )
);
```

Мы создаем экземпляр переменной `'input'`, но пока не присваиваем ей значение. Затем создается объект типа `BooleanOrExpression`, который будет сравнивать результаты двух выражений, полученных из объектов `EqualsExpression`. В первом из этих объектов сравнивается значение объекта `VariableExpression`, который сохранен в переменной `$input`, со значением объекта `LiteralExpression`, содержащим строку `"Четыре"`. Во втором объекте сравнивается значение переменной `$input` с объектом `LiteralExpression`, содержащим строку `"4"`.

Теперь, имея подготовленный оператор, мы подошли к тому, чтобы присвоить значение для входной переменной и выполнить код.

```
foreach ( array( "Четыре", "4", "52" ) as $val ) {
   $input->setValue( $val );
   print "$val:\n";
   $statement->interpret( $context );
   if ( $context->lookup( $statement ) ) {
      print "соответствует \n\n";
   } else {
      print "не соответствует \n\n";
   }
}
```

Фактически мы запускаем код три раза, с тремя различными значениями. Но первый раз мы устанавливаем для временной переменной `$val` значение `"Четыре"`, присваивая его объекту `VariableExpression`, соответствующему переменной `$input`, с помощью его метода `setValue()`. Затем мы вызываем метод `interpret()` для объекта типа `Expression` самого верхнего уровня (объект `BooleanOrExpression`, который содержит ссылки на все другие выражения в операторе). Давайте посмотрим, что происходит внутри этого вызова.

- Для объекта `$statement` вызывается метод `interpret()` для свойства `$l_op` (первый объект типа `EqualsExpression`).
- Для первого объекта `EqualsExpression` вызывается метод `interpret()` для его свойства `$l_op` (в нем содержится ссылка на входной объект типа `Variable Expression`, для которого в настоящий момент установлено значение `"Четыре"`).
- Входной объект `VariableExpression` записывает свое текущее значение в предоставленный объект типа `InterpreterContext`, вызывая метод `Interpreter Context::replace()`.
- Для первого объекта `EqualsExpression` вызывается метод `interpret()` для его свойства `$r_op` (объекту `LiteralExpression` присвоено значение `"Четыре"`).
- Объект `LiteralExpression` регистрирует имя и значение своего ключа в объекте `InterpreterContext`.
- Первый объект `EqualsExpression` извлекает значения для `$l_op` (`"Четыре"`) и `$r_op` (`"Четыре"`) из объекта `InterpreterContext`.
- Первый объект `EqualsExpression` сравнивает эти два значения, проверяя их на равенство, и регистрирует результат (`true`, т.е. "истина") вместе с именем ключа в объекте `InterpreterContext`.
- На вершине дерева для объекта `$statement` (`BooleanOrExpression`) вызывается метод `interpret()` для его свойства `$r_op`. В результате получается значение (в данном случае — `false`, т.е. "ложь") таким же способом, как при использовании свойства `$l_op`.
- Объект `$statement` извлекает значения для каждого из своих операндов из объекта `InterpreterContext` и сравнивает их с помощью оператора `||`. Он сравнивает значения `true` и `false`, поэтому результатом будет `true`. Этот окончательный результат сохраняется в объекте `InterpreterContext`.

И все это — только первая итерация нашего цикла. А вот окончательный результат.

```
Четыре:
соответствует

4:
соответствует

52:
не соответствует
```

Возможно, вам понадобится прочитать этот раздел несколько раз, прежде чем все станет ясно. Вас может смутить здесь старая проблема — разница между деревьями классов и объектов. Классы `Expression` организованы в иерархию наследования, точно так же как объекты `Expression` объединяются в дерево во время выполнения программы. Когда будете перечитывать код, помните об этом различии.

На рис. 11.2 показана полная диаграмма классов для нашего примера.

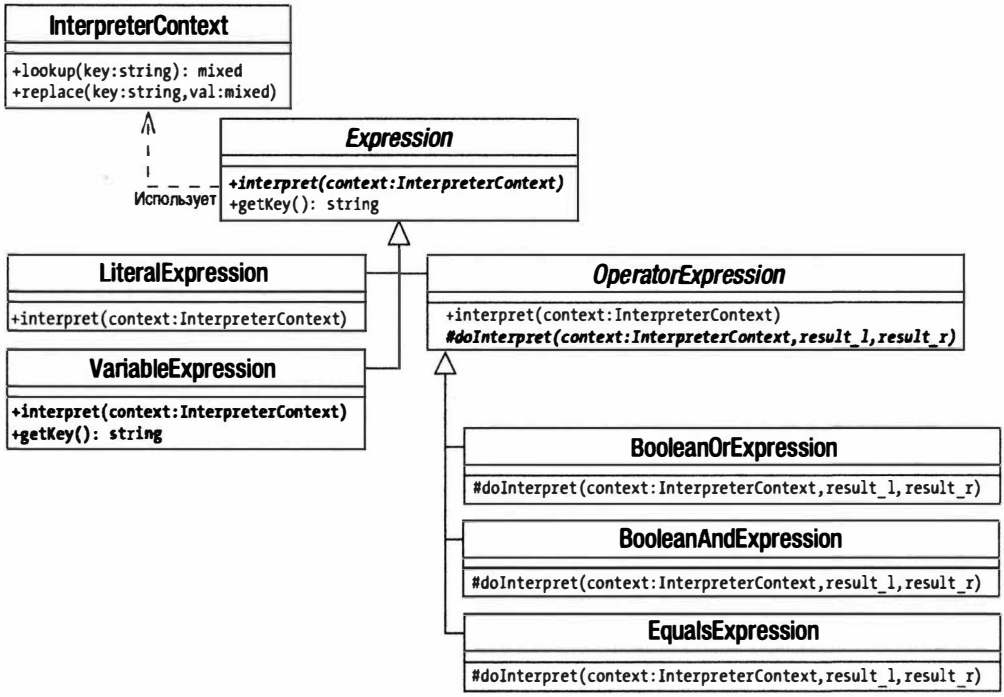


Рис. 11.2. Использование шаблона Interpreter

Проблемы шаблона Interpreter

Как только вы подготовите основные классы для реализации шаблона Interpreter, расширить его будет легко. Цена, которую за это приходится платить, — только количество классов, которые нужно создать. Поэтому шаблон Interpreter более применим для относительно небольших языков. А если вам нужен полноценный язык программирования, то лучше поискать для этой цели инструмент от сторонних фирм.

Поскольку классы Interpreter часто выполняют очень схожие задачи, стоит следить за создаваемыми классами, чтобы не допускать дублирования. Многие люди, которые в первый раз обращаются к шаблону Interpreter, после некоторых начальных экспериментов разочаровываются, обнаружив, что он не выполняет синтаксический анализ. Это означает, что мы пока не в состоянии предложить пользователям хороший дружественный язык. В приложении Б приведен примерный вариант кода, иллюстрирующего одну из стратегий синтаксического анализатора мини-языка.

Шаблон Strategy

Разработчики часто пытаются наделить классы слишком большими функциональными возможностями. И это понятно: вы создаете класс, который выполняет несколько связанных действий. Как и код, некоторые из этих действий нужно изменять в зависимости от обстоятельств. В то же время классы необходимо разбивать на подклассы. И прежде чем вы это поймете, ваш проект будет рассыпаться как карточный домик.

Проблема

Поскольку в предыдущем разделе мы создали интерпретатор небольшого языка, давайте снова воспользуемся примером с викториной. Для викторин нужны вопросы, поэтому мы создали класс `Question` и реализовали в нем метод `mark()`. Все это хорошо до тех пор, пока нам не понадобится поддерживать различные механизмы оценки.

Предположим, нас попросили поддержать простой язык MarkLogic, в котором оценка происходит путем простого сравнения, а также с помощью регулярных выражений. Наверное, ваша первая мысль — создать подклассы, учитывающие эти различия (рис. 11.3).

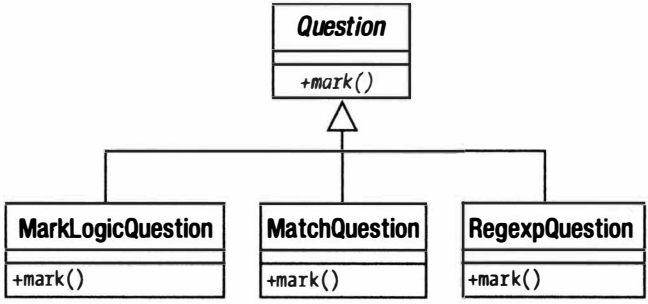


Рис. 11.3. Определение подклассов в соответствии со стратегиями оценки

Но такое положение дел будет устраивать нас до тех пор, пока оценка останется единственным меняющимся аспектом класса. Но предположим, что нас попросили поддержать различные виды вопросов, например текстовые и мультимедийные. И тогда мы столкнемся с проблемой, когда нужно будет объединить все внешние силы в одном дереве наследования (рис. 11.4).

При этом не только увеличится количество классов в иерархии, но и неизбежно возникнет повторение. Логика оценки повторяется в каждой ветви иерархии наследования.

Когда вы обнаруживаете, что алгоритм повторяется у дочерних объектов одного уровня на дереве наследования (либо в результате создания подклассов, либо в повторяющихся условных операторах), подумайте о том, чтобы выделить этот алгоритм в собственный тип.

Реализация

Как и все лучшие шаблоны, Strategy одновременно и прост, и функционален. Когда классы должны поддерживать несколько реализаций интерфейса (например, несколько механизмов оценки), то наилучший подход — это, как правило, выделить эти реализации и поместить их в собственный тип, а не расширять первоначальный класс, чтобы работать с ними.

Так, в нашем примере алгоритм оценки можно поместить в класс `Marker`. Новая структура показана на рис. 11.5.

Помните принцип "Банды четырех" — "отдавайте предпочтение композиции, а не наследованию"? Это отличный пример иллюстрации данного принципа. Определяя и инкапсулируя алгоритмы оценки, мы сокращаем количество создаваемых подклассов и увеличиваем степень гибкости. Мы можем в любой момент добавить новые стратегии оценки, причем для этого нам совершенно не нужно изменять

классы `Question`. Все, что известно классам `Question`, — то, что в их распоряжении есть экземпляр класса `Marker` и что по своему интерфейсу он гарантированно поддерживает метод `mark()`. А детали реализации — это полностью чужие проблемы.

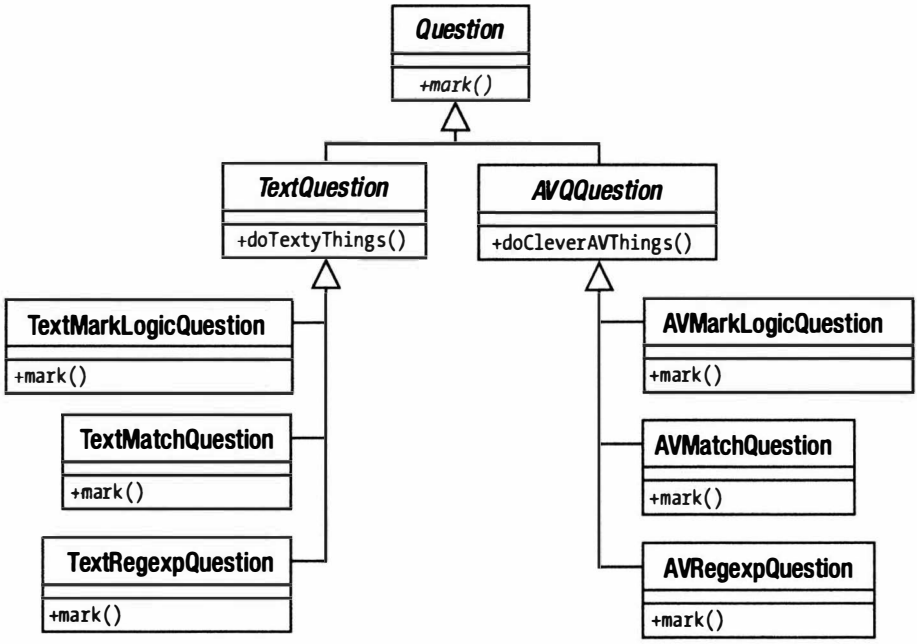


Рис. 11.4. Определение подклассов в соответствии с двумя внешними силами

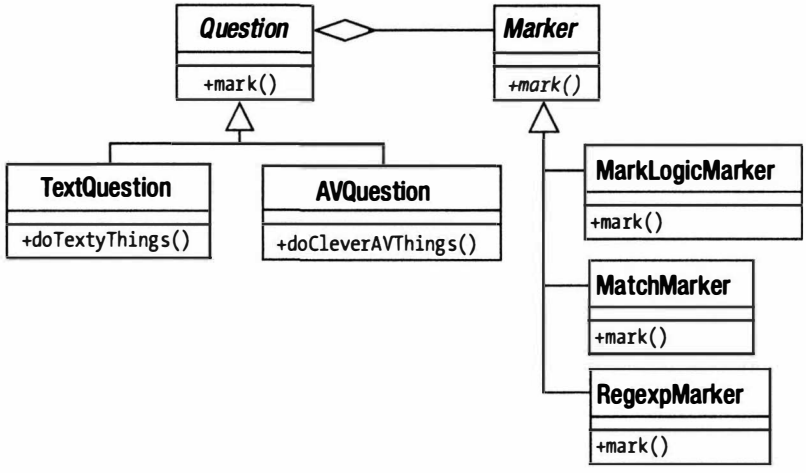


Рис. 11.5. Выделение алгоритмов в их собственный тип

Вот определение семейства классов `Question`, представленное в виде кода.

```
abstract class Question {
   protected $prompt;
   protected $marker;
```

```
    function __construct( $prompt, Marker $marker ) {
        $this->marker=$marker;
        $this->prompt = $prompt;
    }

    function mark( $response ) {
        return $this->marker->mark( $response );
    }
}

class TextQuestion extends Question {
    // Выполняются действия, специфичные для текстовых вопросов
}

class AVQuestion extends Question {
    // Выполняются действия, специфичные для мультимедийных
    // (аудио- и видео-) вопросов
}
```

Как видите, природу различия между классами `TextQuestion` и `AVQuestion` мы не определяем, оставляя это на ваш суд. 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 от конечного пользователя, этот метод просто делегирует решение проблемы своему объекту `Marker`.

Давайте определим некоторые простые объекты `Marker`.

```
abstract class Marker {
    protected $test;

    function __construct( $test ) {
        $this->test = $test;
    }

    abstract function mark( $response );
}

class MarkLogicMarker extends Marker {
    private $engine;

    function __construct( $test ) {
        parent::__construct( $test );
        // $this->engine = new MarkParse( $test );
    }

    function mark( $response ) {
        // return $this->engine->evaluate( $response );
        // Возвратим фиктивное значение
        return true;
    }
}

class MatchMarker extends Marker {
    function mark( $response ) {
        return ( $this->test == $response );
    }
```

```
}

class RegexpMarker extends Marker {
   function mark( $response ) {
      return ( preg_match( $this->test, $response ) );
   }
}
```

Наверное, здесь мало неожиданностей (если они есть вообще), связанных с самими классами `Marker`. Обратите внимание на то, что объект типа `MarkParse` предназначен для работы с простым синтаксическим анализатором, приведенным в приложении Б. Но для данного примера в этом нет необходимости, поэтому мы просто возвращаем значение `true` (истина) из метода `MarkLogicMarker::mark()`. Здесь главное — структура, которую мы определили, а не детали самих стратегий. Мы можем заменить `RegexpMarker` на `MatchMarker`, так что это не повлияет на класс `Question`.

Разумеется, вы еще должны решить, какой метод использовать для выбора между конкретными объектами типа `Marker`. На практике я видел два подхода к решению этой проблемы. В первом используются переключатели для выбора предпочитаемой стратегии оценки. Во втором используется сама структура условия оценки, при этом оператор сравнения остается простым.

```
Пять
```

Перед оператором MarkLogic ставится двоеточие.

```
:$input equals 'Пять'
```

А в регулярном выражении используются косые черты.

```
/П.ть/
```

Вот код, позволяющий проверить наши классы в деле.

```
$markers = array( new RegexpMarker( "/П.ть/" ),
                  new MatchMarker( "Пять" ),
                  new MarkLogicMarker( '$input equals "Пять"' )
                );
foreach ( $markers as $marker ) {
   print get_class( $marker ) . "\n";
   $question = new TextQuestion( "Сколько лучей у Кремлевской звезды?", $marker
);
   foreach ( array( "Пять", "Четыре" ) as $response ) {
      print "\tÎòâåò: $response: ";
      if ( $question->mark( $response ) ) {
         print "Правильно! \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 отношению к двум вариантам ответов.

Приведенный здесь класс `MarkLogicMarker` в настоящее время — это просто заглушка, и его метод `mark()` всегда возвращает значение `true` (истина). Однако закомментированный код работает с вариантом синтаксического анализатора, при-

веденным в приложении Б. Он также может работать с любым анализатором, разработанным сторонними производителями.

Вот что получается на выходе.

```
RegexpMarker
      Ответ: Пять: Правильно!
      Ответ: Четыре: Неверно!
MatchMarker
      Ответ: Пять: Правильно!
      Ответ: Четыре: Неверно!
MarkLogicMarker
      Ответ: Пять: Правильно!
      Ответ: Четыре: Правильно!
```

Не забывайте, что в данном примере класс `MarkLogicMarker` не выполняет никаких действий и всегда возвращает значение `true` (истина), поэтому он оценил оба ответа как правильные.

В данном примере мы передали данные, введенные пользователем (они содержатся в переменной `$response`), объекту-оценщику с помощью метода `mark()`. Но иногда вы будете сталкиваться с ситуациями, когда заранее не всегда известно, сколько информации потребует объект-оценщик при выполнении своей операции. Поэтому можно делегировать решения относительно того, какие данные получать, самому объекту-оценщику, передав ему экземпляр объекта-клиента. Затем этот объект-оценщик сам запросит у клиента необходимые ему данные.

Шаблон Observer

Преимущества ортогональности мы обсуждали раньше. Одной из наших целей как программистов должно быть построение компонентов, которые можно изменять или перемещать с минимальным воздействием на другие компоненты. Если каждое изменение, которое вы осуществляете над одним компонентом, влечет за собой необходимость изменений в других местах основного кода, то задача разработки быстро превратится в спираль создания и устранения ошибок.

Конечно, ортогональность — это, как правило, только мечта, недостижимая цель. У элементов в системе должны быть встроенные ссылки на другие элементы. Но для минимизации этого можно использовать различные стратегии. Мы видели различные примеры полиморфизма, в которых клиентский код понимает интерфейс компонента, но сам реальный компонент может меняться во время выполнения программы.

В некоторых ситуациях у вас может возникнуть необходимость вбить еще больший клин между компонентами. Рассмотрим класс, ответственный за управление доступом пользователя к системе.

```
class Login {
   const LOGIN_USER_UNKNOWN = 1;
   const LOGIN_WRONG_PASS   = 2;
   const LOGIN_ACCESS       = 3;
   private $status = array();

   function handleLogin( $user, $pass, $ip ) {
      $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;
         }
      return $isvalid;
   }

   private function setStatus( $status, $user, $ip ) {
      $this->status = array( $status, $user, $ip );
   }

   function getStatus() {
      return $this->status;
   }
}
```

Разумеется, в реальном примере метод `handleLogin()` должен проверять учетные данные пользователя, хранимые где-либо в системе. Но в данном случае этот класс имитирует процесс входа в систему с помощью функции `rand()`. Существует три возможных результата вызова метода `handleLogin()`. При этом устанавливается соответствующий код состояния: `LOGIN_ACCESS`, `LOGIN_WRONG_PASS` или `LOGIN_USER_UNKNOWN`.

Поскольку класс `Login` — это "ворота", охраняющие сокровища вашего бизнеса, он может привлекать большое внимание как во время разработки системы, так и в течение нескольких последующих месяцев. Например, вас могут вызвать в отдел маркетинга и попросить зарегистрировать IP-адреса пользователей. При этом вы можете добавить вызов соответствующего метода класса `Logger` вашей системы.

```
function handleLogin( $user, $pass, $ip ) {
   $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;
   }
   Logger::logIP( $user, $ip, $this->getStatus() );
   return $isvalid;
}
```

Заботясь о безопасности, системные администраторы могут попросить вести учет неудачных попыток входа в систему. И снова вы можете вернуться к методу управления входом в систему и добавить новый вызов.

```
if ( ! $isvalid ) {
   Notifier::mailWarning( $user, $ip, $this->getStatus() );
}
```

Команда бизнес-разработчиков может анонсировать определенный товар только для пользователей некоторых провайдеров услуг Интернета (ISP) и попросить, чтобы после входа заданных пользователей в систему им высылались cookie-файлы и т.п.

Все эти просьбы достаточно легко удовлетворить, но за счет изменения структуры нашего проекта. Класс `Login` скоро станет очень тесно встроенным в эту конкретную систему. Мы не сможем извлечь его и включить в другой продукт, не пройдя весь код строка за строкой и не удалив все, что связано со старой системой. Конечно, это не слишком сложно, но в результате мы уже не сможем просто скопировать и вставить нужный код в другую систему. Теперь, когда у нас есть два похожих, но различных класса `Login` в наших системах, мы обнаружим, что улучшение в одном классе приведет к необходимости таких же изменений в другом классе. Так будет до тех пор, пока они неизбежно не перестанут соответствовать один другому.

Так что же можно сделать, чтобы спасти класс `Login`? На помощь нам придет шаблон Observer.

Реализация

В основе шаблона Observer лежит принцип отсоединения клиентских элементов (наблюдателей) от центрального класса (субъекта). Наблюдатели должны быть проинформированы, когда происходят события, о которых знает субъект. В то же время мы не хотим, чтобы у субъекта была жестко закодированная связь с его классами-наблюдателями.

Чтобы достичь этого, мы можем разрешить наблюдателям регистрироваться у субъекта. Поэтому мы даем классу `Login` три новых метода — `attach()`, `detach()` и `notify()` — и все это вводим в действие с помощью интерфейса `Observable`.

```
interface Observable {
   function attach( Observer $observer );
   function detach( Observer $observer );
   function notify();
}

// ... Класс Login
class Login implements Observable {

   private $observers=array();
   private $storage;

   const LOGIN_USER_UNKNOWN = 1;
   const LOGIN_WRONG_PASS = 2;
   const LOGIN_ACCESS = 3;

   function __construct() {
      $this->observers = array();
  }
```

```
   function attach( Observer $observer ) {
      $this->observers[] = $observer;
   }

   function detach( Observer $observer ) {

      $this->observers = array_filter( $this->observers,
               function( $a ) use ( $observer )
                     { return (! ($a === $observer )); } );
   }

   function notify() {
      foreach ( $this->observers as $obs ) {
         $obs->update( $this );
      }
   }
//...
}
```

Итак, класс `Login` управляет списком объектов-наблюдателей. Они могут быть добавлены третьей стороной с помощью метода `attach()` и удалены с помощью метода `detach()`. Метод `notify()` вызывается, чтобы сказать наблюдателям о том, что произошло что-то интересное. Этот метод просто проходит в цикле по списку наблюдателей, вызывая для каждого из них метод `update()`.

Сам класс `Login` вызывает метод `notify()` из своего метода `handleLogin()`.

```
function handleLogin( $user, $pass, $ip ) {
   switch ( rand(1,3) ) {
      $isvalid = false;
      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;
}
```

Давайте определим интерфейс для класса `Observer`.

```
interface Observer {
   function update( Observable $observable );
}
```

Любой объект, использующий этот интерфейс, можно добавить к классу `Login` с помощью метода `attach()`. Давайте создадим конкретный экземпляр.

```
class SecurityMonitor implements Observer {
   function update( Observable $observable ) {
      $status = $observable->getStatus();
```

```
        if ( $status[0] == Login::LOGIN_WRONG_PASS ) {
            // Отправим почту системному администратору
            print __CLASS__ . ":\tОтправка почты системному администратору \n";
        }
    }
}
$login = new Login();
$login->attach( new SecurityMonitor() );
```

Обратите внимание на то, как объект-наблюдатель использует переданный ему экземпляр объекта `Observable`, чтобы получить дополнительную информацию о событии. Класс-субъект должен обеспечить методы, которые могут запросить наблюдатели, чтобы узнать о состоянии. В данном случае мы определили метод `getStatus()`, который могут вызывать наблюдатели, чтобы получить информацию о текущем состоянии.

Но это добавление также выявляет проблему. При вызове метода `Login::getStatus()` классу `SecurityMonitor` передается больше информации, чем нужно для ее 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. Мы можем расширить интерфейс `Observable` и включить в него объявление метода `getStatus()`. При этом, вероятно, придется переименовать интерфейс во что-то вроде `ObservableLogin`, чтобы показать, что он характерен для типа `Login`.

Есть и второй вариант — сохранить интерфейс `Observable` общим, но сделать так, чтобы классы `Observer` были ответственными за работу с субъектами правильного типа. Они даже могут выполнять работу по присоединению себя к своему субъекту. Поскольку у нас будет больше одного типа `Observer` и мы планируем выполнять некоторые операции, общие для них всех, давайте создадим абстрактный суперкласс, который будет заниматься этой вспомогательной работой.

```
abstract class LoginObserver implements Observer {
    private $login;

    function __construct( Login $login ) {
        $this->login = $login;
        $login->attach( $this );
    }

    function update( Observable $observable ) {
        if ( $observable === $this->login ) {
            $this->doUpdate( $observable );
        }
    }

    abstract function doUpdate( Login $login );
}
```

Конструктору класса `LoginObserver` нужно передать объект типа `Login`. Он сохраняет на него ссылку и вызывает метод `Login::attach()`. При вызове метода `update()` класса `LoginObserver` вначале проверяется, что ему передана корректная ссылка на объект типа `Observable`. Затем вызывается шаблонный метод `doUpdate()`. Теперь мы можем создать набор объектов типа `LoginObserver`, причем все они могут быть уверены, что работают с объектом `Login`, а не только с любым старым объектом, поддерживающим интерфейс `Observable`.

```
class SecurityMonitor extends LoginObserver {
   function doUpdate( Login $login ) {
      $status = $login->getStatus();
      if ( $status[0] == Login::LOGIN_WRONG_PASS ) {
         // Отправим почту системному администратору
         print __CLASS__ . ":\tОтправка почты системному администратору \n";
      }
   }
}

class GeneralLogger extends LoginObserver {
   function doUpdate( Login $login ) {
      $status = $login->getStatus();
      // Зарегистрируем подключение в журнале
      print __CLASS__ . ":\tРегистрация в системном журнале\n";
   }
}

class PartnershipTool extends LoginObserver {
   function doUpdate( Login $login ) {
      $status = $login->getStatus();
      // Проверим IP-адрес
      // Отправим cookie-файл, если адрес соответствует списку
      print __CLASS__ .
      ":\tОтправка cookie-файла, если адрес соответствует списку\n";
   }
}
```

Теперь создание и подключение к субъекту класса типа `LoginObserver` выполняются сразу во время создания его экземпляра.

```
$login = new Login();
new SecurityMonitor( $login );
new GeneralLogger( $login );
new PartnershipTool( $login );
```

Итак, теперь мы создали гибкую связь между классами-субъектами и наблюдателями. Диаграмма классов для нашего примера показана на рис. 11.6.

В PHP обеспечивается встроенная поддержка шаблона Observer через входящее в поставку расширение SPL (Standard PHP Library). SPL — это набор инструментов, которые помогают решать распространенные объектно-ориентированные задачи. То, что в SPL имеет отношение к шаблону Observer, состоит из трех элементов: `SplObserver`, `SplSubject` и `SplObjectStorage`. `SplObserver` и `SplSubject` — это интерфейсы, которые являются точной аналогией интерфейсов `Observer` и `Observable` в приведенном выше примере из данного раздела. `SplObjectStorage` — это вспомогательный класс, предназначенный для того, чтобы обеспечивать улучшенное сохранение и удаление объектов. Вот измененная версия реализации нашего шаблона Observer.

```
class Login implements SplSubject {
   private $storage;
   //...
   function __construct() {
      $this->storage = new SplObjectStorage();
   }
```

```
    function attach( SplObserver $observer ) {
        $this->storage->attach( $observer );
    }

    function detach( SplObserver $observer ) {
        $this->storage->detach( $observer );
    }

    function notify() {
        foreach ( $this->storage as $obs ) {
            $obs->update( $this );
        }
    }
//...
}

abstract class LoginObserver implements SplObserver {
    private $login;

    function __construct( Login $login ) {
        $this->login = $login;
        $login->attach( $this );
    }

    function update( SplSubject $subject ) {
        if ( $subject === $this->login ) {
            $this->doUpdate( $subject );
        }
    }

    abstract function doUpdate( Login $login );
}
```

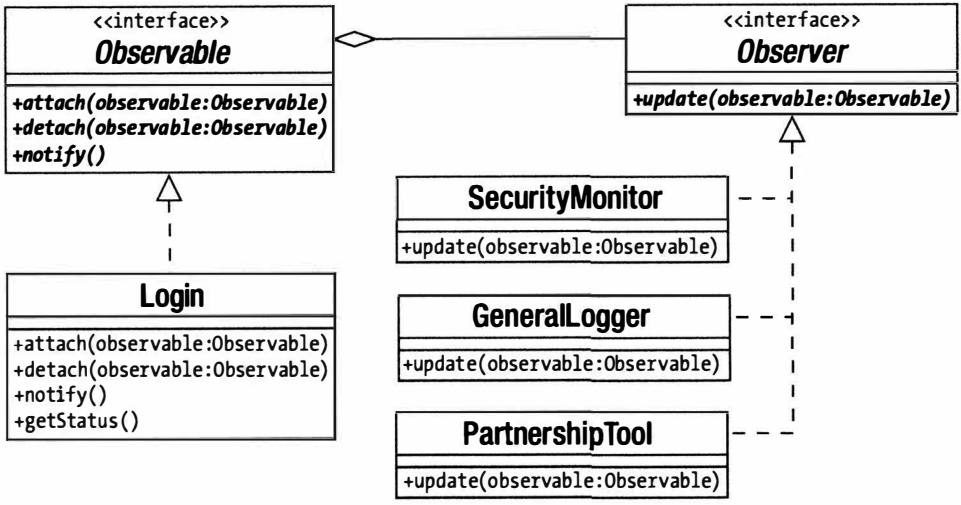


Рис. 11.6. Шаблон Observer

Что касается `SplObserver` и `SplSubject`, то не существует реальных различий между ними и `Observer` и `Observable`, за исключением, конечно, того, что нам больше не нужно объявлять интерфейсы и мы должны изменить уточнение типа в параметрах методов в соответствии с новыми именами. Но класс `SplObjectStorage` предоставляет нам действительно полезный сервис. Вы, наверное, 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-аргумент. Класс `SplObjectStorage` скрытно выполняет всю эту работу. В нем реализованы методы `attach()` и `detach()`, и его можно использовать в цикле `foreach` для выполнения итераций по аналогии с массивом.

На заметку. За более подробной информацией об SPL обращайтесь к документации по PHP по адресу `http://www.php.net/spl`. В частности, там вы найдете много инструментов-итераторов. О встроенном в PHP интерфейсе `Iterator` читайте в главе 13.

Еще один подход к проблеме взаимодействия между субъектом типа `Observable` и наблюдателем типа `Observer` состоит в том, что методу `update()` можно передать конкретную информацию о состоянии, а не экземпляр субъекта типа `Observable`. Для быстрого решения я обычно выбирал этот подход. Поэтому в нашем примере методу `update()` должны были бы передаваться код состояния, имя пользователя и IP-адрес (вероятно, в массиве, для переносимости), а не экземпляр класса `Login`. Благодаря этому нам не пришлось бы создавать отдельный метод в классе `Login` для получения состояния. С другой стороны, поскольку в классе-субъекте сохраняется много информации о состоянии, передача его экземпляра методу `update()` предоставляет наблюдателям гораздо большую степень гибкости.

Вы можете также полностью зафиксировать тип аргумента, чтобы класс `Login` работал только с классом-наблюдателем определенного типа (вероятно, `Login Observer`). Для этого во время выполнения программы предусмотрите проверку типов объектов, переданных методу `attach()`; в противном случае вам, возможно, придется вообще пересмотреть интерфейс `Observable`.

И снова мы использовали композицию во время выполнения, чтобы построить гибкую и расширяемую модель. Класс `Login` можно извлечь из контекста и включить в совершенно другой проект без всяких изменений. И там он сможет работать с другим набором наблюдателей.

Шаблон Visitor

Как мы видели, многие шаблоны предназначены для создания структур данных во время выполнения программы, следуя тому принципу, что композиция обладает большей гибкостью, чем наследование. Прекрасный пример тому — вездесущий шаблон Composite. Когда вы работаете с набором объектов, то вам может понадобиться применять различные операции к структуре, в результате которых задействуется каждый ее отдельный компонент. Такие операции могут быть встроены в сами компоненты. В конце концов, компоненты обычно лучше всего предназначены для вызова друг друга.

Этот подход не лишен недостатков. У вас не всегда есть информация обо всех операциях, которые может понадобиться осуществить на структуре. Если вы добавляете поддержку для новых операций к классам от случая к случаю, то в результате ваш интерфейс может обрасти ненужными функциями, которые на самом деле ему не характерны. Как вы уже, наверное, догадались, эти проблемы позволяет решить шаблон Visitor.

Проблема

Вспомните о примере шаблона Composite из предыдущей главы. Для игры мы создали армию компонентов, так что с целым и его частями можно было обращаться попеременно. Мы видели, что операции можно встроить в компоненты. Обычно объекты-листья выполняют операцию, а объекты-композиты вызывают свои дочерние объекты, чтобы выполнить эту операцию.

```
class Army extends CompositeUnit {

   function bombardStrength() {
      $strength = 0;
      foreach( $this->units() as $unit ) {
         $strength += $unit->bombardStrength();
      }
      return $strength;
   }
}

class LaserCannonUnit extends Unit {
   function bombardStrength() {
      return 44;
   }
}
```

Если конкретная операция является неотъемлемой частью и входит в зону ответственности класса-композита, то никакой проблемы нет. Но существуют более частные задачи, которые не так удачно впишутся в интерфейс.

Ниже приведен пример функции, которая выводит текстовую информацию об узлах-"листьях". Ее можно добавить к абстрактному классу `Unit`.

```
// Unit
function textDump( $num=0 ) {
   $txtout = "";
   $pad  = 4*$num;
   $txtout .= sprintf( "%{$pad}s", "" );
   $txtout .= get_class($this) . ": ";
   $txtout .= "Огневая мощь: " . $this->bombardStrength() . "\n";
   return $txtout;
}
```

Этот метод затем можно переопределить в классе `CompositeUnit` так.

```
// CompositeUnit
function textDump( $num=0 ) {
   $txtout = parent::textDump( $num );
   foreach ( $this->units as $unit ) {
   $txtout .= $unit->textDump( $num + 1 );
   }
   return $txtout;
}
```

Теперь нам нужно продолжать в том же духе и создать методы для подсчета количества боевых единиц на дереве, для сохранения компонентов в базе данных и для подсчета количества элементов пищи, потребленной армией.

Почему нужно включать эти методы в интерфейс композита? На это есть только один действительно убедительный ответ. Мы включаем сюда эти несопоставимые

операции, потому что именно здесь функция может получить легкий доступ к связанным узлам в структуре композита.

Хотя это правда, что легкость прохождения — это одна из особенностей шаблона Composite, отсюда не следует, что каждая операция, которой нужно обойти дерево, должна поэтому требовать себе место в интерфейсе Composite.

Вот как выглядит расстановка сил. Нам нужно получить все преимущества от легкого прохождения по дереву, что предоставляет наша объектная структура, но при этом нужно сделать это, не "раздувая" интерфейс.

Реализация

Давайте начнем с наших интерфейсов. В абстрактном классе `Unit` определим метод `accept()`.

```
abstract class Unit {

   protected $depth = 0;
   // ...
   function accept( ArmyVisitor $visitor ) {
      $method = "visit" . get_class( $this );
      $visitor->$method( $this );
   }

   protected function setDepth( $depth ) {
      $this->depth=$depth;
   }

   function getDepth() {
      return $this->depth;
   }
}
```

Как видите, метод `accept()` ожидает, что ему будет передан объект `ArmyVisitor`. В PHP можно динамически указать метод в объекте `ArmyVisitor`, который мы хотим вызвать. Поэтому я составил имя этого метода, включив в него имя текущего класса, а затем вызвал данный метод по отношению к переданному в качестве параметра объекту `ArmyVisitor`. Таким образом, если мы работаем с классом `Army`, будет вызван метод `ArmyVisitor::visitArmy()`, а если текущим является класс `TroopCarrier`, то вызывается метод `ArmyVisitor::visitTroopCarrier()` и т.п. Это позволит нам не реализовывать метод `accept()` на каждом листовом узле в нашей иерархии классов. Для удобства я также добавил два метода — `getDepth()` и `setDepth()`. Их можно использовать, чтобы сохранять и извлекать значение вложенности элемента на дереве. Метод `setDepth()` вызывается родителем элемента, когда тот добавляет его к дереву из метода `CompositeUnit::addUnit()`.

```
function addUnit( Unit $unit ) {
   foreach ( $this->units as $thisunit ) {
      if ( $unit === $thisunit ) {
         return;
      }
   }
   $unit->setDepth($this->depth+1);
   $this->units[] = $unit;
}
```

Теперь нам осталось определить только один метод `accept()` в абстрактном классе-композите.

```
function accept( ArmyVisitor $visitor ) {
   $method = "visit" . get_class( $this );
   $visitor->$method( $this );
   foreach ( $this->units as $thisunit ) {
      $thisunit->accept( $visitor );
   }
}
```

Этот метод делает то же самое, что и `Unit::accept()`, но с одним добавлением. После вызова подходящего `visit`-метода для переданного объекта типа `ArmyVisitor` он проходит по циклу все дочерние объекты, вызывая метод `accept()`. На самом деле, поскольку метод `accept()` заменяет операцию своего родителя, мы можем здесь избавиться от повторения.

```
function accept( ArmyVisitor $visitor ) {
   parent::accept( $visitor );
   foreach ( $this->units as $thisunit ) {
      $thisunit->accept( $visitor );
   }
}
```

Устранение повторения таким способом может быть очень удачным, но в данном случае мы избавились только от одной строки кода, возможно, немного потеряв в ясности. Но в любом случае метод `accept()` позволяет делать следующее:

- вызывать корректный `visit`-метод для текущего компонента;
- передавать объект-посетитель всем текущим дочерним элементам с помощью метода `accept()` (предполагая, что текущий компонент — это композит).

Мы должны еще определить интерфейс для `ArmyVisitor`. Для этого некоторую информацию должны дать методы `accept()`. Класс-посетитель должен определить методы `accept()` для каждого из конкретных классов в иерархии классов. Это позволит обеспечить разные функции для разных объектов. В моей версии данного класса я также определил стандартный метод `visit()`, который автоматически вызывается, если реализующие классы "решают" не выполнять специальную обработку для определенных классов `Unit`.

```
abstract class ArmyVisitor {
   abstract function visit( Unit $node );

   function visitArcher( Archer $node ) {
      $this->visit( $node );
   }

   function visitCavalry( Cavalry $node ) {
      $this->visit( $node );
   }

   function visitLaserCannonUnit( LaserCannonUnit $node ) {
      $this->visit( $node );
   }

   function visitTroopCarrierUnit( TroopCarrierUnit $node ) {
      $this->visit( $node );
```

```
    }

    function visitArmy( Army $node ) {
        $this->visit( $node );
    }
}
```

Теперь остается только вопрос предоставления реализаций класса `ArmyVisitor`, и мы готовы к работе. Вот простой пример кода, выводящего текстовую информацию, заново реализованный на основе объекта `ArmyVisitor`.

```
class TextDumpArmyVisitor extends ArmyVisitor {
    private $text="";

    function visit( Unit $node ) {
        $txt  = "";
        $pad  = 4*$node->getDepth();
        $txt .= sprintf( "%{$pad}s", "" );
        $txt .= get_class($node).": ";
        $txt .= "Огневая мощь: " . $node->bombardStrength() . "\n";
        $this->text .= $txt;
    }

    function getText() {
        return $this->text;
    }
}
```

Давайте рассмотрим клиентский код, а затем пройдемся по всему процессу.

```
$main_army = new Army();
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );
$main_army->addUnit( new Cavalry() );
$textdump = new TextDumpArmyVisitor();
$main_army->accept( $textdump );
print $textdump->getText();
```

Этот код на выходе даст следующее.

```
Army: Огневая мощь: 50
    Archer: Огневая мощь: 4
    LaserCannonUnit: Огневая мощь: 44
    Cavalry: Огневая мощь: 2
```

Мы создаем объект `Army`. Поскольку `Army` — композит, у него есть метод `addUnit()`, который мы используем для добавления дополнительных объектов типа `Unit`. Затем мы создаем объект `TextDumpArmyVisitor`. Мы передаем его методу `Army::accept()`. Метод `accept()` на ходу создает имя метода и вызывает `TextDumpArmyVisitor::visitArmy()`. В данном случае мы не обеспечили специальной обработки для объектов типа `Army`, поэтому вызов передается общему методу `visit()`. Методу `visit()` передается по ссылке наш объект `Army`. Он вызывает свои методы (включая вновь добавленный `getDepth()`, который сообщает всем, кому нужно знать, глубину вложения элемента в иерархии объекта), чтобы сгенерировать итоговые данные. Вызов `visitArmy()` выполнен, операция `Army::accept()` теперь вызывает по очереди метод `accept()` для своих дочерних объектов, передавая объект-посетителя. В результате класс `ArmyVisitor` посетит каждый объект на дереве.

Добавив всего пару методов, мы создали механизм, посредством которого можно встроить новые функции в классы-композиты, не ухудшая их интерфейс и не используя много повторений кода обхода дерева.

На некоторых клетках в нашей игре армии должны платить налоги. Сборщик налогов посещает армию и берет плату за каждый элемент (подразделение), который он находит. Разные подразделения должны платить разные суммы налогов. Здесь мы можем воспользоваться преимуществами специализированных методов в классе-посетителе.

```
class TaxCollectionVisitor extends ArmyVisitor {
   private $due=0;
   private $report="";

   function visit( Unit $node ) {
      $this->levy( $node, 1 );
   }

   function visitArcher( Archer $node ) {
      $this->levy( $node, 2 );
   }

   function visitCavalry( Cavalry $node ) {
      $this->levy( $node, 3 );
   }

   function visitTroopCarrierUnit(TroopCarrierUnit $node ) {
      $this->levy( $node, 5 );
   }

   private function levy( Unit $unit, $amount ) {
      $this->report .= "Налог для " . get_class( $unit );
      $this->report .= ": $amount\n";
      $this->due += $amount;
   }

   function getReport() {
      return $this->report;
   }

   function getTax() {
      return $this->due;
   }
}
```

В этом простом примере мы непосредственно не используем объекты `Unit`, переданные различным методам `visit`. Но мы используем специализированную природу этих методов, взимая различные суммы налогов в соответствии с конкретным типом вызывающего объекта `Unit`.

Вот пример клиентского кода.

```
$main_army = new Army();
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );
$main_army->addUnit( new Cavalry() );
$taxcollector = new TaxCollectionVisitor();
$main_army->accept( $taxcollector );
```

```
print $taxcollector->getReport() . "\n";
print "ИТОГО: ";
print $taxcollector->getTax() . "\n";
```

Объект `TaxCollectionVisitor` передается методу `accept()` объекта `Army`, как и раньше. И снова `Army` передает ссылку на себя методу `visitArmy()`, прежде чем вызывать метод `accept()` для своих дочерних объектов. Эти компоненты понятия не имеют, какие операции выполняет их посетитель. Они просто сотрудничают с их общедоступным интерфейсом, причем каждый послушно передает ссылку на себя методу, соответствующему его типу.

В дополнение к методам, определенным в классе `ArmyVisitor`, класс `TaxCollectionVisitor` предоставляет два итоговых метода, `getReport()` и `getTax()`. Вызов этих методов предоставляет данные, которые мы и ожидали получить.

```
Налог для Army: 1
Налог для Archer: 2
Налог для LaserCannonUnit: 1
Налог для Cavalry: 3

ИТОГО: 7
```

На рис. 11.7 показаны объекты, участвующие в этом примере.

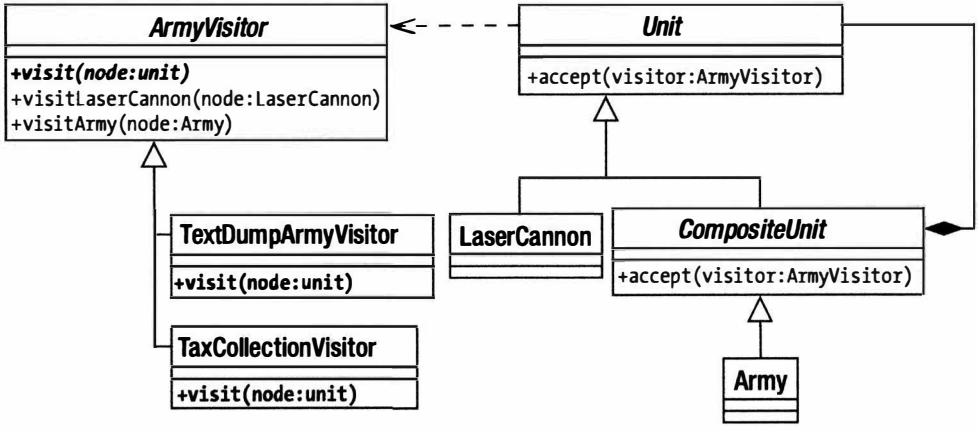


Рис. 11.7. Шаблон Visitor

Проблемы шаблона Visitor

Шаблон Visitor — это еще один шаблон, объединяющий в себе простоту и функциональность. Но при его использовании следует помнить о некоторых моментах.

Хотя шаблон Visitor идеально приспособлен для использования с шаблоном Composite, на самом деле его можно применять с любым набором объектов. Так что можете использовать его, например, со списком объектов, где каждый объект сохраняет ссылку на его "братьев" (т.е. на элементы одного уровня на дереве).

Однако экспортируя операции, вы рискуете "скомпрометировать" инкапсуляцию, т.е. вам может понадобиться показать внутреннее содержание посещенных объектов, чтобы позволить посетителям сделать с ними что-то полезное. Например, в нашем первом примере с `Visitor` мы были вынуждены обеспечить дополнительный метод для интерфейса `Unit`, чтобы предоставить информацию для объектов `TextDumpArmyVisitor`. С этой дилеммой мы уже сталкивались в шаблоне `Observer`.

Поскольку сам процесс итерации отделен от операций, которые выполняют объекты-посетители, вы должны ослабить контроль в некоторой степени. Например, вы не можете легко создать метод `visit()`, который что-то делает и до, и после того, как выполняются итерации дочерних узлов. Один из способов решения — передать ответственность за выполнение итерации в объекты-посетители. Но проблема в том, что это может привести к дублированию кода обхода в каждом посетителе.

По умолчанию я предпочитаю оставлять код обхода внутри посещенных классов, но экспортирование его даст вам одно особое преимущество. Вы сможете изменять способ обработки посещенных классов в зависимости от посетителя.

Шаблон Command

В последние годы я редко завершал веб-проект без использования этого шаблона. Первоначально рассматриваемые в контексте проекта графического пользовательского интерфейса командные объекты способствуют созданию проекта хорошего корпоративного приложения, поддерживают разделение между уровнями контроллера (обработка запросов и диспетчеризации) и моделью предметной области (логика приложения). Проще говоря, шаблон Command помогает создавать хорошо организованные системы, которые легко расширять.

Проблема

Во всех системах должно приниматься решение о том, что делать в ответ на запрос пользователя. В PHP процесс принятия решения часто осуществляется с помощью ряда отдельных контактных страниц. Выбирая страницу (`feedback.php`), пользователь явно дает понять функциям и интерфейсу, что ему требуется. Все чаще программисты на PHP делают выбор в пользу "единственной точки контакта" (этот подход будет обсуждаться в следующей главе). Но в любом случае получатель запроса должен передать полномочия уровню, более связанному с логикой приложения. Такое делегирование особенно важно, если пользователь может сделать запросы на разные страницы. В противном случае дублирование кода в проекте неизбежно.

Итак, предположим, что у нас есть проект с рядом задач, которые нужно выполнить. В частности, наша система должна разрешать одним пользователям входить в систему, а другим — оставлять отклики. Мы можем создать страницы `login.php` и `feedback.php`, которые решают эти задачи, создавая экземпляры соответствующих специализированных классов, которые и выполнят нужную работу. К сожалению, пользовательский интерфейс в системе редко точно соответствует задачам, для решения которых предназначена система. Например, функции входа в систему и оставления откликов могут понадобиться нам на каждой странице. Если страницы должны решать много различных задач, то, вероятно, мы должны представлять себе задачи как нечто, что можно инкапсулировать. Таким способом мы упростим добавление новых задач к системе и построим границу между уровнями системы. И это, конечно, приведет нас к шаблону Command.

Реализация

Интерфейс для командного объекта вряд ли может быть еще проще! Он требует реализовать только один метод — `execute()`.

На рис. 11.8 я представил класс `Command` в виде абстрактного класса. На этом уровне простоты его можно было бы определить как интерфейс. Но я склонен ис-

пользовать абстрактный класс в данном случае, потому что базовый класс также может предоставить полезные общие функции 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; и получатель, на который действует команда.

Получатель может быть передан командному объекту клиентским кодом через конструктор или запрошен от какого-либо объекта-фабрики. Я предпочитаю второй подход, чтобы у конструктора не было аргументов. Затем можно создать экземпляры всех объектов `Command` точно таким же способом.

Ниже приведено определение абстрактного класса.

```
abstract class Command {
   abstract function execute( CommandContext $context );
}
```

Давайте построим конкретный класс `Command`.

```
class LoginCommand extends Command {
   function execute( CommandContext $context ) {
      $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 типа `AccessManager`. `AccessManager` — это воображаемый класс, задача которого — управлять механизмом входа пользователей в систему. Обратите внимание на то, что нашему методу `Command::execute()` требуется передать объект `CommandContext` (в книге *Core J2EE Patterns* он называется `RequestHelper`). Это механизм, посредством которого данные запроса могут быть переданы объектам `Command`, а ответы — отправлены назад на уровень представления. Использовать объект таким способом полезно, потому что мы можем передать различные параметры командам, не нарушая интерфейс. Класс `CommandContext` — это, по сути, объект-оболочка для ассоциативного массива переменных, хотя его часто расширяют для выполнения дополнительных полезных задач. Вот пример простой реализации `CommandContext`.

```
class CommandContext {
   private $params = array();
   private $error  = "";
```

```
    function __construct() {
        $this->params = $_REQUEST;
    }

    function addParam( $key, $val ) {
        $this->params[$key]=$val;
    }

    function get( $key ) {
        if ( isset( $this->params[$key] ) ) {
            return $this->params[$key];
        }
        return null;
    }

    function setError( $error ) {
        $this->error = $error;
    }

    function getError() {
        return $this->error;
    }
}
```

Итак, "вооруженный" объектом `CommandContext`, класс `LoginCommand` может получить доступ к полученным данным запроса: имени пользователя и паролю. Мы используем простой класс `Registry` со статическими методами для генерации общих объектов. Он возвращает объект типа `AccessManager`, с которым будет работать класс `LoginCommand`. Если при выполнении метода `login()` объекта `AccessManager` происходит ошибка, то сообщение об ошибке сохраняется в объекте `CommandContext`, чтобы его можно было отобразить на уровне представления, и возвращается значение `false` ("ложь"). Если все в порядке, то метод `execute()` объекта `LoginCommand` просто возвращает значение `true` ("истина"). Обратите внимание на то, что объекты `Command` сами выполняют мало логических операций. Они проверяют входные данные, обрабатывают ошибки и сохраняют данные, а также вызывают другие методы объектов для выполнения операций, о которых они должны отчитаться. Если вы обнаружили, что логика приложения "вкралась" в ваши командные классы, значит, вам следует подумать о том, как это исправить. Такой код способствует дублированию, поскольку его неизбежно копируют и вставляют из одной команды в другую. Вы должны по меньшей мере посмотреть, к чему относятся эти функции. Возможно, их лучше всего переместить в объекты логики приложения или на уровень фасада. В этом примере нам еще не хватает клиента — класса, который генерирует командные объекты, и вызывающего участника — класса, который работает со сгенерированной командой. Самый простой способ выбора того, экземпляры каких команд требуется создавать в веб-проекте, — использовать параметр в самом запросе. Вот пример упрощенного клиента.

```
class CommandNotFoundException extends Exception {}

class CommandFactory {
    private static $dir = 'commands';

    static function getCommand( $action='Default' ) {
        if ( preg_match( '/\W/', $action ) ) {
            throw new Exception("Недопустимые символы в команде");
```

```
        }
        $class = UCFirst(strtolower($action)) . "Command";
        $file  = self::$dir . DIRECTORY_SEPARATOR . "{$class}.php";
        if ( ! file_exists( $file ) ) {
           throw new CommandNotFoundException( "Файл '$file' не найден" );
        }
        require_once( $file );
        if ( ! class_exists( $class ) ) {
           throw new CommandNotFoundException( "Класс '$class' не обнаружен" );
        }
        $cmd = new $class();
        return $cmd;
    }
}
```

Класс `CommandFactory` просто ищет в каталоге `commands` определенный файл класса. Это имя файла конструируется с помощью параметра `$action` объекта `Command Context`, который, в свою очередь, был передан системе из запроса. Если файл найден и класс существует, то он возвращается вызывающему объекту. Сюда можно добавить еще больше операций проверки ошибок, чтобы убедиться, что найденный класс принадлежит семейству `Command` и что конструктор не ожидает аргументов, но данный вариант полностью подходит для наших целей. Преимущество данного подхода в том, что вы можете добавить новый объект `Command` в каталог команд в любое время, и система сразу станет поддерживать его.

Вызывающий объект теперь — сама простота.

```
class Controller {
   private $context;

   function __construct() {
      $this->context = new CommandContext();
   }

   function getContext() {
      return $this->context;
   }

   function process() {
      $action = $this->context->get('action');
      $action = ( is_null( $action ) ) ? "default" : $action;
      $cmd = CommandFactory::getCommand($action) );
      if ( ! $cmd->execute( $this->context ) ) {
         // Обработка ошибки
      } else {
         // Все прошло успешно
         // Теперь отобразим результаты
      }
   }
}

$controller = new Controller();
// Эмулируем запрос пользователя
$context = $controller->getContext();
$context->addParam('action', 'login' );
$context->addParam('username', 'bob' );
```

```
$context->addParam('pass', 'tiddles' );
$controller->process();
```

Прежде чем вызвать метод `Controller::process()`, мы имитируем веб-запрос, определяя параметры объекта `CommandContext`, экземпляр которого создан в конструкторе контроллера. В методе `process()` запрашивается значение параметра `'action'` и, если его не существует, используется строка `'default'`. Затем метод `process()` делегирует создание экземпляров объектов объекту `CommandFactory`, после чего он вызывает метод `execute()` для возвращенного командного объекта. Обратите внимание на то, что контроллер не имеет представления о внутреннем содержании команды. Именно эта независимость от деталей выполнения команды дает нам возможность добавлять новые классы `Command` без воздействия на всю систему в целом.

Давайте создадим еще один класс типа `Command`.

```
class FeedbackCommand extends Command {
   function execute( CommandContext $context ) {
      $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 вернемся к шаблону Command в главе 12, где будет приведена более полная реализация класса фабрики `Command`. Представленная здесь структура выполнения команд является упрощенной версией другого шаблона, с которым мы познакомимся, — Front Controller.

При условии помещения этого класса в файл `FeedbackCommand.php` и сохранения его в папке `commands` он будет вызываться в ответ на получение из запроса строки `"feedback"`, соответствующей параметру `'action'`. Причем не понадобится вносить никакие изменения в контроллер или классы `CommandFactory`.

На рис. 11.9 показаны все участники шаблона Command.

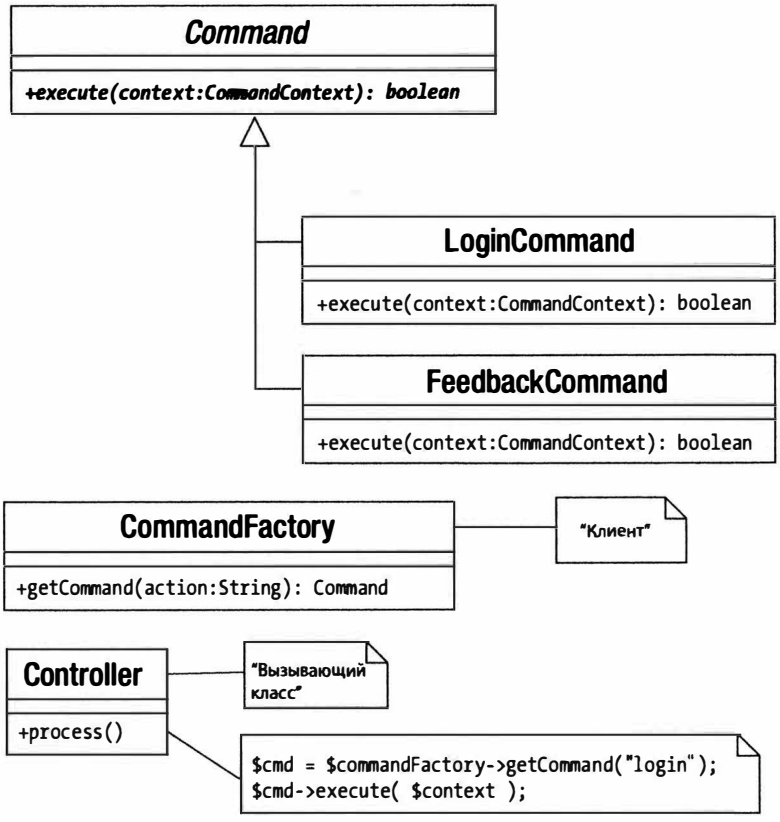


Рис. 11.9. Участники шаблона Command

Резюме

В данной главе мы завершили изучение шаблонов "Банды четырех". Мы разработали мини-язык и построили его интерпретатор на основе шаблона Interpreter. Мы обнаружили в шаблоне Strategy другой способ использования композиции, чтобы увеличить степень гибкости и уменьшить потребность в повторном создании подклассов. Шаблон Observer решил проблему уведомления совершенно различных и меняющихся компонентов о событиях в системе. Мы снова использовали пример с шаблоном Composite и с помощью шаблона Visitor узнали, как осуществить вызов каждого компонента дерева и применить к нему множество операций. И наконец мы увидели, как шаблон Command может помочь нам построить расширяемую многоуровневую систему. В следующей главе мы выйдем за рамки шаблонов "Банды четырех" и изучим некоторые шаблоны, специально предназначенные для использования в корпоративных приложениях.

Глава 12

Шаблоны корпоративных приложений

PHP — это, прежде всего, язык, предназначенный для работы в веб-среде. И поскольку в PHP 5 была существенно расширена поддержка объектов, теперь вы можете пользоваться преимуществами шаблонов, разработанных в контексте других объектно-ориентированных языков, в частности Java.

В данной главе я буду использовать один пример для иллюстрации описываемых шаблонов. Но помните, что, решив использовать один шаблон, вы не обязаны использовать все шаблоны, которые подходят для работы с ним. Не считайте также, что представленные здесь реализации — это единственный способ использования данных шаблонов. Используйте приведенные примеры, чтобы понять суть описываемых шаблонов, и не стесняйтесь брать то, что нужно для ваших проектов.

Поскольку нужно изложить очень объемный материал, данная глава — одна из самых длинных и сложных в книге, поэтому ее будет сложно прочитать в один присест. Она подразделяется на введение и две основные части. Такое деление поможет вам освоить материал поэтапно.

Отдельные шаблоны я также описываю в разделе "Обзор архитектуры". Хотя они в некоторой степени взаимозависимы, вы сможете перейти к любому конкретному шаблону и работать с ним независимо, а в свободное время сможете изучить связанные с ним шаблоны.

В этой главе мы рассмотрим следующие темы.

- *Обзор архитектуры:* знакомство с уровнями, 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 пользователю.
- *Уровень логики приложения:* обращение к настоящей цели разрабатываемой системы — решение проблем бизнеса.

Обзор архитектуры

Поскольку нужно изложить большой объем материала, давайте начнем с обзора шаблонов, которые мы будем рассматривать, и продолжим введением в построение многоуровневых приложений.

Шаблоны

В этой главе мы рассмотрим перечисленные ниже шаблоны. Можете читать текст от начала до конца или подробно рассмотреть те шаблоны, которые вам нужны или интересны на данный момент. Обратите внимание на то, что шаблон Command в данной главе отдельно не рассматривается (я писал о нем в главе 11), но вы с ним снова встретитесь в шаблонах Front Controller и Application Controller.

- *Registry*. Этот шаблон используется для того, чтобы сделать данные доступными для всех классов в текущем процессе. Если аккуратно применять сериализацию, то данный шаблон также можно использовать для сохранения информации на протяжении сеанса работы или даже всего приложения.
- *Front Controller*. Используйте этот шаблон для больших систем, в которых вам понадобится максимально возможная степень гибкости при управлении различными представлениями и командами.
- *Application Controller*. Создайте класс для управления логикой представления данных и выбором команд.
- *Template View*. Создайте страницы, которые управляют только отображением данных и пользовательским интерфейсом, встраивайте динамическую информацию в статические страницы, используя минимум кода.
- *Page Controller*. Более легкий, но менее гибкий, чем Front Controller, шаблон Page Controller решает те же задачи. Используйте его, чтобы управлять запросами и логикой представления данных, если хотите быстро получить результаты, и, скорее всего, сложность вашей системы существенно не увеличится.
- *Transaction Script*. Если вам нужно быстро выполнять задачи с минимальным предварительным планированием, используйте процедурный библиотечный код для создания логики приложения. Этот шаблон плохо масштабируется.
- *Domain Model*. Противоположность шаблону Transaction Script. Используйте его, чтобы строить объектные модели участников и процессов бизнеса.

Приложения и уровни

Многие (если не большинство!) шаблоны в этой главе предназначены для того, чтобы способствовать независимой работе нескольких различных уровней в приложении. Уровни корпоративной системы, подобно классам, представляют собой специализацию обязанностей, только в более крупном масштабе. Типичное разбиение системы на уровни показано на рис. 12.1.

Структура, показанная на рис. 12.1, — не есть нечто неизменное: некоторые уровни могут объединяться, для коммуникаций между ними могут использоваться другие стратегии, в зависимости от сложности системы. Тем не менее на рис. 12.1 проиллюстрирована модель, в которой делается акцент на гибкость и повторное использование кода, и многие корпоративные приложения следуют этой модели в значительной степени.

- *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 в систему.
- *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 и уровень представления данных часто объединяют в один *уровень представления*. Но даже в этом случае роль отображения должна быть строго отделена от ролей обработки запросов и вызова логики приложения.

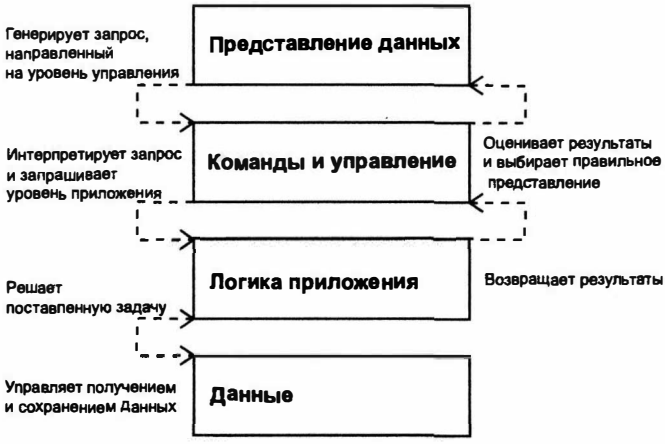


Рис. 12.1. Уровни типичной корпоративной системы

- *Уровень логики приложения* отвечает за обработку бизнес-запроса. Он выполняет все необходимые вычисления и упорядочивает полученные в результате данные.
- *Уровень данных* отделяет остальную систему от механизма хранения и получения необходимой информации. В одних системах уровень управления и команд использует уровень данных, чтобы получить объекты приложения, с которыми ему нужно работать. В других системах уровень данных является скрытым, насколько это возможно.

Так какой же смысл разбивать систему таким 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 без изменения либо с небольшим изменением ее кода.

Представьте себе систему управления списками событий (к концу данной главы мы разберем этот пример довольно подробно). Конечному пользователю нужен симпатичный HTML-интерфейс, что вполне естественно. Администраторам, поддерживающим систему, может понадобиться интерфейс командной строки для встраивания в автоматизированные системы. В то же время вы можете разрабатывать версии системы для работы с мобильными телефонами и другими портативными устройствами. Возможно, вы даже думаете о применении SOAP или RESTful API.

Если вы первоначально объединили 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 заставить вас немедленно приступить к переписыванию системы. Если, с другой стороны, вы создали многоуровневую систему, то сможете

добавить новые методики представления данных, не пересматривая уровни логики приложения и данных.

Кроме того, методики сохранения данных также подвержены изменениям. И снова у вас должна быть возможность выбирать методики сохранения данных с минимальным воздействием на другие уровни в системе.

Тестирование — это еще одна хорошая причина создания систем с отдельными уровнями. Общеизвестно, что веб-приложения трудно тестировать. Автоматический тест любого вида будет вынужден выполнять анализ HTML-интерфейса, с одной стороны, и работать с реальными базами данных, с другой. Это означает, что тесты должны выполняться на работоспособной системе, в результате чего существует риск повреждения самой системы, которую они призваны защищать. На любом уровне классы, которые обращаются к другим уровням, часто создают так, что они расширяют абстрактный суперкласс или реализуют некоторый интерфейс. Этот супертип может затем поддерживать полиморфизм. В контексте теста весь уровень может быть заменен набором пустых объектов (которые часто называют "заглушками" или "фиктивными объектами"). Таким образом, вы можете протестировать логику приложения, например, с помощью фиктивного уровня данных. Подробнее о тестировании читайте в главе 18.

Уровни полезны даже в том случае, если 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 одинаковый тип операций, в одном месте (а не разбрасывая по всему коду, например, вызовы к базе данных или к методам отображения), вы тем самым снизите вероятность появления дублирования. Добавить что-то в систему относительно просто, потому что изменения будут происходить аккуратно по вертикали, а не беспорядочно по горизонтали.

В многоуровневой системе новой функции могут потребоваться новый компонент интерфейса, дополнительная обработка запроса, дополнительная логика приложения и изменение механизма хранения данных. Это вертикальное изменение. В немногоуровневой системе вы можете добавить новую функцию, а затем вспоминать, во скольких отдельных страницах используется измененная таблица базы данных (в пяти, а может, в шести?). Могут существовать десятки мест, где потенциально может быть вызван новый интерфейс. Поэтому вам придется пройтись по всей системе, добавляя в нужных местах соответствующий код. Это горизонтальное изменение.

На заметку. Хотя многие из описываемых здесь шаблонов некоторое время уже существовали (в конце концов, шаблоны являются отражением испытанных методов), их названия и области применения взяты либо из основной работы Мартина Фаулера по шаблонам корпоративных приложений *Patterns of Enterprise Application Architecture*[1], либо из авторитетной книги Дипака Алура *Core J2EE Patterns*[2]. В случае расхождений в этих двух источниках для согласованности я обычно использую соглашения об именовании 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] Мартин Фаулер. *Шаблоны корпоративных приложений* (пер. с англ., ИД "Вильямс", 2009).

[2] Дипак Алур, Джон Крупи, Дэн Малкс. *Образцы J2EE. Лучшие решения и стратегии проектирования* (пер. с англ., изд. "Лори", 2013).

Разумеется, на практике вам никогда не удастся полностью избежать горизонтальных зависимостей такого рода, особенно если речь идет об элементах навигации в интерфейсе. Но многоуровневая система поможет свести к минимуму необходимость в горизонтальных изменениях.

Все примеры в данной главе связаны с вымышленной системой управления текстовыми списками со странным именем "Woo", которое расшифровывается как "What's On Outside"[3].

К участникам системы относятся различные заведения (театры, клубы, кинотеатры), места (экран 1, верхняя часть сцены) и названия мероприятия (например, фильмы *Долгая страстная пятница* и *Как важно быть серьезным*).

Операции, которые я буду описывать, включают создание заведения, добавление к нему места и вывод списка всех заведений в системе.

Помните, что цель этой главы — проиллюстрировать основные проектные шаблоны корпоративных приложений, а не построить рабочую систему. Природа проектных шаблонов такова, что они являются взаимозависимыми, поэтому в большинстве этих примеров коды частично совпадают с примерами других кодов, приведенных в других частях главы. Поскольку этот код, главным образом, предназначен для демонстрации корпоративных шаблонов, по большей части он не удовлетворяет всем критериям корпоративной системы. В частности, я опустил проверку ошибок в тех случаях, когда это помешало бы ясности восприятия кода. Поэтому рассматривайте эти примеры как средства иллюстрации шаблонов, которые они реализуют, а не как "кирпичики" для построения структуры или приложения.

Небольшое отступление перед началом

Большинство шаблонов в этой книге естественным образом "вписываются" в уровни архитектуры корпоративного приложения. Но некоторые шаблоны являются настолько фундаментальными, что оказываются за пределами этой структуры. Таким является, например, шаблон Registry. По сути, Registry — это эффективный способ выйти за пределы ограничений, накладываемых уровнями структуры. Это — исключение, которое только подтверждает правило.

Шаблон Registry

Этот шаблон предназначен для того, чтобы предоставлять доступ к объектам по 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, что необходимо заново создать глобальные данные под другим именем. Мы уже встречались с шаблоном Singleton в главе 9. Это правда, что объекты Singleton не страдают от всех неприятностей, которые несут глобальные переменные. В частности, вы не сможете случайно стереть объект Singleton. Таким образом, объекты Singleton — это глобальные переменные "с низким содержанием жира". Но у нас остаются подозрения в отношении объектов Singleton, потому что они побуждают нас привязывать классы к системе, тем самым вводя тесную связь.

Тем не менее объекты Singleton иногда настолько полезны, что многие программисты (включая меня) не могут заставить себя от них отказаться.

---

[3] Это можно перевести, как "Что происходит в мире". — *Примеч. ред.*

Проблема

Как мы уже видели, многие корпоративные системы разбиты на уровни, причем каждый уровень осуществляет коммуникации с соседями только через строго определенные каналы. Это разделение на уровни делает приложение гибким. Вы можете заменить или переработать каждый уровень, оказав при этом минимальное воздействие на остальную систему. Но что произойдет, если полученная на каком-то уровне информация впоследствии понадобится на другом несмежном уровне?

Предположим, что данные о конфигурации приложения загружаются в классе `ApplicationHelper`.

```
// woo\controller\ApplicationHelper
class ApplicationHelper {
   function getOptions() {
      if ( ! file_exists( "data/woo_options.xml" ) ) {
         throw new \woo\base\AppException(
                     "Файл с параметрами не найден " );
      }
      $options = simplexml_load_file( "data/woo_options.xml" );
      $dsn = (string)$options->dsn;
      // И что нам со всем этим теперь делать?
      // ...
   }
}
```

Получить эту информацию достаточно просто, но как мы доставим ее на тот уровень данных, где она используется впоследствии? И как насчет всей остальной информации о конфигурации, которая должна быть доступна по всей системе?

Один из способов — передавать информацию по системе от одного объекта к другому: от объекта-контроллера, ответственного за обработку запросов, через объекты на уровне логики приложения и к объекту, отвечающему за коммуникации с базой данных.

Это полностью осуществимо. По сути, вы можете передавать сам объект `ApplicationHelper` или же более специализированный объект `Context`. В любом случае контекстно-зависимая информация объекта `Context` передается через уровни системы объекту (или объектам), которому она нужна.

Но компромисс здесь неизбежен: чтобы сделать это, вы должны изменить интерфейс всех объектов и включить в него объект `Context` независимо от того, нужно ли им его использовать. Очевидно, что это в некоторой степени подрывает слабую связь между объектами приложения.

Альтернативный вариант предоставляет шаблон Registry, но его использование влечет свои последствия.

Системный реестр (Registry) — это просто класс, который предоставляет доступ к данным (как правило, но не исключительно к объектам) с помощью статических методов (или с помощью реализации методов шаблона Singleton). Поэтому каждый объект в системе имеет доступ к этим объектам.

Термин "Registry" взят из книги Мартина Фаулера *Patterns of Enterprise Application Architecture*, но, как и для всех шаблонов, реализации возникают везде. Дэвид Хант (David Hunt) и Дэвид Томас (David Thomas) в своей книге *The Pragmatic Programmer* сравнили класс `Registry` 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. Я также видел шаблон Registry под названиями "Whiteboard" и "Blackboard".

Реализация

На рис. 12.2 показан объект `Registry`, назначение которого — сохранять и возвращать по запросу объекты типа `Request`.

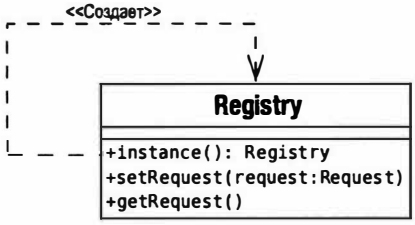


Рис. 12.2. Простой системный реестр

Вот как выглядит этот класс в форме кода.

```
class Registry {
   private static $instance;
   private $request;

   private function __construct() { }

   static function instance() {
      if ( ! isset( self::$instance ) ) {
                  self::$instance = new self();
      }
      return self::$instance;
   }

   function getRequest() {
      if ( is_null( $this->request ) ) {
           $this->request=new Request();

      }
   }

   function setRequest( Request $request ) {
      $this->request = $request;
   }
}

// Пустой класс для тестирования
class Request {}
```

После этого в одной из частей системы вы можете добавить объект типа `Request`

```
$reg = Registry::instance();
$reg->setRequest( new Request() );
```

и получить к нему доступ из другой части системы.

```
$reg = Registry::instance();
print_r( $reg->getRequest() );
```

Как видите, `Registry` — это обычный синглтон (если хотите вспомнить, что представляют собой классы Singleton, обратитесь к главе 9). В коде создается и возвращается единственный экземпляр класса `Registry` с помощью метода `instance()`.

Затем этот класс можно использовать, чтобы задать и извлечь объект типа `Request`. Несмотря на то что PHP не налагает ограничения на возвращаемые типы, значение, возвращенное методом `getRequest()`, гарантированно будет объектом типа `Request` из-за уточнения типа аргумента, заданного в методе `setRequest()`.

Я отбросил все сомнения и использовал систему сохранения параметров на основе ключей следующим образом.

```
class Registry {
   private static $instance = null;
   private $values = array();

   private function __construct() { }

   static function instance() {
      if (is_null( self::$instance ) ) {
                  self::$instance = new self();
      }
      return self::$instance;
   }

   function get( $key ) {
      if ( isset( $this->values[$key] ) ) {
           return $this->values[$key];
      }
      return null;
   }

   function set( $key, $value ) {
      $this->values[$key] = $value;
   }
}
```

Преимущество такого подхода заключается в том, что вам не нужно создавать методы для каждого объекта, который нужно сохранять и возвращать. А недостаток в том, что вы снова вводите глобальные переменные, так сказать, через "черный ход". Использование произвольных строк в качестве ключей для сохраняемых объектов означает следующее: ничто не помешает в одной из частей системы при добавлении объекта изменить пару "ключ–значение". Я решил, что во время разработки удобно использовать данную отображаемую структуру, а когда мне будет ясно, какие данные нужно сохранять и извлекать, можно будет перейти к явно названным методам.

Объекты системного реестра также можно использовать как фабрики для общих объектов в системе. Вместо того чтобы сохранять предоставленный объект, 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.

```
//Класс Registry...
   private $treeBuilder = null;
   private $conf = null;
// ....

   function treeBuilder() {
      if (is_null( $this->treeBuilder ) ) {
                  $this->treeBuilder = new TreeBuilder(
```

```
                                $this->conf()->get('treedir') );
        }
        return $this->treeBuilder;
    }

    function conf() {
        if (is_null( $this->conf ) ) {
                    $this->conf = new Conf();
        }
        return $this->conf;
    }
```

`TreeBuilder` и `Conf` — это просто вымышленные классы, которые были включены в пример для демонстрации сути. Клиентский класс, которому нужен объект `TreeBuilder`, может просто вызвать статический метод `Registry::treeBuilder()`, не заботясь о сложностях инициализации. К таким сложностям могут относиться данные уровня приложения, например работа с нашим вымышленным объектом `Conf`, и большинство классов в системе должны быть изолированы от них.

Объекты системного реестра можно также использовать для тестирования. Статический метод `instance()` можно использовать для обслуживания дочернего класса для класса `Registry`, наполненного пустыми объектами. Вот как можно изменить метод `instance()`, чтобы достичь этого.

```
static function testMode( $mode=true ) {
    self::$instance = null;
    self::$testmode = $mode;
}

static function instance() {
    if ( is_null( self::$instance ) ) {
        if ( self::$testmode ) {
            self::$instance = new MockRegistry();
        } else {
            self::$instance = new self();
        }
    }
return self::$instance;
}
```

Если вам нужно испытать систему, то можете использовать режим тестирования, чтобы переключиться на сымитированный реестр. Он может обслуживать заглушки (объекты, которые имитируют реальную среду для целей тестирования) или фиктивные объекты (аналогичные объекты, которые также анализируют сделанные к ним вызовы и оценивают их корректность).

```
Registry::testMode();
$mockreg = Registry::instance();
```

Подробнее о заглушках и фиктивных объектах вы можете прочитать в главе 18, "Тестирование с помощью PHPUnit".

Реестр, область видимости и PHP

Термин *область видимости* (scope) часто используется для описания видимости объекта или значения в контексте структур кода. Период существования переменной можно также измерять во времени. Есть три уровня области видимости, кото-

рые можно рассматривать в этом смысле. Стандартом является период, который охватывает HTTP-запрос.

В PHP также предусмотрена 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 (ID), который сохраняется в cookie-файле или передается в строке запроса, используется для отслеживания информации о владельце сеанса. Поэтому можно считать, что у некоторых переменных есть сеансовая область видимости. Этим преимуществом можно воспользоваться, сохраняя некоторые объекты между запросами и тем самым избавляя себя от необходимости обращаться к базе данных. Конечно, вы должны быть внимательны, чтобы не получить в итоге несколько версий одного и того же объекта. Поэтому вы должны также подумать о стратегии блокировки, когда во время обработки сеанса окажется, что аналогичный объект уже существует в базе данных.

В других языках, особенно Java и Perl (который работает с модулем ModPerl Apache), существует понятие области видимости приложения. Переменные, занимающие это пространство, доступны всем экземплярам объектов приложения. Это совсем не свойственно PHP, но в больших приложениях очень удобно иметь доступ к пространству всего приложения, чтобы иметь возможность обращаться к переменным конфигурации. Вы можете создать класс реестра, который будет эмулировать область видимости приложения, но при этом вы должны иметь в виду некоторые очень существенные предостережения.

На рис. 12.3 показана возможная структура для классов `Registry`, которая работает на трех описанных мной уровнях.

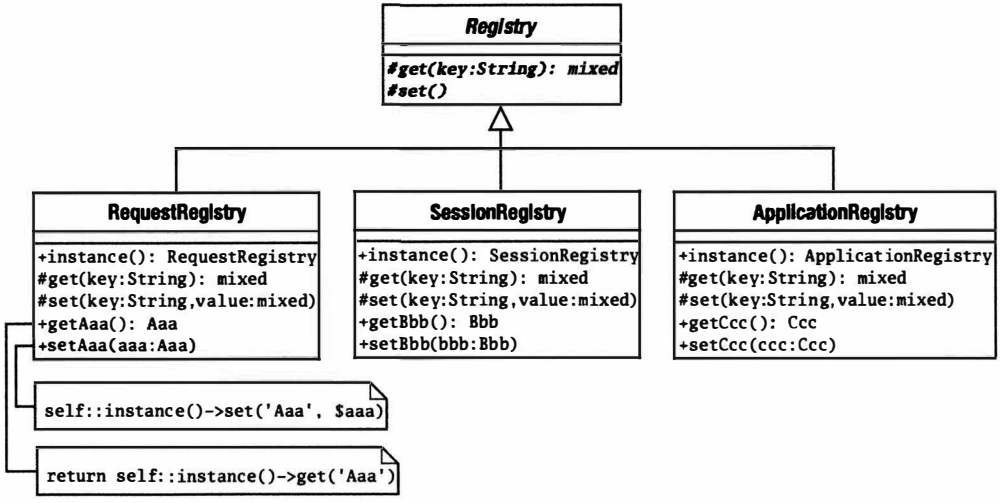


Рис. 12.3. Реализация классов реестра для различных областей видимости

В базовом классе определены два защищенных (`protected`) метода, `get()` и `set()`. Они недоступны клиентскому коду, потому что мы хотим задавать обязательный тип для операций получения (get) и установки (set). В этом базовом классе могут быть определены и другие общедоступные методы, такие как `isEmpty()`, `isPopulated()` и `clear()`, но это я оставляю вам в качестве упражнения.

На заметку. В реальной системе вам может понадобиться расширить эту структуру, чтобы включить в нее еще один уровень наследования. Вы можете поддерживать методы `get()` и `set()` в реализующих их классах, но специализировать общедоступные методы `getAaa()` и `setAaa()`, поместив их в классы, специфичные для выполняемой задачи. Эти новые реализации должны быть синглтонами. Таким образом, вы сможете повторно использовать основные операции сохранения и извлечения в различных приложениях.

Вот код абстрактного класса.

```
namespace woo\base;

abstract class Registry {
   abstract protected function get( $key );
   abstract protected function set( $key, $val );
}
```

На заметку. Обратите внимание на то, что в этом примере используются пространства имен. Поскольку в текущей главе я создам полное, хотя и простое, корпоративное приложение, имеет смысл воспользоваться иерархией пакетов и ощутить преимущества компактности и ясности, которые привносят в проект пространства имен.

Класс уровня запроса довольно прост. В другой вариации нашего предыдущего примера мы закрываем доступ к единственному экземпляру класса `Registry` и предоставляем статические методы для установки и получения объектов. Помимо этого, все остальное — только вопрос работы с ассоциативным массивом.

```
namespace woo\base;
//...
class RequestRegistry extends Registry {
   private $values = array();
   private static $instance = null;

   private function __construct() {}

   static function instance() {
      if ( is_null(self::$instance) ) {
                  self::$instance = new self();
      }
      return self::$instance;
   }

   protected function get( $key ) {
      if ( isset( $this->values[$key] ) ) {
           return $this->values[$key];
      }
      return null;
   }

   protected function set( $key, $val ) {
      $this->values[$key] = $val;
   }

   static function getRequest() {
      $inst = self::instance();
      if ( is_null( $inst->get( "request" ) ) ) {
          $inst->set('request', new \woo\controller\Request() );
      }
```

```
        return $inst->get( "request" );
    }
}
```

В реализации на уровне сеанса просто используется встроенная поддержка сеансов в PHP.

```
namespace woo\base;
//...
class SessionRegistry extends Registry {
    private static $instance = null;

    private function __construct() {
        session_start();
    }

    static function instance() {
        if (is_null(self::$instance) ) {
                    self::$instance = new self();
        }
        return self::$instance;
    }

    protected function get( $key ) {
        if ( isset( $_SESSION[__CLASS__][$key] ) ) {
             return $_SESSION[__CLASS__][$key];
        }
        return null;
    }

    protected function set( $key, $val ) {
        $_SESSION[__CLASS__][$key] = $val;
    }

    function setDSN( $dsn ) {
        self::instance()->set("dsn", $dsn );
    }

    function getDSN( ) {
        return self::instance()->get("dsn");
    }
}
```

Как видите, в этом классе для установки и извлечения значений используется суперглобальная переменная `$_SESSION`. Мы начинаем сеанс в конструкторе, вызвав метод `session_start()`. Как всегда при использовании сеансов, вы не должны отправлять в браузер пользователя никакой текст до вызова конструктора этого класса.

Как можно было ожидать, реализация уровня приложения представляет большую проблему. Как и все примеры кодов в данной главе, приведенный ниже код — только иллюстрация, а не реальный код, взятый из корпоративного приложения.

```
namespace woo\base;
//...
class ApplicationRegistry extends Registry {
    private static $instance = null;
    private $freezedir = "data";
```

```
    private $values    = array();
    private $mtimes    = array();

    private function __construct() { }

    static function instance() {
        if ( is_null(self::$instance) ) {
                    self::$instance = new self();
        }
        return self::$instance;
    }

    protected function get( $key ) {
        $path = $this->freezedir . DIRECTORY_SEPARATOR . $key;
        if ( file_exists( $path ) ) {
            clearstatcache();
            $mtime=filemtime( $path );
            if ( ! isset($this->mtimes[$key] ) ) {
                $this->mtimes[$key]=0;
            }
            if ( $mtime > $this->mtimes[$key] ) {
                $data = file_get_contents( $path );
                $this->mtimes[$key]=$mtime;
                return ($this->values[$key]=unserialize( $data ));
            }
        }
        if ( isset( $this->values[$key] ) ) {
            return $this->values[$key];
        }
        return null;
    }

    protected function set( $key, $val ) {
        $this->values[$key] = $val;
        $path = $this->freezedir . DIRECTORY_SEPARATOR . $key;
        file_put_contents( $path, serialize( $val ) );
        $this->mtimes[$key]=time();
    }

    static function getDSN() {
        return self::instance()->get('dsn');
    }

    static function setDSN( $dsn ) {
        return self::instance()->set('dsn', $dsn);
    }

    static function getRequest() {
        $inst = self::instance();
        if ( is_null( $inst->request ) ) {
                    $inst->request = new \woo\controller\Request();
        }
        return $inst->request;
    }
}
```

В этом классе используется сериализация для сохранения и восстановления значений отдельных свойств. В функции `get()` проверяется существование соответствующего файла, содержащего сериализованные значения. Если этот файл существует и был изменен со времени последнего чтения, то в методе считывается его содержимое и восстанавливается значение переменной. Поскольку нерационально сохранять в отдельном файле значения каждой сериализуемой переменной, можно применить другой подход — поместить значения всех свойств в один файл сохранения. В методе `set()` изменяется значение свойства, на которое указывает переменная `$key`, и локально, и в файле сохранения. В нем обновляется значение свойства `$mtimes`. Это массив с информацией о времени изменения, который используется для проверки файлов сохранения. Затем, при вызове метода `get()`, значение `$mtimes` сравнивается со временем модификации файла, чтобы узнать, был ли он изменен со времени последней записи этого объекта.

Обратите внимание, что представленный здесь метод `getRequest()` идентичен одноименному методу класса `RequestRegistry`, относящемуся к уровню запроса, который был рассмотрен выше. В обоих случаях не существует соответствующего метода `setRequest()`. Здесь мы отошли от практики предоставления сторонним объектам возможности создавать собственные версии объектов `Request` и записывать их в реестр. Вместо этого мы предложили механизм, благодаря которому системный реестр становится единым источником общего объекта `Request`, и предоставили гарантию, что только один экземпляр этого объекта будет доступен всем элементам нашего приложения.

Описанный выше подход также крайне полезен для целей тестирования. Благодаря общему реестру мы можем подменить объект `Request` перед запуском тестовой версии приложения. Это позволит нам создать различные внешние условия, на которые должна отреагировать программа, и посмотреть на получаемый результат. Также обратите внимание, что объект `Request` сохраняется в объекте `ApplicationRegistry` в обычном свойстве `$request` и не сохраняется в файле. Все дело в том, что нам *не нужно*, чтобы именно этот объект сохранял свое состояние между запросами!

Если в вашей инсталляции PHP активизировано расширение `apc`, то вы можете использовать его функции для реализации реестра приложения. Приведем упрощенный пример.

```
namespace woo\base;
// ...
class MemApplicationRegistry extends Registry {
   private static $instance = null;
   private $values=array();
   private $id;

   private function __construct() { }

   static function instance() {
      if ( is_null(self::$instance) ) {
                  self::$instance = new self();
      }
      return self::$instance;
   }

   protected function get( $key ) {
      return \apc_fetch( $key );
   }
```

```
    protected function set( $key, $val ) {
        return \apc_store( $key, $val );
    }

    static function getDSN() {
        return self::instance()->get("dsn");
    }

    static function setDSN( $dsn ) {
        return self::instance()->set("dsn", $dsn);
    }
}
```

Если вы собираетесь использовать вариант этого примера кода, обязательно прочитайте следующий раздел: есть некоторые серьезные моменты, которые следует учесть.

На заметку. Поскольку расширение APC не входит в стандартную поставку PHP, вам нужно самостоятельно его установить. Как это сделать, описано в руководстве по PHP по адресу `http://php.net/manual/ru/apc.installation.php`.

Результаты

Поскольку и `SessionRegistry`, и `ApplicationRegistry` сериализуют данные в файл, важно еще раз сформулировать очевидный факт: объекты, извлекаемые в разных запросах, являются идентичными копиями и не ссылаются на один и тот же объект. Это не должно иметь значения для `SessionRegistry`, потому что к объекту в каждом случае обращается один и тот же пользователь. Но для `ApplicationRegistry` это может быть серьезной проблемой. Сохраняя данные беспорядочно, вы можете оказаться в ситуации, в которой два процесса будут конфликтовать. Рассмотрим описанную ниже последовательность действий.

```
Процесс 1 извлекает объект
Процесс 2 извлекает объект
Процесс 1 изменяет объект
Процесс 2 изменяет объект
Процесс 1 сохраняет объект
Процесс 2 сохраняет объект
```

Изменения, выполненные `Процессом 1`, затираются в результате сохранения `Процессом 2`. Если вы действительно хотите создать совместно используемое пространство для данных, реализуйте в классе `ApplicationRegistry` схему взаимоблокировки для предотвращения подобных противоречий. Существует и альтернативный вариант: рассматривать класс `ApplicationRegistry` в большей степени как ресурс "только для чтения". Именно таким образом я использую этот класс в последующих примерах в данной главе. Первоначально в нем устанавливаются данные, и после этого взаимодействие с ним происходит по принципу "только чтение". Если файл хранилища не найден, то в коде вычисляются новые значения и записываются в этот файл. Следовательно, перезагрузку данных конфигурации можно инициировать только путем удаления файла хранилища. Более того, вы можете усовершенствовать класс так, чтобы он работал в режиме "только для чтения".

Еще один важный момент, о котором нужно помнить, — не каждый объект подходит для сериализации. В частности, если вы будете сохранять ресурсы любого типа (например, дескриптор подключения к базе данных), то он не будет сериали-

зован. Вам придется разработать стратегии, обрабатывающие такой дескриптор во время сериализации и восстанавливающие его при десериализации.

Итак, какую же стратегию выбрать? На практике я почти всегда использую самый простой вариант — реестр, работающий только с запросами. Разумеется, в разрабатываемой программной системе я всегда использую только один тип реестра. Это позволяет избежать некоторых трудноуловимых ошибок! Механизм кеширования, рассмотренный выше в примере с классом `ApplicationRegistry`, позволяет преодолеть недостатки шаблона Front Controller, заключающиеся в высоких накладных расходах, связанных с синтаксическим анализом запутанного файла конфигурации в каждом запросе. В реальных приложениях механизм кеширования следует реализовать отдельно, чтобы максимально упростить реестр и ограничить его функциональные возможности только запросами. Тем не менее в данном случае класс `ApplicationRegistry` служит поставленным мною целям, поэтому я продолжу с ним работать.

На заметку. Один из способов управления сериализацией — реализовать "магические" методы `__sleep()` и `__wakeup()`. Метод `__sleep()` вызывается автоматически, когда объект сериализуется. Вы можете использовать его для выполнения любой операции очистки перед сохранением объекта. Он должен вернуть массив строк, представляющих поля, которые вы хотите сохранить. Метод `__wakeup()` вызывается при десериализации объекта. Вы можете использовать его, чтобы восстановить дескриптор открытого файла или подключения к базе данных, которые, возможно, использовал объект во время сохранения.

Хотя сериализация в PHP довольно эффективна, вы должны быть внимательны в отношении того, что сохраняете. Простой на вид объект может содержать ссылку на огромный набор объектов, полученных из базы данных.

Объекты типа `Registry` делают свои данные глобально доступными. Это означает, что любой класс, действующий как клиент для реестра, будет проявлять зависимость, не объявленную в его интерфейсе. Это может стать серьезной проблемой, если слишком много данных в вашей системе будут зависеть от объектов типа `Registry`. Поэтому объекты типа `Registry` лучше всего использовать сравнительно редко, для четко определенного набора элементов данных.

Уровень представления данных

Когда запрос обращается к вашему приложению, оно должно интерпретировать его требования, затем выполнить всю необходимую обработку и наконец вернуть ответ. Для простых сценариев весь этот процесс обычно полностью происходит внутри самого представления, и только сложные вычислительные процедуры и повторяемый код помещаются в библиотеки.

На заметку. *Представление* — это отдельный элемент на уровне отображения данных. Обычно это PHP-страница (или набор составных элементов отображения), главная обязанность которой — отображать данные и обеспечивать механизм, посредством которого пользователь может генерировать новые запросы. Это также может быть один из шаблонов при использовании системы шаблонов, такой как Smarty.

По мере увеличения размера приложения стандартная стратегия становится менее подходящей для обработки запросов, вызова логики приложения, поскольку логика выборки представления обязательно дублируется от одной страницы к другой.

В этом разделе мы рассмотрим стратегии управления этими тремя ключевыми обязанностями уровня представления. Поскольку границы между уровнем представления и уровнем команд и управления, как правило, довольно размыты, имеет смысл рассматривать их вместе под общим названием "уровень представления".

Шаблон Front Controller

Этот шаблон диаметрально противоположен традиционному PHP-приложению с его несколькими точками входа. Шаблон Front Controller предоставляет центральную точку доступа для обработки всех входящих запросов и в конечном итоге для вывода результатов пользователю передает их уровню представления. Это основной шаблон из семейства шаблонов корпоративных приложений Java. Он очень подробно описан в книге *Core J2EE Patterns*, которая остается одним из самых авторитетных ресурсов по шаблонам корпоративных приложений. Но нельзя сказать, что все PHP-сообщество является приверженцем этого шаблона, отчасти из-за издержек инициализации, которые иногда имеют место.

В большинстве систем, которые я пишу, как правило, используется шаблон Front Controller. Я могу его не использовать полностью в качестве основы, но я знаю, какие шаги необходимо сделать, чтобы развить проект в реализацию Front Controller, если мне понадобится та степень гибкости, которую он предоставляет.

Проблема

Когда запросы обрабатываются в нескольких точках системы, очень трудно избежать дублирования в коде. Возможно, вам нужно аутентифицировать пользователя, перевести элементы интерфейса на другие языки или просто получить доступ к общим данным. Если для запроса требуется выполнять общие действия от одного представления к другому, то вы будете заниматься копированием и вставкой кода. Это усложнит внесение изменений, поскольку самое простое исправление нужно будет повторить в нескольких местах приложения. Поэтому может случиться так, что одни части кода перестанут соответствовать другим. Конечно, можно централизовать общие операции, выделив их в библиотечный код, но все равно останутся вызовы библиотечных функций или методов, разбросанные по всему приложению.

Трудность в управлении продвижением от одного представления к другому — это еще одна проблема, которая может возникнуть в системе, где управление распределено между представлениями. В сложной системе подача запроса в одном представлении может привести к любому количеству страниц результатов в соответствии с входными данными и успешностью выполнения требуемых операций на уровне логики приложения. Логика перемещения от одного представления к другому может стать запутанной, особенно если одно и то же представление используется в различных потоках.

Реализация

По сути, шаблон Front Controller определяет центральную точку входа для каждого запроса. Он обрабатывает запрос и использует его, чтобы выбрать операцию для выполнения. Операции обычно определяются в специальных объектах типа `command`, организованных согласно шаблону Command.

На рис. 12.4 показан общий вид реализации шаблона Front Controller.

На самом деле вы, скорее всего, будете использовать несколько вспомогательных классов, исходя из логики приложения, но давайте начнем с основных участников. Вот определение простого класса `Controller`.

```
namespace woo\controller;
//..
class Controller {
   private $applicationHelper;
```

```
    private function __construct() {}

    static function run() {
        $instance = new Controller();
        $instance->init();
        $instance->handleRequest();
    }

    function init() {
        $applicationHelper =
                ApplicationHelper::instance();
        $applicationHelper->init();
    }

    function handleRequest() {
        $request = \woo\base\ApplicationRegistry::getRequest();
        $cmd_r   = new woo\command\CommandResolver();
        $cmd     = $cmd_r->getCommand( $request );
        $cmd->execute( $request );
    }
}
```

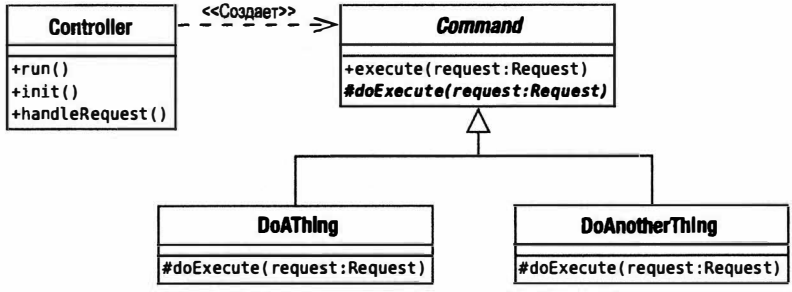


Рис. 12.4. Класс `Controller` и иерархия команд

Несмотря на то что класс `Controller` очень простой и в нем опущена обработка ошибок, больше в этом классе не должно быть ничего. Контроллер находится на верхушке системы, делегируя полномочия другим классам. Именно эти другие классы выполняют большую часть работы.

Статический метод `run()` введен для удобства; в нем вызываются методы `init()` и `handleRequest()`. Поскольку конструктор нашего класса закрытый, для клиентского кода существует только единственная возможность начать выполнение нашей системы — вызвать метод `run()`. Обычно я делаю это в файле `index.php`, который содержит только пару строк кода.

```
require( "woo/controller/Controller.php" );
\woo\controller\Controller::run();
```

На самом деле разница между методами `init()` и `handleRequest()` почти неуловима в PHP. В некоторых языках метод `init()` будет выполняться только при запуске приложения, а метод `handleRequest()` или его эквивалент — при каждом запросе пользователя. В нашем же классе мы придерживаемся такого же различия между инициализацией и обработкой запросов, хотя метод `init()` вызывается для каждого запроса.

Метод `init()` получает экземпляр класса под названием `ApplicationHelper`. В этом классе хранятся данные конфигурации для всего приложения. Метод `init()` нашего класса вызывает метод в классе `ApplicationHelper`, который также называется `init()`. В нем, как вы увидите позже, инициализируются данные, используемые приложением.

В методе `handleRequest()` используется класс `CommandResolver`, чтобы получить объект типа `Command`, который он запускает, вызывая метод `Command::execute()`.

Класс ApplicationHelper

Этот класс необязателен для шаблона Front Controller. Но в большинстве реализаций приложений используются данные конфигурации, поэтому мы должны обсудить стратегию для этого. Вот пример простого класса `ApplicationHelper`.

```
namespace woo\controller;
//..
class ApplicationHelper {
   private static $instance = null;
   private $config = "data/woo_options.xml";

   private function __construct() {}

   static function instance() {
      if (is_null( self::$instance ) ) {
         self::$instance = new self();
      }
      return self::$instance;
   }

   function init() {
      $dsn = \woo\base\ApplicationRegistry::getDSN( );
         if ( ! is_null( $dsn ) ) {
            return;
      }
      $this->getOptions();
}

   private function getOptions() {
      $this->ensure( file_exists( $this->config ),
                     "Файл конфигурации не найден" );
      $options = @SimpleXml_load_file( $this->config );
      $dsn = (string)$options->dsn;
      $this->ensure( $options instanceof SimpleXMLElement,
                      "Файл конфигурации запорчен" );
      $this->ensure( $dsn, "DSN не найден" );
      \woo\base\ApplicationRegistry::setDSN( $dsn );
     // Установите другие значения
   }

   private function ensure( $expr, $message ) {
      if ( ! $expr ) {
         throw new \woo\base\AppException( $message );
      }
   }
}
```

Этот класс просто читает файл конфигурации и делает значения доступными для клиентского кода. Как видите, это еще один синглтон, с помощью которого конфигурационные данные легко можно сделать доступными для любого класса в системе. Существует альтернативный вариант: сделать класс `ApplicationHelper` стандартным и гарантировать, что он передается всем заинтересованным в нем объектам. Компромиссы, на которые здесь приходится идти, я уже обсуждал выше в данной главе и в главе 9.

Тот факт, что в классе `ApplicationHelper` мы используем `ApplicationRegistry`, предполагает выполнение рефакторинга кода. Возможно, стоит сделать сам класс `ApplicationHelper` системным реестром, вместо того чтобы создавать в системе два синглтона с частично совпадающими обязанностями. Для этого, возможно, придется задействовать рефакторинг кода, который был описан в предыдущем разделе (отделение основных функций класса `ApplicationRegistry` от сохранения и извлечения объектов, специфичных для конкретного приложения). Предлагаю вам сделать это самостоятельно!

Итак, метод `init()` отвечает за загрузку данных конфигурации. Фактически он проверяет `ApplicationRegistry`, чтобы узнать, сохранены ли данные в кеш-памяти. Если в объект `Registry` уже загружены данные, то метод `init()` вообще ничего не делает. Это очень полезно для приложений, в которых выполняется много очень ресурсоемких инициализаций. Сложная настройка может быть приемлемой в тех языках, в которых отделяется инициализация приложения от обработки отдельных запросов. В PHP вам нужно минимизировать процесс инициализации.

Кеширование — очень эффективный метод, гарантирующий, что сложные и занимающие много времени процессы инициализации будут выполнены только при начальном запросе (вероятно, всего один раз), а все последующие запросы воспользуются хранимыми в памяти готовыми данными. Разумеется, в этом случае кеширование дает нам только небольшой выигрыш. Чуть ниже в этой главе будет показано, где именно наступает реальный выигрыш.

Если это *действительно* первый запуск приложения (или если кеш-память была очищена, — грубый, но эффективный способ заставить снова прочитать данные конфигурации), то вызывается метод `getOptions()`.

В реальном приложении, вероятно, нужно будет выполнить намного больше работы, чем показано в данном примере. В нашей же версии все заканчивается получением значения параметра DSN. В методе `getOptions()` сначала проверяется, существует ли файл конфигурации (путь к нему сохраняется в свойстве `$config`). Затем он пытается загрузить XML-данные из файла и определить значение параметра DSN.

На заметку. В этих примерах в классах `ApplicationRegistry` и `ApplicationHelper` используются жестко закодированные пути для работы с файлами. Очевидно, что на практике пути к этим файлам должны быть изменяемыми и определяться через системный реестр или объект конфигурации. Реальные пути могут быть установлены во время процесса инсталляции с помощью инструмента построения, такого как PEAR или Phing (подробнее об этих инструментах читайте в главах 15 и 19).

Обратите внимание на то, что в классе `ApplicationHelper` используется особый прием для генерирования исключений. Вместо того чтобы "усеивать" код условными операторами и операторами `throw` наподобие

```
if ( ! file_exists( $this->config ) ) {
   throw new \woo\base\AppException("Файл конфигурации не найден" );
}
```

в этом классе в методе под названием `ensure()` централизованы проверка условия и вызов оператора `throw`. Вы можете убедиться, что условие выполняется, или же в противном случае выдать исключение с помощью единственного (хотя и разбитого на две строки) оператора.

```
$this->ensure( file_exists( $this->config ),
                "Файл конфигурации не найден" );
```

Подход с использованием кеш-памяти, выбранный здесь, дает нам два преимущества. Система может поддерживать простой в использовании файл конфигурации в формате XML, а кеширование означает, что к его значениям можно получать доступ почти без задержки. Конечно, если ваши конечные пользователи — тоже программисты или если вы не собираетесь слишком часто менять конфигурацию, можете включить определение структур данных PHP непосредственно во вспомогательный класс (или в отдельный файл, который включается в проект). И хотя это рискованный подход, он самый быстрый. Для получения данных конфигурации, представленных в виде пар "имя-значение", можно также использовать функцию PHP `parse_ini_file()`. Поскольку этот метод работает быстрее, чем синтаксический анализ XML-файла, именно его я использую в качестве основного для доступа к параметрам конфигурации в моих приложениях. Однако в классе `ApplicationHelper` ниже в этой главе мы будем использовать намного более сложные наборы данных.

Класс `CommandResolver`

Контроллеру нужен способ, позволяющий решить, как интерпретировать HTTP-запрос, чтобы в результате можно было вызывать нужный код для обработки этого запроса. Конечно, можно легко включить эту логику в сам класс `Controller`, но я предпочитаю использовать для этой цели специальный класс. В случае необходимости это позволит легко сделать рефакторинг для полиморфизма.

В шаблоне Front Controller логика приложения часто вызывается путем запуска объекта `Command` (о шаблоне Command говорилось в главе 11). Этот объект обычно выбирается в соответствии со значением параметра запроса или структурой самого URL (например, вы можете использовать настройки сервера Apache, чтобы заменить конкретные URL ключами, которые используются для вызова соответствующего объекта типа `Command`). В приведенных примерах я буду использовать простой параметр: `cmd`.

Существует несколько способов использования заданного параметра для выбора команды. Вы можете сопоставить параметр с конкретным методом через файл конфигурации или структуру данных (*логическая* стратегия). Или можете отобразить его непосредственно на файл класса, расположенный в файловой системе (*физическая* стратегия).

Логическая стратегия более гибкая, но в то же время и более трудоемкая, если говорить о ее установке и поддержке. Пример этого подхода приведен в разделе "Шаблон Application Controller".

Пример фабрики команд, в котором использовалась физическая стратегия, вы видели в предыдущей главе. Приведем небольшой вариант этого примера, в котором используется рефлексия с целью повышения безопасности.

```
namespace woo\command;
//...
class CommandResolver {
   private static $base_cmd = null;
   private static $default_cmd = null;
```

```
   function __construct() {
      if ( is_null( self::$base_cmd ) ) {
         self::$base_cmd    = new \ReflectionClass( "\woo\command\Command" );
         self::$default_cmd = new DefaultCommand();
      }
   }

   function getCommand( \woo\controller\Request $request ) {
      $cmd = $request->getProperty( 'cmd' );
      $sep = DIRECTORY_SEPARATOR;
      if ( ! $cmd ) {
         return self::$default_cmd;
      }
      $cmd=str_replace( array('.', $sep), "", $cmd );
      $filepath = "woo{$sep}command{$sep}{$cmd}.php";
      $classname = "woo\\command\\$cmd";
      if ( file_exists( $filepath ) ) {
         @require_once( $filepath );
         if ( class_exists( $classname) ) {
            $cmd_class = new ReflectionClass($classname);
               if ( $cmd_class->isSubClassOf( self::$base_cmd ) ) {
                  return $cmd_class->newInstance();
               } else {
                  $request->addFeedback( "Объект Command команды '$cmd' не най-
ден" );
               }
         }
      }
      $request->addFeedback( "Команда '$cmd' не найдена" );
      return clone self::$default_cmd;
   }
}
```

В этом простом классе выполняется поиск параметра запроса `cmd`. Предположим, что он найден и соответствует реальному файлу класса в каталоге команд *и* что этот файл класса содержит нужный тип класса. Тогда метод `getCommand()` создает и возвращает экземпляр соответствующего класса.

Если какие-либо из этих условий не удовлетворяются, то метод `getCommand()` возвращает стандартный объект `Command`.

Возможно, вас интересует, почему в этом коде принимается на веру то, что конструктору класса `Command`, который он находит, не требуется передавать никаких параметров.

```
if ( $cmd_class->isSubClassOf( self::$base_cmd ) ) {
   return $cmd_class->newInstance();
}
```

Ответ нужно искать в сигнатуре самого класса `Command`.

```
namespace woo\command;
//...
abstract class Command {
   final function __construct() { }

   function execute( \woo\controller\Request $request ) {
      $this->doExecute( $request );
```

```
    }

    abstract function doExecute( \woo\controller\Request $request );
}
```

Объявляя метод конструктора как `final`, мы делаем невозможным для дочернего класса его переопределение. Поэтому ни один класс `Command` никогда не потребует аргументы для своего конструктора.

Помните, что вы никогда не должны использовать входные данные, полученные от пользователя, не проверив их. Я включил проверку, чтобы убедиться, что в предоставленной строке `"$cmd"` не находятся элементы пути. В результате будут вызваны только файлы из нужного каталога, а не что-то вроде `../../../tmp/DodgyCommand.php`. Вы можете сделать код еще безопаснее, принимая только командные строки, которые соответствуют значениям в файле конфигурации.

Создавая классы команд, старайтесь по возможности не допускать включения в них логики приложения. Как только они начнут выполнять то, что должно делать само приложение, они превратятся в подобие запутанного сценария транзакций, и тогда вскоре появится дублирование. Команды — это что-то вроде станции ретрансляции: они должны интерпретировать запрос, вызвать логику приложения для выполнения операций с какими-то объектами, а затем передать данные для отображения уровню представления. Как только они начнут делать что-то более сложное, вероятно, придет время для рефакторинга. Плюсом является то, что провести рефакторинг достаточно легко. Ведь не трудно найти место, где команда пытается сделать слишком много, и обычно очевидно, что нужно сделать для решения проблемы. Переместите функции вниз в класс фасада или самой предметной области.

Запрос

Запросы магическим образом обрабатываются средствами PHP, и их параметры аккуратно складываются в суперглобальные массивы. Наверное, вы заметили, что мы для представления запроса до сих пор использовали класс. Объект `Request` передается `CommandResolver`, а затем — `Command`.

Почему мы не позволяем этим классам напрямую обращаться к элементам массивов `$_REQUEST`, `$_POST` или `$_GET`? Конечно, мы могли бы это сделать, но, централизуя операции запросов в одном месте, мы открываем для себя новые возможности. Например, можно применить фильтры к входящим запросам. Или, как показано в следующем примере, можно получить параметры не из HTTP-запроса, что позволит приложению запускаться из командной строки или из тестового сценария. Конечно, если в приложении используются сеансы, то вам придется предоставить альтернативный механизм сохранения, чтобы его можно было использовать в контексте командной строки. Тут вам пригодится шаблон Registry, который позволит генерировать различные классы `Registry` в соответствии с контекстом приложения.

Объект `Request` — это также полезное хранилище для данных, которые нужно передать уровню представления. В этом отношении объект `Request` может также использоваться для генерации ответа.

Приведем пример простого класса `Request`.

```
namespace woo\controller;
//...
class Request {
    private $properties;
    private $feedback = array();

    function __construct() {
```

```
        $this->init();
    }

    function init() {
        if ( isset( $_SERVER['REQUEST_METHOD'] ) ) {
            $this->properties = $_REQUEST;
            return;
        }
        foreach( $_SERVER['argv'] as $arg ) {
            if ( strpos( $arg, '=' ) ) {
                list( $key, $val ) = explode( "=", $arg );
                $this->setProperty( $key, $val );
            }
        }
    }

    function getProperty( $key ) {
        if ( isset( $this->properties[$key] ) ) {
            return $this->properties[$key];
        }
        return null;
    }

    function setProperty( $key, $val ) {
        $this->properties[$key] = $val;
    }

    function addFeedback( $msg ) {
        array_push( $this->feedback, $msg );
    }

    function getFeedback( ) {
        return $this->feedback;
    }

    function getFeedbackString( $separator="\n" ) {
        return implode( $separator, $this->feedback );
    }
}
```

Как видите, большую часть этого класса занимают механизмы установки и получения свойств. Метод `init()` отвечает за наполнение закрытого массива `$properties`. Обратите внимание на то, что он работает с аргументами командной строки, так же как и с параметрами HTTP-запроса. Это чрезвычайно полезно, когда речь идет о тестировании и отладке.

Как только будет получен объект `Request`, у вас должна быть возможность обратиться к параметру HTTP-запроса с помощью метода `getProperty()`, которому передается строка ключа, а он возвращает соответствующее значение (сохраненное в массиве `$properties`). Вы можете также добавить данные с помощью метода `setProperty()`.

Этот класс также управляет массивом `$feedback`. Это простой канал связи, через который классы-контроллеры могут передавать сообщения пользователю.

Команда

Вы уже знакомы с базовым классом `Command`, а в главе 11 шаблон Command описывался подробно, поэтому нет необходимости слишком углубляться в эту тему. Но давайте подытожим наши знания, рассмотрев простую реализацию объекта типа `Command`.

```
namespace woo\command;
//...

class DefaultCommand extends Command {
   function doExecute( \woo\controller\Request $request ) {
      $request->addFeedback( "Добро пожаловать в Woo!" );
       include( "woo/view/main.php");
  }
}
```

Этот объект типа `Command` автоматически выбирается классом `CommandResolver`, если не было получено явного запроса на определенный объект типа `Command`.

Как вы, наверное, заметили, в абстрактном базовом классе `Command` реализован метод `execute()`, из которого вызывается метод `doExecute()`, реализованный в дочернем классе. Это позволяет нам добавлять код для начальной установки и очистки ко всем командам, 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 используется для установки приветственного сообщения.

И наконец команда передает управление странице представления, просто вызывая метод `include()`. Вставка кода представления в классах типа `Command` — это самый простой механизм диспетчеризации, но для небольших систем он подходит идеально. О более гибкой стратегии говорится в разделе "Шаблон Application Controller".

В файле `main.php` содержатся HTML-код страницы и вызов метода `getFeedback()` объекта `Request` для отображения сообщений, предназначенных для пользователя (вскоре я расскажу о представлениях более подробно). Теперь у нас есть все компоненты, чтобы запустить систему. Вот 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. Давайте рассмотрим весь процесс, который привел к этому результату.

Описание

Подробное описание классов, сделанное в данном разделе, может дать неверное представление о простоте шаблона Front Controller. На рис. 12.5 показана диаграмма последовательности, иллюстрирующая жизненный цикл запроса.

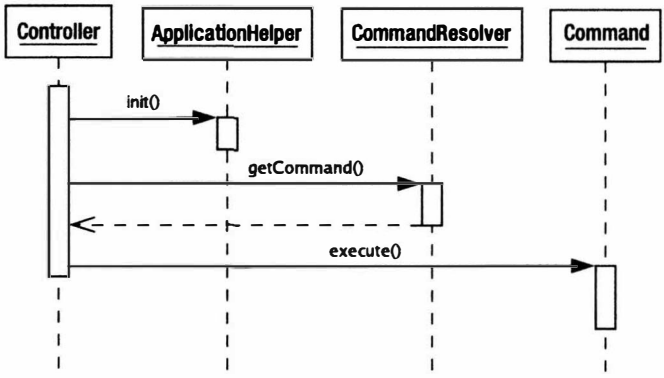


Рис. 12.5. Шаблон Front Controller в действии

Как видите, Front Controller делегирует инициализацию объекту `Application Helper` (который использует кеширование, чтобы сократить время на обработку параметров). Затем `Controller` получает объект `Command` от объекта `CommandResolver`. И наконец он вызывает метод `Command::execute()`, чтобы запустить логику приложения.

В этой реализации шаблона сам объект `Command` отвечает за делегирование полномочий уровню представления. Улучшенный вариант реализации будет приведен в следующем разделе.

Результаты

Шаблон Front Controller — не для слабонервных. Для работы с ним нужно уделить много внимания предварительной разработке, прежде чем вы реально почувствуете все его преимущества. И если проект достаточно мал (в результате структура Front Controller может оказаться сложнее всей остальной системы) или нужно очень быстро его реализовать, то это выливается в серьезный недостаток.

Но успешно применив Front Controller в одном проекте, вы обнаружите, что можете повторно использовать его в других проектах с молниеносной скоростью. Вы можете поместить большинство его функций в библиотечный код, создав для себя эффективную повторно используемую инфраструктуру.

Еще один недостаток данного шаблона — обязательное условие, чтобы вся информация о конфигурации загружалась для каждого запроса. В некоторой степени от этого страдают все шаблоны, но для Front Controller часто требуется дополнительная информация, такая как логические соответствия команд и представлений.

Указанный недостаток можно существенно уменьшить путем кеширования данных конфигурации. Самый эффективный способ это сделать — организовать конфигурационные данные в виде встроенных структур данных PHP. Такой подход прекрасно работает, если вы программист и единолично поддерживаете систему. Но если у вас есть пользователи, не владеющие PHP, то для них нужно будет организовать файл конфигурации. Однако вы можете автоматизировать описанный выше PHP-подход, создав программу, которая считывает файл конфигурации, а за-

тем создает на его основе структуры данных PHP, которые хранятся в оперативной памяти. Как только структуры конфигурационных данных будут созданы, ваша система сможет использовать их до тех пор, пока не будут внесены изменения в исходный файл конфигурации и не возникнет потребность реконструировать кеш-память. Менее эффективный, но более простой подход был использован в классе `ApplicationRegistry` (я просто сериализовал данные).

Преимуществом шаблона Front Controller является то, что в нем централизована логика управления представлениями в системе. Это означает, что вы в одном месте можете осуществлять контроль над тем, как обрабатываются запросы и выбираются представления (но в любом случае в одном наборе классов). Это позволяет сократить дублирование и уменьшить вероятность ошибок.

У шаблона Front Controller также много возможностей для расширения. Когда основная часть приложения будет создана и готова для работы, вы сможете очень легко добавлять новые классы типа `Command` и представления.

В описанном выше примере сами команды управляли диспетчеризацией собственных представлений. Если вы используете шаблон Front Controller с объектом, который помогает выбирать представление (и возможно, команду), то этот шаблон дает отличные возможности 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 допустимо, чтобы команды вызывали собственные представления, но эту ситуацию нельзя назвать идеальной. Гораздо предпочтительнее отделить команды от уровня представления данных, насколько это возможно.

Шаблон Application Controller (контроллер приложения) берет на себя ответственность за соответствие запросов командам и команд их представлениям. Такое разделение означает, что в приложении можно будет очень легко изменять наборы представлений, не трогая базовый код. Это также позволяет владельцу системы изменять ход выполнения приложения, опять-таки, не затрагивая его внутреннее содержание. Учитывая логическую систему интерпретации объектов типа `Command`, этот шаблон также упрощает возможность использования одной и той же команды (Command) в различных контекстах в пределах системы.

Проблема

Вспомните о природе проблемы нашего примера. Администратор хочет иметь возможность добавлять заведение (Venue) к системе и связывать с ним место (Space). Поэтому система может поддерживать команды `AddVenue` и `AddSpace`. Согласно примерам, рассмотренным до сих пор, эти команды выбираются с помощью прямого соответствия параметра запроса (`cmd=AddVenue`) классу (`AddVenue`).

Вообще говоря, успешный вызов команды `AddVenue` должен привести к первоначальному вызову команды `AddSpace`. Эта связь может быть жестко закодирована в самих классах, чтобы в случае успешного выполнения команды `AddVenue` она вызывала команду `AddSpace`. Затем в команду `AddSpace` может быть включено представление, содержащее форму для добавления места к заведению.

Обе команды могут быть связаны по меньшей мере с двумя различными представлениями: основным — для отображения формы ввода входных данных и экраном ошибки или благодарности ("спасибо за то, что..."). Согласно логике, которая уже обсуждалась, эти представления могут быть включены в сами классы типа

Command (для решения того, какое представление в какой ситуации следует вызывать, в них используются условные операторы).

Такой уровень жесткого кодирования приемлем, пока команды всегда используются одинаково. Но ситуация начнет ухудшаться, когда нам понадобится специальное представление для команды AddVenue и нужно будет изменить логику, по которой одна команда ведет к другой (вероятно, в выполняющемся потоке понадобится вывести дополнительный экран после успешного добавления заведения и перед началом добавления места). Если каждая из команд используется только один раз, с одной связью с другими командами и с одним представлением, то вы можете жестко закодировать связи команд одна с другой и с их представлениями. В противном случае продолжайте читать дальше!

Класс контроллера приложения может взять на себя управление этой логикой, освободив классы типа Command, чтобы они могли сосредоточиться на своей работе, т.е. обрабатывать входные данные, вызывать логику приложения и обрабатывать полученные результаты.

Реализация

Как всегда, ключ к этому шаблону — это интерфейс. Контроллер приложения — это класс (или набор классов), который Front Controller может использовать, чтобы получить команды на основе запроса пользователя и найти нужное представление, которое будет выведено после выполнения команды. Суть этой связи показана на рис. 12.6.

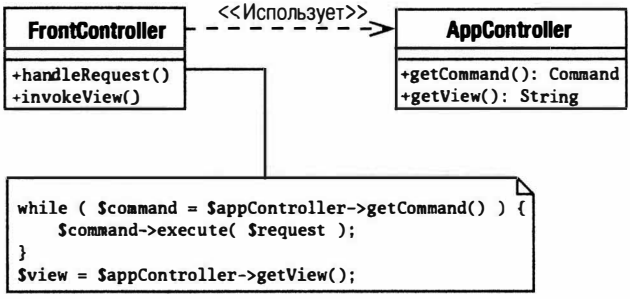


Рис. 12.6. Шаблон Application Controller

Цель данного шаблона, как и всех шаблонов в данной главе, — по возможности упростить работу для клиентского кода; отсюда и наш спартанский класс Front Controller. Но за интерфейсом мы должны развернуть реализацию. Использованный здесь подход — только один из способов это сделать. Читая данный раздел, помните, что суть шаблона в том, как взаимодействуют участники, контроллер приложения, команды и представления, а не в специфике данной реализации.

Давайте начнем с рассмотрения кода, в котором используется контроллер приложения.

Класс FrontController

Вот как FrontController может работать с классом AppController (это упрощенный код, в котором нет обработки ошибок).

```
namespace woo\controller;
// ...
function handleRequest() {
```

```
    $request = \woo\base\ApplicationRegistry::getRequest()
      $app_c = \woo\base\ApplicationRegistry::appController();

    while( $cmd = $app_c->getCommand( $request ) ) {
       $cmd->execute( $request );
    }
    $this->invokeView( $app_c->getView( $request ) );
}

function invokeView( $target ) {
    include( "woo/view/$target.php" );
    exit;
}
```

Как видите, главное отличие от предыдущего примера для шаблона Front Controller заключается в том, что здесь объекты `Command` извлекаются и выполняются в цикле. В этом коде также используется класс `AppController` для получения имени файла представления, который он должен включить. Обратите внимание на то, что в этом коде используется объект реестра для получения как объекта `AppController`, так и объектов типа `Request`.

Так как же нам все-таки перейти от использования параметра запроса `cmd` к цепочке команд и в конце концов — к представлению?

Обзор реализации

Класс `Command`, в зависимости от разных ситуаций, должен отображать разные представления. Стандартным представлением для команды `AddVenue` может быть форма для ввода данных. Если пользователь добавляет неправильный тип данных, форма может быть выведена снова или будет выведена страница с сообщением об ошибке. Если все идет хорошо и заведение в системе создано, то нам понадобится перейти к другому объекту в цепочке `Command`, вероятно, к `AddSpace`.

Объекты `Command` сообщают системе о своем текущем состоянии, устанавливая специальный код состояния. Ниже приведены коды, которые распознаются в нашей минимальной реализации (они задаются в виде статических свойств в суперклассе `Command`).

```
private static $STATUS_STRINGS = array (
      'CMD_DEFAULT'           => 0,
      'CMD_OK'                => 1,
      'CMD_ERROR'             => 2,
      'CMD_INSUFFICIENT_DATA' => 3
    );
```

Контроллер приложения находит и создает экземпляр нужного класса `Command`, используя объект `Request`. После запуска объекту `Command` будет поставлен в соответствие код состояния. Сочетание объекта `Command` и кода состояния можно затем сравнить с внутренней структурой данных, чтобы определить, какая команда должна быть запущена следующей или — если больше никакие команды не должны быть запущены — какое представление нужно отобразить.

Файл конфигурации

Владелец системы может определить, как должны совместно работать команды и представления, указав набор директив в файле конфигурации. Рассмотрим фрагмент этого файла.

```
<control>
   <view>main</view>
   <view status="CMD_OK">main</view>
   <view status="CMD_ERROR">error</view>

   <command name="ListVenues">
      <view>listvenues</view>
   </command>

   <command name="QuickAddVenue">
      <classroot name="AddVenue" />
      <view>quickadd</view>
   </command>

   <command name="AddVenue">
      <view>addvenue</view>
      <status value="CMD_OK">
          <forward>AddSpace</forward>
      </status>
   </command>

   <command name="AddSpace">
      <view>addspace</view>
      <status value="CMD_OK">
         <forward>ListVenues</forward>
     </status>
   </command>
...
</control>
```

На основе этого упрощенного XML-фрагмента показана одна из стратегий абстрагирования потока команд и его привязки к представлениям внутри классов типа `Command`. Все эти директивы содержатся внутри элемента `control`. Логика работы здесь основана на простом поиске. Самые внешние элементы являются самыми общими. Они могут быть замещены их эквивалентами внутри элементов `command`.

Итак, первый элемент, `view`, определяет стандартное представление для всех команд, если никакая другая директива не противоречит заданному в нем условию. В других элементах представления на том же уровне объявляются атрибуты состояния (которые соответствуют кодам состояния, установленным в классе `Command`). Каждый атрибут состояния представляет код, который может быть установлен объектом `Command`, чтобы сообщить о ходе выполнения задачи. Поскольку эти элементы более специфичные, чем первый элемент представления, они имеют приоритет. Если команда устанавливает код состояния `CMD_OK`, то будет выбран соответствующий файл "меню" представления, который и будет включен в приложение, если он не будет замещен еще более специфичным элементом.

После определения стандартных установок в документе определяются элементы команд. По умолчанию эти элементы соответствуют непосредственно классам типа `Command` (и их файлам классов в локальной файловой системе), как в предыдущем примере с `CommandResolver`. Поэтому, если для параметра `cmd` установлено значение `AddVenue`, выбирается соответствующий элемент в файле конфигурации. Строка `"AddVenue"` используется для создания пути к файлу класса `AddVenue.php`.

Кроме того, поддерживаются также псевдонимы команд. Поэтому, если для параметра `cmd` установлено значение `QuickAddVenue`, используется следующий элемент.

```
<command name="QuickAddVenue">
   <classroot name="AddVenue" />
   <view>quickadd</view>
</command>
```

Здесь командный элемент `QuickAddVenue` не соответствует файлу класса. Поэтому такое соответствие определено элементом `classroot`. В результате можно использовать класс `AddVenue` в контексте различных потоков команд и представлений.

Элементы `command` обрабатываются от внешних к внутренним, причем внутренние, более специфичные, имеют приоритет. Устанавливая элемент `view` внутри элемента `command`, мы гарантируем, что данный элемент `command` связан с указанным представлением.

```
<command name="AddVenue">
   <view>addvenue</view>
   <status value="CMD_OK">
      <forward>AddSpace</forward>
   </status>
</command>
```

Итак, здесь представление `addvenue` связано с командой `AddVenue` (которая, в свою очередь, определяется с помощью параметра `cmd` объекта `Request`). Это означает, что представление `addvenue.php` всегда будет включено при вызове команды `AddVenue`. Всегда, если только не выполняется условие `status`. Если класс `AddVenue` устанавливает код состояния `CMD_OK`, то стандартное представление для `Command` замещается.

Элемент `status` может просто содержать другое представление, которое включается вместо стандартного. Но здесь вступает в действие элемент переадресации `forward`. В результате переадресации на другую команду из файла конфигурации все обязанности по управлению представлениями делегируются новому элементу.

Синтаксический анализ файла конфигурации

Выше была описана достаточно гибкая модель управления отображением и логикой выполнения потока команд. Однако файл конфигурации крайне нежелательно анализировать при поступлении каждого отдельного запроса. Выше мы уже видели решение данной проблемы. Класс `ApplicationHelper` обеспечивает механизм кеширования данных конфигурации.

Вот фрагмент кода.

```
Namespace woo\controller;
//...

private function getOptions() {
   $this->ensure( file_exists( $this->config ),
                  "Файл конфигурации не найден" );
   $options = @simplexml_load_file( $this->config );
   // ...Определяем DSN...

   $map = new ControllerMap();

   foreach ( $options->control->view as $default_view ) {
      $stat_str = trim($default_view['status']);
      $status = \woo\command\Command::statuses( $stat_str );
      $map->addView( (string)$default_view, 'default', $status );
   }
```

```
    // ... Анализ остальных кодов опущен...
    \woo\base\ApplicationRegistry::setControllerMap( $map );
}
```

Синтаксический анализ XML-документа, даже с помощью великолепного пакета SimpleXML, — это довольно трудоемкий и не очень интересный процесс, поэтому подробности я рассматривать не буду. Главное, что здесь нужно отметить, — метод `getOptions()` вызывается только в случае, если параметры конфигурации не были сохранены в кеш-памяти объекта `ApplicationRegistry`.

Сохранение данных конфигурации

Итак, мы работаем с объектом кеширования `ControllerMap`. По сути, он представляет собой оболочку для трех массивов. Конечно, мы можем использовать обычные массивы PHP, но благодаря объекту `ControllerMap` мы точно знаем, что каждый массив будет следовать определенному формату. Вот определение класса `ControllerMap`.

```
namespace woo\controller;
//...
class ControllerMap {
    private $viewMap      = array();
    private $forwardMap   = array();
    private $classrootMap = array();

    function addClassroot( $command, $classroot ) {
        $this->classrootMap[$command] = $classroot;
    }

    function getClassroot( $command ) {
        if ( isset( $this->classrootMap[$command] ) ) {
            return $this->classrootMap[$command];
        }
        return null;
    }

    function addView( $view, $command='default', $status=0 ) {
        $this->viewMap[$command][$status]=$view;
    }

    function getView( $command, $status ) {
        if ( isset( $this->viewMap[$command][$status] ) ) {
            return $this->viewMap[$command][$status];
        }
        return null;
    }

    function addForward( $command, $status=0, $newCommand ) {
        $this->forwardMap[$command][$status]=$newCommand;
    }

    function getForward( $command, $status ) {
        if ( isset( $this->forwardMap[$command][$status] ) ) {
            return $this->forwardMap[$command][$status];
        }
        return null;
```

```
    }
}
```

Свойство `$classrootMap` — это просто ассоциативный массив, с помощью которого устанавливается соответствие идентификатора команды (т.е. имени командного элемента в файле конфигурации) корню имени класса типа `Command` (т.е. `AddVenue`, а не `\woo\command\AddVenue`). Данный массив используется для определения реального имени файла класса команды, если в параметре `cmd` указан ее псевдоним. Он наполняется во время синтаксического анализа файла конфигурации путем вызова метода `addClassroot()`.

С помощью двухмерных массивов `$forwardMap` и `$viewMap` определяются сочетания команд и кодов состояний.

Напомним фрагмент файла конфигурации.

```
<command name="AddVenue">
   <view>addvenue</view>
   <status value="CMD_OK">
      <forward>AddSpace</forward>
   </status>
</command>
```

С помощью приведенного ниже вызова во время синтаксического анализа файла конфигурации в свойство `$viewMap` добавляется нужный элемент.

```
$map->addView( 'addvenue', 'AddVenue', 0 );
```

А с помощью этого вызова задается свойство `$forwardMap`.

```
$map->addForward( 'AddVenue', 1, 'AddSpace' );
```

Указанные выше массивы сочетаний команд и кодов состояния используются в классе контроллера приложения при выполнении поиска в определенном порядке. Предположим, команда `AddVenue` вернула `"CMD_OK"` (который соответствует числу `1`, в то время как числу `0` соответствует код `"CMD_DEFAULT"`). Контроллер приложения будет начинать поиск в массиве `$forwardMap` от самого специфичного сочетания команды и кода состояния до самого общего. Первое найденное соответствие станет строкой команды, которая возвращается.

```
$viewMap['AddVenue'][1]; // AddVenue CMD_OK [НАЙДЕНО]
$viewMap['AddVenue'][0]; // AddVenue CMD_DEFAULT
$viewMap['default'][1];  // DefaultCommand CMD_OK
$viewMap['default'][0];  // DefaultCommand CMD_DEFAULT
```

Аналогично выполняется поиск по такой же иерархии элементов массива при определении нужного представления.

Ниже приведено определение класса контроллера приложения.

```
namespace woo\controller;
//...
class AppController {
   private static $base_cmd = null;
   private static $default_cmd = null;
   private $controllerMap;
   private $invoked = array();

   function __construct(ControllerMap $map ) {
      $this->controllerMap = $map;
      if (  is_null(self::$base_cmd)  ) {
         self::$base_cmd    = new \ReflectionClass( "\woo\command\Command" );
```

```
            self::$default_cmd = new \woo\command\DefaultCommand();
        }
    }

    function reset() {
        $this->invoked = array();
    }

    function getView(Request $req ) {
        $view = $this->getResource( $req, "View" );
        return $view;
    }

    private function getForward(Request $req ) {
        $forward = $this->getResource( $req, "Forward" );
        if ( $forward ) {
            $req->setProperty( 'cmd', $forward );
        }
        return $forward;
    }

    private function getResource(Request $req, $res ) {
    // Определим предыдущую команду и ее код состояния
        $cmd_str = $req->getProperty( 'cmd' );
        $previous = $req->getLastCommand();
         $status = $previous->getStatus();

         if ( ! isset( $status ) || ! is_int( $status ) ) { $status = 0; }
         $acquire = "get$res";
         $resource = $this->controllerMap->$acquire( $cmd_str, $status );

    // Определим альтернативный ресурс для команды и кода состояния 0
         if ( is_null( $resource ) ) {
            $resource = $this->controllerMap->$acquire( $cmd_str, 0 );
         }

    // Либо для команды 'default' и текущего кода состояния
         if ( is_null( $resource ) ) {
            $resource = $this->controllerMap->$acquire( 'default', $status );
         }

    // Если ничего не найдено, определим ресурс для команды 'default',
    // и кода состояния 0
        if ( is_null( $resource ) ) {
            $resource = $this->controllerMap->$acquire( 'default', 0 );
        }
        return $resource;
    }

    function getCommand(Request $req ) {
        $previous = $req->getLastCommand();

        if ( ! $previous ) {
        // Это первая команда текущего запроса
            $cmd = $req->getProperty('cmd');
            if ( is_null( $cmd ) ) {
```

```
            // Параметр 'cmd' не определен, используем 'default'
               $req->setProperty('cmd', 'default' );
               return self::$default_cmd;
            }
         } else {
            // Команда уже запущена в текущем запросе
               $cmd = $this->getForward( $req );
               if ( is_null( $cmd ) ) { return null; }
         }
         // Здесь в переменной $cmd находится имя команды
         // Преобразуем его в объект типа Command
         $cmd_obj = $this->resolveCommand( $cmd );
         if (is_null( $cmd_obj ) ) {
            throw new \woo\base\AppException(
                  "Команда '$cmd' не найдена" );
         }
         $cmd_class = get_class( $cmd_obj );
         if ( isset($this->invoked[$cmd_class]) ) {
            throw new \woo\base\AppException( "Циклический вызов" );
         }
         $this->invoked[$cmd_class]=1;
         // Возвращаем объект типа Command
         return $cmd_obj;
      }

      function resolveCommand( $cmd ) {
         $classroot = $this->controllerMap->getClassroot( $cmd );
         $filepath = "woo/command/$classroot.php";
         $classname = "\woo\command\$classroot";
         if ( file_exists( $filepath ) ) {
            require_once( "$filepath" );
            if ( class_exists( $classname) ) {
               $cmd_class = new ReflectionClass($classname);
                  if ( $cmd_class->isSubClassOf( self::$base_cmd ) ) {
                     return $cmd_class->newInstance();
                  }
            }
         }
         return null;
      }
}
```

В методе `getResource()` реализован поиск переадресуемой команды и ее представления. Он вызывается из методов `getView()` и `getForward()` соответственно. Обратите внимание на то, как происходит поиск — от самого специфичного сочетания строки команды и кода состояния до самого общего.

Метод `getCommand()` возвращает все сконфигурированные команды, находящиеся в цепочке переадресации. Он работает следующим образом. При получении первоначального запроса устанавливается параметр запроса `cmd`, и в текущем запросе нет информации о запуске предыдущей команды. Эта информация сохраняется в объекте `Request`. Если параметр запроса `cmd` не был установлен, то в методе используется стандартное значение и возвращается стандартный класс типа `Command`. Строковая переменная `$cmd` передается методу `resolveCommand()`, который использует ее для получения объекта типа `Command`.

Когда метод `getCommand()` вызывается во второй раз в текущем запросе, объект `Request` содержит ссылку на объект типа `Command`, который запускался ранее. Затем в методе `getCommand()` проверяется, установлена ли какая-либо переадресация, для сочетания текущего объекта `Command` и кода его состояния (с помощью вызова метода `getForward()`). Если метод `getForward()` находит соответствие, то он возвращает строку, которая может быть преобразована в объект типа `Command` и возвращена контроллеру приложения.

Нужно отметить еще кое-что в отношении метода `getCommand()`: мы выполняем очень важную проверку, которая позволит предотвратить циклический вызов команд друг другом. Для этого поддерживается массив `$invoked`, индексированный именами классов типа `Command`. Если элемент уже существует к тому времени, когда мы собрались его добавлять, то мы знаем, что эта команда вызывалась ранее. Тем самым мы рискуем попасть в бесконечный цикл, чего мы на самом деле совсем не хотим. Поэтому, если такое случится, генерируется исключительная ситуация.

Стратегии, которые используются в контроллере приложения для получения представлений и команд, могут существенно меняться; но главное то, что они скрыты от более широкой системы. На рис. 12.7 показан высокоуровневый процесс, посредством которого класс Front Controller использует контроллер приложения для получения сначала объекта типа `Command`, а затем — представления.

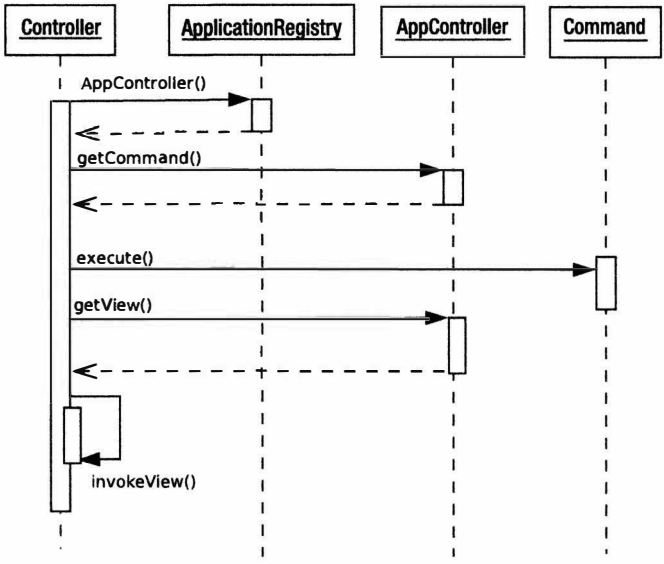


Рис. 12.7. Использование контроллера приложения для получения команд и представлений

Класс `Command`

Наверное, вы заметили, что в классе `AppController` используются предыдущие команды, сохраненные в объекте `Request`. Это делается с помощью базового класса `Command`.

```
namespace woo\command;
//....
abstract class Command {
   private static $STATUS_STRINGS = array (
```

```
        'CMD_DEFAULT'           => 0,
        'CMD_OK'                => 1,
        'CMD_ERROR'             => 2,
        'CMD_INSUFFICIENT_DATA' => 3
              );
   private $status = 0;

   final function __construct() { }

   function execute( \woo\controller\Request $request ) {
      $this->status = $this->doExecute( $request );
      $request->setCommand( $this );
   }

   function getStatus() {
      return $this->status;
   }

   static function statuses( $str='CMD_DEFAULT' ) {
      if ( isset ( self::$STATUS_STRINGS[$str] ) ) {
         // Преобразуем строку в код состояния
         return self::$STATUS_STRINGS[$str];
      }
      throw new \woo\base\Exception("Неизвестный код состояния: $str");
   }
   abstract function doExecute( \woo\controller\Request $request );
}
```

В классе `Command` определяется массив строк состояния (ради краткости и понятности код существенно сокращен). В нем предусмотрены метод `statuses()`, предназначенный для преобразования строки состояния (`"CMD_OK"`) в ее эквивалентное число, и метод `getStatus()` — для возврата текущего кода состояния объекта `Command`. Поскольку возврат из метода `statuses()` непредвиденного значения может вызвать трудно локализуемую ошибку в программе, при получении неизвестного кода состояния генерируется исключение. В методе `execute()` возвращенное из абстрактного метода `doExecute()` значение сохраняется в виде кода состояния, а также сохраняется ссылка на себя в объекте `Request`.

Пример конкретной команды

Вот как может выглядеть простая команда `AddVenue`.

```
namespace woo\command;
//....
class AddVenue extends Command {

   function doExecute( \woo\controller\Request $request ) {
      $name = $request->getProperty("venue_name");
      if ( is_null( $name ) ){
         $request->addFeedback( "Имя не задано" );
         return self::statuses('CMD_INSUFFICIENT_DATA');
      } else {
         $venue_obj = new \woo\domain\Venue( null, $name );
         $request->setObject( 'venue', $venue_obj );
         $request->addFeedback( "'$name' добавлено в ({$venue_obj->getId()})" );
         return self::statuses('CMD_OK');
```

```
        }
    }
}
```

Часть этого кода станет более понятной в следующей главе. А пока что ниже приведена заглушка объекта `Venue`, которая позволит заработать нашей команде.

```
namespace woo\domain;

    class Venue {
        private $id;
        private $name;

        function __construct( $id, $name ) {
            $this->name = $name;
            $this->id = $id;
        }

        function getName() {
            return $this->name;
        }

        function getId() {
            return $this->id;
        }
}
```

Возвратимся к команде. Главное, что нужно отметить, — метод `doExecute()` возвращает код состояния, который сохраняется в свойстве абстрактного базового класса. Решение о том, как реагировать на факт, что объект был вызван и вернул это состояние, полностью управляется файлом конфигурации. Поэтому, согласно приведенному выше фрагменту XML-файла конфигурации, если возвращается строка состояния `"CMD_OK"`, то интерпретатор механизма переадресации создает экземпляр класса `AddSpace`. Эта цепочка событий запускается таким образом, только если в запросе содержится строка `"cmd=AddVenue"`. Если в запросе содержится `"cmd=QuickAddVenue"`, то переадресации не будет, и отобразится представление `quickaddvenue`.

Обратите внимание на то, что в этом примере вообще не включен код сохранения объекта `Venue` в базе данных. Мы рассмотрим это в следующей главе.

Результаты

Реализовать полноценный пример шаблона Application Controller довольно трудно, потому что нужно проделать огромную работу по получению и применению метаданных, которые описывают связи между командой и запросом, командой и командой, командой и представлением.

Поэтому я стараюсь реализовать нечто подобное, когда совершенно очевидно, что это необходимо для приложения. Мне словно кто-то нашептывает это, когда я добавляю 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.

Конечно, в контроллере приложения могут использоваться все возможные механизмы для построения связей между командами и представлениями, а не только тот подход, который я здесь использовал. Даже если вы начинаете с фиксированной связи между строкой запроса, именем команды и представления во всех случаях, то

все равно будете пользоваться преимуществами построения контроллера приложения, в котором все это будет инкапсулировано. Это даст вам значительную степень гибкости, когда необходимо будет провести рефакторинг, чтобы обеспечить более высокий уровень сложности.

Шаблон Page Controller

Хотя я очень люблю шаблон Front Controller, использовать его — не всегда правильно. Вложения в такой проект окажутся плюсом для крупной системы и минусом — для простых проектов, которые нужно осуществить быстро. И здесь вам пригодится шаблон Page Controller, с которым вы, вероятно, уже знакомы, поскольку он используется довольно часто. Тем не менее некоторые вопросы нам стоит обсудить.

Проблема

И снова проблема состоит в том, что необходимо управлять связью между запросом, логикой приложения и представлением данных. Для корпоративных проектов это практически постоянная проблема. Меняются только налагаемые ограничивающие условия.

Если у вас относительно простой проект, для которого развертывание большой многофункциональной структуры приведет только к срыву сроков сдачи, не дав при этом практически никаких преимуществ, то шаблон Page Controller станет хорошим инструментом управления запросами и представлениями.

Предположим, вам нужно вывести страницу, на которой отображается список всех заведений в системе Woo. Даже если у нас написан код для извлечения их списка из базы данных, без использования шаблона Page Controller перед нами встанет трудная задача достижения такого, казалось бы, простого результата.

Итак, список заведений должен отображаться на странице представления; перед этим для его получения нужно выполнить запрос. Поскольку допускается возможность возникновения ошибки, в результате выполнения запроса не всегда будет отображаться новое представление, как можно было бы ожидать в сложной системе. Поэтому самый простой и эффективный метод в нашем случае — связать представление с контроллером и, как правило, на одной и той же странице.

Реализация

Хотя при применении шаблона Page Controller могут возникать трудности, сам шаблон чрезвычайно прост. Контроллер связан с представлением или набором представлений. В простейшем случае это означает, что контроллер находится в самом представлении, хотя его можно отделить, особенно если представление тесно связано с другими представлениями (т.е. в случае, когда в различных ситуациях нужно перенаправлять к разным страницам).

Вот простейший пример реализации шаблона Page Controller.

```
<?php
   require_once("woo/domain/Venue.php");
   try {
      $venues = \woo\domain\Venue::findAll();
   } catch ( Exception $e ) {
      include( 'error.php' );
      exit(0);
   }

// Стандартная страница расположена ниже
```

```
?>

<html>
   <head>
      <title>Заведения</title>
   </head>

   <body>
      <h1>Заведения</h1>
<?php foreach( $venues as $venue ) { ?>
<?php    print $venue->getName(); ?> <br />
<?php } ?>
   </body>
</html>
```

Этот HTML-документ состоит из двух элементов. Элемент-представление управляет отображением, а элемент-контроллер управляет запросом и вызывает логику приложения. Хотя представление и контроллер находятся на одной и той же странице, они строго разделены.

К этому примеру можно мало что добавить (кроме работы с базой данных, которая выполняется "за кулисами", о чем можно подробнее прочитать в разделе "Уровень хранения данных" главы 13). Блок PHP-кода вверху страницы пытается получить список всех объектов типа `Venue`, который он сохраняет в глобальной переменной `$venues`.

Если происходит ошибка, эта страница делегирует полномочия по ее обработке странице `error.php` с помощью функции `include()`, за вызовом которой следует вызов функции `exit()`, чтобы прекратить любую дальнейшую обработку текущей страницы. Я предпочитаю этот механизм механизму переадресации протокола HTTP, который является более сложным и при использовании которого теряются все текущие сеансовые переменные окружения, которые вы сохранили в оперативной памяти. Если вызов `include()` не используется, то отображается заданное внизу страницы представление, описанное HTML-кодом (рис. 12.8).

Рис. 12.8. Шаблон Page Controller, внедренный в страницы представлений

Как видите, мы реализовали шаблон Page Controller в виде тестового примера довольно быстро, но при создании больших систем, вероятно, потребуется приложить гораздо больше усилий, чем в данном случае.

Ранее код шаблона Page Controller был неявно отделен от представления. Сейчас я сделаю это разделение явным, начав с определения элементарного базового класса `PageController`.

```
namespace woo\controller;
//...
abstract class PageController {

   abstract function process();
```

```
    function forward( $resource ) {
        include( $resource );
        exit( 0 );
    }

    function getRequest() {
        return \woo\base\ApplicationRegistry::getRequest();
    }
}
```

В этом классе используются средства, которые мы уже рассматривали, в частности классы `Request` и `ApplicationRegistry`. Основные задачи класса `PageController` — обеспечить доступ к объекту `Request` и управлять включением представлений. Этот список задач в реальном проекте будет быстро расти, по мере того как у дочерних классов будет возникать необходимость в общей функциональности.

Дочерний класс может находиться внутри представления и поэтому отображать его по умолчанию, как и раньше, или находиться отдельно от представления. Мне кажется, второй подход более подходящий, поэтому я выбрал именно его. Приведем пример реализации класса `PageController`, в котором предпринимается попытка добавить новое заведение к системе.

```
namespace woo\controller;
//...
class AddVenueController extends PageController {

    function process() {
        try {
            $request = $this->getRequest();
            $name = $request->getProperty( 'venue_name' );
            if ( is_null( $request->getProperty('submitted') ) ) {
                $request->addFeedback("Выберите имя заведения");
                $this->forward( 'add_venue.php' );
            } else if ( is_null( $name ) ) {
                $request->addFeedback("Имя должно быть обязательно задано");
                $this->forward( 'add_venue.php' );
            }
            // Создадим объект, который затем можно добавить
            // в базу данных
            $venue = new \woo\domain\Venue( null, $name );
            $this->forward( "ListVenues.php" );
        } catch ( Exception $e ) {
            $this->forward( 'error.php' );
        }
    }
}

$controller = new AddVenueController();
$controller->process();
```

В классе `AddVenueController` реализован только метод `process()`, который отвечает за проверку введенных пользователем данных. Если пользователь не заполнил форму или заполнил ее неправильно, то включается стандартное представление (`add_venue.php`), которое обеспечивает обратную связь и выводит форму. Если мы успешно добавили новое заведение, то в этом методе вызывается метод `forward()`, который перенаправляет пользователя контроллеру страницы `ListVenues`.

Обратите внимание на формат, который я использовал для представления. Я стараюсь различать файлы представлений от файлов классов, используя для первых в именах символы нижнего регистра, а для вторых — соединение слов и символы верхнего регистра на границах слов.

Приведем пример представления, связанного с классом `AddVenueController`.

```
<?php
   require_once( "woo/base/Registry.php" );
   $request = \woo\base\ApplicationRegistry::getRequest();
?>

<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 генерации нового запроса. Причем этот запрос делается к классу `PageController`, а не снова к представлению. Помните: именно класс `PageController` отвечает за обработку запросов.

На рис. 12.9 показана схема более сложной версии реализации шаблона Page Controller.

Результаты

У этого подхода есть огромное преимущество: он понятен любому, у кого есть хотя бы минимальный опыт создания веб-приложений. Мы делаем запрос на вывод перечня заведений и именно это получаем. Даже ошибка находится в рамках ожидаемого, поскольку с проблемами типа `"Server error"` и `"Page not found"` приходится сталкиваться постоянно.

Но ситуация становится немного сложнее, если вы отделяете представление от класса Page Controller, однако тесная взаимно однозначная связь между участниками достаточно очевидна.

Сфера, которая довольно сложна для понимания, — это включение представлений. Контроллер страницы (`PageController`) по окончании работы включает свое

представление. Но в некоторых ситуациях он может использовать тот же код включения, чтобы включить другой контроллер страницы. Поэтому, например, когда AddVenue успешно добавляет заведение, то больше не нужно отображать форму для ввода данных. Поэтому полномочия делегируются другому контроллеру страницы с именем ListVenues. Вы должны ясно представлять себе, когда делегируете полномочия представлению, а когда — другому контроллеру страницы. Это обязанность контроллера страницы — гарантировать, что его представления будут иметь необходимые для работы данные.

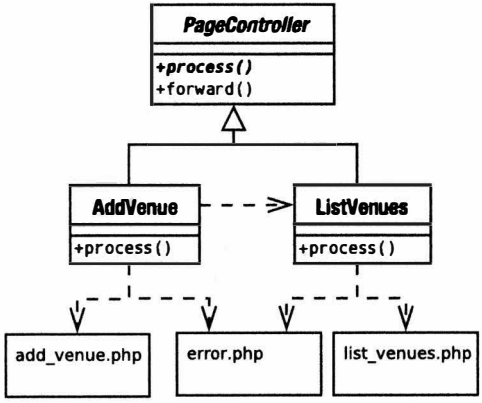


Рис. 12.9. Иерархия класса PageController и его включаемые взаимосвязи

Хотя класс контроллера страницы (PageController) может делегировать полномочия объектам типа Command, преимущество этого не так заметно, как в случае шаблона Front Controller. Классы Front Controller должны сначала определить, какова цель запроса, а классы Page Controller уже это знают. Небольшая проверка параметров запроса и вызовы логики приложения, которые вы бы поместили в класс Command, так же легко помещаются в класс контроллера страницы. И вы сможете извлечь преимущество из того факта, что вам не нужен механизм выбора объектов типа Command.

Дублирование может представлять проблему, но использование общего суперкласса позволит во многом ее решить. Вы также уменьшите время инициализации приложения, поскольку сможете избежать загрузки данных, которые вам не понадобятся в текущем контексте. Конечно, вы сможете это сделать также с помощью шаблона Front Controller, но исследование того, что нужно, а что нет, будет намного сложнее.

Реальный недостаток этого шаблона проявляется в ситуациях, когда пути через представления оказываются сложными — особенно когда одно и то же представление используется разными способами в разное время (хороший пример — экраны для добавления и редактирования данных). Вы вдруг можете обнаружить, что запутались в условных операторах и проверке состояния и что стало трудно получить общее представление о системе.

Но невозможно начать с Page Controller и двигаться к шаблону Front Controller. Это особенно верно, если вы используете суперкласс PageController.

Как правило, если, по моим оценкам, мне нужно меньше недели для завершения разработки системы и в будущем ее не придется расширять, я выбираю шаблон Page Controller и пользуюсь преимуществом быстрого выполнения проекта. Если

же я создаю крупный проект, который имеет сложную логику представлений и со временем будет расширяться, то неизменно останавливаю свой выбор на шаблоне Front Controller.

Шаблоны Template View и View Helper

Шаблон Template View во многом напоминает подход, принятый по умолчанию в PHP (имеется в виду, что на одной странице мы можем поместить как HTML-код разметки, отображающий данные, так и PHP-код веб-приложения). Как я уже говорил, это одновременно и хорошо, и плохо. Простота, с которой все это можно соединить, вызывает искушение объединить логику приложения и отображения в одном месте. Но последствия такого подхода 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 величайшим недоверием!

Для решения этой проблемы в шаблоне View Helper (согласно книге Дипака Алура и других авторов) введен вспомогательный класс, который может быть специфичным для конкретного представления или использоваться в нескольких представлениях. В этот класс помещается весь программный код, требуемый для решения конкретной задачи.

Проблема

Теперь все реже SQL-запросы и другую логику приложения встраивают непосредственно в страницы отображения, но иногда такое случается. Об этой "напасти" я писал очень подробно в предыдущих главах, поэтому сейчас буду краток.

С веб-страницами, содержащими слишком много кода, трудно работать дизайнерам, поскольку компоненты презентации становятся опутанными циклами и условными операторами.

Логика приложения, внедренная в уровень представления, заставляет вас придерживаться этого интерфейса. Вы не можете легко переключиться на новое представление без адаптации и большого количества написанного кода.

Приложения, в которых представление отделено от логики приложения, очень легко тестировать. Все дело в том, что тесты применяются к функциональности уровня логики приложения и никак не связаны с перегруженным и отвлекающим внимание уровнем представления. Кроме того, в системах, содержащих на уровне представления логику приложения, часто возникают уязвимости в системе безопасности. В таких приложениях код запросов к базам данных и код обработки введенных пользователем данных часто разбросаны по разным таблицам, формам и спискам, что резко затрудняет анализ и выявление потенциальных угроз безопасности.

Поскольку многие операции повторяются от одного представления к другому, системы, у которых код приложения 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 ошибки и проблемы сопровождения системы.

Чтобы предотвратить описанные выше проблемы, управление работой приложения нужно перенести в другое место, а представлениям следует разрешить управлять только презентацией. Обычно такого можно достичь, сделав представления пассивными получателями данных. Но если представлению нужно сделать запрос в систему, то имеет смысл предоставить объекту View Helper возможность выполнить всю необходимую работу от имени представления.

Реализация

Как только вы создали общий каркас приложения, программирование уровня представления не стало слишком сложной задачей. Конечно, остается огромная проблема, связанная с проектированием информационной архитектуры, но это выходит за рамки данной книги.

Шаблон Template View был так назван Мартином Фаулером. Это ключевой шаблон, используемый большинством программистов, разрабатывающих корпоративные приложения. В некоторых языках программирования его реализация может включать создание системы шаблонов, которая преобразует теги в значения, установленные в системе. В PHP тоже есть такая возможность. Мы можем использовать интерпретатор шаблонов наподобие замечательного Smarty. Но я предпочитаю использовать встроенные средства PHP, причем использовать осторожно.

Чтобы представление смогло что-то отобразить, у него должна быть возможность получать данные. Я предпочитаю определять вспомогательный класс View Helper, который может использоваться в представлениях. С его помощью они могут получать доступ к объекту `Request` и через него — к любым другим объектам, которые им нужны для выполнения работы.

Вот пример реализации простого класса View Helper.

```
namespace woo\view;
//...
class ViewHelper {

   static function getRequest() {
      return \woo\base\ApplicationRegistry::getRequest();
   }
}
```

Все, что этот класс сейчас делает, — это предоставляет доступ к объекту `Request`. По мере расширения приложения вы можете расширять функциональные возможности этого класса. И если вы обнаружите, что к представлению требуется добавить больше, чем пару строк программного кода, то поместите его в View Helper. В более крупном приложении вы можете создать несколько объектов View Helper в иерархии наследования, чтобы обеспечить различные части системы разными средствами.

Вот пример простого представления, в котором используются и View Helper, и объект `Request`.

```
<?php
   require_once( "woo/view/ViewHelper.php" );
   $request = \woo\view\ViewHelper::getRequest(); // Получаем ссылку на контроллер
   $venue = $request->getObject('venue'); // Получаем ссылку на команду
?>

<html>
   <head>
      <title>Добавьте место к заведению '<?php echo $venue->getName() ?>' </title>
   </head>

   <body>
   <hl> Добавьте место к заведению '<?php print $venue->getName() ?>'</hl>
      <table>
        <tr>
          <td>
<?php print $request->getFeedbackString("</td></tr><tr><td>"); ?>
```

```
        </td>
      </tr>
     </table>

  <form method="post">
    <input type="text"
      value="<?php echo $request->getProperty( 'space_name' ) ?>"
       name="space_name"/>

    <input type="hidden" name="venue_id"
      value="<?php echo $venue->getId() ?>" />

    <input type="submit" value="submit" />
  </form>
  </body>
</html>
```

В этом представлении (файл `add_space.php`) запрашивается объект `Request` от класса `ViewHelper`, а затем используются его методы для отображения динамических данных на HTML-странице. В частности, метод `getFeedback()` возвращает любые сообщения, выведенные командами, а метод `getObject()` позволяет получить любые объекты, сохраненные в кеш-памяти для уровня представления. Метод `getProperty()` используется для доступа к параметрам, установленным в HTTP-запросе. Если запустить это представление само по себе, объекты `Venue` и `Request` могут быть недоступны. Поэтому вернитесь в класс `ApplicationRegistry` и проверьте, установлен ли в нем объект `Request`, и в класс команды `AddVenue`, чтобы убедиться, что объект `Venue` сохранен в объекте `Request`.

Очевидно, что в нашем примере не удастся полностью удалить код из представления, но в нем строго ограничены количество и тип кода, который необходимо написать. Страница содержит простые операторы `print` и несколько вызовов методов. Веб-дизайнер сможет разобраться в этом коде без малейших усилий.

Немного большую проблему представляют условные операторы `if` и циклы. Их трудно делегировать в `ViewHelper`, потому что они обычно связаны с форматированным выводом. Я обычно держу и простые условные операторы, и циклы (которые, как правило, используются при построении таблиц, в которых отображаются строки данных) внутри шаблона Template View. Но чтобы они были по возможности простыми, я, где это возможно, делегирую операторы проверки.

Результаты

Но в том, как данные передаются уровню представления, есть повод для беспокойства. Дело в том, что у представления на самом деле нет фиксированного интерфейса, который предоставляет его окружение. Я рассматриваю каждое представление как вступающее в контакт с системой в целом. Это представление фактически говорит приложению: "Если я вызвано, значит, у меня есть право получать доступ к объекту Этому, Тому и Другому". И здесь все зависит от конкретного приложения — разрешать такой доступ или нет.

Как ни странно, я всегда находил, что подобная стратегия работает для меня идеально. Но вы можете сделать представления более строгими, добавив во вспомогательные классы, которые используются в представлениях, специальные проверки. Если вы зашли так далеко, то можете подумать о полной безопасности и создать во вспомогательных классах специальные методы доступа. Это позволит избавиться

от использования метода `Request::getObject()`, который, по сути, является обычной оболочкой над ассоциативным массивом.

Мне нравится строгая типизация там, где я могу ее получить, но построение параллельной системы на основе представлений и вспомогательных классов `View Helper` — это крайне утомительно. Я стараюсь регистрировать объекты динамически для уровня представления с помощью объекта `Request`, `SessionRegistry` или `ApplicationRegistry`.

Как правило, шаблоны являются пассивными элементами, в которых хранятся данные, полученные в результате выполнения последнего запроса. Однако могут быть случаи, когда в представлении нужно сделать вспомогательный запрос. Для обеспечения этой функциональности отлично подходит шаблон View Helper. В результате механизм доступа к данным надежно скрыт от самого представления. И даже View Helper должен делать как можно меньше работы, делегируя полномочия командам или связываясь с уровнем приложения через фасад (шаблон Facade).

На заметку. Шаблон Facade мы рассматривали в главе 10. Дипак Алур и другие авторы рассматривают только одно его применение при программировании корпоративных приложений с использованием шаблона Session Facade (который предназначен для ограничения мелких сетевых транзакций). Мартин Фаулер также описывает шаблон под названием Service Layer, который обеспечивает простую точку доступа к сложным вещам внутри уровня.

Уровень логики приложения

Если уровень управления организует коммуникации с внешним миром и руководит реакцией системы на него, то на уровне логики приложения решается, собственно, задача, *ради* которой это приложение и было создано. Данный уровень должен быть как можно более свободным от "суеты", возникающей в ходе анализа строк запросов, построения HTML-таблиц и составления ответных сообщений. На уровне логики приложения должны решаться только задачи, диктуемые 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 проблемы, которые должна решать система. Как мы увидим, это гибкое проектное решение. Оно также требует серьезного предварительного планирования.

Давайте начнем с самого быстрого способа настройки и запуска системы.

Шаблон Transaction Script

Этот шаблон, взятый из книги Фаулера *Patterns of Enterprise Application Architecture*, описывает способ эволюции многих систем, пущенных на самотек. Он прост, интуитивно понятен и эффективен, хотя все эти качества ухудшаются по мере роста системы. Шаблон Transaction Script обрабатывает запрос внутри, а не делегирует его специализированным объектам. Это типичный способ получить быстрый результат. Данный шаблон трудно отнести к какой-либо категории, поскольку он сочетает элементы из разных уровней, описанных в этой главе. Я решил представить его как часть уровня логики приложения, поскольку основное назначение данного шаблона — реализовать поставленную перед системой цель.

Проблема

Каждый запрос должен как-то обрабатываться. Как мы уже видели, во многих системах предусмотрен уровень, который анализирует и фильтрует входящие данные. Но в идеале этот уровень должен затем вызывать классы, которые предназначены для выполнения запроса. Как мы увидим, на основе этих классов можно построить логику приложения, вероятно, с помощью интерфейса шаблона Facade. Но этот подход требует серьезного и внимательного проектирования. Для некоторых проектов (обычно небольших по объему и срочных по характеру) такие издержки разработки могут быть неприемлемыми. В этом случае вам, возможно, понадобится встроить логику приложения в набор процедурных операций. Каждая операция будет 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 — это скорость, с которой вы получаете результаты. В каждом сценарии для достижения нужного результата обрабатываются входные данные и выполняются операции с базой данных. Помимо организации связанных методов внутри одного класса и сохранения классов Transaction Script на их собственном уровне (как можно более независимом от уровней команд, управления и представления), для этого требуется минимальное предварительное проектирование.

В то время как классы уровня логики приложения обычно четко отделены от уровня представления, они, как правило, в большей степени встроены в уровень данных. Причина в том, что извлечение и сохранение данных — это ключ к задачам, которые обычно выполняют такие классы. Ниже в этой главе мы увидим механизмы разделения объектов логики приложения и уровня базы данных. Но классы Transaction Script обычно знают все о базе данных (хотя в них могут использоваться промежуточные классы для обработки деталей реальных запросов).

Реализация

Давайте вернемся к нашему примеру обработки списка событий. В данном случае в системе поддерживаются три таблицы реляционных баз данных: `venue`, `space` и `event`. У `venue` (заведения) может быть несколько `space` (мест) (например, у театра может быть несколько сцен, у танцевального клуба — несколько помещений и т.д.) Вот как выглядит схема базы данных.

```
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, нашей системе нужны механизмы для добавления и заведений (venue), и событий (event). Каждая из этих операций представляет одну транзакцию. Мы можем дать каждому методу собственный класс (и организовать наши классы в соответствии с шаблоном Command, который был описан в главе 11). Но в данном случае мы поместим эти методы в один класс, хотя он и является частью иерархии наследования. Структура показана на рис. 12.10.

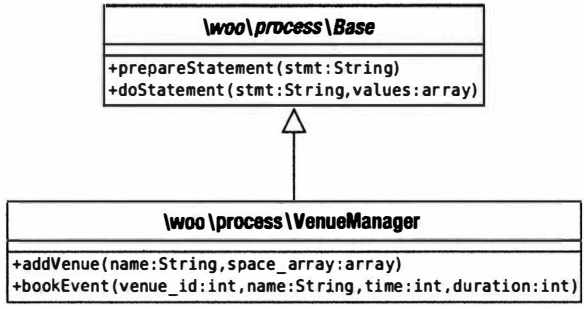


Рис. 12.10. Класс Transaction Script с его суперклассом

Так зачем в этот пример включен абстрактный суперкласс? В сценарии любого размера мы, скорее всего, добавили бы больше конкретных классов к этой иерархии. Поскольку большинство из них будет работать с базой данных, общий суперкласс — это отличное место для помещения основной функциональности для осуществления запросов к базе данных.

Фактически это шаблон сам по себе (Фаулер назвал его Layer Supertype), хотя большинство программистов используют его, не задумываясь об этом. Если классы в уровне имеют общие характеристики, то имеет смысл сгруппировать их в один тип, помещая вспомогательные операции в базовый класс. Этим мы и займемся в оставшейся части главы.

В данном случае базовый класс получает ссылку на объект PDO, которую он сохраняет в статическом свойстве. В нем также предусмотрены методы для кеширования операторов базы данных и осуществления запросов.

```
namespace woo\process;
//...
abstract class Base {
   static $DB;
   static $statements = array();

   function __construct() {
      $dsn = \woo\base\ApplicationRegistry::getDSN();
      if ( is_null( $dsn ) ) {
         throw new \woo\base\AppException( "DSN не определен" );
      }
      self::$DB = new \PDO( $dsn );
      self::$DB->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
   }

   function prepareStatement( $statement ) {
```

```
        if ( isset( self::$statements[$statement] ) ) {
            return self::$statements[$statement];
        }
        $stmt_handle = self::$DB->prepare($statement);
        self::$statements[$statement]=$stmt_handle;
        return $stmt_handle;
    }

    public function doStatement( $statement, array $values ) {
        $sth = $this->prepareStatement( $statement );
        $sth->closeCursor();
        $db_result = $sth->execute( $values );
        return $sth;
    }
}
```

Здесь мы используем класс `ApplicationRegistry`, чтобы получить строку DSN, которая затем передается конструктору PDO.

Метод `prepareStatement()` просто вызывает метод `prepare()` класса `PDO`, который возвращает дескриптор оператора. А его, в конце концов, передают методу `execute()`. При выполнении запроса в этом методе мы кешируем ресурс в статическом массиве `$statements`. В качестве индекса элемента массива используется сам SQL-оператор.

Метод `prepareStatement()` можно вызывать непосредственно из дочерних классов, но более вероятно, что он будет вызван через `doStatement()`. Ему передаются SQL-оператор и смешанный массив значений (строки и целые числа). В этом массиве должны содержаться значения, которые будут переданы базе данных при выполнении оператора. Затем в этом методе SQL-выражение передается методу `prepareStatement()`, который возвращает ресурс оператора, используемый в дальнейшем в методе `PDOStatment::execute()`. Если случается ошибка, то будет сгенерировано исключение. Как вы увидите, вся эта работа скрыта от наших сценариев транзакций. Все, что им нужно сделать, — это построить SQL-выражение и реализовать логику приложения.

Вот начало определения класса `VenueManager`, в котором создаются наши SQL-запросы.

```
namespace woo\process;
//...
class VenueManager extends Base {
    static $add_venue = "INSERT INTO venue
                         ( name )
                          values( ? )";
    static $add_space = "INSERT INTO space
                         ( name, venue )
                         values( ?, ? )";
    static $check_slot = "SELECT id, name
                          FROM event
                          WHERE space = ?
                          AND (start+duration) > ?
                          AND start < ?";
    static $add_event = "INSERT INTO event
                         ( name, space, start, duration )
                         values( ?, ?, ?, ? )";
//...
```

Здесь вы найдете мало чего нового. Эти SQL-выражения будут использоваться в сценарии транзакций. Они созданы в формате метода `prepare()` класса PDO. Знаки вопросов — это места для подстановки значений, которые будут переданы методу `execute()`.

Давайте рассмотрим первый метод, предназначенный для выполнения одной из задач нашего приложения.

```
function addVenue( $name, $space_array ) {
   $venuedata = array();
   $venuedata['venue'] = array( $name );
   $this->doStatement( self::$add_venue, $venuedata['venue']);
   $v_id = self::$DB->lastInsertId();
   $venuedata['spaces'] = array();
   foreach ( $space_array as $space_name ) {
      $values = array( $space_name, $v_id );
      $this->doStatement( self::$add_space, $values);
      $s_id = self::$DB->lastInsertId();
      array_unshift( $values, $s_id );
      $venuedata['spaces'][] = $values;
   }
   return $venuedata;
}
```

Как видите, методу `addVenue()` передаются имя заведения и массив имен мест (space). Он использует их, чтобы заполнить таблицы заведения (venue) и мест (space). Он также создает структуру данных, которая содержит эту информацию, вместе с вновь сгенерированными значениями идентификаторов (ID) для каждой строки.

Благодаря суперклассу в этом методе опущено большое количество скучной работы с базой данных. Мы передаем имя заведения, переданное в качестве параметра, методу `doStatement()`. Не забывайте, что если случится ошибка, то будет сгенерировано исключение. Здесь мы явно не перехватываем никаких исключений, поэтому все исключения, сгенерированные в методах `doStatement()` или (из-за расширения) `prepareStatement()`, также будут сгенерированы и в нашем методе. Мы добились нужного результата, но при этом следует указать в документации, что данный метод может генерировать исключения.

Создав строку данных в таблице `venue`, мы проходим в цикле по массиву `$space_array`, добавляя строку в таблице `space` для каждого места. Обратите внимание на то, что в каждую создаваемую строку в таблице `space` мы включили идентификатор заведения в виде внешнего ключа. Таким образом мы связали место с заведением.

Второй сценарий транзакции точно так же прост.

```
function bookEvent( $space_id, $name, $time, $duration ) {
   $values = array( $space_id, $time, ($time+$duration) );
   $stmt = $this->doStatement( self::$check_slot, $values, false ) ;
   if ( $result = $stmt->fetch() ) {
      throw new \woo\base\AppException(
         "Уже зарегистрировано! Попробуйте еще раз" );
   }
   $this->doStatement( self::$add_event,
            array( $name, $space_id, $time, $duration ) );
}
```

Цель этого сценария — добавить событие к таблице `events`, связанное с местом. Обратите внимание на то, что мы используем SQL-выражение, содержащееся в

свойстве `$check_slot`, чтобы убедиться в том, что предлагаемое событие не будет пересекаться с другим в том же самом месте.

Результаты

Шаблон Transaction Script — это эффективное средство быстрого получения результатов. Это также один из тех шаблонов, которые многие программисты используют годами, не подозревая, что у него есть название. С помощью нескольких хороших вспомогательных методов, таких как те, которые мы добавили к базовому классу, вы можете сосредоточиться на логике приложения, не отвлекаясь на детали работы с базой данных.

Я видел применение шаблона Transaction Script в менее благоприятном контексте. Я писал намного более сложное и более наполненное объектами приложение по сравнению с теми, для которых обычно применяется данный шаблон. Но когда приблизились сроки сдачи проекта, я обнаружил, что поместил очень много логики туда, где должен был быть тонкий фасад шаблона Domain Model (о нем читайте в следующем разделе). И хотя результат был менее изящным, чем ожидалось, я должен был признать, что приложение не пострадало от этого неявного редизайна.

В большинстве случаев следует использовать шаблон Transaction Script для небольших проектов, если вы уверены, что они не перерастут в нечто большее. Этот подход не слишком хорошо масштабируется, потому что дублирование, как правило, возникает, когда сценарии неизбежно пересекаются. Конечно, можно сделать рефакторинг, но вряд ли вам удастся полностью избавиться от дублирования.

В нашем примере мы решили вставить код работы с базой данных в сами классы сценариев транзакций. Но, как вы видели, в этом коде нужно отделить работу с базой данных от логики приложения. Мы можем сделать это разделение полным, изъяв код для работы с базой данных из класса вообще и создав промежуточный класс, роль которого — управлять взаимодействием с базой данных от лица системы.

Шаблон Domain Model

Шаблон Domain Model — это чисто логический движок, который пытались создать, вскормить и защитить многие другие шаблоны, описанные в данной главе. Это отделенное представление сил, работающих в вашем проекте. Это что-то вроде плоскости форм, где естественным образом разворачиваются ваши бизнес-задачи, свободные от таких неприятных материальных проблем, как базы данных и веб-страницы.

Если сказанное кажется вам слишком витиеватым, давайте вернемся к реальности. Шаблон Domain Model — это представление реальных участников вашей системы. Именно в шаблоне Domain Model объект является самим собой в большей степени, чем где-либо еще. Во всех других случаях объекты имеют тенденцию к воплощению дополнительных обязанностей. А в шаблоне Domain Model они, как правило, описывают набор атрибутов, которым добавлены действия. Здесь объект — это *нечто*, которое *что-то* делает.

Проблема

Если вы использовали шаблон Transaction Script, то, наверное, поняли, что дублирование становится проблемой, когда в различных сценариях приходится выполнять одни и те же задачи. Эту проблему можно решить в некоторой степени (вынести повторные операции в отдельный модуль), но со временем все равно все сводится к копированию и вставке кода.

Шаблон Domain Model можно использовать для извлечения и объединения участников и процессов в вашей системе. Вместо того чтобы использовать сценарий для добавления данных о месте к базе данных, а затем связывать с ними данные о событии, можно создать классы `Space` и `Event`. Тогда будет очень просто заказать событие в определенном месте: достаточно вызвать `Space::bookEvent()`. Задача проверки, нет ли конфликта по времени, превращается в вызов `Event::intersects()` и т.д.

Очевидно, для такого простого примера, как Woo, шаблон Transaction Script является более подходящим. Но по мере того, как логика приложения будет усложняться, альтернативный вариант, т.е. использование шаблона Domain Model, будет становиться все более привлекательным. Вам будет проще справляться со сложной логикой и понадобится меньше условных операторов, если вы создадите модель предметной области приложения.

Реализация

Шаблоны Domain Model проектировать достаточно просто. Сложность в основном заключается в том, чтобы проектировать шаблоны, которые должны сохранять "чистоту" модели, т.е. отделять ее от других уровней в приложении.

Отделение участников шаблона Domain Model от уровня представления — это по большей части вопрос обеспечения того, что все должно находиться на своем месте. Отделение участников от уровня данных является намного более проблематичным. Хотя идеальный вариант — это рассматривать шаблон Domain Model только с точки зрения задач, которые он представляет и решает, трудно избежать реальных проблем, связанных с базой данных.

Обычна ситуация, когда классы Domain Model практически непосредственно соответствуют таблицам в реляционной базе данных, что существенно упрощает жизнь. Например, на рис. 12.11 показана диаграмма класса, на которой отображены некоторые из участников системы Woo.

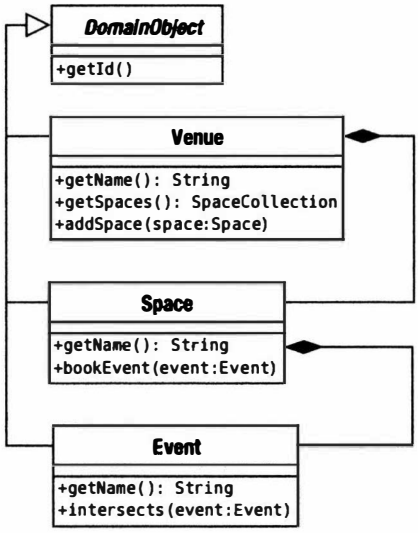


Рис. 12.11. Фрагмент шаблона Domain Model

Объекты на рис. 12.11 соответствуют таблицам, которые были созданы для шаблона Transaction Script. Благодаря этой прямой связи системой становится легче

управлять, но это не всегда возможно, особенно если вы работаете со схемой базы данных, которая была создана до вашего приложения. Такая связь сама может быть источником проблем. Если вы не внимательны, то все ваши усилия будут сосредоточены на работе с базой данных, а не на решении тех проблем, ради которых создавалось приложение.

То, что шаблон Domain Model часто отражает структуру базы данных, не означает, что его классы должны что-то о ней знать. 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 также позволяет сосредоточить ответственность каждого класса на его основных задачах.

Вот пример упрощенного объекта `Venue` вместе с его родительским классом.

```
namespace woo\domain;

abstract class DomainObject {
   private $id;

   function __construct( $id=null ) {
      $this->id = $id;
   }

   function getId( ) {
      return $this->id;
   }

   static function getCollection( $type ) {
      return array(); // Заглушка
   }

   function collection() {
      return self::getCollection( get_class( $this ) );
   }
}

class Venue extends DomainObject {
   private $name;
   private $spaces;

   function __construct( $id=null, $name=null ) {
      $this->name = $name;
      $this->spaces = self::getCollection("\\woo\\domain\\Space");
      parent::__construct( $id );
   }

   function setSpaces(SpaceCollection $spaces ) {
      $this->spaces = $spaces;
   }

   function getSpaces() {
      return $this->spaces;
   }

   function addSpace(Space $space ) {
      $this->spaces->add( $space );
```

```
        $space->setVenue( $this );
    }

    function setName( $name_s ) {
        $this->name = $name_s;
        $this->markDirty();
    }

    function getName( ) {
        return $this->name;
    }
}
```

Есть несколько моментов, которые отделяют этот класс от того, который предназначен для работы без персистентности. Вместо массива мы используем объект типа `SpaceCollection`, чтобы сохранять любые объекты `Space`, которые может содержать `Venue`. (Здесь мы можем утверждать, что массив, обеспечивающий безопасность типов, — это несомненное преимущество, независимо от того, работаете вы с базой данных или нет!) Поскольку этот класс работает со специальным объектом-коллекцией, а не с массивом объектов `Space`, при запуске конструктору нужно создавать экземпляры пустой коллекции. Он делает это, вызывая статический метод для супертипа уровня.

```
$this->spaces = self::getCollection("\\woo\\domain\\Space");
```

К этой коллекции объектов системы я вернусь в следующей главе, а пока что суперкласс просто возвращает пустой массив.

На заметку. В этой и следующей главах я опишу изменения, которые нужно сделать в объектах `Venue` и `Space`. Они являются простыми объектами предметной области и содержат общую функциональность. Если вы параллельно с чтением книги занимаетесь кодированием, вам не составит большого труда реализовать в обоих классах те концепции, которые я опишу. Например, класс `Space` не должен работать с коллекцией объектов типа `Space`, но в нем могут обрабатываться объекты типа `Event` точно таким же образом.

Конструктору передается параметр `$id`, который мы передали суперклассу для хранения. Вряд ли вы будете удивлены, узнав, что параметр `$id` представляет уникальный идентификатор строки в базе данных. Также обратите внимание на то, что мы вызываем метод для суперкласса, который называется `markDirty()` (он будет описан, когда речь пойдет о шаблоне Unit of Work).

Результаты

Сложность реализации шаблона Domain Model зависит от тех бизнес-процессов, которые вам нужно эмулировать. Его преимущество заключается в том, что вы можете сосредоточиться на своей проблеме, проектируя модель и решая вопросы персистентности и представления на других уровнях. Но это теория.

А на практике, я думаю, большинство разработчиков проектируют модели предметной области (Domain Model), все-таки 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 помещения объектов в базу данных и извлечения их из нее.

За отделение модели предметной области от уровня данных нужно платить значительную цену в смысле проектирования и планирования. Можно поместить код базы данных непосредственно в модель (хотя вы, вероятно, захотите спроектировать промежуточный класс для обработки реальных SQL-запросов). Для относитель-

но простых моделей, особенно если каждый класс явно соответствует таблице, этот подход будет очень выигрышным и позволит вам избежать большой дополнительной работы по разработке внешней системы для согласования объектов с базой данных.

Резюме

Я изложил в этой главе огромное количество материала (хотя многое опустил). Пусть вас не пугают большие объемы кода, которые я включил в главу. Шаблоны нужно использовать тогда, когда это уместно, и сочетать, когда это полезно. Используйте шаблоны, описанные в данной главе, если вы считаете, что они удовлетворяют потребности вашего проекта, и не думайте, что вы должны создавать весь каркас до того, как приступить к проекту. С другой стороны, здесь *содержится* достаточно материала, чтобы сформировать основу инфраструктуры или, что так же вероятно, дать представление об архитектуре некоторых готовых каркасов, которые можно выбрать и применить.

И более того! Я оставил вас в неустойчивом положении на краю персистентности, дав всего несколько провокационных намеков о коллекциях и инструментах соответствия (mapper), чтобы подразнить вас. В следующей главе мы рассмотрим некоторые шаблоны, предназначенные для работы с базами данных, которые не зависят от деталей хранения данных.

Глава 13

Шаблоны баз данных

В большинстве веб-приложений любой сложности в той или иной степени приходится иметь дело с сохраняемостью данных, или *персистентностью*. Интернет-магазины должны хранить информацию о своих товарах и клиентах. Игры должны помнить своих игроков и состояние их игры. Сайты социальных сетей должны отслеживать 238 ваших друзей и бессчетное количество любимых групп 80–90-х годов. Каким бы ни было приложение, вполне вероятно, что оно незаметно ведет счет чему-либо. В данной главе мы рассмотрим несколько шаблонов, которые можно применить для решения описанных задач.

В главе будут рассмотрены следующие темы.

- *Интерфейс уровня хранения данных:* шаблоны, которые определяют точки контакта между уровнем хранения данных и остальной системой.
- *Наблюдение за объектами:* отслеживание объектов, избежание 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:* построение 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 разговоров с клиентами. А среди разработчиков самая популярная тема — база данных, размеры которой угрожающе растут. И беспокоит нас не сама база данных; мы верим, что она честно выполняет свою работу, если мы не совсем уж неудачники. Нас волнуют используемые механизмы, которые преобразуют строки и столбцы таблицы базы данных в структуры данных и которые могут стать источниками проблем. В этой главе мы рассмотрим фрагменты программ, которые помогут в решении данных проблем.

Не все, что описано ниже в этой главе, относится к самому уровню хранения данных. Я выделил часть шаблонов, которые помогают решать проблемы персистентности. 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, то в разделе "Шаблон Domain Model" главы 12 я обошел вниманием вопрос сохранения и извлечения объектов `Venue` из базы данных. Теперь же пришло время получить по крайней мере некоторые ответы. Шаблон Data Mapper описан Алуром в книге *Core J2EE Patterns*[1] (как Data Access Object) и Мартином Фаулером в книге *Patterns of Enterprise Application Architecture*[2]. На самом деле Data Access Object — это не точный эквивалент, поскольку он генерирует объекты передачи данных. Однако поскольку подобные объекты в реальной жизни выполняют схожие задачи, то и шаблоны оказываются довольно близкими.

Как и следовало ожидать, Data Mapper — это класс, который отвечает за управление передачей данных от базы данных к объекту.

Проблема

Объекты не организованы как таблицы в реляционной базе данных. Как известно, таблицы 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, и различные структуры данных будут организовывать одни и те же объекты разными способами, по-разному комбинируя их и формируя новые связи 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.

Эту разницу между классами и реляционными базами данных часто называют *реляционным рассогласованием нагрузки* (или просто рассогласованием нагрузки).

Как же сделать такой переход? Один вариант — создать класс (или набор классов), отвечающий за решение только этой проблемы, фактически спрятав базу данных от модели приложения (Domain Model) и сгладив неизбежные "острые края" такого преобразования.

Реализация

Приложив усилия и применив знания в программировании, можно создать один класс `Mapper`, предназначенный для обслуживания нескольких объектов. Однако обычно приходится наблюдать отдельный класс `Mapper`, обслуживающий главный класс модели приложения.

На рис. 13.1 показаны три конкретных класса `Mapper` и абстрактный суперкласс.

На самом деле, поскольку объекты `Space` фактически являются подчиненными для объектов `Venue`, можно выполнить рефакторинг класса `SpaceMapper` в `Venue Mapper`. Однако в данных упражнениях я буду использовать их по отдельности.

---

[1] Дипак Алур, Джон Крупи, Дэн Малкс. *Образцы J2EE. Лучшие решения и стратегии проектирования* (пер. с англ., изд. "Лори", 2013).

[2] Мартин Фаулер. *Шаблоны корпоративных приложений* (пер. с англ., ИД "Вильямс", 2009).

Как видите, в этих классах представлены общие операции сохранения и загрузки данных. В базовом классе сосредоточена общая функциональность, а обязанности по осуществлению операций с конкретными объектами делегируются дочерним классам. Обычно эти операции включают генерацию реального объекта и создание запросов на выполнение операций с базой данных.

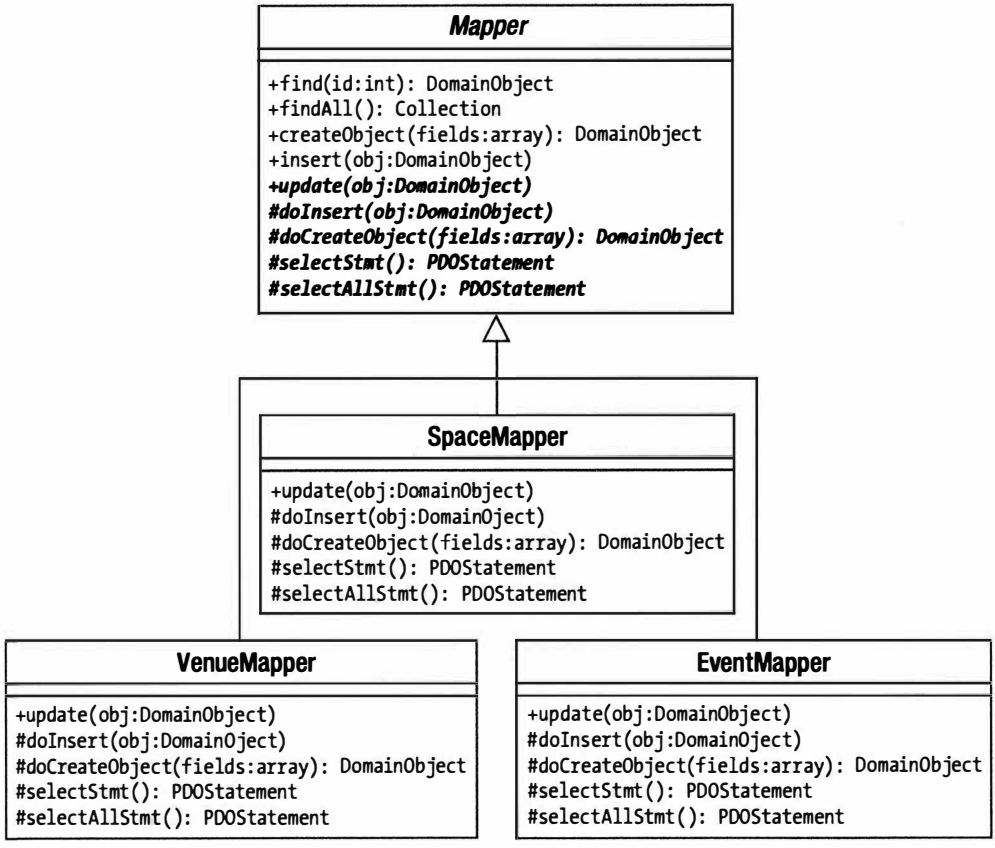


Рис. 13.1. Классы `Mapper`

Наш базовый класс обычно выполняет вспомогательные операции до или после основной операции, поэтому шаблон Template Method используется для явного делегирования (вызовы от конкретных методов, таких как `insert()`, к абстрактным, таким как `doInsert()` и т.д.). Как вы увидите ниже, реализация определяет, какие из методов базового класса сделаны конкретными таким способом.

Вот упрощенная версия базового класса `Mapper`.

```
namespace woo\mapper;
//...

abstract class Mapper {
   protected static $PDO;

   function __construct() {
      if ( ! isset(self::$PDO) ) {
```

```
            $dsn = \woo\base\ApplicationRegistry::getDSN( );
            if ( is_null( $dsn ) ) {
               throw new \woo\base\AppException( "DSN не определен" );
            }
            self::$PDO = new \PDO( $dsn );
            self::$PDO->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
         }
      }

      function find( $id ) {
         $this->selectStmt()->execute( array( $id ) );
         $array = $this->selectStmt()->fetch( );
         $this->selectStmt()->closeCursor( );
         if ( ! is_array( $array ) )     { return null; }
         if ( ! isset( $array['id'] ) ) { return null; }
         $object = $this->createObject( $array );
         return $object;
      }

      function createObject( $array ) {

      $obj = $this->doCreateObject( $array );
      return $obj;
      }

      function insert( \woo\domain\DomainObject $obj ) {

         $this->doInsert( $obj );
      }

      abstract function update( \woo\domain\DomainObject $object );
      protected abstract function doCreateObject( array $array );
      protected abstract function doInsert( \woo\domain\DomainObject $object );
      protected abstract function selectStmt();
}
```

В методе конструктора используется класс `ApplicationRegistry` для получения DSN, который будет использоваться с расширением PDO. Для классов, подобных этому, желаемую информацию можно получить с помощью отдельного синглтона или реестра запросов. Не всегда существует удобный путь от уровня управления к объекту `Mapper`, по которому можно передать данные. Еще один способ создания объекта `Mapper` — передать его самому классу `Registry`. Вместо того чтобы создавать его экземпляр, объекту `Mapper` можно *предоставить* объект PDO в качестве аргумента конструктора.

```
namespace woo\mapper;
//...

abstract class Mapper {
   protected $PDO;

   function __construct( \PDO $pdo ) {
      $this->pdo = $pdo;
   }
}
```

Клиентский код получает новый объект `VenueMapper` от объекта `Registry` с помощью вызова `\woo\base\RequestRegistry::getVenueMapper()`. При этом создается экземпляр объекта `Mapper`, генерирующий также объект PDO. Для последующих запросов этот метод будет возвращать сохраненную в кеше ссылку на объект `Mapper`. Компромисс здесь состоит в том, что мы предоставили объекту `Registry` немного больше информации о системе, чем требуется. Однако это позволило нам не учитывать в объектах `Mapper` сложности, связанные с глобальными данными о конфигурации (даже если в качестве фасада используется реестр). Такой подход я обычно стараюсь использовать в своих приложениях.

Метод `insert()` не делает ничего, кроме делегирования полномочий методу `doInsert()`. В результате я вынес нечто из абстрактного метода `insert()`, поскольку знаю, что эта реализация будет сделана в свое время.

Метод `find()` отвечает за вызов подготовленного оператора (предоставленного реализующим дочерним классом) и получение данных из строки базы данных. Он заканчивает работу, вызывая метод `createObject()`. Конечно, детали преобразования массива в объект будут меняться от случая к случаю, поэтому эти детали будут обрабатываться абстрактным методом `doCreateObject()`. И снова кажется, что метод `createObject()` не делает ничего, кроме делегирования полномочий дочерней реализации. Со временем мы добавим вспомогательные операции, которые сделают использование шаблона Template Method стоящим затраченных усилий.

В дочерних классах будут также реализованы специальные методы нахождения данных в соответствии с заданными критериями (например, нам нужно будет найти объекты `Space`, которые принадлежат объектам `Venue`).

Давайте посмотрим на этот процесс с точки зрения дочернего объекта.

```
namespace woo\mapper;
//...

class VenueMapper extends Mapper {

   function __construct() {
      parent::__construct();
      $this->selectStmt = self::$PDO->prepare(
                          "SELECT * FROM venue WHERE id=?");
      $this->updateStmt = self::$PDO->prepare(
                          "UPDATE venue SET name=?, id=? WHERE id=?");
      $this->insertStmt = self::$PDO->prepare(
                          "INSERT into venue ( name ) values( ? )");
   }

   function getCollection( array $raw ) {
      return new SpaceCollection( $raw, $this );
   }

   protected function doCreateObject( array $array ) {
      $obj = new \woo\domain\Venue( $array['id'] );
      $obj->setName( $array['name'] );
      return $obj;
   }

   protected function doInsert( \woo\domain\DomainObject $object ) {

      $values = array( $object->getName() );
      $this->insertStmt->execute( $values );
```

```
        $id = self::$PDO->lastInsertId();
        $object->setId( $id );
    }

    function update( \woo\domain\DomainObject $object ) {

        $values = array( $object->getName(),
                         $object->getId(), $object->getId() );
        $this->updateStmt->execute( $values );
    }

    function selectStmt() {
        return $this->selectStmt;
    }
```

И снова в этом классе отсутствуют некоторые нужные нам вещи. Тем не менее он делает свое дело. В конструкторе подготавливаются некоторые SQL-операторы для последующего использования. Можно сделать их статическими и совместно использовать во всех экземплярах класса `VenueMapper` или, как было описано ранее, сохранить один объект `Mapper` в `Registry`, тем самым избавившись от повторного создания экземпляров.

В классе `Mapper` реализован метод `find()`, который вызывает `selectStmt()`, чтобы получить подготовленный оператор `SELECT`. Предполагая, что все идет хорошо, `Mapper` вызывает метод `VenueMapper::doCreateObject()`. Здесь мы используем ассоциативный массив для генерации объекта `Venue`.

С точки зрения клиента, этот процесс — сама простота.

```
$mapper = new \woo\mapper\VenueMapper();
$venue = $mapper->find( 12 );
print_r( $venue );
```

Функция `print_r()` позволяет быстро воочию убедиться, что метод `find()` успешно выполнил свою работу. В моей системе (где существует строка в таблице `venue` с идентификатором ID `12`) этот фрагмент кода выводит следующее.

```
woo\domain\Venue Object
(
    [name:woo\domain\Venue:private] => The Eyeball Inn
    [spaces:woo\domain\Venue:private] =>
    [id:woo\domain\DomainObject:private] => 12
)
```

Методы `doInsert()` и `update()` выполняют обратные методу `find()` действия. Каждому из них передается объект `DomainObject`, из которого извлекаются данные, которые должны быть занесены в строку базы данных, и затем вызывается метод `PDOStatement::execute()`. Обратите внимание на то, что в методе `doInsert()` запоминается идентификатор `ID` в предоставленном объекте. Не забывайте, что объекты в PHP передаются по ссылке, поэтому данное изменение отобразится и в клиентском коде по имеющейся в нем ссылке.

Еще нужно отметить, что методы `doInsert()` и `update()` на самом деле не являются строго типизированными. Они примут любой подкласс `DomainObject` без всяких возражений. Вы должны выполнить проверку `instanceof` и выдать исключение `Exception`, если будет передан не тот объект. Это позволит уберечься от неизбежных ошибок.

И снова посмотрим, как выглядят, с точки зрения клиента, операции вставки и обновления данных.

```
$venue = new \woo\domain\Venue();
$venue->setName( "The Likey Lounge-yy" );

// Добавим объект в базу данных
$mapper->insert( $venue );

// Снова найдем объект - просто для проверки, что все работает!
$venue = $mapper->find( $venue->getId() );
print_r( $venue );

// Изменим объект
$venue->setName( "The Bibble Beer Likey Lounge-yy" );

// Вызовем операцию обновления измененных данных
$mapper->update( $venue );

// И снова обратимся к базе данных, чтобы проверить, что все работает
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
```

Обработка нескольких строк

Метод `find()` достаточно прост, потому что ему нужно вернуть только один объект. Но что делать, если вам понадобится извлечь из базы данных много данных? Возможно, ваша первая мысль — вернуть массив объектов. Можно поступить и так, но у этого подхода есть большая проблема.

Если вы возвращаете массив, то для каждого объекта в коллекции нужно сначала создать экземпляр. А если у вас результирующее множество из 1 000 объектов, то задача получается слишком трудоемкой. Альтернативный вариант — просто вернуть массив и позволить вызывающему коду решить задачу создания экземпляров. Это возможно, но нарушает само назначение классов `Mapper`.

Но есть способ сделать так, чтобы и волки были сыты, и овцы целы (т.е. совместить несовместимое). Мы можем использовать встроенный интерфейс `Iterator`.

Интерфейс `Iterator` требует, чтобы в реализующем классе были определены методы для работы со списком. Если вы сделаете это, то класс можно будет использовать в циклах `foreach` точно так же, как массив. Некоторые люди считают, что реализации `Iterator` не нужны в таком языке, как PHP, в котором существует хорошая поддержка массивов. Чепуха! В этой главе я приведу по меньшей мере три причины использовать встроенный интерфейс `Iterator` языка PHP.

В табл. 13.1 перечислены методы, определяемые в интерфейсе `Iterator`.

Таблица 13.1. Методы, определяемые интерфейсом `Iterator`

Имя	Описание
`rewind()`	Перемещает указатель в начало списка
`current()`	Возвращает элемент, находящийся в текущей позиции указателя
`key()`	Возвращает текущий ключ (т.е. значение указателя)
`next()`	Возвращает элемент, находящийся в текущей позиции указателя, и перемещает указатель на следующую позицию
`valid()`	Подтверждает, что существует элемент в текущей позиции указателя

Чтобы реализовать интерфейс `Iterator`, нужно реализовать его методы и следить за тем, где вы находитесь внутри набора данных. А то, как вы получаете данные, упорядочиваете или фильтруете их, скрыто от клиента.

Вот пример реализации интерфейса `Iterator`, в которой используется массив. Однако его конструктору также передается объект типа `Mapper`. Причины этого станут понятными позже.

```
namespace woo\mapper;
//...

abstract class Collection implements \Iterator {
   protected $mapper;
   protected $total = 0;
   protected $raw = array();

   private $result;
   private $pointer = 0;
   private $objects = array();

   function __construct( array $raw=null, Mapper $mapper=null ) {
      if ( ! is_null( $raw ) && ! is_null( $mapper ) ) {
         $this->raw = $raw;
         $this->total = count( $raw );
      }
      $this->mapper = $mapper;
   }

   function add( \woo\domain\DomainObject $object ) {
      $class = $this->targetClass();
      if ( ! ($object instanceof $class ) ) {
         throw new Exception("Это коллекция {$class}");
      }
      $this->notifyAccess();
      $this->objects[$this->total] = $object;
      $this->total++;
   }

   abstract function targetClass();

   protected function notifyAccess() {
      // Специально оставлена пустой!
   }

   private function getRow( $num ) {
      $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];
```

```
        }
    }

    public function rewind() {
        $this->pointer = 0;
    }

    public function current() {
        return $this->getRow( $this->pointer );
    }

    public function key() {
        return $this->pointer;
    }

    public function next() {
        $row = $this->getRow( $this->pointer );
        if ( $row ) { $this->pointer++; }
          return $row;
    }

    public function valid() {
        return ( ! is_null( $this->current() ) );
    }
}
```

Конструктор ожидает, что его будут вызывать без аргументов или с двумя аргументами (данные из базы данных, которые в конечном итоге могут быть преобразованы в объекты, и ссылка на объект `Mapper`).

Предположим, что клиент задал аргумент `$raw` (это будет делать объект `Mapper`). Он сохраняется в свойстве вместе с размером предоставленного набора данных. Если предоставлены данные из базы, то также требуется экземпляр объекта `Mapper`, потому что именно он преобразовывает каждую строку из таблицы в объект.

Если аргументы не были переданы конструктору, то класс начинает свою работу пустым (т.е. без данных). Но обратите внимание на то, что существует метод `add()` для добавления данных к коллекции.

В этом классе поддерживается два массива: `$objects` и `$raw`. Если клиентский код требует конкретный элемент, то в методе `getRow()` сначала выполняется поиск в массиве `$objects`, чтобы узнать, нет ли уже созданного экземпляра. Если есть, то метод возвращает его. В противном случае метод ищет в массиве `$raw` нужные данные. Данные в массиве `$raw` присутствуют, только если объект `Mapper` также присутствует, так что данные для соответствующей строки могут быть переданы методу `Mapper::createObject()`, с которым мы уже встречались. Этот метод возвращает объект типа `DomainObject`, который сохраняется в массиве `$objects` с соответствующим индексом. Вновь созданный объект `DomainObject` возвращается пользователю.

Остальная часть класса — это простая работа со свойством `$pointer` и вызовы метода `getRow()`. Если, конечно, не считать метод `notifyAccess()`, который станет очень важным, когда мы будем иметь дело с шаблоном Lazy Load.

Наверное, вы заметили, что класс `Collection` — абстрактный. Вам нужно предоставить специальные реализации для каждого класса приложения.

```
namespace woo\mapper;
//...
class VenueCollection extends Collection {
```

```
    function targetClass( ) {
    return "\woo\domain\Venue";
    }
}
```

Класс `VenueCollection` просто расширяет `Collection`, в нем реализован метод `targetClass()`. Этот факт, в сочетании с проверкой типа в методе `add()` суперкласса, обеспечивает, что только объекты `Venue` могут быть добавлены к коллекции. Вы также можете проводить дополнительную проверку типов в конструкторе, если хотите обеспечить еще большую безопасность.

Очевидно, этот класс должен работать только с `VenueMapper`. Но в практическом смысле это достаточно строго типизированная коллекция, особенно если дело касается шаблона Domain Model.

Разумеется, существуют параллельные классы для объектов `Event` и `Space`.

Обратите внимание на то, что в классе `VenueCollection` реализуется интерфейс `\woo\domain\VenueCollection`. Здесь применен прием по реализации разделенного интерфейса (Separated Interface), который я вкратце опишу. По сути, он позволяет пакетам, находящимся в иерархии `domain`, определять требования к интерфейсу `Collection` независимо от пакетов, находящихся в иерархии `mapper`. В результате для объектов типа `DomainObject` можно использовать уточнение `\woo\domain\VenueCollection`, а не `\woo\mapper\VenueCollection`. По этой причине в недалеком будущем мы вообще можем удалить реализацию объектов `mapper`. Она может быть заменена классом с совершенно другой реализацией, при этом в пакет `domain` потребуется внести минимальные изменения.

Ниже приведен интерфейс `\woo\domain\VenueCollection` вместе с родственными ему интерфейсами.

```
namespace woo\domain;

interface VenueCollection extends \Iterator {
    function add( DomainObject $venue );
}

interface SpaceCollection extends \Iterator {
    function add( DomainObject $space );
}

interface EventCollection extends \Iterator {
    function add( DomainObject $event );
}
```

Некоторые классы `Collection` показаны на рис. 13.2.

Поскольку шаблону Domain Model нужно создавать экземпляры объектов `Collection` и нам может понадобиться в какой-то момент изменять реализацию (особенно для целей тестирования), мы создаем класс-фабрику на уровне шаблона Domain Model для последовательной генерации объектов `Collection` всех нужных типов.

Вот как мы получаем пустой объект `VenueCollection`.

```
require_once( "woo/domain/HelperFactory.php" );
use woo\domain as dom;

$collection = \woo\domain\HelperFactory::getCollection( dom\Venue::class);
```

Затем можно добавить к нему значения и пройти его по циклу, как будто это массив.

```
$collection->add( new \woo\domain\Venue( null, "Loud and Thumping" ) );
$collection->add( new \woo\domain\Venue( null, "Eeezy" ) );
```

```
$collection->add( new \woo\domain\Venue( null, "Duck and Badger" ) );

foreach( $collection as $venue ) {
   print $venue->getName()."\n";
}
```

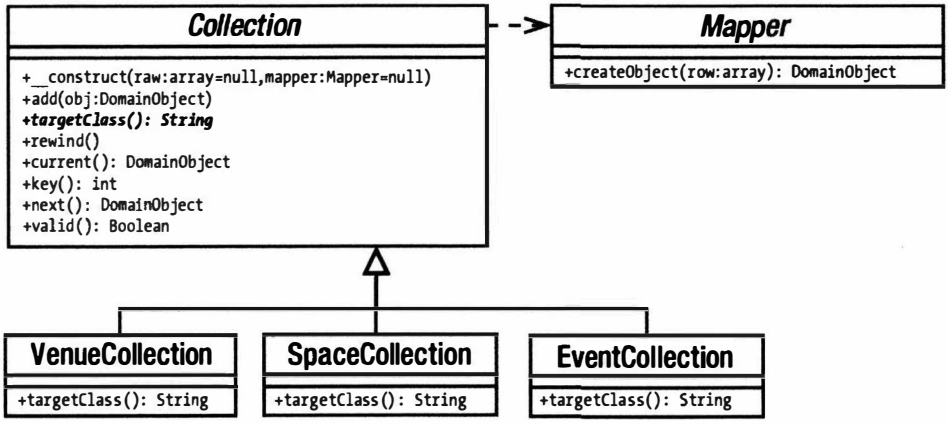


Рис. 13.2. Обработка нескольких строк данных с помощью коллекций

Учитывая реализацию, которую мы здесь создали, с этой коллекцией мало что можно еще сделать, но добавлять `elementAt()`, `deleteAt()`, `count()` и аналогичные методы очень просто. Предоставляю это вам в качестве упражнения. (Выполните его и получите удовольствие!)

Обратите внимание на то, что здесь я использовал синтаксис `::class`, чтобы получить строку, задающую полностью определенное имя класса. Эта возможность появилась только в PHP 5.5. В предыдущих версиях мне нужно было указывать полный путь к классу в пространстве имен и постараться при этом не ошибиться!

Использование генератора вместо интерфейса `Iterator`

Хотя это и не так сложно, реализация интерфейса `Iterator` является довольно рутинной работой. В PHP 5.5 появилась возможность использовать более простой (и часто более эффективный в плане использования памяти) механизм, называемый *генератором* (*generator*). По сути, генератор — это функция, которая может возвращать несколько значений, как правило в цикле. В ней вместо оператора `return` используется оператор `yield`. Как только интерпретатор PHP встречает в коде функции оператор `yield`, он возвращает в вызывающий код итератор типа `Generator`. С этим новым объектом можно выполнять те же самые действия, что и с любым другим итератором. Хитрость заключается в том, что цикл, в котором используется оператор `yield` для возврата нового значения, продолжит свое выполнение только в том случае, если объекту типа `Generator` поступит запрос на следующее значение с помощью вызова метода `next()`. В сущности говоря, весь процесс выполнения кода выглядит так.

- В клиентском коде вызывается функция генератора, содержащая оператор `yield`.
- В функции генератора запрограммирован цикл или какой-то повторяющийся процесс, благодаря которому может возвращаться несколько значений с по-

мощью оператора `yield`. Как только интерпретатор PHP встречает этот оператор, он создает объект типа `Generator` и возвращает его клиентскому коду.

- В этом месте выполнение повторяющегося процесса в функции генератора приостанавливается.
- Полученный клиентским кодом объект типа `Generator` рассматривается как обычный итератор, с которым можно работать в цикле (как правило, `foreach`).
- При выполнении очередной итерации цикла `foreach` объект типа `Generator` будет возвращать следующее значение из функции генератора.

Итак, почему бы нам не воспользоваться этой новой возможностью для нашего базового класса `Collection`? Поскольку функция генератора (точнее — метод) возвращает объект типа `Generator`, в самом классе `Collection` теперь не нужно поддерживать итеративную обработку. Вместо этого я воспользуюсь методом генератора в качестве фабрики, как показано ниже.

```
abstract class Collection {
   protected $mapper;
   protected $total = 0;
   protected $raw = array();
   private $result;
   private $objects = array();

   function __construct( array $raw=null, Mapper $mapper=null ) {
      if ( ! is_null( $raw ) && ! is_null( $mapper ) ) {
         $this->raw = $raw;
         $this->total = count( $raw );
      }
      $this->mapper = $mapper;
   }

   function add( \woo\domain\DomainObject $object ) {
      $class = $this->targetClass();
      if ( ! ($object instanceof $class ) ) {
         throw new Exception("Это коллекция класса {$class}");
      }
      $this->notifyAccess();
      $this->objects[$this->total] = $object;
      $this->total++;
   }

   function getGenerator() {
      for ( $x = 0; $x<$this->total; $x++ ) {
         yield ( $this->getRow($x ) );
      }
   }

   abstract function targetClass();

   protected function notifyAccess() {
   // Специально оставлена пустой!

   }

   private function getRow( $num ) {
```

```
        $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];
        }
    }
}
```

Как видите, такое решение позволило создать более компактный базовый класс. Теперь отпала необходимость в реализации методов `current()`, `reset()` и им подобных. Единственный недостаток состоит в том, что сам класс `Collection` перестал поддерживать напрямую итеративную обработку. Поэтому в клиентском коде нужно вызвать функцию генератора `getGenerator()` и обработать возвращенный оператором `yield` объект типа `Generator` в цикле, как показано ниже.

```
$gen = $collection->getGenerator();

foreach ( $gen as $wrapper ) {
    print_r( $wrapper );
}
```

Поскольку я не собираюсь внедрять эту новую возможность в свою систему, я продолжу рассмотрение данного примера с использованием уже реализованной версии *итератора*. Однако стоит отметить, что генераторы на самом деле предоставляют все средства для создания легковесных итераторов при приложении минимума усилий для их настройки.

Получение объектов класса `Collection`

Суперкласс `DomainObject` — это хорошее место для подходящих методов, которые используются в коллекции.

```
namespace woo\domain;
// ...
// DomainObject

    function collection() {
        return self::getCollection( get_class( $this ) );
    }

    static function getCollection( $type=null ) {
        if ( is_null( $type ) ) {
            return HelperFactory::getCollection( get_called_class() );
        }
        return HelperFactory::getCollection( $type );
    }
```

Этот класс поддерживает два механизма получения объекта `Collection`: статический и на основе создания экземпляра. В обоих случаях методы просто вызывают метод `HelperFactory::getCollection()` с указанием имени класса. В главе 12 мы

видели, как статический метод `getCollection()` используется в примере шаблона Domain Model. Благодаря позднему статическому связыванию (late static bindings) и функции `get_called_class()`, появившейся в PHP 5.3, метод `getCollection()` теперь можно вызывать без аргументов из конкретного объекта `DomainObject` (при условии, что существует связанная с ним коллекция). Дело в том, что функция `get_called_class()` возвращает имя класса, *из которого* был вызван текущий статический метод, а не имя класса, *содержащего* этот метод.

```
$collection = \woo\domain\Venue::getCollection();
```

На рис. 13.3 показана структура класса `HelperFactory`. Обратите внимание на то, что он может использоваться для получения и коллекций, и объектов `Mapper`.

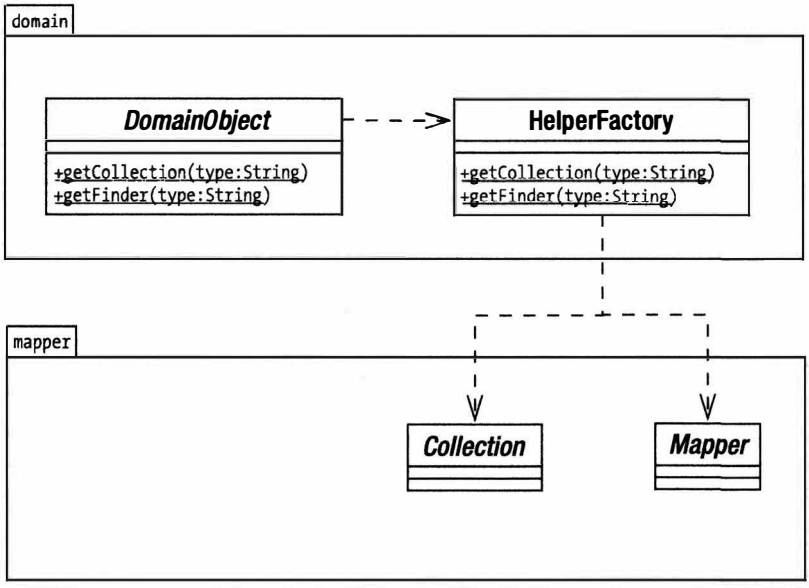


Рис. 13.3. Использование объекта-фабрики в качестве посредника для получения средств персистентности

Для вариации структуры, показанной на рис. 13.3, вам нужно будет создать интерфейсы внутри пакета `domain` для `Mapper` и `Collection`, которые, конечно, должны быть реализованы в их аналогах типа `Mapper`. Таким образом, объекты приложения можно полностью отделить от пакета `mapper` (за исключением тех, которые находятся внутри самого `HelperFactory`, конечно). Этот основной шаблон, который Фаулер называет Separated Interface, будет полезен, если вы знаете, что некоторым пользователям понадобится заменить весь пакет `mapper` его эквивалентом. Если бы нам нужно было реализовать шаблон Separated Interface, метод `getFinder()` возвращал бы экземпляр интерфейса `Finder`, а наши объекты `Mapper` реализовали бы это. Но в большинстве случаев можно оставить это улучшение как повод для рефакторинга в будущем. В данных примерах `getFinder()` просто возвращает объекты `Mapper`.

В свете всего сказанного класс `Venue` можно расширить для работы с персистентностью объектов `Space`. Этот класс предоставляет методы для добавления отдельных объектов `Space` к их `SpaceCollection` или для переключения на совершенно новую реализацию `SpaceCollection`.

```
//Venue
namespace woo\domain;
// ...

function setSpaces(SpaceCollection $spaces ) {
   $this->spaces = $spaces;
}

function getSpaces() {
   if ( is_null ( $this->spaces ) ) {
      $this->spaces = self::getCollection( Space::class );
   }
   return $this->spaces;
}

function addSpace(Space $space ) {
   $this->getSpaces()->add( $space );
   $space->setVenue( $this );
}
```

Операция `setSpaces()` на самом деле предназначена для использования классом `VenueMapper` при построении `Venue`. При этом предполагается, что все объекты `Space` в коллекции ссылаются на текущий `Venue`. Добавить проверку к этому методу было бы довольно просто. Но в данной версии мы опустим это для простоты. Обратите внимание на то, что мы создаем экземпляр свойства `$spaces`, только когда вызывается метод `getSpaces()`. Позже я продемонстрирую, как можно расширить это "ленивое" создание экземпляров, чтобы ограничить запросы к базе данных.

Класс `VenueMapper` должен устанавливать `SpaceCollection` для каждого создаваемого объекта `Venue`.

```
// VenueMapper

namespace woo\mapper;
// ...

protected function doCreateObject( array $array ) {
   $obj = new \woo\domain\Venue( $array['id'] );
   $obj->setName( $array['name'] );
   $space_mapper = new SpaceMapper();
   $space_collection = $space_mapper->findByVenue( $array['id'] );
   $obj->setSpaces( $space_collection );
   return $obj;
}
```

Метод `VenueMapper::doCreateObject()` создает объект `SpaceMapper` и получает из него объект `SpaceCollection`. Как видите, в классе `SpaceMapper` реализован метод `findByVenue()`. Это приводит нас к запросам, которые генерируют несколько объектов. Ради краткости я опустил метод `Mapper::findAll()` из первоначального листинга для `woo\mapper\Mapper`. Здесь он снова присутствует.

```
// Mapper

namespace woo\mapper;
// ...

function findAll( ) {
   $this->selectAllStmt()->execute( array() );
```

```
    return $this->getCollection(
           $this->selectAllStmt()->fetchAll( PDO::FETCH_ASSOC ) );
}
```

Этот метод вызывает дочерний метод `selectAllStmt()`. Как и `selectStmt()`, он должен содержать подготовленный оператор SQL, который выбирает все строки из таблицы. Вот объект `PDOStatement`, созданный в классе `SpaceMapper`.

```
// SpaceMapper::__construct()
   $this->selectAllStmt = self::$PDO->prepare("SELECT * FROM space");

//...
$this->findByVenueStmt = self::$PDO->prepare(
                              "SELECT * FROM space where venue=?");
```

Я включил сюда еще одну инструкцию, `$findByVenueStmt`, которая используется для нахождения объектов `Space`, характерных для отдельного `Venue`.

Метод `findAll()` вызывает еще один новый метод, `getCollection()`, передавая ему найденные данные. Вот определение метода `SpaceMapper::getCollection()`.

```
function getCollection( array $raw ) {
   return new SpaceCollection( $raw, $this );
}
```

В полной версии класса `Mapper` нужно объявить `getCollection()` и `selectAll Stmt()` абстрактными методами, чтобы все объекты типа `Mapper` были способны возвращать коллекцию, содержащую их персистентные объекты приложения. Но чтобы получить объекты `Space`, которые принадлежат `Venue`, нам нужна более ограниченная коллекция. Мы уже видели готовый оператор для получения данных; теперь рассмотрим метод `SpaceMapper::findByVenue()`, который генерирует коллекцию.

```
function findByVenue( $vid ) {
   $this->findByVenueStmt->execute( array( $vid ) );
   return new SpaceCollection(
              $this->findByVenueStmt->fetchAll(), $this );
}
```

Метод `findByVenue()` идентичен `findAll()`, за исключением используемого SQL-оператора. В `VenueMapper` полученная в результате коллекция сохраняется в объекте `Venue` с помощью метода `Venue::setSpaces()`.

Поэтому объекты `Venue` теперь приходят прямо из базы данных, со всеми своими объектами `Space` в аккуратном строго типизированном списке. Ни для одного объекта из этого списка не создается экземпляр до того, как к нему будет сделан запрос.

На рис. 13.4 показаны процесс, посредством которого клиентский класс может получить объект `SpaceCollection`, и то, как класс `SpaceCollection` взаимодействует с `SpaceMapper::createObject()`, чтобы преобразовать свои табличные данные в объект для возвращения клиенту.

Результаты

Недостаток подхода, который мы применили для добавления объектов `Space` к объектам `Venue`, состоит в том, что нужно было делать два обращения к базе данных. Но я считаю, что в большинстве случаев это цена, которую стоит платить. Также заметьте, что работа в методе `Venue::doCreateObject()` по получению правильно заполненного объекта `SpaceCollection` может быть выполнена в методе `Venue::get Spaces()`, чтобы второе подключение к базе данных происходило только по требованию. Вот как может выглядеть такой метод.

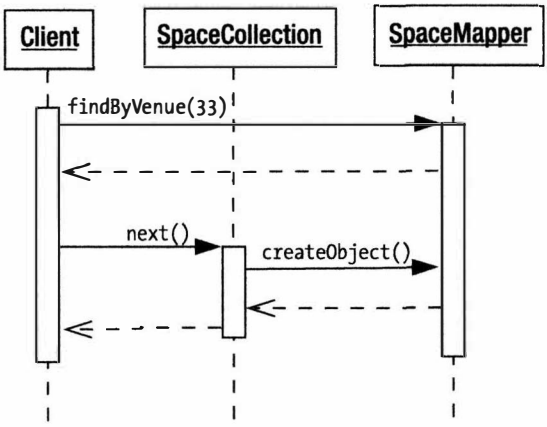


Рис. 13.4. Получение объекта `SpaceCollection` и его использование для получения объекта `Space`

```
// Venue

namespace woo\domain;
// ...

function getSpaces() {
   if ( is_null ( $this->spaces ) ) {
      $finder = self::getFinder( Space::class );
      $this->spaces = $finder->findByVenue( $this->getId() );
   }
   return $this->spaces;
}
```

Но если встанет вопрос эффективности, будет достаточно просто выполнить рефакторинг класса `SpaceMapper` в целом и получить все необходимые данные за один проход с помощью конструкции `JOIN` SQL-запроса.

Конечно, в результате этого код может стать менее переносимым, но за оптимизацию ради эффективности всегда приходится платить!

Разумеется, степень детализации классов `Mapper` будет варьироваться. Если тип объекта сохраняется исключительно другим объектом, то можно иметь объект `Mapper` только для контейнера.

Большое преимущество данного шаблона — это четкое разделение между уровнем приложения и базой данных. Объекты `Mapper` работают "за кулисами" и могут адаптироваться ко всем видам реляционных баз данных.

Наверное, самый большой недостаток данного шаблона — это большое количество утомительной работы по созданию конкретных классов `Mapper`. Но здесь много стандартного кода, который можно генерировать автоматически. Хороший способ генерации общих методов для классов `Mapper` — с помощью рефлексии. Вы можете сделать запрос к объекту приложения, исследовать его методы-получатели и методы-установщики (возможно, учитывая соглашение о наименовании аргументов) и сгенерировать основные классы `Mapper`, готовые для коррекции. Именно так первоначально были созданы все классы `Mapper`, приведенные в данной главе.

Имея дело с классами `Mapper`, нужно помнить об опасности загрузки слишком большого количества объектов одновременно. Но здесь нам поможет реализация

интерфейса `Iterator`. Поскольку сначала объект `Collection` содержит данные только одной строки, второй запрос (для объекта `Space`) делается только тогда, когда обращаются к конкретному объекту `Venue` и преобразуют его из массива в объект. Эту форму "ленивой загрузки" можно усовершенствовать еще больше, как мы увидим далее.

Будьте внимательны с волнообразной загрузкой. Создавая объект `Mapper`, имейте в виду, что использование другого объекта `Mapper` для получения значения свойства нашего объекта может быть верхушкой очень большого айсберга. Этот второй объект `Mapper` сам может использоваться в еще большей степени при создании собственного объекта. И если вы будете не внимательны, то может оказаться, что то, что на поверхности казалось простой операцией поиска, вызывает десятки других аналогичных операций.

Вы должны также знать обо всех директивах, которые используются в приложении базы данных для создания эффективных запросов, и быть готовыми их оптимизировать (если понадобится, то для каждой базы данных). SQL-операторы, которые применимы к нескольким базам данных, — это хорошо; но быстрые приложения — это еще лучше. Хотя введение условных операторов (или классов стратегий) для управления различными версиями одинаковых запросов — это утомительная работа, причем в первом случае результат получается уродливым, не забывайте, что вся эта неприятная оптимизация скрыта от клиентского кода.

Шаблон Identity Map

Помните этот кошмар с ошибками при передаче параметров по значению в PHP 4? Полная путаница, которая происходила, когда вы полагали, что две переменные указывают на один объект, а оказывалось, что на разные, но невероятно похожие? Ну что ж, кошмар возвращается.

Проблема

Давайте рассмотрим тестовый код, созданный для проверки примера Data Mapper.

```
require_once("woo/domain/Venue.php");
$venue = new \woo\domain\Venue();
$mapper = $venue->finder();
$venue->setName( "The Likey Lounge" );

$mapper->insert( $venue );
$venue = $mapper->find( $venue->getId() );
print_r( $venue );

$venue->setName( "The Bibble Beer Likey Lounge" );
$mapper->update( $venue );

$venue = $mapper->find( $venue->getId() );
print_r( $venue );
```

Цель этого кода — продемонстрировать, что объект, который мы добавили к базе данных, мог также быть извлечен с помощью объекта `Mapper` и был бы идентичным, т.е. идентичным во всех отношениях, за исключением того, что это *не один и тот же* объект! Я обошел эту проблему, присвоив новый объект `Venue` и затерев старый. К сожалению, не всегда можно в такой степени контролировать ситуацию. Один и

тот же объект может использоваться в разные моменты времени в рамках *одного* запроса. Если вы изменяете один его вариант и сохраняете в базе данных, можете ли вы быть уверены, что другой вариант этого объекта (вероятно, уже сохраненный в объекте `Collection`) не будет записан поверх ваших изменений?

Дубликаты объектов — это не только риск, но и непроизводительные издержки. Некоторые популярные объекты могут загружаться по три-четыре раза в процессе, и все эти, кроме одного, обращения к базе данных будут совершенно излишними.

К счастью, решить эту проблему достаточно просто.

Реализация

Тождественное отображение (Identity Map) — это просто объект, задача которого — следить за всеми объектами в системе и сделать так, чтобы то, что должно быть одним объектом, вдруг не превратилось в два.

На самом деле шаблон Identity Map не предотвращает эту ситуацию каким-либо активным образом. Его роль состоит в том, чтобы управлять информацией об объектах. Вот простой пример использования шаблона Identity Map.

```
namespace woo\domain;
//...

class ObjectWatcher {
   private $all = array();
   private static $instance=null;

   private function __construct() { }

   static function instance() {
      if ( is_null( self::$instance ) ) {
         self::$instance = new ObjectWatcher();
      }
   return self::$instance;
  }

   function globalKey(DomainObject $obj ) {
      $key = get_class( $obj )."."$obj->getId();
      return $key;
   }

   static function add(DomainObject $obj ) {
      $inst = self::instance();
      $inst->all[$inst->globalKey( $obj )] = $obj;
   }

   static function exists( $classname, $id ) {
      $inst = self::instance();
      $key = "{$classname}.{$id}";
      if ( isset( $inst->all[$key] ) ) {
           return $inst->all[$key];
      }
      return null;
   }
}
```

На рис. 13.5 показано, как объект Identity Map может интегрироваться с другими классами, которые мы уже видели.

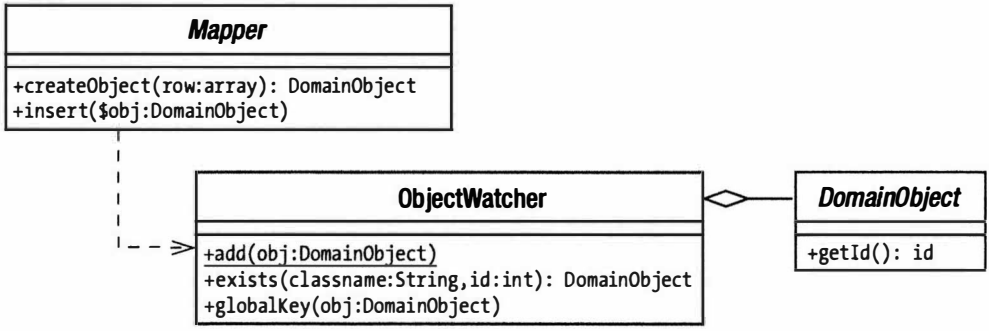


Рис. 13.5. Интеграция объекта Identity Map с другими классами

Главная особенность шаблона Identity Map — это, естественно, определение (идентификация) объектов. Это означает, что нужно пометить каждый объект каким-то способом. Для этого можно использовать ряд различных стратегий. Ключ таблицы базы данных, который уже используют все объекты в системе, не подходит, потому что нет гарантии, что идентификатор ID уникален для всех таблиц.

Вы можете также использовать базу данных, чтобы поддерживать глобальную таблицу ключей. Каждый раз, создавая объект, вы должны будете обратиться к таблице ключей и связать глобальный ключ с объектом, находящимся в его строке. Дополнительная работа здесь достаточно легкая, и выполнить ее просто.

Как вы сможете убедиться, здесь я выбрал более простой подход. Я добавляю к имени класса объекта его табличный идентификатор. Не может быть двух объектов типа `woo\domain\Event` с идентификатором `4`, поэтому выбранный ключ `woo\domain\Event.4` вполне подходит для наших целей.

Метод `globalKey()` обрабатывает детали этого. В этом классе предусмотрен метод `add()` для добавления новых объектов. Каждый объект помечается уникальным ключом в массиве свойства `$all`.

Методу `exists()` передаются имя класса и идентификатор `$id`, а не объект. Мы не хотим создавать экземпляр объекта, чтобы узнать, существует ли он! Наш метод создает ключ на основе этих данных и делает проверку, чтобы узнать, индексирует ли он элемент в свойстве `$all`. Если объект найден, то возвращается ссылка, как и требуется.

Есть только один класс, в котором я работаю с классом `ObjectWatcher` в роли Identity Map. Класс `Mapper` обеспечивает функциональные возможности для генерации объектов, поэтому имеет смысл добавить здесь проверку.

```
// Mapper
namespace woo\mapper;
//...

private function getFromMap( $id ) {
   return \woo\domain\ObjectWatcher::exists(
                              $this->targetClass(), $id );
}

private function addToMap( \woo\domain\DomainObject $obj ) {
   return \woo\domain\ObjectWatcher::add( $obj );
```

```
}

function find( $id ) {
   $old = $this->getFromMap( $id );

   if ( ! is_null( $old ) ) { return $old; }

   // Работаем с db
   return $obj;
}

function createObject( $array ) {
   $old = $this->getFromMap( $array['id']);
   if ( ! is_null( $old ) ) { return $old; }

   // Создаем объект
   $this->addToMap( $obj );
   return $obj;
}

function insert( \woo\domain\DomainObject $obj ) {
   // Обрабатываем вставку. $obj нужно пометить новым идентификатором
   $this->addToMap( $obj );
}
```

В этом классе предусмотрено два удобных метода: `addToMap()` и `getFromMap()`. Это позволяет не запоминать полный синтаксис статического вызова `ObjectWatcher`. И, что еще важнее, они делают вызов к дочерним реализациям (`VenueMapper` и т.д.), чтобы получить имя класса, который в настоящее время ожидает создания экземпляра.

Это достигается путем вызова `targetClass()`, абстрактного метода, который реализуется во всех конкретных классах `Mapper`. Он должен вернуть имя класса, объект которого должен сгенерировать `Mapper`. Вот реализация метода `targetClass()` класса `SpaceMapper`.

```
protected function targetClass() {
   return \woo\domain\Space::class;
}
```

И `find()`, и `createObject()` сначала проверяют, существует ли объект, передавая идентификатор таблицы методу `getFromMap()`. Если объект найден, то он возвращается клиенту, и выполнение метода заканчивается. Но если никаких версий этого объекта не существует, то создание экземпляра объекта продолжается. В методе `createObject()` этот новый объект передается методу `addToMap()`, чтобы предотвратить любые конфликты в будущем.

Так зачем мы проходим эту часть процесса дважды, делая вызовы к `getFromMap()` и в `find()`, и в `createObject()`? Ответ на этот вопрос связан с объектами `Collection`. Когда они генерируют объекты, то делают это, вызывая `createObject()`. Нам нужно убедиться в том, что строка, инкапсулированная объектом `Collection`, не устарела, и обеспечить, чтобы самая последняя версия объекта была возвращена пользователю.

Результаты

Пока вы используете шаблон Identity Map во всех контекстах, где объекты генерируются из базы данных или добавляются в нее, вероятность дублирования объектов в процессе практически равна нулю.

Разумеется, это работает только *внутри* вашего процесса. Различные процессы будут неизбежно обращаться к разным версиям одного и того же объекта одновременно. Важно продумать возможности искажения данных в результате параллельного (одновременного) обращения. Если это серьезная проблема, то реализуйте стратегии блокировки. Можете подумать также о сохранении объектов в совместно используемой памяти или использовании системы кеширования для внешних объектов наподобие `Memcached`. Подробнее о сервере `Memcached` можно узнать на сайте `http://memcached.org/`, а о его поддержке в PHP — на сайте `http://www.php.net/memcache`.

Шаблон Unit of Work

В какой момент вы сохраняете объекты? Пока я не открыл для себя шаблон Unit of Work (описанный Дэвидом Райсом в книге Мартина Фаулера *Patterns of Enterprise Application Architecture*), я посылал приказы на сохранение с уровня представления по завершении команды. Но это проектное решение оказалось довольно накладным.

Шаблон Unit of Work помогает сохранять только те объекты, которые нужно сохранять.

Проблема

Однажды я вывел набор своих SQL-операторов в окно браузера и был шокирован. Я обнаружил, что сохранял одни и те же данные снова и снова в одном и том же запросе. У меня была замечательная система смешанных команд, т.е. одна команда могла запустить несколько других, и каждая выполняла очистку после себя.

Причем я не только сохранял один и тот же объект дважды, я сохранял еще и объекты, которые не нужно было сохранять.

Данная проблема в некоторых отношениях аналогична той, которую решает шаблон Identity Map. Там дело было в ненужной загрузке объектов; а здесь проблема находится на другом конце процесса. И взаимодополняющими являются как проблемы, так и решения.

Реализация

Чтобы определить, какие требуются операции с базой данных, нужно следить за различными событиями, которые происходят с объектами. Вероятно, самое лучшее место, где это можно сделать, — сами объекты.

Нужно также вести список объектов, составленный для каждой операции с базой данных (вставка, обновление, удаление). Сейчас я собираюсь осветить только операции вставки и обновления. Где лучше сохранять список объектов? Поскольку у нас уже есть объект `ObjectWatcher`, можем продолжить разработку.

```
// ObjectWatcher
namespace woo\domain;

//...
private $all    = array();
private $dirty  = array();
private $new    = array();
private $delete = array(); // В нашем примере не используется
private static $instance;
// ...
```

```
static function addDelete( DomainObject $obj ) {
   $self = self::instance();
   $self->delete[$self->globalKey( $obj )] = $obj;
}

static function addDirty(DomainObject $obj ) {
   $inst = self::instance();
   if ( ! in_array( $obj, $inst->new, true ) ) {
      $inst->dirty[$inst->globalKey( $obj )] = $obj;
   }
}

static function addNew(DomainObject $obj ) {
   $inst = self::instance();
   // У нас еще нет идентификатора id
   $inst->new[] = $obj;
}

static function addClean(DomainObject $obj ) {
   $self = self::instance();
   unset( $self->delete[$self->globalKey( $obj )] );
   unset( $self->dirty [$self->globalKey( $obj )] );
   $self->new = array_filter( $self->new,
          function( $a ) use ( $obj ) { return !( $a === $obj ); }
                          );
}

function performOperations() {
   foreach ( $this->dirty as $key=>$obj ) {
      $obj->finder()->update( $obj );
   }
   foreach ( $this->new as $key=>$obj ) {
      $obj->finder()->insert( $obj );
   }
   $this->dirty = array();
   $this->new   = array();
}
```

Класс `ObjectWatcher` по-прежнему соответствует шаблону Identity Map и продолжает выполнять свою функцию отслеживания всех объектов в системе с помощью свойства `$all`. В этом примере мы просто добавили дополнительные функциональные возможности к классу.

Аспекты шаблона Unit of Work для класса `ObjectWatcher` показаны на рис. 13.6.

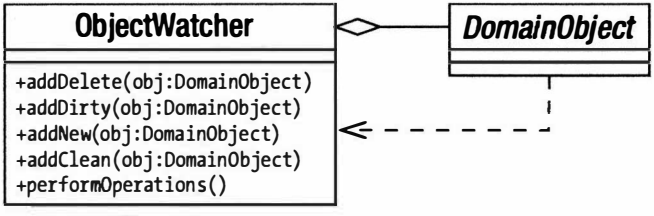


Рис. 13.6. Шаблон Unit of Work

Объекты описываются как "измененные", если они были изменены после извлечения из базы данных. Измененный объект сохраняется в свойстве массива `$dirty` (с помощью метода `addDirty()`), пока не приходит время обновить базу данных. Клиентский код может решить, что измененный объект не должен подвергаться обновлению, по собственным причинам. Клиентский код может обеспечить это, пометив измененный объект как неизмененный (с помощью метода `addClean()`). Как и можно было ожидать, вновь созданный объект должен быть добавлен к массиву `$new` (с помощью метода `addNew()`). Для объектов в этом массиве составлено расписание вставки в базу данных. В этих примерах мы не реализуем функцию удаления, но принцип должен быть понятен.

Каждый из методов `addDirty()` и `addNew()` добавляет объект к соответствующим свойствам массива. Но метод `addClean()` удаляет заданный объект из массива `$dirty`, помечая его как больше не ожидающий обновления.

Когда, наконец, приходит время обрабатывать все объекты, сохраненные в этих массивах, должен быть вызван метод `performOperations()` (вероятно, из класса контроллера или его вспомогательного класса). Этот метод проходит в цикле по массивам `$dirty` и `$new`, обновляя или добавляя объекты.

Класс `ObjectWatcher` теперь обеспечивает механизм обновления и вставки объектов. Но коду все еще не хватает способа добавления объектов к объекту `ObjectWatcher`.

Поскольку операции проводятся именно на этих объектах, вероятно, они сами — лучшее место для выполнения такого уведомления. Вот несколько вспомогательных методов, которые можно добавить к классу `DomainObject`. Также обратите внимание на метод конструктора.

```
// DomainObject
namespace woo\domain;
//...

abstract class DomainObject {
   private $id = -1;

   function __construct( $id=null ) {
      if ( is_null( $id ) ) {
         $this->markNew();
      } else {
         $this->id = $id;
      }
   }

   function markNew() {
      ObjectWatcher::addNew( $this );
   }

   function markDeleted() {
      ObjectWatcher::addDelete( $this );
   }

   function markDirty() {
      ObjectWatcher::addDirty( $this );
   }

   function markClean() {
      ObjectWatcher::addClean( $this );
```

```
    }

    function setId( $id ) {
        $this->id = $id;
    }

    function getId( ) {
        return $this->id;
    }

    function finder() {
        return self::getFinder( get_class( $this ) );
    }

    static function getFinder( $type=null ) {
        if ( is_null( $type ) ) {
            return HelperFactory::getFinder( get_called_class() );
        }
        return HelperFactory::getFinder( $type );
}//...
```

Прежде чем смотреть на код шаблона Unit of Work, стоит отметить, что у класса `Domain` есть методы `finder()` и `getFinder()`. Они работают точно так же, как `collection()` и `getCollection()`, запрашивая простой фабричный класс, `HelperFactory`, чтобы получить объекты `Mapper`, когда они понадобятся. Эта связь проиллюстрирована на рис. 13.3.

Как видите, метод конструктора помечает текущий объект как новый (вызывая `markNew()`), если ему не было передано свойство `$id`. Это похоже на магию, и здесь нужно проявлять некоторую внимательность. В таком виде этот код назначает новый объект для вставки в базу данных без какого-либо вмешательства со стороны создателя объектов. Представьте, что новый программист в вашей команде пишет сценарий для временного использования, чтобы протестировать работу программы, решающей поставленную задачу. Здесь нет никаких признаков персистентного кода, поэтому все должно быть достаточно безопасно, не так ли? А теперь представьте эти тестовые объекты, вероятно, с интересными временными именами, которые попадают в постоянное хранилище! Магия — это хорошо, но ясность — гораздо лучше. Наверное, лучше потребовать от клиентского кода передавать какой-то признак конструктору, чтобы отметить новый объект для вставки.

Нужно также добавить следующий фрагмент кода в класс `Mapper`.

```
// Mapper
function createObject( $array ) {
    $old = $this->getFromMap( $array['id']);
    if ( ! is_null( $old ) ) { return $old; }
    $obj = $this->doCreateObject( $array );
    $this->addToMap( $obj );
    $obj->markClean();
    return $obj;
}
```

Поскольку создание объекта включает его пометку в качестве нового с помощью вызова конструктором метода `ObjectWatcher::addNew()`, мы должны вызвать метод `markClean()`, иначе все без исключения объекты, извлеченные из базы данных, будут сохранены в конце запроса, а это не то, что нам нужно.

Единственное, что осталось сделать, — это добавить вызовы `markDirty()` к методам в классах Domain Model. Помните: измененный объект — это объект, который был изменен после его извлечения из базы данных. Это тот аспект шаблона, который подозрительно напоминает Domain Model. Очевидно, важно гарантировать, чтобы все методы, которые меняют состояние объекта, помечали его как измененный. Но поскольку это нужно делать вручную, вероятность ошибки (поскольку человеку свойственно ошибаться) вполне реальна.

Ниже приведены некоторые методы объекта `Space`, вызывающие `markDirty()`.

```
namespace woo\domain;
//...
class Space extends DomainObject {
//...
function setName( $name_s ) {
   $this->name = $name_s;
   $this->markDirty();
}

function setVenue(Venue $venue ) {
   $this->venue = $venue;
   $this->markDirty();
}
```

Вот код для добавления новых объектов `Venue` и `Space` к базе данных, взятый из класса `Command`.

```
require_once( "woo/domain/Venue.php");
require_once( "woo/domain/Space.php");

$venue = new \woo\domain\Venue( null, "The Green Trees" );
$venue->addSpace(
   new \woo\domain\Space( null, 'The Space Upstairs' ) );
$venue->addSpace(
   new \woo\domain\Space( null, 'The Bar Stage' ) );
// Это должно вызываться из контроллера или вспомогательного класса.
// Чтобы увидеть процесс записи, раскомментируйте оператор печати
// в классе \woo\domain\ObjectWatcher.
\woo\domain\ObjectWatcher::instance()->performOperations();
```

Я добавил код отладки к `ObjectWatcher`, поэтому можно увидеть, что происходит по окончании запроса.

```
inserting The Green Trees
inserting The Space Upstairs
inserting The Bar Stage
```

Поскольку объект-контроллер высокого уровня обычно вызывает метод `performOperations()`, все, что нужно сделать в большинстве случаев, — это создать или модифицировать объект, и класс Unit of Work (`ObjectWatcher`) сделает свою работу только один раз в конце запроса.

Результаты

Этот шаблон очень полезен, но есть несколько моментов, которые нужно иметь в виду. Вы должны быть уверены, что все операции изменения на самом деле помечают модифицированный объект как измененный. Невыполнение этого правила приведет к появлению ошибок, которые будет очень трудно обнаружить.

Возможно, вы захотите поискать другие способы тестирования измененных объектов. Кажется, что здесь подходит рефлексия, но вы должны оценить влияние такого тестирования на производительность — шаблон должен улучшать, а не ухудшать производительность.

Шаблон Lazy Load

Lazy Load — это один из основных шаблонов, которые большинство веб-программистов изучают сами очень быстро. Причина проста — это важнейший механизм, позволяющий избегать массовых обращений к базе данных, т.е. это то, к чему мы все стремимся.

Проблема

В примере, который рассматривается в этой главе, мы установили связь между объектами `Venue`, `Space` и `Event`. Когда создается объект `Venue`, он автоматически получает объект `SpaceCollection`. Если бы нам нужно было перечислить все объекты `Space` в `Venue`, то это действие автоматически вызвало бы запрос к базе данных, который вернул бы все объекты `Event`, связанные с каждым объектом `Space`. Они сохраняются в объекте `EventCollection`. Если мы не хотим просматривать события, нам придется совершить несколько обращений к базе данных без всякой причины. И если у вас много объектов `Venue`, у каждого из которых по два-три объекта `Space`, причем каждый объект `Space` управляет десятками и даже сотнями объектов `Event`, то получается слишком накладно.

Очевидно, в некоторых случаях нам нужно прерывать это автоматическое включение коллекций.

Вот код в `SpaceMapper`, который получает данные `Event`.

```
protected function doCreateObject( array $array ) {
   $obj = new \woo\domain\Space( $array['id'] );
   $obj->setName( $array['name'] );
   $ven_mapper = new VenueMapper();
   $venue = $ven_mapper->find( $array['venue'] );
   $obj->setVenue( $venue );
   $event_mapper = new EventMapper();
   $event_collection = $event_mapper->findBySpaceId( $array['id'] );
   $obj->setEvents( $event_collection );
   return $obj;
}
```

Метод `doCreateObject()` сначала получает объект `Venue`, с которым связан объект `Space`. Это несложно, потому что данный объект почти наверняка уже сохранен в объекте `ObjectWatcher`. Затем этот метод вызывает метод `EventMapper::findBySpaceId()`. Вот здесь у системы могут возникнуть проблемы.

Реализация

Как вы, наверное, знаете, применить шаблон Lazy Load — значит отложить получение значения свойства до тех пор, пока его действительно не запросит клиент.

Как мы видели, самый простой способ для этого — сделать задержку явной в содержащем объекте. Вот как это можно сделать в объекте `Space`.

```
// Space
function getEvents() {
   if ( is_null($this->events) ) {
      $this->events =
       self::getFinder(Event::class)->findBySpaceId( $this->getId() );
   }
return $this->events;
}
```

В этом методе проверяется, установлено свойство `$events` или нет. Если оно не установлено, то метод запрашивает средство поиска (т.е. `Mapper`) и использует его собственное свойство `$id` для получения объекта `EventCollection`, с которым он связан. Очевидно, чтобы этот метод избавил нас от ненужных запросов к базе данных, нам также нужно скорректировать код `SpaceMapper`, чтобы он автоматически не выполнял предварительную загрузку объекта `EventCollection`, как он это делал в предыдущем примере!

Этот подход будет отлично работать, хотя он немного беспорядочный. Не лучше ли убрать этот беспорядок?

В результате мы возвращаемся к реализации итератора, которая создает объект `Collection`. Мы уже скрыли один секрет за этим интерфейсом (то, что данные из базы данных могли еще не использоваться для создания экземпляра объекта приложения на момент обращения к нему клиента). Вероятно, мы можем скрыть еще больше.

Идея заключается в том, чтобы создать объект `EventCollection`, который откладывает обращение к базе данных до тех пор, пока не будет сделан запрос на это. Это означает, что клиентскому объекту (такому, как `Space`, например) никогда не нужно знать, что он содержит пустой объект `Collection` в своем первом экземпляре. Что касается клиента, то он содержит абсолютно нормальный объект `EventCollection`.

Вот объект `DeferredEventCollection`.

```
namespace woo\mapper;
//...
class DeferredEventCollection
            extends EventCollection {
   private $stmt;
   private $valueArray;
   private $run=false;

   function __construct(Mapper $mapper, \PDOStatement $stmt_handle,
                        array $valueArray ) {
      parent::__construct( null, $mapper );
      $this->stmt = $stmt_handle;
      $this->valueArray = $valueArray;
   }

   function notifyAccess() {
      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 `EventCollection`. Его конструктор требует объекты `EventMapper` и `PDOStatement` и массив элементов, который должен соответствовать подготовленному оператору. В первом случае класс ничего не делает, а только сохраняет свои свойства и ждет. Никаких запросов к базе данных не делается.

Если вы помните, в базовом классе `Collection` определяется пустой метод `notifyAccess()`, о котором я упоминал в разделе "Шаблон Data Mapper". Он вызывается из любого метода, вызов которого был выполнен извне.

В классе `DeferredEventCollection` этот метод переопределяется. Теперь, если кто-то попытается обратиться к `Collection`, этот класс будет знать, что пришло время перестать притворяться и получить какие-то реальные данные. Он делает это, вызывая метод `PDOStatement::execute()`. Вместе с методом `PDOStatement::fetch()` это дает массив полей, подходящих для передачи методу `Mapper::createObject()`.

Вот метод в `EventMapper`, который создает экземпляр `DeferredEventCollection`.

```
// EventMapper
namespace woo\mapper;
// ...
function findBySpaceId( $s_id ) {
   return new DeferredEventCollection(
                $this,
                $this->selectBySpaceStmt, array( $s_id ) );
}
```

Результаты

"Ленивая" загрузка — это привычка, которую стоит приобрести независимо от того, добавляете ли вы явно логику отложенной загрузки к доменным классам.

Помимо безопасности типа, особое преимущество от использования для свойств именно коллекции, а не массива — это возможность, позволяющая изменить процесс "ленивой" загрузки, если вам это понадобится.

Шаблон Domain Object Factory

Шаблон Data Mapper очень хорош, но у него есть некоторые недостатки. В частности, класс `Mapper` берет на себя слишком много. Он составляет SQL-операторы, преобразует массивы в объекты и, конечно, объекты обратно в массивы, готовые для добавления в базу данных. Такая многосторонность делает класс `Mapper` удобным и функциональным. Но она может уменьшить степень гибкости.

Это особенно верно, если `Mapper` должен обрабатывать много различных видов запросов или если другим классам нужно совместно использовать функциональные возможности `Mapper`. В оставшейся части главы я разделю шаблон Data Mapper, разбив его на ряд более специализированных шаблонов. При объединении этих мелких шаблонов воспроизводятся все обязанности, которые выполнялись в шаблоне Data Mapper, причем некоторые из них (если не все!) можно использовать в сочетании с этим шаблоном. Их хорошо описал Клифтон Нок (Clifton Nok) в книге *Data Access Patterns* (Addison Wesley 2003), и я использовал его названия в тех случаях, когда есть разночтения.

Давайте начнем с основной функции — генерации объектов приложения.

Проблема

Мы уже сталкивались с ситуацией, когда класс `Mapper` демонстрирует естественное ошибочное поведение. Конечно, метод `createObject()` используется `Mapper` внутри, но объектам `Collection` он тоже нужен для создания объектов приложения по запросу. Это требует от нас передать ссылку на `Mapper` при создании объекта `Collection`. Хотя нет ничего плохого в том, чтобы разрешить обратные вызовы (как мы делали в шаблонах `Visitor`, `Observer` и других), лучше передать ответственность за создание объекта приложения его собственному типу. Затем его могут совместно использовать классы `Mapper` и `Collection`.

Шаблон Domain Object Factory описан в книге *Data Access Patterns*.

Реализация

Представьте себе набор классов `Mapper`, в общих чертах организованный так, что каждый обращен к своему объекту приложения. Шаблон Domain Object Factory просто требует, чтобы вы извлекали метод `createObject()` из каждого класса `Mapper` и помещали его в собственный класс в параллельной иерархии. Эти новые классы показаны на рис. 13.7.

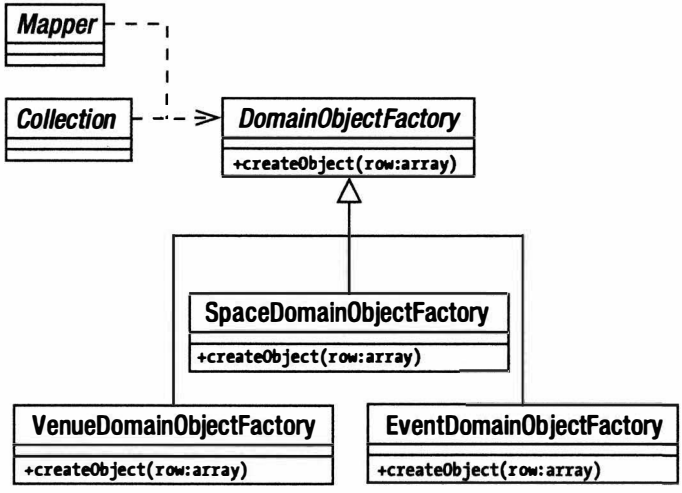


Рис. 13.7. Классы Domain Object Factory

У классов Domain Object Factory есть только одна основная обязанность, поэтому они обычно очень просты.

```
namespace woo\mapper;
//...
abstract class DomainObjectFactory {
   abstract function createObject( array $array );
}
```

Вот конкретная реализация.

```
namespace woo\mapper;
//...
class VenueObjectFactory extends DomainObjectFactory {
   function createObject( array $array ) {
```

```
        $obj = new \woo\domain\Venue( $array['id'] );
        $obj->setName( $array['name'] );
        return $obj;
    }
}
```

Конечно, чтобы не допустить дублирования и предотвратить ненужные обращения к базе данных, вы должны кешировать объекты, как это делал я с классом `Mapper`. Вы можете переместить сюда методы `addToMap()` и `getFromMap()` или создать связь по типу наблюдателя между `ObjectWatcher` и методами `createObject()`. Детали я оставляю вам. Только помните: не допускайте создания клонов объектов приложения, поскольку это сделает систему неуправляемой!

Результаты

Шаблон Domain Object Factory отделяет необработанные данные, полученные из базы данных, от данных полей объектов. Вы можете осуществлять любое количество корректировок внутри метода `createObject()`. Этот процесс прозрачен для клиента, обязанность которого — предоставлять необработанные данные. Если забрать эти функциональные возможности из класса `Mapper`, то они становятся доступными для других компонентов. Например, вот измененная реализация `Collection`.

```
namespace woo\mapper;
//...
abstract class Collection {
    protected $dofact;
    protected $total = 0;
    protected $raw = array();
    // ...
    function __construct( array $raw=null,
                          \woo\mapper\DomainObjectFactory $dofact=null ) {
        if ( ! is_null( $raw ) && ! is_null( $dofact ) ) {
            $this->raw = $raw;
            $this->total = count( $raw );
        }
        $this->dofact = $dofact;
    }
// ...
```

Объект `DomainObjectFactory` можно использовать для генерации объектов по запросу.

```
if ( isset( $this->raw[$num] ) ) {
    $this->objects[$num]=$this->dofact->createObject( $this->raw[$num] );
    return $this->objects[$num];
}
```

Поскольку объекты `DomainObjectFactory` отделены от базы данных, их можно более эффективно использовать для тестирования. Мы можем, например, создать фиктивный объект `DomainObjectFactory`, чтобы протестировать код `Collection`. Сделать это намного проще, чем имитировать целый объект `Mapper` (подробнее о фиктивных объектах и объектах-заглушках см. в главе 18).

Один общий результат разбиения единого компонента на составные части — это неизбежное размножение классов. Поэтому нельзя недооценивать вероятность путаницы. Даже если каждый компонент и его связи с другими равноправными ком-

понентами логичны и явно определены, мне часто бывает трудно составлять пакеты, содержащие десятки компонентов с похожими названиями.

Ситуация сначала ухудшится, а затем — улучшится. Я уже вижу еще одно проблемное направление в Data Mapper. Метод `Mapper::getCollection()` был удобным, но, опять-таки, другим классам может понадобиться получить объект `Collection` для определенного типа приложения, не используя класс, обращенный к базе данных. Итак, у нас есть два связанных абстрактных компонента: `Collection` и `DomainObjectFactory`. В соответствии с объектом приложения, с которым мы работаем, мы будем требовать другой набор конкретных реализаций, например `VenueCollection` и `VenueDomainObjectFactory` или `SpaceCollection` и `SpaceDomainObjectFactory`. Конечно, эта проблема приводит нас непосредственно к шаблону Abstract Factory. На рис. 13.8 показан класс `PersistenceFactory`. Я буду использовать его для организации различных компонентов, из которых состоят несколько следующих шаблонов.

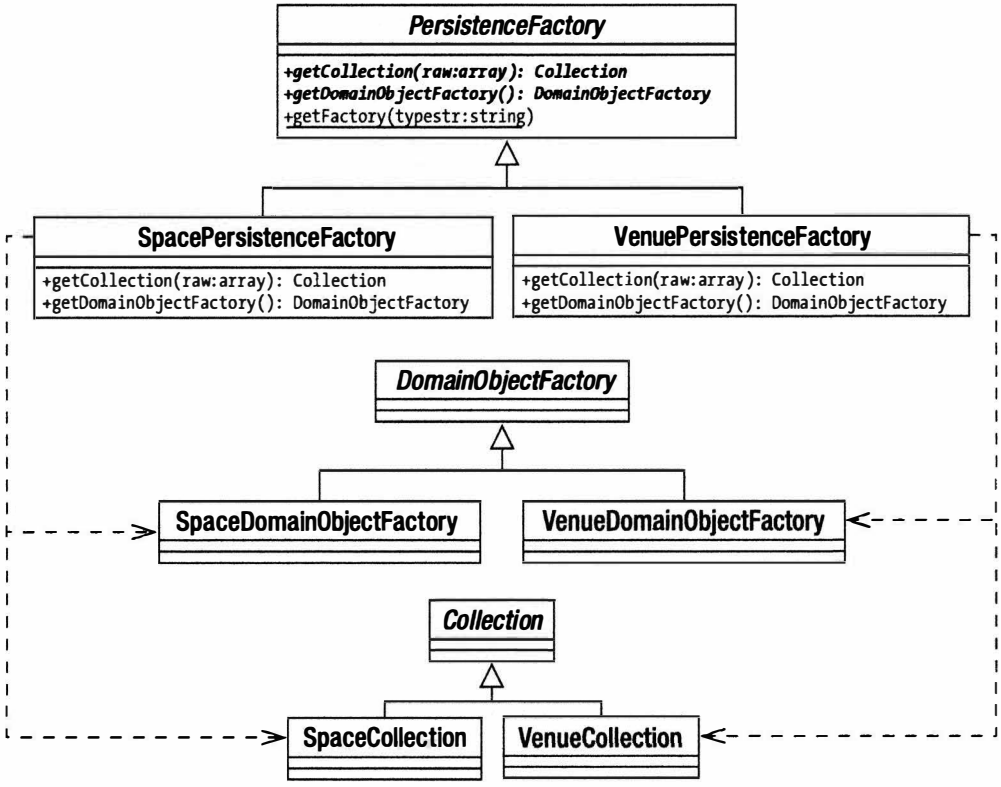


Рис. 13.8. Использование шаблона Abstract Factory для организации связанных компонентов

Шаблон Identity Object

Реализация `Mapper`, которую я здесь представил, отличается недостаточной гибкостью, когда речь идет о нахождении объектов приложения. Найти отдельный объект — это не проблема. Найти все подходящие объекты приложения так же просто. Но если нужно сделать что-то среднее между этими задачами, то тре-

буется добавить специальный метод для создания запроса (в данном случае — `EventMapper::findBySpaceId()`).

Шаблон Identity Object (который у Алура и других также называется Data Transfer Object) инкапсулирует критерии запроса, тем самым отделяя систему от синтаксиса базы данных.

Проблема

Трудно знать наперед, что вам или другим программистам клиентского кода понадобится искать в базе данных. Чем сложнее объект приложения, тем больше фильтров может понадобиться в запросе. В некоторой степени можно решить эту проблему, добавляя дополнительные методы к классам `Mapper` от случая к случаю. Конечно, это не очень гибкий вариант и может появиться дублирование, когда вам понадобится создать много похожих, но различных запросов как внутри одного класса `Mapper`, так и во всех классах `Mapper` в системе.

Шаблон Identity Object инкапсулирует условную часть запроса к базе данных таким способом, что различные комбинации могут объединяться во время выполнения. Например, если объект приложения называется `Person`, то клиент может вызывать методы для Identity Object, чтобы определить мужчину в возрасте от 30 до 40 лет ростом до 180 см. Класс должен быть спроектирован так, чтобы условия объединялись гибко (например, если вас не интересует рост человека или вы хотите удалить нижнюю границу возраста). Шаблон Identity Object до некоторой степени ограничивает возможности программирования клиентского кода. Если вы не написали код, в котором предусмотрено поле `income`, то его нельзя будет внести в запрос, не делая изменений. А возможность применять различные комбинации условий — это шаг вперед в отношении гибкости. Давайте посмотрим, как это работает.

Реализация

Шаблон Identity Object обычно состоит из набора методов, которые можно вызывать для построения критерия запроса. Определив состояние объекта, вы можете передать его методу, который отвечает за создание SQL-оператора.

На рис. 13.9 показан типичный набор классов `IdentityObject`.

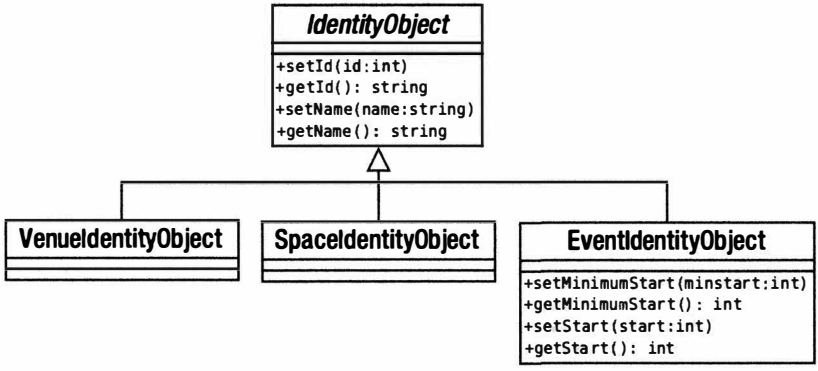


Рис. 13.9. Работа с критериями запросов с помощью Identity Object

Вы можете использовать базовый класс, чтобы управлять общими операциями, и обеспечить, чтобы объекты критериев совместно использовали некоторый тип. Вот реализация, которая еще проще, чем классы, показанные на рис. 13.9.

```
namespace woo\mapper;
//...
class IdentityObject {
   private $name = null;

   function setName( $name ) {
      $this->name=$name;
   }

   function getName() {
      return $this->name;
   }
}

class EventIdentityObject
          extends IdentityObject {
   private $start = null;
   private $minstart = null;

   function setMinimumStart( $minstart ) {
      $this->minstart = $minstart;
   }

   function getMinimumStart() {
      return $this->minstart;
   }

   function setStart( $start ) {
      $this->start = $start;
   }

   function getStart() {
      return $this->start;
   }
}
```

Здесь нет ничего слишком обременительного. Классы просто сохраняют предоставленные данные и отдают их по запросу. Вот код, который может использовать `SpaceIdentityObject` для создания конструкции `WHERE`.

```
$idobj = new EventIdentityObject();
$idobj->setMinimumStart( time() );
$idobj->setName( "A Fine Show" );

$comps = array();
$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 );
```

Эта модель будет работать довольно хорошо, но она не устраивает мою ленивую натуру. Для большого объекта приложения количество методов-получателей и методов-установщиков, которые нужно создать, окажется огромным. Затем, следуя этой модели, вам придется написать код для вывода каждого условия в операторе `WHERE`. Я не мог себя заставить обработать все случаи в коде примера (у меня нет метода `setMaximumStart()`), поэтому представьте мою радость при создании объектов Identity Object в реальных задачах.

К счастью, существуют различные стратегии, которые можно применять для автоматизации сбора данных и генерации SQL-кода. Раньше, например, в базовом классе я заносил в ассоциативные массивы имена полей базы данных. Они сами индексировались при использовании типов сравнения: больше чем, равно, меньше чем или равно. В дочерних классах предусматривались удобные методы для добавления этих данных к основной структуре. Построитель SQL-запросов затем может пройти по структуре в цикле, чтобы создать свой запрос динамически. Я уверен, что реализация подобной системы — это просто вопрос представления, поэтому я рассмотрю здесь один из вариантов.

Я буду использовать текучий интерфейс (fluent interface). Это класс, методы-установщики которого возвращают экземпляры объектов, позволяя пользователям объединять объекты в цепочку свободным образом, как в естественном языке. Это удовлетворит мою лень, но также, я надеюсь, даст программисту клиентского кода гибкий способ определения критериев.

Я начинаю с создания класса `\woo\mapper\Field`, предназначенного для хранения данных сравнения для каждого поля, которое окажется в конструкции `WHERE`.

```
namespace woo\mapper;

class Field {
   protected $name=null;
   protected $operator=null;
   protected $comps=array();
   protected $incomplete=false;

   // Устанавливает имя поля, например age
   function __construct( $name ) {
      $this->name = $name;
   }

  // Добавляет оператор и значение для проверки
  // (> 40, например) и помещает его в свойство $comps
  function addTest( $operator, $value ) {
     $this->comps[] = array( 'name' => $this->name,
                             'operator' => $operator,
                                'value' => $value );
   }

   // $comps - это массив, поэтому мы можем сравнить одно поле с другим
   //   несколькими способами
   function getComps() { return $this->comps; }

   // Если массив $comps не содержит элементов, значит, у нас есть
   // данные для сравнения и это поле не готово для использования
   // в запросе
```

```
    function isIncomplete() { return empty( $this->comps); }
}
```

Этому простому классу передается имя поля, которое он сохраняет внутри. С помощью метода `addTest()` в классе создается массив элементов оператора и значений. Это позволяет поддерживать несколько операторов сравнения для одного поля. А теперь рассмотрим новый класс `IdentityObject`.

```
namespace woo\mapper;

class IdentityObject {
    protected $currentfield = null;
    protected       $fields = array();
    private            $and = null;
    private        $enforce = array();

    // Конструктор identity object может запускаться
    // без параметров или с именем поля
    function __construct( $field=null, array $enforce=null ) {
        if ( ! is_null( $enforce ) ) {
            $this->enforce = $enforce;
        }
        if ( ! is_null( $field ) ) {
            $this->field( $field );
        }
    }

    // Имена полей, на которые наложено это ограничение
    function getObjectFields() {
        return $this->enforce;
    }

    // Вводим новое поле.
    // Генерируется ошибка, если текущее поле неполное
    // (т.е. age, а не age > 40).
    // Этот метод возвращает ссылку на текущий объект
    // и тем самым разрешает свободный синтаксис
    function field( $fieldname ) {
        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;
    }

    // Есть ли уже поля у identity object
    function isVoid() {
        return empty( $this->fields );
    }
```

```
    // Заданное имя поля допустимо?
    function enforceField( $fieldname ) {
        if ( ! in_array( $fieldname, $this->enforce ) &&
             ! empty( $this->enforce ) ) {
            $forcelist = implode( ', ', $this->enforce );
             throw new \Exception("{$fieldname} не является корректным полем
($forcelist)");
        }
    }

    // Добавим оператор равенства к текущему полю
    // т.е. 'age' становится age=40.
    // Возвращает ссылку на текущий объект (с помощью operator())
    function eq( $value ) {
        return $this->operator( "=", $value );
    }

    // Меньше чем
    function lt( $value ) {
        return $this->operator( "<", $value );
    }

    // Больше чем
    function gt( $value ) {
        return $this->operator( ">", $value );
    }

    // Выполняет работу для методов operator.
    // Получает текущее поле и добавляет значение оператора
    // и результаты проверки к нему
    private function operator( $symbol, $value ) {
        if ( $this->isVoid() ) {
            throw new \Exception("Поле не определено");
        }
        $this->currentfield->addTest( $symbol, $value );
        return $this;
    }

    // Возвращает все сравнения, созданные до сих пор в ассоциативном массиве
    function getComps() {
        $comparisons = array();
        foreach ( $this->fields as $key => $field ) {
            $comparisons = array_merge( $ret, $field->getComps() );
        }
        return $comparisons;
    }
}
```

Самый простой способ понять, что здесь происходит, — это начать с клиентского кода и двигаться в обратном направлении.

```
namespace woo\mapper;
require_once("woo/mapper/IdentityObject.php");
$idobj = new IdentityObject();

$idobj->field("name")->eq("The Good Show")
```

```
    ->field("start")->gt( time() )
                    ->lt( time()+(24*60*60) );
```

Мы начинаем с создания объекта `IdentityObject`. Вызов метода `add()` приводит к созданию объекта `Field` и присвоению ссылки на него свойству `$currentfield`. Обратите внимание на то, что метод `add()` возвращает ссылку на объект `IdentityObject`. Это позволяет добавить больше вызовов методов на обратном пути вызова метода `add()`. В каждом методе сравнения `eq()`, `gt()` и так далее вызывается метод `operator()`. Он проверяет, существует ли текущий объект `Field`, с которым нужно работать, и, если существует, передает символ оператора и предоставленное значение. И снова, `eq()` возвращает ссылку на объект, так что мы можем добавить новые проверки или снова вызвать `add()`, чтобы начать работать с новым полем.

Обратите внимание на то, что клиентский код почти всегда похож на предложение: поле `"name"` равно `"The Good Show"`, а поле `"start"` больше, чем текущее время, но меньше, чем целый день.

Конечно, теряя жестко закодированные методы, мы также теряем некоторую степень безопасности. Именно для этого предназначен массив `$enforce`. Подклассы могут вызвать базовый класс с набором ограничений.

```
namespace woo\mapper;

class EventIdentityObject extends IdentityObject {
   function __construct( $field=null ) {
      parent::__construct( $field,
      array('name', 'id','start','duration', 'space' ) );
   }
}
```

Класс `EventIdentityObject` теперь вводит в силу набор полей. Вот что произойдет, если мы попытаемся работать со случайным именем поля.

```
PHP Fatal error: Uncaught exception 'Exception' with message 'banana not a
legal field (name, id, start, duration, space)'...
```

Результаты

Шаблон Identity Object позволяет программистам клиентского кода определять критерии поиска без ссылки на запрос к базе данных. Он также избавляет от необходимости создавать специальные методы запросов для различных видов операций поиска, которые могут понадобиться пользователю.

Одна из целей Identity Object — оградить пользователей от деталей реализации базы данных. Но если вы создаете автоматическое решение, такое как текучий интерфейс из предыдущего примера, то очень важно, чтобы используемые метки явно ссылались на объекты приложения, а не на низкоуровневые имена столбцов. Но если они не соответствуют друг другу, то вы должны создать для них механизм сопоставления.

Если вы используете специализированные объекты, по одному для каждого объекта приложения, то полезно использовать шаблон Abstract Factory (такой, как `PersistenceFactory`, описанный в предыдущем разделе), чтобы обслуживать их вместе с другими объектами, связанными с текущим объектом приложения.

А теперь, когда мы можем воспроизвести критерий поиска, воспользуемся им для создания самого запроса.

Шаблоны Selection Factory и Update Factory

Я уже забрал некоторые обязанности у классов `Mapper`. При наличии данных шаблонов классу `Mapper` не нужно создавать объекты или коллекции. Если критерии запросов обрабатываются в объектах Identity Object, больше нет необходимости управлять несколькими вариациями в методе `find()`. Следующий этап — это удаление обязанности по созданию запроса.

Проблема

Любая система, которая обращается к базе данных, должна генерировать запросы, но сама система организована вокруг объектов приложения и логики работы, а не базы данных. Можно сказать, что многие шаблоны, описанные в данной главе, "наводят мосты" между табличной базой данных и более естественными древовидными структурами приложения. Но существует момент преобразования — точка, в которой данные приложения трансформируются в форму, которую может понять база данных. Именно в этой точке происходит настоящее отделение.

Реализация

Конечно, мы и раньше видели многие эти функции в шаблоне Data Mapper. Но в этой специализации мы можем воспользоваться преимуществами дополнительных функций, предоставляемых шаблоном Identity Object. Это способствует тому, чтобы сделать создание шаблонов более динамичным, просто потому, что возможное количество вариантов столь высоко.

На рис. 13.10 показаны простые шаблоны Selection Factory и Update Factory.

Шаблоны Selection Factory и Update Factory, опять-таки, обычно организованы так, что они соответствуют объектам приложения в системе (возможно, посредниками которых являются объекты Identity Object). По этой причине они также являются кандидатами для моего класса `PersistenceFactory` — шаблон Abstract Factory, который мы поддерживаем в качестве "универсального магазина" для инструментов персистентности объектов приложения. Вот реализация базового класса для объектов Update Factory.

```
namespace woo\mapper;

abstract class UpdateFactory {
   abstract function newUpdate( \woo\domain\DomainObject $obj );

   protected function buildStatement( $table, array $fields,
                                      array $conditions=null ) {
      $terms = array();
      if ( ! is_null( $conditions ) ) {
         $query  = "UPDATE {$table} SET ";
         $query .= implode ( " = ?,", array_keys( $fields ) ). " = ?";
         $terms  = array_values( $fields );
         $cond   = array();
         $query .= " WHERE ";
         foreach ( $conditions as $key=>$val ) {
            $cond[]="$key = ?";
            $terms[]=$val;
         }
         $query .= implode( " AND ", $cond );
```

```
        } else {
            $query  = "INSERT INTO {$table} (";
            $query .= implode( ",", array_keys($fields) );
            $query .= ") VALUES (";
            foreach ( $fields as $name => $value ) {
                $terms[]=$value;
                $qs[]='?';
            }
            $query .= implode( ",", $qs );
            $query .= ")";
        }
        return array( $query, $terms );
    }
}
```

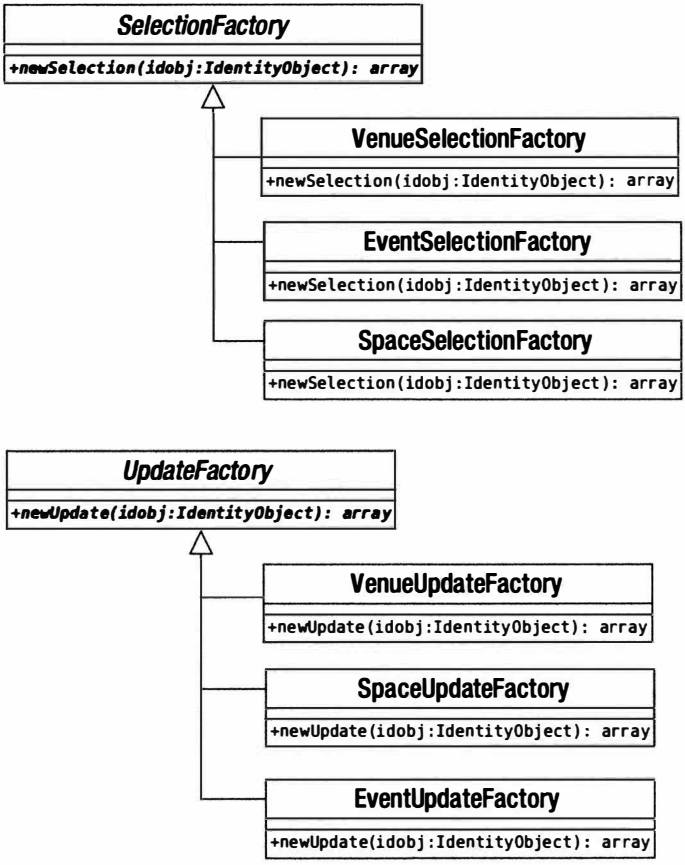


Рис. 13.10. Шаблоны SelectionFactory и UpdateFactory

С точки зрения интерфейса единственное, что делает этот класс, — определяет метод `newUpdate()`. Он вернет массив, содержащий строку запроса, и список условий, которые нужно к ней применить. Метод `buildStatement()` выполняет общую работу по созданию запроса на обновление, причем работу, связанную с отдельны-

ми объектами приложения, выполняют дочерние классы. Методу `buildStatement()` передаются имя таблицы, ассоциативный массив полей и их значений и аналогичный ассоциативный массив условий. Метод объединяет все это для создания запроса. Вот конкретный класс `UpdateFactory`.

```
namespace woo\mapper;

class VenueUpdateFactory extends UpdateFactory {
   function newUpdate( \woo\domain\DomainObject $obj ) {
      // Проверка типов удалена
      $id = $obj->getId();
      $cond = null;
      $values['name'] = $obj->getName();
      if ( $id > -1 ) {
         $cond['id'] = $id;
      }
      return $this->buildStatement( "venue", $values, $cond );
   }
}
```

В этой реализации я работаю непосредственно с объектом `DomainObject`. В системах, где он может работать со многими объектами одновременно при обновлении, мы можем использовать Identity Object для определения набора, который хотим обработать. Это формирует основу массива `$cond`, который здесь содержит только данные `id`.

Метод `newUpdate()` извлекает данные, необходимые для генерации запроса. Это процесс, посредством которого данные объекта преобразуются в информацию базы данных.

Обратите внимание: методу `newUpdate()` можно передать любой объект типа `DomainObject`. Дело в том, что все классы `UpdateFactory` могут совместно использовать интерфейс. Неплохо было бы добавить дополнительную проверку типа, чтобы гарантировать, что неправильный объект передан не будет.

Ниже представлен небольшой фрагмент кода, в котором проверяется работа класса `VenueUpdateFactory`.

```
namespace woo\mapper;
require_once( "woo/mapper/VenueUpdateFactory.php" );
require_once( "woo/domain/Venue.php" );

$vuf = new VenueUpdateFactory();
print_r( $vuf->newUpdate( new \woo\domain\Venue( 334, "The Happy Hairband" ) )
);
```

```
Array
(
    [0] => UPDATE venue SET name = ? WHERE id = ?
    [1] => Array
        (
            [0] => The Happy Hairband
            [1] => 334
        )
)
```

Рассмотрим аналогичную структуру для классов `SelectionFactory`. Вот базовый класс.

```
namespace woo\mapper;

abstract class SelectionFactory {
   abstract function newSelection(IdentityObject $obj );

   function buildWhere(IdentityObject $obj ) {
      if ( $obj->isVoid() ) {
         return array( "", array() );
      }
      $compstrings = array();
      $values = array();
      foreach ( $obj->getComps() as $comp ) {
         $compstrings[] = "{$comp['name']} {$comp['operator']} ?";
         $values[] = $comp['value'];
      }
      $where = "WHERE " . implode( " AND ", $compstrings );
      return array( $where, $values );
   }
}
```

И снова в классе определен общедоступный интерфейс в форме абстрактного класса. Метод `newSelection()` ожидает `IdentityObject`. Также требует `IdentityObject`, но локальный по отношению к типу, — вспомогательный метод `buildWhere()`. Он использует метод `IdentityObject::getComps()`, чтобы получить информацию, необходимую для создания конструкции `WHERE`, и составить список значений, причем и то, и другое он возвращает в двухэлементном массиве.

Вот конкретный класс `SelectionFactory`.

```
namespace woo\mapper;
//..
class VenueSelectionFactory extends SelectionFactory {
   function newSelection(IdentityObject $obj ) {
      $fields = implode( ',', $obj->getObjectFields() );
      $core = "SELECT $fields FROM venue";
      list( $where, $values ) = $this->buildWhere( $obj );
      return array( $core." ".$where, $values );
   }
}
```

Он создает основу SQL-оператора, а затем вызывает метод `buildWhere()`, чтобы добавить условный оператор. На самом деле единственное, что отличает один конкретный `SelectionFactory` от другого в моем тестовом коде, — это имя таблицы. Если я считаю, что в ближайшее время мне не понадобится уникальная специализация, то уберу эти подклассы и буду использовать один конкретный класс `SelectionFactory`, который будет запрашивать имя таблицы из `PersistenceFactory`.

И вот небольшой фрагмент клиентского кода.

```
namespace woo\mapper;
require_once( "woo/mapper/VenueSelectionFactory.php" );
require_once( "woo/mapper/VenueIdentityObject.php" );
require_once( "woo/domain/Venue.php" );

$vio = new VenueIdentityObject();
```

```
$vio->field("name")->eq("The Happy Hairband");

$vsf = new VenueSelectionFactory();
print_r( $vsf->newSelection( $vio ) );
```

```
Array
(
    [0] => SELECT name,id FROM venue WHERE name = ?
    [1] => Array
        (
            [0] => The Happy Hairband
        )
)
```

Результаты

Использование общей реализации Identity Object упрощает применение класса `SelectionFactory` с единственным параметром. Если вы предпочитаете жестко закодированные Identity Object, которые состоят из списка методов-получателей и методов-установщиков, то, скорее всего, вам придется создавать по одному `SelectionFactory` на объект приложения.

Одно из огромных преимуществ объектов Query Factory вместе с Identity Object — диапазон запросов, которые можно генерировать. Но это может также вызвать проблемы при кешировании. Данные методы генерируют запросы "на лету", и трудно узнать, не происходит ли дублирование. Возможно, имеет смысл создать средство сравнения объектов Identity Object, чтобы можно было вернуть кешированную строку, не выполняя всей этой работы. Аналогичный вид объединения операторов базы данных можно рассмотреть также на более высоком уровне.

Еще один вопрос комбинации шаблонов, который рассматривался в этой главе, — то, что они гибкие, но не *настолько* гибкие. Я имею в виду, что они в высшей степени должны обладать способностью приспосабливаться, но в определенных пределах. И остается мало возможностей для исключительных случаев. Классы `Mapper`, которые труднее создавать и поддерживать, очень легко приспосабливаются к любым ошибкам в работе программы или операциям с данными, которые может понадобиться выполнять за их аккуратными API. Эти более изящные шаблоны страдают от того, что при их сфокусированных обязанностях и акценте на композицию может быть трудно пойти наперекор разуму и сделать что-то тупое, но эффективное.

К счастью, мы не потеряли интерфейс более высокого уровня — у нас еще есть уровень контроллера, на котором мы можем показать свое мастерство, если потребуется.

Что теперь осталось от Data Mapper

Итак, мы забрали из Data Mapper функции создания объектов, запросов и коллекций, не говоря уже об управлении условиями. Что же от него осталось? То, что нужно от Data Mapper в рудиментарной форме. Нам по-прежнему нужен объект, который находится над другими созданными объектами и координирует их действия. Это поможет выполнять задачи кеширования и управлять подключением к базе данных (хотя работа с базой данных может быть делегирована еще выше). Клифтон Нок называет эти контроллеры уровня данных сборщиками доменных объектов.

Приведем пример.

```
namespace woo\mapper;
//..
class DomainObjectAssembler {
   protected static $PDO;

   function __construct(PersistenceFactory $factory ) {
      $this->factory = $factory;
      if ( ! isset(self::$PDO) ) {
        $dsn = \woo\base\ApplicationRegistry::getDSN( );
        if ( is_null( $dsn ) ) {
           throw new \woo\base\AppException( "DSN не определен" );
        }
        self::$PDO = new PDO( $dsn );
        self::$PDO->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
      }
   }

   function getStatement( $str ) {
      if ( ! isset( $this->statements[$str] ) ) {
         $this->statements[$str] = self::$PDO->prepare( $str );
      }
      return $this->statements[$str];
   }

   function findOne(IdentityObject $idobj ) {
      $collection = $this->find( $idobj );
      return $collection->next();
   }

   function find(IdentityObject $idobj ) {
      $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 );
   }

   function insert( \woo\domain\DomainObject $obj ) {
      $upfact = $this->factory->getUpdateFactory( );
      list( $update, $values ) = $upfact->newUpdate( $obj );
      $stmt = $this->getStatement( $update );
      $stmt->execute( $values );
      if ( $obj->getId() < 0 ) {
         $obj->setId( self::$PDO->lastInsertId() );
      }
      $obj->markClean();
   }
}
```

Как видите, это не абстрактный класс. Вместо того чтобы самостоятельно разбивать себя на специализации, он использует `PersistenceFactory`, чтобы гарантировать, что он получит правильные компоненты для текущего объекта приложения.

На рис. 13.11 показаны участники высокого уровня, которые мы создали, когда забирали функции у Mapper.

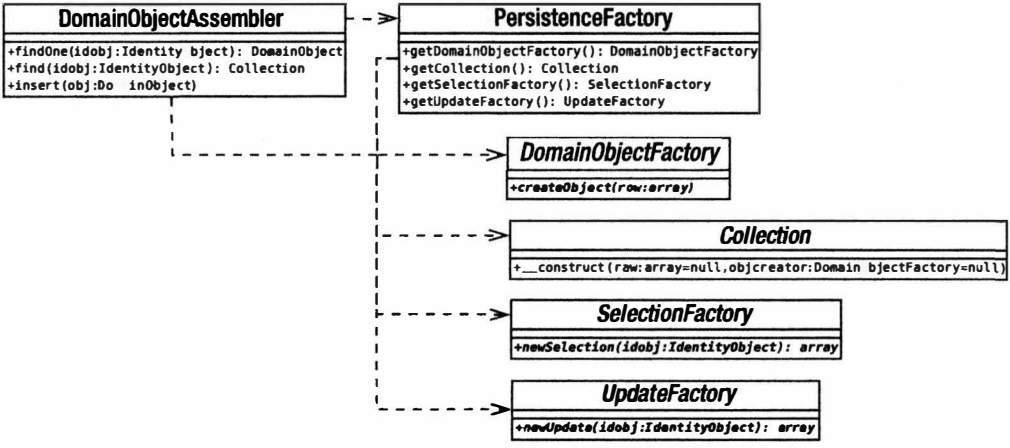


Рис. 13.11. Некоторые персистентные классы, созданные в данной главе

Помимо осуществления связи с базой данных и выполнения запросов, этот класс управляет объектами SelectionFactory и UpdateFactory. В случае SelectionFactory он также работает либо с классом Collection, либо непосредственно с DomainObjectFactory, чтобы сгенерировать возвращаемые значения.

С точки зрения клиента, получить DomainObjectFactory легко. Это просто вопрос получения нужного конкретного объекта PersistenceFactory.

```
$factory = \woo\mapper\PersistenceFactory::getFactory( 'woo\\domain\\Venue' );
$finder  = new \woo\mapper\DomainObjectAssembler( $factory );
```

Хотя, конечно, было бы еще проще добавить метод getFinder() к самому PersistenceFactory и преобразовать предыдущий пример в строку, подобную следующей.

```
$finder = \woo\mapper\PersistenceFactory::getFinder( "woo\\domain\\Venue" );
```

Но я оставляю эту работу вам.

Программист клиентского кода должен затем перейти к получению коллекции объектов Venue.

```
$idobj = $factory->getIdentityObject()->field('name')
                 ->eq('The Eyeball Inn');
$collection = $finder->find( $idobj );

foreach ( $collection as $venue ) {
  (print $venue->getName() . "\n" );
}
```

Резюме

Как всегда, выбор шаблонов зависит от природы проблемы. Я предпочитаю Data Mapper, работающий с 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, в то же время поддерживая аккуратный

интерфейс и отделенный уровень базы данных. Например, мне может понадобиться оптимизировать SQL-запрос или использовать объединение, чтобы получить данные из нескольких таблиц. Даже используя сложный сторонний каркас, основанный на шаблонах, вы можете обнаружить, что это затейливое объектно-реляционное соответствие дает не совсем то, что нужно. Один из критериев хорошего каркаса и хорошей доморощенной системы — простота, с которой можно вставить в нее новый модуль, не нарушая общей целостности всей системы. Мне нравятся изящные, красиво сконструированные решения, но ,тем не менее, я прагматик!

В этой главе я изложил очень объемный материал. Мы изучили следующие шаблоны.

- *Data Mapper*. Создает специализированные классы для отображения объектов Domain Model в реляционные базы данных и из них.
- *Identity Map*. Отслеживает все объекты в системе, чтобы предотвратить повторное создание экземпляров и лишние обращения к базе данных.
- *Unit of Work*. Автоматизирует процесс, посредством которого объекты сохраняются в базе данных, и гарантирует, что обновляются только измененные объекты, а вставляются — только вновь созданные.
- *Lazy Load*. Отсрочивает создание объектов и даже запросы к базе данных, пока они действительно не понадобятся.
- *Domain Object Factory*. Инкапсулирует функции создания объектов.
- *Identity Object*. Позволяет клиентам создавать критерии запросов без ссылки на основную базу данных.
- *Query (Selection и Update) Factory*. Инкапсулирует логику построения SQL-запросов.
- *Domain Object Assembler*. Создает контроллер, который управляет процессом высокого уровня по сохранению и извлечению данных.

В следующей главе мы сделаем долгожданную передышку от изучения кода, и я расскажу о более общих методах, которые помогут создать успешный проект.

Часть IV

Практика

Глава 14

Хорошие и плохие методы работы

До сих пор мы занимались кодированием, делая акцент на роли проектирования в создании гибких и повторно используемых инструментов и приложений. Но разработка не заканчивается кодом. Прочитав книги и пройдя курсы, можно получить прочные знания языка, но как только дело дойдет до запуска и развертывания проекта, можно столкнуться с проблемами.

В этой главе мы отойдем от кода и познакомимся с некоторыми инструментами и методами, создающими основу успешного процесса разработки. В главе будут рассмотрены следующие темы.

- *Пакеты сторонних разработчиков:* где их найти и когда использовать.
- *Построение:* создание и развертывание пакетов.
- *Контроль версий:* гармонизация процесса разработки.
- *Документация:* написание кода, который легко понять, использовать и расширить.
- *Модульное тестирование:* инструмент для автоматического обнаружения и предотвращения ошибок.
- *Непрерывная интеграция:* использование приведенных выше практик и набора средств для автоматизации процесса построения и тестирования проекта, которые могут предупредить разработчика о возникших проблемах.

Что осталось за рамками кода

Когда после самостоятельной практики я впервые начал работу в команде разработчиков, то был поражен тем, как много знают мои коллеги-разработчики. Мы бесконечно спорили по вопросам, казалось, жизненной важности: какой текстовый редактор наилучший? должна ли команда стандартизировать интегрированную среду разработки? должны ли мы устанавливать стандарт кодирования? как следует тестировать код? необходимо ли документировать процесс разработки? Иногда эти вопросы казались более важными, чем сам код, и, видимо, мои коллеги приобрели свои энциклопедические знания в этой области путем некоего странного процесса осмоса.

В книгах по PHP, Perl и Java, которые я читал, рассматривался лишь код. Как я уже говорил, в большинстве книг по платформам программирования, как правило,

даются только описания функций и синтаксиса и не рассматриваются вопросы проектирования. Если проектирование — это не по теме, то можно быть уверенным, что такие вопросы, как контроль версий и тестирование, обсуждаться также не будут. Это не значит, что я критикую такие книги. Если цель книги — осветить основные функции языка, нет ничего удивительного, если это все, что она делает.

Но, изучая вопросы программирования, я понял, что пренебрегал многими механизмами ежедневной жизни проекта. Я обнаружил, что от некоторых этих деталей зависит успех или неудача проектов, в разработке которых я участвовал. В данной и последующих главах мы отойдем от рассмотрения исключительно кода и изучим некоторые инструменты и методы, от которых может зависеть успех проектов.

Изобретаем велосипед

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 проблемы. Это может быть также одним из лучших способов научиться ремеслу. Создавая пакет, вы проникаете в суть проблемы и собираете новые методы, которые могут понадобиться для более широких приложений. Тем самым вы делаете инвестиции одновременно и в проект, и в свои знания и навыки. Поместив функциональные возможности внутри системы, вы избавляете пользователей от необходимости загружать пакеты сторонних фирм и не заниматься время от времени непростыми вопросами лицензирования. И вы испытаете огромное чувство удовлетворения, когда, протестировав компонент, который спроектировали сами, обнаружите, что — о, чудо из чудес! — он работает, причем в точности так, как вы и написали.

Но, конечно, у этого есть и оборотная сторона. Многие пакеты представляют собой инвестиции в виде тысяч человеко-часов: ресурс, которого у вас, возможно, нет. Чтобы решить эту проблему, вы можете разработать только функции, необходимые конкретно для вашего проекта, в то время как продукт сторонних фирм, как правило, выполняет также множество других дополнительных функций. Но остается вопрос: если уже существует инструмент в свободном доступе, зачем тратить свой талант на его воспроизведение? 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 велосипеда. Анализировать проблемы и придумывать их решения — это основная часть того, что мы делаем как программисты. Но серьезно заняться архитектурой гораздо полезнее и перспективнее, чем писать что-то вроде "клея", чтобы соединить три-четыре существующих компонента. Когда мной овладевает это искушение, я напоминаю себе о прошлых проектах. Хотя в моей практике решение начать все "с нуля" никогда не было роковым для проекта, я видел, что это 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 помню о картине в целом.

После составления плана проекта мне нужно понять, какие функции должны находиться внутри моего ядра кода, а какие нужно позаимствовать на стороне. Например, приложение может генерировать (или читать) RSS-канал; вам нужно 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.

Определив задачи, сначала посетите веб-сайт PEAR по адресу `http://pear.php.net`. PEAR (PHP Extension and Application Repository — хранилище расширений и приложений PHP) — это официально поддерживаемое хранилище пакетов, качество которых контролируется. Это также механизм легкой инсталляции пакетов и управления их взаимозависимостями. Подробнее о PEAR я расскажу в следующей главе, где вы узнаете, как использовать функциональные возможности PEAR для создания собственных пакетов. Чтобы дать вам некоторое представление о том, что есть в хранилище PEAR, я приведу примеры того, что можно сделать с помощью пакетов PEAR.

- Кешировать вывод с помощью `Cache_Lite`.
- Тестировать эффективность кода с помощью `Benchmark`.
- Скрывать детали доступа к базе данных с помощью `MDB2`.
- Управлять файлами `.htaccess` сервера Apache с помощью `File_HtAccess`.
- Выделять или кодировать каналы новостей с помощью `XML_RSS`.
- Отправлять почту с вложениями с помощью `Mail_Mime`.
- Выполнять анализ файлов конфигурации разных форматов с помощью `Config`.
- Создавать защищенные паролем среды с помощью `Auth`.

На веб-сайте PEAR представлен список пакетов, разбитых на категории по темам. Вы можете найти здесь пакеты, позволяющие решить ваши проблемы, или воспользоваться помощью поисковых систем. В любом случае всегда стоит уделить время оценке существующих пакетов, прежде чем браться, возможно, за изобретение велосипеда.

Но тот факт, что у вас есть задача и существует пакет для ее решения, не должен быть началом и концом анализа, который вы проводите. Хотя предпочтительно использовать пакет, если он избавит вас от лишней работы по разработке, в некоторых случаях это может означать издержки без получения реальных преимуществ. Например, если клиенту нужно приложение для отправки почты, это не означает, что вы должны автоматически использовать пакет `Mail` из PEAR. В PHP предусмотрена отличная функция `mail()`, поэтому лучше всего начать с нее. А как только вы поймете, что нужно проверять правильность всех адресов электронной почты согласно стандарту RFC822 и отправлять по электронной почте вложенные изображения, начните анализировать другие возможности. Оказывается, в PEAR существуют пакеты для решения обеих этих задач.

Многие программисты, включая меня, часто уделяют слишком много внимания созданию оригинального кода, и иногда в ущерб своим проектам. В основе этого стремления к авторскому коду часто лежит желание создавать, а не использовать код многократного применения.

Реальные программисты смотрят на оригинальный код как на один из инструментов, помогающих создавать эффективные проекты. Такие программисты анализируют имеющиеся в наличии ресурсы и умело их применяют. Если существует пакет, который может взять на себя выполнение некоторой задачи, это то, что надо. Перефразируем известный афоризм из среды Perl — "хорошие программисты ленивы".

Хорошая игра

Истинность знаменитого изречения Сартра "Ад — это другие люди" ежедневно доказывают некоторые программные проекты. Это может служить описанием от-

ношений между клиентами и разработчиками, когда отсутствие нормальной коммуникации ведет к утраченным возможностям и искаженным приоритетам. Но это подходит также 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 второго программиста, как эта стратегия мгновенно рассыпается. Если второй программист работает в том же каталоге, то существует вероятность, что при сохранении он затрет работу первого, если оба не будут достаточно аккуратными и не будут всегда работать с различными версиями файлов.

Возможен и другой вариант. Оба программиста получают версию кодовой базы, чтобы работать отдельно. И все идет хорошо, пока не настает время согласовать эти две версии. Если программисты не работали над совершенно разными наборами файлов, то окажется, что задача объединения двух или более веток разработки — это страшная проблема.

Вот здесь и пригодятся Git, Subversion и аналогичные инструменты, 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 будете довольны результатом. Затем вы можете обновить свою версию в соответствии со всеми изменениями, которые внесли коллеги. Система контроля версий автоматически встроит эти изменения в ваши файлы, уведомляя вас о любых противоречиях, которые она не сможет разрешить. Протестировав этот новый гибрид, вы сможете 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, так что можно вернуться к любому моменту развития проекта или получить соответствующую копию кода. Вы можете также создавать различные ветки и поддерживать общедоступную версию одновременно с разрабатываемой.

После того как вы испытаете возможности контроля версий в проекте, вы уже не захотите делать другой проект без этого. Одновременно работать с несколькими ветками проекта нелегко, особенно вначале, но преимущества вскоре станут очевидными. Контроль версий слишком полезен, чтобы обходиться без него. Подробнее о Git я расскажу в главе 17.

Как дать коду крылья

Вы когда-нибудь сталкивались с тем, что ваш код не используется, потому что его слишком трудно инсталлировать? Это особенно верно для проектов, которые разрабатываются на месте. Такие проекты обосновываются в своем контексте, с паролями и каталогами, так что работа с базами данных и вызовы вспомогательных приложений программируются прямо в коде. Внедрение такого проекта — дело сложное. Командам программистов придется прочесывать исходный код и корректировать параметры настройки, чтобы они соответствовали новой среде.

Эту проблему можно решить до некоторой степени, предусмотрев централизованный файл конфигурации или класс, чтобы все настройки можно было изменить в одном месте. Но даже в этом случае инсталляция будет нелегкой. От трудности или простоты инсталляции будет в значительной степени зависеть популярность

любого распространяемого приложения. Это также будет мешать либо способствовать количеству и частоте развертывания приложения во время разработки.

Как и любую повторяющуюся или отнимающую время задачу, инсталляцию следует автоматизировать. Программа установки, помимо выполнения прочих задач, должна определить стандартные значения для мест инсталляции, проверить и изменить права доступа, создать базы данных и инициализировать переменные. По сути, программа установки может сделать практически все, что нужно для копирования приложения из исходного каталога в установочный каталог, для полного развертывания.

Конечно, это не избавляет пользователя от обязанности добавить к коду информацию о среде, но это упрощает процесс настолько, что он сводится к ответу на несколько вопросов или указанию пары ключей в командной строке.

С точки зрения разработчиков, у программы установки есть еще одно достоинство — память. После того как программа была запущена из установочного каталога, она может запоминать многие параметры, что еще больше упрощает последующие инсталляции. Поэтому когда будете выполнять инсталляцию из установочного каталога во второй раз, вам уже не нужно будет предоставлять информацию о конфигурации, такую как имена базы данных и установочные каталоги. Эти данные будут сохранены в кеш-памяти после первой инсталляции. Это важно для разработчиков, которые часто обновляют локальную среду разработки с помощью системы контроля версий. Контроль версий упрощает получение последней версии проекта. Но нет смысла устранять помехи в получении кода, если есть узкое место, ограничивающее его внедрение.

Поставщики услуг облачных технологий, какие как EC2 от Amazon, позволяют по мере необходимости создавать тестовые и рабочие среды. Хорошие решения для построения и инсталляции проектов очень важны, поскольку они позволяют получить максимум отдачи от этих ресурсов. Не имеет никакого смысла создавать специальный сервер для автоматизации задач построения, если вы не в состоянии "на лету" развернуть свою систему.

Для разработчика существуют различные средства построения. Например, PEAR — это отчасти решение проблемы инсталляции. В большинстве случаев вы будете использовать инсталлятор PEAR для получения кода из официального хранилища PEAR. Но можно создать собственные пакеты PEAR, которые пользователи смогут легко загрузить и установить. Инсталлятор PEAR лучше всего подходит для самодостаточных пакетов с хорошо определенными функциональными возможностями. Существуют довольно строгие правила относительно ролей и мест инсталляции файлов, которые должен содержать пакет, и обычно нужно придерживаться процесса "поместить файл А в каталог Б". Этот аспект PEAR будет подробно рассмотрен в главе 15.

Если вам нужна большая степень гибкости, например, для инсталляции приложения, то выберите более гибкий и расширяемый инсталлятор. В главе 19 мы рассмотрим приложение под названием Phing. Этот проект с открытым исходным кодом является портированной версией популярного средства построения Ant, который написан на Java и для платформы Java. Phing написан на PHP и для платформы PHP, но по архитектуре он аналогичен Ant и использует такой же XML-формат для своих файлов построения.

В то время как PEAR отлично выполняет некоторые задачи и имеет очень простую систему конфигурирования, Phing сначала кажется более устрашающим, но, освоив его, вы получите невероятную степень гибкости. Вы можете не только использовать Phing для автоматизации чего угодно, от копирования файлов до преобразования XSLT, но и легко писать и включать собственные задачи, если вам пона-

добится расширить данное средство. Phing написан с помощью объектно-ориентированных возможностей PHP 5, и в его проекте делается акцент на модульности и простоте расширения.

Сказанное выше вовсе не означает, что PEAR и Phing нельзя использовать вместе. Часто я использую Phing для управления процессом построения приложения и тестирования кода. Утилита Phing может инсталлировать код самостоятельно, однако системные администраторы, вероятнее всего, предпочтут получить приложение из хранилища PEAR, чем связываться с Phing. Именно по этой причине я конфигурирую Phing так, чтобы с его помощью можно было генерировать пакеты PEAR для окончательной версии дистрибутива моего приложения.

Документирование

Мой код такой небольшой и изящный, что не нуждается в документировании. Его назначение становится абсолютно ясным при первом же взгляде. Я знаю, что ваш код такой же. Но у всех остальных есть проблема.

Ладно, оставим иронию. Это правда, что хороший код в некоторой степени сам документирует себя. Определив понятный интерфейс и четко заданные обязанности для каждого класса и метода и предоставив им всем содержательные имена, вы тем самым описываете назначение кода. Но вы можете сделать проект еще боле понятным и избежать лишних неясностей: ясность побеждает ум, если только ум не приносит с собой огромный, и необходимый, выигрыш в эффективности.

Именование свойств, переменных и аргументов тоже играет огромную роль в том, чтобы сделать код более простым и понятным для прочтения пользователями. По возможности старайтесь выбирать содержательные имена. Я часто добавляю к имени информацию о типе переменной — особенно для переменных-аргументов.

```
public function setName( $name_str, $age_int ) {
//...
}
```

Но не имеет значения, насколько понятен ваш код. Он никогда не будет достаточно понятен сам по себе. Мы уже видели, что в объектно-ориентированном проекте часто объединяют многие классы в отношения наследования, агрегирования или того и другого. Если посмотреть на один класс в такой структуре, то обычно очень трудно сделать экстраполяцию более широкой картины, если не иметь какого-то явного указателя.

В то же время каждый программист знает, какое это мучение — писать документацию. Во время разработки вы этим обычно пренебрегаете, потому что код постоянно изменяется, а ведь вскоре наступит момент, когда будет получен правильный вариант кода. Но когда вы достигаете периода стабильности, то внезапно осознаете масштабность задачи документирования работы. Кто бы мог подумать, что вы создадите так много классов и методов? Сроки сдачи проекта уже близки, поэтому некогда заниматься ерундой, нужно сосредоточиться на гарантии качества.

Это вполне понятная, но недальновидная позиция, и вы это поймете, когда вернетесь к своему коду через год, чтобы приступить ко второму этапу. Вот что сказал один программист в популярном чате на сайте `http://www.bash.org`.

`<@Logan>`: В течение минуты я смотрел на свой код.

`<@Logan>`: И думал: "Что, черт возьми, написал этот программист?"

Без документирования вы обречены постоянно проигрывать эту пластинку: тратить время на то, чтобы во второй раз догадываться, почему были приняты те или иные решения (которые вы наверняка приняли по какой-то серьезной причине).

Если, конечно, вы вообще знаете, в чем они заключались. Это очень плохо, но ситуация еще ухудшится, если вам нужно будет передать свою работу коллеге. У него уйдет много времени на то, чтобы разобраться в коде. Он будет вынужден усеять код отладочными сообщениями и продираться сквозь толстые распечатки беспорядочно взаимосвязанных классов.

Так что правильное решение напрашивается само собой: код обязательно нужно документировать и делать это нужно во время кодирования. Но нельзя ли упростить этот процесс? Как вы, наверное, догадались, ответом будет "да", и снова решение заимствовано у утилиты Java. Средство phpDocumentor (`http://www.phpdoc.org/`) — это реинкарнация JavaDoc, приложения для создания документации, которое входит в поставку Java SDK. Для программиста принцип прост. Добавьте специальным образом отформатированные комментарии перед всеми классами, большинством методов и некоторыми свойствами, и phpDocumentor включит их в систему документов с гиперссылками. Даже если вы опустите комментарии, это приложение прочитает код и соберет информацию о классах, которые найдет, и об отношениях между ними. Это большое преимущество само по себе, так как вы сможете переходить от одного класса к другому (щелкать на именах классов) и сразу видеть отношения наследования.

Средство phpDocumentor будет рассмотрено в главе 16.

Тестирование

При создании класса вы вполне уверены, что он работает. В конце концов, вы проверите его в процессе разработки. Вы 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 так, как вы ожидаете? На первый взгляд, может показаться, что это глупый вопрос. В конце концов, вы проверяли код один раз; почему же ни с того, ни с сего он должен перестать работать? Конечно, ничто не происходит просто так, и если вы не добавите ни одной строки к коду, то можете спать спокойно. Но если проект активно развивается, то контекст компонента будет неизбежно меняться, и весьма вероятно, что и сам компонент будет изменен, причем разными способами.

Давайте рассмотрим эти проблемы по очереди. Итак, как изменение контекста компонента может привести к ошибкам? Даже в системе, где компоненты аккуратно отделены один от другого, они все равно остаются взаимозависимыми. Объекты, используемые классом, возвращают значения, выполняют действия и принимают данные. Если любой из этих типов поведения меняется, то результат работы класса может привести к тому 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 тот тип изменений, который не приводит к ошибке на уровне интерпретатора, но, тем не менее, "сбивает с толку" компонент. Если класс делает предположение на основании данных другого класса, то изменение этих данных может привести к принятию неправильного решения. В результате класс делает ошибку, в то время как в нем не изменилось ни одной строки кода.

И вполне вероятно, что вы продолжите изменять только что созданный класс. В большинстве случаев эти изменения будут незначительными и очевидными. Настолько незначительными, что вы не почувствуете необходимости проводить тщательное тестирование, как во время разработки. Вы наверняка даже забудете о них, если только не сделаете комментариев внизу файла класса (как поступаю я ино-

гда). Но небольшие изменения могут привести к серьезным непредвиденным последствиям, которые можно обнаружить, если использовать средства тестирования.

Средства тестирования — это набор автоматических тестов, которые можно применить к системе в целом или к ее отдельным классам. Правильно примененные, средства тестирования помогают не допустить появления и *повторения* ошибок. Одно-единственное изменение может привести к каскаду ошибок, а средства тестирования позволят это обнаружить и устранить. Это означает, что вы можете вносить изменения с определенной уверенностью в том, что ничего не нарушите. Это впечатляет — внести изменение в систему, а затем увидеть список неудачно завершившихся тестов. Все эти ошибки могли распространиться по системе, но теперь она не пострадает.

Непрерывная интеграция

Приходилось ли вам когда-нибудь составлять план, в котором все пункты выполнялись бы замечательно? Вы начинали с задания — это мог быть программистский проект или домашнее задание в школе. Оно было большим и ужасным, обреченным на провал. Но вы брали лист бумаги и делили задание на поддающиеся выполнению фрагменты. Вы определяли, какие книги нужно прочесть и какие детали записать. Возможно, вы выделяли какие-то задания разными цветами. Взятые по отдельности они не были такими ужасными, и вы их могли выполнить. И так постепенно, как вы и планировали, наставал конечный срок сдачи работы. Если вы выполняли что-то понемногу и каждый день, все было прекрасно. Вам можно было расслабиться в конце.

Хотя иногда для выполнения вашего плана требовались просто титанические усилия. Вам приходилось в сомнениях постоянно откладывать его в сторону и преодолевать страх перед тем, что именно сегодня все вот так возьмет и рухнет. И только по прошествии нескольких недель после реализации вашего плана в нем уже не было никакой магии. Фактически вы должны были выполнить свою работу. Конечно, к тому времени страсти вокруг задания уже стихали, и вы могли не обращать на него внимания. И вам ничего другого не оставалось, как создать новый план. И страсти вновь начинали накаляться.

По сути, тестирование и построение проекта напоминает описанное выше. Вы должны запускать тесты. Вы должны выполнить построение проектов, а затем снова и снова перестраивать их в новых условиях, иначе не будет никакой магии!

И если написание тестов — это сплошное мучение, то их запуск — неинтересная и рутинная работа. Особенно в случаях, когда они очень сложные и их непрохождение нарушает ваши планы. Конечно, если вы будете запускать тесты чаще, вероятно, у вас будет меньше проблем и все они будут относиться только к недавно написанному коду.

Очень легко достичь комфорта в своей песочнице, когда все "игрушки" у вас под руками. Это — небольшие скрипты, облегчающие жизнь, средства разработки и полезные библиотеки. Однако на самом же деле неприятность как раз и состоит в том, что ваш проект очень комфортно себя чувствует в своей "песочнице". В нем могут использоваться незавершенные фрагменты кода или зависимости, которые вы забыли включить в файл построения. Это означает, что построение проекта завершится неудачей на любом другом компьютере, кроме вашего рабочего.

Единственный выход заключается в том, что процесс построения нужно тщательно и многократно проверять, причем каждый раз это нужно делать на сравнительно "чистой" машине.

Разумеется, советовать такое — хорошо! И совсем другое дело — взять и сделать! По своей природе программистам нравится сам процесс написания кода. Они всегда стремятся свести все организационные вопросы к минимуму. И вот здесь им на помощь приходит *непрерывная интеграция (НИ)* (Continuous Integration, или CI). Она представляет собой одновременно и методику, и набор инструментальных средств, облегчающих, насколько это возможно, внедрение в жизнь этой методики. В идеале, процессы построения и тестирования должны быть полностью автоматизированы. На худой конец они должны запускаться после ввода одной команды или щелчка кнопкой мыши. При этом любая проблема будет зафиксирована, и вы будете о ней уведомлены прежде, чем произойдет что-то серьезное. Подробнее о непрерывной интеграции поговорим в главе 20.

На заметку. Настоятельно рекомендуем самостоятельно ознакомиться с возможностями средств проведения экспертной оценки, такими как Review Board (`http://www.reviewboard.org`) и Gerrit (`http://code.google.com/p/gerrit/`). Они позволяют команде разработчиков рецензировать, утверждать и отклонять код, предназначенный для помещения в хранилище системы контроля версий.

Резюме

Цель разработчика всегда одна — создать работающую систему. Написание хорошего кода — необходимое, но недостаточное условие для достижения этой цели.

В данной главе я рассказал о PEAR (о котором будет также говориться в следующей главе). Мы обсудили два важных момента, без которых трудно представить себе совместную работу программистов над проектом: документирование и контроль версий. Мы выяснили, что для контроля версий требуется автоматическое средство построения, такое как Phing, PHP-реализация программы Ant, средства построения на языке Java. В конце главы мы обсудили вопросы тестирования программ. При этом я ввел новое понятие непрерывной интеграции как средства, автоматизирующего процесс построения и тестирования программ.

Глава 15

Введение в PEAR и Pyrus

Программисты стремятся создавать код, пригодный для повторного использования. Это одна из главных целей объектно-ориентированного программирования. Нам нравится извлекать полезные функциональные возможности из конкретного контекста и превращать их в инструмент, который можно использовать снова и снова. Но, с другой стороны, хотя программисты и любят повторно используемый код, они ненавидят дублирование. Создавая библиотеки, которые можно применять снова и снова, программисты избавляют себя от необходимости реализовывать аналогичные решения для разных проектов.

Но даже если мы избегаем дублирования в собственном коде, существует более общая проблема. Если взять каждый создаваемый вами инструмент, то сколько других программистов реализовали такое же решение? В большом масштабе получаем зря потраченные усилия. Поэтому не разумнее ли программистам сотрудничать и сосредоточить усилия на том, чтобы создать один качественный инструмент, вместо того чтобы производить сотни вариаций на одну и ту же тему? Вот здесь нам и пригодится PEAR (PHP Extension and Application Repository).

PEAR — это хранилище PHP-пакетов с контролируемым качеством, которые расширяют функциональные возможности PHP. Это также клиент-серверный механизм для распространения и инсталляции пакетов, а также управления зависимостями между пакетами.

В этой главе будут рассмотрены следующие темы.

- *Основы PEAR:* что это еще за странный фрукт?
- *Инсталляция пакетов PEAR:* это делается с помощью одной команды!
- *Работа с Pyrus:* младшим братом PEAR.
- *Добавление пакетов PEAR к вашим проектам:* пример и некоторые замечания по поводу обработки ошибок.
- `package.xml`: структура файла построения.
- *Создание собственного канала:* обеспечение прозрачного управления зависимостями и загрузкой пакетов для пользователей.

Что такое PEAR

Основу PEAR составляет набор пакетов, организованных по крупным категориям, таким как сети, почта и XML. Хранилище PEAR имеет централизованное управление, поэтому, когда вы используете официальный пакет PEAR, можете быть уверены в его качестве.

Список имеющихся пакетов можно просмотреть по адресу `http://pear.php.net`. Прежде чем создавать инструмент для проекта, возьмите себе за правило проверять сайт PEAR, чтобы выяснить, не сделал ли уже это кто-то до вас.

Поддержка PEAR встроена в PHP (по крайней мере, на момент написания данной книги). Это означает, что некоторые из основных пакетов сразу будут доступными в вашей системе (если только PHP не был скомпилирован без поддержки PEAR с помощью опции программы конфигурации `--without-pear`). Пакеты инсталлируются в заданный каталог (в системах Unix или Linux это, как правило, `/usr/local/share/pear`). Вы можете проверить это с помощью приложения командной строки `pear`.

```
$ pear config-get php_dir
/usr/local/share/pear
```

Ключевые пакеты (которые называются PEAR Foundation Classes) создают основу для более широкого хранилища — включая такие базовые функции, как обработка ошибок и аргументов командной строки.

На заметку. При использовании дистрибутивных средств Unix для установки PHP вы можете выбрать минимальную инсталляцию. Например, чтобы установить PHP и PEAR в Fedora 18, введите приведенные ниже команды.

```
sudo yum install php
sudo yum install php-pear
```

Если вы предпочитаете для управления инсталляцией PHP пользоваться средствами установки пакетов системы Unix, обратитесь к соответствующей документации.

Мы уже видели приложение `pear` в действии. Это инструмент для взаимодействия со всеми аспектами PEAR, и как таковой он сам по себе является важной частью PEAR. Приложение `pear` поддерживает ряд подкоманд. Мы использовали подкоманду `config-get`, которая показывает значение конкретного параметра конфигурации. С помощью подкоманды `config-show` можно узнать все параметры и их значения.

```
$ pear config-show
```

```
Configuration (channel pear.php.net):
======================================
Auto-discover new Channels    auto_discover      <not set>
Default Channel               default_channel    pear.php.net
HTTP Proxy Server Address     http_proxy         <not set>
PEAR server [DEPRECATED]      master_server      pear.php.net
Default Channel Mirror        preferred_mirror   pear.php.net
Remote Configuration File     remote_config      <not set>
PEAR executables directory    bin_dir            /usr/local/bin
...
```

Знакомьтесь: Pyrus

Прежде чем двигаться дальше, я должен познакомить вас со сравнительно новым средством в мире PEAR. Pyrus — это инсталлятор пакетов, привязанный к PEAR2 (т.е. к обновленной версии хранилища пакетов PEAR). Большинство пакетов PEAR уже портировано в PEAR2. В качестве стандарта нового хранилища выбрано улучшенное тестирование приложений, использование пространств имен и усовершенствованная обработка ошибок. Инсталлятор Pyrus специально создавался как облегченная версия своего предшественника.

Название "Pyrus" происходит от названия класса деревьев и кустарников, содержащих грушевое дерево (pear tree)[1]. Совершенно случайно оказалось, что латинское название грушевого дерева (т.е. "pear tree") — "Pyrus". Загрузить приложение Pyrus можно с сайта `http://pear2.php.net`. Оно поставляется в виде пакета `phar` (PHP Archive). Если вы знакомы с языком Java, то в этом названии, конечно же, узнаете PHP-аналог файлов типа `jar`. По сути, пакет `phar` представляет собой набор файлов с php-кодом, сжатых в один zip-файл. Это замечательный способ распространения библиотечного кода. После загрузки пакета запустить Pyrus можно простой командой.

```
php pyrus.phar
```

```
Pyrus version 2.0.0a4 SHA-1: 72271D92C3AA1FA96DF9606CD538868544609A52
Pyrus: No user configuration file detected
It appears you have not used Pyrus before, welcome!  Initialize install?
...
```

Как видите, при первом запуске Pyrus это приложение запрашивает ряд параметров конфигурации. Вам нужно будет указать каталог, в котором должен храниться сам файл конфигурации, и папку, в которую будут загружаться пакеты.

Несмотря на то что Pyrus уже используется некоторое время (его первоначальная версия создавалась еще для PHP 5.3.1), PEAR продолжает оставаться стандартным средством установки пакетов главным образом из-за того, что оно входит в дистрибутивную поставку PHP. По этой причине ряд средств сторонних разработчиков, которые прекрасно работают с PEAR, могут не до конца быть совместимы с Pyrus. С другой стороны, за Pyrus будущее. И сделав переход на него сейчас, можно будет получить выгоды в долгосрочной перспективе.

Поскольку в любом случае существует ряд причин для перехода на Pyrus, в этой главе я попытаюсь выделить преимущества как PEAR, так и Pyrus.

Итак, чтобы ввести вас в курс дела, в Pyrus поддерживается команда `get`, которая является эквивалентом команды `config-show` в PEAR.

```
$ php pyrus.phar get
```

```
Pyrus version 2.0.0a4 SHA-1: 72271D92C3AA1FA96DF9606CD538868544609A52
Using PEAR installation found at /usr/local/share/pyrus
System paths:
  php_dir => /usr/local/share/pyrus/php
  ext_dir => /usr/local/lib/php/20131226-zts
  cfg_dir => /usr/local/share/pyrus/cfg
...
```

[1] Здесь автор, конечно же, шутит, пользуясь игрой английских слов. — *Примеч. ред.*

На заметку. Хотя Pyrus должен работать "прямо из коробки" со стандартной сборкой PHP, в инсталляцию PHP в некоторых версиях Linux могут быть не включены все необходимые средства для запуска Pyrus. Речь идет о расширениях PHP: `phar`, `simplexml`, `libxml2`, `spl` и `pcre`. Например, чтобы запустить Pyrus в Fedora 18, нужно установить дополнительный пакет.

```
yum install php-xml
```

В вашей версии Unix для этого могут потребоваться совсем другие команды. Также не забывайте, что для работы Pyrus требуется PHP как минимум версии 5.3.1 или более поздних версий.

Хотя и PEAR, и Pyrus поддерживают много подкоманд, вероятно, чаще всего вы будете использовать только одну из них, связанную с инсталляцией пакетов. Так, команда `pear install` используется для инсталляции пакетов в PEAR.

Инсталляция пакетов

Выбрав пакет, вы можете загрузить и инсталлировать его с помощью одной-единственной команды. Вот как выглядит процесс инсталляции `Log`, пакета, который обеспечивает расширенную поддержку для регистрации ошибок.

```
$ pear install pear/Log
```

На заметку. В большинстве случаев вам понадобятся права доступа системного администратора (`root`), чтобы инсталлировать, обновлять и удалять пакеты с помощью PEAR, поскольку этот процесс включает запись в каталоги операционной системы, которые находятся за пределами домашнего каталога пользователя. Но если у вас нет права записи в эти области, то не все потеряно. В разделе "Создание собственного пакета PEAR" я расскажу, как изменить стандартные каталоги инсталляции. Это позволит инсталлировать пакеты в пределах пространства, где вам разрешен доступ по записи.

Это действительно так просто. Инсталлятор PEAR входит в поставку PHP и находит, загружает и инсталлирует пакет `Log` от вашего имени. Вот что мы получим на выходе этой команды.

```
WARNING: "pear/DB" is deprecated in favor of "pear/MDB2"
Did not download optional dependencies: pear/DB, pear/MDB2, pear/Mail, use
--alldeps to download automatically
pear/Log can optionally use package "pear/DB" (version >= 1.3)
pear/Log can optionally use package "pear/MDB2" (version >= 2.0.0RC1)
pear/Log can optionally use package "pear/Mail"
pear/Log can optionally use PHP extension "sqlite"
downloading Log-1.12.8.tgz ...
Starting to download Log-1.12.8.tgz (46,725 bytes)
.............done: 46,725 bytes
install ok: channel://pear.php.net/Log-1.12.8
```

У пакета `Log` есть несколько необязательных зависимостей, которые вы можете спокойно проигнорировать, если вам не требуются функциональные возможности, связанные с недостающими пакетами. Обратите внимание на последнюю строку: PEAR сообщает нам, что он получил пакет `Log` от канала `pear.php.net`. Фактически при установке этого пакета я указал в командной строке `pear/Log`, а не просто `Log`. К каналам мы вскоре вернемся.

Пакет `Log` можно установить и с помощью `Pyrus`.

```
php ./pyrus.phar install pear/Log
```

```
Pyrus version 2.0.0a4 SHA-1: 72271D92C3AA1FA96DF9606CD538868544609A52
Using PEAR installation found at /usr/local/share/pyrus
Installed pear.php.net/Log-1.12.8am
Optional dependencies that will not be installed, use --optionaldeps:
==============================>] 100% (45/45 kb)
pear.php.net/DB depended on by pear.php.net/Log
pear.php.net/MDB2 depended on by pear.php.net/Log
pear.php.net/Mail depended on by pear.php.net/Log
```

Здесь и далее я буду упоминать только отличия между PEAR и Pyrus, а не приводить параллельные примеры их использования.

Если у пакета, который вы хотите инсталлировать, есть обязательные зависимости, то инсталляция завершится неудачей и будет выдано стандартное предупреждающее сообщение.

```
pear/dialekt requires package "pear/Fandango" (version >= 10.5.0)
No valid packages found
```

Вы можете либо инсталлировать требуемый пакет, прежде чем повторить попытку, либо запустить команду `pear install` с ключом `-o`.

```
pear install -o dialekt
```

Ключ `-o` гарантирует, что инсталлятор PEAR автоматически установит любые требуемые зависимости. В некоторых пакетах PEAR указаны необязательные зависимости, но они игнорируются, если задан ключ `-o`. Чтобы все зависимости устанавливались автоматически, используйте вместо этого ключ `-a`.

На заметку. По умолчанию Pyrus пытается установить основные зависимости автоматически. В нем также поддерживается ключ `-o`, который вызывает установку еще и дополнительных пакетов, помимо основных.

Хотя PEAR предназначен для того, чтобы осуществлять коммуникации с хранилищем по сети в оперативном режиме, вы увидите, что некоторые разработчики создают PEAR-совместимые пакеты для простоты инсталляции своих программ. Вам могут дать адрес архивного файла пакета в формате `tar` и `gzip`. Инсталлировать его с помощью PEAR почти так же просто, как и официальный пакет.

```
$ pear install -o http://www.example.com/dialekt-1.2.1.tgz
```

```
downloading dialekt-1.2.1.tgz ...
Starting to download dialekt-1.2.1.tgz (1,783 bytes)
....done: 1,783 bytes
install ok: channel://pear.php.net/dialekt-1.2.1
```

Вы можете также загрузить пакет и инсталлировать его из командной строки. Здесь мы воспользуемся командой Unix `wget`, чтобы получить пакет `dialekt` до его инсталляции из командной строки.

```
$ wget -nv http://127.0.1.2:8080/dialekt-1.2.1.tgz
```

```
20:21:40 URL:http://127.0.1.2:8080/dialekt-1.2.1.tgz [1783/1783]
-> "dialekt-1.2.1.tgz.1" [1]
```

```
$ pear install dialekt-1.2.1.tgz
```

```
install ok: channel://pear.example.com/Dialekt-1.2.1
```

Вы можете инсталлировать пакет PEAR, указав в командной строке имя XML-файла (обычно совпадающее с именем пакета, например `package.xml`), который предоставляет информацию о том, какие файлы куда нужно устанавливать.

```
$ pear install package.xml
```

```
install ok: channel://pear.example.com/Dialekt-1.2.1
```

Каналы PEAR

Впервые каналы появились в PEAR версии 1.4. Эта мощная функциональная возможность позволяет опрашивать хранилища, отличные от `pear.php.net`, на предмет обновлений и зависимостей. Это означает, что вы можете создать приложение, которое запрашивает пакеты из нескольких хранилищ. Затем PEAR может заниматься получением зависимостей от имени пользователя. До появления каналов разработчик приложения должен был проинструктировать пользователя о том, чтобы он заранее установил нужные зависимости, или включить эти пакеты в состав собственной версии приложения.

Реальным примером может служить пакет PHPUnit Себастьяна Бергмана (Sebastian Bergmann). Чтобы установить этот пакет, сначала нужно, чтобы PEAR знал о канале, в котором его можно найти.

```
$ pear channel-discover pear.phpunit.de[2]
```

```
Adding Channel "pear.phpunit.de" succeeded
Discovery of channel "pear.phpunit.de" succeeded
```

Установив связь с этим каналом с помощью подкоманды `channel-discover`, вы можете указать название пакета, который находится внутри этого канала, поставив перед его именем `phpunit/`. На самом деле `phpunit` — это просто псевдоним для канала `pear.phpunit.de`. Узнать о псевдониме канала можно, запустив команду `channel-info`.

```
$ pear channel-info pear.phpunit.de
```

```
Channel pear.phpunit.de Information:
====================================
Name and Server      pear.phpunit.de
Alias                phpunit
Summary              PHPUnit channel server
...
```

На заметку. Обратите внимание на то, что Pyrus не поддерживает подкоманду `channel-info`.

Перед установкой PHPUnit я должен подключить еще один канал PEAR, как показано ниже.

```
$ pear channel-discover pear.symfony.com
```

```
Adding Channel "pear.symfony.com" succeeded
Discovery of channel " pear.symfony.com " succeeded
```

[2] Канал **`pear.phpunit.de`** был окончательно закрыт 31 декабря 2014 года. Теперь приложение PHPUnit устанавливается с помощью PEAR, как показано ниже. — *Примеч. ред.*

```
pear install -a PHPUnit
```

Итак, теперь я могу инсталлировать PHPUnit.

```
$ pear install -a PHPUnit
```

```
WARNING: "pear/PHPUnit" is deprecated in favor of "channel://pear.phpunit.
de/PHPUnit"
downloading PHPUnit-1.3.2.tgz ...
Starting to download PHPUnit-1.3.2.tgz (20,913 bytes)
........done: 20,913 bytes
downloading PHP_Compat-1.5.0.tgz ...
Starting to download PHP_Compat-1.5.0.tgz (44,133 bytes)
...done: 44,133 bytes
install ok: channel://pear.php.net/PHPUnit-1.3.2
install ok: channel://pear.php.net/PHP_Compat-1.5.0
```

Обратите внимание на то, что я использовал ключ `-a`, который просит PEAR загрузить все зависимые пакеты.

Использование пакета PEAR

Инсталлировав пакет PEAR, вы должны иметь возможность немедленно использовать его в своих проектах. Для этого нужно указать каталог PEAR в пути `include` — тогда у вас не будет проблем с включением пакета после его установки. Давайте инсталлируем `PEAR_Config` и все зависимости, которые у него есть.

```
$ pear install -a Config
```

```
WARNING: "pear/XML_Parser" is deprecated in favor of "pear/XML_Parser2"
downloading Config-1.10.12.tgz ...
Starting to download Config-1.10.12.tgz (32,291 bytes)
.........done: 32,291 bytes
downloading XML_Parser-1.3.4.tgz ...
Starting to download XML_Parser-1.3.4.tgz (16,040 bytes)
...done: 16,040 bytes
install ok: channel://pear.php.net/Config-1.10.12
install ok: channel://pear.php.net/XML_Parser-1.3.4
```

Вот как надо включить пакет.

```
require_once("Config.php");

class MyConfig {
   private $rootObj;

   function __construct( $filename=null, $type='xml' ) {
      $this->type=$type;
      $conf = new Config();
         if ( ! is_null( $filename ) ) {
            $this->rootObj = $conf->parseConfig($filename, $type);
         } else {
            $this->rootObj = new Config_Container( 'section', 'config' );
            $conf->setroot($this->rootObj);
         }
   }

   function set( $secname, $key, $val ) {
```

```
        $section=$this->getOrCreate( $this->rootObj, $secname );
        $directive=$this->getOrCreate( $section, $key, $val );
        $directive->setContent( $val );
    }

    private function getOrCreate( Config_Container $cont, $name, $value=null ) {
        $itemtype=is_null( $value )?'section':'directive';
        if ( $child = $cont->searchPath( array($name) ) ) {
            return $child;
        }
        return $cont->createItem( $itemtype, $name, null );
    }

    function __toString() {
        return $this->rootObj->toString( $this->type );
    }
}
```

Мы начинаем с включения файла `Config.php`. Большинство пакетов PEAR работает именно так и предоставляет для включения единственную точку доступа верхнего уровня. Все последующие операторы `require` затем выдает сам пакет.

В остальной части примера происходит работа с классами, предоставленными пакетом `Config`, — `Config` и `Config_Container`. Пакет `Config` позволяет создавать файлы конфигурации различных форматов и получать к ним доступ. В нашем простом классе `MyConfig` используется пакет `Config` для работы с данными конфигурации.

Приведем простой пример использования.

```
$myconf = new MyConfig();
$myconf->set("directories", "prefs"  , "/tmp/myapp/prefs" );
$myconf->set("directories", "scratch", "/tmp/" );
$myconf->set("general"    , "version", "1.0" );
echo $myconf;
```

По умолчанию этот код генерирует вывод в XML-формате.

```
<config>
    <directories>
        <prefs>/tmp/myapp/prefs</prefs>
        <scratch>/tmp/</scratch>
    </directories>
    <general>
        <version>1.0</version>
    </general>
</config>
```

Как часто бывает с кодами примеров, этот класс неполный — для него еще требуется реализовать дополнительную обработку ошибок, а также методы записи данных конфигурации в файл. Тем не менее этот код уже очень полезен благодаря функциональности пакета PEAR. Передавая различные строки типов пакету `Config`, мы можем преобразовать предыдущий вывод в различные форматы конфигурации (например, такой, как формат `.INI`, который используется в самой программе PHP). Конечно, подробное описание пакета `Config` выходит за рамки данной главы. Но для официальных пакетов PEAR инструкции по API можно найти на веб-сайте по адресу `http://pear.php.net/`. В любом случае можно рассчитывать, что вы сможете

добавить функциональные возможности пакета PEAR к сценарию с минимальными усилиями. В пакете должен быть предусмотрен понятный и хорошо задокументированный API.

На заметку. Неприятный момент, связанный с пакетами PEAR, состоит в том, что отчаянные попытки разработчиков поддержать устаревшие версии PHP привели к тому, что эти пакеты стали сильно привязанными к этим старым версиям PHP. Как и многие пакеты PEAR, в пакете `Config` используются устаревшие и не рекомендованные к использованию средства языка, от которых не так-то просто избавиться, учитывая стремление к обратной совместимости. Чтобы отключить вывод предупреждающих сообщений интерпретатора PHP по этому поводу, задайте в файле `php.ini` директиву `error_reporting` так, как показано ниже.

```
error_reporting = E_ALL & ~E_DEPRECATED
```

Обработка ошибок PEAR

Практически во всех официальных пакетах PEAR используется стандартный класс для обработки ошибок `PEAR_Error`. Он часто возвращается вместо ожидаемого значения, если что-то идет не так. Это поведение должно быть задокументировано, и вы можете проверить возвращенные значения с помощью статического метода `PEAR::isError()`.

```
$this->rootObj = @$conf->parseConfig($filename, $type);
if ( PEAR::isError( $this->rootObj ) ) {
   print "Сообщение: ". $this->rootObj->getMessage() ."\n";
   print "Код: ". $this->rootObj->getCode() ."\n\n";
   print "Трассировка:\n";

   foreach ( $this->rootObj->getBacktrace() as $caller ) {
      print $caller['class'].$caller['type'];
      print $caller['function']."() ";
      print "строка ".$caller['line']."\n";
   }
   die;
}
```

Здесь мы проверяем значение, возвращенное методом `Config::parseConfig()`.

```
PEAR::isError( $this->rootObj )
```

Этот вызов функционально эквивалентен следующему.

```
$this->rootObj instanceof PEAR_Error
```

Итак, внутри нашего блока условных операторов мы знаем, что `$this->rootObj` — это `PEAR_Error`, а не объект `Config_Container`.

Убедившись в том, что у нас есть объект `PEAR_Error`, мы можем запросить у него больше информации об ошибке. В нашем примере есть три самых полезных метода: `getMessage()` возвращает сообщение, которое описывает ошибку; `getCode()` — целое значение, соответствующее типу ошибки (это произвольное число, которое автор пакета объявит константой и, будем надеяться, задокументирует); и наконец `getBacktrace()` возвращает массив методов и классов, которые ведут к ошибке. Это позволяет пройти по сценарию в обратном направлении и найти источник проблемы, который стал причиной ошибки. Как видите, метод `getBacktrace()` возвращает массив, описывающий метод или функцию, которые приводят к ошибке. Эти элементы описаны в табл. 15.1.

Таблица 15.1. Поля, предусмотренные в методе `PEAR_Error::getBacktrace()`

Поле	Описание
`file`	Полный путь к PHP-файлу
`args`	Аргументы, передаваемые методу или функции
`class`	Имя класса (если в контексте класса)
`function`	Имя функции или метода
`type`	Если в контексте класса, природа вызова метода (`::` или `->`)
`line`	Номер строки

То, что объект типа `PEAR_Error` изменял возвращаемое значение метода, было неприятным фактом до появления PHP 5. Поэтому неудивительно, что, когда жизнь PHP 4 подходила к концу, `PEAR_Error` уже не рекомендовали для использования.

Хотя во многих пакетах `PEAR_Error` продолжает использоваться и, вероятно, еще будет использоваться некоторое время, все более широко применяется класс `PEAR_Exception`. Если бы вы вместо пакета `Config` использовали, например, пакет `XML_Feed_Parser`, то вместо проверки типов возвращаемых значений вам нужно было бы перехватывать обработку исключений.

```
$source="notthere";
try {
     $myfeed = new XML_Feed_Parser( $source );
} catch ( XML_Feed_Parser_Exception $e ) {
     print "Сообщение: "        . $e->getMessage()        . "\n";
     print "Код: "              . $e->getCode()           . "\n";
     print "Класс с ошибкой: " . $e->getErrorClass()     . "\n";
     print "Метод с ошибкой: " . $e->getErrorMethod()    . "\n";
     print "Трассировка: "      . $e->getTraceAsString() . "\n";
     print "Данные с ошибкой: ";
     print_r(                    $e->getErrorData() );
}
```

Обычно в пакете PEAR создается класс, расширяющий `PEAR_Exception`. Частично это делается с целью добавления необходимых функциональных возможностей, но главным образом для того, чтобы можно было использовать оператор `catch` для определения типов исключений и их соответствующей обработки. Класс `PEAR_Exception`, конечно, сам расширяет класс `Exception`, так что вы можете пользоваться стандартными методами, которые были описаны в главе 4. При этом вы также получаете преимущества от некоторых дополнений. Например, методы `getErrorClass()` и `getErrorMethod()` сообщают вам класс и метод, в которых произошла ошибка. Метод `getErrorData()` может поместить дополнительную информацию об ошибке в ассоциативный массив, хотя реализовать это должны расширяющие классы. Перед генерацией исключения объект `PEAR_Exception` может быть инициализирован другим исключением или массивом объектов `Exception`. В этом отношении пакеты PEAR могут служить оболочками для объектов типа `Exception`. При обработке исключения для определения причины ошибки можно вызвать метод `PEAR::getCause()`. Он вернет либо объект-оболочку для `Exception`, либо массив, если объектов несколько, либо значение `null`, если ничего не найдено.

В `PEAR_Exception` также используется шаблон Observer, позволяющий регистрировать функции или методы обратного вызова, которые будут вызваны всякий раз при выдаче исключения. Сначала давайте создадим какую-то ошибочную ситуацию.

```
class MyPearException extends PEAR_Exception { }

class MyFeedThing {
   function acquire( $source ) {
      try {
         $myfeed = @new XML_Feed_Parser( $source );
         return $myfeed;
      } catch ( XML_Feed_Parser_Exception $e ) {
         throw new MyPearException( "Запрос на подключение отвергнут", $e );
      }
   }
}
```

Здесь я расширяю класс `PEAR_Exception` и создаю простой класс, который служит оболочкой для `XML_Feed_Parser`. Если конструктор `XML_Feed_Parser` выдает исключение, я обнаруживаю его и передаю конструктору `MyPearException`, а затем генерирую это исключение снова. Этот прием позволяет мне генерировать собственную ошибку, в то же время связывая исходную причину.

Ниже приведены клиентский класс и несколько строк кода для его вызова.

```
class MyFeedClient {
   function __construct() {
      PEAR_Exception::addObserver( array( $this, "notifyError") );
   }

   function process() {
      try {
        $feedt = new MyFeedThing();
        $parser = $feedt->acquire('wrong.xml');
      } catch ( Exception $e ) {
        print "Возникла ошибка. Подробности см. в журнале \n";
      }
   }

   function notifyError( PEAR_Exception $e ) {
      print get_class( $e )  . ":";
      print $e->getMessage() . "\n";
      $cause = $e->getCause();
      if ( is_object( $cause ) ) {
         print "[Причина] " . get_class( $cause ) . ":";
         print $cause->getMessage() . "\n";
      } else if ( is_array( $cause ) ) {
         foreach( $cause as $sub_e ) {
            print "[Причина] " . get_class( $sub_e ) . ":";
            print $sub_e->getMessage() . "\n";
         }
      }
      print "----------------------\n";
   }
}

$client = new MyFeedClient();
$client->process();
```

Все обычные предупреждения и оговорки по поводу кодов примеров, конечно, применимы и к данному примеру. В особенности потому, что этот конкретный

пример *предназначен* для того, чтобы закончиться неудачей. Прежде всего, обратите внимание на конструктор. `PEAR_Exception::addObserver()` — это статический метод, которому передается ссылка на функцию обратного вызова, в виде либо имени функции, либо массива, содержащего ссылку на объект и имя метода. Этот метод или функция будет вызываться каждый раз при генерации исключения `PEAR_Exception`. Этот прием позволяет спроектировать `MyFeedClient` так, чтобы он регистрировал все исключения.

Метод `process()` передает имя несуществующего файла методу `MyFeedThing::acquire()`, который передает его конструктору `XML_Feed_Parser`, тем самым гарантируя возникновение ошибки. Мы обнаруживаем это неизбежное исключение и выводим небольшое сообщение. `notifyError()` — это метод обратного вызова, который я указал в конструкторе `MyFeedClient`. Обратите внимание на то, что он ожидает объект типа `PEAR_Exception`. В данном случае я просто запрашиваю объект и вывожу на печать информацию об ошибке, хотя в реальной ситуации я бы, вероятно, отправил эти данные в файл системного журнала. Обратите внимание на вызов метода `PEAR_Exception::getCause()`. Поскольку он может вернуть массив или один объект типа `Exception`, я обрабатываю оба случая. Если бы я запустил этот "игрушечный" код, вот что бы я получил.

```
XML_Feed_Parser_Exception:Invalid input: this is not valid XML
----------------------
MyPearException: Запрос на подключение отвергнут
[Причина] XML_Feed_Parser_Exception:Invalid input: this is not valid XML
----------------------
Возникла ошибка. Подробности см. в журнале
```

Наш метод-регистратор вызывается для обоих исключений, выданных в данном примере (первое выдано `XML_Feed_Parser`, второе — `MyFeedThing`). Объект `XML_Feed_Parser_Exception` во второй раз появляется в выходных данных журнала, потому что мы добавили его к объекту `MyPearException` в качестве причины.

Создание собственного пакета PEAR

Пакеты из хранилища PEAR хорошо документированы и отличаются простотой использования. Но насколько легко их создавать и как создать собственный пакет? В данном разделе мы изучим структуру пакета PEAR.

Файл `package.xml`

Файл `package.xml` — это сердце любого пакета PEAR. В нем указывается информация о пакете и определяется, где и как должны быть инсталлированы его участники, а также все зависимости. Инсталлятору PEAR нужен файл `package.xml` для получения инструкций, независимо от того, работает он с URL, локальной файловой системой или архивом в формате `tar` и `gzip`.

Независимо от того, насколько хорошо спроектирован и структурирован ваш пакет, если вы опустите файл построения, инсталляция завершится неудачей. Вот что произойдет, если вы попытаетесь инсталлировать архив, который не содержит файл `package.xml`.

```
$ pear install baddialekt.tgz
```

```
could not extract the package.xml file from "baddialekt.tgz"
Cannot initialize 'baddialekt.tgz', invalid or missing package file
```

```
Package "baddialekt.tgz" is not valid
install failed
```

Инсталлятор PEAR сначала распаковывает архив во временный каталог, а затем ищет файл `package.xml`. Поэтому он неудачно завершает работу при первом же препятствии. Если файл `package.xml` так важен, из чего же он состоит?

Элементы пакета

Файл `package.xml` должен начинаться со стандартного заголовка XML. Затем все элементы помещаются в корневой элемент `package`.

```
<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.11" version="2.0"
 xmlns="http://pear.php.net/dtd/package-2.0"
 xmlns:tasks="http://pear.php.net/dtd/tasks-1.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://pear.php.net/dtd/tasks-1.0
 http://pear.php.net/dtd/tasks-1.0.xsd
 http://pear.php.net/dtd/package-2.0
 http://pear.php.net/dtd/package-2.0.xsd">

<!-- Здесь идут дополнительные элементы -->

</package>
```

Этот пример завершится ошибкой. Инсталлятору PEAR требуется ряд элементов для работы. Чтобы начать, мы должны предоставить общую информацию.

```
<name>Dialekt</name>
<channel>pear.example.com</channel>
<summary>Пакет, предназначенный для озвучивания текста и веб-страниц
примитивным компьютерным голосом</summary>
<description>Пусть все друзья завидуют вам черной завистью, поскольку
этот простой и забавный пакет легко расширять и он доставляет
удовольствие при работе!
</description>

<!-- Здесь идут дополнительные элементы -->
```

Эти новые элементы достаточно очевидны и не требуют объяснений. Элемент `name` определяет идентификатор, с помощью которого пользователь будет ссылаться на пакет. Элемент `summary` содержит одну строку общей информации о пакете, а элемент `description` дает немного более подробные сведения. Все эти элементы являются обязательными, за исключением `channel`. Если вы не собираетесь добавлять пакет к каналу, то можете вместо него использовать элемент `uri`. Он должен содержать URI, указывающий на файл пакета.

```
<uri>http://www.example.com/projects/Dialekt-1.2.1</uri>
```

Имя файла не должно включать расширение, даже если сам файл пакета будет, вероятно, иметь расширение `.tgz`.

Далее вы должны предоставить информацию о "команде участников" вашего пакета. Необходимо включить по меньшей мере один элемент `lead`.

```
<lead>
 <name>Matt Zandstra</name>
 <user>mattz</user>
```

```
 <email>matt@getinstance.com</email>
 <active>yes</active>
</lead>
```

После этого вы можете аналогично определить других участников проекта. Но вместо `lead` вы можете использовать элементы `developer`, `contributor` или `helper`. Это обозначения, принятые в PEAR-сообществе, но они подходят также для большинства не PEAR-проектов. Элемент `user` указывает на имя пользователя, который внес свой вклад в PEAR. Большинство команд использует аналогичные идентификаторы, чтобы дать пользователям возможность входить в систему контроля версий, на сервер разработки или и туда, и туда.

Прежде чем перейти к файлам проекта, рассмотрим дополнительную информацию, которую вы должны предоставить.

```
<date>2010-02-13</date>
<time>18:01:44</time>
<version>
 <release>1.2.1</release>
 <api>1.2.1</api>
</version>
<stability>
 <release>beta</release>
 <api>beta</api>
</stability>
<license uri="http://www.php.net/license">PHP License</license>
<notes>initial work</notes>
```

Хотя в основном здесь все понятно, следует остановиться на некоторых возможностях. Из всех элементов, которые содержатся внутри элемента `version`, элемент `release` действительно имеет значение для пакета. Он используется PEAR при определении зависимостей. Если другая система заявляет, что ей требуется пакет Dialekt версии 1.0.0, а у пользователя, который осуществляет инсталляцию, на компьютере установлена только версия 0.2.1, PEAR остановит инсталляцию или попробует получить более новую версию пакета, в зависимости от режима, в котором он работает. С другой стороны, элемент `api` помогает отслеживать изменения в интерфейсе кода, которые могут повлиять на совместимость.

Элемент `stability` аналогично содержит элементы `release` и `api`. Они могут принимать одно из значений: `snapshot`, `devel`, `alpha`, `beta` или `stable`; вы должны выбрать то, которое точнее всего описывает ваш проект.

Выпуская пакет в соответствии с определенными условиями лицензирования (такими, как лицензия GPL GNU, например), вы должны добавить эту информацию к элементу `license`.

В отличие от элементов `summary` и `description`, элемент `notes` допускает переносы на новую строку в добавляемом содержимом.

Элемент `contents`

Теперь мы готовы к тому, чтобы поговорить о файлах и каталогах в пакете. Элемент `contents` определяет файлы, которые будут включены в архив пакета (который иногда называется тарболлом, потому что он заархивирован с помощью утилит `tar` и `gzip`). Чтобы описать структуру архива, используйте комбинацию элементов `dir` и `file`.

Рассмотрим упрощенный пример.

```
<contents>
  <dir name="/">
    <dir name="data">
       <file name="alig.txt" role="data" />
       <file name="dalek.txt" role="data" />
    </dir> <!-- /data -->
    <dir name="Dialekt">
       <file name="AliG.php" role="php" />
       <file name="Dalek.php" role="php" />
    </dir>
  </dir>

</contents>
```

У каждого файла пакета PEAR своя роль, которая связана со стандартным (конфигурируемым) местоположением. Распространенные роли описаны в табл. 15.2.

Таблица 15.2. Некоторые распространенные роли файлов пакета

Роль	Описание	Параметр конфигурации PEAR	Пример местоположения
php	PHP-файл	php_dir	/usr/local/share/pear
test	Файл проверки модуля	test_dir	/usr/local/share/pear/tests/<*пакет*>
script	Сценарий командной строки	bin_dir	/usr/local/bin
data	Файл ресурсов	data_dir	/usr/local/share/pear/data/<*пакет*>
doc	Файл документации	doc_dir	/usr/local/share/doc/pear/<*пакет*>

Когда происходит инсталляция, файлы ролей doc, data и test не попадают непосредственно в соответствующие им каталоги. Вместо этого в каталогах doc_dir, data_dir и test_dir создается подкаталог с именем пакета, и файлы инсталлируются в него.

В проекте PEAR все должно иметь свою роль, и у каждой роли есть место. Если у вас нет нужных прав доступа для работы со стандартными местоположениями ролей, то можете установить собственные местоположения с помощью инструмента командной строки pear.

```
$ pear config-set php_dir  ~/php/lib/
$ pear config-set data_dir ~/php/lib/data/
$ pear config-set bin_dir  ~/php/bin/
$ pear config-set doc_dir  ~/php/lib/doc/
$ pear config-set test_dir ~/php/lib/test/
```

На заметку. В системе Pyrus вместо подкоманд config-set для тех же целей используется подкоманда set.

Теперь PEAR будет использовать указанные вами каталоги, а не те, которые описаны в табл. 15.2. При этом не забудьте добавить каталог lib в свой путь include: в файле php.ini или в файле .htaccess или с помощью функции ini_set() в сценарии. Вы должны также гарантировать, что каталог bin находится в пути оболочки, чтобы можно было найти команды командной строки. Наши примеры связаны с вымышленным пакетом Dialekt. Ниже приведены каталог пакета и его файловая структура.

```
./package.xml
./data
```

```
    ./data/dialek.txt
    ./data/alig.txt
./script
    ./script/dialekt.sh
    ./script/dialekt.bat
./cli-dialekt.php
./Dialekt.php
./Dialekt
    ./Dialekt/AliG.php
    ./Dialekt/Dalek.php
```

Как видите, мы отразили некоторые стандартные роли PEAR в нашей структуре данных. Итак, мы включаем каталоги `data` и `script`. Каталог верхнего уровня содержит два PHP-файла. Они должны быть инсталлированы в каталог PEAR (по умолчанию это `/usr/local/share/pear`). Файл `Dialekt.php` предназначен для того, чтобы весь пакет можно было подключить к клиентскому коду с помощью единственного включаемого файла. У пользователя должна быть возможность легко подключить `Dialekt` с помощью команды.

```
require_once("Dialekt.php");
```

Дополнительные PHP-файлы (`Dalek.php` и `AliG.php`) сохраняются в каталоге `Dialekt`, который будет добавлен к каталогу PEAR (в них реализуются все детали процесса преобразования веб-страниц и текстовых файлов в их привлекательные версии). Они включаются в клиентский код из файла `Dialekt.php`. Так как инсталлированный пакет Dialekt будет доступным для вызова из командной строки, мы включили сценарий оболочки, который будет перемещен в каталог сценариев PEAR. Пакет Dialekt использует информацию о конфигурации, сохраненную в текстовых файлах. Они будут инсталлированы в каталог данных PEAR.

Давайте рассмотрим полный дескриптор `contents`.

```
<contents>
  <dir name="/">
  <dir name="data">
    <file name="alig.txt" role="data" />
    <file name="dalek.txt" role="data" />
  </dir> <!-- /data -->
  <dir name="Dialekt">
    <file name="AliG.php" role="php" />
    <file name="Dalek.php" role="php" />
  </dir> <!-- /Dialekt -->
  <dir name="script">
    <file name="dialekt.bat" role="script">
      <tasks:replace from="@php_dir@" to="php_dir" type="pear-config" />
      <tasks:replace from="@bin_dir@" to="bin_dir" type="pear-config" />
      <tasks:replace from="@php_bin@" to="php_bin" type="pear-config" />
    </file>
    <file name="dialekt.sh" role="script">
      <tasks:replace from="@php_dir@" to="php_dir" type="pear-config" />
      <tasks:replace from="@bin_dir@" to="bin_dir" type="pear-config" />
      <tasks:replace from="@php_bin@" to="php_bin" type="pear-config" />
    </file>
  </dir> <!-- /script -->
  <file name="cli-dialekt.php" role="php" />
  <file name="Dialekt.php" role="php">
```

```
    <tasks:replace from="@bin_dir@" to="bin_dir" type="pear-config" />
  </file>
  </dir> <!-- / -->
</contents>
```

Мы включили в этот фрагмент новый элемент. Элемент `tasks:replace` заставляет инсталлятор PEAR искать в файле строку-триггер, заданную в атрибуте `from`, заменяя ее параметром `pear-config` в атрибуте `to`. Поэтому файл `Dialekt.php`, например, может выглядеть следующим образом.

```
<?php
/*
* Используйте это из PHP-сценариев, для реализации CLI используйте
* @bin_dir@/dialekt
*/
class Dialekt {
const DIALEKT_ALIG=1;
const DIALEKT_DALEK=2;
//...
}
```

После инсталляции тот же комментарий класса должен выглядеть следующим образом.

```
/*
* Используйте это из PHP-сценариев, для реализации CLI используйте
* /home/mattz/php/bin/dialekt
*/
```

Зависимости

Хотя обычно пакеты — это автономные модули, в них часто используются другие пакеты. Любое использование другого пакета означает зависимость. Если используемый пакет не установлен на компьютере пользователя, то пакет, который его использует, не будет работать, как ожидалось.

Дескриптор `dependencies` — это необходимый элемент, и внутри него вы должны определить по меньшей мере версии PHP и инсталлятора PEAR.

```
<dependencies>
  <required>
    <php>
      <min>5.3.0</min>
    </php>
  <pearinstaller>
    <min>1.4.1</min>
  </pearinstaller>
  <!-- При необходимости поместите сюда другие зависимости -->
</required>
</dependencies>
```

И `php`, и `pearinstaller` могут содержать элементы `min`, `max` и `exclude`. Элемент `exclude` определяет версию, которая будет считаться несовместимой с пакетом, и вы можете включить столько версий, сколько вам нужно. Элемент `pearinstaller` может также содержать рекомендуемый элемент, в котором вы можете установить предпочтительный инсталлятор для пакета.

Если эти или другие зависимости внутри элемента `required` не выполнены, то по умолчанию PEAR откажется инсталлировать пакет. Пакет может зависеть от другого пакета, PHP-расширения (такого, как zlib или GD) или конкретной версии PHP. В данном случае мы настаиваем, чтобы Dialekt имел доступ к пакету Fandango версии 10.5.0 или выше.

```
<package>
  <name>Fandango</name>
  <channel>pear.example.com</channel>
  <min>10.5.0</min>
</package>
```

Обратите внимание на элемент `channel`; он определяет, где `pear` должен искать пакет, если он будет вызван с ключом `-a` (который означает, что нужно получить все зависимости). Вы должны указать элемент либо `channel`, либо `uri`. Элемент `uri` должен указывать на файл пакета.

```
<package>
  <name>Fandango</name>
  <uri>http://www.example.com/packages/fandango-10.5.0.tgz</uri>
</package>
```

Дескриптор `package` имеет те же спецификаторы зависимости, что и элемент `pearinstaller`, с добавлением `conflicts`, в котором можно указать версию, с которой этот пакет работать *не* будет.

Помимо дескриптора `package`, можно также определить `extension`, `os` или `arch`. Эти элементы зависимости приведены в табл. 15.3.

Таблица 15.3. Типы зависимости файла `package.xml`

Элемент	Описание
`php`	PHP-приложение
`package`	Пакет PEAR
`extension`	PHP-расширение (функциональная возможность, скомпилированная в PHP, например zlib или GD)
`arch`	Архитектура операционной системы и процессора
`os`	Операционная система

До сих пор мы определяли обязательные зависимости. На самом деле после `requires` можно определить дескриптор `optional`. В нем можно указать те же элементы зависимости. Когда PEAR встречает невыполненную необязательную зависимость, он выдает предупреждение, но, тем не менее, продолжит инсталляцию. Вы должны добавить зависимости к элементу `optional`, если пакет не может адекватно работать без указания предпочтительного пакета или расширения.

Если пользователь запускает команду `pear install` с ключом `-o`

```
pear install -o package.xml
```

то PEAR попытается загрузить и инсталлировать все *необходимые* зависимости (не забывайте, что ключ `-o` в системе Pyrus означает, что будут установлены необязательные зависимости). Запуск этой команды с ключом `-a` также автоматизирует загрузку зависимостей, но включит как необязательные, так и необходимые зависимости.

Настройка инсталляции с помощью phprelease

Хотя мы определили файлы в архиве пакета с помощью дескриптора contents, можно использовать дескриптор phprelease, чтобы указать имена файлов, которые должны быть инсталлированы на компьютере пользователей. Ниже приведен пример двух дескрипторов phprelease в нашем пакете.

```
<phprelease>
  <installconditions>
    <os>
      <name>unix</name>
    </os>
  </installconditions>
  <filelist>
    <install as="dialekt" name="script/dialekt.sh" />
    <install as="dalek" name="data/dalek.txt" />
    <install as="alig" name="data/alig.txt" />
    <ignore name="script/dialekt.bat" />
  </filelist>
</phprelease>

<phprelease>
  <installconditions>
    <os>
      <name>windows</name>
    </os>
  </installconditions>
  <filelist>
    <install as="dialekt" name="script/dialekt.bat" />
    <install as="dalek" name="data/dalek.txt" />
    <install as="alig" name="data/alig.txt" />
    <ignore name="script/dialekt.sh" />
  </filelist>
</phprelease>
```

Элемент installconditions можно использовать, чтобы определить элемент phprelease, который выполняется. В нем можно указать элементы-спецификаторы os, extension, arch и php. Эти элементы работают так же, как их "тезки" для зависимостей. Так же, как при указании элементов phprelease, определенных install conditions, вы можете предоставить стандартную версию, которая должна быть выполнена, если никакая другая не подошла.

Давайте сосредоточимся на unix phprelease. Элемент install указывает, что файл dialekt.sh во время инсталляции должен быть переименован в dialekt.

Мы определяем, что наши файлы данных должны быть инсталлированы без расширения .txt. Нам не нужно указывать подкаталог dialekt — он автоматически включается для файлов с ролью data. Обратите внимание на то, что элемент as элемента install также убирает данные о ведущем каталоге, которые мы задали в элементе contents для этих файлов. Это означает, что они инсталлируются как <data_dir>/dialekt/dalek и <data_dir>/dialekt/alig.

Также обратите внимание: в режиме Unix нам не нужно инсталлировать файл dialekt.bat. Об этом позаботился элемент ignore. Все в порядке, теперь наш пакет готов к локальной инсталляции.

Подготовка пакета к выпуску

Теперь, когда мы создали пакет и сгенерировали файл `package.xml`, вручную или автоматически, пришло время создать заархивированный и сжатый продукт.

Для этого нужно выполнить всего одну команду PEAR. Убедимся, что мы находимся в корневом каталоге проекта, и запустим следующую подкоманду.

```
$ pear package package.xml
Analyzing Dialekt/AliG.php
Analyzing Dialekt/Dalek.php
Analyzing cli-dialekt.php
Analyzing Dialekt.php
Package Dialekt-1.2.1.tgz done
```

В результате будет сгенерирован архив в форматах `tar` и `gzip` (включая все файлы, на которые есть ссылки, а также сам файл `package.xml`), пригодный для распространения. Вы можете сделать его доступным для непосредственной загрузки. Если между пакетами существуют зависимости, то можете сделать ссылки на URI в элементах пакета и использовать элемент `uri` вместо канала. Но если вы предлагаете пользователям много взаимозависимых пакетов, то, вероятно, вам нужно подумать о переходе на следующий уровень.

Настройка собственного канала

Зачем настраивать собственный канал? Помимо того что это просто круто, главное преимущество заключается в автоматическом управлении зависимостями PEAR и как следствие в простоте инсталляции и обновления пакетов для пользователей. Пользователю достаточно легко инсталлировать один пакет с помощью полного пути к URL файла архива. Если вы разработали многоярусную библиотечную систему, работающую от вспомогательных пакетов низкого уровня до приложений высокого уровня, то все значительно усложняется. Пользователям будет очень трудно управлять несколькими взаимозависимыми пакетами на своих компьютерах, особенно по мере развития этих пакетов.

Для того чтобы создать собственный канал и обеспечить его хостинг, в идеальном случае у вас должны быть

- права доступа `root` к компьютеру веб-сервера, на котором выполняется хостинг канала;
- права доступа администратора к веб-серверу (вероятно, Apache) и возможность создания субдоменов (например, `pear.yourserver.com`).

Если у вас нет перечисленных выше прав доступа, не стоит огорчаться, поскольку выполнить хостинг канала можно и на серверах сторонних провайдеров, таких как Github (`https://github.com/`). В любом случае, прежде чем принимать решение о хостинге канала, вначале нужно создать сам канал и наполнить его пакетами.

Определение канала с помощью Pirum

Pirum — это утилита от независимых разработчиков, предназначенная для создания простого канала PEAR и управления его работой. Загрузить ее можно с вебсайта по адресу `http://pirum.sensiolabs.org/`. Перед установкой Pirum необходимо создать корневой каталог хранилища PEAR, например `/usr/local/www/pear/htdocs`, и определить URL, по которому пользователи смогут к нему обращаться, например

http://pear.example.com/. Как и следовало ожидать, легче всего выполнить инсталляцию с помощью PEAR.

```
$ pear channel-discover pear.pirum-project.org
```

```
Adding Channel "pear.pirum-project.org" succeeded
Discovery of channel "pear.pirum-project.org" succeeded
```

```
$ pear install pirum/Pirum
```

```
downloading Pirum-1.1.5.tgz ...
Starting to download Pirum-1.1.5.tgz (15,658 bytes)
......done: 15,658 bytes
```

Для работы Pirum нужно поместить файл конфигурации под именем pirum.xml в корневой каталог каталога PEAR. В этом файле указываются имя канала, его псевдоним и URL, как показано ниже.

```
<server>
   <name>pear.example.com</name>
   <summary>Пример канала PEAR</summary>
   <alias>getinstance</alias>
   <url>http://pear.example.com/</url >
</server>
```

После создания файла конфигурации можно двигаться дальше и создать канал.

```
$ pirum build /usr/local/www/pear/htdocs
```

```
Running the build command:
   INFO   Parsing package
   INFO   Building channel.
   INFO   Building maintainers.
   INFO   Building categories.
   INFO   Building packages.
   INFO   Building composer repository.
   INFO   Building releases.
   INFO   Building index.
   INFO   Building feed.
   INFO   Updating PEAR server files.
   INFO   Command build run successfully.
```

Вот и все! Теперь у меня есть свой канал PEAR! Разумеется, в нем еще нет ни одного пакета. Однако перед добавлением пакетов сначала нужно убедиться, что канал виден с удаленного хоста.

```
$ pear channel-discover pear.example.com
```

```
INFO Updating PEAR server files.
Adding Channel "pear.example.com" succeeded
Discovery of channel "pear.example.com" succeeded
```

Как видите, только что созданный канал доступен извне. Убедиться в этом можно также с помощью обычного браузера, посетив веб-страницу, которую создал Pirum по адресу http://pear.example.com (рис. 15.1).

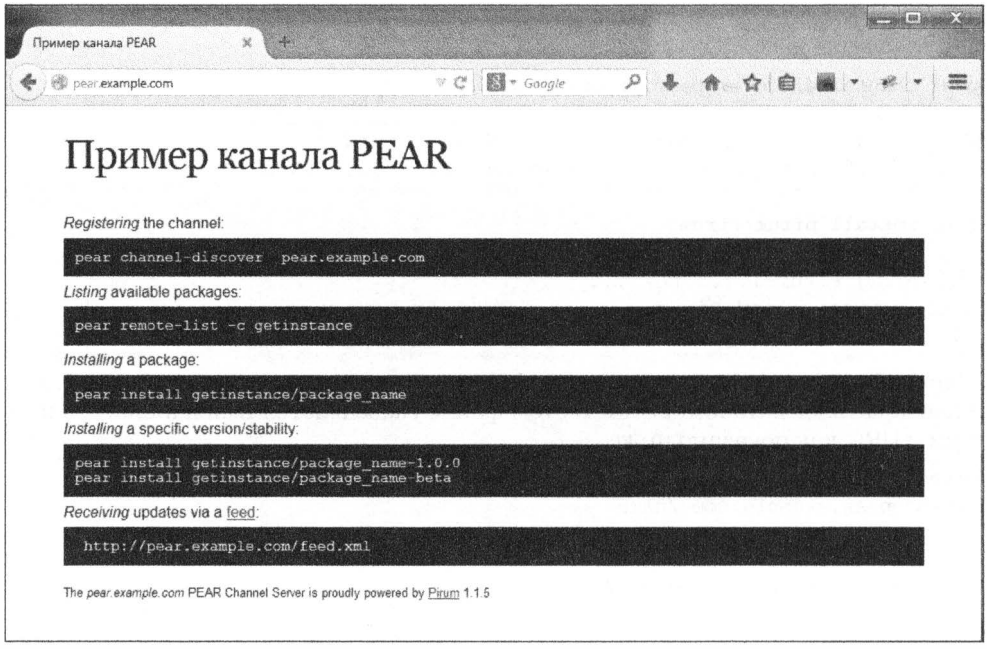


Рис. 15.1. Веб-страница канала PEAR

Размещение пакета в канале

Теперь добавим наш вымышленный пакет в только что созданное хранилище PEAR. Это делается с помощью простой команды.

```
$ pirum add /usr/local/www/pear/htdocs Dialekt-1.2.1.tgz
```

```
...
Running the add command:
INFO Parsing package Dialekt 1.2.1
INFO Building channel.
INFO Building maintainers.
INFO Building categories.
INFO Building packages.
INFO Building package Dialekt.
INFO Building composer repository.
INFO Building releases.
INFO Building releases for Dialekt.
INFO Building release 1.2.1
INFO Building index.
INFO Building feed.
INFO Updating PEAR server files.
INFO Command add run successfully.
```

Вот теперь действительно все! Я создал собственный канал PEAR, содержащий пакет. Поскольку я уже использовал подкоманду `channel-discover` для исследования канала `pear.example.com`, для установки пакета можно использовать его псевдоним `dialekt`, как показано ниже.

```
$ pear install getinstance/dialekt
```

```
downloading Dialekt-1.2.1.tgz ...
Starting to download Dialekt-1.2.1.tgz (1,780 bytes)
....done: 1,780 bytes
install ok: channel://pear.localhost/Dialekt-1.2.1
```

Чтобы воочию убедиться в том, что пакет Dialekt действительно помещен в наше хранилище, снова воспользуемся обычным браузером. В своем окне он должен показать обновленную информацию о канале (рис. 15.2).

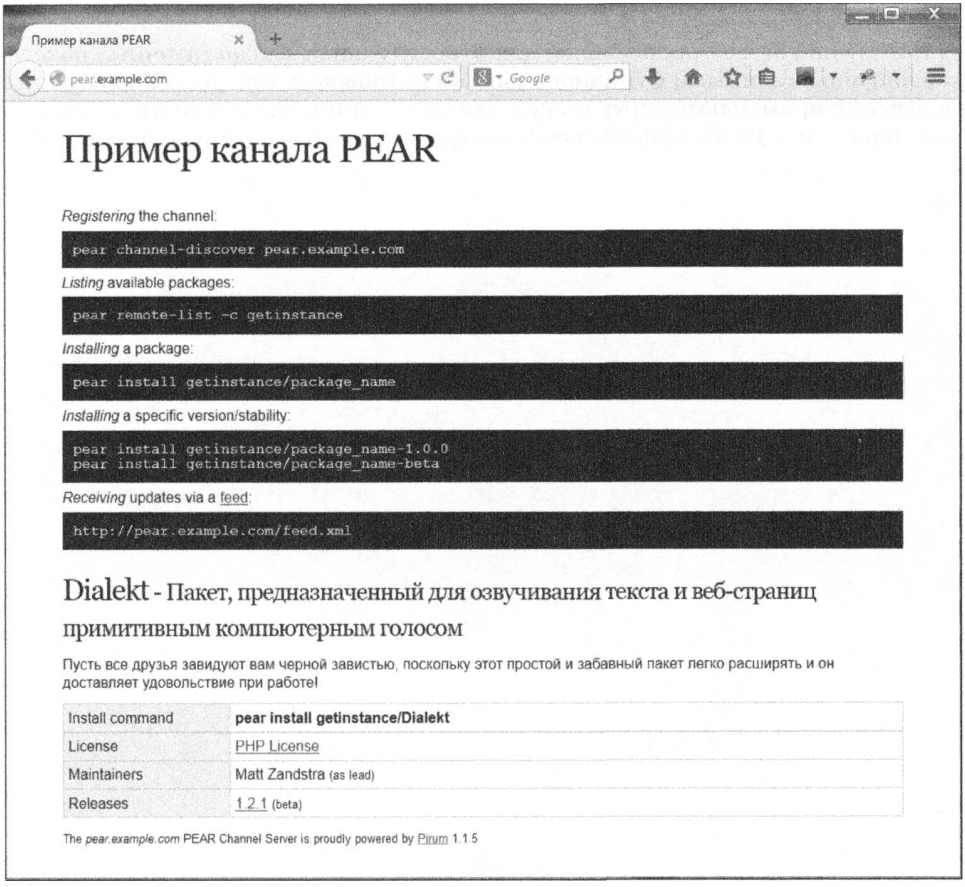


Рис. 15.2. Наш пакет действительно находится в канале

На заметку. Если вы не хотите заниматься созданием и поддержкой работы собственного канала PEAR, воспользуйтесь независимыми хранилищами пакетов, таким как OpenPear (`http://openpear.org/`).

Резюме

По определению система PEAR очень обширна, поэтому мне хватило места только для ее общего описания. Тем не менее у вас должно остаться ощущение того, насколько легко использовать пакеты PEAR, чтобы сделать проекты более функциональными. С помощью файла `package.xml` и инсталлятора PEAR (сейчас его повсеместно стараются заменить на Pyrus) можно также сделать код доступным для других пользователей. Настроив канал, вы тем самым автоматизируете загрузку зависимостей для пользователей, предоставите пакетам сторонних производителей возможность использовать ваш пакет без необходимости его встраивания в их пакеты и избежите сложного управления зависимостями.

PEAR лучше всего приспособлен для относительно самодостаточных пакетов с хорошо определенными функциональными возможностями. Для более крупных приложений применяются другие средства построения. Далее в книге мы рассмотрим Phing — мощный и эффективный инструмент для построения приложений.

Глава 16

Генерация документации с помощью phpDocumentor

Помните тот сложный фрагмент кода, в котором вы рассматривали аргумент метода как строку, или он был целым числом, а может, булевым значением? Вы узнали бы его, если бы увидели? А может, вы его уже удалили? Программирование — это сложное и трудоемкое дело, и очень трудно следить за тем, как работают системы и что нужно сделать. Ситуация еще больше усугубляется, когда возникает необходимость подключить к работе над проектом нескольких программистов. Независимо от того, хотите ли вы отметить потенциально опасные области или фантастические возможности, документация поможет вам в этом. В случае большой кодовой базы наличие или отсутствие документации означает успех или неудачу проекта.

В данной главе мы рассмотрим следующие темы.

- *Приложение phpDocumentor:* инсталляция phpDocumentor и его запуск из командной строки.
- *Синтаксис документации:* дескрипторы комментариев и документации DocBlock.
- *Документирование кода:* использование комментариев DocBlock для предоставления информации о классах, свойствах и методах.
- *Создание ссылок в документации:* ссылки на веб-сайты и другие элементы документации.

Зачем нужно что-то документировать

Программисты любят и ненавидят документацию в равной мере. Когда сроки сдачи проекта поджимают, а менеджеры или клиенты торопят вас, то документация — это, как правило, первое, от чего приходится отказаться. Все поглощает стремление получить результат. Если нужно написать красивый и изящный код (хотя для этого нужно еще чем-то пожертвовать), но кодовая база подвергается быстрым изменениям, то кажется, что документирование — это просто потеря времени. В конце концов, вам, вероятно, понадобится изменить систему классов несколько раз за несколько дней. Конечно, каждый согласится с тем, что желательно иметь хорошую документацию. Но ради этого никто не захочет снизить производительность труда.

Представьте себе очень крупный проект. Огромная кодовая база, состоящая из очень умных фрагментов кода, написанных очень грамотными людьми. Члены ко-

манды работали над одним этим проектом (или набором связанных с ним проектов) более пяти лет. Они хорошо знают друг друга и идеально знают и понимают код. Конечно, в коде очень мало комментариев. У каждого в голове есть план проекта и ряд неофициальных договоренностей в отношении правил кодирования, которые помогают понять, что происходит в каждом конкретном фрагменте кода. Но затем команда программистов увеличивается. Двум новым программистам объясняют основы сложной архитектуры, но дальше им предстоит во всем разобраться самостоятельно. Это момент, когда проявляется истинная цена недокументированного кода. В результате "акклиматизация" новых программистов длится не несколько недель, а несколько месяцев. Столкнувшись с недокументированным классом, новые программисты вынуждены отслеживать аргументы для каждого метода, разыскивать все глобальные переменные, на которые есть ссылки, проверять все методы в иерархии наследования. И каждый раз этот процесс повторяется снова и снова. Если вы, как и я, были бы таким новым членом команды программистов, то очень быстро научились бы любить документацию.

За отсутствие документации придется платить. Платить временем, которое понадобится новым членам команды, чтобы присоединиться к проекту, или старым членам команды, если они выйдут за пределы своей области специализации. Платить ошибками, если программисты попадут в ловушки, которые есть в любом проекте. Например, происходит вызов метода, который отмечен как закрытый, аргументам-переменным присваиваются данные неправильного типа, снова безо всякой необходимости создаются уже существующие функции.

Трудно выработать в себе привычку создавать документацию, потому что проблемы, вызванные пренебрежением ею, возникают не сразу. Но процесс документирования совсем не труден, особенно если вы занимаетесь этим во время кодирования. Причем этот процесс можно существенно упростить, если добавлять документацию в сам исходный код во время написания программы. Затем можно запустить специальную программу и извлечь комментарии в аккуратно отформатированные веб-страницы. Именно о таком инструменте и пойдет речь в данной главе.

В основе phpDocumentor лежит утилита, написанная на Java и называемая "JavaDoc". Обе эти программы извлекают специальные комментарии из исходного кода, создавая сложную документацию программного интерфейса приложения (API) на основе комментариев программиста и конструкций кода, найденных в исходном коде.

Инсталляция

Самый простой способ инсталлировать phpDocumentor — это воспользоваться интерфейсом командной строки PEAR.

```
$ pear channel-discover pear.phpdoc.org
$ pear install phpdoc/phpdocumentor
```

На заметку. Чтобы инсталлировать или обновить пакет PEAR в системах Unix, обычно необходимо запустить команду `pear` с правами пользователя `root`.

В результате будет выполнено подключение к серверу по адресу `http://pear.phpdoc.org` и автоматически произойдет инсталляция либо обновление phpDocumentor на вашем компьютере.

Можно также загрузить phpDocumentor в виде пакета phar (архив php) по адресу `http://phpdoc.org/phpDocumentor.phar`. После этого для запуска phpDocumentor введите приведенную ниже команду.

```
$ php phpDocumentor.phar -h
```

Генерация документации

Идея генерации документации еще до написания самой программы может показаться странной. Но phpDocumentor анализирует структуры кода в исходном коде, поэтому может собирать информацию о проекте даже до того, как мы начнем что-то писать.

Мы будем документировать некоторые аспекты воображаемого проекта под названием "megaquiz". Он состоит из двух каталогов, `command` и `quiztools`, которые содержат файлы классов. Это также имена пакетов в проекте. Пакет phpDocumentor можно запустить из командной строки или с помощью удобного графического веб-интерфейса. Мы рассмотрим метод командной строки, поскольку с его помощью проще вносить обновления в документацию, в средства построения или в сценарии командной оболочки. Команда для вызова phpDocumentor — это `phpdoc`. Для генерации документации нужно запустить ее с несколькими аргументами. Приведем пример.

```
$ phpdoc --directory=megaquiz/ \
         --target=docs/megaquiz/ \
         --title='Mega Quiz' \
         --template=abstract
```

Ключ `--directory` определяет каталог, содержимое которого вы собираетесь документировать. Вместо длинного ключа `--directory` вы можете также использовать короткий `-d` без знака равенства перед значением аргумента. Ключ `--target` (или `-t`) обозначает целевой каталог (т.е. каталог, в который вы хотите записать файлы документации). Для определения заголовка проекта используйте ключ `--title` (или `-ti`). В PhpDocumentor предусмотрены несколько встроенных шаблонов, которые определяют внешний вид и стиль созданной документации. Для указания названия шаблона используется ключ `--template`. В приведенном выше примере я выбрал шаблон **Abstract**, благодаря которому удается упаковать огромные массивы информации в довольно компактном формате. Ознакомиться с полным списком шаблонов можно по адресу `http://www.phpdoc.org/templates`.

Поскольку на момент написания этой книги самым устойчивым был стандартный шаблон **Clean**, я выбрал именно его. При этом при вызове программы PhpDocumentor ключ `--template` можно не указывать.

```
$ phpdoc --directory=megaquiz/ \
         --target=docs/megaquiz/ \
         --title='Mega Quiz'
```

Если мы запустим эту команду для нашего недокументированного проекта, то получим удивительно подробный документ. Страница меню пакета для полученного результата показана на рис. 16.1.

Как видите, все классы, интерфейсы и трейты в проекте перечислены в левом навигационном фрейме, а также на главной странице. Все имена классов представлены в виде гиперссылок. На рис. 16.2 показана часть документации для класса `Command`, который мы создали в главе 11, "Выполнение задач и представление результатов".

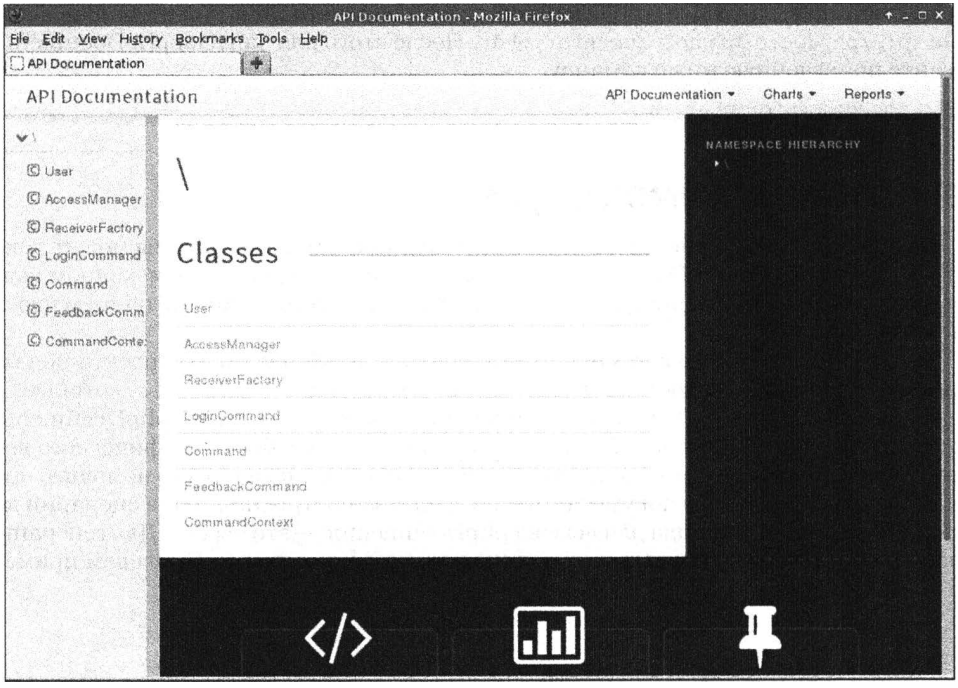


Рис. 16.1. Основное меню, полученное в результате работы phpDocumentor

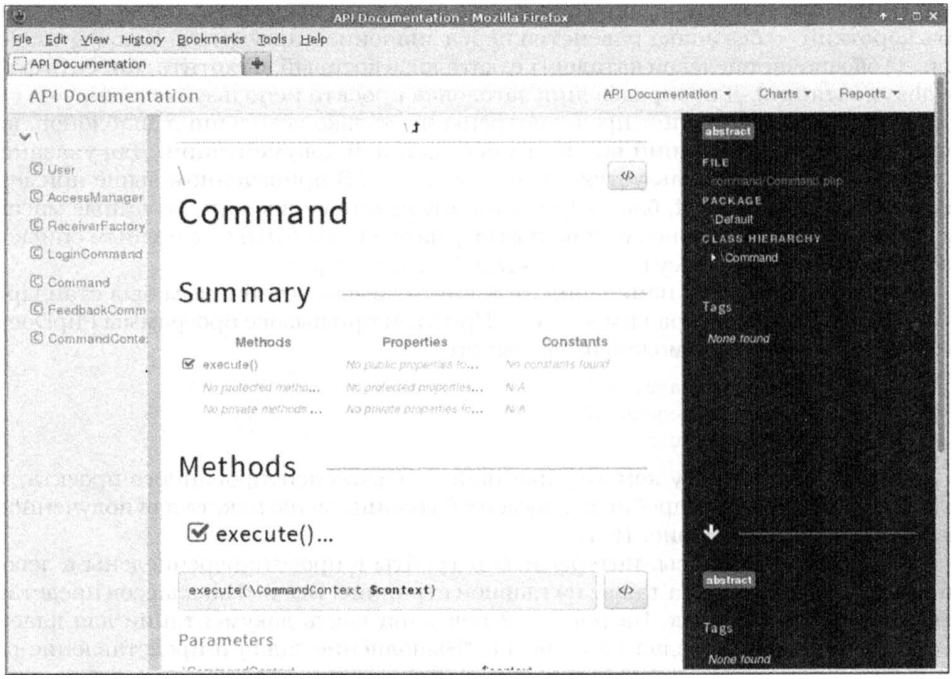


Рис. 16.2. Стандартная документация для класса `Command`

Пакет phpDocumentor достаточно умен, чтобы определить, что `Command` — это абстрактный класс. Также обратите внимание: указаны имя и тип аргумента для метода `execute()`.

Поскольку такой уровень детализации сам по себе достаточен, чтобы обеспечить общую информацию о крупном проекте с простой системой навигации, это большой шаг вперед по сравнению с тем, чтобы вообще не иметь никакой документации. Но мы можем значительно улучшить ситуацию, добавив комментарии к исходному коду.

Комментарии DocBlock

Комментарии DocBlock отформатированы особым образом, чтобы их могло распознавать приложение для создания документации. Они имеют форму стандартных многострочных комментариев с небольшим дополнением — к каждой строке внутри комментария добавлена звездочка.

```
/**
 * Мой комментарий DocBlock
 */
```

Пакет phpDocumentor распознает содержимое внутри комментариев DocBlock, если оно отформатировано должным образом. Это содержимое включает обычный текст с описанием элемента, который нужно документировать (для наших целей это — файл, класс, метод или свойство). Оно включает также специальные ключевые слова, которые называются тегами. Теги определяются с помощью символа `@` и могут быть связаны с аргументами. Поэтому приведенный ниже комментарий DocBlock, помещенный вверху класса, указывает phpDocumentor на пакет, к которому он принадлежит.

```
/**
 * @package command
 */
```

Если мы добавим этот комментарий к каждому классу в нашем проекте (конечно, с соответствующим именем пакета), то phpDocumentor упорядочит наши классы в зависимости от указанного шаблона. При выборе стандартного шаблона Clean на уровне класса будут показаны пакеты и пространства имен, но сами пакеты не включаются в навигационную панель. Тем не менее, существует шаблон Abstract, позволяющий отобразить и пространства имен, и имена пакетов на навигационной панели. Давайте сгенерируем документацию с помощью шаблона Abstract.

```
$ phpdoc --directory=megaquiz/ \
         --target=docs/megaquiz/ \
         --title='Mega Quiz' \
         --template=abstract
```

На рис. 16.3 показан вывод phpDocumentor, в который включены пакеты.

Обратите внимание: на рис. 16.3 показано, что пакеты `command`, `quizobjects` и `quiztools` помещены на навигационную панель, расположенную в правом верхнем углу.

Обычно пакеты в документации отражают структуру каталога. Поэтому пакет `command` соответствует каталогу `command`. Но это необязательно. Например, сторонний разработчик хочет создать класс `Command`, который является частью пакета `command`, но находится в собственном каталоге. Итак, с помощью тега `@package` можно связывать классы с пакетами. Этот тег также предоставляет определенную гиб-

кость, которая недоступна, когда для определения имен пакетов используется файловая система.

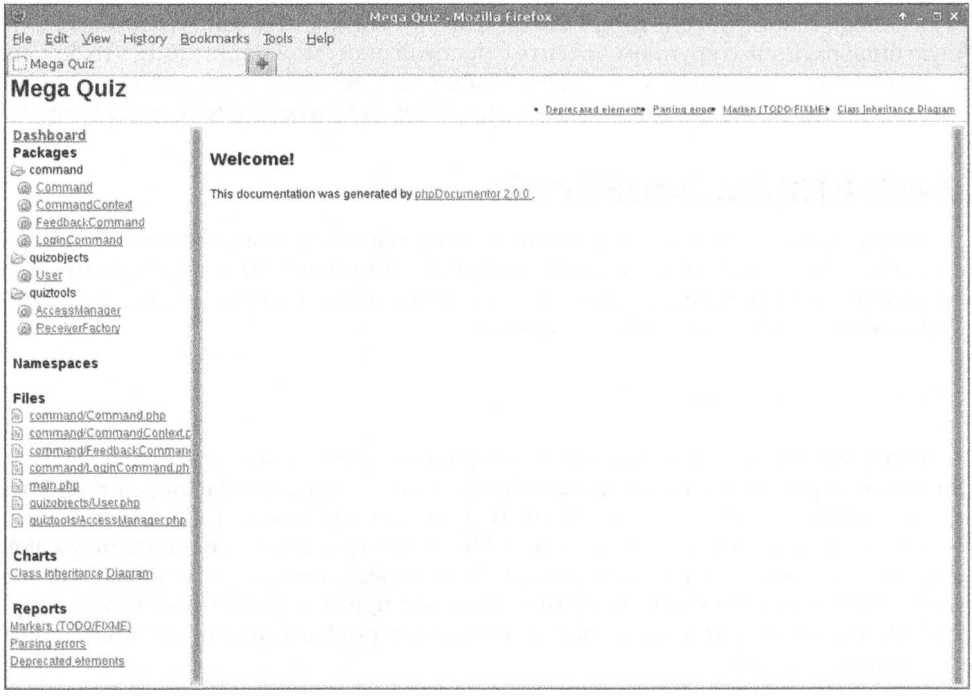


Рис. 16.3. Вывод документации, в котором определен тег `@package`

Документирование классов

Давайте добавим дополнительные теги и текст, которые полезны на уровне класса или файла в комментариях DocBlock. Мы должны определить класс, объяснить сферы его использования и добавить информацию об авторстве и авторском праве.

Вот полное описание класса `Command`.

```
/**
 * Defines core functionality for commands.
 * Command classes perform specific tasks in a system via
 * the execute() method.
 *
 * Определяет основную функциональность для команд.
 * Классы типа Command выполняют определенные задачи в системе
 * с помощью вызова метода execute().
 *
 * @package command
 * @author Clarrie Grundie
 * @copyright 2015 Ambridge Technologies Ltd
 */
abstract class Command {
   abstract function execute( CommandContext $context );
}
```

Комментарий DocBlock существенно увеличился. Первое предложение — это общая информация (резюме), состоящая из одной строки. Именно она выделяется при выводе и извлекается для использования в листингах обзоров. В последующих строках текста содержится более подробное описание. Именно здесь можно предоставить подробную информацию об использовании класса для программистов, которые придут после вас. Как мы увидим, в этом разделе, помимо текста описания, могут содержаться ссылки на другие элементы в проекте и фрагменты кода. Мы также включаем теги `@author` и `@copyright`, которые не требуют объяснений. Результат расширенного комментария класса показан на рис. 16.4.

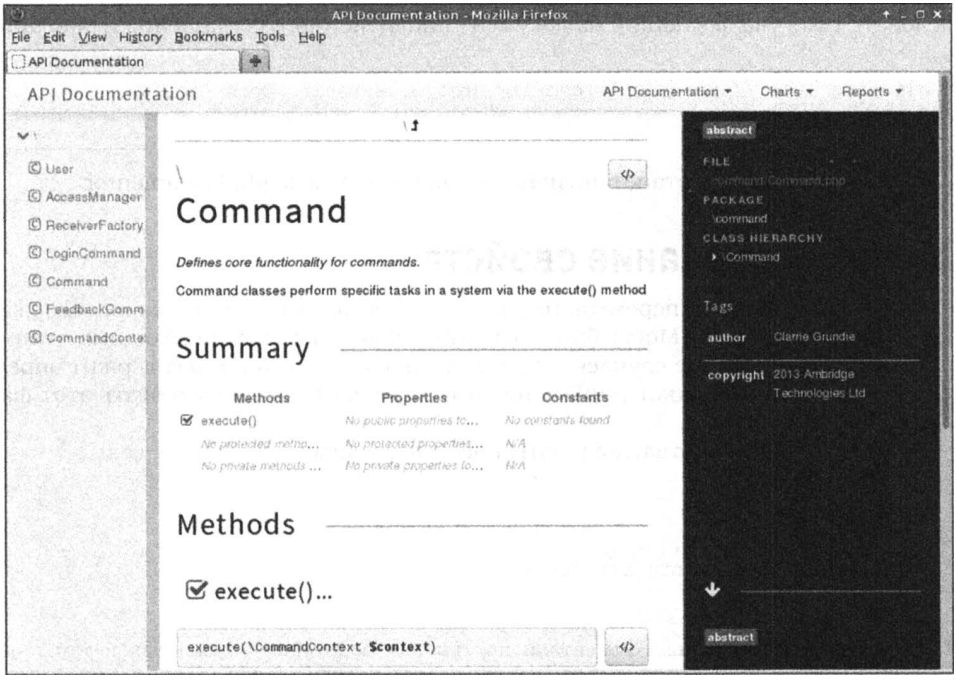


Рис. 16.4. Подробности о классе в выводе документации

Обратите внимание на то, что нам не нужно сообщать phpDocumentor, что класс `Command` — абстрактный. Это подтверждает то, что мы уже знаем, — phpDocumentor анализирует классы, с которыми работает, даже без нашей помощи. Но важно также понять, что комментарии DocBlock зависят от контекста.

Пакет phpDocumentor понимает, что мы документируем класс из предыдущего листинга, потому что комментарий DocBlock, который он встречает, непосредственно предшествует объявлению класса.

Документация на уровне файла

Хотя я склонен размышлять с точки зрения классов, а не содержащих их файлов, в некоторых проектах есть серьезные причины выполнять документацию на уровне файлов. В файлы обычно помещается информация об авторском праве или лицензионное соглашение.

Файловый комментарий должен быть первым комментарием DocBlock в документе. Он должен содержать тег `@package` и не должен непосредственно предшествовать блоку кода. Другими словами, если вы добавляете комментарий DocBlock уровня файла, то должны также обязательно добавить комментарий уровня класса перед первым объявлением класса.

Во многих проектах с открытым исходным кодом требуется, чтобы в каждый файл было включено уведомление о лицензионном соглашении или ссылка на него. Поэтому можно использовать комментарии DocBlock уровня файла для включения информации о лицензионном соглашении, которую вы не хотите повторять для каждого класса. Для этого можно использовать тег `@license`. За тегом `@license` должен следовать URL, указывающий на документ лицензионного соглашения и описание.

```
/**
 * @license http://www.example.com/lic.html Borsetshire Open License
 * @package command
 */
```

URL в теге `@license` станет активной ссылкой в выводе phpDocumentor.

Документирование свойств

В PHP типы свойств перемешаны, т.е. свойство может потенциально содержать значение любого типа. Могут быть ситуации, в которых вам необходима эта гибкость, но в большинстве случаев вы должны знать, что свойство содержит определенный тип данных. Пакет phpDocumentor позволяет документировать этот факт с помощью тега `@var`.

Вот некоторые свойства, документированные в классе `CommandContext`.

```
class CommandContext {
/**
 * The application name.
 * Used by various clients for error messages, etc.
 *
 * Имя приложения.
 * Используется различными клиентами при выводе сообщений об ошибках и т.п.
 *
 * @var string
 */
  public $applicationName;

/**
 * Encapsulated Keys/values.
 * This class is essentially a wrapper for this array.
 *
 * Содержит данные в формате "ключ-значение".
 *  Этот класс, по сути, является оболочкой для ассоциативного массива
 * @var array
 */
  private $params = array();

/**
 * An error message.
 * Сообщение об ошибке.
 * @var string
```

```
 */
  private $error = "";

// ...
```

Как видите, мы предоставляем краткую информацию для каждого свойства и более полную — для первых двух. Мы используем тег `@var`, чтобы определить тип каждого свойства. Документированные свойства показаны на рис. 16.5.

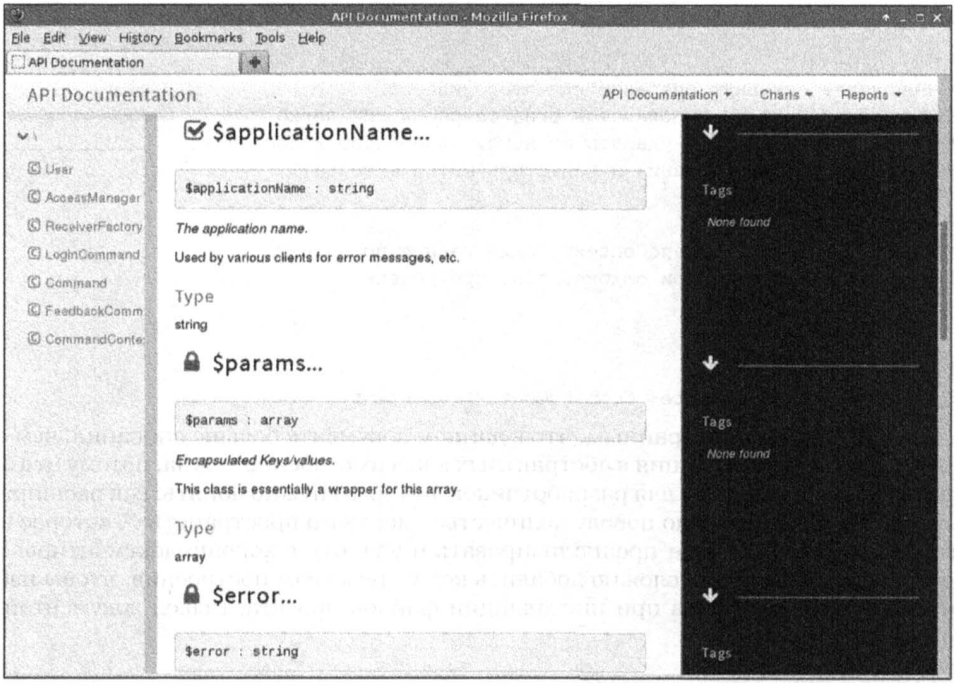


Рис. 16.5. Документирование свойств

Документирование методов

Методы, наряду с классами, — это то, что необходимо документировать прежде всего. Как минимум читатели должны знать аргументы метода, действие, которое он выполняет, и возвращаемое значение.

Как и комментарии DocBlock уровня класса, документация метода должна состоять из двух блоков текста: одной строки общей информации и необязательного описания. Вы можете предоставить информацию о каждом аргументе метода с помощью тега `@param`. Каждый тег `@param` должен начинаться с новой строки, а за ним должны следовать имя аргумента, его тип и краткое описание.

Поскольку в PHP 5 нет ограничений для возвращаемых типов, особенно важно документировать значение, которое возвращает метод. Это можно сделать с помощью тега `@return`. Тег `@return` должен начинаться с новой строки, а за ним должны следовать тип возвращаемого значения и краткое описание. Мы объединим эти элементы в следующем примере.

```
/**
 * Perform the key operation encapsulated by the class.
 * Command classes encapsulate a single operation. They
 * are easy to add to and remove from a project, can be
 * stored after instantiation and execute() invoked at
 * leisure.
 * @param $context CommandContext Shared contextual data
 * @return bool false on failure, true on success
 */

/**
 * Выполняет основную операцию для этого класса.
 * В классах Command выполняется одна операция. Их легко
 * добавлять в проект и удалять из него, экземпляры класса можно
 * сохранить после создания и запустить метод execute()
 * на досуге.
 *
 * @param $context CommandContext Общие данные приложения
 * @return bool false при ошибке, true при успехе
 */


abstract function execute( CommandContext $context );
```

Может показаться странным, что в данном документе больше описания, чем самого кода. Но документация в абстрактных классах особенно важна, потому что она предоставляет указания для разработчиков, которым нужно понять, как расширять классы. Не волнуйтесь по поводу количества "мертвого пространства", которое интерпретатор PHP должен проанализировать и удалить в хорошо документированном проекте. Совсем не сложно добавить код к средствам построения, чтобы избавиться от комментариев при инсталляции файлов проекта. Вывод документации показан на рис. 16.6.

Поддержка пространств имен

Как вы уже знаете, до появления PHP 5.3 большие проекты приходилось разбивать на логические фрагменты, используя соглашения об именовании файлов, а для их документирования — тег `@package`. В новых версиях PHP появилась возможность упорядочивать программные компоненты в пространства имен. Как вы уже, наверное, догадались, phpDocumentor распознает и документирует пространства имен, которые используются в вашем проекте.

Давайте обновим наш проект "megaquiz" так, чтобы в нем использовались пространства имен. Ниже приведен код обновленного класса `Command` с удаленными комментариями, чтобы вы смогли лучше рассмотреть его структуру и понять, как используется пространство имен.

```
namespace megaquiz\command;

   abstract class Command {
      abstract function execute( megaquiz\command\CommandContext $context );
   }
```

На рис. 16.7 показано, что в документации пространство имен расположено на вершине иерархии классов.

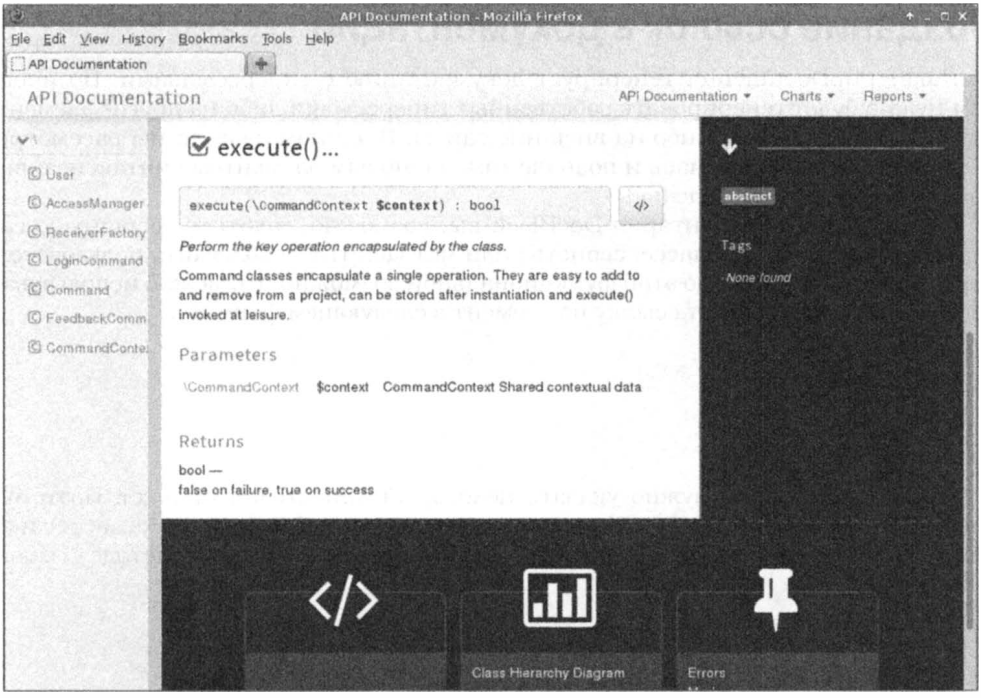


Рис. 16.6. Документирование методов

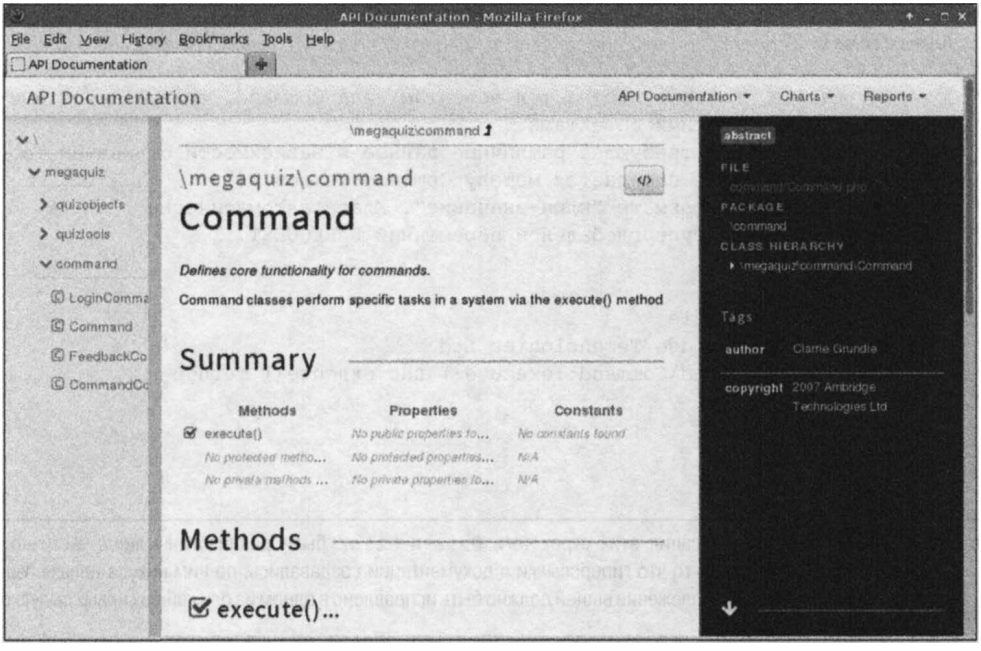


Рис. 16.7. Документирование пространства имен

Создание ссылок в документации

Пакет phpDocumentor генерирует документацию с гиперссылками. Но иногда вам нужно будет генерировать собственные гиперссылки либо на другие элементы внутри документации, либо на внешние сайты. В данном разделе мы рассмотрим теги для обоих этих случаев и познакомимся с новым элементом синтаксиса: внутристрочным тегом (inline tag).

При создании комментария DocBlock вы, возможно, захотите сообщить о связанном (родственном) классе, свойстве или методе. Чтобы облегчить пользователю перемещение к описанию этой функциональной возможности, можно использовать тег `@see`. Этот тег требует ссылку на элемент в следующем формате.

```
class
class::method()
```

или

```
class::$property
```

После этой ссылки нужно указать некоторый описательный текст. Поэтому в следующем комментарии DocBlock мы документируем объект `CommandContext` и акцентируем внимание на том факте, что он обычно используется в методе `Command::execute()`.

```
namespace megaquiz\command;
/**
 * Encapsulates data for passing to, from and between Commands.
 * Commands require disparate data according to context. The
 * CommandContext object is passed to the Command::execute()
 * method and contains data in key/value format. The class
 * automatically extracts the contents of the $_REQUEST
 * superglobal.
 *
 * Содержит данные, которые передаются объектам типа Command, возвращаются ими
 * и передаются между этими объектами.
 * Для объектов Command требуются различные данные в зависимости от контекста.
 * Объект CommandContext передается методу Command::execute()
 * и содержит данные в формате "ключ-значение". Класс автоматически
 * извлекает данные из суперглобальной переменной $_REQUEST.
 *
 * @package command
 * @author Clarrie Grundie
 * @copyright 2015 Ambridge Technologies Ltd
 * @see \megaquiz\command\Command::execute() the execute() method
 */

class CommandContext {
// ...
```

На заметку. На момент написания этих строк теги `@see` и `@link` были реализованы лишь частично в phpDocumentor. Несмотря на то что гиперссылки в документации создавались, по ним никуда нельзя было перейти. Разумеется, такое положение вещей должно быть исправлено в одном из ближайших новых выпусков phpDocumentor.

Как видно на рис. 16.8, тег `@see` превращается в ссылку. Щелкнув на ней, вы переходите к методу `execute()`.

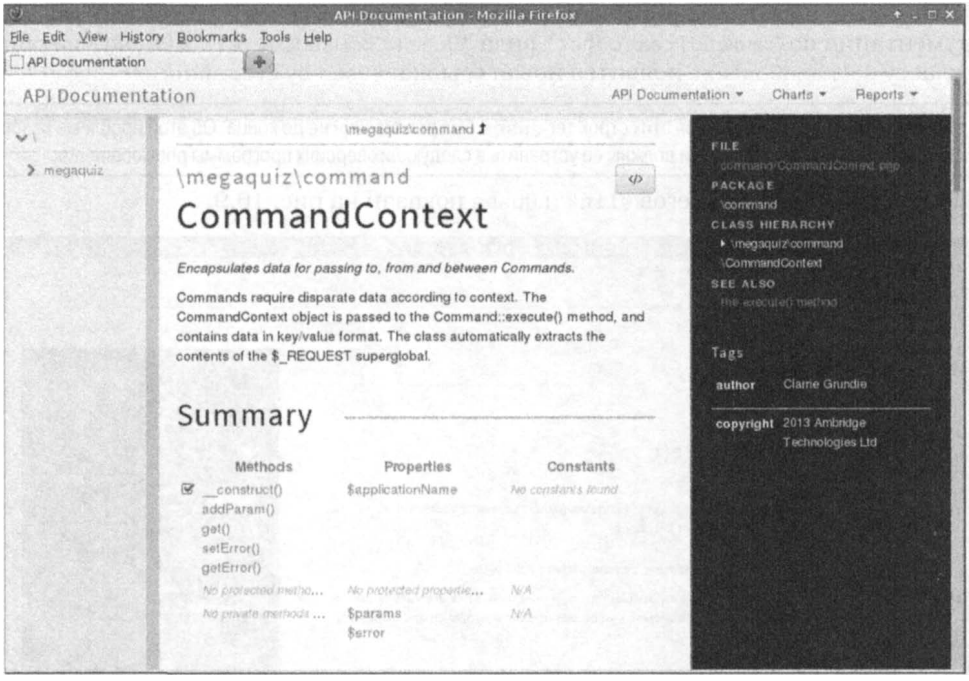


Рис. 16.8. Создание ссылки с помощью тега `@see`

На заметку. В предыдущих версиях программы phpDocumentor для встраивания тега `@see` в текст документации нужно было использовать следующий синтаксис.

```
{@see class::method()}
```

Сейчас такой синтаксис не поддерживается, хотя по секрету мне сказали, что вполне возможно, что он будет снова поддерживаться в одной из следующих версий phpDocumentor.

Для создания ссылок на веб-ресурсы используется тег `@link`. Просто укажите после него адрес URL и текст с описанием, как показано ниже.

```
@link http://www.example.com Подробнее здесь...
```

И снова URL — это адрес перехода, а следующее за ним описание — активный текст, на котором можно щелкнуть.

Вам может понадобиться сделать обратную ссылку. В классе `Command` используются объекты `CommandContext`, поэтому мы можем создать ссылку от `Command::execute()` на класс `CommandContext` и обратную ссылку в противоположном направлении. Конечно, мы можем сделать это с помощью двух тегов — `@link` и `@see`. Но `@uses` позволяет все это сделать с помощью одного тега.

```
/**
 * Perform the key operation encapsulated by the class.
 * ...
 * @param $context Shared contextual data
 * @return bool    false on failure, true on success
 * @link http://www.example.com More info
 * @uses \megaquiz\command\CommandContext
 */
abstract function execute( CommandContext $context );
```

Как только программа phpDocumentor встречает тег `@uses`, она создает ссылку в документации по `Command::execute()` вида `"Uses: CommandContext"`. В документации по классу `CommandContext` появится новая ссылка: `"Used by Command::execute()"`.

На заметку. На момент написания этих строк тег `@uses` был реализован не до конца. Об этой проблеме разработчикам давно известно, и они должны ее устранить в следующих версиях программы phpDocumentor.

Результат обработки тегов `@link` и `@uses` показан на рис. 16.9.

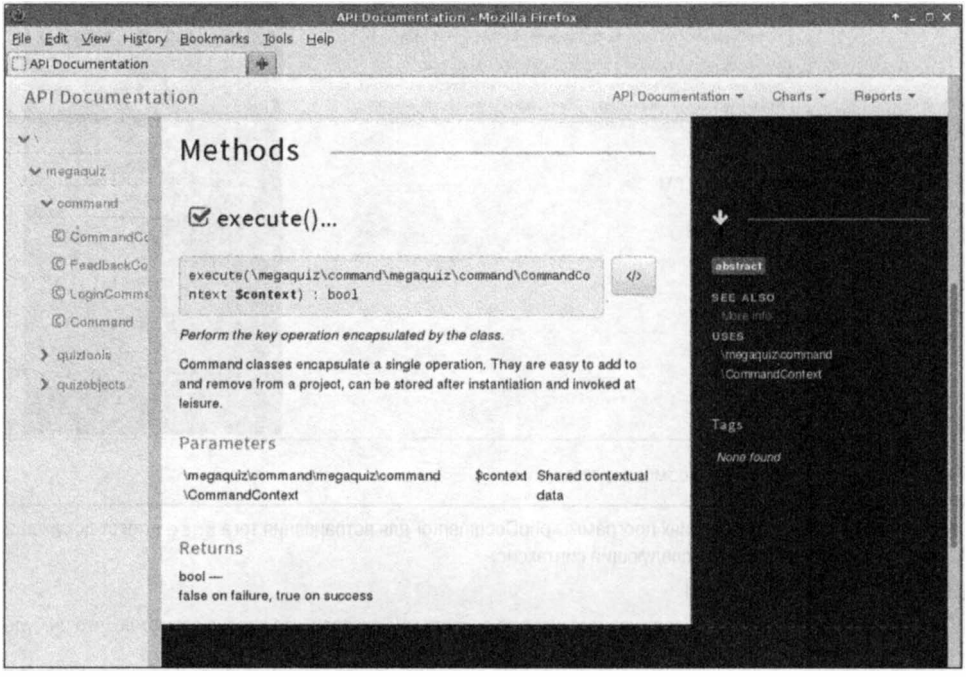


Рис. 16.9. Документация, включающая теги `@link` и `@uses`

Резюме

В данной главе я описал несколько основных возможностей пакета phpDocumentor. Мы познакомились с синтаксисом комментариев DocBlock и тегами, которые можно использовать, а также рассмотрели способы документирования классов, свойств и методов. Вы получили достаточно информации для того, чтобы преобразовать свою документацию и тем самым значительно улучшить возможности совместной работы с коллегами. Это особенно верно, если возможности документирования используются в сочетании со средствами построения и контролем версий. В рамках данной книги я не могу уделить данному приложению больше внимания, поэтому, обязательно посетите сайт, посвященный программе phpDocumentor, по адресу `http://www.phpdoc.org`.

Глава 17

Контроль версий с помощью Git

У всех несчастий есть критический момент, когда порядок окончательно нарушается и события просто выходят из-под контроля. Вы когда-либо оказывались в такой ситуации во время работы над проектом? Были ли вы в состоянии распознать этот критический момент? Возможно, это произошло, когда вы сделали "всего пару изменений" и обнаружили, что из-за этого все вокруг нарушилось (и, что еще хуже, вы теперь не знаете, как вернуться к моменту стабильности, который вы только что нарушили). Это могло произойти также, если три члена команды работали над одним набором классов и сохранили свои изменения поверх изменений остальных. А может, вы обнаружили, что код исправления ошибки, который вы уже реализовывали дважды, снова каким-то образом исчез из кодовой базы. Разве в таких ситуациях вам не пригодился бы инструмент, который помогает управлять совместной работой, делать "срезы" проектов и возвращаться назад, если потребуется, а также объединять несколько веток разработки? В данной главе мы рассмотрим Git — средство, которое делает все это и даже больше.

В этой главе будут освещены следующие темы.

- *Базовая конфигурация:* несколько советов по настройке Git.
- *Импортирование:* начало нового проекта.
- *Фиксация изменений:* сохранение работы в хранилище.
- *Обновление:* объединение работы других людей с вашей.
- *Ветвление:* поддержка параллельных веток разработки.

Для чего нужен контроль версий

Контроль версий изменит вашу жизнь, если он еще не сделал этого (хотя бы ту сторону жизни, которая связана с разработкой программного обеспечения). Сколько раз вы достигали стабильного момента в проекте, делали глубокий вдох и снова погружались в хаос разработки? Легко ли было возвращаться к стабильной версии, когда нужно было продемонстрировать ход выполнения работы? Конечно, по достижении стабильного момента вы могли сохранить "срез" проекта, сделав копию каталога разработки. А теперь представьте, что ваш коллега работает с той же кодовой базой. Возможно, он сохранил стабильную копию кода, как это сделали вы. Но разница в том, что сделанная им копия — это "срез" его работы, а не вашей. И конечно,

у него такой же запутанный каталог разработки. Поэтому вам нужно скоординировать четыре версии проекта. А теперь представьте проект, над которым работают четыре программиста и разработчик пользовательского веб-интерфейса. Кажется, вы побледнели? Может, вам лучше прилечь?

Утилита Git предназначена исключительно для решения описанной выше проблемы. С помощью Git все разработчики могут создать собственные копии кодовой базы, извлеченные из центрального хранилища. Каждый раз, доходя до стабильного момента в коде, они могут получить последнюю версию кодовой базы с сервера и объединить ее с собственной работой. Когда все будет готово и все конфликты будут устранены, разработчики могут поместить новую стабильную версию кодовой базы в общее центральное хранилище на сервере.

Git представляет собой распределенную систему контроля версий. Это означает, что после получения из центрального хранилища ветки кодовой базы разработчики могут вносить изменения в собственное локальное хранилище без установки сетевого доступа к серверу центрального хранилища. Такой подход позволяет получить сразу несколько преимуществ. Прежде всего, он ускоряет выполнение ежедневных рутинных операций, а также позволяет работать над проектом вне офиса, например в командировке, при длительных перелетах в самолете, в поезде или даже в автомобиле. И несмотря на это в конечном итоге вы сможете без проблем сохранить результат своей работы в центральном хранилище на сервере и поделиться ею с коллегами.

Сам факт, что каждый разработчик может объединить свою работу с работой других разработчиков в центральном хранилище, означает, что вам не нужно больше беспокоиться о согласовании различных веток разработки и делать это вручную. Более того, вы можете извлекать из хранилища версии кодовой базы на основе даты или метки. Поэтому, когда код дойдет до стабильного момента и можно будет продемонстрировать клиенту ход выполнения работы, можно пометить этот этап специальной меткой. Затем с помощью этой метки вы сможете извлечь нужную кодовую базу, когда клиент вдруг нагрянет в ваш офис, и произвести впечатление на заказчика.

Но это еще не все! Можно также управлять несколькими ветками разработки одновременно. Если вам кажется, что это излишние сложности, представьте себе зрелый проект. Вы уже выпустили первую версию, и разработка версии 2 идет полным ходом. 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? Git позволяет поддерживать различные ветки кодовой базы. Поэтому вы можете создать специальную ветку для разработчиков, предназначенную для исправления ошибок версии 1.*n* прямо в текущем рабочем коде. В ключевых точках эту ветку можно объединить с кодом версии 2 (т.е. с основной веткой), так что новая версия выиграет от улучшения версии 1.*n*.

На заметку. Git — это не единственная система контроля версий. Вы можете познакомиться также с системами Subversion[1] (`http://subversion.apache.org/`) и Mercurial (`http://mercurial.selenic.com/`).

В этой главе описан необходимый минимум по такой обширной теме, как системы контроля версий. Тем, кто хочет разобраться во всех нюансах Git, рекомендуем прочитать книгу *Pro Git* (Apress, 2009). Ее версия в Интернете находится по адресу `http://git-scm.com/book`.

А теперь давайте рассмотрим эти возможности на практике.

[1] Система Subversion описана в предыдущем издании этой книги. — *Примеч. ред.*

Установка Git

Если вы работаете с операционной системой Unix (такой, как Linux или FreeBSD), то Git-клиент у вас уже наверняка установлен и готов к использованию.

Попробуйте ввести следующую команду из терминала командной строки.

```
$ git help
```

Вы должны увидеть некоторую информацию, подтверждающую, что система установлена и готова к работе. Если у вас еще нет Git, то обратитесь к системной документации. Почти наверняка вы легко можете получить доступ к простому механизму инсталляции, такому как Yum или Apt, либо загрузите Git напрямую с вебсайта по адресу `http://git-scm.com/downloads`.

На заметку. В этой главе команды, вводимые из командной строки, выделяется полужирным шрифтом. А знак доллара $ обозначает приглашение на ввод команды.

Конфигурирование сервера Git

Программа 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 заметить, Git работает на локальном компьютере до момента, когда пользователю нужно будет загрузить файлы из центрального хранилища или обновить их в центральном хранилище. А это означает, что пользователям для выполнения своей работы более не требуется сетевое подключение.

Для работы с Git совершенно не обязательно иметь центральное сетевое хранилище, однако на практике почти всегда имеет смысл его создать, особенно если вы работаете в команде.

В данном разделе мы рассмотрим действия, которые необходимо выполнить, чтобы установить и запустить в работу удаленный сервер Git. При этом я предполагаю, что у вас есть права доступа пользователя `root` на Unix-машине.

Создание сетевого хранилища

Перед созданием сетевого хранилища Git нужно создать для него на сервере подходящий каталог. Для этого подключитесь к удаленному серверу с помощью SSH и введите логин и пароль учетной записи, имеющей права пользователя `root`. Мы будем создавать наше хранилище в папке `/var/git`. Поскольку только пользователь `root` может создавать и модифицировать структуру системных каталогов, все приведенные ниже команды нужно выполнять от его имени.

```
$ mkdir -p /var/git/megaquiz
$ cd /var/git/megaquiz/
```

Здесь мы создали родительский каталог для сетевого хранилища `/var/git`, а в нем — подкаталог `megaquiz` для нашего учебного одноименного проекта. Теперь можно выполнить подготовку самого каталога.

```
$ sudo git init --bare
```

```
Initialized empty Git repository in /var/git/megaquiz/
```

Ключ `--bare` предписывает программе Git создать пустое сетевое хранилище в текущем каталоге, поскольку переменная окружения `GIT_DIR` у нас не еще установлена. Если не выполнить этот шаг, то при попытке записи файлов в хранилище Git выдаст сообщение об ошибке.

На данном этапе только пользователь `root` может работать с каталогом `/var/git`, а это совсем не то, что нам нужно. Поэтому создадим специального пользователя `git` и группу `git` и сделаем их владельцами каталога.

```
$ adduser git
$ chown -R git:git /var/git
```

Подготовка хранилища для локальных пользователей

Поскольку мы работаем с удаленным выделенным сервером, нам нужно сделать так, чтобы локальные пользователи могли помещать файлы своих проектов в хранилище. Если не предпринять специальных действий, при попытке записи файлов в хранилище локальный пользователь столкнется с проблемой в отказе доступа, особенно если до него файлы в хранилище записывали привилегированные пользователи. Поэтому нужно запустить приведенную ниже команду.

```
$ chmod -R g+rws /var/git
```

Она разрешает пользователям группы `git` доступ по записи в каталог `/var/git` и делает так, чтобы всем вновь создаваемым файлам и подкаталогам в этой иерархии присваивались права группы `git`. После того как мы принудительно назначили всем файлам права группы `git`, локальные пользователи могут записывать свои файлы в сетевое хранилище. Более того, любые созданные в хранилище файлы будут доступны по записи всем членам группы `git`.

Чтобы добавить учетную запись локального пользователя в группу `git`, выполните приведенную ниже команду.

```
$ usermod -aG git bob
```

Теперь пользователь `bob` входит в группу `git`.

Предоставление доступа для пользователей

В предыдущем разделе был описан процесс добавления пользователя `bob` к группе `git`. Теперь, подключившись к удаленному серверу с помощью SSH, этот пользователь сможет работать с центральным хранилищем с помощью командной строки. Однако, как правило, системные администраторы ограничивают доступ к серверу через командную оболочку для большинства пользователей. Кроме того, поскольку по своей природе Git является распределенной системой, практически все пользователи предпочитают работать со своими локальными копиями файлов, полученными из центрального хранилища.

Одним из способов предоставления пользователям доступа к системной оболочке через SSH является использование аутентификации с открытым ключом. Для этого сначала нужно каким-то образом получить открытый ключ пользователя SSH. Такой ключ у пользователя уже может быть. Например, на UNIX-машине он, скорее всего, будет находиться в конфигурационном подкаталоге `.ssh` домашнего каталога в файле с именем `id_rsa.pub`. Если же такого файла у вас нет, всегда можно легко сгенерировать новый ключ. На UNIX-машинах для этого нужно запустить команду `ssh-keygen` и скопировать в нужное место сгенерированный ключ, как показано ниже.

```
$ ssh-keygen
$ cat .ssh/id_rsa.pub
```

Копию данного ключа вы должны будете предоставить администратору хранилища. После получения такого ключа администратор должен добавить его в настройки параметров SSH для пользователя `git` на сервере хранилища. Это одна из причин, по которым открытый ключ пользователя нужно добавить в файл `.ssh/authorized_keys`. Для этого создадим конфигурационный подкаталог `.ssh`, если ключ устанавливается впервые. (Приведенные ниже команды выполняются в домашнем каталоге пользователя `git`.)

```
$ mkdir .ssh
$ chmod 0700 .ssh
```

А теперь создадим файл `authorized_keys` и вставим в него открытый ключ пользователя с помощью текстового редактора vi, как показано ниже.

```
$ vi .ssh/authorized_keys
```

На заметку. Как правило, отказ в доступе через SSH происходит из-за того, что при создании конфигурационного подкаталога `.ssh` ему назначаются слишком либеральные права доступа. Данный каталог должен быть доступен для чтения и записи только для владельца учетной записи.

В книге Майкла Штанке (Michael Stahnke) *Pro OpenSSH* (Apress, 2005) содержится всеобъемлющее описание SSH.

Закрытие доступа к системной оболочке для пользователя `git`

К серверу должно быть ровно столько путей доступа, сколько необходимо, и не более того. Поэтому вам нужно сделать так, чтобы пользователь мог работать только с Git.

В системах UNIX полный путь к системной оболочке, назначенной для конкретного пользователя, можно увидеть в файле `/etc/passwd`. Ниже приведена строка из этого файла, относящаяся к пользователю `git` на моем сервере.

```
git:x:1001:1001::/home/git:/bin/bash
```

В поставку Git входит специальная оболочка, которая называется `git-shell`. С ее помощью можно ограничить доступ пользователя к системной оболочке только для команд системы Git. Для этого нужно отредактировать соответствующую строку файла `/etc/passwd`, как показано ниже[2].

```
git:x:1001:1001::/home/git:/usr/bin/git-shell
```

Для того чтобы оболочка `git-shell` работала без ошибок, нужно создать специальный подкаталог `git-shell-commands` в домашнем каталоге соответствующего пользователя (в нашем случае — `git`). Я собираюсь полностью закрыть доступ к системной интерактивной оболочке и вывести соответствующее сообщение для

---

[2] Прямое редактирование файла `passwd` работает не во всех системах UNIX. Для изменения пути к системной оболочке существуют специальные команды. Например, в BSD-совместимых системах нужно ввести

```
pw usermod git -s /usr/bin/git-shell.
```

В системах Linux используется команда

```
usermod -s /usr/bin/git-shell git.
```

За подробностями обратитесь к руководству пользователя для вашей ОС. — *Примеч. ред.*

тех пользователей, кто попытается подключиться к нашему серверу через SSH. Для этого необходимо создать простенький сценарий и поместить его в выполняемый файл `no-interactive-login`, находящийся в подкаталоге `git-shell-commands`, как показано ниже.

```
$ su -s /bin/bash - git
$ mkdir git-shell-commands
$ echo 'echo "Sorry. No interactive access.";' > git-shell-commands/no-
interactive-login
$ chmod 755 git-shell-commands/no-interactive-login
```

В этом сценарии я сначала переключился на пользователя `git`. Обратите внимание, что при этом в команде `su` пришлось указать путь к командному интерпретатору `bash` для пользователя `git`, поскольку раньше мы изменили путь к его стандартной оболочке. Далее я создал подкаталог `git-shell-commands` и поместил в него простенький сценарий, выводящий сообщение для пользователей. Последняя команда делает файл с этим сценарием выполняемым.

Теперь при попытке подключения к серверу с помощью авторизованной учетной записи будет выдано соответствующее сообщение, и соединение с сервером оборвется.

```
$ ssh git@my-git-server
```

```
Enter passphrase for key '/home/mattz/.ssh/id_rsa':
Last login: Mon Oct 21 19:03:58 2013 from some-remote-server
Sorry. No interactive access.
Connection to my-git-server closed.
```

Начало проекта

После установки и настройки удаленного сервера Git и обеспечения доступа к нему со стороны моей локальной учетной записи самое время добавить мой проект в удаленное хранилище `/var/git/megaquiz`.

Прежде чем начать, еще раз внимательно посмотрите на файлы и каталоги и удалите все временные элементы, которые обнаружите. Пренебрежение этим — распространенная и досадная ошибка. К временным элементам, на которые вы должны обратить особое внимание, относятся автоматически сгенерированные файлы, например вывод программы `phpDocumentor`, каталоги построителя, журналы инсталлятора и т.д.

На заметку. Вы можете указать файлы и шаблоны имен файлов, которые нужно игнорировать, создав файл `.gitignore` в вашем хранилище. В системах UNIX примеры содержимого этого файла можно посмотреть по команде `man gitignore`. В них вы увидите, как задаются шаблоны имен файлов, которые позволяют исключить различные lock-файлы и временные каталоги, созданные построителями, редакторами и графическими оболочками. С этой же справочной страницей можно ознакомиться также в Интернете по адресу `http://git-scm.com/docs/gitignore`.

Прежде чем двигаться дальше, я должен зарегистрировать идентификационную информацию о себе в Git. Эта операция позволяет отследить, кто и что делал в хранилище.

```
$ git config --global user.name "matt z"
$ git config --global user.email "matt@getinstance.com"
```

После того как я сообщил свои персональные данные и проверил, что в проекте нет ненужных файлов, можно создать личное хранилище в своем домашнем каталоге и выгрузить из него код проекта на сервер.

```
$ cd megaquiz
$ git init
```

```
Initialized empty Git repository in /home/mattz/work/megaquiz/.git/
```

Теперь добавим сами файлы:

```
$ git add .
```

В результате Git отследит в локальном хранилище все файлы и каталоги, находящиеся внутри каталога нашего проекта `megaquiz`. Отслеживаемые файлы могут находиться в одном из трех состояний: *неизмененный* (unmodified), *измененный* (modified) и *на стадии* (staged) фиксации. Для проверки состояния введите приведенную ниже команду.

```
$ git status
```

```
On branch master

Initial commit

Changes to be committed:
(use "git rm --cached <file>..." to unstage)

new file: command/Command.php
new file: command/CommandContext.php
new file: command/FeedbackCommand.php
new file: command/LoginCommand.php
new file: main.php
new file: quizobjects/User.php
new file: quiztools/AccessManager.php
```

Как видите, благодаря вводу команды `git add` все мои файлы проекта были подготовлены к фиксации в хранилище. Теперь можно запустить команду фиксации данных в хранилище, как показано ниже.

```
$ git commit -m 'my first commit'
```

```
[master (root-commit) f4454af] my first commit
7 files changed, 214 insertions(+), 0 deletions(-)
create mode 100755 command/Command.php
create mode 100755 command/CommandContext.php
create mode 100755 command/FeedbackCommand.php
create mode 100755 command/LoginCommand.php
create mode 100755 main.php
create mode 100755 quizobjects/User.php
create mode 100755 quiztools/AccessManager.php
```

С помощью ключа `-m` я добавил специальное сообщение. Если бы я не сделал этого, программа Git запустила бы текстовый редактор и заставила меня набрать нужное сообщение в нем.

Если вы работали ранее с системами контроля версий, такими как CVS и Subversion, то можете подумать, что больше ничего делать не нужно. На самом деле

хотя теперь я могу успешно продолжать редактировать файлы, добавлять их в хранилище, фиксировать их состояние и создавать отдельные ветки, существует еще одно состояние, которое мы должны обсудить, если мы собираемся помещать код нашего проекта в центральное хранилище на сервере. Как будет показано ниже в этой главе, система Git позволяет создавать и поддерживать несколько веток разработки проекта. Благодаря этому я могу поддерживать отдельную ветку для каждого выпуска продукта и безопасно продолжать работы над новым кодом, не смешивая его с выпущенным ранее. После нашего первого запуска Git создал единственную ветку разработки и назвал ее `master`. Чтобы убедиться в этом, достаточно ввести приведенную ниже команду.

```
$ git branch -a
```

```
* master
```

Ключ `-a` указывает программе Git, что нужно показать все ветки разработки проекта, кроме тех, которые находятся на удаленном сервере (это стандартное поведение программы). Как видите, в выводе программы присутствует только ветка `master`.

По сути, я еще ничего не сделал для того, чтобы как-то привязать мое локальное хранилище к удаленному серверу. Самое время это сделать.

```
$ git remote add --track master origin git@mygitserver:/var/git/megaquiz
```

К сожалению, эта команда удручающе немногословна и совсем не выводит никакой информации, учитывая ту работу, которую она выполняет. По сути, она "говорит" Git: "Свяжи псевдоним `origin` с указанным хранилищем на сервере. Кроме того, отслеживай изменения между локальной веткой разработки проекта под именем `master` и ее эквивалентом на удаленном сервере `origin`".

Чтобы убедиться в том, что Git связал псевдоним `origin` с удаленным сервером, введите приведенную ниже команду.

```
$ git remote -v
```

```
origin git@mygitserver:/var/git/megaquiz (fetch)
origin git@mygitserver:/var/git/megaquiz (push)
```

Однако я еще не выгрузил ни один свой файл на удаленный сервер Git. Поэтому следующий мой шаг будет таким.

```
$ git push origin master
```

```
Enter passphrase for key '/home/mattz/.ssh/id_rsa':
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (12/12), 2.65 KiB, done.
Total 12 (delta 1), reused 0 (delta 0)
To git@mygitserver:/var/git/megaquiz
* [new branch] master -> master
```

Теперь давайте снова запустим команду `git branch` и убедимся в том, что на удаленном сервере также появилась ветка разработки под именем `master`.

```
$ git branch -a
```

```
* master
  remotes/origin/master
```

На заметку. Я выполнил довольно приличный кусок работы, прежде чем мне удалось создать так называемую *отслеживающую ветку* (tracking branch) разработки проекта. По сути, это локальная ветка, которая связана со своей удаленной копией. Благодаря этомуGit знает, что, когда мне понадобится выгрузить (команда `push`) изменения, сделанные в ветке `master`, я буду делать это в ветку `origin/master`, находящуюся на удаленном сервере. Поэтому в ходе клонирования хранилища (команда `clone`) Git автоматически создаст для вас новую отслеживающую ветку разработки.

Клонирование хранилища

В целях изложения материала данной главы я придумал нового члена команды по имени "Боб", который работает со мной над проектом MegaQuiz. Разумеется, Боб захочет получить личную версию кода и поработать над ней самостоятельно. Я уже добавил открытый ключ его учетной записи на сервер Git, поэтому Боб уже может начинать работу.

Чтобы запросить из удаленного хранилища персональную копию кода, Боб должен ввести приведенную ниже команду.

```
$ git clone git@mygitserver:/var/git/megaquiz
```

```
Cloning into 'megaquiz'...
Enter passphrase for key '/home/bob/.ssh/id_rsa':
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 12 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (1/1), done.
Now both Bob and I can develop locally and, when we're ready, share our
code with one another.
```

Теперь мы с Бобом можем работать самостоятельно над одним проектом на своих локальных компьютерах. Как только каждый из нас закончит свою часть работы, мы сможем легко обменяться друг с другом сделанными изменениями.

Обновление и фиксация изменений

Разумеется, Боб — классный и талантливый парень. Но у него есть одна общая для всех программистов и очень раздражающая черта: он не может оставить в покое код, написанный кем-то другим.

Боб умен и любознателен, быстро увлекается новыми направлениями разработки и стремится помочь оптимизировать новый код. В результате, куда бы я ни заглянул, всюду вижу руку Боба. Боб добавил что-то к документации; Боб реализовал идею, которую мы обсуждали с ним за чашкой кофе. Наверное, его пора убить. Но пока мне необходимо объединить код, над которым я работаю, с тем кодом, который добавил Боб.

Вот файл с именем `quizobjects/User.php`. В настоящее время в нем нет ничего, кроме голого каркаса.

```
<?php
  class User {}
?>
```

Я решил добавить документацию. Как вы знаете из предыдущей главы, для этого достаточно добавить комментарии к файлу и классу. Сначала я добавил комментарий в свою версию файла.

```
<?php
/**
 * @license http://www.example.com Borsetshire Open License
 * @package quizobjects
 */
  class User {}
?>
```

Помните, выше я говорил о том, что каждый файл в хранилище может находиться в одном их трех состояний: *неизмененный* (unmodified), *измененный* (modified) и *на стадии* (staged) фиксации? Так вот, файл `User.php` теперь перешел из состояния *неизмененный* в состояние *измененный*. Чтобы убедиться в этом, введите приведенную ниже 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: quizobjects/User.php

no changes added to commit (use "git add" and/or "git commit -a")
```

Как видите, файл `User.php` был *изменен*, но он не находится *на стадии* фиксации. Для изменения этого положения дел воспользуемся командой `git add`, как показано ниже.

```
$ git add quizobjects/User.php
$ git status
```

```
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified: quizobjects/User.php
```

Теперь все готово к выполнению фиксации изменений в хранилище.

```
$ git commit -m 'added documentation' quizobjects/User.php
```

```
[master 2302dd8] added documentation
1 files changed, 5 insertions(+), 0 deletions(-)
```

Команда `git commit` зафиксировала сделанные изменения в моем локальном хранилище. Если я считаю, что эти изменения очень важны для остальных участников проекта, я должен выгрузить мой новый код в удаленное хранилище.

```
$ git push
```

```
Enter passphrase for key '/home/mattz/.ssh/id_rsa':
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 409 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
To git@mygitserver:/var/git/megaquiz
f4454af..2302dd8 master -> master
```

А тем временем, работая в собственной "песочнице", Боб создал комментарий к классу.

```
<?php
/**
 * @package quizobjects
 */
  class User {}
?>
```

Теперь уже Бобу нужно добавить изменения, зафиксировать их в локальном хранилище и выгрузить на сервер. Поскольку операции добавления и фиксации обычно выполняются сразу одна за другой, в Git все это можно выполнить с помощью одной команды.

```
$ git commit -a -m 'my great documentation'
```

```
[master 2d880e6] my great documentation
1 files changed, 3 insertions(+), 0 deletions(-)
```

В результате у нас появились две различные версии файла `User.php`. Одна версия — это та, которую я раньше выгрузил в удаленное хранилище, а вторая — версия Боба, которую он зафиксировал в своем локальном хранилище, но еще не выгрузил на сервер. Что же произойдет, если Боб попытается выгрузить свою версию файла в удаленное хранилище на сервере?

```
$ git push
```

```
Enter passphrase for key '/home/bob/.ssh/id_rsa':
To git@mygitserver:/var/git/megaquiz
! [rejected] master -> master (non-fast-forward)
error: failed to push some refs to 'git@mygitserver:/var/git/megaquiz'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
```

Как видите, Git не позволяет вам выполнить команду `push`, если существует конфликт версий файлов, и для его разрешения нужно применить неочевидные изменения. Поэтому Боб сначала должен получить мою версию файла `User.php` и разобраться во внесенных в нее изменениях.

```
$ git pull
```

```
Enter passphrase for key '/home/bob/.ssh/id_rsa':
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
```

```
Unpacking objects: 100% (4/4), done.
From mygitserver:/var/git/megaquiz
f4454af..2302dd8 master -> origin/master
Auto-merging quizobjects/User.php
CONFLICT (content): Merge conflict in quizobjects/User.php
Automatic merge failed; fix conflicts and then commit the result.
```

Система Git может без проблем объединить данные из двух файлов в одном только в том случае, если сделанные в них изменения не перекрываются. У Git нет средств для обработки изменений, которые затронули одинаковые строки. Как она может решить, у каких изменений более высокий приоритет? Следует ли записать мои изменения поверх изменений Боба или наоборот? И должны ли сосуществовать оба варианта изменений? Какой вариант должен идти первым? У Git нет другого выбора, кроме как сообщить о конфликте и предоставить Бобу возможность уладить проблему.

Вот что увидит Боб, если откроет этот файл.

```
<?php
/**
<<<<<<< HEAD
* @package quizobjects
*/
=======
* @license http://www.example.com Borsetshire Open License
* @package quizobjects
*/
>>>>>>> 2302dd80ad0da869997lbd70e49f43b6fd431518
class User {}
?>
```

Система Git включила комментарий Боба и все изменения, вызвавшие конфликт, вместе с метаданными, которые говорят о том, какая часть откуда взята. Информация о конфликте разделена строкой знаков равенства. Изменения Боба отмечены строкой знаков "меньше чем", за которой следует слово `'HEAD'`. Данные, извлеченные из хранилища, находятся ниже строки-разделителя и отмечены строкой знаков "больше чем".

После того как Боб поймет причину конфликта, он сможет отредактировать файл и тем самым устранить конфликт.

```
<?php
/**
 * @license http://www.example.com Borsetshire Open License
 * @package quizobjects
 */

/**
 * @package quizobjects
 */
class User {}
?>
```

После этого для устранения конфликта Боб должен поместить этот файл в свое локальное хранилище и зафиксировать в нем изменения.

```
$ git add quizobjects/User.php
$ git commit -m 'documentation merged'
```

```
[master a407692] documentation merged
```

И только после этого он может выгрузить измененный файл в удаленное хранилище.

```
$ git push
```

Добавление и удаление файлов и каталогов

По мере развития проекты меняются. Программы контроля версий должны это учитывать, позволяя добавлять пользователям новые файлы и удалять старые, которые в противном случае будут только помехой.

Добавление файла

Выше мы неоднократно сталкивались с командой `git add`. Я использовал ее во время создания проекта для добавления кода в пустое хранилище `megaquiz`, а впоследствии — для подготовки файлов перед фиксацией. Запустив команду `git add` и указав в ней неотслеживаемый файл или каталог вы тем самым даете указание Git начать отслеживание изменений в нем и подготовить его к фиксации. Сейчас я добавлю к проекту новый документ `Question.php`.

```
$ touch quizobjects/Question.php
$ git add quizobjects/Question.php
```

В реальной ситуации я, наверное, начал бы с добавления некоторого содержимого к документу `Question.php`. Но здесь я ограничусь созданием пустого файла с помощью стандартной команды `touch`. После добавления документа мне еще нужно вызвать подкоманду `commit`, чтобы зафиксировать изменения.

```
$ git commit -m 'initial checkin'
```

```
[master 96da108] initial checkin
0 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 quizobjects/Question.php
```

Теперь документ `Question.php` находится в локальном хранилище.

Удаление файла

Если я слишком поспешил и теперь мне нужно удалить документ, неудивительно, что для этого нужно использовать подкоманду `rm`.

```
$ git rm quizobjects/Question.php
```

```
rm 'quizobjects/Question.php'
```

И снова для фиксации изменений необходима подкоманда `commit`. Как и прежде ,я могу в этом убедиться, запустив команду `git status`.

```
$ git status
```

```
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

deleted: quizobjects/Question.php
```

```
$ git commit -m'removed Question'
```

```
[master 0d571b2] removed Question
0 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 quizobjects/Question.php
```

Добавление каталога

С помощью подкоманд `add` и `rm` можно также добавлять и удалять каталоги. Предположим, что Бобу нужно создать новый каталог.

```
$ 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 выгрузить их на удаленный сервер как обычно.

Удаление каталогов

Как и можно было ожидать, для удаления каталогов используется подкоманда `rm`. В данном случае я хочу удалить сам каталог и его содержимое, поэтому к подкоманде `rm` нужно добавить ключ `-r`. Здесь я категорически не согласен с Бобом, что нужно добавить в проект каталог `resources`.

```
$ git rm -r resources/
```

```
rm 'resources/blah.gif'
```

Обратите внимание на то, что эта подкоманда с ключом `-r` работает рекурсивно. Несмотря на это, для фиксации изменений в хранилище нужна подкоманда `commit`.

Маркировка готовой версии продукта

Все идет хорошо, проект в конце концов достигает состояния готовности, и вы хотите начать его продажи или развертывание. При выпуске готовой версии продукта вы должны оставить в хранилище специальную метку, которая позволит в дальнейшем вернуться к текущему состоянию кода и внести в него нужные изменения. Как и можно было ожидать, метка (tag) в коде делается с помощью команды `git tag`.

```
$ git tag -a 'version1.0' -m 'release 1.0'
```

Увидеть, какие метки сделаны в вашем хранилище, можно, вызвав команду `git tag` без параметров.

```
$ git tag
```

```
version1.0
```

До сих пор мы работали на локальном компьютере. Для того чтобы переслать метки в удаленное хранилище, нужно воспользоваться командой `git push`, указав в ней ключ `--tags`.

```
$ git push --tags
```

```
Enter passphrase for key '/home/mattz/.ssh/id_rsa':
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@mygitserver:/var/git/megaquiz
* [new tag] version1.0 -> version1.0
```

После создания метки в хранилище кода у вас наверняка возникнет вопрос, как этим можно воспользоваться, чтобы впоследствии внести изменения в старые версии ваших программ. Однако сначала мы должны рассмотреть вопрос разветвления проекта, уж в чем-чем, а в этом Git особенно хорош!

Разветвление проекта

Теперь, когда проект закончен, можно подумать о том, чтобы сделать что-то новенькое, правда? В конце концов, ваш код был так хорошо написан и отлажен, что ошибки в нем просто невозможны, и так тщательно продуман, что пользователям не понадобятся новые возможности!

Но в реальности мы должны продолжать работу с кодовой базой по меньшей мере на двух уровнях. Ведь вам 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. Конечно, исправить ошибки можно в процессе разработки и выпустить исправления за один заход, когда появится следующая стабильная версия системы. Но так пользователям придется долго ждать, пока они увидят какие-то исправления. А это совершенно неприемлемо. С другой стороны, мы можем выпустить проект в его текущем состоянии, но с исправленными ошибками, продолжая над ним работу. В этом случае мы рискуем выпустить не до конца работоспособный код. Очевидно, нам нужны две ветки разработки. Тогда мы сможем, как и раньше, продолжать работу над проектом и добавлять новые и неотлаженные возможности в его основную ветку (ее часто называют транком[3]), а исправления замеченных пользователями ошибок вносить в другую ветку. Поэтому сейчас нам нужно создать ветку для только что выпущенной версии продукта, чтобы впоследствии в нее можно было вносить требуемые исправления.

Создать новую ветку и переключиться на нее можно с помощью команды `git checkout`. Но сначала давайте посмотрим, какие существуют ветки разработки на данный момент.

```
$ git branch -a
```

```
* master
remotes/origin/master
```

[3] От англ. trunk — "основа", "магистраль". — *Примеч. ред.*

Как видите, у меня есть всего одна ветка разработки `master` и ее эквивалент на удаленном сервере. Давайте создадим новую ветку разработки и переключимся на нее.

```
$ git checkout -b megaquiz-branch1.0
```

```
Switched to a new branch 'megaquiz-branch1.0'
```

Чтобы можно было отслеживать мои ветки разработки, я воспользуюсь каким-нибудь файлом, например `command/FeedbackCommand.php`. Кажется, что я создал специальную ветку для исправления ошибок как раз вовремя. Пользователи программы уже сообщили мне, что им не удается воспользоваться механизмом обратной связи, реализованным в моей системе. Мне удалось локализовать ошибку, как показано ниже.

```
//...
$result = $msgSystem->dispatch( $email, $msg, $topic );
if ( ! $user ) {
    $this->context->setError( $msgSystem->getError() );
//...
```

По сути, мне нужно проверять значение переменной `$result`, а не `$user`. Я внес изменения в код.

```
//...
$result = $msgSystem->dispatch( $email, $msg, $topic );
if ( ! $result ) {
    $this->context->setError( $msgSystem->getError() );
//...
```

Поскольку я уже переключился на ветку `megaquiz-branch1.0` и работаю с ней, я могу зафиксировать изменения в локальном хранилище.

```
$ git add command/FeedbackCommand.php
$ git commit -m 'bugfix'
```

```
[megaquiz-branch1.0 5c963b6] bugfix
1 files changed, 1 insertions(+), 1 deletions(-)
```

Чтобы перенести изменения в удаленное хранилище, воспользуемся командой `git push`.

```
$ git push -u origin megaquiz-branch1.0
```

```
Enter passphrase for key '/home/mattz/.ssh/id_rsa':
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 431 bytes, done.
Total 4 (delta 2), reused 0 (delta 0)
To git@mygitserver:/var/git/megaquiz
* [new branch] megaquiz-branch1.0 -> megaquiz-branch1.0
Branch megaquiz-branch1.0 set up to track remote branch megaquiz-branch1.0
from origin.
```

Напомню, что в качестве псевдонима для удаленного хранилища было выбрано имя `origin`. Ключ `-u` позволяет привязать указанную ветку в локальном хранилище к аналогичной ветке в удаленном хранилище и отслеживать сделанные в ней из-

менения. Если при выполнении команды `git push` опустить этот ключ, то для установления связи между двумя хранилищами воспользуйтесь приведенной ниже командой.

```
$ git branch --set-upstream megaquiz-branch1.0 origin/megaquiz-branch1.0
```

А теперь снова вернемся к Бобу. Наверняка он уже рьяно взялся за дело и тоже исправил некоторые ошибки. Для начала он должен запустить команду `git pull`, которая тут же сообщит ему о появлении новой ветки разработки.

```
$ git pull
```

```
Enter passphrase for key '/home/bob/.ssh/id_rsa':
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From mygitserver:/var/git/megaquiz
* [new branch] megaquiz-branch1.0 -> origin/megaquiz-branch1.0
Already up-to-date.
```

При этом Боб должен у себя создать локальную ветку и привязать ее к удаленной ветке.

```
$ git checkout -b megaquiz-branch1.0 origin/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. Он может вносить любые изменения и фиксировать их в локальном хранилище. Все они окажутся в удаленном хранилище после выполнения команды `git push`.

А тем временем мне понадобилось внести несколько новых изменений в транк, т.е. в основную ветку разработки. Давайте снова взглянем на состояние моих веток разработки с точки зрения локального хранилища.

```
$ git branch -a
```

```
  master
* megaquiz-branch1.0
  remotes/origin/master
```

Для переключения на другую ветку я должен воспользоваться командой `git checkout`, но на этот раз ключ `-b` указывать не нужно.

```
$ git checkout master
```

```
Switched to branch 'master'
```

И если я теперь снова взгляну на содержимое файла `command/FeedbackCommand.php`, то увижу, что внесенное ранее мною исправление ошибки загадочно исчезло! Разумеется, оно осталось в хранилище в ветке `megaquiz-branch1.0`. Поэтому позже я всегда смогу слить все сделанные изменения в основную ветку разработки `master`. Так что волноваться не о чем! Вместо этого я пока сосредоточусь на добавлении нового фрагмента кода.

```
class FeedbackCommand extends Command {
   function execute( CommandContext $context ) {
      // Здесь будут новые и неотлаженные
      // функции программы
      $msgSystem = ReceiverFactory::getMessageSystem();
      $email = $context->get( 'email' );
      $msg = $context->get( 'pass' );
      $topic = $context->get( 'topic' );
      $result = $msgSystem->dispatch( $email, $msg, $topic );
      if ( ! $user ) {
         $this->context->setError( $msgSystem->getError() );
         return false;
      }
      $context->addParam( "user", $user );
      return true;
   }
}
```

Здесь я добавил комментарий, чтобы сымитировать вставку в код нового фрагмента.

Итак, теперь у меня есть две параллельные ветки разработки. Разумеется, рано или поздно я захочу внести все сделанные в ветке `megaquiz-branch1.0` исправления ошибок в мою основную ветку разработки. Сделать это очень легко, в Git предусмотрена специальная команда `merge`.

```
$ git checkout master
```

```
Switched to branch 'master'
```

```
$ git merge megaquiz-branch1.0
```

```
Updating 102966e..5c963b6
Fast-forward
command/FeedbackCommand.php | 2 +-
1 files changed, 1 insertions(+), 1 deletions(-)
```

На заметку. Объединять или не объединять? Этот выбор не всегда так прост, как может показаться. Например, в некоторых случаях исправление ошибки может быть видом временной работы, которая заменяется более тщательным рефакторингом основной ветки или больше не применяется по причине изменения в спецификации. Этот вопрос обязательно надо решить заранее. Но в большинстве команд, в которых я работал, обычно по возможности объединяли работу в основной ветке, в то же время сводя работу над ответвлениями к минимуму. Новые для нас возможности обычно появляются в основной ветке и быстро доходят до пользователей путем политики "быстрого и частого выпуска версий".

Теперь, если мы посмотрим на версию `FeedbackCommand` в основной ветке, то увидим, что все слияния были сделаны.

```
function execute( CommandContext $context ) {
      // Здесь будут новые и неотлаженные
      // функции программы
      $msgSystem = ReceiverFactory::getMessageSystem();
      $email = $context->get( 'email' );
      $msg = $context->get( 'pass' );
      $topic = $context->get( 'topic' );
      $result = $msgSystem->dispatch( $email, $msg, $topic );
```

```
        if ( ! $result ) {
            $this->context->setError( $msgSystem->getError() );
            return false;
        }
        $context->addParam( "user", $user );
    return true;
}
```

Теперь в метод `execute()` включены и имитированный код основной ветки

```
// Здесь будут новые и неотлаженные
// функции программы
```

и исправление ошибки:

```
if ( ! $result ) {
```

Я создал новую ветку сразу после того как была "выпущена" первая версия программы MegaQuiz. Именно в нее мы вносили исправления ошибок.

Также напомню, что на этом этапе я создал метку в хранилище. Тогда я обещал, что со временем покажу, как с ней можно работать. По сути, вы уже это увидели. Вы можете создать локальную ветку разработки на основе метки точно так, как Боб создал свою локальную копию ветки для исправления ошибок. Разница заключается в том, что эта новая ветка будет полностью чистой. В ней не будут отслеживаться изменения, внесенные в удаленное хранилище.

```
$ git checkout -b new-version1.0-branch version1.0
```

```
Switched to a new branch 'new-version1.0-branch'
```

Теперь, после создания новой ветки, я могу выгрузить ее в удаленное хранилище и сделать доступной для других участников команды, как было показано выше.

Резюме

Система Git содержит огромное количество подкоманд с множеством опций и возможностей. Поэтому в рамках этой главы я старался дать только краткое введение в данную тему. Тем не менее, если вы будете использовать хотя бы те возможности, которые описаны в данной главе, то почувствуете, насколько полезна система Git — и для защиты от потери данных, и для улучшения совместной работы.

В данной главе я кратко описал лишь основы Git. Мы изучили настройку хранилища перед импортированием проекта. Мы извлекли, зафиксировали и обновили код и в конце концов пометили и экспортировали новую версию. В конце главы мы узнали, что такое ветки, и продемонстрировали их полезность для поддержания параллельных веток разработки и исправления ошибок в проекте.

Есть еще один момент, который я обошел стороной. Мы установили принцип, согласно которому разработчики должны сначала получить из удаленного хранилища собственные версии проекта. Но вообще говоря, проекты не запускаются на одном месте. Чтобы протестировать внесенные изменения, разработчики должны развернуть проект локально на своем компьютере. Иногда для этого достаточно скопировать несколько каталогов. Но гораздо чаще при развертывании необходимо решать ряд вопросов конфигурации. В следующей главе мы рассмотрим некоторые методы автоматизации этого процесса.

Глава 18

Тестирование с помощью PHPUnit

Успешная работа 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 нарушения. Улучшая классы и пакеты, вы должны не забывать изменять код, который с ними связан. В случае некоторых изменений это может вызвать "волновой" эффект, в результате чего будут затронуты компоненты, далекие от кода, который вы первоначально изменили. Зоркость и проницательность, а также энциклопедические знания зависимостей системы помогут в решении данной проблемы. И хотя это отличные качества, сложность системы обычно возрастает так быстро, что вряд ли вы сможете легко предсказать каждый нежелательный эффект. Немалую роль в этом играет то, что, как правило, над системой работает много разработчиков. Для решения данной проблемы имеет смысл регулярно тестировать каждый компонент. Конечно, это сложная задача, которую к тому же нужно повторять многократно, поэтому лучше ее автоматизировать.

Среди тестовых программ, имеющихся в распоряжении PHP-программистов, PHPUnit, вероятно, является самой распространенной и точно уж самой полнофункциональной. В данной главе мы изучим перечисленные ниже моменты использования PHPUnit.

- *Инсталляция:* использование PEAR для инсталляции PHPUnit.
- *Написание тестов:* создание тестов для различных контрольных примеров и использование методов с утверждениями (assertion methods).
- *Обработка исключений:* стратегии для подтверждения отказа.
- *Запуск нескольких тестов:* объединение тестов в наборы.
- *Построение логики утверждений:* использование ограничений.
- *Фиктивные компоненты:* имитации и заглушки.
- *Тестирование веб-приложений:* как это делается с помощью и без помощи специализированных средств.

Функциональные тесты и модульное тестирование

Тестирование имеет большое значение для любого проекта. Если этот процесс не формализован, то вы должны создать неформальный список действий, которые позволят проверить работоспособность системы и выявить ее слабые места. Вероятно, это вам скоро надоест, что приведет к наплевательскому отношению к тестированию проектов.

Один из подходов к тестированию начинается с интерфейса проекта, когда моделируются различные способы коммуникации пользователя с системой. Вероятно, этот способ вы будете использовать при тестировании вручную, хотя существуют различные схемы автоматизации этого процесса. Эти функциональные тесты иногда называют приемочными испытаниями, потому что список действий, выполненных успешно, можно использовать как критерий завершения этапа проекта. При использовании этого подхода система обычно рассматривается как "черный ящик" — тесты преднамеренно ничего не знают о скрытых взаимодействующих компонентах, формирующих тестируемую систему.

В то время как функциональные тесты работают извне, модульные тесты (тема данной главы) работают изнутри наружу. Модульное тестирование фокусируется на классах, причем методы тестирования группируются для контрольных примеров. Каждый контрольный пример подвергает один класс серьезному испытанию, проверяя, каждый ли метод работает, как нужно, и дает сбой там, где это должно быть. Цель заключается в том, чтобы по возможности протестировать каждый компонент отдельно от более широкого контекста. Во многих случаях это помогает понять, насколько успешно вы разделили части системы.

Тесты можно запускать как часть процесса построения непосредственно из командной строки или даже через веб-интерфейс. В данной главе я рассмотрю метод запуска из командной строки.

Модульное тестирование — это хороший способ обеспечить качество проекта в системе. Тесты выявляют обязанности классов и функций. Некоторые программисты даже отстаивают подход "сначала тестирование". Они говорят, что тесты нужно написать до того, как вы вообще начнете работу над классом. Это позволяет установить назначение класса, создать понятный интерфейс и краткие, сфокусированные методы. Лично я никогда не стремлюсь к такому уровню правильности — это просто не соответствует моему стилю программирования. Тем не менее я стараюсь писать тесты по мере разработки. Тестирование обеспечивает уровень безопасности, нужный мне для рефакторинга кода. Я могу извлечь и заменить целые пакеты, если знаю, что у меня есть шанс найти в системе неожиданные ошибки.

Тестирование вручную

В предыдущем разделе я говорил, что тестирование очень важно для любого проекта. Но точнее было бы сказать, что тестирование *неизбежно* для любого проекта. Все мы занимаемся тестированием. Но трагедия в том, что мы часто пренебрегаем этой важной работой.

Итак, давайте создадим несколько классов для тестирования. Ниже приведен класс, в котором хранится информация о пользователе и из которого она извлекается. Для целей демонстрации сделаем так, чтобы он генерировал массивы, а не объекты типа `User`, которые обычно используются в таких случаях.

```
class UserStore {
   private $users = array();

   function addUser( $name, $mail, $pass ) {
      if ( isset( $this->users[$mail] ) ) {
         throw new Exception(
              "Пользователь {$mail} уже зарегистрирован.");
      }

      if ( strlen( $pass ) < 5 ) {
         throw new Exception(
              "Длина пароля должна быть не менее 5 символов.");
      }

      $this->users[$mail] = array( 'pass' => $pass,
                                   'mail' => $mail,
                                   'name' => $name );
      return true;
}

   function notifyPasswordFailure( $mail ) {
      if ( isset( $this->users[$mail] ) ) {
         $this->users[$mail]['failed']=time();
      }
   }

   function getUser( $mail ) {
      return ( $this->users[$mail] );
   }
}
```

Этому классу передаются данные о пользователе с помощью метода `addUser()`, а для их извлечения используется метод `getUser()`. Адрес электронной почты пользователя используется в качестве ключа для извлечения. Если вы похожи на меня, то напишете пример реализации в ходе разработки, чтобы убедиться, что все работает так, как вы задумали. Это будет выглядеть примерно так.

```
$store=new UserStore();
$store->addUser( "bob williams",
                 "bob@example.com",
                 "12345" );
$user = $store->getUser( "bob@example.com" );
print_r( $user );
```

Во время работы над классом я могу поместить этот фрагмент в конце файла, который его содержит. Конечно, проверка правильности теста осуществляется вручную; я должен просмотреть результаты и убедиться в том, что данные, возвращенные методом `UserStore::getUser()`, соответствуют информации, которую я добавил первоначально. Тем не менее это возможный вариант теста.

Вот клиентский класс, в котором используется класс `UserStore` для подтверждения того, что пользователь предоставил правильную информацию для аутентификации.

```
class Validator {
   private $store;

   public function __construct( UserStore $store ) {
```

```
        $this->store = $store;
    }

    public function validateUser( $mail, $pass ) {
        if ( ! is_array($user = $this->store->getUser( $mail )) ) {
            return false;
        }
        if ( $user['pass'] == $pass ) {
            return true;
        }
        $this->store->notifyPasswordFailure( $mail );
        return false;
    }
}
```

Конструктору этого класса требуется указать объект типа `UserStore`, ссылка на который сохраняется в свойстве `$store`. Это свойство используется методом `validateUser()`, во-первых, для гарантии того, что пользователь, на которого ссылается данный адрес электронной почты, существует в хранилище, а во-вторых, что пароль пользователя соответствует предоставленному аргументу. Если оба эти условия выполняются, то метод возвращает истинное значение. И опять-таки, я могу протестировать это по ходу разработки.

```
$store = new UserStore();
$store->addUser( "bob williams", "bob@example.com", "12345" );

$validator = new Validator( $store );
if ( $validator->validateUser( "bob@example.com", "12345" ) ) {
    print "Привет, друзья!\n";
}
```

Я создаю экземпляр объекта `UserStore`, который наполняю данными и передаю вновь созданному экземпляру объекта `Validator`. Затем я могу подтвердить сочетание имени пользователя и пароля. Когда результаты работы меня полностью удовлетворят, я смогу совсем удалить эти тесты или закомментировать их. Это ужасная растрата ценного ресурса. Однако данные тесты могут сформировать основу для тщательной проверки системы в ходе разработки. Один из инструментов, который может мне помочь, — это PHPUnit.

Знакомство с PHPUnit

PHPUnit — это член семейства инструментов тестирования xUnit. Предшественник этих инструментов — SUnit, каркас, который создал Кент Бек (Kent Beck) для тестирования систем, построенных на языке Smalltalk. Но система xUnit, вероятно, стала популярным инструментом благодаря реализации на Java, jUnit и популярности таких методологий, как экстремальное программирование (Extreme Programming, или XP) и Scrum, в которых очень большое внимание уделяется тестированию.

Нынешняя "реинкарнация" PHPUnit была создана Себастьяном Бергманом (Sebastian Bergmann), который изменил название пакета с PHPUnit2 (автором которого он тоже являлся) в начале 2007 года и переместил его из канала `pear.php.net`

в pear.phpunit.de[1]. Поэтому во время инсталляции вы должны сообщить приложению pear, где искать этот каркас.

```
$ pear config-set auto_discover 1
$ pear install --alldeps phpunit
```

На заметку. Команды, которые вводятся в командной строке, выделяются полужирным шрифтом, чтобы отличать их от выводимых данных.

Создание контрольного примера

Имея такой инструмент, как PHPUnit, мы можем писать тесты для класса UserStore. Тесты для каждого целевого компонента должны объединяться в одном классе, который расширяет PHPUnit_Framework_TestCase. Это один из классов, которые стали доступны благодаря пакету PHPUnit. Вот как создать минимальный класс с контрольным примером.

```
require_once 'PHPUnit/Framework/TestCase.php';

class UserStoreTest extends PHPUnit_Framework_TestCase {
   public function setUp() {
   }

   public function tearDown() {
   }
//...
}
```

Я назвал класс этого контрольного примера UserStoreTest. Конечно, вы не обязаны использовать имя тестируемого класса в названии теста, но так поступают многие разработчики. Соглашения о наименовании подобного рода могут значительно улучшить доступность системы тестов, особенно когда количество компонентов и тестов в системе начнет увеличиваться. Как правило, тесты группируют в каталогах пакетов, которые непосредственно соответствуют тем каталогам, в которых хранятся классы системы. С помощью такой логической структуры можно запускать тест из командной строки, даже не посмотрев, существует ли он! Каждый тест из класса с контрольным примером запускается отдельно от других тестов. Метод setUp() автоматически вызывается для каждого тестового метода, что позволяет создать стабильную и соответствующим образом наполненную среду для теста. Метод tearDown() вызывается после запуска каждого тестового метода. Если тесты изменяют более широкое окружение системы, то можете использовать данный метод, чтобы восстановить исходное состояние. Общая платформа, которой управляют методы setUp() и tearDown(), называется *фиксацией* (fixture).

Чтобы протестировать класс UserStore, нам нужен его экземпляр. Мы можем создать этот экземпляр в методе setUp() и присвоить его свойству. Давайте создадим также тестовый метод.

```
require_once('UserStore.php');
require_once('PHPUnit/Framework/TestCase.php');

class UserStoreTest extends PHPUnit_Framework_TestCase {
```

[1] Канал **pear.phpunit.de** был окончательно закрыт 31 декабря 2014 года. Теперь старая версия приложения PHPUnit устанавливается с помощью PEAR, как показано ниже, а новая просто скачивается с сайта https://phpunit.de/ в виде архива phar. — *Примеч. ред.*

```
    private $store;

    public function setUp() {
        $this->store = new UserStore();
    }

    public function tearDown() {
    }

    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" );
    }
}
```

Методы тестирования следует называть так, чтобы имена начинались со слова `"test"` и не требовали аргументов. Дело в том, что операции с классом контрольного примера осуществляются с помощью рефлексии.

На заметку. Рефлексия подробно описана в главе 5.

Объект, который запускает тесты, просматривает все методы класса и вызывает только те, которые соответствуют этому шаблону (т.е. методы, имена которых начинаются со слова `"test"`).

В данном примере я протестировал извлечение информации о пользователе. Нам не нужно создавать экземпляр `UserStore` для каждого теста, потому что мы это сделали в методе `setUp()`. Поскольку `setUp()` вызывается для каждого теста, свойство `$store` гарантированно содержит вновь созданный экземпляр объекта.

Внутри метода `testGetUser()` мы сначала предоставляем методу `UserStore::addUser()` фиктивные данные, а затем извлекаем эти данные и тестируем каждый их элемент.

Методы с утверждениями

В программировании утверждение (assertion) — это высказывание или метод, позволяющий проверить предположения о некоторых аспектах системы. При использовании утверждения вы обычно определяете некоторое высказывание, например `$cheese is "blue"` или `$pie is "apple"`. Если это высказывание опровергается, то генерируется предупреждение. Утверждения — это настолько хороший способ сделать систему более безопасной, что в некоторых языках программирования существует их естественная и встроенная (inline) поддержка, причем вы можете отключить ее в окончательной версии программы (пример такого языка — Java). PHPUnit поддерживает утверждения с помощью набора статических методов.

В предыдущем примере я использовал наследуемый статический метод `assertEquals()`. Он сравнивает значения двух своих аргументов и проверяет их эквивалентность. Если они не эквивалентны, то метод тестирования будет отмечен как закончившийся неудачей. Имея подкласс `PHPUnit_Framework_TestCase`, мы получаем доступ к набору методов с утверждениями. Эти методы перечислены в табл. 18.1.

Таблица 18.1. Методы с утверждениями класса `PHPUnit_Framework_TestCase`

Метод	Описание
`assertEquals($val1, $val2, $delta, $message)`	Заканчивается неудачей, если `$val1` не эквивалентен `$val2` (`$delta` представляет допустимую погрешность)
`assertFalse( $expression, $message)`	Вычисляет выражение `$expression`. Заканчивается неудачей, если *не* обращается в значение "ложь"
`assertTrue( $expression, $message)`	Вычисляет выражение `$expression`. Заканчивается неудачей, если *не* обращается в значение "истина"
`assertNotNull($val, $message)`	Заканчивается неудачей, если `$val` — нуль
`assertNull($val, $message)`	Заканчивается неудачей, если `$val` отличен от нуля
`assertSame($val1, $val2, $message)`	Заканчивается неудачей, если `$val1` и `$val2` *не* являются ссылками на один и тот же объект или если они — переменные, имеющие разные типы и значения
`assertNotSame($val1, $val2, $message)`	Заканчивается неудачей, если `$val1` и `$val2` являются ссылками на один и тот же объект или если они — переменные, имеющие одинаковые типы и значения
`assertRegExp($regexp, $val, $message)`	Заканчивается неудачей, если `$val` не соответствует регулярному выражению `$regexp`
`assertType($typestring, $val, $message)`	Заканчивается неудачей, если `$val` имеет не тот тип, который описан в `$typestring`
`assertAttributeSame($val, $attribute, $classname, $message)`	Заканчивается неудачей, если `$val` имеет не такие тип и значение, как `$classname::$attribute`
`fail()`	Заканчивается неудачей

Тестирование посредством исключений

Цель программиста обычно заключается в том, чтобы заставить систему *работать*, причем работать хорошо. Как правило, такое отношение переносится и на тестирование, особенно если вы тестируете собственный код. При этом хочется проверить, работает ли метод так, как должен. Поэтому очень легко забыть, насколько важно протестировать его на неудачу. Насколько хорошо работает проверка ошибок в методе? Выдает ли он исключения, когда это необходимо? Выдает ли он правильное исключение? Приводит ли он все в порядок после ошибки, если, например, операция была выполнена наполовину перед тем, как возникла проблема? Ваша роль как тестировщика заключается в том, чтобы все это проверить. К счастью, в этом поможет PHPUnit.

Вот тест, который проверяет работу класса `UserStore`, когда операция завершается неудачно.

```
//...
public function testAddUser_ShortPass() {
   try {
      $this->store->addUser( "bob williams", "bob@example.com", "ff" );
      $this->fail("Ожидалось исключение из-за короткого пароля.");
      } catch ( Exception $e ) {  }
}
//...
```

Если вы снова посмотрите на метод `UserStore::addUser()`, то увидите, что он выдает исключение, если пароль пользователя имеет менее пяти символов. Наш тест пытается это подтвердить. Мы добавляем пользователя с недопустимым паролем

в блоке try. Если выдано ожидаемое исключение, то все в порядке и выполнение программы переходит в блок catch. Если же метод addUser() не генерирует ожидаемое исключение, то блок catch не вызывается, управление программы переходит к следующему оператору, в котором вызывается метод fail().

Еще один способ проверить, выдается ли исключение, — это использовать расширение класса в PHPUnit, которое называется PHPUnit_Extensions_ExceptionTestCase. Оно даст вам метод для проверки утверждений под названием setExpectedException(), требующий имя типа исключения, которое должно быть выдано (либо Exception, либо подкласс). Если метод тестирования заканчивает работу, не выдав правильного исключения, то тест завершается неудачей.

Вот новая реализация предыдущего теста.

```
require_once('PHPUnit/Framework/TestCase.php');
require_once('UserStore.php');

class UserStoreTest extends PHPUnit_Framework_TestCase {
   private $store;

   public function setUp() {
      $this->store = new UserStore();
   }

   public function testAddUser_ShortPass() {
      $this->setExpectedException('Exception');
      $this->store->addUser( "bob williams", "bob@example.com", "ff" );
   }
}
```

Запуск наборов тестов

Если мы тестируем класс UserStore, то должны протестировать и класс Validator. Вот сокращенная версия класса ValidatorTest, в котором тестируется метод Validator::validateUser().

```
require_once('UserStore.php');
require_once('Validator.php');
require_once('PHPUnit/Framework/TestCase.php');

class ValidatorTest extends PHPUnit_Framework_TestCase {
   private $validator;

   public function setUp() {
      $store = new UserStore();
      $store->addUser( "bob williams", "bob@example.com", "12345" );
      $this->validator = new Validator( $store );
   }

   public function tearDown() {
   }

   public function testValidate_CorrectPass() {
      $this->assertTrue(
            $this->validator->validateUser( "bob@example.com", "12345" ),
                  "Ожидалась успешная проверка."
```

```
            );
    }
}
```

Теперь, когда у нас больше одного контрольного примера, возникает вопрос, как запустить их вместе? Лучше всего поместить все тестовые классы в отдельный каталог и назвать его `test`. Затем вы можете указать имя этого каталога, и PHPUnit запустит все тесты, которые в нем находятся.

```
$ phpunit test/
```

```
PHPUnit 3.7.24 by Sebastian Bergmann.
.....
Time: 104 ms, Memory: 3.75Mb

OK (5 tests, 10 assertions)
```

В больших проектах может понадобиться организовать тесты в отдельные подкаталоги. Лучше всего, если структура этих каталогов будет такой же, как и структура ваших пакетов. Тогда при необходимости вы сможете указать имена отдельных пакетов для тестирования.

Ограничения

В большинстве случаев вы будете использовать готовые утверждения в своих тестах. На самом деле вы за один раз можете получить очень много с помощью одного только метода `AssertTrue()`. Но в версии PHPUnit 3.0 в класс `PHPUnit_Framework_TestCase` включен набор фабричных методов, которые возвращают объекты типа `PHPUnit_Framework_Constraint`. Вы можете объединить их и передать методу `PHPUnit_Framework_TestCase::AssertThat()`, чтобы построить собственные утверждения.

Пришло время для небольшого примера. Объект `UserStore` не должен разрешать дублировать добавляемые адреса электронной почты. Вот тест, который позволяет это проверить.

```
class UserStoreTest extends PHPUnit_Framework_TestCase {
    private $store;

    public function setUp() {
        $this->store = new UserStore();
    }

//....
    public function testAddUser_duplicate() {
        try {
            $ret = $this->store->addUser( "bob williams", "a@b.com", "123456" );
            $ret = $this->store->addUser( "bob stevens",  "a@b.com", "123456" );
            self::fail( "Здесь должно быть вызвано исключение." );
        } catch ( Exception $e ) {

          $const = $this->logicalAnd(
                   $this->logicalNot( $this->contains("bob stevens")),
                   $this->isType('array')
                   );
```

```
        self::AssertThat( $this->store->getUser( "a@b.com"), $const );
    }
}
```

Этот тест добавляет пользователя к объекту `UserStore`, а затем добавляет второго пользователя с тем же адресом электронной почты. Тем самым этот тест подтверждает, что исключение выдается при втором вызове метода `addUser()`. В блоке `catch` мы создаем объект ограничений с помощью доступных нам подходящих методов. Они возвращают соответствующие экземпляры класса `PHPUnit_Framework_Constraint`. Давайте разберем это сложное ограничение из предыдущего примера.

```
$this->contains("bob stevens")
```

Оно возвращает объект `PHPUnit_Framework_Constraint_TraversableContains`. Переданный методу `AssertThat`, этот объект сгенерирует ошибку, если тестируемый объект не содержит элемент, соответствующий заданному значению (`"bob stevens"`). Но я выполняю операцию отрицания, передавая это ограничение другому объекту: `PHPUnit_Framework_Constraint_Not`. И снова я использую подходящий метод, доступный через класс `TestCase` (на самом деле через суперкласс `Assert`).

```
$this->logicalNot( $this->contains("bob stevens"))
```

Теперь утверждение `AssertThat` завершится неудачей, если тестируемое значение (которое должно пройти тест) содержит элемент, который соответствует строке `"bob stevens"`. Таким способом можно построить достаточно сложные логические структуры. Ко времени завершения ограничение можно сформулировать так: "Не заканчивать неудачей, если тестируемое значение является массивом и не содержит строку `"bob stevens"`". Таким способом можно создать намного более сложные ограничения. Значение, полученное после запуска ограничения вместе с тестируемым значением, передается методу `AssertThat()`.

Конечно, мы можем достичь этого с помощью стандартных методов утверждения, но у ограничений есть несколько 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. Вы можете создать библиотеку сложных ограничений и использовать их в различных тестах. Вы даже можете объединять сложные ограничения.

```
$const = $this->logicalAnd(
                $a_complex_constraint,
                $another_complex_constraint );
```

В табл. 18.2 показаны некоторые методы ограничений, имеющиеся в классе `TestCase`.

Таблица 18.2. Некоторые методы ограничений класса `TestCase`

Метод класса	Завершается неудачей при несоблюдении условия
`greaterThan($num)`	Тестовое значение больше, чем `$num`
`contains($val)`	Тестовое значение (проходимое) содержит элемент, который соответствует `$val`
`identicalTo($val)`	Тестовое значение — это ссылка на такой же объект, как `$val`, или, для не объектов, имеет такие же тип и значение
`greaterThanOrEqual($num)`	Тестовое значение больше или равно `$num`
`lessThan($num)`	Тестовое значение меньше `$num`

Окончание табл. 18.2

Метод класса	Завершается неудачей при несоблюдении условия
`equalTo($value, $delta=0, $depth=10)`	Тестовое значение равно `$val`. Аргумент `$delta`, если он определен, определяет погрешность для числовых сравнений, а `$depth` определяет глубину вложенности рекурсии при сравнении массивов или объектов
`stringContains($str, $casesensitive=true)`	Тестовое значение содержит `$str`. По умолчанию оно зависит от регистра символов
`matchesRegularExpression( $pattern)`	Тестовое значение соответствует регулярному выражению, указанному в `$pattern`
`logicalAnd( PHPUnit_Framework_Constraint $const, [, $const..])`	Все предоставленные ограничения должны пройти тест
`logicalOr( PHPUnit_Framework_Constraint $const, [, $const..])`	По меньшей мере одно из предоставленных ограничений должно пройти тест
`logicalNot( PHPUnit_Framework_Constraint $const )`	Предоставленное ограничение не проходит

Имитации и заглушки

Цель модульных тестов — протестировать компонент отдельно от содержащей его системы в максимально возможной степени. Но немногие компоненты существуют в вакууме. 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 видели один способ решения подобных проблем. Методы `setUp()` и `tearDown()` можно использовать для управления фиксацией, т.е. общим набором ресурсов для тестов, который может включать соединения с базой данных, конфигурируемые объекты, временные файлы и т.д.

Еще один подход — имитировать контекст тестируемого класса. Для этого нужно создать объекты, которые играют роль объектов, выполняющих реальные функции. Например, вы можете передать сымитированный объект преобразователя базы данных тестовому конструктору объекта. Поскольку этот сымитированный объект использует тип совместно с настоящим классом преобразователя (расширяется от общей абстрактной базы или даже замещает сам настоящий класс), ваш предмет тестирования не заметит подмены. Вы можете заполнить сымитированный объект допустимыми данными. Объекты, которые формируют своего рода "песочницу" для модульных тестов, называются *заглушками (stubs)*. Они могут быть полезны, потому что позволяют сосредоточиться на классе, который нужно протестировать, не тестируя в то же самое время самопроизвольно всю систему.

Однако использование сымитированных объектов этим не ограничивается. Поскольку тестируемые объекты могут каким-то способом вызывать методы сымитированного объекта, вы можете всегда возвратить нужные вам данные. Такое использование сымитированного объекта в качестве "шпиона" называется *проверкой поведения*, и именно это отличает имитацию (mock) от заглушки (stub).

Вы можете проектировать сымитированные объекты самостоятельно, создавая классы, в которых жестко закодированы возвращаемые значения и выдается сообщение о вызовах методов. Это простой процесс, но он отнимает время.

PHPUnit предоставляет доступ к более простому и динамичному решению. Он будет "на лету" генерировать для вас сымитированные объекты. PHPUnit делает это, изучая класс, который вы хотите имитировать, и создавая дочерний класс, замеща-

ющий его методы. Получив этот сымитированный экземпляр, вы можете вызывать для него методы, чтобы заполнить его данными и определить условия прохождения теста.

Давайте создадим пример. Класс `UserStore` содержит метод `notifyPassword Failure()`, который устанавливает значение поля для заданного пользователя. Этот метод должен вызываться объектом `Validator` при вводе пользователем неправильного пароля. Здесь я сымитировал класс `UserStore`, так что он предоставляет данные объекту `Validator` и подтверждает, что его метод `notifyPasswordFailure()` был вызван, как ожидалось.

```
class ValidatorTest extends PHPUnit_Framework_TestCase {
   //...
   public function testValidate_FalsePass() {
      $store = $this->getMock("UserStore");
      $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(array("name"=>"bob@example.com",
                                             "pass"=>"right")));

      $this->validator->validateUser("bob@example.com", "wrong");
   }
}
```

Сымитированные объекты используют *текучий интерфейс* (*fluent interface*), т.е. языкоподобную структуру. Их намного легче использовать, чем описать. Такие конструкции работают слева направо, каждый вызов возвращает ссылку на объект, который затем может быть вызван с помощью вызова следующего модифицирующего метода (который сам возвращает объект). Это облегчает использование, но затрудняет отладку.

В предыдущем примере я вызвал метод `PHPUnit_Framework_TestCase::getMock()`, передав ему `"UserStore"`, имя класса, который я хочу сымитировать. Метод динамически сгенерирует класс и создаст экземпляр его объекта. Я сохраняю этот сымитированный объект в свойстве `$store` и передаю его объекту `Validator`. Это не приводит к ошибке, потому что вновь созданный класс объекта расширяет `UserStore`. Я обманул `Validator`, заставив его принять к себе "шпиона".

У сымитированных объектов, сгенерированных PHPUnit, есть метод `expects()`. Этому методу требуется передать сравнивающий объект (на самом деле он относится к типу `PHPUnit_Framework_MockObject_Matcher_Invocation`, но не волнуйтесь; вы можете использовать подходящие методы в `TestCase`, чтобы сгенерировать этот объект). Сравнивающий объект определяет мощность ожидания, т.е. сколько раз метод должен быть вызван.

В табл. 18.3 показаны методы сравнивающего объекта, имеющиеся в классе `TestCase`.

Таблица 18.3. Некоторые методы сравнивающего объекта

Метод `TestCase`	Завершается неудачей при несоблюдении условия
`any( )`	Сделано нуль или больше вызовов соответствующего метода (полезно для объектов-заглушек, которые возвращают значения, но не тестируют вызовы)

Окончание табл. 18.3

Метод TestCase	Завершается неудачей при несоблюдении условия
never( )	Не сделан ни один вызов соответствующего метода
atLeastOnce( )	Сделан один или больше вызовов соответствующего метода
once( )	Сделан единственный вызов соответствующего метода
exactly($num)	Сделано $num вызовов соответствующего метода
at($num)	Вызов соответствующего метода сделан по индексу $num (каждый вызов метода для сымитированного объекта записывается и индексируется)

Определив требование соответствия, я должен указать метод, к которому оно применяется. Например, `expects()` возвращает объект (`PHPUnit_Framework_MockObject_Builder_InvocationMocker`, если вы знаете), у которого есть метод `method()`. Я могу просто вызвать его по имени метода. Достаточно сделать небольшую имитацию.

```
$store = $this->getMock("UserStore");
$store->expects( $this->once() )
      ->method('notifyPasswordFailure');
```

Но я должен пойти дальше и проверить параметры, переданные `notifyPasswordFailure()`. Метод `InvocationMocker::method()` возвращает экземпляр объекта, для которого он был вызван. В класс `InvocationMocker` включен метод `with()`, которому передается переменный список параметров для сравнения. Ему также передаются объекты ограничений, так что можно тестировать диапазоны и т.д. Имея это все на вооружении, мы можем закончить оператор и гарантировать, что ожидаемый параметр передан методу `notifyPasswordFailure()`.

```
$store->expects($this->once() )
      ->method('notifyPasswordFailure')
      ->with( $this->equalTo('bob@example.com') );
```

Сейчас вы поймете, почему эта конструкция называется текучим интерфейсом. Этот оператор можно сформулировать в виде простой фразы: "Объект `$store` *ожидает* единственный вызов, чтобы уведомить метод `notifyPasswordFailure()` с параметром `bob@example.com`".

Обратите внимание, что я передал ограничение методу `with()`. На самом деле это лишнее — любые явные аргументы преобразуются в ограничения внутри, так что я могу написать оператор следующим образом.

```
$store->expects($this->once() )
      ->method('notifyPasswordFailure')
      ->with( 'bob@example.com' );
```

Иногда сымитированные объекты PHPUnit используются только как заглушки, т.е. объекты, возвращающие значения, которые позволяют тестам работать. В таких случаях вы можете вызвать `InvocationMocker::will()` из вызова метода `method()`. Метод `will()` требует возвращаемое значение (или значения, если метод вызывается несколько раз), которое должен был вернуть связанный с ним метод. Вы можете передать это возвращаемое значение, вызвав либо `TestCase::returnValue()`, либо `TestCase::onConsecutiveCalls()`. И снова это намного проще сделать, чем описать. Вот фрагмент одного из предыдущих примеров, в котором я заполнил объект `UserStore` для возврата значения.

```
$store->expects( $this->any() )
      ->method("getUser")
      ->will( $this->returnValue(
```

```
            array( "name"=>"bob williams",
                   "mail"=>"bob@example.com",
                   "pass"=>"right")));
```

Я наполнил сымитированный объект `UserStore` данными, ожидая произвольное количество вызовов к `getUser()`, — сейчас меня волнует вопрос предоставления данных, а не тестирования вызовов. Затем я вызываю `will()` с результатом вызова `TestCase::returnValue()` с данными, которые я хочу вернуть (случилось так, что это объект `PHPUnit_Framework_MockObject_Stub_Return`, так что на вашем месте я бы просто вспомнил подходящий метод, который использовался для его получения).

Существует альтернативный вариант: передать результат вызова метода `TestCase::onConsecutiveCalls()` методу `will()`. Ему можно передать произвольное число параметров, каждый из которых будет возвращен сымитированным методом при последующих вызовах.

Тесты, заканчивающиеся неудачей и достигающие цели

Большинство людей согласятся, что тестирование — это хорошая вещь. Но по-настоящему вы его оцените только после того, как оно несколько раз спасет вас от неприятностей. Давайте смоделируем ситуацию, когда изменение в одной части системы приведет к неожиданному эффекту в другой.

Наш класс `UserStore` уже работал некоторое время, когда в ходе анализа кода все пришли к выводу, что для класса лучше генерировать объекты типа `User`, а не ассоциативные массивы. Вот новая версия класса.

```
class UserStore {
   private $users = array();

   function addUser( $name, $mail, $pass ) {
      if ( isset( $this->users[$mail] ) ) {
         throw new Exception(
            "Пользователь {$mail} уже зарегистрирован.");
      }
      $this->users[$mail] = new User( $name, $mail, $pass );
      return true;
   }

   function notifyPasswordFailure( $mail ) {
      if ( isset( $this->users[$mail] ) ) {
         $this->users[$mail]->failed(time());
      }
   }

   function getUser( $mail ) {
      if ( isset( $this->users[$mail] ) ) {
         return ( $this->users[$mail] );
      }
      return null;
   }
}
```

Вот простой класс `User`.

```
class User {
   private $name;
   private $mail;
```

```
    private $pass;
    private $failed;

    function __construct( $name, $mail, $pass ) {
        if ( strlen( $pass ) < 5 ) {
            throw new Exception(
                "Длина пароля должна быть не менее 5 символов.");
        }
        $this->name = $name;
        $this->mail = $mail;
        $this->pass = $pass;
    }
    function getName() {
        return $this->name;
    }

    function getMail() {
        return $this->mail;
    }

    function getPass() {
        return $this->pass;
    }

    function failed( $time ) {
        $this->failed = $time;
    }
}
```

Разумеется, мы должны изменить класс `UserStoreTest` в соответствии с этими изменениями. Поэтому предназначенный для работы с массивом код

```
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" );
    //...
```

преобразуется в код, предназначенный для работы с объектом.

```
public function testGetUser() {
    $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 набор тестов, то получаем предупреждение о том, что работа еще не сделана.

```
$ phpunit test/
```

```
PHPUnit 3.7.24 by Sebastian Bergmann.
....FF

Time: 359 ms, Memory: 3.75Mb

There were 2 failures:
1) ValidatorTest::testValidate_CorrectPass
```

```
Expecting successful validation
Failed asserting that false is true.
/.../test/ValidatorTest.php:22
2) ValidatorTest::testValidate_FalsePass

Expectation failed for method name is equal to
<string:notifyPasswordFailure> when invoked 1 time(s).
Method was expected to be called 1 times, actually called 0 times.

FAILURES!

Tests: 6, Assertions: 5, Failures: 2.
```

Существует проблема с классом `ValidatorTest`. Давайте еще раз посмотрим на метод `Validator::validateUser()`.

```
public function validateUser( $mail, $pass ) {
   if ( ! is_array($user = $this->store->getUser( $mail )) ) {
      return false;
   }
   if ( $user['pass'] == $pass ) {
      return true;
   }
   $this->store->notifyPasswordFailure( $mail );
   return false;
}
```

Мы вызываем метод `getUser()`. Хотя `getUser()` теперь возвращает объект, а не массив, наш метод не генерирует предупреждение. Метод `getUser()` первоначально возвращал запрошенный пользовательский массив (в случае успеха) или `null` (в случае неудачи), так что мы проверяли допустимость пользователей путем проверки наличия массива с помощью функции `is_array()`. Теперь, конечно, `getUser()` возвращает объект, а метод `validateUser()` всегда будет возвращать значение "ложь". Без нашего тестового каркаса объект `Validator` просто отвергал бы всех пользователей как недопустимых, не выдавая при этом никакого предупреждения.

А теперь представьте, что вы сделали это небольшое изменение в пятницу вечером без соответствующего тестирования. И теперь вас атакуют возмущенными посланиями, которые не дают вам спокойно расслабиться дома, в баре или ресторане: "Что вы сделали? Все наши клиенты заблокированы!"

Самые коварные ошибки — те, которые не заставляют интерпретатор сообщить, что что-то идет не так. Они скрываются за идеально допустимым кодом и незаметно нарушают логику системы. Причем многие ошибки не проявляются в том месте, куда вы внесли изменения. Там находится их причина, но эффект может проявиться в любом другом месте через несколько дней и даже недель. Тестовый каркас поможет обнаружить по меньшей мере некоторые из них и тем самым предотвратить, а не обнаружить проблемы в системах.

Пишите тесты по мере создания нового кода и запускайте их часто. Если кто-то сообщает об ошибке, сначала добавьте тест к каркасу, чтобы подтвердить это, а затем исправьте ошибку, чтобы тест прошел, — у ошибок есть странная привычка снова появляться в том же месте. Написание тестов для того, чтобы подтвердить ошибки, а затем защитить исправление от последующих проблем, называется *регрессионным тестированием*. Кстати, если у вас есть отдельный каталог для регрессионных тестов, не забывайте давать файлам содержательные имена. В одном проекте наша команда решила назвать регрессионные тесты по номерам ошибок.

В результате у нас получился каталог, содержащий 400 файлов тестов, у каждого из которых было имя типа `test_973892.php`. Представляете, во что превратился поиск отдельного теста?

Написание тестов веб-приложений

Вы должны проектировать свое веб-приложение так, чтобы его можно было легко вызвать как из командной строки, метода API, так и из веб-сервера. В главе 12 я уже описывал несколько приемов, которые могут вам в этом помочь. В частности, если все данные HTTP-запроса сохранять в классе `Request`, то его экземпляр можно легко и просто заполнить данными при вызове приложения из командной строки или из списка аргументов, переданных методу. Тогда ваше приложение сможет работать в любой среде.

Если ваше приложение не так просто запустить в разных исполняющих средах, значит, при его проектировании был допущен изъян. Если, например, пути к многочисленным файлам жестко закодированы в ваших компонентах, то, вероятнее всего, вы столкнулись со случаем тесной связи. В таком случае вы должны переместить элементы, жестко привязанные к их контексту, в инкапсулирующие их объекты, которые можно запросить из центрального хранилища данных. В главе 12 был также описан шаблон Registry, который вам в этом может помочь.

Таким образом, если ваше веб-приложение можно легко запустить, непосредственно вызвав один из методов, вы с удивлением обнаружите, что для него достаточно просто написать высокоуровневые веб-тесты без привлечения дополнительных средств.

Тем не менее вы можете столкнуться с тем, что для запуска таких тестов даже в тщательно продуманных проектах нужно будет выполнить небольшой рефакторинг. По своему опыту могу сказать, что подобное всегда приводит к улучшению в структуре проекта. Я собираюсь это продемонстрировать, подогнав один из аспектов примера WOO, описанного в главах 12 и 13, под модульное тестирование.

Рефакторинг веб-приложения для выполнения тестов

Мы фактически оставили разработку примера приложения WOO в состоянии, пригодном для тестирования. Поскольку в нем использовался унифицированный шаблон Front Controller, наше приложение обладает простым интерфейсом API. Ниже приведен пример простого класса, называемого `Runner.php`.

```
require_once( "woo/controller/Controller.php");

\woo\controller\Controller::run();
```

К нему очень легко можно добавить модульный тест, правда? Но вот что делать с передаваемыми в строке запроса параметрами? В некоторой степени эта проблема уже решена с помощью класса `Request`.

```
// \woo\controller\Request

function init() {
   if ( isset( $_SERVER['REQUEST_METHOD'] ) ) {
      $this->properties = $_REQUEST;
      return;
   }

   foreach( $_SERVER['argv'] as $arg ) {
```

```
      if ( strpos( $arg, '=' ) ) {
         list( $key, $val ) = explode( "=", $arg );
         $this->setProperty( $key, $val );
      }
   }
}
```

В методе `init()` проверяется, в каком контексте работает приложение (серверном или в командной строке), и соответствующим образом заполняется массив `$properties` (либо непосредственным присваиванием, либо через вызов метода `setProperty()`). В результате наше приложение прекрасно работает при его вызове из командной строки. Это значит, что я могу ввести команду

```
$ php runner.php cmd=AddVenue venue_name=bob
```

и в ответ получить следующее.

```
<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 (5)</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="5" />
   <input type="submit" value="submit" />
</form>
</body>
</html>
```

И хотя наше приложение работает из командной строки, все еще тяжело передать ему параметры при запуске через вызов метода. Одно из грубых и неэлегантных решений заключается в том, что перед вызовом метода `run()` контроллера нужно вручную сформировать массив параметров `$argv`. Неудивительно, что оно мне не очень нравится. При непосредственной работе с загадочными массивами возникает чувство, что что-то идет не так, а операции со строками, которые нужно проводить в любом случае, могут привести к ошибке. Поэтому давайте посмотрим на класс контроллера более внимательно. При этом я вижу возможность улучшения как структуры приложения, так и его возможностей к тестированию. Ниже приведен фрагмент метода `handleRequest()`.

```
// \woo\controller\Controller

function handleRequest() {
   $request = \woo\base\ApplicationRegistry::getRequest();
   $app_c = \woo\base\ApplicationRegistry::appController();
```

```
    while( $cmd = $app_c->getCommand( $request ) ) {
        $cmd->execute( $request );
    }

    \woo\domain\ObjectWatcher::instance()->performOperations();
     $this->invokeView( $app_c->getView( $request ) );
}
```

Этот метод предназначен для вызова из статического метода `run()`. Обратите внимание на то, что экземпляр объекта `Request` не создается напрямую, а запрашивается из реестра `ApplicationRegistry`. Поскольку в реестре хранится единственный экземпляр объекта `Request`, я могу получить ссылку на него и загрузить его данными, необходимыми для выполнения тестирования, перед запуском системы путем вызова метода контроллера. Точно так я могу эмулировать получение веб-запроса. Поскольку в моей системе используется объект `Request` в качестве интерфейса к данным, полученным из веб-запроса, она не будет тесно привязана к источнику данных. Поскольку мы используем объект `Request`, нашей системе будет безразлично, откуда поступают данные, — из теста или с веб-сервера. Я выработал для себя общий принцип, согласно которому нужно всегда стараться помещать экземпляры объектов в системный реестр там, где это возможно. В результате чуть позже я смогу расширить реализацию статического метода-фабрики `ApplicationRegistry::instance()` для нашего случая так, чтобы он возвращал сымитированный объект системного реестра, заполненный фиктивными данными. Для этого достаточно будет задать всего один флаг, чтобы создать полностью фиктивное окружение для нашего приложения. Как мне нравится обманывать свою систему!

Однако пока что для начала я опишу один из самых традиционных способов предварительной загрузки объекта `Request` данными для тестирования.

Простые веб-тесты

Ниже приведен тестовый пример, который выполняет очень простой тест веб-приложения WOO.

```
class AddVenueTest extends PHPUnit_Framework_TestCase {

    function testAddVenueVanilla() {
        $this->runCommand("AddVenue", array("venue_name"=>"bob") );
    }

    function runCommand( $command=null, array $args=null ) {
        $request = \woo\base\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();
    }
}
```

По сути, этот тест ничего особенного не тестирует, а только доказывает тот факт, что наше приложение может быть вызвано из метода. Нечто реальное выполняется в методе `runCommand()`. Здесь ничего непонятного нет. Сначала у объекта `ApplicationRegistry` запрашивается объект `Request`, а затем он заполняется данными в виде ключей и значений, переданных методу в качестве параметров. Поскольку объект `Controller` будет обращаться к тому же источнику для получения объекта `Request`, я могу быть уверенным, что он будет работать с заданными мною данными.

Запуск этого теста подтверждает, что все идет хорошо! Я вижу то, что ожидаю увидеть. Однако проблема состоит в том, что данные приложения выводятся для визуального восприятия и поэтому их тяжело тестировать. Я могу устранить эту проблему очень легко, выполнив буферизацию выходных данных.

```
class AddVenueTest extends PHPUnit_Framework_TestCase {

   function testAddVenueVanilla() {
      $output = $this->runCommand("AddVenue", array("venue_name"=>"bob") );
      self::AssertRegexp( "/added/", $output );
   }

   function runCommand( $command=null, array $args=null ) {
      ob_start();
      $request = \woo\base\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 приложения в буфер, я смогу вернуть его из метода `runCommand()`. Чтобы это продемонстрировать, к возвращаемым данным я применил простое утверждение.

Вот что происходит при вызове из командной строки.

```
$ phpunit test/AddVenueTest.php
```

```
PHPUnit 3.7.24 by Sebastian Bergmann.
Time: 215 ms, Memory: 3.25Mb
OK (1 test, 1 assertion)
```

Если вы планируете запускать в системе многочисленные тесты описанным выше способом, то имеет смысл создать суперкласс для веб-интерфейса пользователя и поместить в него метод `runCommand()`.

Здесь я упустил ряд деталей, с которыми вам предстоит столкнуться при выполнении собственных тестов. Вы должны сделать так, чтобы система могла работать с изменяемыми в процессе конфигурации источниками данных. Очевидно, что не стоит выполнять тесты на том же хранилище данных, которое используется в процессе разработки приложения. Вот и еще один подходящий случай для улучшения структуры приложения. Поищите в своей программе жестко закодированные имена и пути к файлам, а также строки, определяющие источники данных DSN, и вынесите их в объект `Registry`. Затем, чтобы ваши тесты выполнялись только в пределах одной "песочницы", задайте все требуемые значения для тестового примера в методе `setUp()`. Рассмотрите вопрос замены объекта `ApplicationRegistry` на `MockRequestRegistry`, который обеспечит ваше приложение заглушками, сымитированными объектами и другими поддельными штучками.

Описанные выше подходы отлично работают при тестировании входных и выходных данных веб-приложения. Однако при этом существуют некоторые явные ограничения. Этот метод никак не заменит нам браузер. И если в вашем приложении используются JavaScript, Ajax и другие интеллектуальные клиентские технологии, тестирование текста, сгенерированного вашей системой, не сможет нам ответить на вопрос, увидел ли пользователь на экране что-то вразумительное.

К счастью, решение существует!

Знакомство с Selenium

Система Selenium (`http://seleniumhq.org/`) состоит из набора команд (иногда их называют селенами), которые позволяют определить тесты для веб-приложения. Кроме того, в ней есть средства и подходящий API для создания и запуска тестов для браузера, а также средства для привязки тестов к существующим платформам тестирования.

В этом кратком введении я создам быстрый тест системы WOO, которая была описана в главе 12. Тест будет работать в связке с Selenium Server через API, которое называется веб-драйвером для PHP (php-webdriver).

Получение и установка Selenium

Компоненты Selenium можно загрузить с веб-сервера по адресу: `http://seleniumhq.org/download/`. Для выполнения нашего примера мне понадобится Selenium Server.

После загрузки пакета у вас будет файл архива с именем `selenium-server-standalone-2.45.0.jar` (к моменту чтения вами книги его версия уже может быть другой). Скопируйте этот файл куда-нибудь в центральное хранилище. Чтобы двинуться дальше, вам понадобится установить на своем компьютере Java. Как только вы это сделаете, то сможете запустить Selenium Server.

В приведенном ниже примере я копирую файл с сервером в подкаталог `/usr/local/lib` и запускаю его.

```
$ cp server-standalone-2.45.0.jar /usr/local/lib
$ java -jar /usr/local/lib/server-standalone-2.45.0.jar
```

```
13:15:08.323 INFO - Launching a standalone server
13:15:08.459 INFO - Java: Oracle Corporation 25.40-b25
13:15:08.460 INFO - OS: FreeBSD 11.0-CURRENT amd64
13:15:08.490 INFO - v2.45.0, with Core v2.45.0. Built from revision 5017cb8
13:15:08.631 INFO - Default driver org.openqa.selenium.
```

```
ie.InternetExplorerDriver registration is skipped: registration
capabilities Capabilities [{ensureCleanSession=true, browserName=internet
explorer, version=, platform=WINDOWS}] does not match with current
platform: UNIX
13:15:08.676 INFO - RemoteWebDriver instances should connect to:
http://127.0.0.1:4444/wd/hub
13:15:08.677 INFO - Version Jetty/5.1.x
13:15:08.678 INFO - Started HttpContext[/selenium-server,/selenium-server]
13:15:08.852 INFO - Started org.openqa.jetty.jetty.servlet.
ServletHandler@2a18f23c
13:15:08.852 INFO - Started HttpContext[/wd,/wd]
13:15:08.852 INFO - Started HttpContext[/selenium-server/driver,/selenium-
server/driver]
13:15:08.852 INFO - Started HttpContext[/,/]
13:15:08.868 INFO - Started SocketListener on 0.0.0.0:4444
13:15:08.868 INFO - Started org.openqa.jetty.jetty.Server@b1bc7ed
```

Обратите внимание на то, что при запуске программа сообщает нам URL, по которому можно будет слать запросы к серверу. Этот URL нам понадобится чуть позже.

Теперь все готово для тестирования.

PHPUnit и Selenium

Несмотря на то что в PHPUnit реализовано API для работы с Selenium, на момент написания этих строк в него постоянно вносились изменения. Надежно и стабильно работала только поддержка возможностей, реализованных в Selenium 1. Как вы уже, наверное, заметили в предыдущем разделе, Selenium Server имеет текущую версию 2.45. А это означает, что по умолчанию в PHPUnit обеспечиваются полная интеграция и доступ только к ограниченным функциональным возможностям Selenium Server.

PHPUnit обеспечивает лишь неполную поддержку для WebDriver API (интерфейс ко всем тестовым возможностям Selenium 2) через класс `PHPUnit_Extensions_Selenium2TestCase`. Однако на момент написания книги данный код не работал в системе Linux. Этот факт являлся основной проблемой при развертывании многих автоматизированных систем тестирования (речь не идет о производственных системах), работающих на платформе Linux.

Поэтому, чтобы получить доступ к максимально возможному количеству функциональных возможностей Selenium, имеет смысл воспользоваться PHPUnit, работающим в связке со средством, в котором уже реализованы все нужные нам привязки.

Общие сведения о веб-драйвере для PHP

Веб-драйвер (WebDriver) представляет собой механизм, появившийся в Selenium 2, посредством которого Selenium может управлять браузером. Разработчики Selenium предусмотрели API веб-драйвера для таких языков, как Java, Python и C#. Что касается PHP, то для него доступно сразу несколько API. Я выбрал для использования веб-драйвер под названием php-webdriver, который был создан разработчиками Facebook. В настоящий момент он находится в стадии активной разработки и соответствует официальному API.

Веб-драйвер php-webdriver размещен на GitHub, поэтому его можно загрузить с помощью команды клонирования хранилища, как показано ниже.

```
$ git clone git@github.com:facebook/php-webdriver.git
```

Эта команда создаст каталог под названием `php-webdriver`. Поместите его в список поиска классов PHP, и все будет готово к использованию.

Создание тестового каркаса

Я собираюсь тестировать экземпляр приложения WOO, запущенного на моем компьютере и откликающегося по URL `http://localhost/webwoo`.

Начнем с шаблона текстового класса.

```
require_once('PHPUnit/Framework/TestCase.php');
require_once( "php-webdriver/lib/__init__.php" );

class seleniumtest extends PHPUnit_Framework_TestCase {
   protected function setUp() {
   }

   public function testAddVenue() {
   }
}
```

Здесь я включил в него файл `PHPUnitFramework/TestCase.php` как обычно. Я также включил файл `php-webdriver/lib/__init__.php`, который, в свою очередь, включает все другие необходимые для работы с API файлы. Теперь нам нужно убедиться, что мой пустой тестовый пример работает.

```
$ phpunit seleniumtest.php
```

```
PHPUnit 3.7.24 by Sebastian Bergmann.
Time: 7 ms, Memory: 3.50Mb
OK (1 test, 0 assertions)
```

Все идет хорошо, поэтому самое время сделать так, чтобы наш тест выполнял что-нибудь полезное.

Подключение к серверу Selenium

Вспомните, что при запуске сервер Selenium выводит на консоль URL, по которому можно к нему подключиться. Для подключения к серверу из тестов мне нужно указать этот URL и массив характеристик (capabilities array) в конструкторе класса `RemoteWebDriver`.

```
require_once('PHPUnit/Framework/TestCase.php');
require_once( "php-webdriver/lib/__init__.php" );

class seleniumtest extends PHPUnit_Framework_TestCase {

   protected function setUp() {
      $host = "http://127.0.0.1:4444/wd/hub";
      $capabilities = array(WebDriverCapabilityType::BROWSER_NAME => 'firefox');
      $this->driver = new RemoteWebDriver($host, $capabilities);
   }

   public function testAddVenue() {
   }
}
```

С полным списком характеристик можно ознакомиться в файле класса `remote/RemoteWebDriverCapablityType.php`. Однако для моих демонстрационных целей мне достаточно указать только название браузера. Строку `$host`, содержащую URL подключения, и массив характеристик `$capabilities` я передал конструктору класса `RemoteWebDriver`. Ссылка на созданный объект была сохранена в свойстве `$driver`. При запуске теста мы должны увидеть, как сервер Selenium открывает новое окно браузера перед выполнением тестирования.

Написание теста

Я собираюсь протестировать простую последовательность действий, выполняемых пользователями в моем приложении. Мне нужно перейти на страницу `AddVenue`, добавить заведение для проведения мероприятия и затем — место проведения. В процессе этого нужно взаимодействовать с тремя веб-страницами.

Ниже приведен мой тест.

```
public function testAddVenue() {
   $this->driver->get("http://localhost/webwoo/?cmd=AddVenue");

   $venel = $this->driver->findElement( WebDriverBy::name("venue_name") );
   $venel->sendKeys( "my_test_venue" );
   $venel->submit();

   $tdel = $this->driver->findElement( WebDriverBy::xpath("//td[1]") );
   $this->assertRegexp( "/'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->assertRegexp( "/'my_test_space' added/", $el->getText() );
}
```

При запуске этого теста из командной строки произойдет следующее:

```
$ phpunit seleniumtest3.php

PHPUnit 3.7.24 by Sebastian Bergmann.
Time: 8.75 seconds, Memory: 3.75Mb
OK (1 test, 2 assertions)
```

Разумеется, это не все, что происходит на самом деле. Сервер Selenium откроет также окно браузера и выполнит в нем указанные мною действия. Должен признаться, что получающийся эффект меня несколько пугает!

Давайте теперь проанализируем сам код. Сначала в нем вызывается метод `WebDriver::get()`, который запрашивает начальную страницу моего приложения. Обратите внимание, что в параметре метода нужно указывать полный URL нашего приложения. Это позволяет запускать его не обязательно на том же физическом сервере, что и Selenium. Сервер Selenium загрузит указанную страницу в открытое окно запущенного им браузера, как показано на рис. 18.1.

После загрузки страницы к ней можно получить доступ через WebDriver API. Ссылку на элемент страницы можно получить с помощью вызова метода `RemoteWebDriver::findElement()`. Для этого нам потребуется передать ему объект типа `WebDriverBy`. В классе `WebDriverBy` предусмотрен набор методов-фабрик, каждый из

которых возвращает объект типа `WebDriverBy`, сконфигурированный согласно выбранному нами способу поиска конкретного элемента. Создавая форму на веб-странице, я задал значение атрибута `name` для ее элемента равным `"venue_name"`. Поэтому для поиска этого элемента формы мне нужно использовать метод `WebDriverBy::name()` и возвращенный им объект передать в качестве параметра методу `findElement()`. В табл. 18.4 приведен список всех доступных методов-фабрик.

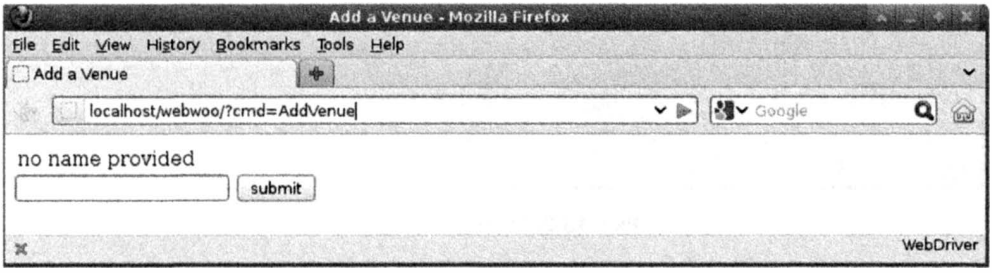


Рис. 18.1. Страница `AddVenue`, загруженная сервером Selenium

Таблица 18.4. Методы-фабрики класса `WebDriverBy`

Метод	Описание
`className()`	Ищет элементы по имени класса CSS
`cssSelector()`	Ищет элементы по селектору CSS
`id()`	Ищет элементы по значению атрибута `id` дескриптора HTML
`name()`	Ищет элементы по значению атрибута `name` дескриптора HTML
`linkText()`	Ищет элементы по тексту, указанному в их гиперссылке
`partialLinkText()`	Ищет элементы по фрагменту текста, указанному в их гиперссылке
`tagName()`	Ищет элементы по их дескриптору HTML
`xpath()`	Ищет элементы, которые соответствуют выражению Xpath

После получения ссылки на объект типа `RemoteWebElement`, соответствующий элементу формы с именем `"venue_name"`, я могу воспользоваться его методом `sendKeys()` для установки значения. Важно отметить, что метод `sendKeys()` присваивает значение элементу формы и не делает ничего более. Кроме того, он эмулирует процесс набора текста в поле элемента управления формы. Эту полезную особенность можно использовать для тестирования кода JavaScript, перехватывающего события от клавиатуры.

После установки значения элемента формы она отправляется на обработку серверу. Об этом позаботится само API. После вызова метода `submit()` для объекта элемента формы сервер Selenium находит нужную форму и отправляет ее на обработку веб-серверу.

Разумеется, что отправка формы веб-серверу вызовет загрузку новой веб-страницы. Поэтому на следующем шаге я проверяю, все ли прошло так, как ожидалось. И снова я воспользовался методом `WebDriver::findElement()`, хотя на этот раз передал ему в качестве параметра объект типа `WebDriverBy`, сконфигурированный для выражения Xpath. Если поиск завершится удачно, метод `findElement()` возвратит новый объект типа `RemoteWebElement`. Если же поиск не увенчается успехом, генерируется исключение, и процесс тестирования останавливается. Будем считать, что все прошло успешно, и мне удалось получить значение элемента с помощью вызова метода `RemoteWebElement::getText()`.

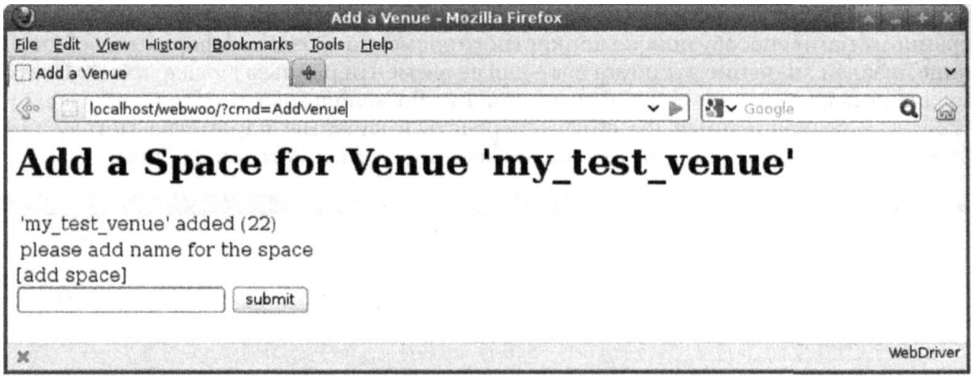


Рис. 18.2. Страница `AddSpace`

На этом этапе мы отправили форму на обработку веб-серверу и проанализировали данные на веб-странице, возвращенной в ответ сервером, как показано на рис. 18.2.

Теперь нам осталось снова ввести данные в форму, отправить ее серверу и проанализировать новую страницу. Для этой цели используется та же методика, которая была описана выше.

Разумеется, в этой главе я смог описать систему Selenium лишь поверхностно. Но, я надеюсь, вам этого было вполне достаточно, чтобы понять саму идею и возможности программы. За более полными сведениями по Selenium обратитесь к руководству по этой программе по адресу `http://seleniumhq.org/docs/index.html`. Вам также пригодится документация по Selenium, которая находится на сайте PHPUnit по адресу `https://phpunit.de/manual/current/en/selenium.html`.

Несколько предупреждений

Легко увлечься преимуществами автоматических тестов. Я добавляю модульные тесты к своим проектам, а также использую PHPUnit для функциональных тестов. Одним словом, я тестирую на уровне системы, а также на уровне класса. Я видел реальные и заметные преимущества, но считаю, что за все приходится платить.

Тесты увеличивают стоимость разрабатываемого проекта. Например, если вы встраиваете в проект систему безопасности, то должны предусмотреть и дополнительное время, затрачиваемое на ее построение, которое может повлиять на выпуск новых версий. Время, которое тратится на написание и запуск тестов, является составной частью этого времени. В одной системе могут быть наборы функциональных тестов, которые запускаются для нескольких баз данных и нескольких систем контроля версий. Добавьте сюда еще несколько контекстуальных переменных, и мы столкнемся с реальной преградой для запуска набора тестов. Вполне понятно, что от тестов, которые не запускаются, пользы никакой. Одно из решений данной проблемы — полностью автоматизировать тесты, чтобы они запускались автоматически по расписанию с помощью такого приложения, как `cron`. Другое решение — поддерживать подмножество тестов, которые могут легко запускать разработчики при фиксации кода в хранилище. Такое подмножество тестов может существовать наряду с более длинным и медленным тестом.

Вопрос, на который нужно обратить внимание, — это хрупкая природа многих тестовых программ. Тесты могут придать вам уверенность при внесении изменений, но когда, по мере увеличения сложности системы, количество тестов тоже уве-

личится, то несколько тестов могут заканчиваться неудачей. Конечно, во многих случаях это хорошо. Ведь вам нужно знать, когда не происходит ожидаемого поведения и когда имеет место поведение, которого вы не ожидаете.

Но часто тестовая программа может закончиться неудачей из-за относительно малого изменения, например при изменении строки с адресом обратной связи. Каждый закончившийся неудачей тест — это проблема, которой нужно заниматься срочно. Но нелегко изменить 30 контрольных примеров в связи с незначительным изменением архитектуры или вывода. Модульные тесты менее подвержены проблемам подобного рода, поскольку, как правило, они фокусируются на каждом компоненте в отдельности.

Усилия по поддержанию тестов на одном уровне с развивающейся системой — это цена, которую придется платить. Но я считаю, что преимущества оправдывают эти затраты.

Можно также предпринять некоторые шаги по снижению хрупкости тестовой программы. Имеет смысл писать тесты, предусматривающие возможность некоторых изменений. Например, для тестирования вывода я обычно использую регулярные выражения, а не прямую проверку полной эквивалентности. Тогда при тестировании нескольких ключевых слов маловероятно, что тест закончится неудачей, если я удалю символ разделителя строк из строки вывода. Конечно, делать тесты слишком "снисходительными" тоже опасно, так что решать вам.

Еще один вопрос — степень, в которой следует использовать фальшивые объекты и заглушки, чтобы имитировать систему за пределами компонента, который нужно протестировать. Некоторые считают, что необходимо изолировать компонент, насколько это возможно, и сымитировать все вокруг него. В одних проектах у меня это срабатывало. Но в других я обнаружил, что на поддержание системы имитаций тратится очень много времени. Вам придется не только обеспечивать, чтобы тесты "шли в ногу" с системой, но и обновлять фальшивые объекты. А теперь представьте, что изменился возвращаемый тип метода. Если вы не обновите метод соответствующего объекта-заглушки, чтобы он возвращал новый тип, то клиентские тесты могут стать источником ошибки. Если имитируется сложная система, то существует реальная опасность того, что в сымитированные объекты вкрадутся ошибки. Отладка тестов — это утомительная работа, особенно если в самой системе ошибок нет.

В этом деле я стараюсь использовать интуицию. Обычно я использую имитации и заглушки, но без колебаний перехожу к реальным компонентам, если приходится прилагать все больше усилий. Сосредоточенность на тестировании уменьшится, но зато появится следующее преимущество: вы будете знать, что ошибки, возникающие в контексте компонента, — это по меньшей мере реальные проблемы системы. Конечно, вы можете использовать сочетание реальных и сымитированных элементов. Например, в тестовом режиме я регулярно использую базу данных, размещенную в памяти. Это особенно легко сделать, если вы используете PDO. Вот пример упрощенного класса, в котором используется PDO для коммуникации с базой данных.

```
class DBFace {
   private $pdo;

   function __construct( $dsn, $user=null, $pass=null ) {
      $this->pdo = new PDO( $dsn, $user, $pass );
      $this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
   }

   function query( $query ) {
      $stmt = $this->pdo->query( $query );
```

```
        return $stmt;
    }
}
```

Если `DBFace` передается по системе и используется в объектах-преобразователях, то очень легко наполнить его данными, используя SQLite, таблицы которого размещены в оперативной памяти компьютера.

```
public function setUp() {
    $face = new DBFace("sqlite::memory:");
    $face->query("create table user ( id INTEGER PRIMARY KEY, name TEXT )");
    $face->query("insert into user (name) values('bob')");
    $face->query("insert into user (name) values('harry')");
    $this->mapper = new ToolMapper( $face );
}
```

Как вы, наверное, уже поняли, я не догматик в том, что касается тестирования. Я регулярно "жульничаю", сочетая реальные и сымитированные компоненты, и поскольку наполнение данными происходит неоднократно, я централизую средства тестирования (fixtures) в том, что Мартин Фаулер называет "материнскими объектами" (Object Mothers). Эти классы — просто фабрики, которые генерируют заполненные объекты для целей тестирования. Совместно используемые средства тестирования такого рода — проклятие для некоторых.

После рассмотрения ряда проблем, с которыми вы можете столкнуться при тестировании, я хочу еще раз повторить преимущества тестирования, превосходящие все его недостатки.

Итак, тестирование поможет в следующем:

- предотвратить ошибки (вы сможете их обнаружить во время разработки и рефакторинга);
- обнаружить ошибки (по мере того как вы будете увеличивать количество тестов);
- побудит вас сосредоточиться на проекте системы;
- позволит улучшить проект кода, в результате чего вы не будете опасаться, что изменения вызовут больше проблем, чем они решают;
- придаст вам уверенность при выпуске кода.

В каждом проекте, для которого я писал тесты, у меня были поводы порадоваться этому раньше или позже.

Резюме

В данной главе были рассмотрены виды тестов, которые все мы пишем как разработчики, но от которых так же часто легкомысленно отказываемся. Мы познакомились с пакетом PHPUnit, который позволяет писать такие же виды тестов во время разработки, но при этом сохранять их и ощущать преимущества от долговременного использования! Мы создали реализацию тестового примера, и я описал имеющиеся методы с утверждениями. Мы рассмотрели группировку тестов в наборы, изучили ограничения и исследовали фальшивый мир сымитированных объектов. Я продемонстрировал, как рефакторинг системы с целью выполнения тестирования может положительно сказаться на ее структуре в целом, и описал несколько методик тестирования веб-приложений, сначала с использованием только PHPUnit, а затем — и системы Selenium. И наконец я рискнул предупредить о затратах, которые влекут за собой тесты, и рассказал о возможных компромиссах.

Глава 19

Автоматическое построение с помощью Phing

Если контроль версий — это одна сторона монеты, то автоматическое построение — другая ее сторона. Контроль версий позволяет нескольким разработчикам совместно работать над одним проектом. И когда несколько разработчиков должны развернуть проект каждый на своем компьютере, автоматическое построение становится крайне необходимым. 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 использовать различные пароли для доступа к базе данных, библиотечные каталоги или почтовые программы. Гибкая кодовая база может легко приспосабливаться ко всем этим изменениям. Но усилия по изменению настроек и копированию каталогов вручную в файловой системе вскоре превратятся в утомительную работу — особенно если вам нужно инсталлировать код несколько раз в день или даже в час.

В одной из предыдущих глав мы рассмотрели механизмы инсталляции пакетов PEAR. Практически всегда вы должны выбирать именно этот механизм распространения своих пакетов до конечного пользователя. На то есть несколько причин. Во-первых, он не создает практически никаких проблем для конечного пользователя, поскольку PEAR наверняка уже есть на его компьютере и этот механизм поддерживает установку по сети. Во-вторых, PEAR великолепно справляется с заключительными этапами процесса инсталляции, однако перед созданием пакета нужно выполнить много рутинной работы, которую можно автоматизировать. Например, перед созданием дистрибутивного пакета вам придется как-то извлечь файлы из общего хранилища системы контроля версий. Вам также понадобится запустить тесты и собрать все файлы в отдельном каталоге. В конце концов вы наверняка захотите автоматизировать процесс создания самого пакета PEAR. В этой главе я познакомлю вас с Phing, средством, предназначенным для выполнения именно такой работы. В главе будут рассмотрены следующие темы.

- *Получение и инсталляция Phing:* кто построит построитель?
- *Свойства:* установка и получение данных.
- *Типы:* описание сложных частей проекта.
- *Задания:* разбиение построения на связанные функциональные блоки, которым присвоены имена.
- *Задачи:* нечто, выполняющее нужные действия.

Что такое Phing

Phing — это PHP-инструмент для построения проектов. Он очень похож на чрезвычайно популярный (и очень эффективный) Java-инструмент под названием Ant. Ant (в переводе "муравей") назван так потому, что он небольшой, но способен создавать очень крупные проекты. И Phing, и Ant используют XML-файл (обычно с именем `build.xml`) для определения того, что нужно сделать, чтобы инсталлировать или выполнить какие-либо другие операции над проектом.

PHP-сообщество *действительно* нуждается в хорошем средстве для установки приложений. В прошлом у серьезных разработчиков было несколько разнообразных инструментов. Можно было использовать утилиту `make`, это вездесущее средство построения для Unix, которое все еще используется в большинстве проектов на C, C++ и Perl. Но `make` очень требовательна к синтаксису и требует серьезных знаний командной оболочки, вплоть до включения сценариев, — а это может быть трудно для некоторых программистов на PHP, которые не умеют программировать для командной оболочки Unix или Linux. Более того, `make` предоставляет очень мало встроенных инструментов для общих операций построения, таких как преобразование имен файлов и содержимого. По сути, это обычное связующее звено для команд оболочки Unix. Поэтому с помощью `make` трудно писать программы, которые должны устанавливаться на разных платформах. Не во всех средах будет одна и та же версия `make`, если она будет вообще. Но даже если она будет, то может не быть всех команд, указанных в `makefile` (файл конфигурации, который управляет работой `make`).

Отношение Phing к `make` иллюстрируется его именем: Phing расшифровывается как "Phing Is Not Gnu make" (Phing — это не GNU `make`). Это распространенная шутка программистов (например, GNU расшифровывается как "Gnu is Not Unix" (GNU — это не Unix).

Phing — это "родное" PHP-приложение, которое интерпретирует созданный пользователем XML-файл, чтобы выполнить операции с проектом. Такие операции обычно включают копирование файлов из дистрибутивного каталога в различные каталоги назначения, но Phing — это нечто большее. Phing можно использовать для генерации документации, запуска тестов, вызова команд, запуска произвольного PHP-кода, создания пакетов PEAR, замены ключевых слов в файле, удаления комментариев и генерации заархивированных версий пакета в формате `tar/gzip`. Даже если Phing еще не делает то, что вам нужно, в нем предусмотрены широкие возможности для расширения.

Поскольку сам Phing — это PHP-приложение, 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.

Мы видели, что пакеты PEAR удивительно легко инсталлировать. PEAR поддерживает собственный автоматический механизм построения. Поскольку PEAR входит в поставку PHP, почему бы не воспользоваться механизмом PEAR для инсталляции собственных проектов? В конечном итоге ответ на этот вопрос всегда положительный. PEAR облегчает процесс инсталляции и поддерживает зависимости. В итоге вы всегда можете гарантировать, что ваши пакеты будут совместимы друг с другом. Во время разработки ПО вся кропотливая и нудная работа должна быть автоматизирована, включая процесс создания пакетов. Методика использования Phing в процессе разработки проекта и генерации пакета PEAR при выпуске продукта использована непосредственно при производстве самого Phing.

Получение и инсталляция Phing

Если трудно установить инструмент инсталляции, значит, здесь что-то не так! Но, исходя из того, что у вас установлен PHP 5 или более поздняя версия (если нет, то эта книга не для вас!), нет ничего проще инсталляции Phing.

Получить и инсталлировать Phing можно с помощью двух простых команд.

```
$ pear channel-discover pear.phing.info
$ pear install phing/phing
```

С помощью этих команд Phing устанавливается как пакет PEAR. Для работы с ним вы должны иметь права на запись в каталоги PEAR. В большинстве систем Unix или Linux это означает, что нужно запускать команды, обладая правами пользователя `root`.

Если у вас возникли какие-либо проблемы с инсталляцией, то вы должны загрузить и распаковать дистрибутив Phing по адресу `http://phing.info/trac/wiki/Users/Download`. Здесь вы найдете подробные инструкции по инсталляции.

Создание документа построения

Теперь вы готовы работать с Phing! Давайте это проверим.

```
$ phing -v

Phing version 2.10.1
```

Ключ `-v` в команде `phing` говорит сценарию вернуть информацию о номере версии. К тому времени, когда вы будете читать эту книгу, номер версии может измениться. Но, запустив эту команду на своем компьютере, вы должны увидеть аналогичное сообщение.

А теперь давайте запустим `phing` без аргументов.

```
$ phing
Buildfile: build.xml does not exist!
```

Как видите, Phing не может работать без инструкций. По умолчанию он ищет в текущем каталоге файл `build.xml`. Давайте создадим минимальный документ, чтобы по крайней мере избавиться от этого сообщения об ошибке.

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz" default="main">
   <target name="main"/>
</project>
```

Это минимум, которым можно обойтись в файле построения. Если мы сохраним предыдущий пример в файле `build.xml` и снова дадим команду `phing`, то получим интересный вывод.

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml
Warning: target 'main' has no tasks or dependencies

megaquiz > main:

BUILD FINISHED
Total time: 0.1023 seconds
```

Вы можете подумать: так много усилий, чтобы не получить совсем ничего! Как видите, Phing снова указал нам, что ничего особо полезного в нашем файле построения нет. Но мы же должны с чего-то начать! Давайте снова посмотрим на файл построения. Поскольку мы имеем дело с XML, то включаем объявление XML. Как вы, наверное, знаете, комментарии в XML выглядят так.

```
<!-- Здесь можно указать все, что угодно, поскольку это комментарий -->
```

Итак, вторая строка в файле построения игнорируется, поскольку это комментарий. Вы можете поместить в файл построения столько комментариев, сколько хотите, и когда он станет достаточно большим, сможете этим в полной мере воспользоваться. При сборке больших проектов в файле построения трудно разобраться без соответствующих комментариев.

Настоящее начало любого файла построения — это элемент `project`, который может включать до пяти атрибутов. Из них атрибуты `name` и `default` являются обязательными. Атрибут `name` определяет имя проекта, а атрибут `default` — задание, которое нужно запустить, если в командной строке ничего не указано. Необязательный атрибут `description` может предоставлять краткую информацию о проекте. С помощью атрибута `basedir` вы можете указать контекстный каталог для построения. Если этот атрибут опущен, то будет использоваться текущий рабочий каталог. И последний атрибут `phingVersion` позволяет указать минимальную версию программы Phing, которая требуется для построения ваших приложений. Данные атрибуты описаны в табл. 19.1.

Таблица 19.1. Атрибуты элемента `project`

Атрибут	Требуется ли	Описание	Стандартное значение
`name`	Да	Имя проекта	Нет
`description`	Нет	Краткое описание проекта	Нет
`default`	Да	Стандартное задание для запуска	Нет
`phingVersion`	Нет	Минимальная версия программы Phing, требуемая для построения приложения	Нет
`basedir`	Нет	Каталог файловой системы, в котором будет выполняться построение	Текущий каталог (`.`)

Определив элемент `project`, мы должны создать как минимум одно задание — то, на которое мы будем ссылаться в атрибуте `default`.

Задания

В некотором отношении задания аналогичны функциям. Задание — это набор действий, сгруппированных для достижения результата, например для копирования каталога из одного места в другое или генерирования документации.

В предыдущем примере мы включили минимальную реализацию для задания.

```
<target name="main"/>
```

Как видите, в задании должен быть определен как минимум атрибут `name`. Мы использовали это в элементе `project`. Поскольку стандартный элемент указывает на задание `main`, именно оно будет вызвано при каждом запуске утилиты `phing` без указания аргументов в командной строке. Это подтверждается следующим выводом.

```
megaquiz > main:
```

Задания можно организовать так, чтобы они зависели одно от другого. Установив зависимость между заданиями, вы сообщаете Phing, что первое задание не должно

быть запущено до того, как завершится задание, от которого оно зависит. Давайте добавим зависимость в файл построения.

```
<?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 ввели новый атрибут для элемента `target`. Атрибут `depends` сообщает Phing, что задание, на которое есть ссылка, должно выполняться до текущего задания. Поэтому можно сделать так, чтобы задание, которое копирует определенные файлы в каталог, вызывалось до задания, выполняющего преобразование всех файлов в этом каталоге. Мы добавили в этот пример два новых задания: `runsecond`, от которого зависит задание `main`, и `runfirst`, от которого зависит `runsecond`. Давайте посмотрим, что происходит, когда мы запускаем Phing с этим файлом построения.

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml
Warning: target 'runfirst' has no tasks or dependencies

megaquiz > runfirst:

megaquiz > runsecond:

megaquiz > main:

BUILD FINISHED
Total time: 0.1022 seconds
```

Как видите, зависимости соблюдаются. Phing встречает задание `main`, видит его зависимость и переходит к выполнению задания `runsecond`. У задания `runsecond` есть своя зависимость, и Phing вызывает задание `runfirst`. Удовлетворив его зависимость, Phing снова вызывает задание `runsecond`. И наконец вызывается задание `main`. В атрибуте `depends` можно указывать несколько заданий одновременно. Можно указать список зависимостей, разделенных запятыми, и каждая будет соблюдаться по очереди.

Теперь, когда у нас есть несколько заданий, давайте явно укажем задание из командной строки и тем самым изменим действие атрибута `default`.

```
$ phing runsecond

Buildfile: /home/bob/working/megaquiz/build.xml
Warning: target 'runfirst' has no tasks or dependencies

megaquiz > runfirst:

megaquiz > runsecond:

BUILD FINISHED
Total time: 0.2671 seconds
```

Поскольку передано имя задания, атрибут `default` игнорируется. Вместо этого вызывается задание, соответствующее указанному аргументу, а также задание, от которого оно зависит. Это полезно для вызова специализированных задач, таких как очистка каталога построения или запуск после инсталляционных сценариев.

В элементе `target` также можно указать необязательный атрибут `description`, в котором можно описать назначение задания.

```
<?xml version="1.0" encoding="windows-1251" ?>
<!-- build xml -->

<project name="megaquiz"
         default="main"
         description="Движок проекта">

   <target name="runfirst"
         description="Первое задание" />

   <target name="runsecond"
         depends="runfirst"
         description="Второе задание" />

   <target name="main"
         depends="runsecond"
         description="Основное задание" />
</project>
```

От добавления описания к заданиям в нормальном процессе построения ничего не меняется. Но если пользователь запускает Phing с ключом `-projecthelp`, то описания будут использоваться для получения краткой информации о проекте.

```
$ phing -projecthelp
Buildfile: /home/bob/working/megaquiz/build.xml
Warning: target 'runfirst' has no tasks or dependencies
Движок проекта
Default target:
-------------------------------------------------------------------------------
main      Основное задание
Main targets:
-------------------------------------------------------------------------------
main      Основное задание
runfirst  Первое задание
runsecond Первое задание
```

Обратите внимание на то, что я добавил атрибут `description` и к элементу `project`. Если вам нужно убрать определенное задание из листинга, приведенного выше, добавьте к элементу атрибут `hidden`. Такая возможность может понадобиться для тех заданий, которые выполняют вспомогательные действия и не должны вызываться напрямую из командной строки.

```
<target name="housekeeping" hidden="true" />
   <!-- Вспомогательные действия, не вызываются напрямую -->
</target>
```

Свойства

Phing позволяет устанавливать значения свойств с помощью элемента `property`.

Свойства аналогичны глобальным переменным в сценарии. Как правило, они объявляются в начале файла, чтобы разработчикам легче было определить, "что

есть что" в файле построения. Мы создали файл построения, в котором указана информация о базе данных.

```
<?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>
```

Мы ввели новый элемент, `property`, для которого нужно указать атрибуты `name` (имя) и `value` (значение). Также обратите внимание на то, что мы добавили к заданию `main` нечто новое. Элемент `echo` — это пример задачи. Задачи подробнее рассмотрим в следующем разделе. А пока достаточно знать, что `echo` делает в точности то, чего можно было ожидать, — выводит свое содержимое. Обратите внимание на синтаксис, который используется для ссылки на значение свойства, — используя знак доллара и заключая имя свойства в фигурные скобки, вы говорите Phing, что нужно заменить строку значением свойства.

```
${propertyname}
```

Все, что делает данный файл построения, — это объявляет три свойства и выводит их в стандартное устройство вывода. Давайте рассмотрим это в действии.

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

   [echo] database: megaquiz
   [echo] pass:     default
   [echo] host:     localhost

BUILD FINISHED

Total time: 0.1065 seconds
```

Теперь, когда мы познакомились со свойствами, можем завершить изучение заданий. У элемента `target` может быть еще два дополнительных атрибута: `if` и `unless`. Для каждого из них нужно указать имя свойства. Если вы используете `if` с именем свойства, задание будет выполнено, если только определено заданное свойство. Если свойство не определено, то выполнение задания завершится без всякого вывода. В данном примере мы закомментировали свойство `dbpass` и сделали так, чтобы его потребовало задание `main` с помощью атрибута `if`.

```
<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>
```

Давайте снова запустим `phing`.

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

BUILD FINISHED

Total time: 0.1064 seconds
```

Как видите, это не вызвало ошибки, но задание `main` не запустилось. Зачем нам это нужно? Существует еще один способ определения свойств в проекте. Их можно определить в командной строке. О передаче свойства Phing сообщает ключ `-D`, за которым следуют имя и значение. Поэтому аргумент будет выглядеть так.

```
-Dname=value
```

В данном примере нам нужно, чтобы значение свойства `dbpass` можно было определить из командной строки. Попробуем это сделать.

```
$ phing -Ddbpass=userset
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

   [echo] database: megaquiz
   [echo] pass:     userset
   [echo] host:     localhost

BUILD FINISHED

Total time: 0.1047 seconds
```

Атрибут `if` задания `main` удовлетворен тем, что свойство `dbpass` присутствует и выполнение задания разрешено.

Как можно было ожидать, атрибут `unless` — это противоположность `if`. Если свойство определено и на него есть ссылка в атрибуте `unless` задания, то задание не запустится. Это полезно в случае, если вы хотите иметь возможность подавить определенное задание из командной строки. Поэтому мы можем добавить к заданию `main` следующее.

```
<target name="main" unless="suppressmain">
```

Задание `main` будет выполнено только в том случае, если не присутствует свойство `suppressmain`.

```
$ phing -Dsuppressmain=yes
```

Завершив рассмотрение элемента `target`, давайте подытожим информацию о его атрибутах в табл. 19.2.

Таблица 19.2. Атрибуты элемента `target`

Атрибут	Требуется ли	Описание
`name`	Да	Имя задания
`depends`	Нет	Задания, от которых зависит текущее задание
`if`	Нет	Выполнить задание, только если присутствует заданное свойство
`unless`	Нет	Выполнить задание, только если не присутствует заданное свойство
`hidden`	Нет	Убрать задание из выводимых листингов и отчетов
`description`	Нет	Краткое описание назначения задания

Если свойство определено в командной строке, оно замещает все объявления свойств в файле построения. Существует еще одно условие, при котором значение свойства может быть заменено. По умолчанию, если свойство объявляется дважды, то первоначальное значение будет иметь приоритет. Но это положение вещей можно изменить, определив атрибут `override` во втором элементе `property`. Приведем пример.

```
<?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>
```

Мы определяем свойство `dbpass`, присваивая ему начальное значение `"default"`. В задании `main` мы снова определяем свойство, добавляя атрибут `override`, для которого установлено значение `"yes"`, и предоставляя новое значение. Это новое значение отражено в выводе.

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

   [echo] pass: specific

BUILD FINISHED

Total time: 0.3802 seconds
```

Если бы мы не определили элемент `override` во втором элементе `property`, то первоначальное значение `"default"` осталось бы прежним. Важно отметить, что задания — это не функции: для них нет понятия локального контекста. Если заменить свойство задачей, то оно останется замещенным для всех других задач в файле построения. Но, конечно, это можно обойти, сохранив значение свойства во временном свойстве до замещения, а затем восстановив его после окончания работы локально.

До сих пор мы имели дело со свойствами, которые вы определили сами. В Phing также предусмотрены встроенные свойства. На них ссылаются точно так же, как на свойства, которые вы объявили сами. Приведем пример.

```
<?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>
```

Здесь мы используем всего несколько встроенных свойств Phing. Свойство `phing.project.name` заменяется именем проекта, как определено в атрибуте `name` элемента `project`; свойство `project.basedir` сообщает базовый каталог; а `user.home` — домашний каталог пользователя (эти свойства используются для определения стандартных мест инсталляции проекта).

И наконец префикс `env` в ссылке на свойство указывает на переменную окружения операционной системы. Так, указывая `${env.DBPASS}`, мы ищем переменную окружения `DBPASS`. Давайте запустим Phing для этого файла.

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

   [echo] name:    megaquiz
   [echo] base:    /home/bob/working/megaquiz
   [echo] home:    /home/bob
   [echo] pass:    ${env.DBPASS}

BUILD FINISHED

Total time: 0.1120 seconds
```

Обратите внимание на то, что последнее свойство не было заменено значением. Это стандартное поведение в случае, если свойство не найдено, — строка, ссылающаяся на свойство, остается без изменений. Если мы определим переменную окружения `DBPASS` и запустим `phing` снова, то увидим эту переменную в выводе.

```
$ export DBPASS=wooshpoppow
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:
...
   [echo] pass:     whooshpoppow

BUILD FINISHED

Total time: 0.2852 seconds
```

Итак, мы увидели три способа определения свойства: с помощью элемента `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 именно этим способом. Давайте еще раз посмотрим на содержимое основного файла построения.

```
<?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, причем они нигде заранее не определены и не выполняется проверка, присвоено ли им что-то. Ниже показано, что произойдет, если запустить программу `phing` без параметров.

```
$ phing
...
   [echo] database: ${dbname}
   [echo] pass:     ${dbpass}
   [echo] host:     ${dbhost}
...
```

А теперь давайте определим значения свойств в отдельном файле. Я назвал его `megaquiz.properties`.

```
dbname=filedb
dbpass=filepass
dbhost=filehost
```

После этого можно подключить этот файл к процессу построения с помощью ключа `-propertyfile` командной строки утилиты `phing`.

```
$ phing -propertyfile megaquiz.properties
```

```
...
   [echo] database: filedb
   [echo] pass:     filepass
   [echo] host:     filehost
...
```

Я полагаю, что такой способ намного удобнее указания длинного списка значений свойств в командной строке. Однако при этом нужно быть осмотрительным и не помещать файл свойств в систему контроля версий!

Задания можно использовать для того, чтобы гарантировать, что нужные свойства определены. Предположим, в проекте требуется свойство `dbpass`. Мы хотим, чтобы пользователь определял `dbpass` в командной строке (у этого способа всегда есть приоритет над методами присвоения значений свойств). Но если это не сдела-

но, то мы должны искать переменную окружения. Если и она не найдена, то нужно использовать стандартное значение.

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
        default="main"
>
   <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>
```

Итак, как обычно, стандартное задание `main` вызывается первым. У него есть набор зависимостей, поэтому Phing возвращается к заданию `setpass`. Но `setpass` зависит от задания `setenvpass`, поэтому мы начинаем с него. Задание `setenvpass` сконфигурировано так, чтобы запускаться, только если свойство `dbpass` не было определено и если присутствует `env.DBPASS`. Если эти условия удовлетворяются, то мы определяем свойство `dbpass` с помощью элемента `property`. Тогда на этом этапе свойству `dbpass` присваивается значение либо аргумента командной строки, либо переменной окружения. Если ни одного из этих элементов нет, то на данном этапе свойство остается неопределенным. Теперь выполняется задание `setpass`, но только если `dbpass` еще не определено. В данном случае свойству присваивается стандартная строка: `"default"`.

Условное присвоение значений свойств с помощью задачи `condition`

В предыдущем примере была показана довольно сложная логика присвоения значений свойств. Однако чаще всего нам требуется просто определить значение, принятое по умолчанию. Для этой цели предусмотрена задача `condition`, которая позволяет задать значение свойства на основе устанавливаемых условий. Пример приведен ниже.

```
<?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>
```

Для задачи `condition` требуется задать атрибут `property`. Кроме того, можно указать необязательный атрибут `value`, значение которого определяет значение свойства в случае, если проверка вложенного условия окажется истинной. Если атрибут `value` не задан, то в этом случае свойству будет присвоено значение `true`.

Проверяемое условие задается с помощью одного из дескрипторов, часть из которых, как, например, используемый в примере `not`, допускает свои вложенные дескрипторы. В примере используется элемент `isset`, который возвращает значение `true`, если задано значение указанного в нем свойства. Поскольку мы хотим присвоить значение свойству `dbpass` только в случае, если оно еще *не определено*, значение элемента `isset` нужно инвертировать, поместив его в дескриптор `not`. Этот дескриптор инвертирует значение содержащегося в нем дескриптора. В результате, используя синтаксис PHP, задачу `condition` для нашего примера можно записать так, как показано ниже.

```
if ( ! isset( $dbname ) ) {
   $dbname = "default";
}
```

На заметку. Список всех условных элементов файла построения приведен в документации Phing по адресу `https://www.phing.info/docs/guide/stable/ch05s08.html`

Типы

Наверное, вы думаете, что, рассмотрев свойства, мы закончили с данными. На самом деле Phing поддерживает ряд специальных элементов — типов, которые инкапсулируют различные виды информации, полезной для процесса построения.

`fileset`

Приведем пример обычной ситуации: нам нужно определить каталог в файле построения. Конечно, для этой цели можно использовать свойство, но если разработчики используют различные платформы, в которых различаются символы — разделители каталогов, то вы сразу же столкнетесь с проблемами. Поэтому в данной ситуации нужно использовать тип данных `fileset`, который не зависит от платформы. Если вы определите каталог с прямыми косыми чертами в пути, то они автоматически будут преобразованы в обратные косые черты, когда построение будет запущено в системе Windows. Минимальный элемент `fileset` можно определить следующим образом.

```
<fileset dir="src/lib" />
```

Как видите, мы используем атрибут `dir`, чтобы определить каталог, который хотим представить. Можно также добавить необязательный атрибут `id`, чтобы впоследствии можно было сослаться на `fileset`.

```
<fileset dir="src/lib" id="srclib">
```

Тип данных `fileset` особенно полезен при определении типов документов, которые нужно включать или исключать. Например, при инсталляции набора файлов вы не хотите включать те, которые соответствуют определенному шаблону. Такие условия можно определять в атрибуте `excludes`.

```
<fileset dir="src/lib" id="srclib"
   excludes="**/*_test.php **/*Test.php" />
```

Обратите внимание на синтаксис, который мы использовали в атрибуте `excludes`. Две звездочки представляют любой каталог или подкаталог внутри `src/lib`. Одна звездочка представляет нуль или несколько символов. Итак, мы указываем, что хотим исключить файлы, которые заканчиваются на `_test.php` или `Test.php`, во всех подкаталогах, определенных в атрибуте `dir`. В атрибуте `excludes` можно указать несколько шаблонов, разделенных пробелом.

Такой же синтаксис можно применить к атрибуту `includes`. Возможно, каталоги `src/lib` содержат много не PHP-файлов, которые полезны для разработчиков, но не должны инсталлироваться. Конечно, мы можем исключить эти файлы, но, возможно, проще определить виды файлов, которые нужно включить. В данном случае, если файл не заканчивается на `.php`, то он не должен инсталлироваться.

```
<fileset dir="src/lib" id="srclib"
   excludes="**/*_test.php **/*Test.php"
   includes="**/*.php" />
```

По мере создания правил `include` и `exclude` элемент `fileset`, вероятно, станет слишком длинным. К счастью, можно извлечь отдельные правила `exclude` и поместить каждое из них в собственный элемент `exclude`. То же самое можно сделать для правил `include`. Теперь можно переписать элемент `fileset` следующим образом.

```
<fileset dir="src/lib" id="srclib">
   <exclude name="**/*_test.php" />
   <exclude name="**/*Test.php" />
   <include name="**/*.php" />
</fileset>
```

Некоторые из атрибутов элемента `fileset` описаны в табл. 19.3.

Таблица 19.3. Некоторые атрибуты элемента `fileset`

Атрибут	Требуется ли	Описание
`id`	Нет	Уникальный идентификатор для ссылки на элемент
`dir`	Нет	Каталог `fileset`
`excludes`	Нет	Список шаблонов для исключения
`includes`	Нет	Список шаблонов для включения
`refid`	Нет	Текущий `fileset` является ссылкой на `fileset` с заданным ID

patternset

При создании шаблонов в элементе `fileset` (и в других) существует вероятность, что мы начнем повторять группы элементов `exclude` и `include`. В предыдущем примере мы определили шаблоны для тестовых файлов и обычных файлов кодов. Со временем мы можем что-то добавить к ним (например, мы захотим включить расширения `.conf` и `.inc` к нашему определению файлов кодов). Если мы определили другие элементы `fileset`, в которых тоже используются эти шаблоны, то будем вынуждены вносить все изменения во всех соответствующих элементах `fileset`.

Чтобы решить эту проблему, сгруппируем шаблоны в элементы `patternset`. Элемент `patternset` группирует элементы `exclude` и `include` так, что на них можно сослаться впоследствии из других типов. Давайте выделим элементы `exclude` и `include` из примера `fileset` и добавим их к элементам `patternset`.

```
<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>
```

Мы создаем два элемента `patternset`, определяя для их атрибутов `id` значения `inc_code` и `exc_test` соответственно. Элемент `inc_code` содержит элементы `include` для включения файлов кодов, а `exc_test` — элементы `exclude` для исключения тестовых файлов. Теперь мы можем сослаться на эти элементы `patternset` внутри `fileset`.

```
<fileset dir="src/lib" id="srclib">
   <patternset refid="inc_code" />
   <patternset refid="exc_test" />
</fileset>
```

Чтобы сослаться на существующий элемент `patternset`, вы должны использовать другой элемент `patternset`. В этом втором элементе должен быть определен единственный атрибут — `refid`. Этот атрибут должен ссылаться на `id` элемента `patternset`, который вы хотите использовать в текущем контексте. Таким образом, можно повторно использовать элементы `patternset`.

```
<fileset dir="src/views" id="srcviews">
   <patternset refid="inc_code" />
</fileset>
```

Любые изменения, вносимые в элементы `inc_code patternset`, будут автоматически отражаться на всех типах, которые их используют. Как и с `fileset`, вы можете поместить правила `exclude` либо в атрибут `excludes`, либо в набор подэлементов `exclude`. То же самое верно для правил `include`.

Некоторые атрибуты элемента `patternset` описаны в табл. 19.4.

Таблица 19.4. Некоторые атрибуты элемента `patternset`

Атрибут	Требуется ли	Описание
`id`	Нет	Уникальный идентификатор для ссылки на элемент
`excludes`	Нет	Список шаблонов для исключения
`includes`	Нет	Список шаблонов для включения
`refid`	Нет	Текущий `patternset` является ссылкой на `patternset` с заданным ID

`filterchain`

Типы, с которыми мы имели дело до сих пор, обеспечивали механизмы выбора наборов файлов. А `filterchain` обеспечивает гибкий механизм преобразования содержимого текстовых файлов.

Как и в случае всех типов, определение элемента `filterchain` само по себе не вызывает никаких изменений. Этот элемент и его дочерние элементы должны сначала быть связаны с задачей, т.е. элемент сообщает Phing, какие действия нужно выполнить. К задачам мы вернемся несколько позже.

Элемент `filterchain` группирует любое количество фильтров. Фильтры работают с файлами по принципу конвейера — первый изменяет его файл и передает результат второму, который выполняет собственные изменения, и т.д. Объединяя несколько фильтров в элемент `filterchain`, можно гибко осуществлять преобразования.

Давайте создадим фильтр, который удаляет PHP-комментарии из любого переданного ему текста.

```
<filterchain>
   <stripphpcomments />
</filterchain>
```

Задача `stripphpcomments` делает то, что соответствует ее имени, т.е. удаляет комментарии. Если вы включили подробную документацию в исходный код, то облегчили жизнь разработчикам, но добавили много лишнего в проект. Поскольку все значительные операции выполняются внутри исходных каталогов, нет никакой причины, мешающей удалить комментарии при инсталляции.

На заметку. Если для развертывания проектов вы используете инструмент построения, то необходимо сделать так, чтобы никто не вносил изменений в инсталлированный код. Инсталлятор скопирует файлы поверх измененных, и все изменения будут потеряны. Я сталкивался с такими случаями.

Давайте немного забежим наперед и добавим элемент `filterchain` в задачу.

```
<target name="main">
   <copy todir="build/lib">
      <fileset refid="srclib"/>
      <filterchain>
         <stripphpcomments />
      </filterchain>
   </copy>
</target>
```

Задачей `copy` вы, наверное, будете пользоваться чаще всего. Она копирует файлы из одного места в другое. Как видите, мы определяем каталог назначения в атрибуте `todir`. Источник файлов определяется элементом `fileset`, который мы создали в предыдущем разделе. Затем следует элемент `filterchain`. К любому файлу, скопированному задачей `copy`, будет применено это преобразование.

Phing поддерживает фильтры для многих операций, включая удаление разрывов строки (`striplinebreaks`) и замену символов табуляции пробелами (`tabtospaces`). Существует даже фильтр `xsltfilter` для применения XSLT-преобразований к исходным файлам! Но, наверное, самый широко используемый фильтр — это `replacetokens`. Он позволяет заменить токены в исходном коде значениями свойств, определенных в файле построения, извлеченных из переменных окружения или переданных в командной строке. Это очень полезно для настройки процесса инсталляции. Имеет смысл поместить все токены в центральном файле конфигурации, чтобы легче было следить за меняющимися аспектами проекта.

Фильтр `replacetokens` имеет два необязательных атрибута, `begintoken` и `endtoken`. Их можно использовать, чтобы задать символы, определяющие границы токенов. Если опустить эти атрибуты, то Phing будет использовать стандартный символ `@`. Чтобы распознавать и заменять токены, вы должны добавить элементы `token` к элементу `replacetokens`. Давайте добавим элемент `replacetokens` в нашем примере.

```
<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 `token` нужно указать атрибуты `key` и `value`. Давайте посмотрим на результат выполнения этой задачи по преобразованию файлов в нашем проекте. Первоначальный файл находится в исходном каталоге `src/lib/Config.php`.

```
/**
 * Quick and dirty Conf class
 **/
class Config {
   public $dbname ="@dbname@";
   public $dbpass ="@dbpass@";
   public $dbhost ="@dbhost@";
}
```

Выполнение нашего основного задания, содержащего задачу `copy`, которая была определена ранее, дает следующий вывод.

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

   [copy] Copying 8 files to /home/bob/working/megaquiz/build/lib
   [filter:ReplaceTokens] Replaced "@dbname@" with "megaquiz"
   [filter:ReplaceTokens] Replaced "@dbpass@" with "default"
   [filter:ReplaceTokens] Replaced "@dbhost@" with "localhost"

BUILD FINISHED

Total time: 0.1413 seconds
```

Конечно, первоначальный файл остался нетронутым, но благодаря задаче `copy` он был скопирован в каталог `build/lib/Config.php`.

```
class Config {
   public $dbname ="megaquiz";
   public $dbpass ="default";
   public $dbhost ="localhost";
}
```

Теперь не только удалены комментарии, но и токены заменены эквивалентными им значениями свойств.

Задачи

Задачи — это элементы в файле построения, которые выполняют нужные нам действия. Без использования задач много не сделаешь, поэтому, опережая ход событий, мы уже использовали некоторые из них. А теперь давайте рассмотрим их подробнее.

echo

Задача `echo` идеальна для традиционного примера `"Hello World"`. А на практике вы можете использовать ее, чтобы сообщить пользователю, что вы собираетесь делать или что вы сделали. Можно также сделать "санитарную" проверку процесса построения, отобразив значения свойств. Как мы видели, любой текст, помещенный внутри открывающих и закрывающих тегов элемента `echo`, будет выведен в окне браузера.

```
<echo>The pass is '${dbpass}', shhh!</echo>
```

Есть и альтернативный вариант — добавить выходное сообщение к атрибуту `msg`.

```
<echo msg="The pass is '${dbpass}', shhh!" />
```

Результат будет таким же.

```
[echo] The pass is 'default', shhh!
```

copy

Копирование — это на самом деле то, что представляет собой процесс инсталляции. Обычно вы создаете одно задание, которое копирует файлы из исходных каталогов и объединяет их во временном каталоге построения. Затем у вас будет другое задание, которое копирует объединенные (и преобразованные) файлы по месту назначения. Разбиение процесса инсталляции на отдельные этапы построения и инсталляции не является абсолютно необходимым, но означает, что вы можете проверить результаты первоначального построения перед тем, как перезаписать рабочий код. Вы также можете изменить значение свойства и инсталлировать файлы в другом месте, не выполняя снова трудоемкий этап копирования/замены.

Самое простое, что позволяет сделать задача `copy`, — это указать исходный файл и каталог либо файл назначения.

```
<copy file="src/lib/Config.php" todir="build/conf" />
```

Как видите, мы определили исходный файл с помощью атрибута `file`. Наверное, вы уже знакомы с атрибутом `todir`, который используется для указания целевого каталога. Если целевой каталог не существует, то Phing автоматически создаст его.

Если вам нужно указать файл назначения, а не содержащий его каталог, то вместо атрибута `todir` используйте атрибут `tofile`.

```
<copy file="src/lib/Config.php" tofile="build/conf/myConfig.php" />
```

И снова в случае необходимости автоматически создается каталог `build/conf`, но на этот раз файл `Config.php` переименовывается в `myConfig.php`.

Как мы видели, чтобы скопировать несколько файлов одновременно, нужно добавить элемент `fileset` к `copy`.

```
<copy todir="build/lib">
   <fileset refid="srclib"/>
</copy>
```

Исходные файлы определяются элементом `srclib fileset`, поэтому все, что нам нужно определить в `copy`, — это атрибут `todir`.

Phing достаточно умен, чтобы протестировать, изменился ли исходный файл со времени создания файла назначения. Если изменений не было, то Phing копировать его не будет. Это означает, что вы можете строить проект много раз, но при этом будут инсталлированы только измененные файлы.

Это хорошо до тех пор, пока других изменений не предвидится. Если файл преобразуется согласно заданным параметрам конфигурации, например элемента `replacetokens`, то нужно сделать так, чтобы файл преобразовывался при каждом вызове задачи `copy`. Это можно сделать, определив атрибут `overwrite`.

```
<copy todir="build/lib" overwrite="yes">
   <fileset refid="srclib"/>
   <filterchain>
      <stripphpcomments />
      <replacetokens>
         <token key="dbpass" value="${dbpass}" />
      </replacetokens>
   </filterchain>
</copy>
```

Теперь, при каждом запуске задачи `copy`, файлы, соответствующие элементу `fileset`, заменяются независимо от того, обновлялся исходный файл или нет.

Атрибуты элемента `copy` описаны в табл. 19.5.

Таблица 19.5. Атрибуты элемента `copy`

Атрибут	Требуется ли	Описание	Стандартное значение
`todir`	Да (если нет `tofile`)	Каталог, в который выполняется копирование	Нет
`tofile`	Да (если нет `todir`)	Файл, в который выполняется копирование	Нет
`file`	Нет	Исходный файл	Нет
`tstamp`	Нет	Сравнивает время создания/модификации перезаписываемого файла (оно остается неизменным)	`false` (ложь)
`preservemode`	Нет	Копирует вместе с файлом и его атрибуты прав доступа	`false` (ложь)
`includeemptydirs`	Нет	Копирует пустые каталоги	`false` (ложь)
`mode`	Нет	Задает права доступа в виде восьмеричного числа	`755`
`haltonerror`	Нет	Останавливает процесс построения в случае возникновения ошибки	`true` (истина)
`overwrite`	Нет	Перезаписывает целевые файлы, если они существуют	`no` (нет)

input

Мы видели, что элемент `echo` используется для вывода сообщений пользователю. Чтобы получить входные данные *от* пользователя, мы применяли ряд методик, включающих использование командной строки и переменной окружения. Но эти механизмы не являются ни структурированными, ни интерактивными.

На заметку. Причина, по которой пользователям разрешают определять значения во время построения, заключается в том, чтобы обеспечить гибкость в разных средах построения. В случае применения паролей к базе данных эти конфиденциальные данные не должны указываться в самом файле построения. Но, конечно, после запуска построения пароль будет сохранен в исходном файле, поэтому уже от разработчика зависит, как он обеспечит безопасность системы!

Элемент `input` позволяет выводить приглашение на ввод данных от пользователя. Затем Phing ждет входных данных от пользователя и присваивает их указанному свойству. Вот как это выглядит в действии.

```
<target name="setpass" unless="dbpass">
   <input message="You don't seem to have set a db password"
          propertyName="dbpass"
         defaultValue="default"
         promptChar=" >" />
</target>

<target name="main" depends="setpass">
   <echo>pass: ${dbpass}</echo>
</target>
```

И снова у нас есть стандартное задание: `main`. Оно зависит от другого задания, `setpass`, которое ответственно за то, чтобы свойству `dbpass` было присвоено значение. Для этого мы используем атрибут `unless` элемента `target`, который гарантирует, что задание не будет запущено, если `dbpass` уже определен.

Задание `setpass` состоит из единственного элемента задачи `input`. У элемента `input` может быть атрибут `message`, который должен содержать приглашение для пользователя. Атрибут `propertyName` является обязательным и определяет свойство, в которое помещаются введенные пользователем данные. Если в ответ на приглашение пользователь нажимает клавишу <Enter>, не определяя значение, то свойству назначается стандартное значение, определенное в атрибуте `defaultValue`. И наконец можно изменить сам символ приглашения с помощью атрибута `promptChar` — он дает пользователю визуальный сигнал к тому, чтобы вводить данные. Давайте запустим Phing, используя предыдущие задания.

```
$ phing
```

```
Buildfile: /home/bob/working/megaquiz/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: 6.0322 seconds
```

Атрибуты элемента `input` описаны в табл. 19.6.

Таблица 19.6. Атрибуты элемента `input`

Атрибут	Требуется ли	Описание
`propertyName`	Да	Свойство, которому присваиваются введенные пользователем данные
`message`	Нет	Приглашение
`defaultValue`	Нет	Значение, которое нужно назначить свойству, если пользователь не ввел данные

Окончание табл. 19.6

Атрибут	Требуется ли	Описание
validArgs	Нет	Список приемлемых входных значений, разделенных запятыми. Если пользователь ввел значение, которого нет в этом списке, то Phing снова выдаст приглашение
promptChar	Нет	Символ приглашения

delete

Процесс инсталляции обычно предусматривает создание, копирование и преобразование файлов. Но удаление тоже иногда необходимо, особенно если вы хотите осуществить инсталляцию "с нуля". Как мы уже говорили, из исходного места в место назначения обычно копируются только исходные файлы, которые изменились со времени последнего построения. Удаление каталога построения приведет к выполнению полного процесса сборки проекта.

Давайте удалим каталог.

```
<target name="clean">
   <delete dir="build" />
</target>
```

Когда мы запускаем Phing с аргументом `clean` (имя задания), то вызывается задача `delete`. Вот вывод Phing.

```
$ phing clean
```

```
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > clean:

[delete] Deleting directory /home/bob/working/megaquiz/build

BUILD FINISHED
```

Элемент `delete` также имеет атрибут `file`, который можно использовать для указания конкретного файла. Есть альтернативный вариант: настроить операции удаления, добавив вложенные элементы `fileset` к элементу `delete`.

Резюме

Серьезный процесс разработки редко происходит в каком-то одном месте. Кодовая база должна быть отделена от своей инсталляции, 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 для своей среды разработки. И наконец, наверное, самое важное: клиент (даже если клиент — это вы сами через год, когда забудете все детали собственного кода) должен суметь инсталлировать проект, просмотрев файл `README`.

В данной главе я описал основы Phing — замечательного инструмента, который привнес значительную часть функциональности Ant в мир PHP. Мы только поверхностно рассмотрели возможности Phing. Тем не менее, когда вы разберетесь в зада-

ниях, задачах, типах и свойствах, которые мы здесь осветили, вам уже будет легко использовать новые элементы для более сложных возможностей, таких как создание заархивированных дистрибутивов в формате `tar/gzip`, автоматическое генерирование инсталляций пакета PEAR и запуск PHP-кода непосредственно из файла построения.

Если по какой-то причине Phing не удовлетворяет все ваши потребности в построении проекта, то у него, также как и в Ant, предусмотрены возможности расширения. В результате вы можете пойти дальше и построить собственные задачи! Даже если вы не собираетесь добавлять новые возможности к Phing, не пожалейте времени на то, чтобы изучить исходный код. Phing полностью написан на объектно-ориентированном PHP, и его код изобилует примерами использования стандартных проектных шаблонов.

Глава 20

Непрерывная интеграция

В предыдущих главах вы познакомились с набором великолепных инструментов, предназначенных для поддержки хорошо управляемых программных проектов. Модульное тестирование, автоматическое создание документации, средства построения и контроля версий — все это просто чрезвычайно полезно! Правда, некоторые из них, особенно средства тестирования, очень нудные и могут быстро надоесть.

Даже если на прогонку тестов требуется всего пару минут, часто бывает так, что вы всецело поглощены кодированием и не уделяете тестам должного внимания. И причина не только в этом! Ваши коллеги и клиенты часто ждут от вас новых реализаций кода и новых возможностей. Поэтому искушение полностью погрузиться в процесс кодирования есть всегда. Однако ошибки намного легче выявить и локализовать, если после написания фрагмента кода сразу же его протестировать, не откладывая все в долгий ящик. Так происходит потому, что вы сможете быстро сообразить, в какой фрагмент кода внесли изменения и какие из них стали источником проблем, и, соответственно, быстро их устранить.

В этой главе я познакомлю вас с *непрерывной интеграцией* (Continuous Integration) — методикой автоматизации процесса тестирования и построения проекта, а также попытаюсь систематизировать все те сведения, о которых шла речь в предыдущих главах.

В этой главе будут рассмотрены следующие темы.

- Понятие о непрерывной интеграции (НИ).
- Подготовка проекта к НИ.
- Знакомство с Jenkins — сервером НИ.
- Настройка сервера Jenkins для PHP-проектов с помощью специализированного модуля дополнения.

Что же такое непрерывная интеграция

В старые добрые времена интеграция была чем-то таким, что вам приходилось выполнять после завершения приятной части работы. Она также была неким этапом, на котором вы начинали отчетливо понимать, сколько еще работы вам предстоит выполнить. *Интеграция* — это процесс, в котором собирались все части проекта в единый пакет, который затем можно было отправить конечному потребителю

и развернуть на его компьютерах. В нем не было ничего захватывающего, этот процесс на самом деле был очень трудным.

Интеграция тесно связана с процессом контроля качества. В самом деле, нельзя же отправить продукт конечному потребителю, если он не соответствует поставленным им целям. А это означает тестирование, много тестирования. И если до этого не было проведено никаких тестов, то на этапе интеграции возникало много неприятных сюрпризов. И даже очень много!

В главе 18 мы говорили о том, что настоятельно рекомендуется выполнять раннее и частое тестирование. В главах 15 и 19 я уже говорил о том, что при разработке проекта вы должны сразу же продумать процесс его развертывания. Многие из нас соглашаются, что так должно быть в идеале, но как часто это случается в реальности?

Если вы практикуете методику *разработки через тестирование*[1] (Test-Oriented Development, или TOD), то должны знать, что написание тестов — это менее утомительное занятие, чем может показаться. Ведь в любом случае вам приходится писать тесты в процессе написания кода. И каждый раз после разработки очередного компонента вы должны создать фрагмент кода (возможно, в конце этого же файла с классом), в котором выполняется создание экземпляра объекта и вызываются его методы. Если теперь взять и собрать воедино все эти разбросанные фрагменты кода, которые вы написали для тестирования компонента в процессе его разработки, то у вас получится тестовый пример (test case). Сохраните его в виде класса и добавьте в ваш набор тестов.

Очень странно, что в процессе написания кода разработчики часто избегают *запуска* тестов. В конце концов, через какое-то время для прогонки тестов вам понадобится гораздо больше времени. Отказы, связанные с известными проблемами, накапливаются как снежный ком, что затрудняет выявление новых проблем. Более того, вы подозреваете, что кто-то поместил в хранилище код, который не проходит тестирования, и у вас просто нет времени на то, чтобы прервать собственную работу и заняться исправлением ошибок, внесенных другими разработчиками. Поэтому гораздо лучше сразу запустить несколько тестов, непосредственно относящихся к вашей работе, чем потом тестировать все в комплексе.

При сбое во время запуска тестов вы не сможете устранить проблемы, которые эти тесты обнаружат, а это, в свою очередь, еще больше затрудняет выявление причин ошибок. Основные усилия при поиске ошибок обычно тратятся на диагностику проблемы, а не на ее устранение. Очень часто на устранение проблемы нужно потратить всего пару минут, тогда как на ее выявление — несколько часов. Если же тест завершился неудачей сразу или по прошествии нескольких часов после внесения исправления в код, вы сможете очень быстро определить то место, где нужно искать причину.

Аналогичные проблемы возникают и при построении приложения. 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 такими тестовыми инсталляциями, тем сложнее будет обнаружить причину этого отказа.

Чаще всего причина этого очень проста: необъявленная зависимость к некоторой библиотеке, установленной на вашем компьютере, либо один или несколько классов, которые вы забыли включить в пакет. Конечно же, это все легко устранить, если вы находитесь рядом. Но вот что делать, если процесс установки закончился

[1] Этот термин мне больше нравится, чем *сначала тестирование (Test-First Development, или TFD)*, поскольку он лучше отражает реалии большинства хороших проектов, в которых мне довелось принимать участие.

неудачно, а вы в это время уехали в отпуск? Тот неудачник, 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 ко всем членам вашего коллектива, но ведь всем хорошо известно, что это именно они, а не вы забыли запустить 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 по крайней мере свести их к минимуму) за счет автоматизации процессов построения и тестирования.

НИ — это одновременно и набор методик, и набор средств. С точки зрения методики, требуется частая фиксация кода в хранилище (по крайней мере раз в день). При каждой фиксации должны быть запущены соответствующие тесты и выполнено построение всех требуемых пакетов. Что касается средств, требуемых для НИ, то с некоторыми из них (в частности, с PHPUnit и Phing) вы уже знакомы. Хотя самостоятельных средств часто недостаточно. Для координации и автоматизации всех процессов здесь требуется система высокого уровня — сервер НИ. Без него очень высока вероятность, что все методики НИ будут сведены на нет и подчинены нашему природному желанию обходить трудности. В конце концов, мы ведь всегда предпочитаем кодирование другим видам деятельности.

Поддерживая систему, подобную описанной выше, вы получаете три очевидных преимущества. Главное из них — построение и тестирование ваших проектов будет выполняться часто. Собственно, это и является конечной целью НИ. Сам факт автоматизации процессов привносит два других преимущества. Тестирование и построение проектов будет выполняться в другой среде, которая отличается от вашей среды разработки. Причем все это будет происходить в фоновом режиме, и от вас не потребуется отвлекаться от своей работы и запускать тесты. Кроме того, как и в случае с тестированием, НИ способствует улучшению структуры проекта в целом. Для того чтобы можно было автоматизировать процесс инсталляции на удаленном компьютере, вы должны с самого начала упростить его, насколько это возможно.

Я не могу точно сказать, сколько раз мне приходилось сталкиваться с проектами, в которых процедура инсталляции была тайной, покрытой мраком, и была известна только нескольким разработчикам. "Ты говоришь, что не можешь задать правила замены URL? — спросил меня один из опытных разработчиков с едва скрываемым чувством раздражения. — 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 инсталляцию системы. Это означает, что нам нужно потратить немного времени на начальном этапе работы, зато потом эти усилия окупятся сторицей и сделают нашу жизнь легче. Намного легче!

Итак, для начала я собираюсь описать некоторые из этих сложных подготовительных операций. По сути, в этой главе вам придется с ними столкнуться.

Подготовка проекта для непрерывной интеграции

Конечно, прежде всего мне нужен проект, который должен быть непрерывно интегрирован. Поскольку я по жизни лентяй, мне нужно поискать фрагменты кода, для которых уже написаны тесты. Самый очевидный кандидат — это проект, который я

создал в главе 18 для иллюстрации PHPUnit. Я собираюсь назвать его `userthing`, поскольку это сущность (*thing*), содержащая объект `User`.

Прежде всего познакомимся со структурой каталогов моего проекта (рис. 20.1)

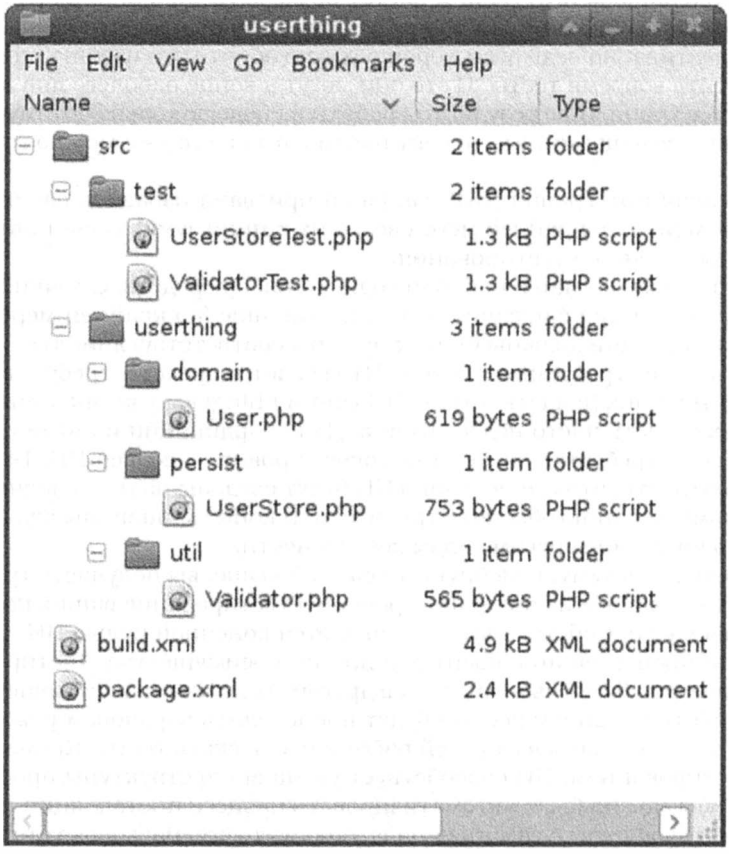


Рис. 20.1. Часть демонстрационного проекта, иллюстрирующего возможности НИ

Как видите, я немного привел ее в порядок и добавил несколько каталогов для пакетов. Внутри кода я поддерживаю пакетную структуру на основе пространства имен.

После того как у меня появился проект, я должен добавить его в хранилище системы контроля версий.

Непрерывная интеграция и контроль версий

Для НИ система контроля версий играет важную роль. Система НИ должна уметь запрашивать самую свежую версию файлов проекта без вмешательства человека (по крайней мере после того, как все будет настроено).

Поэтому мне нужно создать хранилище для моего проекта на сервере Git, как это было описано в главе 17.

```
$ sudo mkdir /var/git/userthing
$ sudo chown git:git /var/git/userthing
```

```
$ sudo su git
$ cd /var/git/userthing
$ git --bare init
```

Здесь я сначала создал каталог `userthing` и назначил ему корректного владельца и группу. Затем от имени пользователя `git` я создал хранилище. Теперь самое время импортировать простую кодовую базу проекта `userthing` из локального хранилища, находящегося на моем рабочем компьютере.

```
$ cd /home/mattz/work/userthing
$ git init
$ git add .
$ git commit -m 'first commit'
$ git remote add origin git@getinstance.com:/var/git/userthing
$ git push origin master
```

Здесь я сначала перешел в мой каталог разработки и проинициализировал локальное хранилище кода. Затем я добавил ветку `origin` на удаленном сервере, в которую я буду помещать код. Для проверки того, что все работает как нужно, выполним тестовое клонирование кода.

```
$ git clone git@getinstance.com:/var/git/userthing
```

```
Cloning into 'userthing'...
Enter passphrase for key '/home/mattz/.ssh/id_rsa':
remote: Counting objects: 61, done.
remote: Compressing objects: 100% (56/56), done.
remote: Total 61 (delta 14), reused 0 (delta 0)
Receiving objects: 100% (61/61), 9.09 KiB, done.
Resolving deltas: 100% (14/14), done.
```

Теперь у меня есть хранилище проекта `userthing` на удаленном сервере и локальная копия клона этого кода. Самое время автоматизировать процесс построения и тестирования.

Phing

Система автоматического построения проектов Phing была описана в главе 19. Ниже приведены команды, с помощью которых я установил библиотеку Phing.

```
$ pear channel-discover pear.phing.info
$ pear install phing/phing
```

Я собираюсь использовать этот замечательный инструмент для построения проектов в качестве связующего звена для среды НИ моих проектов. Мне нужно создать отдельные задачи для построения и тестирования кода, а также для запуска всех остальных утилит проверки качества, о которых пойдет речь в этой главе.

На заметку. При установке пакета не забудьте проанализировать сообщения, выводимые системой PEAR. Иногда перед установкой основного пакета вам может быть выдан запрос на установку некоторых зависимых пакетов. Команда `pear` выполняет полезную работу и всегда информирует вас о том, какие действия вы должны предпринять для установки зависимостей. Если вы не будете обращать никакого внимания на выводимые сообщения, то легко упустите из виду, какие именно действия требует выполнить от вас программа.

Давайте создадим базовый файл построения.

```
<project name="userthing" default="build">
   <property name="build" value="./build" />
```

```
   <property name="src" value="./src" />
   <property name="version" value="1.1.1" />

   <target name="build">
      <mkdir dir="${build}" />
      <copy todir="${build}/userthing">
         <fileset dir="${src}/userthing">
         </fileset>
      </copy>

      <copy todir="${build}/test">
         <fileset dir="${src}/test">
         </fileset>
      </copy>

   </target>

   <target name="clean">
      <delete dir="${build}" />
   </target>

   <!-- ... -->
</project>
```

Здесь я задал значения трех свойств проекта. В свойстве `build` указывается каталог, в котором выполняется предварительная сборка файлов проекта перед созданием пакета. Именно в нем я буду в конечном итоге запускать тесты и другие программы, о которых идет речь в этой главе. В свойстве `src` указывается каталог исходных файлов, а свойство `version` определяет номер версии пакета.

В задаче `build` копируются каталоги `userthing` и `test` в каталог построения проекта (определенный в свойстве `build`). В более сложных проектах на этом этапе мне бы потребовалось выполнять различные преобразования данных, создавать "на лету" файлы конфигурации и собирать различные двоичные данные. Эта задача будет вызвана по умолчанию для нашего проекта при запуске команды `phing` без параметров.

В задаче `clean` удаляются каталог построения и все его содержимое. Теперь давайте запустим процесс построения.

```
$ phing
```

```
Buildfile: /home/mattz/work/userthing/build.xml
userthing > build:
  [mkdir] Created dir: /home/mattz/work/userthing/build
   [copy] Created 4 empty directories in /home/mattz/work/userthing/build/
userthing
   [copy] Copying 3 files to /home/mattz/work/userthing/build/userthing
   [copy] Created 1 empty directory in /home/mattz/work/userthing/build/test
   [copy] Copying 2 files to /home/mattz/work/userthing/build/test

BUILD FINISHED

Total time: 0.7534 seconds
```

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 с помощью PHPUnit было описано в главе 18. Хотя, если вы еще не читали эту главу, вам непременно следует установить это замечательное средство, перед тем как двигаться дальше[2].

```
$ pear config-set auto_discover 1
$ pear install --alldeps pear.phpunit.de/phpunit
```

Кроме того, в главе 18 я написал тесты для одной из версий кода проекта `userthing`, который я собираюсь использовать в данной главе. Здесь же мне нужно запустить эти тесты снова (из каталога `src`), чтобы убедиться в том, что в процессе реорганизации ничто не перестало работать.

```
$ phpunit test
```

```
PHPUnit 3.7.24 by Sebastian Bergmann.
.....
Time: 317 ms, Memory: 3.75Mb
OK (5 tests, 5 assertions)
```

Как видите, запуск теста подтвердил, что все работает корректно. Хотя мне нужно будет запускать тест из программы Phing. Для этого в ней предусмотрена задача `exec`, которую мы будем использовать для вызова команды `phpunit`. В любом случае всегда лучше всего использовать специализированное средство, если оно предусмотрено в программе. Ниже приведено определение встроенной задачи для выполнения этих действий.

```
<target name="test" depends="build">
   <phpunit>
      <formatter type="plain" usefile="false"/>
      <batchtest classpath="${build}">
          <fileset dir="${build}/test">
             <include name="**/*Test.php"/>
          </fileset>
      </batchtest>
   </phpunit>
</target>
```

Как видите, задача `test` зависит от задачи `build`. Это гарантирует, что перед запуском тестов в каталоге `build` в нем будут присутствовать все необходимые для этой цели файлы. Помимо остальных атрибутов, в задаче `phpunit` можно указать атрибут `printsummary`, который выводит общее состояние выполненных тестов.

Большая часть функциональных возможностей задачи сконфигурирована с использованием вложенных элементов. Элемент `formatter` влияет на способ генерации тестовых данных. В нашем случае я выбрал простой формат, пригодный для анализа человеком. Элемент `batchtest` позволяет определить несколько тестовых файлов с помощью вложенного элемента `fileset`. Обратите внимание, что в эле-

[2] Канал **`pear.phpunit.de`** был окончательно закрыт 31 декабря 2014 года. Теперь старая версия приложения PHPUnit устанавливается с помощью PEAR, как показано ниже, а новая просто скачивается с сайта `https://phpunit.de/` в виде архива phar. — *Примеч. ред.*

менте batchtest указан атрибут classpath. В нем определяется каталог, из которого будут подключаться тесты. Имя этого атрибута явно унаследовано из Java.

На заметку. Задача phpunit имеет большое количество параметров настройки. Все они описаны в справочном руководстве по Phing по адресу https://www.phing.info/docs/guide/stable/apcs59.html.

Ниже приведен результат запуска тестов из программы Phing.

```
$ phing test
```

```
Buildfile: /home/mattz/work/userthing/build.xml
userthing > build:
userthing > test:
   [phpunit] Testsuite: ValidatorTest
   [phpunit] Tests run: 2, Failures: 0, Errors: 0, Incomplete: 0, Skipped: 0, Time
elapsed: 0.07081 s
   [phpunit] Testsuite: UserStoreTest
   [phpunit] Tests run: 3, Failures: 0, Errors: 0, Incomplete: 0, Skipped: 0, Time
elapsed: 0.02879 s
   [phpunit] Total tests run: 5, Failures: 0, Errors: 0, Incomplete: 0, Skipped: 0,
   Time elapsed: 0.10772 s

BUILD FINISHED

Total time: 1.0921 second
```

Документация

Один из принципов НИ — это прозрачность. По этой причине очень важно поддерживать в актуальном состоянии документацию, в которой будут описаны все недавно добавленные классы и методы, особенно когда вы собираетесь выполнить построение проекта в среде НИ. В главе 16 был описан пакет phpDocumentor, так что вы уже знакомы с приведенными ниже командами.

```
$ pear channel-discover pear.phpdoc.org
$ pear install phpdoc/phpdocumentor
```

Давайте лучше запустим эту программу, чтобы все увидеть, но на этот раз из каталога build.

```
$ mkdir docs
$ phpdoc --directory=userthing --target=docs --title=userthing
--template=abstract
```

Эти команды сгенерируют довольно приличную документацию. Перед тем как опубликовать ее на сервере НИ, мне пришлось попотеть над тем, чтобы написать несколько реальных блоков документации.

Как и раньше, мне нужно добавить эти команды в файл build.xml. Хотя для интеграции с PHPDocumentor в Phing была предусмотрена специальная встроенная задача phpdoc2, на момент написания этих строк она работала некорректно. Тем не менее особой беды в этом нет. Я вполне могу обойтись задачей exec для запуска из командной строки утилиты phpdoc, как показано ниже.

На заметку. Вполне возможно, что, когда вы будете читать эти строки, задача phpdoc2 снова корректно заработает с утилитой PHPDocumentor. Ее описание приведено в справочном руководстве по Phing по адресу https://www.phing.info/docs/guide/stable/apcs56.html.

```
<target name="doc" depends="build">
   <mkdir dir="${build}/docs" />
   <exec executable="phpdoc" dir="${build}">
      <arg line="--directory=userthing --target=docs --title=userthing
--template=abstract" />
   </exec>
</target>
```

И снова мы видим, что задача `doc` зависит от задачи `build`. Сначала создается каталог `build/docs` для вывода файлов, а затем запускается задача `exec`. Команда `phpdoc` указана в атрибуте `executable`. Атрибут `dir` определяет каталог, в котором будет запущена указанная команда. Для элемента `exec` указывается вложенный элемент `arg`, с помощью которого команде `phpdoc` передаются параметры командной строки. Если требуется передать в командной строке единственный аргумент, то его можно указать в атрибуте `value` элемента `arg`. В нашем же случае нужно передать список аргументов, поэтому мы воспользовались атрибутом `line`, а сами аргументы разделили пробелами.

На заметку. Параметры задачи `exec` полностью описаны в справочном руководстве по Phing по адресу `https://www.phing.info/docs/guide/stable/apbs17.html`.

Покрытие кода

Не имеет смысла доверять тестам, если они не применяются к написанному вами коду. В PHPUnit включена возможность создавать отчеты по покрытию кода (code coverage). Ниже приведена выдержка из инструкции по использованию PHPUnit.

```
--coverage-html   <dir>   Сгенерировать отчет по покрытию кода
                            в формате HTML.
--coverage-clover <file>  Записать данные по покрытию кода
                            в формате Clover XML.
```

Для того чтобы воспользоваться этой возможностью, вы должны установить расширение Xdebug. Подробную информацию по этому вопросу можно найти на сайте `http://pecl.php.net/package/Xdebug`, а информацию по инсталляции — на `http://xdebug.org/docs/install`. Данное расширение можно также установить напрямую, воспользовавшись системой управления дистрибутивными пакетами Linux. Например, приведенные ниже команды должны работать в Fedora.

```
$ yum install php-pecl-xdebug
```

Ниже я запустил PHPUnit из каталога `src` с включенной опцией покрытия кода.

```
$ mkdir /tmp/coverage
$ phpunit --coverage-html /tmp/coverage test
```

```
PHPUnit 3.7.24 by Sebastian Bergmann.
.....
Time: 3.36 seconds, Memory: 4.75Mb

OK (5 tests, 5 assertions)
Generating code coverage report in HTML format ... done
```

После этого вы можете взглянуть на отчет, открыв его в окне браузера (рис. 20.2).

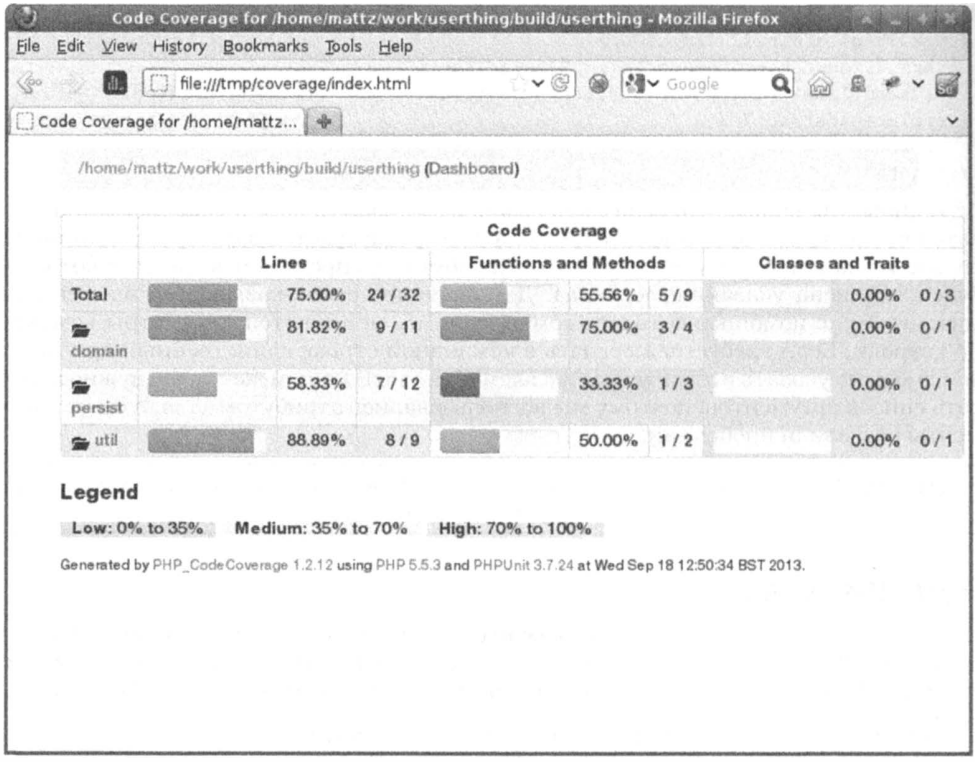


Рис. 20.2. Отчет по покрытию кода

Важно отметить, что достижение полного покрытия кода не означает, что было проведено в достаточной степени тестирование системы. С другой стороны, всегда полезно знать обо всех недочетах, выявленных в ваших тестах. Как видно из отчета, показанного на рис. 20.2, мне еще нужно немного потрудиться.

Убедившись в том, что отчет по покрытию кода работает из командной строки, внесем эту функциональную возможность в файл построения, как показано ниже.

```
<target name="citest" depends="build">
   <mkdir dir="${build}/reports/coverage" />

   <coverage-setup database="${build}/reports/coverage.db">
      <fileset dir="${build}/userthing">
         <include name="**/*.php"/>
      </fileset>
   </coverage-setup>

   <phpunit codecoverage="true">
      <formatter type="plain" usefile="false"/>
      <formatter type="xml" outfile="testreport.xml" todir="${build}/reports" />
      <formatter type="clover" outfile="cloverreport.xml" todir="${build}/
reports" />
      <batchtest classpath="${build}">
         <fileset dir="${build}/">
```

```
            <include name="test/**"/>
         </fileset>
      </batchtest>
   </phpunit>

   <coverage-report outfile="${build}/reports/coverage.xml">
      <report todir="${build}/reports/coverage" />
   </coverage-report>
</target>
```

Здесь я создал новую задачу под именем `citest`. Большая часть файла построения для этой задачи попросту воспроизводит задачу `test`, которая была рассмотрена выше. В начале создается каталог `reports` и в нем подкаталог `coverage`. Для создания параметров конфигурации элемента `coverage` используется задача `coverage-setup`. В ней с помощью атрибута `database` указано место хранения необработанных данных по покрытию кода. Во вложенном элементе `fileset` определены файлы, которые должны быть подвержены анализу по покрытию кода.

В задачу `phpunit` я добавил три элемента `formatter`. Элемент `formatter`, имеющий тип `xml` предназначен для генерирования файла `testreport.xml`, содержащего результаты тестирования. Элемент `formatter`, имеющий тип `clover`, также генерирует данные о покрытии кода в формате XML.

И в самом конце определения задачи `citest` приводится в действие задача `coverage-report`. При этом берется готовая информация о покрытии кода, генерируется новый XML-файл и создается отчет в формате HTML.

На заметку. Параметры задачи `coverage-report` полностью описаны в справочном руководстве по Phing по адресу `https://www.phing.info/docs/guide/stable/apcs05.html`.

Стандарты кодирования

Я могу спорить дни напролет о том, где лучше всего ставить фигурные скобки, как лучше делать отступы в коде (с помощью табуляции или с помощью пробелов) или как лучше всего именовать закрытые переменные свойств. Правда, было бы здорово, если бы можно было навязать все мои предпочтения и перенести их в код с помощью какого-нибудь инструмента? Благодаря PHP_CodeSniffer я могу это сделать. CodeSniffer может применить один из наборов стандартов кодирования к проекту и сгенерировать отчет, в котором будет отражено, насколько ваш стиль кодирования соответствует принятым стандартам.

Все это может обернуться большими проблемами для разработчика. По сути, так оно и бывает. Однако существует разумное применение средствам, подобным описанному выше. И я собираюсь это продемонстрировать, однако не будем забегать вперед. Итак, сначала — инсталляция.

```
$ sudo pear install PHP_CodeSniffer
```

А теперь я собираюсь применить стандарты кодирования Zend к моему коду.

```
$ phpcs --standard=Zend build/userthing/persist/UserStore.php
```

```
FILE: ...userthing/build/userthing/persist/UserStore.php
--------------------------------------------------------------------------------

FOUND 9 ERROR(S) AFFECTING 8 LINE(S)
--------------------------------------------------------------------------------
 6 | ERROR | Opening brace of a class must be on the line after the definition
```

```
 7 | ERROR | Private member variable "users" must contain a leading underscore
 9 | ERROR | Opening brace should be on a new line
13 | ERROR | Closing parenthesis of a multi-line function call must be on a
   |       | line by itself
...
```

На заметку. Один из технических рецензентов этого издания книги, Вес Хант, сообщил о некоторых проблемах с установкой CodeSniffer в системе Windows: "Поскольку Pear у меня был уже установлен, для успешной инсталляции CodeSniffer мне пришлось почистить папку `C:\Users\{username}\AppData\Local\Temp\pear\cache`".

Совершенно очевидно, для того чтобы зафиксировать мой код в хранилище Zend, мне нужно соответствующим образом изменить свой стиль кодирования! Тем не менее при работе в команде имеет смысл определить правила оформления кода. По сути, решение, *какой* набор правил выбрать, вероятно, менее важное, чем решение, *каких* общих стандартов придерживаться сначала. Если сохранена целостность кодовой базы, то сам код легко читать и с ним легко работать. Например, принятое соглашение по присвоению имен поможет вам сразу же выяснить назначение переменной или свойства.

Соглашения по оформлению кода помогают также уменьшить вероятность ошибки и избежать создания предрасположенного к ошибкам кода.

Тем не менее это все-таки опасная зона. Некоторые решения, связанные со стилем кодирования, очень субъективны, и люди часто с пеной у рта защищают свое право делать что-то так, а не иначе. Пакет CodeSniffer позволяет определить собственный набор правил. Поэтому я надеюсь, что при их создании вы прислушаетесь к мнению коллектива и тем самым облегчите жизнь разработчикам, а не усложните ее до предела.

Еще одно преимущество автоматических средств заключается в их беспристрастной и безликой природе. Поэтому, если члены вашего коллектива решат изменить стиль оформления кода, очевидно, лучше иметь лишенный чувства юмора сценарий, который скорректирует ваш стиль, чем такого же сотрудника, выполняющего те же действия.

Как и следовало ожидать, сейчас я добавлю задачу `codesniffer` в свой файл построения.

```
<target name="sniff" depends="build">
   <mkdir dir="${build}/reports" />
   <phpcodesniffer standard="Zend">
      <fileset dir="${build}/userthing">
         <include name="**/*.php"/>
      </fileset>
      <formatter type="checkstyle" outfile="${build}/reports/checkstyle.xml"/>
      <formatter type="default" usefile="false"/>
   </phpcodesniffer>
</target>
```

Задача `phpcodesniffer` сделает всю описанную выше работу вместо меня. В ней я воспользовался атрибутом `standard`, чтобы указать набор правил `Zend`. Набор файлов для проверки определен с помощью вложенного элемента `fileset`. Здесь я определил два элемента `formatter`. В первом из них для атрибута `type` выбрано значение `checkstyle`, а файлы отчета в формате XML будут записываться в каталог `reports`. Во втором элементе `formatter` для атрибута `type` выбрано значение `default`, а сводная информация об отчете выводится в командную строку.

На заметку. Для задачи `codesniffer` можно создать собственные правила. Как это сделать, описано на вебсайте PEAR по адресу `http://pear.php.net/manual/en/package.php.php-codesniffer.coding-standard-tutorial.php`.

Построение пакета

После тестирования кодовой базы и запуска различных тестовых утилит нужно также убедиться, что процесс построения пакета проходит без ошибок. Для создания пакета PEAR воспользуемся задачей `pearpkg2`, как показано ниже.

```
<target name="makepackagefile" depends="build">
   <pearpkg2 name="userthing" dir="${build}">
      <option name="packagefile" value="userthing_package.xml"/>
      <option name="packagedirectory" value="${build}"/>
      <option name="baseinstalldir" value="/"/>
      <option name="channel" value="pear.php.net"/>
      <option name="summary" value="blah blah"/>
      <option name="description" value="blah blah blah"/>
      <option name="apiversion" value="1.1.0"/>
      <option name="apistability" value="beta"/>
      <option name="releaseversion" value="${version}"/>
      <option name="releasestability" value="beta"/>
      <option name="license" value="none"/>
      <option name="phpdep" value="5.4.0"/>
      <option name="pearinstallerdep" value="1.4.6"/>
      <option name="packagetype" value="php"/>
      <option name="notes" value="notes notes notes"/>
      <mapping name="maintainers">
         <element>
            <element key="handle" value="mattz"/>
            <element key="name" value="matt"/>
            <element key="email" value="matt@getinstance.com"/>
            <element key="role" value="lead"/>
         </element>
      </mapping>
      <fileset dir="${build}">
         <include name="userthing/**" />
      </fileset>
   </pearpkg2>
</target>
```

Анализ многочисленных элементов в этом листинге не должен вызвать у вас затруднения, поскольку они достаточно очевидны, особенно после освежения в памяти структуры пакетного файла, которая была приведена в главе 15. Обратите внимание, что для установки значения `releaseversion` элемента `option` я воспользовался значением свойства `version`, которое было задано в самом начале файла построения. Также обратите внимание, что я указываю, какие именно файлы должны быть включены в пакет с помощью элемента `fileset`.

На заметку. Параметры задачи `pearpkg2` полностью описаны в справочном руководстве по Phing по адресу https://www.phing.info/docs/guide/stable/apcs47.html.

Благодаря наличию опции `packagefile` задача `makepackagefile` создаст файл `userthing_package.xml`. Для того чтобы преобразовать его в реальный пакетный файл, мне нужно снова обратиться к задаче `exec`.

```
<target name="buildpackage" depends="makepackagefile">
   <exec dir="build" checkreturn="true" executable="/usr/bin/pear">
      <arg value="package" />
      <arg value="userthing_package.xml" />
   </exec>
</target>
```

Эта задача `exec` эквивалентна вызову команды

```
$ pear package userthing_package.xml
```

из каталога `build`. В результате будет создан архивный файл `userthing-1.1.1.tgz`.

На заметку. Архивный пакетный файл можно также создать с помощью задачи `tartask`, которая описана в справочном руководстве по Phing по адресу `https://www.phing.info/docs/guide/stable/apcs81.html`.

Обратите внимание, что в элементе `exec` используется атрибут `checkreturn`. Он заставляет программу Phing выдать предупреждение, если одна из вызываемых им команд завершается с отличным от нуля кодом возврата. Без этого атрибута вы так никогда и не узнаете, что задача `exec` аварийно завершилась. Такое может произойти, например в случае, если команда `pear` не будет найдена в указанной мною папке. Теперь давайте запустим задачу `buildpackage`.

```
$ phing buildpackage
```

```
Buildfile: /home/mattz/work/userthing/build.xml

userthing > build:

userthing > makepackagefile:

[pearpkg2] Creating [default] package.xml file in base directory.
   Analyzing userthing/domain/User.php
   Analyzing userthing/persist/UserStore.php
   Analyzing userthing/util/Validator.php

userthing > buildpackage:

BUILD FINISHED

Total time: 1.7685 second
```

Теперь давайте протестируем пакет, установив его.

```
$ pear install --force build/userthing-1.1.1.tgz
```

```
install ok: channel://pear.php.net/userthing-1.1.1
```

Итак, в моем распоряжении есть несколько полезных инструментов, с помощью которых я могу протестировать проект. Разумеется, такие личности, как я, довольно скоро могут утратить интерес к запуску тестов, даже имея под рукой такой замечательный файл построения Phing. Вероятнее всего, я воспользуюсь старой идеей по поводу фазы интеграции и буду применять эти средства только на стадии заверше-

ния проекта. К тому времени их эффективность как системы раннего предупреждения будет сведена на нет. Поэтому мне нужен сервер интеграции, который запускал бы все тесты вместо меня.

Сервер НИ Jenkins

Jenkins — это сервер непрерывной интеграции с открытым исходным кодом, ранее называвшийся "Hudson". Несмотря на то что он написан на языке Java, его довольно легко использовать вместе с утилитами PHP. Все дело в том, что сервер НИ никак не связан с теми проектами, построение которых выполняется. Он просто запускает разнообразные команды и отображает получаемый при этом результат. Сервер Jenkins также очень хорошо интегрируется с проектами на PHP, поскольку в нем предусмотрен интерфейс для работы с дополнительными модулями. Кроме того, существует очень активное сообщество разработчиков, поддерживающих средства расширения основных функциональных возможностей этого сервера.

На заметку. Почему я выбрал сервер НИ Jenkins? Все дело в том, что он прост в использовании и его легко расширять. К тому же он имеет солидную репутацию и активное сообщество пользователей. Сервер Jenkins совершенно бесплатный и имеет открытый исходный код. К нему созданы дополнительные модули, поддерживающие интеграцию с PHP, а также большинство утилит для построения и тестирования проекта, которые могут вам понадобиться. Однако стоит отметить, что на сегодняшний день существует несколько серверов НИ. В предыдущем издании книги был описан сервер CruiseControl (`http://cruisecontrol.sourceforge.net/`), который также является хорошим выбором. Если вам нужен сервер, полностью реализованный на PHP, тогда обратите внимание на Xinc (`http://code.google.com/p/xinc/`).

Установка сервера Jenkins

Поскольку сервер НИ Jenkins написан на языке Java, для его работы требуется установленная среда JRE этого языка. Установка этой среды зависит от используемой компьютерной платформы. Так, при использовании дистрибутива Linux Fedora это делается с помощью приведенной ниже команды.

```
$ yum install java
```

На других платформах среду Java можно загрузить непосредственно с сайта `www.java.com`. Чтобы убедиться, что среда Java установлена и нормально функционирует, введите приведенную ниже команду.

```
$ java -version
```

```
openjdk version "1.8.0_40"
OpenJDK Runtime Environment (build 1.8.0_40-b25)
OpenJDK 64-Bit Server VM (build 25.40-b25, mixed mode)
```

Сервер Jenkins можно загрузить прямо с сайта `http://jenkins-ci.org/`. Его можно установить в виде файла веб-архива Java (Java Web Archive, или WAR) либо можно загрузить готовый пакет для используемой вами операционной системы. Все ссылки приведены на начальной странице сайта проекта Jenkins. Поскольку я использую дистрибутив Linux Fedora, мне нужно ввести следующие команды.

```
$ wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/
jenkins.repo
$ rpm --import http://pkg.jenkins-ci.org/redhat/jenkins-ci.org.key
$ yum install jenkins
```

На сайте проекта Jenkins приведены инструкции по установке для большинства популярных платформ. После установки сервер Jenkins запускается непосредственно из командной строки с помощью утилиты `java`. как показано ниже.

```
$ sudo java -jar /usr/lib/jenkins/jenkins.war
```

Следует заметить, что, запустив сервер подобным образом, вы в будущем можете столкнуться с определенными проблемами безопасности. Поэтому, очевидно, что лучше всегда использовать специальный сценарий, который запустит сервер Jenkins от имени пользователя `jenkins`, а не `root`. В случае использования Fedora сервер НИ Jenkins запускается так, как показано ниже.

```
$ service jenkins start
```

Загрузить универсальный сценарий запуска сервера НИ можно по адресу: `https://wiki.jenkins-ci.org/display/JENKINS/JenkinsLinuxStartupScript`. По умолчанию сервер `Jenkins` после запуска начинает прослушивать порт 8080 (в некоторых операционных системах, например FreeBSD, используется порт 8180). Чтобы проверить, работает ли сервер НИ на вашем компьютере, посетите в веб-браузере страницу `http://`*`адрес_хоста`*`:8080/` (в некоторых ОС, например в FreeBSD, в адресной строке браузера нужно ввести `http://`*`адрес_хоста`*`:8180/jenkins`). Вы должны увидеть на экране панель управления сервером, показанную на рис. 20.3.

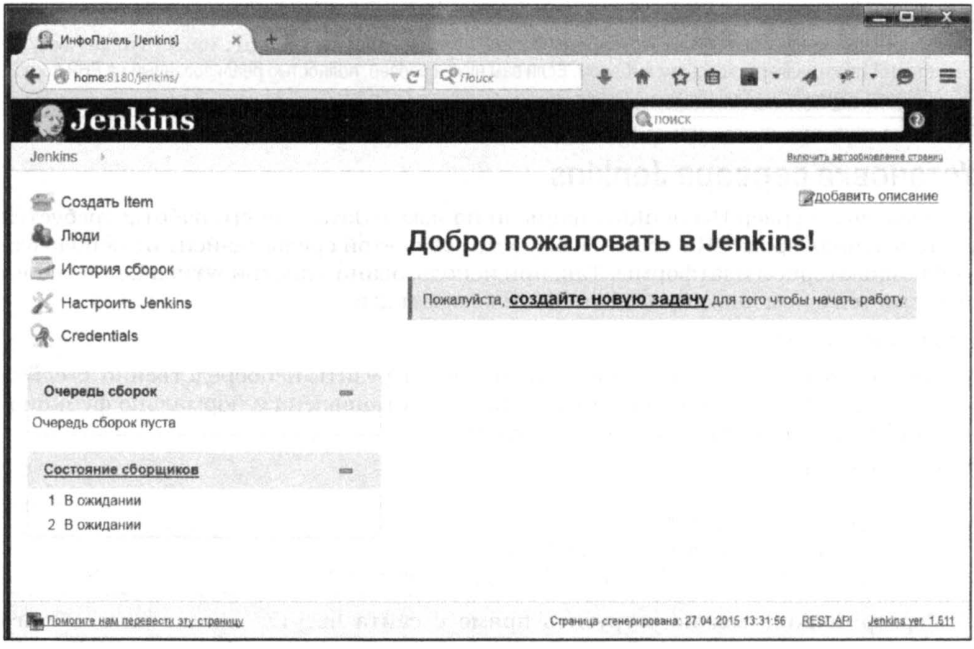


Рис. 20.3. Панель управления сервером НИ Jenkins

Установка дополнительных модулей сервера Jenkins

Сервер НИ Jenkins является полностью конфигурируемым программным продуктом. Поэтому для поддержки описанных выше возможностей интеграции с проектами на PHP мне понадобится установить несколько дополнительных модулей.

Для этого в окне веб-интерфейса Jenkins щелкните на ссылке **Настроить Jenkins (Manage Jenkins)**, а затем — на **Управление плагинами (Manage Plugins)**. Перейдите на вкладку **Доступные (Available)**, там вы увидите длинный список дополнительных модулей для сервера НИ Jenkins, которые можно установить. В столбце **Установить (Install)** отметьте флажком нужные нам модули, которые должны быть установлены. Их список приведен в табл. 20.1.

Таблица 20.1. Необходимые дополнительные модули сервера Jenkins

Модуль	Описание
Git Plugin	Поддерживает интеграцию с хранилищем Git
xUnit Plugin	Поддерживает интеграцию с утилитами семейства xUnit, включая PHPUnit
Phing Plugin	Позволяет использовать Phing для построения PHP-проектов
Clover PHP Plugin	Обеспечивает доступ к служебному файлу XML и файлам HTML, созданным PHPUnit, и создает отчет
HTML Publisher Plugin	Средство интеграции отчетов в формате HTML, используется для анализа данных, выводимых программой PHPDocumentor
Checkstyle Plugin	Обеспечивает доступ к файлу XML, созданному PHPCodeSniffer, а также генерирует отчеты

Список установленных дополнительных модулей приведен на рис. 20.4.

После установки перечисленных выше дополнительных модулей почти все готово для создания и настройки моего проекта.

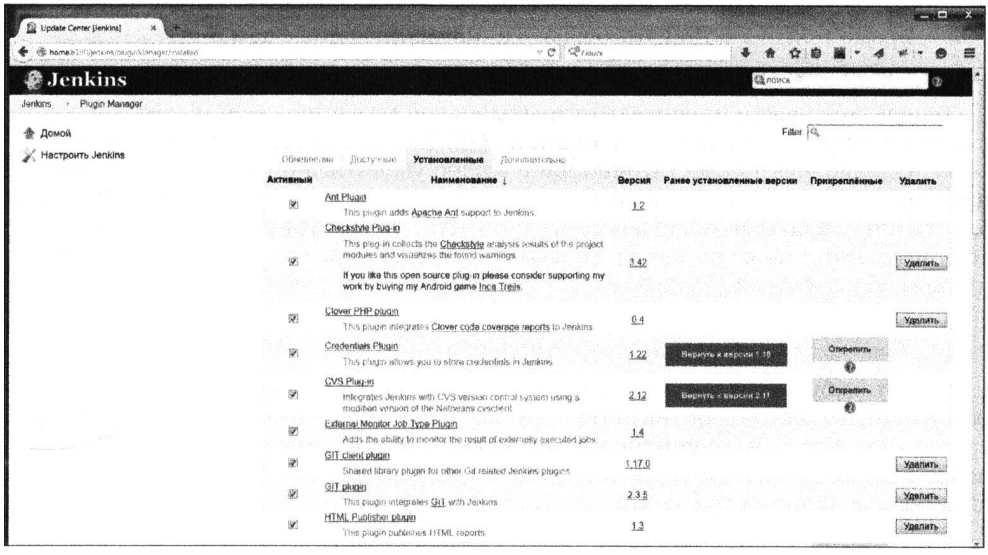


Рис. 20.4. Экран с установленными дополнительными модулями сервера Jenkins

Установка открытого ключа Git

Перед использованием модуля Git plugin необходимо убедиться, что сервер НИ Jenkins имеет доступ к хранилищу Git. В главе 17 был описан процесс генерации открытого ключа для доступа к этому удаленному хранилищу. Теперь нам нужно повторить данный процесс в домашнем каталоге сервера Jenkins. Но где именно он находится?

Расположение домашнего каталога можно изменить, но обычно его легко узнать, посмотрев настройки в самом сервере Jenkins. Для этого щелкните сначала на ссылке Настроить Jenkins (Configure Jenkins), а затем — на ссылке Конфигурирование системы (Configure System). В открывшемся окне вы увидите путь к домашнему каталогу сервера Jenkins. Кроме того, можно просмотреть содержимое файла `/etc/passwd` и узнать путь к домашнему каталогу для пользователя `jenkins`. В моем случае это `/usr/local/jenkins`.

Теперь нам нужно сконфигурировать ключи и поместить их в подкаталог `.ssh`, как показано ниже.

```
$ sudo su jenkins -s /bin/bash
$ cd ~
$ mkdir .ssh
$ chmod 0700 .ssh
$ ssh-keygen
```

Здесь я сначала переключился на учетную запись пользователя `jenkins` и указал в командной строке путь к оболочке, поскольку по умолчанию доступ к системной оболочке может быть отключен. Далее я перешел в домашний каталог этого пользователя, создал в нем подкаталог `.ssh` и установил права доступа к нему, запрещающие доступ других пользователей и групп. С помощью команды `ssh-keygen` я сгенерировал ключи для ssh. Когда на экране появилось приглашение ввести пароль, я просто нажал клавишу <Enter>. В результате доступ к серверу Jenkins будет осуществляться только по его ключу. Также нужно убедиться в том, что созданный программой файл `.ssh/id_rsa` недоступен по чтению для всех остальных пользователей, кроме текущего. Для этого я ввел приведенную ниже команду.

```
$ chmod 0600 .ssh/id_rsa
```

Теперь мне нужно прочитать открытый ключ из файла `.ssh/id_rsa.pub` и добавить его к моему удаленному хранилищу Git. Как это сделать, было описано в главе 17. Однако этого недостаточно. Мне нужно убедиться в том, что мой сервер Git находится в списке разрешенных хостов для связи по протоколу SSH. Чтобы добавить ключ в список и заодно протестировать конфигурацию сервера Git, введите приведенные ниже команды. Только не забудьте перед этим войти в систему под именем пользователя `jenkins`.

```
$ cd /tmp
$ git clone git@appulsus.com:/var/git/userthing
```

При этом будет выдан запрос на мое согласие на добавление хоста Git в список разрешенных хостов для связи по протоколу SSH. Его адрес будет записан в файл `.ssh/known_hosts`, находящийся в домашнем каталоге пользователя `jenkins`. Это позволит впоследствии исключить отключение сервера Jenkins при попытке создать подключение к серверу Git.

Создание и настройка проекта

Перейдите на панель управления сервером Jenkins и щелкните на ссылке, позволяющей создать новую задачу. В появившемся новом диалоговом окне введите имя проекта, `userthing`, как показано на рис. 20.5.

После щелчка на кнопке OK появится экран конфигурации проекта. Напомню, что моя первоочередная задача — подключиться к удаленному хранилищу Git. Для этого в разделе Управление исходным кодом (Source Code Manager) нужно щелкнуть на кнопке Git и ввести URL удаленного хранилища, как показано на рис. 20.6.

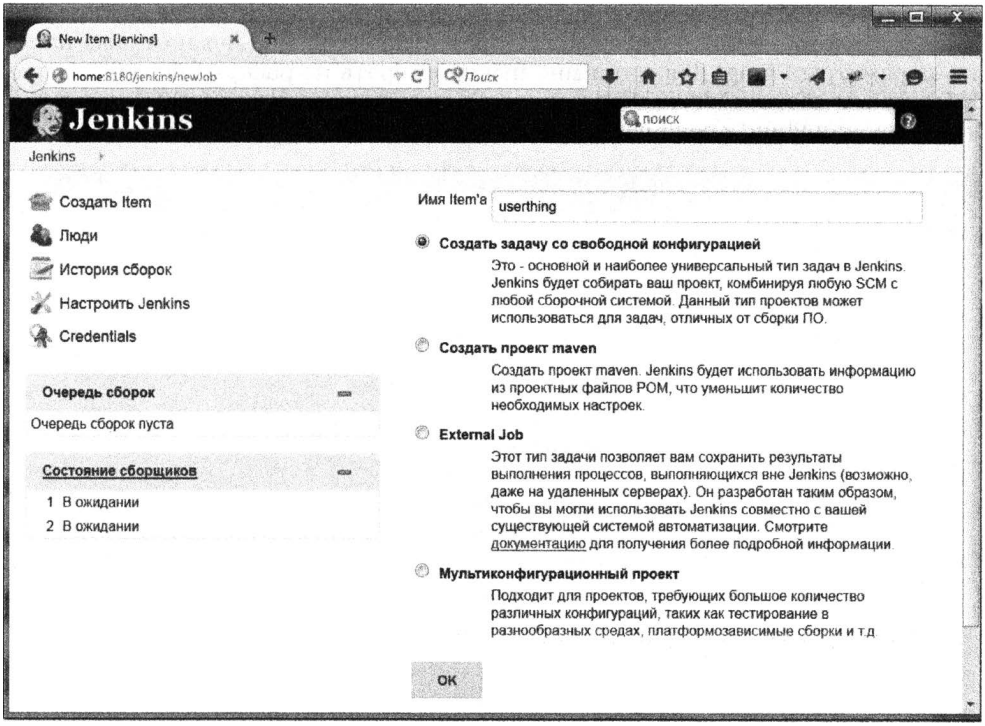


Рис. 20.5. Диалоговое окно создания проекта

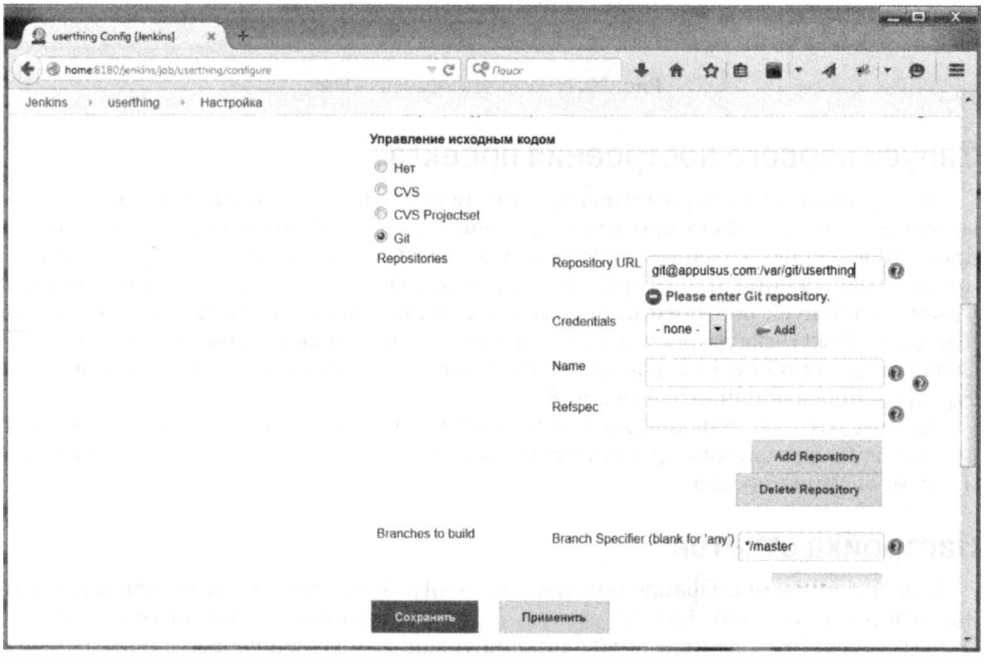


Рис. 20.6. Настройка подключения к удаленному хранилищу системы контроля версий

Если все пройдет успешно, я должен получить доступ к моему исходному коду. Однако для того чтобы полностью все протестировать, мне нужно также настроить параметры Phing. Для этого мне нужно выбрать из раскрывающегося списка **Добавить шаг сборки (Add build step)** опцию **Invoke Phing targets** (рис. 20.7), а затем добавить задачи Phing в появившемся текстовом поле.

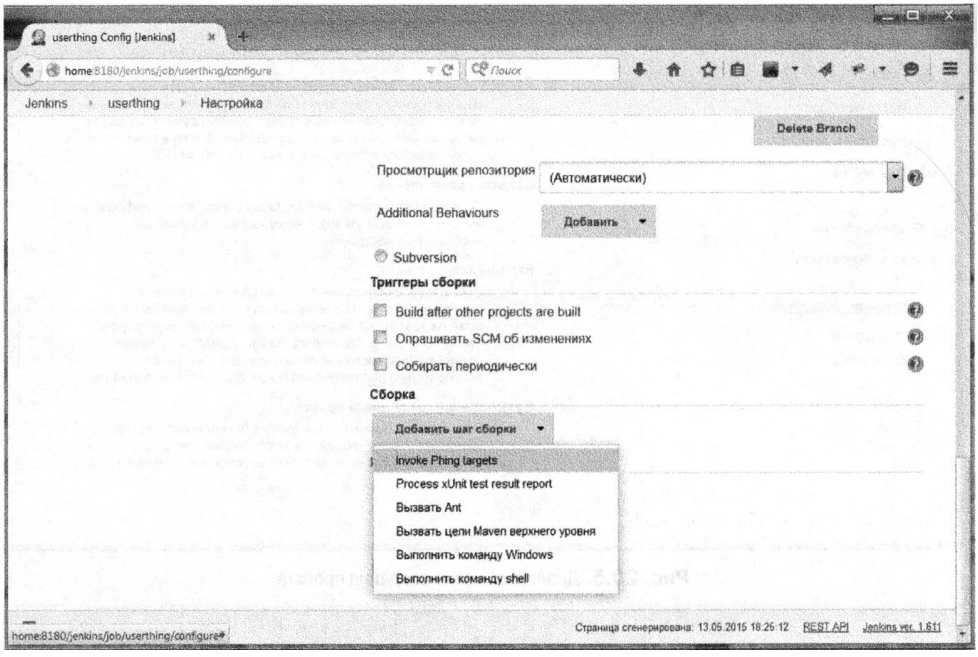


Рис. 20.7. Настройка параметров Phing

Запуск первого построения проекта

После сохранения параметров проекта перейдите на главный экран и в раскрывающемся списке меню проекта (стрелочка, расположенная справа от названия проекта) выберите команду **Собрать сейчас (Build Now)**. В результате будет запущен процесс построения и тестирования проекта. Вот он настал — момент истины! Ссылка на запуск построения проекта должна появиться в списке хронологий построения **(Build History)** проекта. После щелчка на ней появится окно консоли **Console Output**, на которой будут отображены сообщения, подтверждающие, что процесс построения проекта начался (рис. 20.8).

Как видите, все завершилось успешно! Сервер Jenkins загрузил код проекта `userthing` из удаленного хранилища сервера Git и запустил все задачи построения и тестирования проекта.

Настройка отчетов

Как указано в моем файле построения, программа Phing должна сохранять файлы отчетов в каталоге `build/reports`, а документацию — в каталоге `build/docs`. Параметры установленных мною дополнительных модулей сервера Jenkins можно

изменить с помощью раскрывающегося списка Добавить шаг после сборки (Add post-build action), расположенного в окне настройки параметров проекта (рис. 20.9).

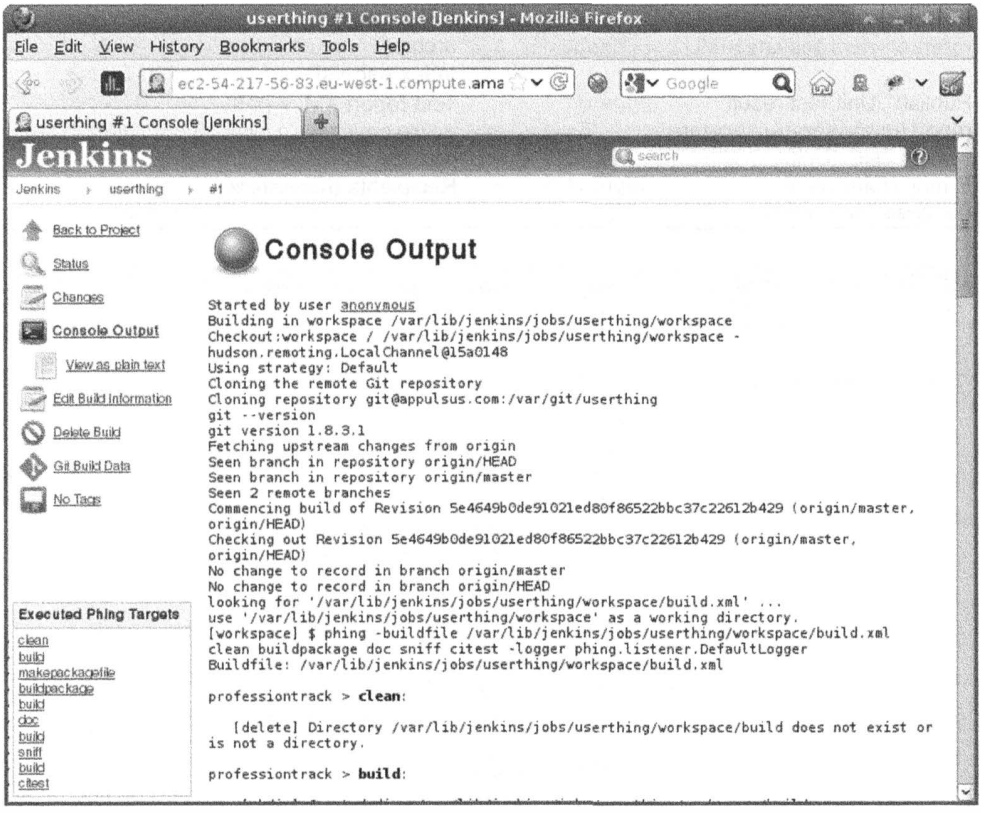


Рис. 20.8. Сообщения, выводимые на консоли

Чтобы не приводить здесь большое количество копий экрана, я решил свести все варианты конфигураций в таблицу и вкратце описать их. В табл. 20.2 приведены некоторые послесборочные шаги и соответствующие им значения, заданные в моем файле построения Phing.

Таблица 20.2. Настройка параметров отчета

Опция конфигурации	Задача Phing	Поле	Значение поля
Publish Checkstyle analysis results (Опубликовать результаты анализа стандартов кодирования)	`phpcodesniffer`	Checkstyle results	`build/reports/checkstyle.xml`
Publish Clover PHP Coverage Report (Опубликовать результаты анализа покрытия кода)	`phpunit`	Clover XML Location	`build/reports/cloverreport.xml`
		Clover HTML report directory	`build/reports/coverage/`

Окончание табл. 20.2

Опция конфигурации	Задача Phing	Поле	Значение поля
Publish HTML reports (Опубликовать отчеты в формате HTML)	exec или phpdoc2	HTML directory to archive	build/docs
		Index page[s]	index.html
Publish JUnit test result report (Опубликовать результаты тестирования JUnit)	phpunit	Test report XMLs (XML файлы с отчетами о тестировании)	build/reports/testreport.xml
E-mail Notification (Уведомление по почте)	phpunit	Recipients (Получатели)	someone@somemail.com

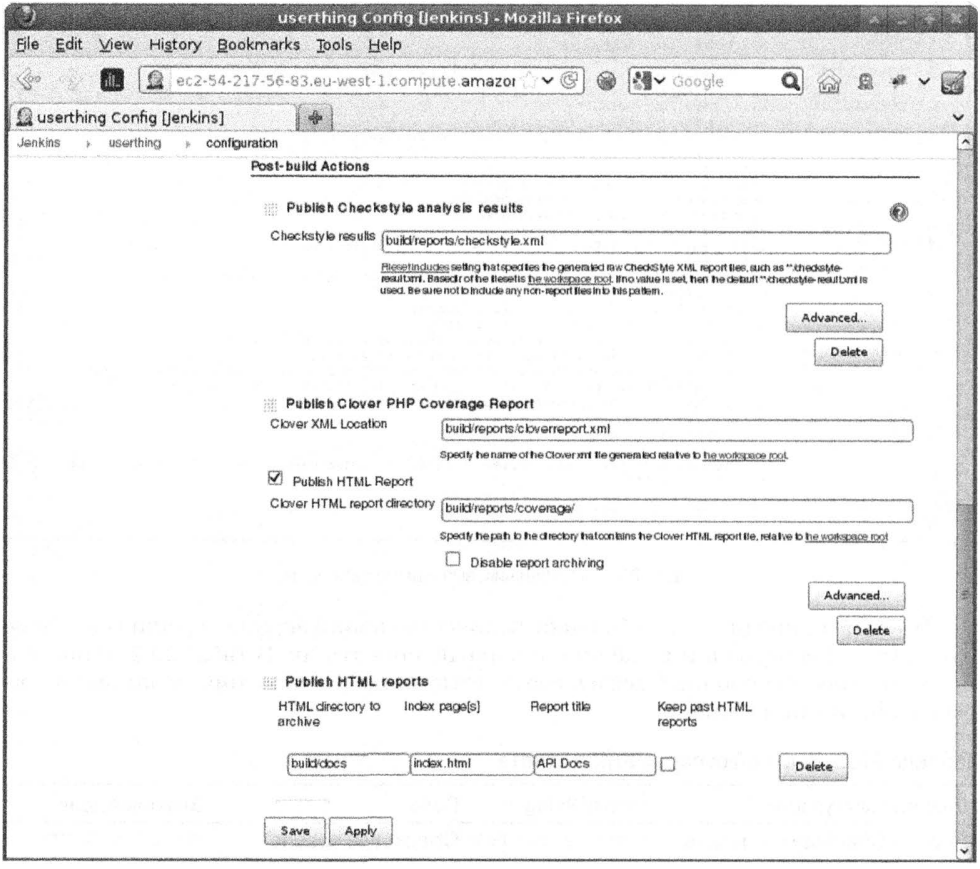


Рис. 20.9. Изменение параметров дополнительного модуля построения отчетов

Все параметры конфигурации, указанные в табл. 20.2, соответствуют тем, которые я указал в файле построения проекта. Все, за исключением последнего. Параметр E-mail Notification позволяет указать список разработчиков, которым будут присылаться уведомления о неудачах, возникших при построении проекта.

Сделав все описанные выше настройки, я могу снова вернуться к экрану управления проектом и снова запустить процесс построения. На рис. 20.10 показано, насколько улучшился вывод информации об этом процессе.

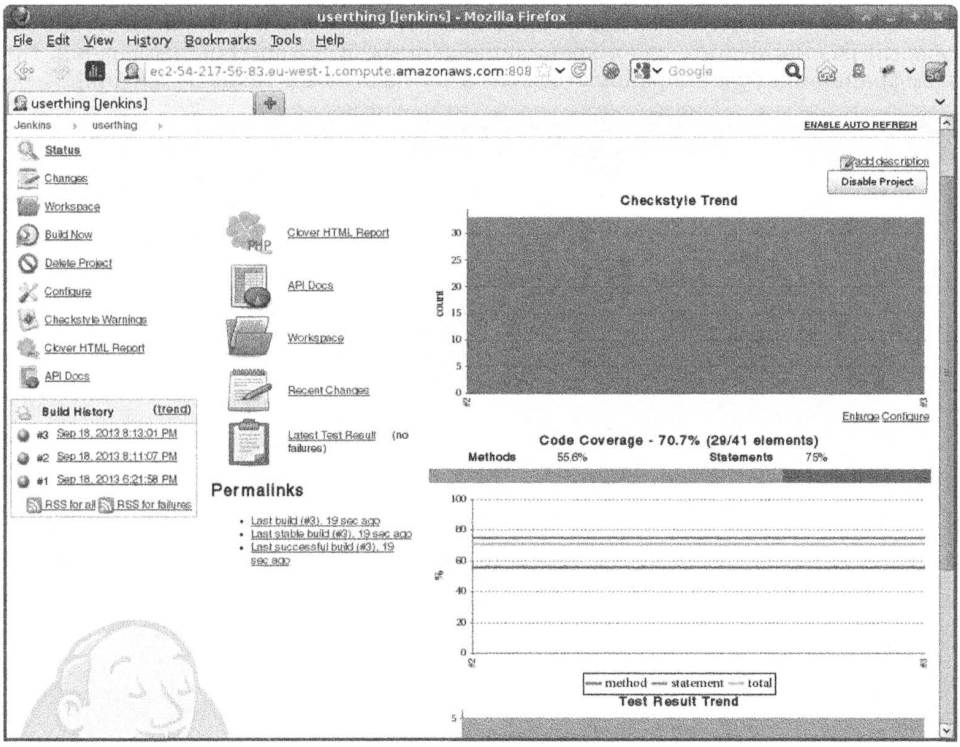


Рис. 20.10. На экране управления проектом информация стала представляться в графическом виде

Через некоторое время на экране управления проектом вы увидите графики, показывающие тенденции, полученные в ходе тестирования на производительность, покрытие кода и анализа стандартов кодирования. Там же будут приведены ссылки на самую свежую документацию по API, подробные результаты тестирования и полная информация по покрытию кода.

Автоматический запуск тестов

Все приведенное выше обилие информации практически не представляет никакой пользы, если обязанность по запуску тестов сборок вручную возложена на одного из забывчивых членов вашей команды разработчиков. Именно по этой причине в сервере Jenkins предусмотрен механизм автоматического запуска построения сборок.

В параметрах сервера вы можете указать, чтобы Jenkins автоматически выполнял запуски через определенные интервалы времени либо отслеживал изменения, внесенные в хранилище системы контроля версий, опять же, через определенные интервалы времени. Интервалы задаются в формате программы `cron` системы Unix. Это позволяет определить желаемое расписание запуска тестов, хотя иногда оно и

будет выглядеть крайне запутанным и непонятным. К счастью, для тех, кому не требуется создавать сложные расписания запуска тестов, в Jenkins предусмотрено несколько простых псевдонимов, включая `@hourly`, `@midnight`, `@daily`, `@weekly` и `@monthly`, которые очень хорошо описаны в справочной системе. На рис. 20.11 я установил, чтобы процесс построения проекта запускался один раз в сутки либо каждый раз после того, как в хранилище данных будут внесены изменения. При этом хранилище будет опрашиваться раз в час на предмет таких изменений.

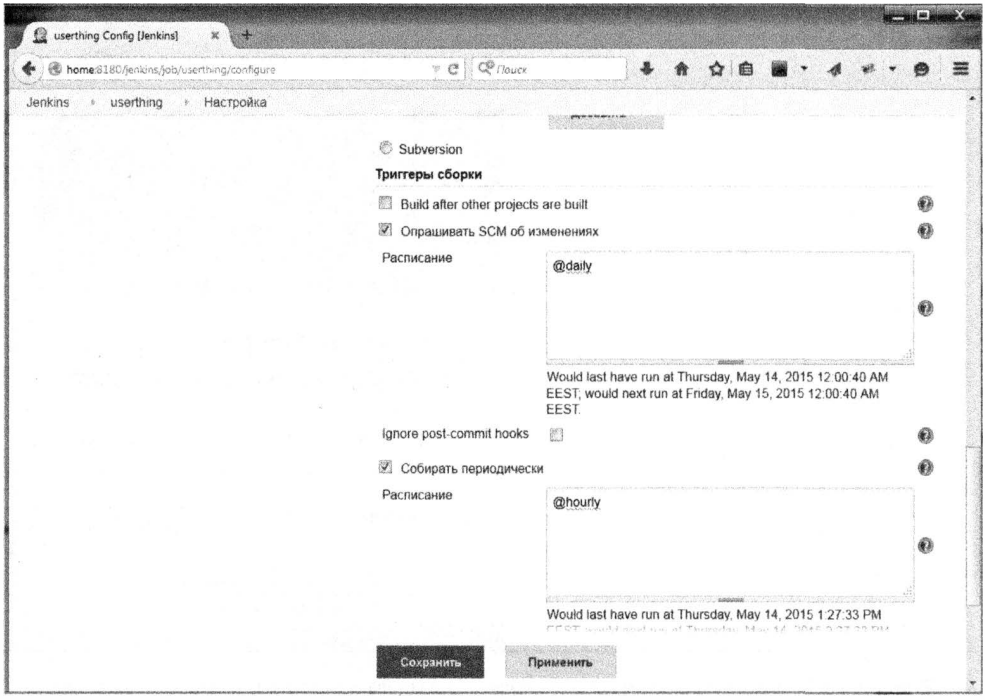


Рис. 20.11. Расписание запуска сборок проекта

Неудачное тестирование

До сих пор у нас все складывалось довольно удачно, даже несмотря на то, что мы не собирались в ближайшее время помещать проект `userthing` в кодовую базу Zend. Но тестирование можно назвать удачным только в том случае, если будут выявлены какие-либо изъяны в коде. Поэтому сейчас мне придется "сломать" что-нибудь и убедиться, что Jenkins сообщит об этом.

Ниже приведен фрагмент класса `Validate` в пространстве имен `userthing\util`.

```
public function validateUser( $mail, $pass ) {
   // Сделаем так, чтобы проверка всегда
   // завершалась неудачно
   return false;

   $user = $this->store->getUser( $mail );
   if ( is_null( $user ) ) {
      return null;
```

```
    }

    if ( $user->getPass() == $pass ) {
        return true;
    }

    $this->store->notifyPasswordFailure( $mail );
    return false;
}
```

Вы обратили внимание, какую диверсию я устроил в методе `validateUser()`? Теперь он всегда будет возвращать значение `false`. Ниже приведен тестовый пример, находящийся в файле `test/ValidatorTest.php`, который должен завершиться неудачей в связи с этим.

```
public function testValidate_CorrectPass() {
    $this->assertTrue(
        $this->validator->validateUser( "bob@example.com", "12345" ),
                                        "Expecting successful validation"
                                       );
}
```

После внесения изменений в код все, что мне нужно сделать, — зафиксировать их в хранилище и ждать! Довольно скоро на странице состояния моего проекта `userthing` появится красный значок, обращающий внимание на то, что что-то пошло не так. Щелкнув на приведенной ниже ссылке, я могу узнать подробности происшедшего. На рис. 20.12 показан отчет об ошибках.

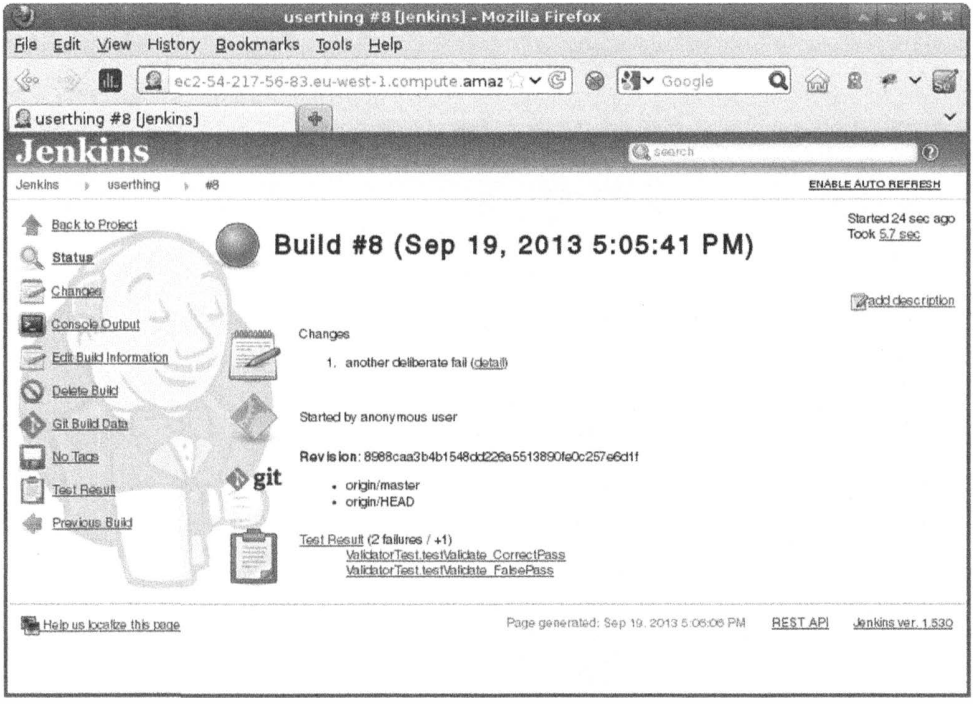


Рис. 20.12. Неудачное построение проекта

Кроме того, мне будет отослано уведомление по электронной почте, содержащее в строке темы надпись "Build failed in Jenkins: userthing #8". В нем будет указана информация, относящаяся к моменту последней фиксации кода в хранилище, процессу построения и ошибках тестирования.

Резюме

В этой главе я применил много инструментов, с которыми вы уже сталкивались в предыдущих главах, и объединил их под эгидой использования в сервере Jenkins. Я подготовил небольшой проект для НИ, применил к нему спектр средств, включая PHPUnit (и для тестирования, и для покрытия кода), PHP_CodeSniffer, PHP_CodeBrowser, phpDocumentor и Git. После этого я выполнил установку сервера Jenkins и показал, как добавляются проекты в систему НИ. Я запустил систему непрерывной интеграции и показал, как можно расширить функциональные возможности сервера Jenkins для рассылки уведомлений об ошибках по электронной почте, и с его помощью выполнил тестирование процессов построения и инсталляции.

Часть V

Заключение

Глава 21

Объекты, шаблоны, практика

Цель этой книги — помочь читателю в создании успешного PHP-проекта. Чтобы достичь ее, мы рассмотрели широкий спектр тем, начиная с основ объектно-ориентированного программирования и принципов проектирования с помощью шаблонов и заканчивая инструментами и методами.

В данной главе я подытожу некоторые основные темы и моменты, которые были рассмотрены в книге.

- *PHP и объекты:* как в PHP продолжает усиливаться поддержка объектно-ориентированного программирования и как использовать эти возможности.
- *Объекты и проектирование:* краткое описание некоторых принципов объектно-ориентированного подхода к проектированию.
- *Шаблоны:* преимущества шаблонов.
- *Принципы шаблонов:* краткий перечень главных принципов объектно-ориентированного проектирования, которые лежат в основе многих шаблонов.
- *Инструменты для работы:* напоминание о средствах, которые я описал в предыдущих главах, а также рассмотрение тех, которые не были описаны.

Объекты

Как упоминалось в главе 2, в течение долгого времени объекты были чем-то вроде дополнения к PHP. Поддержка объектов в PHP 3 была, мягко говоря, незначительной, причем объекты представляли собой всего лишь ассоциативные массивы в маскарадно-причудливом одеянии. Хотя в PHP 4, к радости приверженцев ООП, ситуация существенно изменилась к лучшему, все еще остались серьезные проблемы. Одна из проблем заключалась в том, что объекты присваивались и передавались по значению.

С появлением PHP 5 объекты, наконец-то, вышли из тени. Конечно, вы по-прежнему можете программировать на PHP, даже не объявляя класс, но теперь уже нет сомнений, что этот язык оптимизирован для объектно-ориентированного проектирования.

В главах 3–5 я подробно рассмотрел поддержку объектно-ориентированного программирования в PHP. Вот некоторые новые возможности, которые появились в PHP 5, — рефлексия, исключения, закрытые и защищенные методы и свойства, метод `__toString()`, модификатор `static`, абстрактные классы и методы, завершенные методы и свойства, интерфейсы, итераторы, методы-перехватчики, уточнение типов, мо-

дификатор const, передача по ссылке, методы __clone(), __construct(), позднее статическое связывание и пространства имен. Этот неполный список говорит о той степени, в какой будущее PHP связано с объектно-ориентированным программированием.

Я хотел бы видеть и ряд других возможностей, которые сейчас активно обсуждаются, такие как уточнения для основных типов данных. Я хотел бы также увидеть поддержку уточнений для возвращаемых типов, когда в объявлении метода указывается возвращаемый им тип данных. Тогда для текущих и переопределенных методов можно было бы это контролировать на уровне интерпретатора.

Но это все придирки. Zend Engine 2 и PHP 5 поставили объектно-ориентированное программирование во главу угла любого PHP-проекта, тем самым открывая язык для новых разработчиков и новые возможности для существующих приверженцев.

В главе 6 я рассмотрел преимущества, которые могут дать объекты при разработке проектов. Поскольку объекты и проектирование — это одна из главных тем данной книги, стоит еще раз вспомнить некоторые выводы.

Выбор

Нет такого закона, который говорит о том, что вы должны разрабатывать проект только с помощью классов и объектов. Хорошо спроектированный объектно-ориентированный код обеспечивает понятный интерфейс, доступный из любого клиентского кода, процедурного или объектно-ориентированного. Даже если вам не интересно создавать объекты (что маловероятно, поскольку вы все еще читаете эту книгу), вероятно, вы все равно будете использовать их, хотя бы как клиент пакетов PEAR.

Инкапсуляция и делегирование

Объекты выполняют свою работу "за закрытыми дверями". Они обеспечивают интерфейс, посредством которого передаются запросы и результаты. Данные, которые не должны быть раскрыты, а также детали реализации скрыты за этим интерфейсом.

Это придает различные 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 большая степень вмешательства. Логика управления в большей степени внедряется в реализацию, ссылаясь на переменные, оценивая возвращенные значения и направляясь по различным путям действия в зависимости от обстоятельств.

Разделение

Разделить значит удалить взаимозависимость между компонентами так, чтобы изменение в одном компоненте не влекло за собой необходимость изменения других компонентов. Хорошо спроектированные объекты самодостаточны, т.е. им не нужно ссылаться на что-то внешнее, чтобы "вспомнить" детали, полученные в предыдущем вызове.

Поддерживая внутреннее представление состояния, объекты снижают потребность в глобальных переменных, которые являются общеизвестной причиной возникновения тесной связи. Используя глобальную переменную, вы связываете одну часть системы с другой. Если компонент (функция, класс или блок кода) ссылается на глобальную переменную, 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 работу первого компо-

нента, то третий компонент может перестать работать. Цель объектно-ориентированного проекта — уменьшить такую взаимозависимость, сделав каждый компонент настолько самодостаточным, насколько это возможно.

Еще одна причина тесной связи — дублирование кода. Когда мы повторяем реализацию алгоритма в различных частях проекта, то обнаруживаем тесную связь. А что произойдет, если нужно будет изменить сам алгоритм? Очевидно, вы должны не забыть изменить его код везде, где он встречается. А если забудете это сделать, то в системе начнутся проблемы.

Распространенная причина дублирования кода — параллельные условия. Если проект должен работать одним способом в соответствии с определенной ситуацией (например, при запуске на Linux) и другим способом в альтернативной ситуации (при запуске на Windows), то вы часто будете обнаруживать, что операторы `if/else` возникают в различных частях системы. Если добавить новую ситуацию вместе со стратегией ее обработки (например, работу в системе MacOS), то вы должны обеспечить обновление всех условных операторов.

В объектно-ориентированном программировании предусмотрен метод решения данной проблемы. Вы можете заменить условные операторы *полиморфизмом*. Полиморфизм, также известный как *переключение классов*, — это прозрачное использование различных подклассов в зависимости от ситуации. Поскольку каждый подкласс поддерживает тот же интерфейс, что и общий суперкласс, клиентский код никогда не знает и не хочет знать, какую конкретно реализацию он использует.

Это 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

Инкапсуляция способствует разделению, которое, в свою очередь, способствует повторному использованию. Самодостаточные компоненты, которые осуществляют коммуникации с более широкими системами только через их общедоступные интерфейсы, как правило, можно перемещать из одной системы в другую и использовать без изменений.

На самом деле такие компоненты встречаются реже, чем вы думаете. Даже идеально ортогональный код может быть зависим от проекта. Например, при создании набора классов для управления контентом определенного веб-сайта имеет смысл потратить некоторое время на этапе планирования, чтобы определить возможности, которые характерны для клиента и могут сформировать основу для будущих проектов по управлению контентом.

Еще один совет по повторному использованию: централизуйте классы, которые можно использовать в нескольких проектах. Другими словами, не копируйте повторно используемый класс в новый проект. Это может стать причиной тесной связи на макроуровне, поскольку вы неизбежно измените этот класс в одном проекте и забудете это сделать в другом. Гораздо лучше работать с общими классами в центральном хранилище, которое может совместно использоваться в различных проектах.

Эстетика

Я не собираюсь никого убеждать, но считаю, что объектно-ориентированный код вызывает эстетическое удовлетворение. Беспорядочность реализации скрыта за аккуратными интерфейсами, в результате чего объект кажется клиенту простым.

Я люблю аккуратность и изящество полиморфизма, потому что API позволяет совершать операции над различными объектами, которые, тем не менее, работают взаимозависимо и прозрачно, так что эти объекты можно складывать или вставлять один в другой, как кубики детского конструктора.

Конечно, найдутся люди, которые будут утверждать противоположное. В объектно-ориентированном коде используются запутанные имена классов, которые необходимо объединять с именами методов, чтобы сформировать еще более сложные вызовы. Это особенно верно для PEAR, где в имена классов включены имена пакетов, чтобы компенсировать отсутствие поддержки пространства имен в PHP. Сегодня пространства имен стали частью языка. Но из-за проблем обратной совместимости пройдет некоторое время, прежде чем старые соглашения об именовании перестанут действовать.

Стоит также упомянуть, что красивое решение — не всегда самое лучшее или самое эффективное. Иногда возникает искушение использовать полномасштабное объектно-ориентированное решение там, где вполне достаточно небольшого сценария или нескольких системных вызовов.

Еще одно обоснованное критическое замечание состоит в том, что объектно-ориентированный код может раствориться в путанице классов и объектов, в которых очень трудно разобраться. Согласен, такое бывает, но данную проблему можно в значительной степени решить с помощью подробной документации, содержащей примеры применения.

Шаблоны

Недавно один программист на Java, поступая на работу в компанию, с которой я сотрудничал, извинялся за то, что использовал шаблоны всего пару лет. Почему-то считается, что шаблоны проектных решений (или просто, проектные шаблоны) — это недавнее изобретение, революционное достижение прогресса. Именно этим объясняется тот ажиотаж, который они вызвали. На самом деле этот опытный программист наверняка использовал шаблоны намного дольше, чем он думал.

С помощью шаблонов описывают общие проблемы и проверенные решения, присваивают названия, кодируют и систематизируют лучшие методы решения практических задач. Они не являются элементами изобретения или сухой теорией. Шаблон не будет эффективным, если он не описывает методы, которые уже распространены на момент разработки.

Помните, что концепция 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 проблем пространства и функциональных потребностей.

Учитывая сказанное, легко понять, почему шаблоны вызывают эмоции, связанные с религиозными и политическими дискуссиями. Верующие бредут по коридору 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 некоторые критики считают, что проектные шаблоны — это надувательство.

В таких языках, как Perl и PHP, применение шаблонов также вызывают споры из-за стойкой ассоциации с объектно-ориентированным программированием. При условии, что объекты — это обоснованное проектное решение, а не самоцель, ориентация на проектные шаблоны говорит только о предпочтениях. И это вовсе не означает, что шаблоны должны порождать шаблоны, а объекты — объекты.

Преимущества шаблонов

О шаблонах я рассказал в главе 7. Давайте еще раз вспомним преимущества, которые дают нам шаблоны.

Испытанные и проверенные

Прежде всего, как я уже отмечал, шаблоны — это испытанные решения конкретных проблем. Но проводить аналогию между шаблонами и рецептами опасно: рецептам можно следовать слепо, в то время как шаблоны являются "полуфабрикатами" (Мартин Фаулер) по природе и требуют более вдумчивого обращения. Тем не менее у рецептов и шаблонов есть одна важная общая характеристика: перед описанием они были тщательно проверены и испытаны.

Шаблоны предполагают другие шаблоны

У шаблонов есть вырезы и изгибы, которые соответствуют один другому. Некоторые шаблоны "вставляются один в другой с приятным щелчком". Решение проблемы с помощью шаблона неизбежно будет иметь последствия. Эти последствия могут стать условиями, которые предполагают дополнительные шаблоны. Конечно, при выборе связанных шаблонов важно решать реальные проблемы, а не просто нагромождать изящные, но бесполезные башни взаимосвязанного кода. Очень заманчиво создать программный эквивалент архитектурной причуды.

Общий словарь

Шаблоны — это средство разработки общего словаря для описания проблем и решений. Имена очень важны — они заменяют описания и позволяют очень быстро осветить множество тем. Но, конечно, имена также делают неясным смысл для тех, кто еще не разобрался в словаре, и это одна из причин, по которым шаблоны иногда очень сильно раздражают.

Шаблоны способствуют проектированию

Как уже упоминалось, при правильном использовании шаблоны способствуют хорошему проектированию. Но здесь есть важное замечание: шаблоны — это не волшебная палочка.

Шаблоны и принципы проектирования

Проектные шаблоны по своей природе связаны с хорошим проектом. При правильном использовании они помогут создать слабосвязанный и гибкий код. Но критики шаблонов в чем-то правы, когда говорят, что шаблонами часто злоупотребляют, особенно новички. Поскольку реализации шаблонов формируют красивые и изящные структуры, легко забыть, что хороший проект всегда соответствует цели. Помните, что шаблоны существуют для решения проблем.

Впервые работая с шаблонами, я обнаружил, что использовал Abstract Factories по всему коду. Мне нужно было генерировать объекты, и шаблон Abstract Factory помогал мне в этом.

Но на самом деле мне было лень думать, и я делал ненужную работу. Наборы объектов, которые мне нужно было создавать, действительно были связаны, но у них не было альтернативных реализаций. Классический шаблон Abstract Factory идеален для ситуации, когда вам нужно создавать альтернативные наборы объектов в зави-

симости от ситуации. Чтобы шаблон Abstract Factory работал, нужно создать классы Abstract Factory для каждого типа объекта и класса, с которым будет работать класс Factory. Даже описывать этот процесс утомительно.

Код был бы намного аккуратнее, если бы я создал базовый класс Factory, осуществив только рефакторинг для реализации Abstract Factory, если бы мне понадобилось сгенерировать параллельный набор объектов.

Использование шаблонов — это еще не гарантия хорошего проекта. При разработке стоит помнить о двух формулировках одного и того же принципа: KISS ("Keep it simple, stupid" — "Делай просто и примитивно") и "Do the simplest thing that works" ("Ищите самое простое решение, которое работает"). В экстремальном программировании также используется похожее правило YAGNI: "You aren't going to need it" — "Вам это не понадобится". Это означает: не нужно реализовать возможность, если в ней нет крайней необходимости.

Сделав такие необычные замечания, я вернусь к более восторженному тону. Как я утверждал в главе 9, в шаблонах обычно воплощается ряд принципов, которые можно обобщить и применить ко всему коду.

Предпочитайте композицию наследованию

Отношения наследования очень эффективны. Наследование используется для поддержки переключения классов во время выполнения программы (полиморфизм), что лежит в основе многих шаблонов и методов, рассмотренных в данной книге. Но, полагаясь в проекте исключительно на наследование, можно создать негибкие структуры, подверженные дублированию.

Избегайте тесной связи

Я уже обсуждал это в данной главе, но повторю еще раз ради полноты изложения. Вам придется считаться с тем фактом, что изменение в одном компоненте может потребовать изменений в других частях проекта. Но это можно свести к минимуму, избегая дублирования (типичной причиной которого являются параллельные условные операторы) и злоупотребления глобальными переменными (или шаблонами Singleton). Можно также свести к минимуму использование конкретных подклассов, где абстрактные типы используются для поддержки полиморфизма. Последнее высказывание приводит нас к следующему принципу.

Кодируйте на основе интерфейса, а не его реализации

Проектируйте программные компоненты с четко определенными общедоступными интерфейсами, которые берут на себя обязанности каждого прозрачного компонента. Если вы определяете интерфейс в абстрактном суперклассе и у вас есть клиентские классы, которым нужно работать с этим абстрактным типом, то тем самым вы отделяете клиентов от конкретных реализаций.

Учитывая сказанное, вспомните о принципе YAGNI. Если сначала вам нужна только одна реализация для типа, 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 делать больше одной операции одновременно, то сможете переопределить конкретный класс в качестве абстрактного родителя двух подклассов. Клиентский код будет ничуть не "умнее", поскольку он продолжает работать с одним типом.

Классический признак того, что вам нужно разделить реализацию и спрятать получающиеся в результате классы за абстрактным родительским классом, — это появление условных операторов в реализации.

Инкапсулируйте то, что меняется

Если вы обнаружите, что "утонули" в подклассах, то, возможно, следует выяснить причину и создать для всех этих подклассов собственный тип. Особенно если причина состоит в том, чтобы достичь цели, которая связана с основной целью типа.

Возьмем, к примеру, тип `UpdatableThing`. Вы можете создать подтипы `FtpUpdatableThing`, `HttpUpdatableThing` и `FileSystemUpdatableThing`. Но обязанность этого типа должна быть *обновляемой* — для этой цели предназначены механизмы сохранения и извлечения. `Ftp`, `Http` и `FileSystem` — это вещи, которые здесь меняются, и они принадлежат собственному типу; давайте назовем его `UpdateMechanism`. У `UpdateMechanism` будут подклассы для различных реализаций. Затем можно будет добавить столько механизмов обновления, сколько нужно, не трогая тип `UpdatableThing`, который сконцентрирован на выполнении своей основной обязанности.

Также обратите внимание на то, что я заменил статическую структуру, создаваемую на этапе компиляции, динамической, создаваемой во время выполнения программы. Она словно случайно возвращает нас к основному принципу: "Предпочитайте композицию наследованию".

Практика

Вопросы, которые я осветил в данной главе (и представил в главе 14), часто игнорируют и авторы книг, и программисты. Но как программист я обнаружил, что эти инструменты и методы по меньшей мере так же необходимы для достижения успеха, как и хорошее проектирование. Но, конечно, документирование и автоматическое построение не настолько поражают воображение, как шаблон Composite.

На заметку. Давайте вспомним о красоте шаблона Composite: простое дерево наследования, объекты которого можно объединить во время выполнения, чтобы сформировать структуры, которые тоже являются деревьями, но на несколько порядков более гибкими и сложными. Несколько объектов совместно используют один интерфейс, который является их представлением во внешнем мире. Взаимодействие между простым и сложным, единичным и множественным заставит учащенно биться ваше сердце — это не просто проектирование программного обеспечения, это поэзия!

Но даже если такие понятия, как документация и построение, тестирование и контроль версий, более прозаичны, чем шаблоны, они не менее важны. На практике даже самый фантастический проект не будет долговечным, если несколько разработчиков не смогут легко добавлять к нему код или не поймут исходный код. Системы трудно поддерживать и расширять без автоматического тестирования. Без инструментов построения никто не будет утруждать себя, чтобы развернуть вашу работу. Поскольку количество PHP-проектов все время расширяется, мы как разработчики обязаны обеспечить качество и простоту их развертывания.

Проект существует в двух ипостасях: это структуры кода и функциональности и это набор файлов и каталогов, основа для сотрудничества, набор источников и целей, предмет для преобразования. В этом смысле проект — это система снаружи в той же степени, что и внутри кода. Механизмы построения, тестирования, документирования и контроля версий требуют такого же внимания к деталям, поскольку в коде поддерживаются эти механизмы. Уделяйте такое же пристальное внимание метасистеме, как и самой системе.

Тестирование

Хотя тестирование — это часть схемы, применяемой к проекту извне, оно глубоко интегрировано в сам код. Поскольку полное отделение невозможно и даже нежелательно, тестовые структуры — это эффективное средство 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 инструмент улучшения объектно-ориентированного проекта. Если сначала (или хотя бы одновременно) вы будете проводить тестирование, это поможет сосредоточиться на интерфейсе класса и тщательно обдумать обязанности и поведение каждого метода. О пакете PHPUnit, который используется для тестирования, я рассказал в главе 18.

Документация

Ваш код не настолько понятен, как вы думаете. Тот, кто впервые его увидит, может прийти в ужас, поскольку ничего не поймет. Даже вы, автор кода, в конце концов забудете, как все это было сделано. В главе 16 я описал phpDocumentor, который позволяет документировать код по ходу разработки и автоматически генерирует текст документации с гиперссылками.

Результат, который выдает phpDocumentor, особенно полезен в объектно-ориентированном контексте, поскольку позволяет пользователю с помощью щелчков мышью переходить от одного класса к другому. Поскольку классы, как правило, содержатся в собственных файлах, непосредственный анализ исходного кода может включать переход по сложным траекториям от одного исходного файла к другому.

Контроль версий

Сотрудничество — непростое дело. Давайте посмотрим правде в глаза: люди неуклюжи, а программисты — тем более. Распределив роли и задачи внутри команды, самое последнее, чем вы захотите заниматься, — это разбираться в несоответствиях в самом исходном коде. Как мы видели в главе 17, Git (и аналогичные инструменты, такие как CVS и Subversion) позволяет объединять работу нескольких программистов в одном хранилище. В тех случаях, когда конфликты (несоответствия) неизбежны, Git сообщает об этом и указывает, в каком месте исходного кода нужно решить проблему.

Даже если вы программист, работающий в одиночку, система контроля версий необходима. Система Git поддерживает ветвление, так что можно поддерживать одну ветвь версии программы и одновременно разрабатывать следующую, перенося код с исправленными ошибками из стабильной версии в ветку разработки.

Git также регистрирует все фиксации кода, которые когда-либо были сделаны в проекте. Это означает, что вы можете вернуться к более старой версии, найдя нужный вариант программы по дате или метке. Поверьте мне, 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 усилия. Различные файлы нужно переместить в разные каталоги системы, файлы конфигурации — преобразовать, чтобы они содержали нужные параметры для текущей

платформы и базы данных, таблицы базы данных нужно создать или преобразовать. Я описал два инструмента для инсталляции. Первый, PEAR (см. главу 15), идеален для автономных пакетов или небольших приложений. Второй описанный мной инструмент построения — Phing (см. главу 19) — обладает достаточной эффективностью и гибкостью для автоматизации инсталляции самых крупных и запутанных проектов.

Автоматическое построение превращает процесс развертывания из утомительной работы в выдачу одной-двух команд из командной строки. Вы легко и непринужденно вызываете тестовую программу и вывод документации из инструмента 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 недостаточно. Вы должны это делать *непрерывно*! Причем важность этого возрастает по мере расширения и усложнения проекта и поддержки в нем нескольких веток разработки. Вам придется постоянно выполнять построение и тестирование стабильной ветки, в которую были внесены небольшие исправления замеченных ошибок. Кроме нее, могут существовать несколько экспериментальных веток разработки, а также ваша основная ветка. Если вы будете пытаться все делать вручную, даже с помощью соответствующих инструментов построения и тестирования, 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 20 я рассмотрел непрерывную интеграцию (НИ) — набор методик и средств, предназначенных для максимально возможной автоматизации процесса построения и тестирования проекта.

Что я упустил

Несколько инструментов, описание которых я опустил в данной книге по причине ограничений времени и места, тем не менее, чрезвычайно полезны для любого проекта.

Наверное, главный из них — Bugzilla. Его название говорит о следующем. Во-первых, этот инструмент предназначен для регистрации ошибок. Во-вторых, это часть проекта Mozilla.

Как и Git, Bugzilla — один из тех эффективных инструментов, испытав которые однажды, уже нельзя представить свою работу без них. Bugzilla можно загрузить по адресу `http://www.bugzilla.org`.

Bugzilla позволяет пользователям сообщать о проблемах в проекте, но по своему опыту могу сказать, что он часто используется и как средство описания необходимых возможностей и распределения задач по 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 ошибки есть собственная страница, на которой можно обсудить текущие проблемы. Записи в обсуждении и изменения статуса ошибки можно скопировать и отослать по почте членам команды, что позволяет следить за ходом событий, не заходя постоянно на веб-сайт Bugzilla.

Bugzilla вам очень нужен, поверьте мне.

В каждом серьезном проекте нужно иметь хотя бы одну рассылку, чтобы пользователи могли получать информацию об изменениях и вопросах использования, а разработчики — обсуждать архитектуру и распределение ресурсов. Моя любимая программа рассылки — это Mailman (`http://www.gnu.org/software/mailman/`). Она бесплатная, очень гибкая в конфигурировании, и ее достаточно просто инсталлировать. Но если вы не хотите устанавливать собственную программу рассылки, существует множество сайтов, где можно бесплатно создавать собственные списки рассылки или вести группы новостей.

Хотя внутренняя документация очень важна, к проектам также прилагается куча печатных материалов. В них включаются инструкции по применению, советы по будущим направлениям развития, активы клиентов, протоколы собраний и уведомления о корпоративных вечеринках. В течение жизни проекта такие материалы постоянно меняются, и нужен механизм, позволяющий членам команды разрабатывать их совместно.

Технология "вики" (в переводе с гавайского wiki означает "очень быстро") — идеальный инструмент для создания совместно используемых систем документов с гиперссылками. Создавать и редактировать страницы можно с помощью щелчка мышью, а гиперссылки автоматически генерируются для слов, соответствующих именам страниц. Вики — это еще один такой простой, важный и очевидный инструмент, что сразу кажется, будто у вас была такая идея, но вы просто не взялись за ее осуществление. Существует ряд вики на выбор. У меня был позитивный опыт работы с Foswiki, которую можно загрузить по адресу `http://foswiki.org`. Foswiki написана на языке Perl. Конечно, существуют также вики-приложения, написанные на PHP. Среди них достойны внимания приложение PhpWiki, которое можно загрузить по адресу `http://phpwiki.sourceforge.net`, и приложение DokuWiki, которое можно найти на сайте `http://wiki.splitbrain.org/wiki:dokuwiki`.

Резюме

В этой главе я подвел итоги, повторив основные темы, которые освещались в книге. Я не затрагивал здесь конкретные вопросы, касающиеся отдельных шаблонов или функций объектов, потому что цель данной главы — обобщить материал книги.

Для того чтобы охватить все вопросы, никогда не хватает ни места, ни времени. Но я надеюсь, что данная книга сыграет свою роль в доказательстве того, что PHP развивается. В настоящее время это один из самых популярных языков программирования в мире. Надеюсь, PHP останется любимым языком непрофессиональных программистов, и многие начинающие PHP-программисты с радостью обнаружат, насколько многого можно достичь с помощью совсем небольшого кода. В то же время следует отметить, что все больше и больше профессиональных команд создают крупные и сложные системы с помощью PHP. Такие проекты заслуживают более серьезного подхода. Благодаря уровню расширения PHP всегда был гибким и разносторонним языком, предоставляя доступ для сотен приложений и библиотек. С другой стороны, его поддержка объектно-ориентированного программирования предоставляет доступ к различным наборам инструментов. Как только вы начнете мыслить в категориях объектов, то сможете воспользоваться опытом других программистов, добытым ценой больших усилий. Вы сможете находить и внедрять шаблоны, разработанные не только для PHP, но и для Smalltalk, C++, C# или Java. Наша обязанность — решать сложные проблемы, применяя тщательное проектирование и эффективные методы. Будущее за многократно используемым кодом!

Часть VI

Приложения

Приложение А

Дополнительные источники информации

Книги

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.

   (Кристофер Александер, Сара Исикава, Мюррей Силверстайн. *Язык шаблонов. Города. Здания. Строительство.* — Пер. с англ. — Изд. 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.

   (Дипак Алур, Джон Крупи, Дэн Малкс. *Образцы J2EE. Лучшие решения и стратегии проектирования.* — Пер. с англ. — Изд. "Лори", 2013.)

3. Beck Kent. *Extreme Programming Explained: Embrace Change.* Reading, MA: Addison-Wesley, 1999.

   (Кент Бек. *Экстремальное программирование.* — Пер. с англ. — Изд. "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, 1999.

   (Мартин Фаулер. *UML. Основы. Краткое руководство по стандартному языку объектного моделирования.* — Пер. с англ. — Изд. "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, 1999.

   (Мартин Фаулер, Кент Бек, Джон Брант, Уильям Апдайк, Дон Робертс, Эрих Гамма. *Рефакторинг. Улучшение существующего кода.* — Пер. с англ. — Изд. "Символ-Плюс", 2013.)

8. Fowler Martin. *Patterns of Enterprise Application Architecture*. Reading, MA: Addison-Wesley, 2003.

   (Мартин Фаулер. 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, 1995.

   (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. — Пер. с англ. — Изд. "Питер", 2007.)

10. Hunt Andrew and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA: Addison-Wesley, 2000.

    (Эндрю Хант, Дэвид Томас. *Программист-прагматик. Путь от подмастерья к мастеру*. — Пер. с англ. — Изд. "Лори", 2012.)

11. Kerievsky Joshua. *Refactoring to Patterns*. Reading, MA: Addison-Wesley, 2004.

    (Кериевски Джошуа. *Рефакторинг с использованием шаблонов*. — Пер. с англ. — И.Д. "Вильямс", 2006.)

12. Metsker, Steven John. *Building Parsers with Java*. Reading, MA: Addison-Wesley, 2001.

13. Nock Clifton. *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Reading, MA: Addison-Wesley, 2004.

14. Shalloway Alan and James R Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Reading, MA: Addison Wesley, 2002.

15. Stelting Stephen and Olav Maasen. *Applied Java Patterns*. Palo Alto, CA: Sun Microsystems Press, 2002.

Статьи в Интернете

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. Gutmans Andi. *Zend's Andi Gutmans on PHP 6 & how Apple is the 'biggest barrier' to mobile's future.* `http://venturebeat.com/2012/10/24/zends-andi-gutmans-on-php-6-being-a-developer-ceo-and-how-apple-is-the-biggest-barrier-to-the-future-of-mobile/`

4. Lerdorf Rasmus. *"PHP/FI Brief History."* `http://www.php.net//manual/phpfi2.php#history`

5. Suraski Zeev. *"The Object-Oriented Evolution of PHP."* `http://www.devx.com/webdev/Article/10007/0/page/1`

Сайты

1. Bugzilla: `http://www.bugzilla.org`
2. CruiseControl: `http://cruisecontrol.sourceforge.net/`

3. CVS: http://www.cvshome.org/
4. CvsGui: http://www.wincvs.org/
5. CVSNT: http://www.cvsnt.org/wiki
6. DokuWiki: http://wiki.splitbrain.org/wiki:dokuwiki
7. Foswiki: http://foswiki.org/
8. Eclipse: http://www.eclipse.org/
9. Java: http://www.java.com
10. Jenkins: http://jenkins-ci.org/
11. GNU: http://www.gnu.org/
12. Git: http://git-scm.com/
13. Книга по Git: http://git-scm.com/book
14. Google Code: http://code.google.com
15. Mailman: http://www.gnu.org/software/mailman/
16. Мартин Фаулер: http://www.martinfowler.com/
17. Memcached: http://danga.com/memcached/
18. Mercurial: http://mercurial.selenic.com
19. OpenPear: http://openpear.org/
20. Phing: http://phing.info/trac/
21. PHPUnit: http://www.phpunit.de
22. PhpWiki: http://phpwiki.sourceforge.net
23. PEAR: http://pear.php.net
24. Pyrus: http://pear2.php.net
25. PECL: http://pecl.php.net/
26. Phing: http://phing.info/
27. PHP: http://www.php.net
28. PhpWiki: http://phpwiki.sourceforge.net
29. PHPDocumentor: http://www.phpdoc.org/
30. Pirum: http://pirum.sensiolabs.org
31. Portland Pattern Repository's Wiki (Ward Cunningham): http://www.c2.com/cgi/wiki
32. Pyrus: http://pear2.php.net
33. RapidSVN: http://rapidsvn.tigris.org/
34. QDB: http://www.bash.org
35. Selenium: http://seleniumhq.org/
36. SPL: http://www.php.net/spl
37. Subversion: http://subversion.apache.org/
38. Ximbiot — CVS Wiki: http://ximbiot.com/cvs/wiki/
39. Xdebug: http://xdebug.org/
40. Zend: http://www.zend.com
41. Xinc: http://code.google.com/p/xinc/

Приложение Б

Простой синтаксический анализатор

Шаблон Interpreter, рассмотренный в главе 11, не описывает сам процесс синтаксического анализа. Интерпретатор без анализатора достаточно бесполезен, если, конечно, вы не заставите своих пользователей для вызова интерпретатора писать код на PHP! Существует несколько хороших синтаксических анализаторов сторонних разработчиков, которые можно использовать совместно с шаблоном Interpreter. И это, пожалуй, будет самое лучшее решение в реальных проектах. Тем не менее в этом приложении мы представим простой объектно-ориентированный синтаксический анализатор, предназначенный для встраивания в интерпретатор языка MarkLogic, рассмотренного в главе 11. Учтите, что приведенные ниже примеры предназначены только для демонстрации концепции и без доработки их нельзя использовать в реальных проектах.

На заметку. Интерфейс и основные структуры этого анализатора взяты из книги Стивена Мецкера (Steven Metsker) *Building Parsers with Java* (Addison-Wesley, 2001). Поскольку я его бессовестно упростил, все ошибки следует направлять в мой адрес. Стивен любезно разрешил мне использовать свой оригинальный код, как мне заблагорассудится.

Сканер

Перед началом синтаксического анализа любой конструкции языка мы должны разобрать ее по словам и служебным символам, которые называются *токенами* (tokens). Для определения токенов в приведенном ниже классе используется ряд регулярных выражений. Полученный результат представляется в виде удобного стека, который мы будем использовать в последующих разделах. Вот определение класса `Scanner`.

```
namespace gi\parse;

class Scanner {

   // Типы токенов.
   const WORD            = 1;
   const QUOTE           = 2;
   const APOS            = 3;
```

```
const WHITESPACE      = 6;
const EOL             = 8;
const CHAR            = 9;
const EOF             = 0;
const SOF             = -1;

protected $line_no    = 1;
protected $char_no    = 0;
protected $token      = null;
protected $token_type = -1;

// Доступ к исходным данным осуществляется через
// класс Reader. Результирующие данные сохраняются
// в предоставленном контексте.
function __construct(Reader $r, Context $context ) {
   $this->r = $r;
   $this->context = $context;
}

function getContext() {
   return $this->context;
}

// Пропускает все пробельные символы.
function eatWhiteSpace( ) {
   $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 токена.
// Возвращается либо текущий токен, либо тот,
// который указан в аргументе $int.
function getTypeString( $int=-1 ) {
   if ( $int<0 ) { $int=$this->tokenType(); }
   if ( $int<0 ) { return null; }
   $resolve = array(
                    self::WORD       => 'WORD'      ,
                    self::QUOTE      => 'QUOTE'     ,
                    self::APOS       => 'APOS'      ,
                    self::WHITESPACE => 'WHITESPACE',
                    self::EOL        => 'EOL'       ,
                    self::CHAR       => 'CHAR'      ,
                    self::EOF        => 'EOF'
                   );
   return $resolve[$int];
}
```

```
// Возвращает текущий тип токена, представленный
// целым числом.
function tokenType() {
   return $this->token_type;
}

// Возвращает содержимое текущего токена.
function token() {
   return $this->token;
}

// Возвращает истинное значение, если текущий токен
// имеет тип WORD.
function isWord( ) {
   return ( $this->token_type == self::WORD );
}

// Возвращает истинное значение, если текущий токен
// является кавычками.
function isQuote( ) {
   return ( $this->token_type == self::APOS ||
            $this->token_type == self::QUOTE );
}

// Возвращает номер текущей строки в исходном файле.
function line_no() {
   return $this->line_no;
}

// Возвращает номер текущего символа в исходном файле.
function char_no() {
   return $this->char_no;
}

// Клонирует этот объект
function __clone() {
   $this->r = clone($this->r);
}

// Перемещается к следующему токену в исходном файле.
// Устанавливает текущий токен и отслеживает номер строки
// и номер символа.
function nextToken() {
   $this->token = null;
   $type;
   while ( ! is_bool($char=$this->getChar()) ) {
      if ( $this->isEolChar( $char ) ) {
           $this->token = $this->manageEolChars( $char );
           $this->line_no++;
           $this->char_no = 0;
           $type = self::EOL;
           return ( $this->token_type = self::EOL );

      } else if ( $this->isWordChar( $char ) ) {
```

```
            $this->token = $this->eatWordChars( $char );
            $type = self::WORD;

        } else if ( $this->isSpaceChar( $char ) ) {
                    $this->token = $char;
                    $type = self::WHITESPACE;

        } else if ( $char == "'" ) {
                    $this->token = $char;
                    $type = self::APOS;

        } else if ( $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 токена и содержимое токена
// для следующего токена
function peekToken() {
    $state = $this->getState();
    $type  = $this->nextToken();
    $token = $this->token();
    $this->setState( $state );
    return array( $type, $token );
}

// Получает объект ScannerState, содержащий текущую позицию
// анализатора в исходной строке и данные около текущего токена
function getState() {
    $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
function setState(ScannerState $state ) {
    $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;
    return;
}

// Возвращает следующий символ из исходного файла
private function getChar() {
    return $this->r->getChar();
}

// Возвратить все символы до конца слова
private function eatWordChars( $char ) {
    $val = $char;
    while ( $this->isWordChar( $char=$this->getChar() )) {
        $val .= $char;
    }
    if ( $char ) {
        $this->pushBackChar( );
    }
    return $val;
}

// Возвратить все пробельные символы
private function eatSpaceChars( $char ) {
   $val = $char;
    while ( $this->isSpaceChar( $char=$this->getChar() )) {
        $val .= $char;
    }
    $this->pushBackChar( );
    return $val;
}

// Отодвинуться на один символ в исходном файле
function pushBackChar( ) {
    $this->r->pushBackChar();
    return;
}

// Проверяет, не является ли аргумент символом слова
private function isWordChar( $char ) {
    return preg_match( "/[A-Za-z0-9_\-]/", $char );
}

// Проверяет, не является ли аргумент пробельным символом
private function isSpaceChar( $char ) {
    return preg_match( "/\t| /", $char );
}

// Проверяет, не является ли аргумент символом конца строки
private function isEolChar( $char ) {
    return preg_match( "/\n|\r/", $char );
}
```

```
   // Обрабатывает конец строки: \n, \r или \r\n
   private function manageEolChars( $char ) {
      if ( $char == "\r" ) {
         $next_char=$this->getChar();
         if ( $next_char == "\n" ) {
            return "{$char}{$next_char}";
         } else {
            $this->pushBackChar();
         }
      }
      return $char;
}

   function getPos() {
      return $this->r->getPos();
    }
}

class ScannerState {
   public $line_no;
   public $char_no;
   public $token;
   public $token_type;
   public $r;
}
```

Вначале мы определяем типы всех токенов с помощью констант. Мы собираемся работать с текстовыми символами, словами, пробелами и кавычками. Для тестирования символов созданы специальные методы для каждого типа токенов: `isWord Char()`, `isSpaceChar()` и т.п. Ядром класса является метод `nextToken()`. В нем из текущей строки выделяется следующий токен и определяется его тип. Вся информация сохраняется объектом `Scanner` в контекстном объекте, который используется объектами синтаксического анализатора для обмена данными по мере обработки исходного текста.

Обратите внимание на второй класс — `ScannerState`. Сканер спроектирован так, что объекты `Parser` могут сохранять свое состояние, изменять его и восстанавливать, если что-то пошло не так. Метод `getState()` возвращает объект типа `ScannerState`, заполненный данными. Метод `setState()` использует предоставленный объект типа `ScannerState` для восстановления состояния, если это необходимо.

Вот определение класса `Context`.

```
namespace gi\parse;

class Context {
   public $resultstack = array();

   function pushResult( $mixed ) {
      array_push( $this->resultstack, $mixed );
   }

   function popResult( ) {
      return array_pop( $this->resultstack );
   }
```

```
    function resultCount() {
        return count( $this->resultstack );
    }

    function peekResult( ) {
        if ( empty( $this->resultstack ) ) {
            throw new Exception( "empty resultstack" );
        }
        return $this->resultstack[count( $this->resultstack ) -1 ];
    }
}
```

Как видите, это обычная реализация стека, такая себе доска объявлений, с которой удобно работать классам синтаксического анализатора. Этот класс выполняет ту же работу, что и класс контекста в шаблоне Interpreter, но это не один и тот же класс!

Обратите внимание: класс `Scanner` не выполняет самостоятельно работу с файлами и строками. Для этого ему нужен объект `Reader`. Это позволит нам при необходимости легко изменить источник обрабатываемых данных. Ниже приведены определение интерфейса `Reader` и реализация объекта `StringReader`.

```
namespace gi\parse;

interface Reader {

    function getChar();
    function getPos();
    function pushBackChar();
}

class StringReader implements Reader {
    private $in;
    private $pos;

    function __construct( $in ) {
    $this->in = $in;
    $this->pos = 0;
    }

    function getChar() {
        if ( $this->pos >= strlen( $this->in ) ) {
            return false;
        }
        $char = substr( $this->in, $this->pos, 1 );
        $this->pos++;
        return $char;
    }

    function getPos() {
        return $this->pos;
    }

    function pushBackChar() {
        $this->pos--;
```

```
    }

    function string() {
        return $this->in;
    }
}
```

Этот класс просто считывает из предоставленной строки по одному символу за раз. Конечно, если нужно, мы всегда сможем реализовать версию класса, считывающего символы из файла. Но чтобы увидеть, как работает класс `Scanner`, вполне достаточно и нашей простой реализации. Ниже представлен фрагмент кода, разбивающий нашу языковую конструкцию на токены.

```
$context = new \gi\parse\Context();
$user_in = "\$input equals '4' or \$input equals 'four'";
$reader = new \gi\parse\StringReader( $user_in );
$scanner = new \gi\parse\Scanner( $reader, $context );

while ( $scanner->nextToken() != \gi\parse\Scanner::EOF ) {
    print $scanner->token();
    print "\t{$scanner->char_no()}";
    print "\t{$scanner->getTypeString()}\n";
}
```

Мы инициализируем объект `Scanner` и затем в цикле извлекаем все токены из исходной строки с помощью повторяющегося вызова метода `nextToken()`. Метод `token()` возвращает текущую порцию данных из исходной строки, соответствующей токену. Метод `char_no()` позволяет определить нашу позицию в исходной строке. Метод `getTypeString()` возвращает строковую версию типа текущего токена, определенную нами выше в виде констант. Пример вывода показан ниже.

```
$       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
```

Конечно, при желании мы можем придумать более изощренные токены, чем эти, но их вполне достаточно для демонстрации возможностей программы. Синтаксический анализ строки не представляет трудностей. Так как же нам теперь в коде построить грамматику?

Объект Parser

Один из подходов — создать древовидную структуру объектов типа Parser. Вот как выглядит абстрактный класс Parser, который мы будем использовать.

```
namespace gi\parse;

abstract class Parser {

   const GIP_RESPECTSPACE  = 1;
   protected $respectSpace = false;
   protected static $debug = false;
   protected $discard      = false;
   protected $name;
   private static $count=0;

   function __construct( $name=null, $options=null ) {
      if ( is_null( $name ) ) {
         self::$count++;
         $this->name = get_class( $this ) . " (" . self::$count.")";
      } else {
         $this->name = $name;
      }

      if ( is_array( $options ) ) {
         if ( isset( $options[self::GIP_RESPECTSPACE] ) ) {
            $this->respectSpace=true;
         }
      }
   }

   protected function next(Scanner $scanner ) {
      $scanner->nextToken();
      if ( ! $this->respectSpace ) {
         $scanner->eatWhiteSpace();
      }
   }

   function spaceSignificant( $bool ) {
      $this->respectSpace = $bool;
   }

   static function setDebug( $bool ) {
      self::$debug = $bool;
   }

   function setHandler(Handler $handler ) {
      $this->handler = $handler;
   }

   final function scan(Scanner $scanner ) {
      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 returning $ret");
        return $ret;
    }

    function discard() {
        $this->discard = true;
    }

    abstract function trigger(Scanner $scanner );

    function term() {
        return true;
    }

    // Закрытые и защищенные функции

    protected function invokeHandler(Scanner $scanner ) {
        if ( ! empty( $this->handler ) ) {
            $this->report( "calling handler: "
                           . get_class( $this->handler ) );
            $this->handler->handleMatch( $this, $scanner );
        }
    }

    protected function report( $msg ) {
        if ( self::$debug ) {
            print "<{$this->name}> " . get_class( $this ) . ": $msg\n";
        }
    }

    protected function push(Scanner $scanner ) {
        $context = $scanner->getContext();
        $context->pushResult( $scanner->token() );
    }

    abstract protected function doScan( Scanner $scan );
}
```

Основным в этом классе является метод `scan()`. В нем расположена основная логика нашего анализатора. Методу `scan()` передается объект типа `Scanner`, с которым он работает. И первое, что делает класс `Parser`, — считается с мнением конкретного дочернего класса, для чего вызывается абстрактный метод `doScan()`. Этот

метод возвращает либо истинное, либо ложное значение. Конкретный пример реализации будет рассмотрен ниже.

Если метод `doScan()` возвращает истинное значение и соблюдается ряд других условий, то результат синтаксического анализа помещается в стек объекта `Context`. Ссылка на объект типа `Context`, используемый в объекте `Parser`, хранится также в классе `Scanner`, в результате чего становится возможен обмен результатами между классами. Реальные результаты синтаксического анализа помещаются в стек в методе `Parser::push()`.

```
protected function push(Scanner $scanner ) {
   $context = $scanner->getContext();
   $context->pushResult( $scanner->token() );
}
```

Кроме ошибок при синтаксическом анализе, существуют еще два условия, которые могут помешать поместить данные в стек сканера. Во-первых, из клиентского кода может поступить запрос на аннулирование корректных результатов синтаксического анализа путем вызова метода `discard()`. В результате устанавливается истинное значение свойства под названием `$discard`. Во-вторых, значения в стек могут поместить только оконечные (terminal) синтаксические анализаторы (т.е. те, которые не состоят из других анализаторов). Композитные анализаторы (т.е. экземпляры объектов `CollectionParser`, которые ниже мы будем часто называть коллекцией анализаторов), напротив, поручают сохранить результаты успешного анализа своим дочерним классам. Мы проверяем, является ли анализатор оконечным, путем вызова метода `term()`. В коллекциях анализаторов этот метод должен быть переопределен и всегда возвращать ложное 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, который реализует интерфейс `Handler`) подключен к объекту `Parser` (с помощью вызова метода `setHandler()`), то в нем вызывается метод `handleMatch()`. Мы используем обработчики, чтобы успешно построенная грамматика реально могла выполнять какие-либо полезные действия, как мы вскоре увидим.

После возврата в метод `scan()` мы вызываем объект `Scanner` (через метод `next()`), чтобы переместить позицию в исходном тексте на следующий токен. При этом вызываются методы `nextToken()` и `eatWhiteSpace()` объекта `Scanner`. И наконец мы возвращаем значение, полученное от метода `doScan()`.

Кроме `doScan()`, обратите внимание на абстрактный метод `trigger()`. Он используется для того, чтобы посмотреть, сможет ли анализатор что-то выделить из строки. Если метод `trigger()` возвращает ложное значение, значит, условия для выполнения синтаксического анализа не выполняются. Давайте посмотрим на конкретную реализацию оконечного объекта `Parser`. Класс `CharacterParse` создан для того, чтобы выделять одиночные символы.

```
namespace gi\parse;

class CharacterParse extends Parser {
   private $char;

   function __construct( $char, $name=null, $options=null ) {
      parent::__construct( $name, $options );
      $this->char = $char;
   }
```

```
   function trigger(Scanner $scanner ) {-
      return ( $scanner->token() == $this->char );
   }

   protected function doScan(Scanner $scanner ) {
      return ( $this->trigger( $scanner ) );
   }
}
```

Конструктору 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. Поскольку никакого дальнейшего сканирования выполнять не нужно, метод `doScan()` просто вызывает метод `trigger()`.

Как вы уже убедились, построить оконечный синтаксический анализатор сравнительно просто. А теперь давайте посмотрим, как работает коллекция анализаторов. Для начала определим общий суперкласс, а затем продолжим создавать конкретный пример.

```
namespace gi\parse;

// В этом абстрактном классе хранятся вложенные анализаторы
abstract class CollectionParse extends Parser {
   protected $parsers = array();

   function add(Parser $p ) {
      if ( is_null( $p ) ) {
         throw new Exception( "argument is null" );
      }
      $this->parsers[]= $p;
      return $p;
   }

   function term() {
      return false;
   }
}

class SequenceParse extends CollectionParse {

   function trigger(Scanner $scanner ) {
      if ( empty( $this->parsers ) ) {
         return false;
      }
      return $this->parsers[0]->trigger( $scanner );
   }

   protected function doScan(Scanner $scanner ) {
      $start_state = $scanner->getState();
      foreach( $this->parsers as $parser ) {
         if ( ! ( $parser->trigger( $scanner ) &&
                  $scan=$parser->scan( $scanner )) ) {
            $scanner->setState( $start_state );
            return false;
         }
```

```
        }
        return true;
    }
}
```

В абстрактном классе `CollectionParse` просто реализован метод `add()`, с помощью которого происходит агрегация анализаторов и переопределяется метод `term()` так, чтобы он возвращал ложное значение.

В методе `SequenceParse::trigger()` тестируется только первый дочерний объект `Parser`, который содержится в коллекции анализаторов; при этом вызывается его метод `trigger()`. При вызове объекта `Parser` сначала вызывается метод `CollectionParse::trigger()`, чтобы удостовериться, что самое время вызывать метод `CollectionParse::scan()`. Если метод `CollectionParse::scan()` вызван, то затем вызываются метод `doScan()` и методы `trigger()` и `scan()` всех дочерних объектов `Parser`. Если при вызове хотя бы одного метода происходит ошибка, то и метод `CollectionParse::doScan()` завершается с ошибкой.

Одной из проблем при выполнении синтаксического анализа является необходимость испытывать материал во внешней среде. Объект `SequenceParse` может содержать целое дерево синтаксических анализаторов, каждый из которых, в свою очередь, может агрегировать другие анализаторы. В результате объект `Scanner` поместит в стек один или несколько токенов и зарегистрирует полученные данные в объекте `Context`. Если при этом последний дочерний объект `Parser` вернет ложное значение, то что тогда должен делать объект `SequenceParse` со всеми результатами, помещенными в объект `Context` другими дочерними объектами, вернувшими истинное значение? Должна быть получена либо вся последовательность токенов, либо ничего. Поэтому у нас не остается выбора, и мы должны откатить объекты `Context` и `Scanner` к первоначальному состоянию. Для этого в начале метода `doScan()` выполняется сохранение их состояния, а перед самым возвратом, в случае неудачи, вызывается метод `setState()`. Естественно, что в случае успеха нам не нужно восстанавливать первоначальное состояние объектов.

Ради полноты картины ниже мы приводим все оставшиеся классы анализатора.

```
namespace gi\parse;

// Здесь достигается совпадение, если один или более
// вложенных анализаторов достигают совпадения.
class RepetitionParse extends CollectionParse {
    private $min;
    private $max;

    function __construct( $min=0, $max=0, $name=null, $options=null ) {
        parent::__construct( $name, $options );
        if ( $max < $min && $max > 0 ) {
            throw new Exception(
                    "maximum ( $max ) larger than minimum ( $min )");
        }
         $this->min = $min;
         $this->max = $max;
    }

    function trigger(Scanner $scanner ) {
        return true;
    }
```

```
   protected function doScan(Scanner $scanner ) {
      $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++;
      }
      return true;
   }
}

// Здесь достигается совпадение, если один или другой
// вложенный анализатор достигает совпадения.
class AlternationParse extends CollectionParse {

   function trigger(Scanner $scanner ) {
      foreach ( $this->parsers as $parser ) {
         if ( $parser->trigger( $scanner ) ) {
            return true;
         }
       }
       return false;
   }

   protected function doScan(Scanner $scanner ) {
      $type = $scanner->tokenType();
      foreach ( $this->parsers as $parser ) {
         $start_state = $scanner->getState();
         if ( $type == $parser->trigger( $scanner ) &&
```

```
            $parser->scan( $scanner ) ) {
            return true;
        }
      }
      $scanner->setState( $start_state );
      return false;
   }
}

// В этом оконечном анализаторе проверяется совпадение
// со строковым литералом.
class StringLiteralParse extends Parser {

   function trigger(Scanner $scanner ) {
      return ( $scanner->tokenType() == Scanner::APOS ||
               $scanner->tokenType() == Scanner::QUOTE );
   }

   protected function push(Scanner $scanner ) {
      return;
   }

   protected function doScan(Scanner $scanner ) {
      $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;
   }
}

// В этом оконечном анализаторе проверяется совпадение
// со словом.
class WordParse extends Parser {

   function __construct( $word=null, $name=null, $options=null ) {
      parent::__construct( $name, $options );
      $this->word = $word;
   }

   function trigger(Scanner $scanner ) {
      if ( $scanner->tokenType() != Scanner::WORD ) {
         return false;
      }
      if ( is_null( $this->word ) ) {
```

```
            return true;
        }
        return ( $this->word == $scanner->token() );
    }

    protected function doScan(Scanner $scanner ) {
        $ret = ( $this->trigger( $scanner ) );
        return $ret;
    }
}
```

В результате комбинирования оконечных и неоконечных объектов типа `Parser` мы можем построить достаточно интеллектуальный анализатор. Все классы типа `Parser`, используемые в нашем примере, представлены на рис. Б.1.

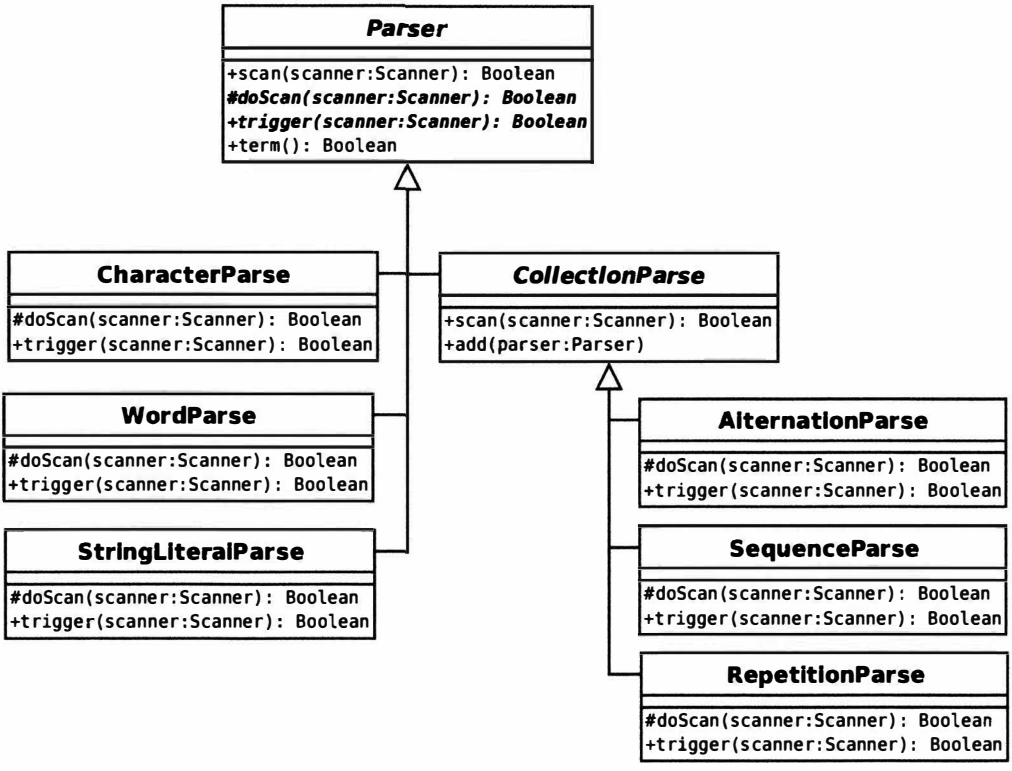


Рис. Б.1. Все классы типа `Parser`

Идея, лежащая в основе использования шаблона Composite, заключается в том, что в клиентском коде можно построить грамматики, формы записи которых очень тесно соответствуют расширенной нормальной форме Бэкуса-Наура. Эта параллель между нашими классами и фрагментами формы Бэкуса-Наура отслеживается в табл. Б.1.

Таблица Б.1. Шаблон Composite и расширенная нормальная форма Бэкуса-Наура

Класс	Пример формы записи	Описание
AlternationParse	orExpr | andExpr	Либо одно, либо другое
SequenceParse	'and' operand	Список (требуется все в нужном порядке)
RepetitionParse	(eqExpr)*	Требуется нуль или более раз

Итак, давайте создадим фрагмент клиентского кода для реализации нашего мини-языка. Напомню, что ниже приводится фрагмент расширенной нормальной формы Бэкуса-Наура, представленный в главе 11.

```
expr        ::= operand (orExpr | andExpr )*
operand     ::= ( '(' expr ')' | <stringLiteral> | variable ) ( eqExpr )*
orExpr      ::= 'or' operand
andExpr     ::= 'and' operand
eqExpr      ::= 'equals' operand
variable    ::= '$' <word>
```

В приведенном ниже простом классе строится грамматика на основе этого фрагмента и запускаются вычисления.

```
namespace gi\parse;

class MarkParse {
   private $expression;
   private $operand;
   private $interpreter;
   private $context;

   function __construct( $statement ) {
      $this->compile( $statement );
   }

   function evaluate( $input ) {
      $icontext = new InterpreterContext();
      $prefab = new VariableExpression('input', $input );

     // Добавим переменную input в контекст
     $prefab->interpret( $icontext );

     $this->interpreter->interpret( $icontext );
     $result = $icontext->lookup( $this->interpreter );
     return $result;
   }

   function compile( $statement_str ) {
      // Построим дерево анализатора
      $context = new \gi\parse\Context();
      $scanner = new \gi\parse\Scanner(
         new \gi\parse\StringReader($statement_str), $context );
      $statement = $this->expression();
      $scanresult = $statement->scan( $scanner );

      if ( ! $scanresult ||
```

```
            $scanner->tokenType() != \gi\parse\Scanner::EOF ) {
         $msg  = "";
         $msg .= " line: {$scanner->line_no()} ";
         $msg .= " char: {$scanner->char_no()}";
         $msg .= " token: {$scanner->token()}\n";
         throw new Exception( $msg );
      }
      $this->interpreter = $scanner->getContext()->popResult();
   }

   function expression() {
      if ( ! isset( $this->expression ) ) {
         $this->expression = new \gi\parse\SequenceParse();
         $this->expression->add( $this->operand() );
         $bools = new \gi\parse\RepetitionParse( );
         $whichbool = new \gi\parse\AlternationParse();
         $whichbool->add( $this->orExpr() );
         $whichbool->add( $this->andExpr() );
         $bools->add( $whichbool );
         $this->expression->add( $bools );
      }
      return $this->expression;
   }

   function orExpr() {
      $or = new \gi\parse\SequenceParse( );
      $or->add( new \gi\parse\WordParse('or') )->discard();
      $or->add( $this->operand() );
      $or->setHandler( new BooleanOrHandler() );
      return $or;
   }

   function andExpr() {
      $and = new \gi\parse\SequenceParse();
      $and->add( new \gi\parse\WordParse('and') )->discard();
      $and->add( $this->operand() );
      $and->setHandler( new BooleanAndHandler() );
      return $and;
   }

   function operand() {
      if ( ! isset( $this->operand ) ) {
         $this->operand = new \gi\parse\SequenceParse( );
         $comp = new \gi\parse\AlternationParse( );
         $exp = new \gi\parse\SequenceParse( );
         $exp->add( new \gi\parse\CharacterParse( '(' ))->discard();
         $exp->add( $this->expression() );
         $exp->add( new \gi\parse\CharacterParse( ')' ))->discard();
         $comp->add( $exp );
         $comp->add( new \gi\parse\StringLiteralParse() )
              ->setHandler( new \gi\parse\StringLiteralHandler() );
         $comp->add( $this->variable() );
         $this->operand->add( $comp );
         $this->operand->add( new \gi\parse\RepetitionParse( ) )
```

```
                ->add($this->eqExpr());
        }
        return $this->operand;
    }

    function eqExpr() {
        $equals = new \gi\parse\SequenceParse();
        $equals->add( new \gi\parse\WordParse('equals') )->discard();
        $equals->add( $this->operand() );
        $equals->setHandler( new EqualsHandler() );
        return $equals;
    }

    function variable() {
        $variable = new \gi\parse\SequenceParse();
        $variable->add( new \gi\parse\CharacterParse( '$' ))->discard();
        $variable->add( new \gi\parse\WordParse());
        $variable->setHandler( new VariableHandler() );
        return $variable;
    }
}
```

Приведенный выше код может создать у вас иллюзию сложного класса, однако все, что он делает, — это строит грамматику, которую мы уже определили. Большинство методов аналогично продукциям (т.е. именам, расположенным в начале продукционной строки расширенной нормальной формы Бэкуса-Наура, таким как `eqExpr` или `andExpr`). Если вы проанализируете метод `expression()`, то обнаружите, что мы строим в нем правило, аналогичное определенному выше в расширенной нормальной форме Бэкуса-Наура.

```
// expr ::= operand (orExpr | andExpr )*
function expression() {
    if ( ! isset( $this->expression ) ) {
        $this->expression = new \gi\parse\SequenceParse();
        $this->expression->add( $this->operand() );
        $bools = new \gi\parse\RepetitionParse( );
        $whichbool = new \gi\parse\AlternationParse();
        $whichbool->add( $this->orExpr() );
        $whichbool->add( $this->andExpr() );
        $bools->add( $whichbool );
        $this->expression->add( $bools );
    }
    return $this->expression;
}
```

И в коде, и в расширенной нормальной форме Бэкуса-Наура мы определяем последовательность, которая состоит из ссылок на операнды, за которыми нуль или более раз следуют альтернативы `orExpr` или `andExpr`. Обратите внимание на то, что мы сохраняем объект `Parser`, возвращенный этим методом, в переменной-свойстве. Это сделано для предотвращения зацикливания, поскольку методы, вызываемые из `expression()`, сами вызывают `expression()`.

Единственные методы, которые, кроме построения грамматики, делают еще что-то, — это `compile()` и `evaluate()`. Метод `compile()` можно вызвать напрямую или автоматически из конструктора. Ему передается строка выражения, которая используется для создания объекта `Scanner`. Он вызывает метод `expression()`, кото-

рый возвращает дерево объектов типа `Parser`, составляющее грамматику. Затем он вызывает метод `Parser::scan()`, передавая ему объект `Scanner`. Если исходный код невозможно проанализировать, метод `compile()` генерирует исключение. В противном случае он извлекает результат компиляции из контекста объекта `Scanner`. Как мы вскоре увидим, это должен быть объект типа `Expression`. Результат сохраняется в свойстве `$interpreter`.

Метод `evaluate()` делает значение доступным дереву объектов `Expression`. Для этого заранее определяется объект `VariableExpression`, соответствующий переменной `$input`, который затем регистрируется в объекте `Context`. В результате он передается основному объекту типа `Expression`. Так же, как и глобальные переменные в PHP типа `$_REQUEST`, эта переменная `$input` будет всегда доступна программистам, использующим язык MarkLogic.

На заметку. Класс `VariableExpression`, являющийся частью шаблона Interpreter, подробно описан в главе 11.

Метод `evaluate()` вызывает метод `Expression::interpret()` для получения окончательного результата. Не забывайте, что результаты работы интерпретатора нужно извлечь из объекта `Context`.

Выше был описан процесс синтаксического анализа текста и построения грамматики. В главе 11 мы также увидели, как используется шаблон Interpreter для комбинирования объектов `Expression` и выполнения запроса. Однако до сих пор пока непонятно, как связаны между собой эти два процесса. Как нам передать результаты из дерева анализатора интерпретатору? Ответ заключается в использовании объектов типа `Handler`, которые связываются с объектами типа `Parser` посредством вызова `Parser::setHandler()`. Давайте рассмотрим один из способов обработки переменных. Мы связываем объект `VariableHandler` с объектом `Parser` в методе `variable()`.

```
$variable->setHandler( new VariableHandler() );
```

Ниже приведено определение класса `Handler` и интерфейса, который он реализует.

```
namespace gi\parse;

interface Handler {

   function handleMatch(Parser $parser,
                        Scanner $scanner );
}
```

Ниже приведена реализация класса `VariableHandler`.

```
class VariableHandler implements \gi\parse\Handler {

   function handleMatch(\gi\parse\Parser $parser,
                        \gi\parse\Scanner $scanner ) {
      $varname = $scanner->getContext()->popResult();
      $scanner->getContext()->pushResult(
              new VariableExpression( $varname ) );
   }
}
```

Если объект `Parser`, с которым связан объект `VariableHandler`, во время операции сканирования находит соответствие, вызывается метод `handleMatch()`. По определению последний элемент в стеке будет соответствовать имени переменной.

Мы удаляем его из стека и заменяем новым объектом `VariableExpression`, содержащим правильное имя. Аналогичные подходы используются для создания объектов `EqualsExpression`, `LiteralExpression` и т.п.

Ниже приведены остальные обработчики.

```
class StringLiteralHandler implements \gi\parse\Handler {

   function handleMatch( \gi\parse\Parser $parser,
                         \gi\parse\Scanner $scanner ) {
      $value = $scanner->getContext()->popResult();
      $scanner->getContext()->pushResult(
            new LiteralExpression( $value ) );
   }
}

class EqualsHandler implements \gi\parse\Handler {

   function handleMatch( \gi\parse\Parser $parser,
                         \gi\parse\Scanner $scanner ) {
      $compl = $scanner->getContext()->popResult();
      $comp2 = $scanner->getContext()->popResult();
      $scanner->getContext()->pushResult(
          new EqualsExpression( $compl, $comp2 ) );
   }
}

class BooleanOrHandler implements \gi\parse\Handler {

   function handleMatch( \gi\parse\Parser $parser,
                         \gi\parse\Scanner $scanner ) {
      $compl = $scanner->getContext()->popResult();
      $comp2 = $scanner->getContext()->popResult();
      $scanner->getContext()->pushResult(
         new BooleanOrExpression( $compl, $comp2 ) );
   }
}

class BooleanAndHandler implements \gi\parse\Handler {

   function handleMatch( \gi\parse\Parser $parser,
                         \gi\parse\Scanner $scanner ) {
      $compl = $scanner->getContext()->popResult();
      $comp2 = $scanner->getContext()->popResult();
      $scanner->getContext()->pushResult(
          new BooleanAndExpression( $comp1, $comp2 ) );
   }
}
```

Не забывайте, что нам еще нужен пример с шаблоном Interpreter, описанный в главе 11. Тогда мы сможем работать с классом `MarkParse`, как показано ниже.

```
$input = 'five';
$statement = "( \$input equals 'five')";

$engine = new MarkParse( $statement );
```

```
$result = $engine->evaluate( $input );
print "input: $input evaluating: $statement\n";
if ( $result ) {
   print "true!\n";
} else {
   print "false!\n";
}
```

Этот фрагмент должен вывести следующие результаты.

```
input: five evaluating: ( $input equals 'five')
true!
```

Предметный указатель

К

Л

M

H

O

П

Р

С

Т

У

Ф

Х

Ш

Э

Ю

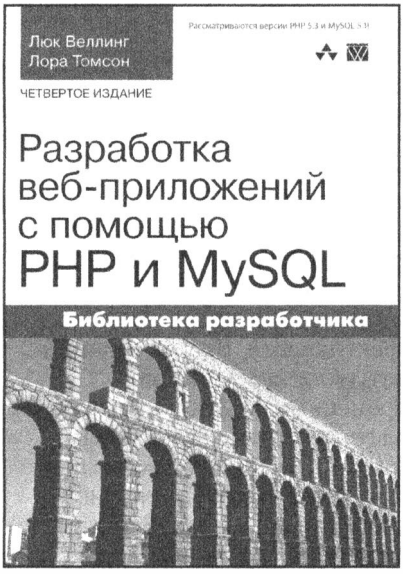
Я

РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ PHP И MYSQL

БИБЛИОТЕКА РАЗРАБОТЧИКА

ЧЕТВЕРТОЕ ИЗДАНИЕ

Люк Веллинг
Лора Томсон



Эта книга предназначена для тех, кто знаком с основами HTML и ранее разрабатывал программы на современных языках программирования, но, возможно, не занимался программированием для веб или не использовал реляционные базы данных. В ней подробно описано применение последних версий PHP и MySQL для построения крупных коммерческих веб-сайтов. Основное внимание в книге уделено реальным приложениям. Здесь рассматриваются как простые интерактивные системы приема заказов, так и различные аспекты электронных систем продажи и безопасности во взаимосвязи с созданием реального веб-сайта. Подробно описаны все стадии разработки типовых проектов на PHP и MySQL, в числе которых служба веб-почты, приложение поддержки веб-форумов и электронный книжный магазин. Книга ориентирована на профессиональных разработчиков, но будет полезной и начинающим.

www.williamspublishing.com

ISBN 978-5-8459-1574-0

в продаже

PHP и MySQL: создание интернет-магазинов *2-е издание*

Ларри Ульман

www.williamspublishing.com

В книге рассматриваются примеры двух полнофункциональных интернет-магазинов, благодаря изучению которых читатели смогут сравнить разные сценарии электронной коммерции. Вы узнаете, как спроектировать визуальный интерфейс и создать базу данных сайта, как реализовать представление контента и сгенерировать онлайн-каталог, как управлять корзиной товаров и проводить платежи, как принимать и выполнять заказы с учетом требований безопасности и эффективности. Второе издание книги включает описание современных функциональных средств, присущих платежным системам PayPal и Authorize.net, демонстрируется применение технологий Ajax и JavaScript, а также описано подключение интернет-магазинов к платежной системе Яндекс. Деньги.

ISBN 978-5-8459-1939-7 **в продаже**

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

PHP ОБЪЕКТЫ, ШАБЛОНЫ И МЕТОДИКИ ПРОГРАММИРОВАНИЯ

4-е издание

Четвертое издание книги было пересмотрено и дополнено новым материалом. Книга начинается с обзора объектно-ориентированных возможностей PHP, в который включены важные темы, такие как определение классов, наследование, инкапсуляция, рефлексия и многое другое. Этот материал закладывает основы объектно-ориентированного проектирования и программирования на PHP. Вы изучите также некоторые основополагающие принципы проектирования. В этом издании книги также описаны возможности, появившиеся в PHP версии 5.5, такие как трейты, дополнительные расширения на основе рефлексии, уточнения типов параметров методов, улучшенная обработка исключений и много других мелких расширений языка.

Следующая часть книги посвящена проектным шаблонам, которые органически дополняют тему ООП и являются описанием элегантных решений распространенных проблем, возникающих при проектировании программного обеспечения. В ней описываются концепции шаблонов проектных решений и показаны способы реализации нескольких важных шаблонов в приложениях на PHP. В этой же части приведен материал, посвященный шаблонам корпоративных приложений и баз данных.

В последней части книги описывается несколько важных утилит и методик, помогающих осуществить успешный проект на основе разрозненных кусков кода. В этой части книги описано, как управлять работой нескольких программистов с помощью Git, как выполнить построение и развертывание проекта с помощью Phing и PEAR. Вы также изучите стратегии автоматического тестирования и построения проектов.

Кроме обсуждения передовых средств построения и тестирования проектов, а также серверов непрерывной интеграции, в последней части книги описаны лучшие методики организации работы на основе системы контроля версий Git. Тем самым была отражена общая тенденция перехода на новую платформу, которая наметилась в среде разработчиков с момента выхода третьего издания книги.

Эта книга посвящена трем важным темам: основам объектов, принципам объектно-ориентированного проектирования и лучшим методикам их применения. Изучив их в комплексе читатель сможет создавать первоклассные и отказоустойчивые программные системы.

ОСНОВНЫЕ ТЕМЫ КНИГИ:

- основы объектов: написание классов и методов, создание объектов, разработка сложной иерархии классов на основе наследования;
- сложные объектно-ориентированные темы, включая статические методы и свойства, обработку ошибочных ситуаций с помощью исключений, создание абстрактных классов и интерфейсов;
- общие сведения об основных принципах объектно-ориентированного проектирования и их использование для создания эффективной структуры классов и объектов приложения;
- изучение базовых проектных шаблонов, их структуры и основных принципов, положенных в их основу;
- описание ряда основополагающих шаблонов, которые можно применить в собственных проектах;
- изучение методик и средств, гарантирующих успешное выполнение проекта, включая модульное тестирование, контроль версий, построение и развертывание проекта, управление пакетами, а также сервер непрерывной интеграции Jenkins.

НА ВЕБ-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с веб-сайта издательства по адресу: http://www.williamspublishing.com/Books/978-5-8459-1922-9.html.

Мэтт Зандстра почти 20 лет проработал веб-программистом, консультантом по PHP и составителем технической документации. Он был старшим разработчиком в компании Yahoo! и работал в офисах компании как в Лондоне, так и в Силиконовой долине. Сейчас он зарабатывает себе на жизнь в качестве свободного консультанта и писателя.

Категория: программирование/PHP
Предмет рассмотрения: PHP 5.5
Уровень: для пользователей средней и высокой квалифи

ISBN 978-5-8459-1922-9

ВИЛЬЯМС

www.williamspublishing.c