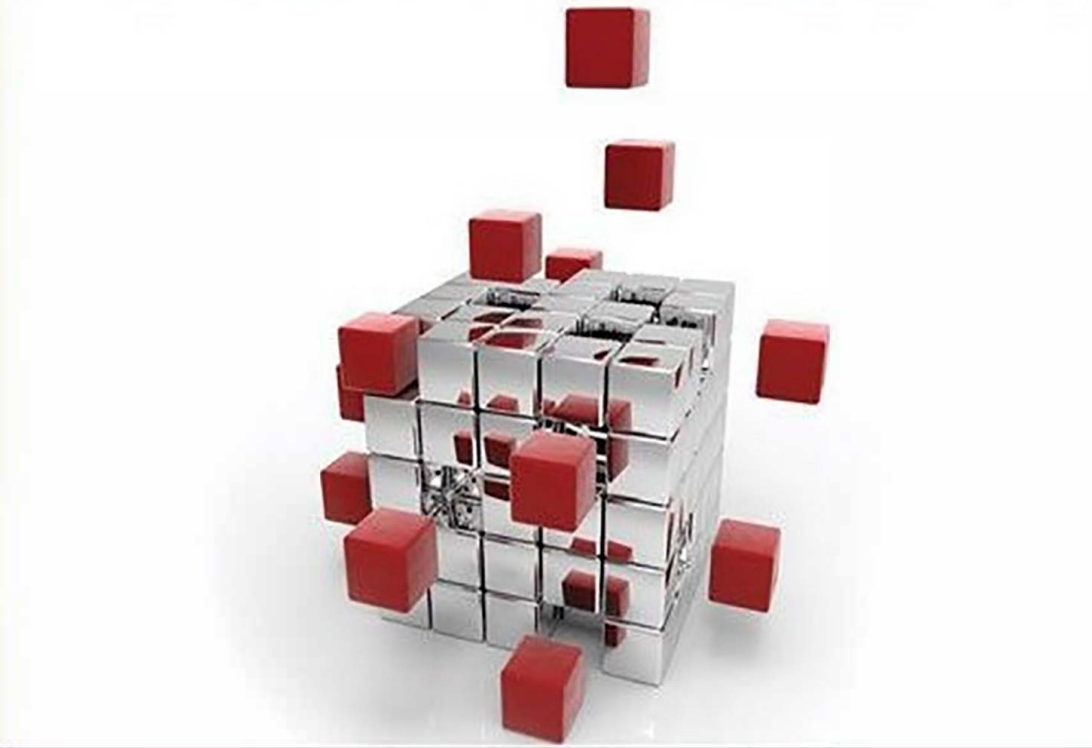
Java EE

паттерны проектирования для профессионалов



Мурат Йенер, Алекс Фидом

ПИТЕР

Мурат Йенер, Алекс Фидом

Java EE
паттерны проектирования
для профессионалов

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Киев · Екатеринбург · Самара · Минск

2016

ББК 32.973.2-018-02
УДК 004.42
И30

Мурат Йенер, Алекс Фидом

И30 Java EE. Паттерны проектирования для профессионалов. — СПб.: Питер, 2016. — 240 с.: ил.

ISBN 978-5-496-01945-3

Книга «Java EE. Паттерны проектирования для профессионалов» — незаменимый ресурс для всех, кто желает более эффективно работать с Java EE, а также единственная книга, в которой рассмотрены как теория, так и практика использования паттернов проектирования на примерах реальных прикладных задач.

Авторы знакомят читателя и с фундаментальными, и с наиболее передовыми возможностями Java EE 7, досконально рассматривают каждый из паттернов и демонстрируют, как эти паттерны применяются при решении повседневных прикладных задач.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.42

ISBN 978-1118843413 англ.
ISBN 978-5-496-01945-3

Краткое содержание

Часть III. Подведем итоги

Оглавление

Часть II. Реализация паттернов проектирования в Java EE

Часть III. Подведем итоги

Посвящается Нилай и всей моей семье
(Семре и Мустафе Йенерам) в благодарность
за всю помощь и необходимое мне
для написания этой книги время.

Мурат

Посвящается Мариу —
за всю помощь и поддержку.

Алекс

Об авторах

Мурат Йенер — фанатик программирования и коммитер[1] открытого программного обеспечения; в данный момент разработчик под платформу Android в подразделении Intel New Devices Group. Он обладает обширным опытом разработки приложений на языке Java, веб-фреймворков, приложений для платформы Java EE и модульных приложений на основе спецификации OSGi. Кроме того, он занимается созданием учебных курсов и преподает. Мурат — коммитер свободной среды разработки Eclipse и один из первых коммитеров проекта Eclipse Libra. Сейчас он разрабатывает нативные и гибридные мобильные приложения с применением языка HTML5 и фреймворка mGWT.

Мурат был руководителем пользовательской группы конференции GDC в Стамбуле с 2009 года, организуя там различные мероприятия, участвуя и выступая в них. Он также регулярно выступает на конференциях JavOne, EclipseCon и Devoxx.

- **Linkedin** — www.linkedin.com/in/muratyener.
- **Twitter** — @yenerm.
- **Blog** — www.devchronicles.com.

Алекс Фидом — ведущий Java-разработчик в Indigo Code Collective indigocodecollective.com (подразделение E-scape Group), где он играет ключевую роль в создании архитектурного дизайна и разработке основанной на микросервисе, созданной на заказ лотереи и платформы мгновенных лотерей.

До этого он разрабатывал программное обеспечение для банкоматов международного испанского банка и программное обеспечение для анализа качества кода для ИТ-консалтинга.

Алекс обладает опытом разработки веб-приложений на языке Java для различных сфер деятельности, включая финансовое дело, электронное обучение, лотереи и разработку программного обеспечения. Страсть к разработке приводила его в проекты по всей Европе и за ее пределами. Он ведет блог на alextheedom.com и помогает коллегам в решении проблем на онлайн-форумах.

- **Linkedin** — www.linkedin.com/in/alextheedom.
- **Twitter** — @alextheedom.
- **Blog** — www.alextheedom.com.

---

[1] Коммит (от англ. commit — «фиксировать») — сохранение, фиксация (в архиве, репозитарии и др.) изменений в программном коде. Коммитер — специалист, выполняющий коммиты. — *Примеч. ред.*

О техническом редакторе

Мухаммед Санаулла — разработчик программного обеспечения с более чем пятилетним опытом разработки. В настоящее время работает на крупнейшее индийское предприятие в сфере электронной коммерции, а также является модератором на форуме JavaRanch. В свободное от работы время он присматривает за своей прелестной маленькой дочкой. Своими экспериментами и мыслями на тему разработки программного обеспечения он делится по адресу http://blog.sanaulla.info/.

Благодарности

Как всегда говорит мой соавтор Алекс, мы стремились написать книгу, которую нам самим хотелось бы иметь у себя и читать. Я хочу поблагодарить Алекса за все его терпение, тяжелый труд и глубокие знания. Без него эта книга и в помине не была бы так хороша.

Я благодарен Мэри Джеймс, нашему бывшему редактору, которая связалась со мной насчет написания книги по фреймворку Spring, но прислушалась к моим идеям, сформировавшим основу данной книги. Без ее поддержки и руководства эта книга никогда бы не стала реальностью. Никаких слов не было бы достаточно, чтобы отблагодарить Адаоби Оби Тультон, которая терпеливо работала над всеми деталями плана, в то же время освобождая нас от львиной доли связанного с ним стресса. И спасибо, конечно, всем в издательстве Wrox/Wiley, благодаря кому эта книга попала на полки. Спасибо также Резе Рахману за всю его поддержку.

Я должен поблагодарить трех важных для меня людей, в огромной мере повлиявших на то, где я сейчас нахожусь в моей профессиональной жизни.

Во-первых, спасибо моему папе, Мустафе Йенеру, за покупку мне в раннем детстве первого компьютера — C64, в то время как я просил о рельсовых игрушечных машинках. Именно на этом компьютере я написал свои первые программы.

Во-вторых, спасибо моему диссертационному научному руководителю, профессору Махиру Вардару. Я обязан ему всеми своевременными советами, которые были мне необходимы для начала карьеры.

И наконец, спасибо наставнику и другу всей моей жизни (а также экс-боссу) Наджи Даи, который научил меня практически всему, что я знаю о том, что значит быть профессиональным разработчиком программного обеспечения.

Мурат

Мы очень гордимся этой, нашей первой, книгой и надеемся, что вы получите так же много от ее прочтения, как мы получили от написания. Мы подходили к написанию с той точки зрения, что это должна быть такая книга, которую мы сами бы купили, если бы не написали ее. Нам это удалось.

Однако эта книга не была бы возможна без преданности, терпения и понимания многих других людей, которые напрямую и косвенно внесли свой вклад в ее создание. Мы хотели бы выразить признательность за вклад, внесенный опытной и преданной своему делу командой издательства Wiley Publishing. Они были с нами от начала и до конца, веря, что все получится. Мы хотели бы отдельно поблагодарить Мэри Джеймс, нашего редактора, чья поддержка сделала эту книгу реальностью. Спасибо также Адаоби Оби Тультон, чье терпение

и мягкое подзадоривание держало нас в тонусе и чье внимание к деталям спасло нас от ошибок. Я хотел бы поблагодарить моего соавтора Мурата Йенера за его вдохновение и чувство юмора, которые сделали эту книгу оригинальной. И в последнюю по счету, но отнюдь не по важности очередь я хотел бы поблагодарить мою жену, Марию Евгению Гарсиа Гарсиа, за поддержку и понимание во время написания мной этой книги. Спасибо тебе.

Алекс

Предисловие

Невежда поднимает вопросы, на которые мудрецы
ответили тысячелетия тому назад.

Иоганн Вольфганг фон Гете

Паттерны проектирования — наша связь с прошлым и будущим. Из них состоит базовый язык, представляющий собой хорошо понятные решения для распространенных проблем, которые одаренные инженеры добавили к нашей совокупной базе знаний. Паттерны проектирования, или «синьки», существуют в том или ином виде в каждой инженерной сфере деятельности. Разработка программного обеспечения не исключение. Более того, именно паттерны проектирования, вероятно, наша наиболее осязаемая связь с инженерным искусством, а не более систематизированный и жестко организованный мир кустарей или ремесленников.

Искусство и наука паттернов проектирования были принесены в мир инженерии разработки ПО, а в особенности на корпоративную платформу Java (Java Enterprise Platform) основополагающей книгой, написанной «Бандой четырех» (Gang of Four, GoF). С тех пор они были с нами на протяжении наших приключений на платформах J2EE, Spring и вот теперь современной облегченной Java EE. Для этого есть более чем достаточные основания. Разработчики серверных приложений на языке Java склонны к написанию предназначенной для решения критически важных задач разновидности приложений, должных выдержать испытание временем и, следовательно, извлекающих выгоду из того порядка, который олицетворяют паттерны проектирования.

На самом деле требуется особый тип личности для написания книги по паттернам проектирования, не говоря уже про книгу о том, как использовать паттерны проектирования в приложениях Java EE. Для этого нужно не только базовое знание интерфейсов программирования приложений (API) и самих паттернов, но и то глубокое понимание, которое может прийти только с добытым тяжелым трудом опытом, а также врожденное умение изящно объяснять сложные концепции. Я рад, что у платформы Java EE теперь есть Мурат и Алекс для свершения этого колоссального подвига.

Эта книга заполняет необходимый пробел в знаниях о паттернах проектирования, и заполняет его хорошо. Очень хорошо также то, что эта книга основана на последних достижениях и включает в себя не только платформы Java EE 6 или Java EE 5, но и Java EE 7. На деле многие из охваченных этой книгой паттернов проектирования, такие как *«Одиночка»* (Singleton), *«Фабрика»* (Factory), *«Модель — представление — контроллер»* (Model — View — Controller, MVC), *«Декоратор»* (Decorator) и *«Наблюдатель»* (Observer), сейчас включены в саму платформу Java EE. Другие, такие как *«Фасад»* (Facade), *«Объект доступа к данным»* (DataAccess

Object, DAO) и *«Объект передачи данных»* (Data Transfer Object, DTO), изящно прилажены сверху. Мурат и Алекс энергично берутся за каждый паттерн, объясняют его практическую мотивацию и обсуждают его соответствие Java EE.

Для меня честь — написать маленькое вступление к этой очень важной книге, которая, я надеюсь, станет весьма полезной для каждого хорошего разработчика под платформу Java EE. Я надеюсь, вы получите от нее удовольствие и она поможет вам писать лучшие, более полно отвечающие требованиям корпоративные приложения на языке Java.

М. Реза Рахман,
пропагандист Java EE/Glassfish.
Корпорация Oracle

Введение

В этом издании рассматриваются классические паттерны проектирования, впервые упомянутые в знаменитой книге, написанной GoF[1], с учетом модернизации их применительно к платформам Java EE 6 и 7.

В каждой главе мы описываем традиционную реализацию паттерна и затем показываем, как реализовать его, используя ориентированную на платформу Java EE семантику.

Мы используем полные примеры кода для демонстрации как традиционной реализации, так и реализации под платформу Java EE, и дополняем каждую главу примерами из практики, демонстрирующими правильное (и ошибочное) применение паттернов.

Мы исследуем «за» и «против» каждого паттерна и изучаем области их применения. В конце каждой главы приведены упражнения для проверки степени вашего понимания данного паттерна в Java EE.

Для кого предназначена эта книга

Эта книга — для всех, вне зависимости от уровня опыта. Она охватывает почти всю информацию о паттернах, начиная с того, как их описывают в других книгах, до кода простой реализации на языке Java, реализации на платформе Java EE и, наконец, до примеров из практики: как и когда использовать конкретный паттерн. В ней также есть истории из реальной жизни, в которых обсуждаются удачные и неудачные практики применения паттернов.

Полезным при прочтении книги будет наличие базовых знаний паттернов проектирования и платформы Java EE.

Если вы уже имели дело с паттернами и базовыми реализациями на языке Java, то можете перейти сразу к реализациям для Java EE. Впрочем, может оказаться полезным освежить вашу память и знания паттернов проектирования.

Что охватывает эта книга

Эта книга охватывает все классические паттерны проектирования, предлагаемые платформой Java EE в качестве части стандартной реализации, а также некоторые

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2010. — 368 с.: ил. (Серия «Библиотека программиста»).

новые паттерны. Охват начинается с Java EE 5 и включает последнюю из доступных на настоящий момент версий платформы — Java EE 7.

Мы надеемся, что это издание станет справочником, который надолго поселится на вашей книжной полке.

Как эта книга структурирована

Каждая глава сконцентрирована на одном паттерне проектирования. Если паттерн классический, то после объяснения его сущности дается простая реализация на языке Java. Каждая глава предлагает истории из практики, иллюстрирующие паттерн, на котором сконцентрирована глава, положительными или отрицательными примерами из реальной жизни. За историей из практики следует реализация на платформе Java EE, пример и объяснение. Каждый пример кода может быть запущен отдельно от других. Наконец, каждая глава завершается описанием того, где и как эффективно использовать этот паттерн.

Что нужно, чтобы пользоваться этой книгой

Для запуска примеров, приведенных в этой книге, достаточно любого современного компьютера с операционной системой, для которой существует реализация виртуальной машины Java (Java Virtual Machine, JVM). Для удобства написания кода вам понадобится интегрированная среда разработки (IDE) по вашему выбору. Примеры могут быть запущены под любой распространенной современной IDE, включая Eclipse, NetBeans и IntelliJ.

Вам понадобится комплект разработчика для языка Java (Java Development Kit, JDK) под платформу Java EE 7, чтобы компилировать и запускать примеры кода, но некоторые из них также работают на большинстве JDK под предыдущие версии платформы Java EE.

Для запуска примеров вы можете использовать любой совместимый с Java EE 7 сервер приложений. Мы запускали все примеры на Glassfish, являющемся эталонной реализацией сервера приложений, и TomEE — версии популярного Java-веб-сервера Tomcat под платформу Java EE. Вы можете использовать любой сервер, но поскольку Glassfish — эталонная реализация, вы можете захотеть попробовать его для примеров кода.

Для запуска примеров из книги вам понадобится следующее:

- операционная система, для которой есть JDK под платформу Java EE 7, такая как Linux, Mac OS X или Windows;
- JDK под платформу Java EE 7;
- IDE на ваш выбор, например Eclipse для разработчиков Java EE, NetBeans или IntelliJ;
- совместимый с Java EE 7 сервер приложений, например Glassfish или TomEE.

Исходный код примеров доступен для загрузки с сайта издательства Wrox: http://www.wrox.com/go/projavaeedesignpatterns.

Мотивация для написания

В ноябре 2011 года, после спора о том, выбрать Java EE или Spring для проекта, я[1] вернулся за свой стол и написал в своем блоге сообщение, озаглавленное «Java EE 6 и эвоки»[2], очень быстро ставшее популярным. История была основана на телесериале под названием «Как я встретил вашу маму». В этом сериале плейбой Барни выдвигает теорию, касающуюся эвоков — похожих на плюшевых медвежат созданий, появившихся в эпизоде VI саги «Звездные войны». Поклонники саги неоднозначно относятся к эвокам.

Согласно Барни, те, кто родился до 25 мая 1983 года — дня, когда вышел в прокат фильм «Возвращение джедая», считают эвоков несерьезными и просто ненавидят их. Однако родившиеся после этой даты находят эвоков привлекательными, поскольку те напоминают им плюшевых медвежат.

Теперь вернемся к моему рассказу. Спор с заказчиком насчет платформы Java EE по сравнению с фреймворком Spring привел меня к осознанию схожести с теорией про эвоков. Люди, которые настолько стары, что использовали платформу J2EE 1.4 (EJB 1.0/2.0/2.1) в корпоративных проектах, помнят медленную, непродуктивную среду разработки с пожирающими оперативную память IDE и серверами, загружающимися по нескольку минут. Архитектура была технически переусложненной и, вероятно, неудачной, что привело к миграции на Spring. Такие пользователи склонны страстно ненавидеть Java EE вне зависимости от того, с какой версией они имеют дело. Выпуск Java EE 5 был недооценен и никого, по правде говоря, не впечатлил.

Платформа Java EE никогда не станет опять J2EE. Сейчас она открыта, поддерживается большим сообществом специалистов и трансформируется, усваивая полезные идеи от таких фреймворков, как Spring и Hibernate. Первым большим изменением были архитектура и стиль кодирования. Архитектура Enterprise JavaBeans (EJB) последовала за облегченной моделью «простых объектов Java в старом стиле» (Plain Old Java Object, POJO), практически непригодные для использования компоненты-сущности (entity beans) были заменены на интерфейс программирования приложений Java Persistence API (JPA), REST (Representational State Transfer — репрезентативная передача состояния), и веб-сервисы стали стандартными и неотъемлемыми частями среды выполнения, а аннотации заменили XML-конфигурации. Тем не менее некоторые могли спорить, что платформа Java EE 5 была не готова к серьезному изменению, поскольку была не столь зрелой, как Spring, а среда разработки все еще оставалась довольно неповоротливой. Использование Spring с веб-сервером Tomcat вместо компонентов EJB и Java EE 5 на сервере приложений значительно повышало производительность разработки, но Java EE 5 все равно была большим шагом вперед по направлению к проектированию, улучшению и конструированию платформы Enterprise Java с нуля.

За этим изменением последовали Java EE 6 и 7, использовавшие те же принципы и идеи, что и Java EE 5. Платформа Java EE — отличный выбор для разработки, но благодаря теории эвоков споры еще не окончены.

---

[1] Раздел написан муратом Йенером.

[2] Java EE 6 and the Ewoks: http://www.devchronicles.com/2011/11/javaee6-and-ewoks.html.

Был жаркий августовский день, когда мне впервые позвонили из издательства Wrox/Wiley насчет того, не заинтересует ли меня написание книги по фреймворку Spring. У меня был опыт разработки и уверенность в своих силах относительно реализации и разработки под Spring, но уже существовали тонны написанных о нем книг, так что было непонятно, какая будет польза в написании еще одной.

Более того, я использовал платформу Java EE больше чем когда-либо с тех пор, как была выпущена версия 6. Учитывая споры о сравнении Java EE с фреймворком Spring, мои сообщения в блоге и эвоков, я захотел написать о Java EE. Однако, подобно Spring, уже существовало много восхищавших меня великолепных книг по платформе Java EE. У меня всегда было чувство, что некоторые свойства Java EE были недооценены. Платформа Java EE имеет замечательные встроенные реализации паттернов проектирования с весьма простым использованием аннотаций.

Классические паттерны, перечисленные в книге GoF, в значительной степени применялись почти во всех языках, фреймворках и почти на всех платформах. Ни J2EE, ни Java EE не были исключениями. На самом деле платформа Java EE сделала смелый шаг вперед, обеспечивая реализации по умолчанию для многих из этих паттернов, но до сих пор большинство даже опытных разработчиков недооценивают значение таких готовых для использования реализаций.

Я писал в своем блоге об этих паттернах на протяжении почти года, так что решил сделать встречное предложение: написать книгу по классическим паттернам проектирования в Java EE. И раз вы читаете сейчас эту книгу, легко можете догадаться, что ответная реакция была положительной.

Эта книга заполняет пробел между платформой Java EE и классическими паттернами проектирования из GoF, а также содержит описание новых паттернов. Таким образом, мы создали не просто еще одну книгу о платформе Java EE, а систематический каталог паттернов проектирования в Java EE.

Я начал писать, читать лекции и вести блог на тему паттернов проектирования в Java EE, чтобы расширить свои знания и получить опыт работы на платформе, в которую по-настоящему поверил, так что самым лучшим в написании этой книги для меня был шанс написать о чем-то, чем я действительно страстно увлечен. Хотя примеры в моем блоге были несложными, я при необходимости использовал его в качестве справочника.

У каждой написанной мной и моим соавтором Алексом главы одна и та же цель: написать то, что нам самим хотелось бы прочесть. Результат — книга, которую мы оба хотели бы хранить в качестве справочника.

Мы надеемся, что чтение этой книги доставит вам столько же удовольствия, сколько нам доставило ее написание.

Соглашения

Чтобы помочь вам следить за происходящим и извлечь как можно больше из текста, мы используем в книге несколько соглашений.

ПРИМЕЧАНИЕ

Примечания указывают на примечания, советы, оригинальные решения или отступления от текущего обсуждения.

В тексте используются следующие стили:

- *курсивом* мы выделяем новые термины и просто важные формулировки, когда знакомим вас с ними;
- нажатия клавиш на клавиатуре мы указываем следующим образом: **Ctrl+A**;
- имена файлов, URL и код внутри текста мы оформляем вот так: `persistence.properties`;
- примеры кода мы приводим двумя способами:

```
Используем моноширинный шрифт без выделения для большинства примеров кода.
```

```
Используем полужирный шрифт, чтобы выделить код, который особенно важен в данном
контексте, или чтобы показать отличия от предыдущего фрагмента кода.
```

Исходный код

При работе с примерами в этой книге вы можете выбрать один из двух вариантов: или набирать весь код вручную, или использовать файлы исходного кода, которые к ней прилагаются. Весь исходный код, использованный в книге, доступен для скачивания по ссылке **http://www.wrox.com/go/projavaeedesignpatterns**.

Каждая глава начинается со знакомства с простой реализацией паттерна (если глава таковому посвящена) на языке Java. Затем приводится реализация паттерна под платформу Java EE, которая может быть скомпилирована и запущена только на Java EE JDK и Java EE-совместимом сервере приложений.

Большая часть кода на сайте **http://www.wrox.com/** упакована в ZIP, RAR или аналогичный формат архива, подходящий для нужной платформы. После скачивания кода просто разархивируйте его, используя подходящую утилиту-архиватор.

Ошибки

Мы сделали все, что было в наших силах, чтобы гарантировать отсутствие ошибок в тексте и коде. Однако никто не совершенен, и ошибки все же встречаются. Если вы найдете ошибку — орфографическую или неработоспособный фрагмент кода — в одной из наших книг, мы будем глубоко благодарны за обратную связь. Присылая сообщения об ошибках, вы можете уберечь другого читателя от нескольких часов фрустрации.

Чтобы найти страницу с ошибками для этой книги, пройдите по ссылке **http://www.wrox.com/go/projavaeedesignpatterns**.

Затем выберите ссылку **Errata** (Ошибки). На этой странице вы найдете список всех ошибок, сообщения о которых были присланы нам и опубликованы редакторами издательства Wrox.

Если вы не нашли «вашу» ошибку на странице Book Errata, перейдите по ссылке http://www.wrox.com/contact/techsupport.shtml и заполните там форму для отправки нам сообщения об обнаруженной вами ошибке. Мы проверим информацию и, если она подтвердится, напишем сообщение на странице ошибок книги, а также исправим ошибку в следующих изданиях.

Как связаться с авторами

Если у вас появились какие-либо вопросы относительно содержания этой книги, кода или любых других родственных тем, вы можете связаться с авторами непосредственно в их блогах или через Twitter. Повторим эту информацию.

Мурат Йенер:

- блог — www.devchronicles.com;
- Twitter — @yenerm.

Алекс Фидом:

- блог — www.alextheedom.com;
- Twitter — @alextheedom.

ЧАСТЬ I

Введение в паттерны проектирования Java EE

1 Краткий обзор паттернов проектирования

В этой главе:

- обзор паттернов проектирования;
- краткая история паттернов проектирования и пояснение, чем они так важны;
- использование паттернов проектирования на практике;
- история и эволюция Java Enterprise Edition;
- появление корпоративных паттернов;
- как эти паттерны проектирования развиваются в корпоративном окружении;
- как и почему паттерны становятся антипаттернами.

Цель этой книги — наведение мостов между традиционной реализацией паттернов проектирования в среде Java SE и их реализацией в Java EE.

Если вы новичок в паттернах проектирования, эта книга поможет вам быстро войти в курс дела, так как каждая глава знакомит с паттерном проектирования в простой и понятной манере, с множеством примеров работающего кода.

Если же вы уже хорошо знакомы с паттернами проектирования и их реализацией, но не с их реализацией в среде Java EE, то эта книга идеальна для вас. Каждая глава соединяет традиционную реализацию и новую, зачастую более простую реализацию в Java EE.

Если вы эксперт в языке Java, это издание послужит вам надежным справочником по реализациям большинства распространенных паттернов проектирования на платформах Java SE и Java EE.

Эта книга сосредоточена на наиболее распространенных паттернах проектирования платформы Java EE и наглядно показывает, как они реализованы во вселенной Java EE. Каждая глава вводит новый паттерн, объясняя его суть и обсуждая приносимую им пользу. Затем в ней демонстрируется реализация этого паттерна в Java SE и дается детальное описание того, как он работает. В издании наглядно показывается, как паттерн в данный момент реализован в Java EE, и обсуждается наиболее распространенное его применение, его преимущества и подводные камни. Все объяснения сопровождаются подробными примерами кода, каждый из которых может быть скачан с прилагаемой к книге веб-страницы. В конце любой главы вы найдете заключительное обсуждение и краткое резюме, подытоживающее

все прочитанное вами ранее. А еще для вас там есть несколько интересных и иногда весьма непростых упражнений — для проверки понимания охваченных этой главой паттернов.

Что такое паттерн проектирования

> Паттерны проектирования — это «описания обменивающихся информацией объектов и классов, которые адаптированы для решения общей проблемы проектирования в конкретных условиях».
>
> *«Банда четырех»*

Паттерны проектирования предлагают решения распространенных проблем проектирования приложений. В объектно-ориентированном программировании паттерны проектирования обычно нацелены на решение задач, связанных с созданием и взаимодействием объектов, а не крупномасштабных задач, стоящих перед общей архитектурой программного обеспечения. Они обеспечивают обобщенные решения в виде шаблонов, которые могут быть применены к практическим задачам.

Обычно паттерны проектирования наглядно представляют в виде диаграмм классов, демонстрирующих поведение классов и отношения между ними. Типичная диаграмма классов показана на рис. 1.1.

Здесь продемонстрированы отношения наследования между тремя классами. Подклассы `CheckingAccount` и `SavingsAccount` наследуют от их абстрактного родительского класса `BankAccount`.

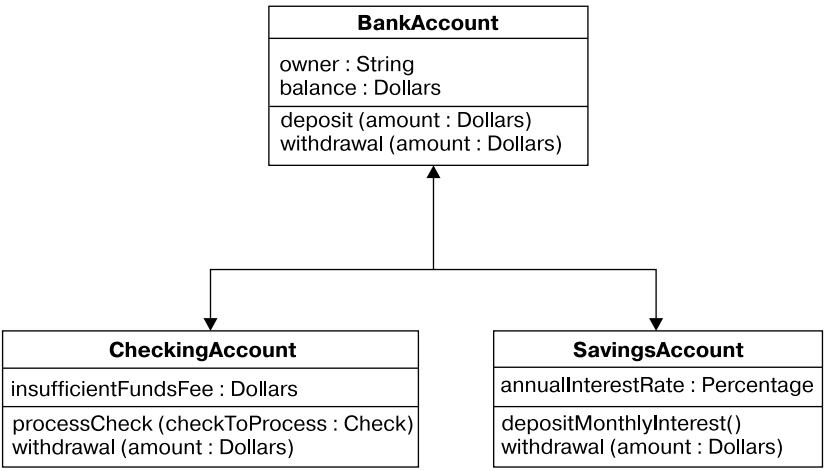


Рис. 1.1. Диаграмма классов, иллюстрирующая наследование

За такой диаграммой следует реализация на языке Java, демонстрирующая простейшую реализацию. Пример паттерна «Одиночка» (Singleton), который будет описан в дальнейших главах, показан на рис. 1.2.

Рис 1.2. Диаграмма классов паттерна «Одиночка»

А вот пример его простейшей реализации.

```
public enum MySingletonEnum {
    INSTANCE;
    public void doSomethingInteresting(){}
}
```

Как были изобретены паттерны проектирования и почему они нам нужны

Паттерны проектирования стали злободневным вопросом с тех пор, как знаменитая «Банда четырех» (GoF, состоящая из Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса) написала книгу «Приемы объектно-ориентированного проектирования. Паттерны проектирования», наконец-то предложив разработчикам со всего мира испытанные и проверенные решения для наиболее распространенных проблем программотехники. Эта важнейшая книга описывает различные методики разработки, их подводные камни и предоставляет 23 паттерна проектирования. Паттерны разделены на три категории: порождающие, структурные и паттерны поведения.

Но почему? Почему мы неожиданно осознали, что нам так нужны паттерны проектирования?

Решение не было таким уж внезапным. Объектно-ориентированное программирование появилось в 1980-х годах, и вскоре за ним последовали несколько основанных на этой новой идее языков программирования. Smalltalk, C++ и Objective C — часть из того небольшого количества объектно-ориентированных языков, которые до сих пор остались широко распространены. Впрочем, они принесли и свои собственные проблемы, и, в отличие от эволюции процедурного программирования, в этот раз перемены были слишком быстрыми, чтобы можно было увидеть, что было работоспособным, а что — нет.

Хотя паттерны проектирования решили много проблем (например, спагетти-код, то есть плохо структурированные программы), возникавших у специалистов по программному обеспечению с процедурными языками программирования, такими как C и COBOL, объектно-ориентированные языки внесли свой набор спорных вопросов. C++ быстро развивался и из-за своей сложности увел многих разработчиков на минные поля программных ошибок, таких как утечки памяти, неудачное проектирование объектов, небезопасное использование памяти и неудобный в сопровождении унаследованный код.

Однако большинство задач, с которыми сталкиваются разработчики, строятся по одним и тем же паттернам, и вполне разумно предположить, что кто-то когда-то уже нашел для них решение. В те времена, когда появилось объектно-ориентированное программирование, в доинтернетовском мире, было непросто обмениваться опытом с широкими массами народа. Поэтому прошло некоторое время, прежде чем GoF создали набор паттернов для известных повторяющихся проблем.

Паттерны в реальном мире

Паттерны проектирования — бесконечно полезные и проверенные решения для задач, с которыми вы неизбежно встретитесь. Они не только передают многолетние совокупные знания и опыт, но и предлагают полезный справочник для общения между разработчиками и проливают свет на многие проблемы.

Тем не менее паттерны проектирования не волшебная палочка, они не предоставляют, подобно фреймворкам или наборам утилит, готовой для использования реализации. Излишнее — только потому, что это модно, или потому, что вы хотите произвести впечатление на начальника, — использование паттернов проектирования может вылиться в переусложненную систему, которая не решит никаких задач, но взамен продемонстрирует наличие ошибок, неумелый дизайн, низкую производительность и проблемы с сопровождением. Большинство паттернов может решить проблемы дизайна, обеспечить надежные решения известных задач и позволит разработчикам общаться общими идиомами через пропасть между разными языками. Паттерны, вообще говоря, следует использовать лишь тогда, когда существует вероятность возникновения проблем.

Паттерны проектирования были изначально разделены на три группы.

- **Порождающие паттерны** управляют созданием и инициализацией объекта, а также выбором класса. Примерами паттернов из этой группы могут быть «Одиночка» (см. главу 4) и «Фабрика» (см. главу 6).
- **Паттерны поведения** управляют связью, обменом сообщениями и взаимодействием между объектами. Примером паттерна из этой группы может быть «Наблюдатель» (см. главу 11).
- **Структурные паттерны** упорядочивают отношения между классами и объектами, обеспечивая критерии соединения и совместного использования связанных объектов для достижения желаемого поведения. Хороший пример паттерна из этой группы — «Декоратор» (см. главу 7).

Паттерны проектирования предлагают разработчикам своеобразный общий справочник. Разработчики могут использовать их, чтобы общаться гораздо проще, не изобретая колесо для каждой задачи. Хотите показать вашему приятелю, как вы собираетесь добавлять динамическое поведение во время исполнения? Довольно пошаговых рисунков и недоразумений! Все просто и ясно: вам достаточно произнести несколько слов: «Давай использовать для этой задачи паттерн “Декоратор”!» Ваш друг тотчас же поймет, о чем вы говорите, не нужны никакие дополнительные разъяснения. Если вы уже знаете, что такое паттерн, и используете его в правильном контексте, вы явно встали на путь создания надежного и легкосопровождаемого приложения.

РЕКОМЕНДУЕМОЕ ЧТЕНИЕ

Мы настоятельно рекомендуем вам прочитать «Приемы объектно-ориентированного проектирования. Паттерны проектирования» Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса или «Паттерны проектирования» Эрика Фримена, Элизабет Фримен, Кэтти Сьерра и Берта Бейтса[1] (Head First Design Patterns). Обе книги — прекрасные дополнения к той, которую вы сейчас читаете, и бесценные руководства по изучению паттернов проектирования.

Основы паттернов проектирования

Одним из ключевых нюансов, относящихся к паттернам проектирования, является то, что злоупотребление ими или излишнее их использование может стать источником проблем. Как только некоторые разработчики выучивают новые паттерны, им страшно хочется использовать их где только можно. Однако подобное поведение часто приводит к тому, что их проекты разбухают от одиночек или заключены в оболочку фасадов или неоправданно сложных декораторов. Паттерны проектирования — ответ на все вопросы, так что, если нет проблемы или хотя бы шанса, что проблема появится, реализовывать паттерн смысла нет. Для примера: использование паттерна «Декоратор» только из-за незначительной вероятности будущего изменения поведения объекта приносит сложность разработки сейчас и кошмар при сопровождении в будущем.

Корпоративные паттерны

Язык Java 1.0 быстро стал популярным после своего выхода в начале 1996 года. Выбор момента был идеален для представления нового языка, который бы устранил сложность управления памятью, синтаксиса и работы с указателями языков C/C++. Java предлагал постепенную кривую обучения, позволявшую многим разработчикам быстро его усвоить и начать программировать на нем. Однако было еще нечто, ускорившее перемены, — апплеты. Апплет — это маленькое приложение, запускаемое на сайте в отдельном процессе браузера и добавляющее сайту функциональность, невозможную при использовании только HTML и CSS. Примером может служить интерактивный график или передача потокового видео.

С быстрым ростом Интернета статические веб-страницы вскоре стали устаревшими и скучными. Пользователям хотелось получать от веб-серфинга больше информации и делать это быстрее. И в этот момент появились апплеты, которые предлагали невероятные интерактивность, эффекты и экшен для статичной тогда Всемирной паутины.

Вскоре танцующий Дюк (символ языка Java) стал общей тенденцией для современных сайтов. Однако в Интернете ничего не остается неизменным надолго. Пользователи хотели большего, а апплеты потерпели полный крах в попытках приспособиться к этим требованиям и не сумели сохранить свою популярность.

[1] *Фримен Э., Фримен Э., Сьерра К., Бейтс Б.* Паттерны проектирования. — СПб.: Питер, 2011. — 656 с.

Тем не менее апплеты были той движущей силой, которая стояла за быстрым приспособлением и популярностью платформы Java. Сегодня (на момент написания этой книги) Java все еще один из двух самых популярных языков программирования в мире: согласно индексу TIOBE, Java находится на втором месте по популярности после C[1] (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html).

От языка Java к корпоративной платформе Java

Вслед за выпуском стандартной версии Java (Java Standard Edition) IBM представила в 1997 году корпоративную модель Java-компонентов (Enterprise JavaBeans, EJB), которая была в 1999 году принята Sun и составила часть корпоративной платформы Java (Enterprise Java Platform, J2EE) версии 1.2. В 1998 году, перед выпуском J2EE[2], Sun выпустила профессиональную версию Java, получившую название JPE. Однако только после выхода EJB поставщики и разработчики заинтересовались возможностью использования корпоративной платформы Java. А с выходом J2EE 1.3 в 2001 году Java становится ключевым игроком мира корпоративных приложений, еще более закрепив свои позиции выпуском J2EE 1.4 в 2003 году.

Версия 1.4 была одной из важнейших вех в истории Java. Она широко применялась и удерживала свою популярность долгие годы, несмотря на выпуск новых версий. Поставщики и корпорации не торопились переходить на новые версии, хотя причины жаловаться на J2EE 1.4 были у многих. Использование ее можно было сравнить с поездкой по магазинам на монстр-траке вместо семейного седана. Несомненно, она была производительной, но попросту слишком сложной и слишком раздутой от XML-файлов, и как фреймворки, так и контейнеры были весьма тяжеловесными.

Все же J2EE стала наиболее популярной корпоративной платформой разработки. У нее был набор возможностей, делавший ее отличным выбором для корпоративной разработки.

- **Переносимость** — JVM позволяла Java-коду запускаться в любой операционной системе. Программисты могли вести разработку в Windows, тестировать в Linux, а вводить в промышленную эксплуатацию в UNIX-системе.
- **Безопасность** — J2EE предлагала свою собственную ролевую модель безопасности.
- **Транзакции** — J2EE предлагала встроенные транзакции.
- **Возможность языка платформы J2SE** — J2SE предлагала простой синтаксис, сборку мусора и великолепные объектно-ориентированные возможности программирования.

---

[1] Согласно вышедшему в июне 2015 года очередному ежегодному выпуску индекса TIOBE, Java укрепил свои позиции и потеснил C с первого места. — *Примеч. пер.*

[2] До версии 5 Java EE обычно называлась J2EE. С этого места мы будем использовать термин J2EE для версий, предшествовавших Java EE 5.

Однако J2EE была несовершенной. Достаточно скоро сложное устройство платформы с ее интенсивным использованием XML-конфигураций создало идеальную среду для возникновения проблем.

Появление корпоративных паттернов Java

Сложные модели программирования J2EE вскоре поставили многие проекты в затруднительное положение. Созданные с помощью J2EE-технологий приложения имели склонность содержать непомерное количество «канализационного» кода вроде кода поиска JNDI[1], файлов XML-конфигураций и блоков `try/catch`, запрашивающих и освобождающих ресурсы JDBC[2]. Написание и обслуживание такого кода на практике становилось причиной серьезного расхода ресурсов и было источником множества программных ошибок и проблем с производительностью. Компонентная модель EJB была призвана снизить сложность при реализации бизнес-логики, но не преуспела в этом. Модель оказалась слишком сложной и зачастую используемой чрезмерно.

Спустя всего несколько лет после первого выпуска платформы Дипак Алур, Джон Друки и Дэн Малкс выступили на конференции JavaOne в 2000 году с докладом «Макетирующие паттерны для платформы J2EE», в котором представили несколько паттернов, нацеленных на решение распространенных проблем, встречающихся при дизайне приложений под J2EE. Этот доклад стал книгой. На следующий год они опубликовали книгу под названием *Core J2EE Patterns: Best Practices and Design Strategies*[3]. В дополнение к 15 уже широко известным паттернам авторы представили еще шесть новых паттернов проектирования для J2EE. Новые паттерны включали Context Object и Application Controller для слоя представления, Application Service и Business Object — для слоя бизнес-логики, Domain Store и Web Service Broker — для слоя интеграции.

Некоторые из этих паттернов развились из «классических» паттернов GoF, в то время как остальные были новыми, направленными на борьбу с изъянами J2EE. В последующие годы было выпущено несколько проектов и фреймворков, таких как Apache Camel, облегчающих жизнь корпоративных разработчиков. Некоторые даже под руководством Рода Джонсона[4] поступили смелее, отойдя от J2EE и выпустив фреймворк Spring. Spring быстро обрел популярность, что оказало свое влияние на масштабные перемены в новой модели программирования в Java EE. На сегодняшний день большинство из этих паттернов используются и остаются вполне пригодными. Впрочем, некоторые устарели и более не нужны благодаря упрощенной модели программирования Java EE.

---

1 Java Naming and Directory Interface — стандартный программный интерфейс к корпоративной службе каталогов. — *Примеч. пер.*

2 Java Database Connectivity — интерфейс Java для доступа к базам данных. — *Примеч. пер.*

3 *Алур Дипак, Крупи Джон, Малкс Дэн.* Образцы J2EE. Лучшие решения и стратегии проектирования. — М.: Лори, 2013. — 375 с.

4 Род Джонсон (@springrod) — известный австралийский специалист по компьютерной технике, создавший фреймворк Spring, сооснователь компании SpringSource. http://en.wikipedia.org/wiki/Rod_Johnson_(programmer).

Отличия паттернов проектирования от корпоративных паттернов

Корпоративные паттерны отличаются от паттернов проектирования тем, что первые предназначены для корпоративного ПО с его проблемами, которые существенно отличаются от проблем приложений для настольных систем. Новый подход, сервис-ориентированная архитектура (SOA, Service Oriented Architecture) ввели несколько принципов, которых следует придерживаться при построении хорошо организованного, пригодного для многократного использования корпоративного ПО. Основу этих фундаментальных принципов составили «Четыре догмата SOA» Дона Бокса[1]. Этот набор принципов обращен на распространенные потребности корпоративных проектов.

ЧЕТЫРЕ ДОГМАТА SOA ДОНА БОКСА

1. Границы — явные. 2. Сервисы — автономные. 3. Сервисы совместно используют схему и контракт, но не класс. 4. Совместимость сервисов определяется на основании политики.

Однако «классические» паттерны все еще имеют что предложить. С выходом платформы Java EE 5 корпоративная Java снова оказалась в центре внимания, что слишком долго было прерогативой сторонних фреймворков, таких как Spring и Struts. Выход Java EE 6 стал еще большим шагом вперед и сделал платформу еще более конкурентоспособной.

Сегодня — в Java EE 7 — большинство «классических» паттернов проектирования, описанных в книге GoF, встроены в платформу и готовы для немедленного использования. В отличие от эпохи J2EE, большинство этих паттернов могут быть реализованы посредством аннотаций, не требуя запутанных XML-конфигураций. Это стало огромным скачком вперед и предоставило разработчику существенно упрощенную модель программирования.

Хотя и существует несколько великолепных книг по паттернам проектирования и новым возможностям платформы Java EE, похоже, что недостающее звено — то, как эти паттерны реализованы в Java EE.

Простые паттерны проектирования в старом стиле встречаются с Java EE

Платформа Java с самого начала благоприятствовала паттернам проектирования. Для некоторых паттернов, таких как паттерн «Наблюдатель» в Java SE, существует готовая к использованию встроенная реализация. Сама по себе Java тоже использует многие паттерны проектирования, например, паттерн «Одиночка» применяется в системных и исполняемых классах, компараторы — отличный пример реализации паттерна «Стратегия».

Эта традиция продолжилась в корпоративной Java, особенно в Java EE, имеющей встроенные реализации многих из описанных в книге GoF паттернов. Большая часть этих паттернов может быть подключена и применена с простыми и легкими

[1] Дон Бокс — выдающийся инженер. http://en.wikipedia.org/wiki/Don_Box.

в использовании аннотациями. Так что вместо разглядывания диаграмм классов и написания шаблонного кода любой опытный разработчик может подключить паттерн всего несколькими строками кода. Магия? Не совсем. За счет более сложной конструкции среда выполнения Java EE способна предложить множество возможностей, основываясь на мощи базовой платформы. Большая часть необходимых для этих паттернов возможностей не была бы доступна без расширенного набора возможностей платформы Java EE, таких как EJB и CDI (контекст и внедрение зависимостей, Context and Dependency Injection). Контейнер Java EE выполняет для вас большую часть работы, добавляя на сервер многие встроенные сервисы и функциональность. Недостаток в том, что это приводит к тяжеловесной среде выполнения сервера, особенно в сравнении с такими базовыми веб-серверами, как Apache Tomcat. Однако сейчас это усовершенствовали и последние динамические сборки на Java EE 7 стали более «легкими».

И все же почему людям по-прежнему нужны паттерны проектирования в корпоративных приложениях? Необходимость в паттернах сейчас больше, чем когда-либо прежде. Большинство из корпоративных приложений создаются для корпораций различными командами разработчиков, и часто возникает потребность в повторном использовании разных частей. В отличие от случая решения проблемы паттерна своими силами или небольшой командой, здесь ваши решения видны всей корпорации и сверх того, возможно, что и всему миру (если ваш проект — с открытым исходным кодом). Совсем несложно опубликовать плохо спроектированное приложение и позволить этому стать корпоративной традицией или стратегией разработки. Поскольку библиотеки, утилитные классы и различные API доступны для использования большему количеству разработчиков, становится все сложнее разрушать совместимость и производить радикальные перемены. Изменение одного-единственного типа возвращаемого значения или даже добавление к интерфейсу нового метода может нарушить функционирование всех программ, зависящих от этого фрагмента кода.

Очевидно, что разработка корпоративного ПО требует более высокого уровня дисциплины и координации между группами разработчиков. Паттерны проектирования — хороший метод для решения этой проблемы. Однако большинство разработчиков корпоративных приложений до сих пор не научились извлекать пользу из классических паттернов проектирования, хотя они есть в Java EE еще с версии 5.0. Хотя корпоративные паттерны могут решить многие задачи, первоначальные паттерны проектирования по-прежнему могут немало предложить. Это хорошо отработанные и проверенные решения, выдержавшие испытание временем, и они реализованы почти во всех объектно-ориентированных языках программирования.

И наконец, поскольку большинство из этих паттернов уже интегрировано в платформу Java EE, нет необходимости в написании полного кода их реализации. Для отдельных из них может потребоваться небольшая XML-конфигурация, но большая часть может быть реализована применением аннотации к классу, методу или переменной экземпляра. Хотите создать «Одиночку»? Просто добавьте аннотацию `@Singleton` в начале вашего файла класса. Необходимо реализовать паттерн «Фабрика»? Просто добавьте аннотацию `@Produces`, и метод превратится в фабрику с требуемым типом возвращаемого значения.

Java EE еще и устанавливает стандарты. Аннотация `@Inject` служит реализацией по умолчанию и может комбинироваться и сочетаться практически с любым другим фреймворком (например, с фреймворком Spring), поскольку они используют одну и ту же аннотацию.

В каких случаях паттерны становятся антипаттернами

Паттерны проектирования можно назвать собранной воедино мудростью, но это не значит, что их нужно использовать постоянно. Как весьма уместно заметил известный американский психолог Абрахам Маслоу[1]: «Если единственный имеющийся у вас инструмент — молоток, вы склонны считать любую проблему гвоздем». Если вы попытаетесь использовать для всех проблем только те паттерны, которые вам известны, они попросту не подойдут или, что еще хуже, подойдут плохо и станут причиной еще больших проблем. Более того, излишнее использование паттернов имеет тенденцию переусложнять систему и приводит к низкой производительности. Не нужно применять паттерн «Декоратор» к каждому объекту только потому, что он вам нравится. Паттерны работают лучше всего в тех условиях и для тех задач, которые действительно требуют их использования.

Резюме

Java и паттерны проектирования прошли долгий путь до нынешнего их положения. Некогда они были разобщены и ничего не знали друг о друге, но сейчас они вместе, чтобы навечно быть интегрированными в корпоративную версию Java. Чтобы лучше понять этот тесный союз, вы должны знать их историю. Вы уже открыли истоки этой пары и то, как они нашли друг друга. Вы прочли о шатких первых шагах J2EE и том, как GoF пролили свет на 23 паттерна проектирования. Вы увидели, как подобные Spring фреймворки выросли вслед за Java и одержали над ней верх и как обновленная платформа Java EE сейчас наносит ответный удар и переходит в наступление. Знания, содержащиеся в этой книге, дадут вам подготовку, необходимую для уверенного решения большинства проблем проектирования, с которыми вы столкнетесь в своей карьере разработчика. Вы можете спать спокойно, зная, что перенесенные корпоративной версией Java годы борьбы в сочетании со свойственной паттернам проектирования мудростью привели к появлению долговечных и гибких языка и среды программирования.

Желаем вам с удовольствием работать с этим бесценным руководством для разработки паттернов проектирования в Java EE и использовать почерпнутую здесь мудрость в каждом своем проекте.

---

[1] Абрахам Маслоу (1908–1970) — американский психолог.

2 Основы Java EE

В этой главе:

- введение в базовые концепции платформы Java EE;
- обсуждение многоуровневой структуры корпоративного приложения;
- описание Java-EE совместимых серверов и веб-профиля;
- обзор соглашений по конфигурации;
- обзор CDI;
- обзор перехватчиков.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода вы найдете в файле Chapter 02.zip, каждый из них назван в соответствии с наименованиями в тексте.

Модель программирования платформы Java EE со времен J2EE была существенно упрощена. Аннотации заменили XML-файлы дескрипторов, соглашения по конфигурации заменили утомительную ручную конфигурацию, а внедрение зависимостей скрывает создание и поиск ресурсов. Соответственно, разработчики тоже должны пересмотреть подход к проектированию и программированию.

Разработка корпоративных приложений на платформе Java EE стала проще. Все, что вам нужно, — стандартный Java-объект в старом стиле (POJO), аннотированный некоторыми метаданными, и в зависимости от использованной аннотации POJO превращается в компонент Enterprise JavaBeans (EJB, с сохранением состояния или без), в сервлет, в управляемый компонент JSF, в постоянную сущность, в одиночку или в веб-сервис REST[1]. При желании можно объявить большинство этих сервисов, используя XML в дескрипторе развертывания.

Листинг 2.1 демонстрирует, как сделать из POJO компонент-одиночку, экземпляр которого создается и инициализируется при запуске, а затем управляется контейнером, простым добавлением аннотаций `@Singleton` и `@Startup` к классу и `@PostConstruct` к методу инициализации. Для подробного объяснения использования этих аннотаций см. главу 4.

---

[1] Representational State Transfer — репрезентативная передача состояния. — *Примеч. пер.*

Листинг 2.1. С добавлением некоторых аннотаций POJO становится управляемым контейнером компонентом-одиночкой

```
package com.devchronicles.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Startup
@Singleton
public class CacheSingletonBean {

    private Map<Integer, String> myCache;

    @PostConstruct
    public void start(){
        myCache = new HashMap<Integer, String>();
    }

    public void addUser(Integer id, String name){
        myCache.put(id, name);
    }

    public String getName(Integer id){
        return myCache.get(id);
    }
}
```

Цели Java EE не изменились; она по-прежнему признает требования, выдвигаемые разработчиками и корпорациями к распределенным и транзакционным приложениям, требующим быстроты, безопасности и безотказности в работе. Платформа Java EE разработана, чтобы сделать производство крупномасштабных многозвенных приложений легче, надежнее и безопаснее.

Многоуровневая архитектура

Архитектура приложений для платформы Java EE разделена на уровни: уровень клиента, промежуточный уровень (состоящий из веб-слоя и бизнес-слоя) и уровень корпоративной информационной системы (Enterprise Information System, EIS). Каждый из них имеет свои обязанности и использует различные технологии Java EE. Расщепление приложения на четко выраженные уровни придает ему большую гибкость и адаптируемость. Вы получаете альтернативу рефакторингу всего приложения: возможность добавить или изменить только конкретный уровень. Все уровни разделены физически и расположены на разных машинах. А в случае веб-приложения уровень клиента распределен по всему миру.

Java EE функционирует внутри промежуточного уровня, хотя соприкасается как с уровнем клиента, так и с уровнем EIS. Промежуточный уровень получает запросы от приложения уровня клиента. Веб-слой промежуточного уровня обрабатывает запрос и формирует отклик, который отправляет обратно уровню клиента, в то время как бизнес-слой применяет бизнес-логику перед сохранением его на уровне EIS. А пока происходит подготовка ответа уровню клиента, в пределах промежуточного уровня ведется оживленное общение между его слоями и уровнем EIS. Многоуровневая архитектура может быть представлена графически, как показано на рис. 2.1.

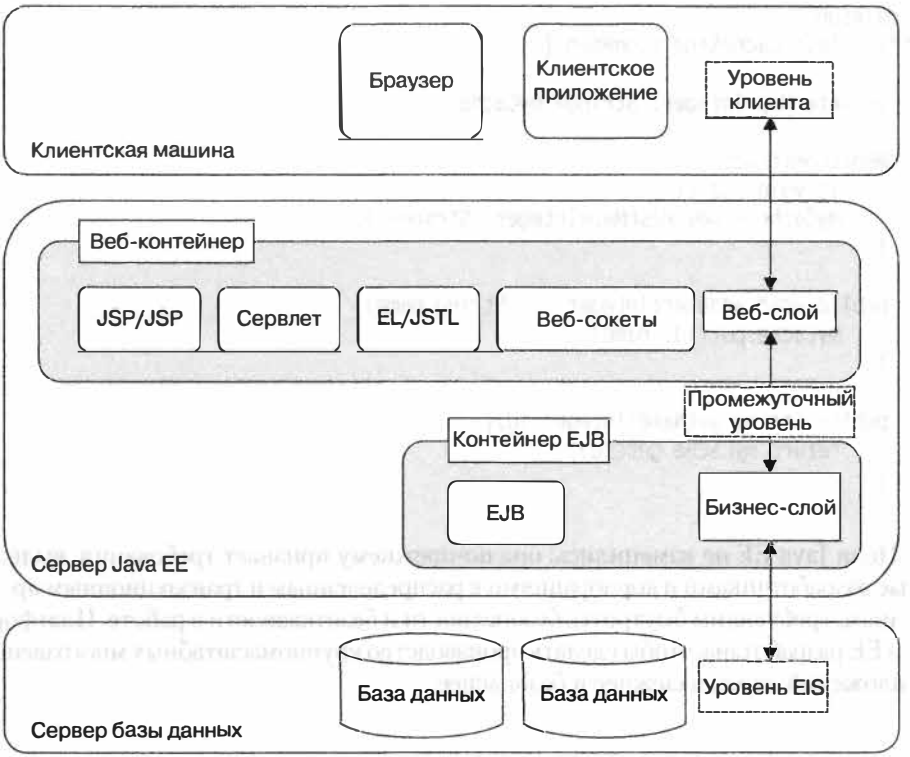


Рис. 2.1. Многоуровневая архитектура. Демонстрация взаимодействия между уровнями

Уровень клиента

Уровень клиента обычно представляет собой браузер, подключающийся к серверу Java EE посредством протокола передачи гипертекста (Hypertext Transfer Protocol, HTTP), хотя им может быть любое приложение на любой машине при условии, что в связке «клиент — сервер» оно ведет себя как клиент. Клиентское приложение посылает серверу запрос на получение ресурса; сервер обрабатывает запрос и возвращает отклик. Таковы обычные пределы взаимоотношений между клиентом и сервером.

ПРИМЕЧАНИЕ

Уровень клиента часто называют уровнем представления.

Промежуточный уровень

Сервер Java EE размещается в промежуточном уровне и предоставляет два логических контейнера: веб-контейнер и контейнер EJB. Эти контейнеры приблизительно соответствуют веб-слою и бизнес-слою соответственно. Каждый слой имеет четко определенные, но иногда пересекающиеся обязанности.

Шаблон MVC обычно используется, чтобы четко отделить обязанности веб-слоя по генерации представления от обязанностей бизнес-слоя по моделированию данных. В главе 14 детально обсуждается, как реализовать такое разделение сфер интересов.

Веб-слой

Веб-слой управляет обменом данными между уровнем клиента и бизнес-слоем.

Веб-слой принимает запрос на получение ресурса от уровня клиента. Запрос может включать данные, которые ввел пользователь, такие как логин и пароль, или регистрационную информацию. Запрос обрабатывается, и, если необходимо, происходит обмен данными между веб-слоем и бизнес-слоем. Отклик динамически создается в одной из нескольких форм (для запроса, инициированного браузером, обычно в форме веб-страницы на языке HTML) и отправляется клиенту.

Веб-слой поддерживает состояние пользователя во время сессии и может даже выполнять некоторую бизнес-логику и временно сохранять данные в памяти.

Технологии, обычно используемые в веб-слое, относятся к управлению обменом данными между уровнем клиента и промежуточным уровнем и формированием отклика. Сервлеты контролируют потоки веб-данных и управляют обменом данными, в то время как JavaServer Pages (JSP), язык выражений (Expression Language, EL) и стандартная библиотека тегов JSP (JavaServer Pages Standard Tag Library, JSTL) готовят отклик для клиента. Такова краткая характеристика технологий, которые вы можете использовать в веб-слое. Полный список приведен в табл. 2.1.

В Java EE 7 четыре новые технологии были добавлены к вселенной EE: веб-сокеты (WebSockets), утилиты параллелизма (Concurrency Utilities), пакеты (Batch) и JSON-P. В обоих слоях вы можете использовать все, кроме веб-сокетов.

Бизнес-слой

Бизнес-слой выполняет бизнес-логику, решая задачи или удовлетворяя конкретные потребности в коммерческих проектах. Обычно при этом задействуются данные, извлекаемые из базы данных, расположенной в уровне EIS или собираемые на клиенте. В банковской сфере к сумме транзакции может быть применен транзакционный сбор, а данные отправлены клиенту через веб-слой для подтверждения транзакции. В сфере электронной коммерции до передачи веб-слою к товару могут

быть применены различные ставки налогов в зависимости от физического месторасположения клиента, а веб-страница — сформирована в соответствии с этой информацией.

Бизнес-слой — место, где сосредоточена основная логика бизнес-приложения. Бизнес-логика обернута в EJB, и данные, используемые бизнес-логикой, извлекаются из уровня EIS через Java Persistence API (JPA), Java Transaction API (JTA) и Java Database Connectivity (JDBC).

Распространенной практикой является запрос и изменение данных через веб-сервисы, использующие JAX-RS и JAX-WS (см. главу 13 для получения более подробной информации по этому вопросу). Такова краткая характеристика технологий, которые вы можете использовать в бизнес-слое. Полный список приведен в табл. 2.1.

Таблица 2.1. Технологии, используемые в веб- и бизнес-слое

Полные требования к продукту[1]	Стандарт JSR	Дополнительно	Веб-профиль[2]	Новое в Java EE 7	Веб-контейнер	EJB-контейнер
Java API for WebSocket	JSR 356					
Java API for JSON Processing	JSR 353					
Java Servlet 3.1	JSR 340					
JavaServer Faces 2.2	JSR 344					
Expression Language 3.0	JSR 341					
JavaServer Pages 2.3	JSR 245					
Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52					
Batch Applications for the Java Platform	JSR 352					
Concurrency Utilities for Java EE 1.0	JSR 236					
Contexts and Dependency Injection for Java 1.1	JSR 346					
Dependency Injection for Java 1.0	JSR 330					
Bean Validation 1.1	JSR 349					
Enterprise JavaBeans 3.2 (за исключением сущностей-компонентов EJB и ассоциированного с ними языка запросов EJB QL, которые были сделаны поставляемыми по запросу)	JSR 345		1		1	

[1] Источник: корпоративная платформа Java (Java EE 7), JSR 342, EE.9.7. «Полные требования к продукту Java EE».

[2] Источник: корпоративная платформа Java (Java EE 7), спецификации веб-профиля, JSR 342, WP.2.1 «Необходимые компоненты».

Полные требования к продукту	Стандарт JSR	Дополнительно	Веб-профиль	Новое в Java EE 7	Веб-контейнер	EJB-контейнер
Managed Beans 1.0	JSR-316					
Interceptors 1.2	JSR 318					
Java EE Connector Architecture 1.7	JSR 322					
Java Persistence 2.1	JSR 338					
Common Annotations for the Java Platform 1.2	JSR 250					
Java Message Service API 2.0	JSR 343					
Java Transaction API (JTA) 1.2	JSR 907					
JavaMail 1.5	JSR 919					
Java API for RESTful Web Services (JAX-RS) 2.0	JSR 339					
Implementing Enterprise Web Services 1.4	JSR 109					
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224					
Web Services Metadata for the Java Platform	JSR 181					
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101					
Java API for XML Registries (JAXR) 1.0	JSR 93					
Java Authentication Service Provider Interface for Containers 1.1	JSR 196					
Java Authorization Contract for Containers 1.5	JSR 115					
Java EE Application Deployment 1.2	JSR 88					
J2EE Management 1.1	JSR 77					
Debugging Support for Other Languages 1.0	JSR 45					
Java Architecture for XML Binding (JAXB) 2.2	JSR 222					

ПРИМЕЧАНИЕ

Промежуточный уровень часто называют также логическим уровнем, уровнем доступа к данным и уровнем приложений.

Уровень EIS

Уровень EIS состоит из блоков хранения данных, часто в виде баз данных, впрочем, они могут быть любым ресурсом, который предоставляет данные, например старомодной системой или файловой системой.

ПРИМЕЧАНИЕ

Уровень EIS часто называют также уровнем данных, уровнем сохраняемости и уровнем интеграции.

Серверы Java EE

Как вы видели, промежуточный уровень служит для размещения сервера Java EE, который обеспечивает функциональность Java EE, необходимую для корпоративных приложений.

Java EE основана на 30 стандартах, именуемых *запросами на спецификацию Java* (Java Specification Requests, JSRs): http://www.oracle.com/technetwork/java/javaee/tech/index.html.

Прежде чем стать частью вселенной Java, эти запросы проходят через процесс обсуждения в Java-сообществе (Java Community Process, JCP). JCP — открытый процесс, в котором любой желающий может не только принимать участие, внося замечания и предложения относительно различных JSR, но и представлять на рассмотрение свои собственные JSR (https://www.jcp.org/en/home/index).

Эти спецификации, собранные воедино, описывают технологии, которые серверное приложение должно реализовать, чтобы иметь право заявить о своей Java EE-совместимости.

Помимо этого, корпорация Oracle требует, чтобы серверное приложение прошло через «Комплект технологической совместимости» (Technology Compatibility Kit, TCK) — нетривиальный набор тестов, проверяющих, что приложение ведет себя так, как требует спецификация. Таким образом гарантируется, что, если вы разработали ваше приложение, следуя спецификациям Java EE, вы сможете установить и выполнить его на любой машине с Java EE.

На момент написания этой книги три сервера приложений были сертифицированы как полностью совместимые с Java EE 7. Это GlassFish Server Open Source Edition 4.0 (http://glassfish.java.net), Wildfly 8.0.0 (http://wildfly.org) и TMAX JEUS 8 (http://tmaxsoft.com/product/jeus/certification/)[1]; 11 серверов приложений являются совместимыми с Java EE 6 (http://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition#Java_EE_6_certified).

Веб-профиль Java EE

Веб-профиль Java EE — это подмножество технологий, включающее наиболее подходящие из них для разработки основанных на веб-технологиях корпоративных приложений. Профиль сводит размер и сложность платформы только к тем технологиям, которые необходимы для разработки современного веб-приложения. Веб-профиль — это полный комплект, включающий технологии, связанные с потоковой обработкой и базовой функциональностью (Servlet), представлением (JSF и JSP),

[1] По состоянию на июнь 2015 года к ним добавился еще один сервер приложений: Cosminexus: Hitachi Application Server v10.0 (http://www.hitachi.com/products/it/software/prod/cosminexus/products/apserver/feature.html). — *Примеч. пер.*

бизнес-логикой (EJB Lite), транзакциями (JTA), сохраняемостью (JPA), новым WebSocket, и многое другое. Он не включает множество технологий, относящихся к корпоративным, например утилиты параллелизма (Concurrency Utilities), сервисы передачи сообщений (Java Message Services), JAX-RPC, JAXR и JAX-WS. Для полной сводки технологий, включенных в веб-профиль, см. табл. 2.1.

Базовые принципы платформы Java EE

Базовые принципы Java EE включают несколько парадигм проектирования и паттернов, существенных для того способа, которым вы разрабатываете корпоративные приложения. Одной из центральных для Java EE является парадигма проектирования о соглашениях по конфигурации: способ упростить разработку корпоративных приложений без потери гибкости и без скрытия предназначения кода. Эта идея ненова, в течение некоторого времени — в иных случаях почти десятка лет — она была частью других фреймворков, включая Grails, Ruby on Rails и Spring Boot.

Платформа Java EE удачно пользуется своей компонентной моделью, которая включает сущности (Entities), JavaBeans, EJB-компоненты, управляемые компоненты (Managed Beans), сервлеты, SOAP и веб-сервисы RESTful. Все эти компоненты могут быть внедряемыми зависимостями. Контейнер в некотором смысле управляет их жизненным циклом (от создания экземпляра до уничтожения) — ограничены они контекстом или нет — и разъединением их с зависимыми компонентами путем внедрения зависимости.

Слабосвязанные приложения допускают расширяемость: старые классы могут быть заменены новыми без необходимости изменения зависимых классов. Внедрение зависимости разъединяет объект с его зависимостями, тогда как перехватчики разделяют коммерческую функциональность с технической и сквозной функциональностями. Такой технологической функциональностью могли бы быть журналирование и код, обеспечивающий производительность, а сквозной функциональностью — код, реализующий защиту.

Соглашения по конфигурации

В рамках соглашений все имена классов должны начинаться с прописной буквы. Хотя это не обязательно: класс будет компилироваться и с именем, начинающимся со строчной буквы, однако прописная буква в начале имени класса делает код более легким для понимания и поддержки. При создании проекта в IDE необходимо только указать тип проекта и его имя для формирования наиболее подходящей структуры каталогов, импорта самых распространенных API и создания файлов по умолчанию для облегчения разработки. Все это делается на основе принятых соглашений.

Количество работы, которую вы должны выполнить, и решений, которые нужно вам принять как разработчику, существенно уменьшается, когда вы полагаетесь на соглашения. Нет необходимости задавать подпадающую под соглашения конфигурацию, нужно только задать неподпадающую. Это дает значительный

эффект. Используя всего лишь несколько аннотаций на POJO, вы можете покончить с массой безобразных XML-дескрипторов развертывания и конфигурационных файлов приложения. Как вы видели в листинге 2.1, нужно применить всего три аннотации, чтобы преобразовать POJO в компонент-*одиночку*, экземпляр которого будет создан и инициализирован при запуске, после чего будет управляться контейнером.

ПРИМЕЧАНИЕ

Соглашения по конфигурации также называют программированием по соглашениям.

Контекст и внедрение зависимостей

Внедрение зависимости (Dependency Injection) — паттерн проектирования (см. главу 5), который разъединяет связь компонента и его зависимостей. Это производится путем внедрения зависимости в объект вместо создания зависимости объектом с помощью ключевого слова `new`. Забирая у объекта возможность создания зависимости и передавая эту обязанность контейнеру, вы можете отдать зависимость другому совместимому объекту как во время компиляции, так и во время выполнения.

Компоненты, которыми управляет контейнер, называются CDI-управляемыми (CDI — Context and Dependency Injection) компонентами, их экземпляры создаются в момент запуска контейнера. Все POJO, имеющие конструктор по умолчанию и не созданные с использованием ключевого слова `new`, — CDI-компоненты, внедренные в объект на основании соответствия типов. Принимающий объект, для того чтобы в него была выполнена инъекция, должен объявить поле, конструктор или метод, используя аннотацию `@Inject`. Впоследствии тип объявленного объекта используется для выбора внедряемой зависимости.

В листинге 2.2 вы можете видеть POJO, который будет управляться как CDI-компонент, так как у него есть конструктор по умолчанию; а в листинге 2.3 осуществляется внедрение управляемого компонента. На основании его типа контейнер знает, что должен внедрить компонент `Message`. Контейнер управляет только одним CDI-компонентом типа `Message`, так что это и есть внедряемый компонент.

Листинг 2.2. Пример внедрения зависимости. Зависимость

```
package com.devchronicles.basicsofjavaee;

public class Message {

    public String getMessage(){
    return "Hello World!!";
    }

}
```

Листинг 2.3. Пример внедрения зависимости. Получатель

```
package com.devchronicles.basicsofjavaee;

import javax.inject.Inject;
```

```
public class Service {

@Inject
private Message message;

    public void showMessage(){
    System.out.println(message.getMessage());
    }
}
```

Дотошный человек мог бы спросить: что случится, если контейнер станет управлять несколькими компонентами типа Message? Чтобы такое произошло, Message должен иметь интерфейс с несколькими реализациями. Вот тут все становится интереснее. Существует несколько методик разрешения неоднозначностей такого типа. С некоторыми из них вы познакомитесь во время чтения этой книги. Если же вы не можете преодолеть любопытство, перейдите сразу к главе 5.

Контекст — отличительная черта CDI-управляемых компонентов от EJB-компонентов. CDI-компоненты существуют в рамках определенного контекста, EJB-компоненты — нет. CDI-компоненты создаются в контексте области видимости, они существуют на время жизни области видимости и уничтожаются в ее конце. Существует четыре области видимости, аннотируемые следующим образом: @ApplicationScope, @ConversationScope, @SessionScope, @RequestScope. CDI-контейнер управляет временем жизни компонентов, основываясь на определенной для компонента области видимости. Например, компонент, аннотированный @ essionScope, существует до тех пор, пока живет HTTP-сессия. В конце области видимости компонент уничтожается и помечается для «сборки мусора». Такое поведение противоположно поведению EJB-компонентов, которые не ограничены областью видимости. Это значит, что вы должны явно уничтожить компонент путем вызова метода, аннотированного аннотацией @Remove.

Перехватчики

У большинства приложений есть какая-либо функциональность, которая не вполне вписывается в ядро логики приложения, но не может быть четко отделена от проекта приложения или его реализации. Эта функциональность сквозная, она влияет на различные части приложения. Она часто несет ответственность за дублирование кода и взаимные зависимости, отрицательно влияющие на расширяемость системы. Реализация такой, не основной, функциональности в виде перехватчиков позволяет разделять ее с основной. Это осуществляется с помощью логического разделения их реализаций, перехвата обращений метода к ядру и вызова соответствующего метода.

Перехватчики реализуются путем использования аннотации @Interceptors со следующим за ней именем класса сквозной функциональности. В листинге 2.4 метод setValue перехватывается во время его вызова классом LoggerInterceptor.class.

Листинг 2.4. Метод ядра, перехватываемый перехватчиком-регистратором

```
@Interceptors(LoggerInterceptor.class)
public void setValues(Integer value, String name) {
    this.value = value;
    this.name = name;
}
```

Перехватчик-регистратор имеет доступ к параметрам перехватываемого метода и использует сквозную логику перед возвратом к полному выполнению перехватываемого метода.

В листинге 2.5 перехватчик-регистратор осуществляет доступ к параметрам метода `setValue` и регистрирует их в системном журнале.

Листинг 2.5. Перехватчик-регистратор

```
@AroundInvoke
public logger(InvocationContext ic) throws Exception {
    Object[] params = ic.getParameters();
    logger.warning("Parameters passed: " + params);
}
```

Вы можете определить перехватчики в бизнес-коде и в файлах дескрипторов развертывания. Этот аспект перехватчиков и многое другое обсуждается в главе 8.

Резюме

В этой главе был проведен краткий обзор Java EE и рассмотрена история принципов, на которых данная платформа в настоящий момент базируется.

Вы открыли для себя, каким образом правильно разделить на слои архитектуру проекта Java EE. Мы также предоставили вам длинный список JSR-совместимости, чтобы помочь определить, какой контейнер лучше подходит для вашего проекта. Наконец, глава обращает внимание на базовые принципы Java EE, показывая соглашения по конфигурации и предоставляя краткую сводку CDI.

Теперь мы готовы перейти к рассмотрению паттернов, концентрируясь на их реализации и приводя конкретные примеры.

Упражнения

1. Обдумайте банковское приложение, которое требуется интегрировать с машинным интерфейсом мейнфрейма и в котором нужно обеспечить сервисы для веб-клиентов, мобильных клиентов и нативных клиентов для настольных компьютеров.
2. Обдумайте реализацию веб-приложения для проекта, который вы разработали в предыдущем упражнении. На каком слое оно должно располагаться?
3. После долгих споров банк, на который вы работаете, принял решение отказаться от мейнфрейма, предложив вам спроектировать систему на замену. Какие части текущего проекта будут затронуты?

ЧАСТЬ II

Реализация паттернов проектирования в Java EE

3 Паттерн «Фасад»

В этой главе:

- введение в замысел паттерна «Фасад»;
- краткое обсуждение выгод от этого паттерна;
- три возможных способа реализации паттерна: POJO, сеансовый компонент-фасад с сохранением состояния и без сохранения состояния;
- важные отличия между сеансовым компонентом-фасадом с сохранением состояния и без сохранения состояния;
- когда и где использовать паттерн;
- предупреждения относительно его использования и возможные подводные камни.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода вы найдете в файле Chapter 03.zip, каждый из них назван в соответствии с наименованиями в тексте.

Паттерн «Фасад» — один из структурных паттернов проектирования, описанных в книге GoF. Его задача — в инкапсуляции сложной бизнес-логики в высокоуровневом интерфейсе, что облегчает использование обращения к подсистеме. Это часто осуществляется путем группировки связанных обращений к методам и вызова их последовательно из одного метода.

С высокоуровневой точки зрения каждый API может рассматриваться как реализация паттерна «Фасад», так как обеспечивает простой интерфейс, скрывающий внутреннюю сложность. Любое обращение к одному из методов API приводит к вызову множества других методов из скрытой за ним подсистемы. Примером фасада может служить интерфейс `javax.servlet.http.HttpSession`. Он скрывает сложную логику, связанную с поддержанием сессии, раскрывая ее функциональность через небольшое количество простых в использовании методов.

Что такое фасад

Книга GoF описывает этот паттерн как «предоставляющий унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме». Книга «Паттерны

проектирования» дает это же толкование и обращает внимание, что, скрывая сложность подсистемы, паттерн «Фасад» в то же время предоставляет все возможности подсистемы через удобный для использования интерфейс.

Для простого практического примера того, как работает паттерн «Фасад», представьте стиральную машину со всего лишь двумя режимами стирки: для сильно загрязненного белья и для слабо загрязненного. Для каждого режима стиральная машина должна выполнить предопределенный набор операций: установить температуру воды, нагреть воду, установить длительность цикла стирки, добавить стиральный порошок, добавить отбеливающее средство, добавить смягчитель ткани и т. д. Каждый режим требует различного набора инструкций по стирке (разное количество стирального порошка, более высокая/низкая температура, более долгий/короткий цикл отжима и т. д.).

Простой интерфейс предоставляет два режима стирки, скрывающих сложную логику выбора подходящей температуры воды, длительности отжима и цикла стирки, а также различные методики добавления стирального порошка, отбеливающего средства или смягчителя ткани. Пользователь стиральной машины не должен думать о сложной логике стирки вещей (выбирать температуру, длительность цикла и т. д.). Единственное, что должен сделать пользователь, — решить, сильно загрязнено белье или нет. В этом состоит сущность паттерна «Фасад» применительно к конструкции стиральных машин. Далее в этой главе вы увидите реализацию этого сценария использования.

Паттерн «Фасад» обычно реализуется в следующих целях и случаях:

- для обеспечения простого и унифицированного доступа к унаследованной системе управления производством;
- для создания общедоступного API к таким классам, как драйвер;
- для предоставления крупномодульного доступа к доступным сервисам. Сервисы сгруппированы как в вышеприведенном примере со стиральной машиной;
- чтобы снизить количество сетевых вызовов. Фасад выполняет множество обращений к подсистеме, в то время как удаленный клиент должен выполнить одно-единственное обращение к фасаду;
- для инкапсуляции последовательности выполняемых действий и внутренних деталей приложения, чтобы обеспечить простоту и безопасность.

ПРИМЕЧАНИЕ

Фасады также иногда реализуют как абстрактные фабрики-одиночки.

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

В ранние, тяжелые для J2EE времена я работал стажером-разработчиком, трудясь над громадным банковским приложением, в котором мы реализовали практически все паттерны проектирования J2EE. Все корпоративные Java-компоненты (EJB) были обернуты в фасады, и каждый сервисный EJB, использующий этот фасад, был обернут в еще один фасад. Для уверенности, что мы не нарушили API, у нас также были интерфейсы к фасадам. В J2EE, EJB требуется локальный и удаленный интерфейс, так что написание одного компонента EJB означало написание четырех интерфейсов и двух классов. У нас не было спагетти-кода, но было больше слоев, чем в мясной лазанье. Мы были вполне счастливы в нашем маленьком мирке — до тех пор пока другие группы разработчиков не начали использовать наши базовые сервисы. Очень скоро начала страдать как производительность системы, так и наша способность обрабатывать запросы на внесение изменений.

Мы наняли известного и весьма недешевого консультанта у одного из поставщиков серверов для анализа нашей системы. Он провел несколько совещаний, потратил немного времени, чтобы пролистать нашу базу кода и заключить, что не помешало бы провести небольшой рефакторинг, и в результате он удалил все фасады и связанные с ними интерфейсы. Так мы получили меньше требующего поддержки кода и гораздо лучшую производительность, и все были довольны. Мораль этой истории: использовать паттерны — даже простые — следует скупо и только тогда, когда они вам действительно нужны, и уж, конечно, не для того, чтобы продемонстрировать ваше знание паттернов.

Диаграмма классов фасада. Как можно увидеть на диаграмме классов на рис. 3.1, паттерн «Фасад» предоставляет простой интерфейс для базовой системы, инкапсулируя сложную логику.

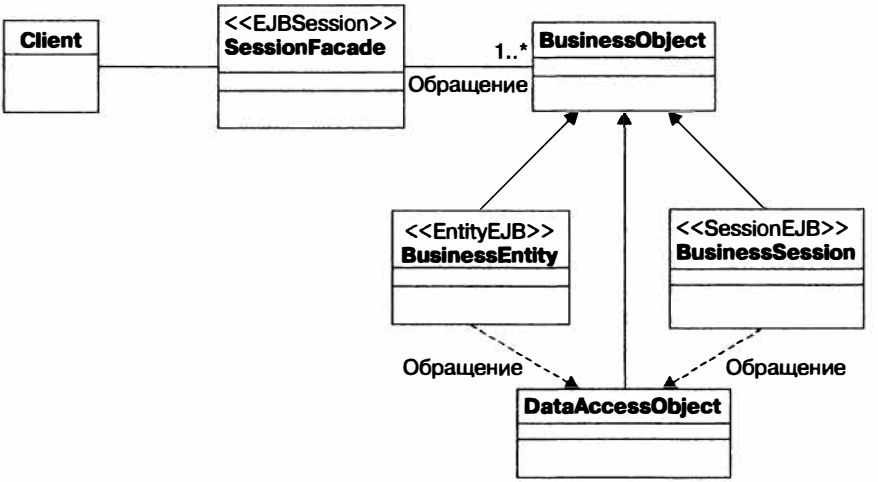


Рис. 3.1. Диаграмма классов паттерна «Фасад»

Реализация паттерна «Фасад» в простом коде

Реализация паттерна «Фасад» проста. Она не требует использования жесткой структуры или набора правил. Любой метод, обеспечивающий простой доступ к сложным последовательностям выполняемых действий, может считаться реализацией паттерна «Фасад».

Сейчас мы реализуем приведенный выше пример со стиральной машиной, как показано в листинге 3.1. Нам понадобятся два метода: `heavilySoiled` и `lightlySoiled`, которые представляют два режима стирки. Вся сложная работа (выбор температуры воды, длительность цикла отжима, решение о том, добавить или нет стиральный порошок и отбеливающее средство) выполняется в методах, вызываемых из внутренностей фасада.

Листинг 3.1. Реализация аналогии со стиральной машиной

```
public class WashingMachine {

    public void heavilySoiled() {
```

```
    setWaterTemperature(100);
    setWashCycleDuration(90);
    setSpinCycleDuration(10);
    addDetergent();
    addBleach();
    addFabricSoftener();
    heatWater();
    startWash();
    }

    public void lightlySoiled() {
    setWaterTemperature(40);
    setWashCycleDuration(20);
    setSpinCycleDuration(10);
    addDetergent();
    heatWater();
    startWash();
    }
}

// Чтобы использовать фасад
new WashingMachine().lightlySoiled();
```

Если вы хотите использовать эту функциональность, просто вызовите метод фасада `heavilySoiled` (или `lightlySoiled`) и позвольте ему выполнить всю сложную логику стирки белья. Сложная логика скрыта фасадом и доступна для использования через два его метода.

Реализация методов не связана с клиентом. Это разделение позволяет реализации меняться без изменения способа, которым клиент получает доступ к сервисам стирки. Клиент ничего не знает о реализации этих методов, и его это не волнует. Все, что ему важно, — получить требуемый ему сервис.

Этот пример демонстрирует одну из многих выгод, получаемых от паттерна «Фасад». Мы не будем углубляться в их детали, так что просто приведем список наиболее важных.

- Снижение связности, поскольку клиент ничего не знает о подсистеме.
- Повышение удобства сопровождения и легкости управления при необходимости изменений.
- Повторное использование функциональности, поскольку данный паттерн содействует повторному применению управляющих элементов и мелкоструктурной логики.
- Переиспользование сервисов за счет вызовов одного и того же метода согласованно от одного вызова к другому.
- Снижение сложности бизнес-логики за счет группирования связанных методов и вызова их из одного метода.
- Централизация безопасности и контроля за управлением транзакциями.
- Реализация паттерна легко поддается тестированию и проверке с помощью объектов-имитаций.

ПРИМЕЧАНИЕ

Как вы можете заметить, на эту реализацию мы будем ссылаться как на POJO-фасад, чтобы различать ее с реализациями с сохранением состояния и без сохранения состояния, которые вы увидите далее в этой главе.

Реализация паттерна «Фасад» в Java EE

В отличие от многих других паттернов, описанных в этой книге, для фасада Java EE не предлагает встроенной реализации. Тем не менее несложно реализовать его с использованием EJB-компонентов с сохранением состояния или без него. Использование EJB-компонентов дает преимущество легкого доступа к другим EJB-компонентам, которые могут потребоваться фасаду.

Фасад с компонентами без сохранения состояния

Чтобы продемонстрировать эту реализацию, предположим, что у вас есть три EJB-компонента, как показано в листинге 3.2, с различной, но связанной функциональностью: CustomerService, LoanService и AccountService.

Листинг 3.2. Код трех EJB-компонентов, которые составляют подсистему для фасада

```
package com.devchronicles.facade;

import javax.ejb.Stateless;

@Stateless
public class CustomerService {

    public long getCustomer(int sessionID) {
    // Зарегистрировать идентификатор клиента
    return 100005L;
    }

    public boolean checkId(long x) {
    // Проверить, действителен ли идентификатор клиента
    return true;
    }
}

package com.devchronicles.facade;

import javax.ejb.Stateless;

@Stateless
public class LoanService {
    public boolean checkCreditRating(long id, double amount) {
    // Проверить, имеет ли клиент право на данную сумму
    return true;
    }
```

```
}

package com.devchronicles.facade;

import javax.ejb.Stateless;

@Stateless
public class AccountService {

    public boolean getLoan(double amount) {
    // Проверить, достаточно ли денег в банковском хранилище
    return true;
    }

    public boolean setCustomerBalance(long id, double amount) {
    // Установить новый баланс клиента
    return true;
    }
}
```

Вы можете сгруппировать эти EJB-сервисы в логическую совокупность связанной функциональности, чтобы образовать реализацию паттерна «Фасад», как показано в листинге 3.3.

Листинг 3.3. Реализация фасада без сохранения состояния

```
package com.devchronicles.facade;

import javax.ejb.Stateless;
import javax.inject.Inject;

@Stateless
public class BankServiceFacade {

    @Inject
    CustomerService customerService;

    @Inject
    LoanService loanService;

    @Inject
    AccountService accountService;

    public boolean getLoan(int sessionId, double amount) {
        boolean result = false;
        long id = customerService.getCustomer(sessionId);

        if(customerService.checkId(id)){
            if(loanService.checkCreditRating(id, amount)){
                if(accountService.getLoan(amount)){
                    result = accountService.setCustomerBalance(id, amount);
                }
```

```
            }
        }
        return result;
    }
}
```

Один фасад может вызывать другие в других подсистемах, которые, в свою очередь, инкапсулируют свою собственную логику и последовательность действий. Это демонстрирует одно из преимуществ использования фасадов — более простую иерархию вызовов методов. Для каждой подсистемы — один фасад, и подсистемы общаются между собой посредством этих паттернов.

Фасад с компонентами с сохранением состояния

Тот же компонент может быть реализован как сеансовый компонент с сохранением состояния или даже как компонент-одиночка, поскольку он скрывает некоторую сложную логику и делает доступным клиенту только простой в использовании интерфейс. Единственное отличие состоит в добавлении аннотации `@Stateful`, помечающей компонент как EJB с сохранением состояния.

В J2EE (до версии 5.0) использование паттерна «Фасад» поддерживалось в реализации сеансового паттерна «Фасад». Однако даже в упрощенном подходе Java EE фасады все еще находят свое применение, если необходимы контроль и инкапсуляция последовательности выполняемых действий.

Где и когда использовать паттерн «Фасад»

Паттерн желательно использовать для инкапсуляции сложной (бизнес-)логики на высоком уровне и для обеспечения одной «чистой» точки доступа через API.

Всякий раз, когда вы обеспечиваете кому-то интерфейс или API, подумайте сначала о сложности логики и возможных изменениях. Паттерн «Фасад» выполняет неплохую работу по обеспечению «чистого» API, в то же время скрывая части, которые могут меняться.

Однако чрезмерное оборачивание методов в фасады — порочная практика, добавляющая лишние слои. Преждевременная инкапсуляция может привести к слишком большому количеству вызовов и слоев, не приносящих пользы.

При реализации паттерна «Фасад» вы должны определить, требует ли данный сценарий сохранения состояния. Сценарий использования, в котором вызывается только один метод фасада для получения нужного ему сервиса, считается не диалоговым, так что нет необходимости в поддержании состояния диалога между одним обращением к методу и следующим. Такой фасад вам лучше реализовывать как сеансовый компонент без сохранения состояния.

С другой стороны, если состояние диалога должно поддерживаться между вызовами метода, наиболее подходящий вариант реализации фасада — сеансовый компонент с сохранением состояния.

Вам следует быть осторожными при использовании сеансового фасада с сохранением состояния, поскольку он занимает ресурсы сервера до тех пор, пока клиент,

инициировавший диалог, не освобождает их или не истекает время ожидания. Это означает, что большую часть времени сеансовый компонент с сохранением состояния «привязан» к клиенту и ничего не делает — просто поддерживает состояние и использует ресурсы. И, в отличие от сеансовых компонентов-фасадов без сохранения состояния, он не может быть применен повторно и совместно использоваться другими клиентами, поскольку каждый запрос создает новый экземпляр фасада без сохранения состояния, поддерживая состояние для сессии этого клиента.

Так что будьте осторожны, применяя этот паттерн. Тщательно обдумайте сценарий использования и принимайте адекватные решения.

Резюме

Вы можете реализовать паттерн «Фасад» как POJO, как сеансовый компонент с сохранением состояния или сеансовый компонент без сохранения состояния. Различные способы реализации этого паттерна решают различные задачи для разных сценариев использования. Но разнообразие реализаций не должно отвлекать вас от его основного замысла: обеспечивать простой высокоуровневый интерфейс к более сложной подсистеме.

Обратите внимание: принимая решение о реализации фасада как сеансового компонента с сохранением состояния, удостоверьтесь, что он не вызовет проблем с потреблением ресурсов.

Хорошо спроектированное приложение может воспользоваться паттерном «Фасад», чтобы инкапсулировать сложную логику и разделить подсистемы с клиентами; однако преждевременное и чрезмерное использование паттерна может привести к слишком сложной системе с множеством слоев.

Сеансовый паттерн «Фасад» схож с «Границей» (Boundary) в архитектурном паттерне «Сущность — управление — граница» (Entity — Control — Boundary) и родственен паттернам «Адаптер» и «Обертка».

Упражнения

1. Перечислите несколько общедоступных API — реализаций паттерна «Фасад» и объясните, как они скрывают сложную логику подсистемы.
2. Разработайте фасад, скрывающий сложную логику системы заказа и оплаты.
3. Инкапсулируйте вызовы методов к двум подсистемам — оплата и заказ — всего лишь в двух методах.
4. Разработайте фасад как одиночку.

4 Паттерн «Одиночка»

В этой главе:

- различные пути, которыми разработчик может реализовать паттерн проектирования «Одиночка», а также его практическое использование и подводные камни;
- проблемы, вызываемые в многопоточной среде использованием статических методов и членов классов;
- улучшения в Java SE 5, связанные с появлением перечислимого типа `enum`, и его использование для создания ориентированных на многопоточное использование одиночек;
- использование аннотации `@Singleton` в Java EE и как это в корне изменило способ реализации паттерна «Одиночка» в сеансовых компонентах;
- использование компонент- и контейнер-управляемого параллелизма и как аннотация `@LockType` управляет доступом к бизнес-методам;
- основные проблемы, преследующие паттерн «Одиночка», и почему он считается антипаттерном, утратившим расположение разработчиков.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода вы найдете в файле Chapter 04.zip, каждый из которых назван в соответствии с наименованиями в тексте.

Паттерн «Одиночка» — один из самых простых и наиболее популярных паттернов проектирования, однако он «вышел из моды». Некоторые даже считают его антипаттерном, что мы и обсудим дальше в этой главе. Однако корпоративные фреймворки, такие как Spring, интенсивно его применяют, а Java EE предлагает изящную и легкую в использовании реализацию. В этой главе вы увидите, почему одиночки необходимы, почему они «вышли из моды», чем они могут быть полезны в приложениях Java EE и как вы можете их реализовать.

Паттерн «Одиночка» — один из порождающих шаблонов, описанных в книге GoF. Класс-одиночка гарантирует, что будет создан только один экземпляр его типа. Наличие только одного экземпляра может быть полезно в нескольких случаях, таких как глобальный доступ и кэширование дорогостоящих ресурсов; однако оно может создать некоторые проблемы из-за состояния гонки, когда

одиночка используется в многопоточной среде. Поскольку большинство языков программирования не предлагают встроенного механизма для создания одиночек, разработчикам приходится программировать свои собственные реализации.

Однако в платформе Java EE есть встроенный механизм, дающий разработчику простой способ создания одиночки посредством добавления аннотации к классу.

Что такое одиночка

Согласно GoF, паттерн «Одиночка» «гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа». В книге «Паттерны проектирования» дается такое же объяснение. Одиночки часто используются в сочетании с фабриками (обсуждаются в главе 6).

Одиночки, как правило, используются в следующих целях и случаях:

- для получения доступа к совместно используемым данным, таким как конфигурационные данные, повсюду в контексте приложения;
- чтобы улучшать производительность, загружать и кэшировать дорогостоящие ресурсы только один раз, разрешая глобальный совместный доступ;
- для создания экземпляра регистратора для приложения, поскольку обычно необходим только один регистратор;
- для управления объектами внутри класса, реализующего паттерн «Фабрика»;
- для создания объекта «Фасад», так как обычно требуется только один такой объект;
- для отложенного создания статических классов, поскольку одиночки могут быть инициализированы отложенно.

Spring использует одиночек при создании компонентов (по умолчанию компоненты Spring — одиночки), платформа Java EE применяет одиночек внутренним образом, как, например, в локаторе сервисов. Платформа Java SE также использует паттерн «Одиночка» в реализации `runtime` класса. Так что одиночки определенно полезны, если вы используете их в правильной ситуации.

Тем не менее интенсивное применение паттерна «Одиночка» может означать ненужное кэширование ресурсов и препятствование возвращению сборщиком мусора объектов в пользование и освобождению им ценных ресурсов памяти. Оно также может означать, что вы на самом деле не используете выгоды от создания объектов и наследования. Необычно интенсивное применение одиночек считается признаком плохого объектно-ориентированного проектирования приложения, что может привести к проблемам с памятью и производительностью. Другая проблема — одиночки не очень хорошо себя проявляют при поблочном тестировании приложения. Проблемы, сопровождающие использование паттерна «Одиночка», будут детальнее обсуждаться дальше в этой главе.

Диаграмма классов одиночки

Как вы можете видеть из диаграммы классов на рис. 4.1, паттерн «Одиночка» основан на одном классе, хранящем ссылку на свой единственный экземпляр, наряду с контролем его создания и доступа к нему через единственный метод чтения.

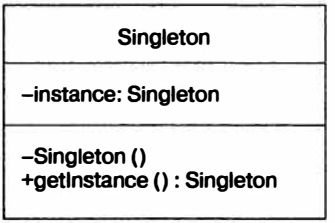


Рис. 4.1. Диаграмма класса паттерна «Одиночка»

Реализация паттерна «Одиночка» в простом коде

Поскольку вам нужно гарантировать, что одиночки используют только один экземпляр, первая вещь, которую нужно делать, — контролировать создание объекта. Вы можете легко это осуществить, сделав конструктор невидимым для внешнего мира.

```
package com.devchronicles.singleton;

public class MySingleton {

    private MySingleton() {
        // Здесь код реализации
    }
}
```

Далее вам понадобится метод, создающий экземпляр или возвращающий уже созданный. Поскольку экземпляр класса `MySingleton` еще не существует, необходимо пометить этот метод как статический, чтобы разрешить доступ к нему по имени класса, например `MySingleton.getInstance()`.

В месте, обозначенном комментарием 1 в листинге 4.1, проверяется, создан ли уже одиночка, и если нет, то создается; в противном случае возвращается экземпляр, созданный во время предыдущего вызова метода `getInstance()`. Каждый последующий вызов возвращает ранее созданный экземпляр класса `MySingleton`. Этот код может показаться функционирующим, однако он неполный и содержит множество ошибок. Поскольку метод создания объекта неатомарен, он подвержен ошибкам в состоянии гонки, допуская создание более чем одного экземпляра одиночки в многопоточной среде.

Листинг 4.1. Простая реализация паттерна "Одиночка"

```
package com.devchronicles.singleton;

public class MySingleton {
```

```
    private static MySingleton instance;

    private MySingleton() {}

    public static MySingleton getInstance() {
        if (instance==null){ // 1
            instance=new MySingleton();
        }
    return instance;
    }
}
```

Чтобы решить проблему состояния гонки, необходимо запросить блокировку и не освобождать ее до момента возврата экземпляра. В языке Java вы можете реализовать механизм блокировок путем использования ключевого слова synchronized, как показано в листинге 4.2.

Листинг 4.2. Синхронизация одиночки для реализации многопоточности

```
package com.devchronicles.singleton;

public class MySingleton {

    private static MySingleton instance;

    private MySingleton() {}

    public static synchronized MySingleton getInstance() {
        if (instance==null){
            instance=new MySingleton();
        }
    return instance;
    }
}
```

Другой подход — создать экземпляр одиночки во время загрузки класса, как показано в листинге 4.3. Это позволяет избавиться от необходимости синхронизации создания экземпляра одиночки и создает объект одиночки сразу же, как только JVM загрузила все классы (и, следовательно, до того, как класс может вызвать метод getInstance()). Это происходит потому, что статические члены класса и блоки инициализации выполняются при загрузке класса.

Листинг 4.3. Создание объекта одиночки во время загрузки класса

```
package com.devchronicles.singleton;

public class MySingleton {

    private final static MySingleton instance=new MySingleton();

    private MySingleton() {}

    public static MySingleton getInstance() {
```

```
        return instance;
    }
}
```

Еще один подход — использовать статический блок инициализации, как показано в листинге 4.4. Однако это ведет к отложенной инициализации, поскольку статические блоки вызываются перед вызовом конструктора.

Листинг 4.4. Создание объекта одиночки в статическом блоке

```
package com.devchronicles.singleton;

public class MySingleton {

    private static MySingleton instance=null;

    static {
        instance=new MySingleton();
    }
    private MySingleton() {}

    public static MySingleton getInstance() {
        return instance;
    }
}
```

Еще одним очень популярным механизмом создания одиночек является блокировка с двойной проверкой. Она считается безопаснее других методов, поскольку проверяет экземпляр одиночки один раз перед блокировкой класса одиночки и второй — перед созданием объекта. Листинг 4.5 демонстрирует этот метод.

Листинг 4.5. Реализация блокировки с двойной проверкой

```
package com.devchronicles.singleton;

public class MySingleton {

    private volatile MySingleton instance;

    private MySingleton() {}

    public MySingleton getInstance() {
        if (instance == null) { // 1
            synchronized (MySingleton.class) {
                if (instance == null) { // 2
                    instance = new MySingleton();
                }
            }
        }
        return instance;
    }
}
```

Метод getInstance() дважды сравнивает приватный член экземпляра MySingleton (сначала в месте, отмеченном комментарием 1, и затем еще раз — возле комментария 2) с null перед созданием экземпляра MySingleton и присвоением ему значения.

Однако ни один из этих методов не является безопасным на все 100 %. Например, Java Reflection API позволяет разработчикам менять модификатор доступа конструктора на public, таким образом делая одиночку доступным для повторного создания.

Лучший способ создания одиночек в Java — использование типа enum, введенного Java EE 5 и представленного в листинге 4.6. Этот подход настойчиво рекомендует Джошуа Блох в своей книге *Effective Java*[1]. Перечислимые типы по своей сущности одиночки, так что JVM берет на себя большую часть работы по созданию одиночки. Таким образом, при использовании типа enum вы освобождаетесь от забот по синхронизации создания и обеспечения деятельности объекта и избегаете проблем, связанных с инициализацией.

Листинг 4.6. Реализация паттерна «Одиночка» на основе типа enum

```
package com.devchronicles.singleton;

public enum MySingletonEnum {
    INSTANCE;
    public void doSomethingInteresting(){}
}
```

В этом примере вы получаете ссылку на экземпляр одиночки следующим образом:

```
MySingletonEnum mse = MySingletonEnum.INSTANCE;
```

Как только у вас есть ссылка на одиночку, вы можете вызвать любой из его методов вот так: mse.doSomethingInteresting();.

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Несколько лет назад один мой близкий друг, владелец маленькой компании-разработчика ПО, попросил меня помочь с проведением собеседования при приеме на работу одного соискателя. А поскольку я всегда считал работу по проведению собеседований разработчиков отличной практикой технического общения, то, не колеблясь, ухватился за эту возможность. У соискателя было лишь несколько лет опыта работы, но он был выпускником хорошего университета и явно был весьма сообразительным. Мы долго беседовали о Java, JPA, Spring и других известных Java-фреймворках. Он стремился учить и пробовать новые технологии и хладнокровно расправлялся со всеми вопросами.

Спустя примерно час собеседование завершилось, но мы продолжали разговаривать. Я спросил его, какие книги он прочитал в последнее время, и он назвал Head First Design Patterns. Я осведомился, какой паттерн он считает самым интересным или о каком просто хотел бы поговорить. Ничего удивительного, что это оказался паттерн «Одиночка». Это заставило меня почувствовать себя пауком, к которому залетела в гости муха. Соискатель думал, что паттерн «Одиночка» прост и легок в реализации, так что это было бы безопасной темой для беседы. Правда заключалась в том, что это было не так. Если бы он выбрал другой паттерн, например «Декоратор», у меня не было бы такой возможности расспросить про все тонкости этого кажущегося простым паттерна.

---

[1] *Блох Дж.* Java. Эффективное программирование. 2-е изд. — М.: Лори, 2014. — 461 с.

Тогда я спросил соискателя, как бы он реализовал одиночку. Он ответил, что с помощью классического метода приватного конструктора, что четко демонстрировало, что он не читал Effective Java и понятия не имеет об одиночке перечислимого типа. Затем я спросил: «Что, если я использую рефлексию для изменения уровня доступа конструктора на общедоступный?» Он удивился, но я ясно видел по его глазам, что ему не терпелось вернуться домой и попробовать это. Он не мог предложить мне решение, но и не стал придумывать взамен какую-то глупость. Он был занят перевариванием и обдумыванием идеи рефлексии.

Этот соискатель вакансии, возможно, не знал правильного ответа, но продемонстрировал желание узнать и научиться чему-то новому. Он был принят на работу и стал одним из лучших разработчиков, с кем я когда-либо работал.

Реализация паттерна «Одиночка» в Java EE

Все ранее приведенные примеры кода демонстрировали использование одиночек в контексте платформы Java SE. Хотя вы можете использовать их и на платформе Java EE, есть более изящный и простой подход: компоненты-одиночки.

Компоненты-одиночки

В главе 2 вы видели использование сеансовых компонентов с сохранением состояния и без него с помощью простой конфигурации аннотации. К счастью, одиночки предлагают схожий подход. Класс можно превратить в компонент-одиночку простым добавлением к нему аннотации `@Singleton`, как показано в листинге 4.7.

Листинг 4.7. Реализация паттерна "Одиночка" с использованием `@Singleton`

```
package com.devchronicles.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import java.util.logging.Logger;

@Singleton
public class CacheSingletonBean {

    private Map<Integer, String> myCache;

    @PostConstruct
    public void start(){
        Logger.getLogger("MyGlobalLogger").info("Started!");
        myCache = new HashMap<Integer, String>();
    }

    public void addUser(Integer id, String name){
        myCache.put(id, name);
    }

    public String getName(Integer id){
```

```
        return myCache.get(id);
    }
}
```

Благодаря простоте использования аннотаций в Java EE не требуется конфигурационный XML-файл. Вы можете заметить в проекте файл `beans.xml`, но он почти всегда остается пустым. Он понадобится вам только для запуска контейнера CDI. Аннотация `@Singleton` помечает класс как EJB-одиночку, и контейнер управляет созданием и использованием единственного экземпляра.

Если вы выполняете код этого компонента на вашем сервере, вы не увидите вывода регистратора от одиночки, поскольку метод, аннотированный `@PostConstruct`, не был вызван. Почему так происходит?

Использование одиночек при запуске

Одиночки на платформе Java EE по умолчанию инициализируются отложенно. Это подходит в большинстве ситуаций: разрешать создание экземпляра только тогда, когда он становится нужен и к нему обращаются в первый раз. Однако вы можете захотеть создать экземпляр при запуске, чтобы разрешить доступ к одиночке немедленно, особенно если создание экземпляра ресурсоемко или компонент вам заведомо будет необходим с момента запуска приложения. Чтобы гарантировать, что экземпляр создается при запуске, используйте для класса аннотацию `@Startup`, как показано в листинге 4.8.

Листинг 4.8. Вызов одиночки при запуске приложения

```
package com.devchronicles.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import java.util.logging.Logger;

@Startup
@Singleton
public class CacheSingletonBean {

    private Map<Integer, String> myCache;

    @PostConstruct
    public void start(){
        Logger.getLogger("MyGlobalLogger").info("Started!");
        myCache = new HashMap<Integer, String>();
    }

    public void addUser(Integer id, String name){
        myCache.put(id, name);
    }
```

```
    public String getName(Integer id){
        return myCache.get(id);
    }
}
```

Если вы перезапустите ваш сервер, вызовется метод постконструкции, поскольку одиночка теперь создается при запуске сервера. Регистратор будет получать сообщение `Started!`.

Определение порядка запуска

Это может поднять еще один вопрос. Что, если одиночка, который вы только что создали, зависит от другого ресурса? Как вы организуете ожидание готовности этого другого ресурса? Хотя это и может представляться «тупиковым случаем»[1], но таковым определенно не является. Подумайте об одиночке, который загружает и кэширует сообщения от базы данных. Это может показаться тривиальным, но даже элементарный доступ к базе данных только для чтения может зависеть от других сервисов. Что, если пул соединений создается другим одиночкой или, еще лучше, что, если журналирование зависит от другого одиночки? Платформа Java EE предлагает простую аннотацию для выхода из этой ситуации. Вы можете использовать аннотацию `@DependsOn`, передавая ей в качестве параметра имя компонента, от которого зависит класс (листинг 4.9). Теперь вы можете легко определить последовательность выполнения одиночек.

Листинг 4.9. Задание последовательности запуска одиночек с использованием аннотации `@DependsOn`

```
package com.devchronicles.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.DependsOn;
import javax.ejb.EJB;

@Startup
@DependsOn("MyLoggingBean")
@Singleton
public class CacheSingletonBean {

    private Map<Integer, String> myCache;

    @EJB
    MyLoggingBean loggingBean;
```

[1] В англоязычной литературе используются термины corner case или pathological case. См.: https://en.wikipedia.org/wiki/Corner_case. — *Примеч. пер.*

```
    @PostConstruct
    public void start(){
        loggingBean.logInfo("Started!");
        myCache = new HashMap<Integer, String>();
    }

    public void addUser(Integer id, String name){
        myCache.put(id, name);
    }

    public String getName(Integer id){
        return myCache.get(id);
    }
}
```

Теперь создадим другой компонент-одиночку (листинг 4.10), на который предыдущий компонент уже ссылался.

Листинг 4.10. Задание последовательности запуска

```
package com.devchronicles.singleton;

import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import java.util.logging.Logger;

@Startup
@Singleton
public class MyLoggingBean {

    private Logger logger;

    @PostConstruct
    public void start(){
        logger = Logger.getLogger("MyGlobalLogger");
        logger.info("Well, I started first!!!");
    }

    public void logInfo(String msg){
        logger.info(msg);
    }
}
```

Здесь вы можете также использовать аннотацию `@PostConstruct`, чтобы проверить, что ваш компонент был создан и его жизненный цикл начался. Методы, аннотированные с помощью `@PostConstruct`, вызываются на вновь созданных компонентах после внедрения всех зависимостей и перед вызовом первого бизнес-метода. Конечно, в реальной жизни вам придется использовать компоненты-одиночки внутри других компонентов. Дальнейшие главы уделят больше внимания интеграции и доступу к EJB-компонентам и тому, должны ли они быть одиночками.

Компоненты из предыдущего примера запускаются при старте сервера. CacheSingletonBean будет ожидать, поскольку он зависит от инициализации MyLoggingBean. Вывод регистратора будет примерно таким:

```
> Well. I started first!!!
> Started!
```

Ваш компонент-одиночка может зависеть от инициализации последовательности других компонентов. В этом случае вы можете указать в аннотации @DependsOn несколько компонентов. Следующий компонент-одиночка зависит от MyLoggingBean и MyInitializationBean:

```
@Startup
@DependsOn({"MyLoggingBean","MyInitializationBean"})
@Singleton
public class CacheSingletonBean {
    // Здесь приводится код реализации
}
```

Порядок, в котором инициализируются MyLoggingBean и MyInitializationBean, определяется их собственными аннотациями @DependsOn. Если ни один из них не зависит явно от другого, компоненты инициализируются контейнером в неопределенном порядке.

Управление параллелизмом

Наиболее важной из проблем, с которыми вы столкнетесь, является параллелизм. С реализацией Java EE вам больше не нужно волноваться о создании компонента, но по-прежнему придется быть осмотрительным в отношении доступа к методам, поскольку ваш одиночка будет доступен для использования в условиях параллелизма. Платформа Java EE опять же решает эту проблему с помощью аннотаций.

Платформа Java EE предлагает два типа управления параллелизмом: *контейнерно-управляемый параллелизм* и *компонентно-управляемый параллелизм*. При контейнерно-управляемом параллелизме контейнер отвечает за обработку всего относящегося к доступу для чтения и записи, в то время как при компонентно-управляемом параллелизме ожидается, что разработчик будет управлять параллелизмом, используя традиционные методы языка Java, такие как синхронизация. Вы можете включить компонентно-управляемый параллелизм с помощью аннотации ConcurrencyManagementType.BEAN.

Платформа Java EE по умолчанию использует контейнерно-управляемый параллелизм, но вы можете объявить его явным образом с помощью аннотации ConcurrencyManagementType.CONTAINER.

```
@Startup
@DependsOn("MyLoggingBean")
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class CacheSingletonBean {
    // Здесь код реализации
}
```

Теперь вернемся к нашему примеру и используем аннотацию @Lock для управления доступом (листинг 4.11).

Листинг 4.11. Управление параллелизмом с помощью @Locktype

```
package com.devchronicles.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.DependsOn;
import javax.ejb.EJB;
import javax.ejb.Lock;
import javax.ejb.LockType;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Startup
@DependsOn("MyLoggingBean")
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class CacheSingletonBean {

    private Map<Integer, String> myCache;

    @EJB
    MyLoggingBean loggingBean;

    @PostConstruct
    public void start(){
        loggingBean.logInfo("Started!");
        myCache = new HashMap<Integer, String>();
    }

    @Lock(LockType.WRITE)
        public void addUser(Integer id, String name){
        myCache.put(id, name);
    }

    @Lock(LockType.READ)
    public String getName(Integer id){
        return myCache.get(id);
    }
}
```

Доступом к бизнес-методам компонента управляют два типа блокировок: @ ock(LockType.WRITE), которая блокирует компонент для других клиентов, пока осуществляется вызов метода, и @Lock(LockType.READ), разрешающая параллельный доступ к методу и не блокирующая компонент для других клиентов. Методы, вызывающие изменение данных, обычно аннотируются типом доступа WRITE, чтобы

предотвратить доступ к данным во время их изменения. В приведенном примере метод addUser() аннотирован типом блокировки WRITE, так что, если какой-то клиент обратится к методу getName(), ему придется подождать возврата из метода addUser(), прежде чем он сможет завершить выполнение своего обращения. Это может привести к генерации контейнером исключения ConcurrentAccessTimeoutException, если метод addUser() не завершится до истечения заданного времени ожидания. Вы можете изменить существующие настройки времени ожидания с помощью аннотации, пример которой приведен в листинге 4.12.

Вы можете установить аннотацию LockType на уровне класса. В таком случае она будет применена ко всем бизнес-методам, собственная LockType которых не задана явным образом. Поскольку LockType по умолчанию — WRITE, обычно достаточно задать настройки только тех методов, которые требуют параллельного доступа.

Листинг 4.12. Задание времени ожидания параллельного доступа одиночки

```
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.DependsOn;
import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.AccessTimeout;
import java.util.Map;
import javax.ejb.EJB;
import java.util.HashMap;
import javax.ejb.Lock;
import javax.ejb.LockType;
import java.util.concurrent.TimeUnit;

@Startup
@DependsOn("MyLoggingBean")
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
@AccessTimeout(value=120000) // по умолчанию - в миллисекундах
public class CacheSingletonBean {

    private Map<Integer, String> myCache;

    @EJB
    MyLoggingBean loggingBean;

    @PostConstruct
    public void start(){
        loggingBean.logInfo("Started!");
        myCache = new HashMap<Integer, String>();
    }

    @AccessTimeout(value=30, unit=TimeUnit.SECONDS)
    @Lock(LockType.WRITE)
```

```
    public void addUser(Integer id. String name){
        myCache.put(id. name);
    }

    @Lock(LockType.READ)
    public String getName(Integer id){
        return myCache.get(id);
    }
}
```

Аннотация `@AccessTimeout` может принимать в качестве параметра различные константы `TimeUnit`, такие как `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS` и `SECONDS`. Если значение параметра `TimeUnit` не задано, то по умолчанию оно считается миллисекундами. Вы можете также поместить эту аннотацию на уровень класса и применить ее ко всем методам, которые не определяют явным образом аннотацию времени ожидания доступа.

Где и когда использовать паттерн «Одиночка»

Эмпирическое правило гласит, что интенсивное использование одиночек может быть признаком неправильного их применения. Вам следует применять одиночки там, где это имеет смысл, например при кэшировании данных, доступ к которым (осуществляемый довольно часто) требует значительного расхода ресурсов, при совместном использовании данных для глобального доступа или в целях использования единой точки контакта (например, при журналировании).

Создание и кэширование лишних ресурсов негативно влияет на память, ресурсы процессора и начальную загрузку, так что необходимо осторожно обращаться с одиночками при использовании их для кэширования данных. Однако одиночки могут быть весьма полезны и легко конфигурированы в контейнере Java EE. Для серьезных решений, использующих кэширование, следует обдумать возможность применения фрейморка, такого как широко используемый Encache (**http://www.ehcache.org/**) или компонент Apache — распределенная система кэширования JCS (Java Caching System, **https://commons.apache.org/proper/commons-jcs/**).

Вы можете использовать одиночку для управления доступом к прикладной части систем, сделанных без учета многопоточности или имеющих проблемы с лицензированием. Использование на методах аннотации `LockType.WRITE` позволяет осуществлять последовательный доступ к системам, в которых параллельный доступ мог бы вызвать проблемы с производительностью или лицензированием.

Резюме

Мы уже вкратце упоминали, что паттерн «Одиночка» утратил расположение разработчиков вплоть до того, что многие разработчики и программные архитекторы теперь считают его антипаттерном. Увеличивают непопулярность паттерна проблемы,

вызванные его излишним и неправильным использованием, а также его очевидные недостатки при работе в многопоточных приложениях.

Программисты излишне и неправильно использовали паттерн «Одиночка» потому, что он прост в реализации. Так что каждый класс становился одиночкой. Это превращалось в ночной кошмар для разработчиков, которым приходилось потом поддерживать такой код, и становилось еще большей головной болью для тех, кому нужно было переделывать одиночки в объектно-ориентированный код, когда оказывалось, что требуется более чем один экземпляр класса-одиночки.

Использование классов-одиночек усложняло тестирование, поскольку приходилось задавать значения глобальных состояний ради запуска теста простого модуля. Более того, одиночки сделали тесты менее детерминированными, поскольку эти состояния могли меняться, влияя на результаты тестов.

Вы видели в примерах кода многочисленные трудности, доставляемые одиночками в многопоточных средах. До появления Java SE 5 и перечислимого типа было непросто создать одиночку, который гарантированно корректно работал бы в условиях многопоточности.

Однако с достигнутым в платформе Java EE 5 прогрессом проблема многопоточных одиночек была по большей части решена благодаря аннотации `@Singleton` и контейнерно-управляемому параллелизму.

Контейнер управляет созданием одиночки и гарантирует, что ни один бизнес-метод не будет вызван до завершения выполнения `@PostConstruct`. Контейнер также управляет параллелизмом доступа к компоненту с помощью аннотации `@oncurrencyManagement`, а его связанная аннотация `@LockType` делает возможным управление доступом каждого метода на уровне мелких структурных единиц.

Никоим образом нельзя сказать, что все проблемы одиночек были решены. Все еще могут проявляться проблемы в многоузловой среде, если компонент используется для доступа к прикладной части не реализованных в многопоточном исполнении ресурсов. Дополнительными проблемами могут стать узкие места и сильная связь классов.

Несмотря даже на излишнее и неправильное использование паттерна «Одиночка» разработчиками, ведущее к его постепенному низведению до антипаттерна, он тем не менее существенно развился с того времени, как был представлен в книге GoF, и может рассматриваться как полезный и жизнеспособный паттерн проектирования.

Упражнения

1. Разработайте счетчик посещений веб-страницы с двумя методами: один метод будет наращивать счетчик, а второй — возвращать последнее значение. Обеспечьте многопоточное исполнение с помощью задания соответствующих типов блокировок.
2. Разработайте простой кэш, хранящий список книг для приложения управления библиотекой. Данные должны загружаться в кэш при старте приложения. До-

бавьте методы для выборки книг на основе различных критериев, таких как ISBN, автор и жанр.

3. Разработайте сложный кэш, который читает данные из базы данных при запуске. Методы выборки данных должны сначала выполнять запрос к кэшу, и, только если требуемые данные не найдены, компонент должен выполнять запрос к базе данных. Если запрошенные данные найдены в базе данных, они должны быть сохранены в кэше.
4. Добавьте к упражнению 3 механизм, удаляющий из кэша данные, к которым редко обращаются, и обновляющий устаревшие данные. Обеспечьте соответствующее управление полным жизненным циклом кэша.

5 Внедрение зависимостей и CDI

В этой главе:

- основы внедрения зависимостей (DI);
- почему DI в Java EE так важно;
- как реализовать DI в простом коде;
- как DI реализовано в Java EE;
- введение в контекст и внедрение зависимостей;
- ключевые различия между CDI- и EJB-контейнерами.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 05.zip, каждый из них назван в соответствии с наименованиями в тексте.

«Внедрение зависимости» (Dependency Injection) — один из немногих широко известных и общепринятых паттернов проектирования, *не* упомянутый в книге «Банды четырех». Сегодня он широко используется в современных языках программирования как для внутренних целей, так и в качестве устоявшегося варианта обеспечения слабого связывания.

Платформа J2EE была спроектирована с расчетом на самые сложные системы, но потерпела полный крах из-за чрезмерного усложнения разработки систем более простых. Первоначальный замысел J2EE полагался на сложность и сильную связь элементов, что привело к популярности таких фреймворков, как Spring и Pico Container. В 2004 году Мартин Фаулер опубликовал статью на тему инверсии контейнеров управляющих элементов и паттерна «Внедрение зависимости»[1]. Большинство поставщиков не поддерживали и не поощряли использование разработчиками J2EE-контейнеров. Однако вскоре «легковесные» контейнеры взяли верх, стали официально поддерживаться и, более того, фреймворк Spring стал дефакто неофициальным стандартом, что привело к реконструкции корпоративной версии Java практически с нуля.

[1] *Fowler Martin.* Inversion of Control Containers and the Dependency Injection Pattern. — 2004. — http://martinfowler.com/articles/injection.html.

Что такое внедрение зависимостей

Паттерн «Внедрение зависимости» основан на идее инверсии управления. Вместо создания жестких зависимостей и создания новых объектов как с помощью ключевого слова new, так и посредством преобразований, вы внедряете требуемый ресурс в целевой объект. Такой подход имеет много преимуществ:

- клиенту не требуется знать о различных реализациях внедренных ресурсов, что облегчает изменения проекта;
- намного облегчается реализация модульного тестирования с использованием объектов-имитаций;
- конфигурацию можно экспортировать, снижая тем самым влияние изменений;
- слабосвязанная архитектура делает возможным использование различных подключаемых структур.

Основная идея DI заключается в изменении места создания объектов и в использовании механизма внедрения для того, чтобы в нужный момент внедрить определенные реализации в целевые объекты. Это может показаться подобным реализации паттерна «Фабрика» (см. главу 6), но концепция в целом куда шире простого создания объекта. Инверсия управления (Inversion of Control, IoC) меняет всю схему соединения объектов и позволяет выполнять работу механизму внедрения (в большинстве случаев каким-то «магическим» образом). Вместо обращения к фабрике, для того чтобы обеспечить реализацией вызывающую программу, механизм внедрения работает на упреждение, чтобы определить, когда целевому объекту понадобится объект-источник, и выполнить инъекцию надлежащим образом.

Реализация DI в простом коде

Java не предлагала стандартную реализацию DI из EJB-контейнера до появления контекста и внедрения зависимостей (Context and Dependency Injection, CDI). Хотя и есть различные DI-фреймворки, такие как Spring и Guice, запрограммировать простую реализацию совсем не сложно.

Простейшая реализация DI — фабрика, создающая зависимость по запросу с помощью метода `getInstance()`. Сейчас вы реализуете пример, который покажет, как сделать это в простом коде.

Простая реализация DI должна разделять разрешение зависимости от поведения класса. Это означает, что класс должен иметь конкретную функциональность без определения того, как он получает ссылки на классы, от которых зависит. В этом заключается сущность DI: расцепление создания объекта с тем, где объект используется.

Вы начнете с изучения примеров в листингах 5.1–5.4, которые тесно связаны между собой, и перепишете их для использования своего собственноручно сделанного DI.

Листинг 5.1. Класс UserService, создающий в конструкторе новую зависимость

```
package com.devchronicale.di;
class UserService {
    private UserDataRepository udr;
    UserService() {
        this.udr = new UserDataRepositoryImpl();
    }
    public void persistUser(User user) {
        udr.save(user);
    }
}
```

Листинг 5.2. Интерфейс UserDataRepository

```
package com.devchronicale.di;

public interface UserDataRepository {
    public void save(User user);
}
```

Листинг 5.3. Конкретная реализация UserDataRepository

```
package com.devchronicale.di;

public class UserDataRepositoryImpl implements UserDataRepository {
    @Override
    public void save(User user) {
        // Здесь код сохранения
    }
}
```

Листинг 5.4. Пользовательский класс

```
package com.devchronicale.di;

public class User {
    // Здесь пользовательский код
}
```

В листинге 5.1 класс UserService обеспечивает сервисы бизнес-логики для управления пользователями, например сохранение пользователя в базе данных. В этом примере создание объекта выполняется в конструкторе. Это сцепляет бизнес-логику (поведение класса) с созданием объекта.

А теперь вы поменяете этот пример, вынеся создание объекта из вашего класса и поместив его в фабрику.

В листинге 5.5 класс UserDataRepository создается и передается конструктору класса UserService. Вы поменяете конструктор класса UserService так, чтобы он принимал этот новый параметр.

Листинг 5.5. Класс UserServiceFactory, создающий объекты UserService

```
package com.devchronicale.di;

public class UserServiceFactory {
    public UserService getInstance(){
```

```
        return new UserService(new UserDataRepositoryImpl());
    }
}
```

В листинге 5.6 конструктор класса `UserService` запрашивает, чтобы в него была выполнена инъекция экземпляра класса `UserDataRepository`. Класс `UserService` расцепляется с классом `UserDataRepositoryImpl`. Фабрика теперь отвечает за создание объекта и внедряет реализацию в конструктор класса `UserService`. Итак, вы успешно разделили бизнес-логику с созданием объекта.

Листинг 5.6. Переработанный класс UserService

```
package com.devchronicale.di;

class UserService {

    private UserDataRepository udr;

    UserService(UserDataRepository udr) {
        this.udr = udr;
    }

    public void persistUser(User user) {
        udr.save(user);
    }
}
```

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Когда я получил задание написать приложение под Android, я решил проанализировать DI-фреймворки для мобильных устройств. Мне, как разработчику ПО с опытом разработки корпоративных приложений, это казалось правильным выбором. Система пользовательского интерфейса (UI) Android уже зависела от DI-подобной структуры, связывавшей основанные на XML компоненты пользовательского интерфейса с кодом на языке Java, так что казалось разумным реализовать полнофункциональный DI-фреймворк.

Я работал с прекрасной архитектурой, в которой все объекты и ресурсы были собраны в единое целое. Инъекции работали чудесно, однако того же нельзя было сказать о приложении. Запуск приложения занимал гораздо больше времени, чем у схожих приложений, и навигация была тоже не слишком главной. Мы все верили, что DI был обязательным для достижения слабой связи и хорошо организованного кода, поэтому искали проблемы в других местах. Мы довели до совершенства облегченный пользовательский интерфейс и асинхронные фоновые задания, чтобы ничто не могло блокировать приложение и чтобы минимизировать выполняемую при запуске работу, но это не помогло.

Скоро нас осенило, что источником проблемы был DI-фреймворк. Он выполнял поиск всех ссылок и ресурсов инъекций во время запуска приложения и пытался выполнить все соединения в начале жизненного цикла приложения. Это могло быть неплохой идеей при загрузке сервера с множеством пользователей, небольшим количеством перезагрузок и колоссальным объемом оперативной памяти. Но это отнюдь не было хорошей идеей для мобильного устройства с одним пользователем, многочисленными перезагрузками и ограниченной памятью.

Мы приняли решение «зашить» ресурсы. Хотя наше приложение из-за этого и стало «уродливее», запускаться оно стало молниеносно, что полностью решило наши проблемы с производительностью.

Мораль этой истории не в том, что DI — неудачный паттерн для реализации на мобильных устройствах, а в том, что плохая реализация DI (на мобильном устройстве или нет) в неправильном контексте может привести к огромным проблемам.

Реализация DI в Java EE

Платформа J2EE не предлагала готовый к использованию DI вплоть до версии Java EE 5. Взамен этого в J2EE доступ к компонентам и ресурсам осуществлялся путем контекстных поисков интерфейса каталогов и служб именования Java (Java Naming and Directory Interface, JNDI). Такой подход приводил к жесткой связи («зашиванию») и опирался на тяжеловесный серверный контейнер, который делал тестирование едва ли не более сложным, чем само написание кода.

С выходом платформы Java EE 5 и EJB 3 паттерн DI стал неотъемлемой частью корпоративной платформы Java. Чтобы избавиться от основанных на XML конфигурационных файлов, для осуществления инъекций было предложено несколько аннотаций:

- `@Resource` (JSR250) — для внедрения источников данных, сервисов передачи сообщений (Java Message Service, JMS), унифицированных указателей ресурсов (URL), почты и переменных окружения;
- `@EJB` (JSR220) — для внедрения EJB-компонентов;
- `@WebServiceRef` — для внедрения веб-сервисов.

С выходом платформы Java EE 6, CDI и EJB 3.1 паттерн DI стал обладать большим потенциалом, а потому стал более интересным.

В EJB 3.1 интерфейс больше не был обязательным для EJB-компонентов. Кроме того, был представлен новый веб-профиль EJB, предлагавший более простой и «легкий» EJB-контейнер. Была представлена новая, улучшенная аннотация инъекции `@Inject` (JSR229 и JSR330).

Аннотация `@Inject` обеспечивает безопасность типов, поскольку внедряет зависимость, основываясь на типе ссылки на объект. Если бы вам нужно было провести рефакторинг кода в листинге 5.1, вы могли бы убрать конструктор и добавить аннотацию `@Inject` к полю `UserDataRepository`. Код при этом выглядел бы примерно как в листинге 5.7.

Листинг 5.7. Переделанный класс `UserService` с использованием `@Inject`

```
package com.devchronicale.di;

import javax.inject.Inject;

class UserService {

    @Inject
    private UserDataRepository udr;

    public void persistUser(User user) {
        udr.save(user);
    }
}
```

Контейнер CDI создает один экземпляр `UserDataRepositoryImpl` как контейнерно-управляемый компонент и внедряет его везде, где только находит `@Inject`, аннотирующую поле типа `UserDataRepository`.

Вы можете внедрить контейнерно-управляемый компонент в конструкторы, методы и поля вне зависимости от модификаторов доступа, хотя поля не могут быть объявленными как final, а методы не должны быть абстрактными.

Возникают важные вопросы. Что произойдет при наличии нескольких реализаций интерфейса UserDataRepository? Как CDI-контейнер выберет для инъекции нужную реализацию? Для устранения этой неоднозначности между конкретными реализациями интерфейса UserDataRepository вы можете аннотировать конкретный класс, используя определяемый разработчиком квалификатор.

Представьте, что у вас есть две реализации интерфейса UserDataRepository: одна для коллекции базы данных Mongo (документоориентированная БД), другая — для базы данных MySQL (реляционная БД). Вам пришлось бы создать два квалификатора (один для реализации для Mongo, другой — для реализации для MySQL), конкретный класс был бы аннотирован на уровне класса соответствующим квалификатором, и в классе, в который должна быть осуществлена инъекция UserDataRepository, поле было бы аннотировано таким же квалификатором.

Если вы переделаете класс UserService из листинга 5.7 для использования реализации UserDataRepository для Mongo, вам придется добавить аннотацию @Mongo к полю udr, как показано ниже:

```
@Inject @Mongo
private UserDataRepository udr;
```

Использование квалификаторов более детально обсуждается далее, а также в главе 6.

Аннотация @Named

Еще одним замечательным достижением стало введение аннотации @Named вместо строковых квалификаторов. Неопределенности в зависимостях EJB были разрешены путем использования строкового значения в атрибуте beanName аннотации @ JB, которая задавала внедряемую реализацию: @EJB(beanName="UserDataRepository"). Аннотация @Named также поддерживает разрешение неопределенностей посредством использования атрибута String. В листинге 5.8 реализация UserDataRepository для Mongo внедряется в поле udr.

Листинг 5.8. Использование аннотации @Named для разрешения неопределенности

```
package com.devchronicale.di;

import javax.inject.Inject;
import javax.inject.Named;

class UserService {

    @Inject
    @Named("UserDataRepositoryMongo")
    private UserDataRepository udr;

    public void persistUser(User user) {
```

```
        udr.save(user);
    }
}
```

Соответствующим образом именованная аннотация `@Named` требует явного аннотирования реализации для Mongo. В листинге 5.9 реализация интерфейса `UserDataRepository` для Mongo аннотирована тем же строковым значением, которое использовано для разрешения неопределенности при инъекции реализации в листинге 5.8.

Листинг 5.9. Конкретная реализация требует аннотации `@Named`

```
package com.devchronicale.di;

import javax.inject.Named;

@Named("UserDataRepositoryMongo")
public class UserDataRepositoryMongo implements UserDataRepository {

    @Override
    public void save(User user) {
        // Здесь код сохранения
    }
}
```

Использование строк для идентификации зависимостей устарело (хотя все еще используется), поскольку не обеспечивает безопасность типов, да и спецификация CDI JSR-299 не рекомендует их применять. Однако есть вариант использования аннотации `@Named`, позволяющий избежать применения строковых идентификаторов в точке внедрения.

```
@Inject @Named
private UserDataRepository UserDataRepositoryMongo;
```

В листинге 5.9 имя внедряемой реализации выводится из имени поля `UserDataRepositoryMongo`. В сущности, `@Named("UserDataRepositoryMongo")` замещает аннотацию `@Named`.

Контекст и внедрение зависимостей (CDI)

Контекст и внедрение зависимостей (CDI) принесли платформе Java EE, бывшей до того связанной с EJB и гораздо более ограниченной, развитую систему внедрения зависимостей и поддержку контекста. После появления EJB 3 компания JBoss представила Seam (фреймворк для разработки веб-приложений), ставший довольно популярным за счет поддержки прямых взаимодействий между JavaServer Faces (JSF) и JavaBeans, а также EJB. Успех Seam привел к разработке JSR299 Web Beans («Веб-компоненты»)[1]. Как и Hibernate, знаменитый фреймворк постоянства Java, был вдох-

[1] Первоначальное наименование Web Beans («Веб-компоненты») было позднее изменено на Contexts and Dependency Injection for Java («Контексты и внедрение зависимостей для Java»), а затем и на Contexts and Dependency Injection for the Java EE Platform («Контексты и внедрение зависимостей для платформы Java EE»). См. https://jcp.org/en/jsr/detail?id=299. — *Примеч. пер.*

новлен стандартизацией Java Persistence API (JPA), так и Seam вдохновил и сформировал базовую реализацию CDI.

CDI может работать с любыми простыми Java-объектами в старом стиле (POJO) посредством инициализации и внедрения объектов друг в друга. Для внедрения доступны следующие типы объектов:

- объекты POJO;
- корпоративные ресурсы, такие как источники данных и очереди;
- удаленные EJB-ссылки;
- сеансовые компоненты;
- объекты `EntityManager`;
- ссылки на веб-сервисы;
- поля и объекты производителей данных, возвращаемые методами производителей данных.

CDI и EJB

Хотя CDI и EJB кажутся конкурентами, они сосуществуют вполне гармонично. CDI может функционировать в одиночку, без EJB-контейнера. Фактически CDI может обеспечивать работу приложения для настольных систем или любого веб-приложения, которое не зависит от EJB-контейнеров. CDI обеспечивает фабрику и внедрение в любой Java-компонент.

Тем не менее EJB-компонентам по-прежнему требуется EJB-контейнер. Даже простейшая архитектура EJB-компонентов сложнее, чем объекты POJO, так что EJB-компонентам по-прежнему нужен EJB-контейнер. EJB-контейнер обеспечивает дополнительные сервисы, такие как безопасность, транзакции и параллелизм, необходимые EJB-компонентам.

Проще говоря, CDI-контейнер — более «легкий», мощный, но менее функциональный контейнер для объектов POJO. Все же оба контейнера настолько хорошо интегрированы, что CDI-аннотации могут стать шлюзом и стандартным интерфейсом для взаимодействия с EJB-контейнером. Например, аннотация `@Inject` может работать как с объектами POJO, так и с EJB-компонентами и внедрить любую их комбинацию путем вызова соответствующего контейнера для выполнения работы.

Компоненты CDI

Контейнерно-управляемый компонент лишь немногим больше, чем просто POJO, подчиняющийся нескольким простым правилам.

- У него должен быть конструктор без параметров или же конструктор должен объявлять аннотацию `@Inject`.
- Класс должен быть конкретным классом верхнего уровня или должен быть аннотирован аннотацией `@Decorate`; он не может быть нестатическим внутренним классом.

- Он не может быть определен как EJB-компонент.
- Если компонент определен как управляемый компонент с помощью другой технологии Java EE, такой как технология JSF, он также может управляться контейнером.

Любой класс, удовлетворяющий этим требованиям, будет получать значение и управляться посредством контейнера и станет доступен для внедрения. Никакой специальной аннотации для определения класса как управляемого компонента не требуется.

Контейнер выполняет поиск архивов типа «компонент-в-компоненте». Существует два типа архивов компонентов: явные и неявные. Явный архив содержит дескриптор развертывания `bean.xml`, который обычно пуст. CDI просматривает классы в архиве в поисках какого-нибудь класса, подходящего под ранее детально описанные требования к компоненту, после чего обрабатывает и внедряет любой подобный класс, за исключением аннотированных аннотацией `@Vetoed`. Эта аннотация запрещает классу быть управляемым контейнером.

В некоторых случаях может быть нежелательно позволять контейнеру управлять любым найденным им подходящим компонентом. Если вы хотите ввести ограничения на то, что CDI-контейнер будет считать управляемыми компонентами, то можете определить свойство `bean-discovery-mode` в дескрипторе развертывания `bean.xml`. Листинг 5.10 показывает фрагмент файла `bean.xml`, определяющий свойство `bean-discovery-mode` как `ALL`.

Листинг 5.10. Установка режима обнаружения компонентов в `bean.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
    ...
</beans>
```

Свойство `bean-discovery-mode` может принимать одно из трех значений: `ALL`, `NONE` или `ANNOTATED`. Значение `ALL` указывает CDI-контейнеру управлять всеми найденными в архиве компонентами. Это значение по умолчанию. Значение `NONE` говорит о том, что CDI-контейнер не будет управлять никакими компонентами, а `ANNOTATED` приводит к тому, что архив будет вести себя как неявный. В последнем случае контейнер ищет компоненты с аннотированными типами области видимости.

Неявный архив компонентов не содержит дескриптора развертывания `bean.xml`. Это указывает CDI-контейнеру, что контейнер должен управлять только компонентами с областью видимости. Дальнейшие подробности о компонентах с областью видимости обсуждаются в подразделе «Контексты и области видимости».

Аннотация @Inject

Аннотация `@Inject` и ее возможности уже были описаны ранее. До появления CDI в платформе Java EE каждый DI-фреймворк предлагал собственный способ

внедрения ресурсов. Когда Java EE CDI-контейнер был выпущен для работы бок о бок с EJB-контейнером, аннотация `@Inject` стала единственным и абстрактным интерфейсом для практически всех операций по внедрению. Аннотация `@Inject` позволяет вам использовать любой подходящий контейнер или DI-фреймворк.

Контексты и области видимости

Именно в контексте заключается различие между EJB- и CDI-контейнерами. Жизненный цикл каждого CDI-компонента привязан к области видимости контекста. CDI предлагает четыре различные области видимости:

- `@RequestScoped` — длительность жизни — пользовательский HTTP-запрос;
- `@SessionScoped` — длительность жизни — пользовательский HTTP-сеанс;
- `@ApplicationScoped` — состояние совместно используется всеми пользователями на время длительности жизни приложения;
- `@ConversationScoped` — длительность жизни области видимости управляется разработчиком.

Компонент, аннотированный областью видимости, сохраняет состояние на время длительности жизни области видимости и использует его вместе с любым запущенным в той же области видимости клиентом. Например, компонент в области видимости запроса сохраняет состояние на время жизни HTTP-запроса, а компонент с областью видимости сеанса сохраняет состояние на время жизни HTTP-сеанса. Компонент с областью видимости автоматически создается в нужный момент и уничтожается, когда заканчивается контекст, в котором он участвует.

Аннотации области видимости часто применяются для определения области видимости компонентов, используемых в языке выражений (Expression Language, EL) на веб-страницах фреймворка Facelets.

Именование и EL

Компонент, аннотированный `@Named`, доступен через EL. По умолчанию имя, используемое в выражении, — это имя класса с первой буквой в нижнем регистре. Для ссылки на `getter`-методы, начинающиеся с `get` или `is`, опустите часть `get` или `is` имени метода. Листинг 5.11 демонстрирует пример этого.

Листинг 5.11. Аннотация `@Named` делает компонент видимым для EL

```
package com.devchronicale.di;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named // Определяет, что это управляемый компонент
@RequestScoped // Определяет область видимости
public class  User {
```

```
    private String fullName;

    public String getFullName(){
        return this .fullName;
    }

// Некоторые методы опущены для краткости
}
```

Это простая реализация именованного компонента, возвращающего строку при вызове метода `getFullName()`. На веб-странице Facelets вам нужно было бы ссылаться на этот метод как на `user.fullname`.

```
<h:form id="user">
    <p><h:outputText value="#{user.fullname}"/></p>
</h:form>
```

CDI-компоненты для управляемых JSF

Как показано в предыдущем примере, CDI-компоненты могут служить управляемыми компонентами для страниц JSF. Вы можете обращаться к поименованным компонентам через имя компонента со строчной первой буквой. Вы можете обращаться к полям и методам `Getter/Setter` внутри страниц JSF, используя соглашения Java. Детальное рассмотрение JSF выходит за пределы этой книги, однако листинг 5.11 демонстрирует элементарное использование CDI-компонентов с JSF.

Квалификаторы

Этот раздел показывает, как вы могли бы конструировать классы квалификаторов.

В листинге 5.12 вы создаете квалификатор с именем `Mongo`, который можете использовать для аннотирования полей. Если вы хотите применять эту аннотацию с `METHOD`, `PARAMETER` или с классом/интерфейсом (`TYPE`), то можете добавить соответствующий параметр к аннотации `@Target`.

Листинг 5.12. Создание пользовательского квалификатора с именем `@Mongo`

```
package com.devchronicale.di;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
```

```
@Target({FIELD})
public @interface Mongo {}
```

Дискуссия относительно различных вариантов использования аннотаций будет более детально продолжена в главе 6.

Альтернативы

В приведенных до сих пор примерах вы увидели, как можно, используя квалификаторы, разрешать неоднозначность между двумя различными реализациями интерфейса `UserDataRepository`. Обычно вы делаете этот выбор реализации во время разработки путем изменения исходного кода. Однако вы можете также сделать его во время развертывания системы путем использования аннотации `@Alternative` и некоторых конфигурационных изменений в дескрипторе развертывания `bean.xml`.

Переделывая до сих пор примеры, вы аннотировали две реализации интерфейса `UserDataRepository` и добавляли кое-какие конфигурационные изменения в файл `bean.xml`. Именно здесь вы решаете, какую реализацию внедрять.

```
@Alternative
public class UserDataRepositoryMongo implements UserDataRepository {...}

@Alternative
public class UserDataRepositoryMySQL implements UserDataRepository {...}
```

Реализация, используемая вами в приложении, объявляется в файле `bean.xml`.

```
<beans ...>
    <alternatives>
    <class>com.devchronicale.di.UserDataRepositoryMongo</class>
    </alternatives>
</beans>
```

Альтернативы часто применяются при разработке во время этапа испытаний для создания объектов-имитаций.

Стереотипы

Вы можете рассматривать стереотипы как шаблоны, определяющие характеристики некоторой функциональности типа компонента. Например, компонент, используемый в слое модели приложения, основанного на паттерне «Модель — представление — контроллер» (Model — View — Controller, MVC), требует определенных аннотаций для выполнения своих функций. Они могли бы включать следующее:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
```

Для определения компонента «Модель» достаточно лишь `@Named` и `@RequestScoped`. Остальные необходимы для создания аннотации `@Model`.

Вы можете применить эти аннотации к каждому компоненту, который их требует, или определить стереотип с именем `@Model` и применить к компонентам только его. Последний из перечисленных вариантов делает ваш код более легким для чтения и поддержки.

Чтобы создать стереотип, вы определяете новую аннотацию и применяете необходимые аннотации, как показано в листинге 5.13.

Листинг 5.13. Аннотация стереотипа

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface Model {}
```

Любой компонент, аннотированный аннотацией `@Model`, имеет область видимости запроса (`@RequestScoped`) и видим для EL (`@Named`). К счастью, сопутствующий этому стереотипу CDI-контейнер уже был определен.

Типичное использование аннотации стереотипа — сочетание с аннотацией альтернативы, что дает вам способ аннотации имитационных объектов.

Другие паттерны посредством CDI

CDI предоставило для Java EE-разработчиков огромные возможности: оно вышло за пределы простого DI-фреймворка, сделав возможным реализацию всех тех паттернов с минимальным количеством кода.

В следующих главах мы рассмотрим эти паттерны более детально; однако, чтобы разжечь ваш аппетит, дадим краткое введение в эти существующие за счет CDI паттерны.

Глава 7 охватывает паттерн «Декоратор». Декораторы оборачивают целевой объект для динамического добавления новых обязанностей во время выполнения. Каждый декоратор может быть обернут другим, что допускает неограниченное теоретически количество декорированных целевых объектов во время выполнения. Паттерн «Декоратор» использует аннотации `@Decorator` и `@Delegate`. Вы должны определить порядок декорации в `bean.xml`.

Паттерн «Фабрика» рассматривается в главе 6. Фабрики минимизируют использование ключевого слова `new` и могут инкапсулировать процесс инициализации и различные конкретные реализации. Паттерн «Фабрика» применяет аннотацию `@Produces`, чтобы отмечать методы производителя данных. Целевой объект может внедрять или наблюдать произведенные объекты.

Паттерн «Наблюдатель» и события исследуются в главе 11. Этот паттерн меняет направление сообщения, то есть порядок вызывающего и вызываемого. С помощью паттерна «Наблюдатель», вместо того чтобы активно проверять ресурс, объект может просто «подписаться» на оповещения о его изменениях. Целевой наблюдатель (-ли) может видеть любое сработавшее событие.

Аспекты и перехватчики — в центре внимания главы 8. Они позволяют вам менять поток выполнения во время самого процесса выполнения. Любой аспект

или перехватчик может быть помечен для прерывания выполнения и перехода к заданной точке. Такой подход позволяет запускать динамические изменения даже на большой базе кода.

Резюме

В этой главе вы познакомились с концепцией внедрения зависимостей на платформе Java EE. Эта концепция позволяет нам строить слабосвязанные системы легче, чем можно было бы себе представить. Мы рассмотрели, как внедрение зависимостей позволяет нам исключить использование ключевого слова `new`, а значит, и создание объекта «вручную».

Мы сосредоточили наше внимание на CDI, которое высвобождает огромный потенциал, предоставляя совершенно новый контейнер. С помощью CDI к любому объекту может быть применено внедрение зависимостей и легко реализованы многие из обсуждаемых в этой книге паттернов.

Упражнения

1. Разработайте класс-сервис, возвращающий строку клиенту.
2. Реализуйте программу считывания файлов и внедрите ее в разработанный ранее сервис.
3. На этот раз реализуйте объект, считывающий HTML-содержимое как строку с указанного URL.
4. Обдумайте, что вам понадобится переделать в классе сервиса, чтобы он мог внедрять оба провайдера данных по одинаковой ссылке.
5. Существует ли способ динамически внедрять каждую реализацию в зависимости от различных обстоятельств? Например, можете ли вы обеспечить, чтобы программа считывания файлов внедрялась во время разработки, а программа считывания HTML использовалась во время введения в эксплуатацию?

6 Паттерн «Фабрика»

В этой главе:

- что такое паттерн «Фабрика» и зачем он вам нужен;
- как реализовать различные варианты паттерна: фабричный метод и абстрактную фабрику;
- как реализовать паттерн «Фабрика» в Java EE, используя аннотации `@Produces` и `@Inject`;
- как создать пользовательские аннотации и `@Qualifier` для устранения неоднозначности между конкретными реализациями.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 06.zip, каждый из них назван в соответствии с наименованиями в тексте.

Паттерн проектирования «Фабрика» — один из широко используемых базовых паттернов проектирования в современных языках программирования. Он применяется не только веб-программистами и разработчиками приложений, но и разработчиками сред выполнения и фреймворков, таких как Java и Spring.

У паттерна «Фабрика» есть две разновидности: фабричный метод и абстрактная фабрика. Цель этих паттернов одна: обеспечить интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов без уточнения их конкретных классов. Эта глава знакомит вас с обеими разновидностями и демонстрирует примеры их реализации.

Вы увидите, как паттерн «Фабрика» был реализован на платформе Java SE, в чем отличия от реализации на платформе Java EE и как он использует в своих интересах контекст и внедрение зависимостей.

Что такое фабрика

Цель фабрики как одного из порождающих паттернов — в создании объектов. Порождающая логика инкапсулирована внутри фабрики и предоставляет метод, возвращающий недавно созданный объект (паттерн «Фабричный метод»), или делегирует создание объекта подклассу (паттерн «Абстрактная фабрика»).

В обоих случаях создание объекта отделяется от места его будущего использования.

Клиенту не нужно быть осведомленным о различных реализациях интерфейса или класса. Клиенту только нужно знать, какую фабрику (фабричный метод или абстрактную фабрику) применять для получения экземпляра одной из реализаций интерфейса. Клиенты разграничиваются с созданием объектов.

Разграничение происходит как результат применения принципа инверсии зависимостей и приносит немало практических выгод, важнейшая из которых — расцепление высокоуровневых классов с низкоуровневыми. Оно позволяет изменять реализации конкретных классов, не затрагивая клиента и таким образом снижая степень связи между классами и повышая гибкость.

Паттерн «Фабрика» дает нам возможность расцеплять создание объекта с основной системой посредством инкапсуляции кода, ответственного за создание объектов. Такой подход упрощает нашу жизнь, когда дело доходит до рефакторинга, так как теперь у нас есть единственное место, где будут происходить изменения.

Часто сама фабрика реализована как одиночка или статический класс, поскольку в обычных условиях требуется только один экземпляр фабрики. Это централизует создание объекта фабрики, что дает при выполнении изменений и обновлений лучшую структурную организацию и удобство сопровождения и уменьшает количество ошибок.

ПРИМЕЧАНИЕ

Принцип инверсии зависимостей следующий[1].

1. **Высокоуровневые модули не должны зависеть от низкоуровневых модулей. И те и другие должны зависеть от абстракций.**
2. **Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.**

В Java EE внедрение зависимости употребляется, чтобы разграничить высокоуровневые классы с низкоуровневыми при реализации паттерна «Фабрика». Совместное использование аннотаций `@Produces` и `@Inject` делает их реализацию относительно несложной.

Фабричный метод

Книга GoF описывает фабричный метод так: «Определяет интерфейс для создания классов, но оставляет подклассам решение о том, какой класс инстанцировать». «Паттерны проектирования» добавляет, что «фабричный метод делегирует операцию создания экземпляра субклассам».

Фабрики минимизируют использование ключевого слова `new` и могут инкапсулировать процесс инициализации и различные конкретные реализации. Способность к централизации этих потребностей минимизирует воздействие добавления или удаления конкретных классов в систему или из нее и воздействие зависимостей конкретных классов.

[1] http://ru.wikipedia.org/wiki/Принцип_инверсии_зависимостей.

Диаграмма классов фабричного метода показана на рис. 6.1.

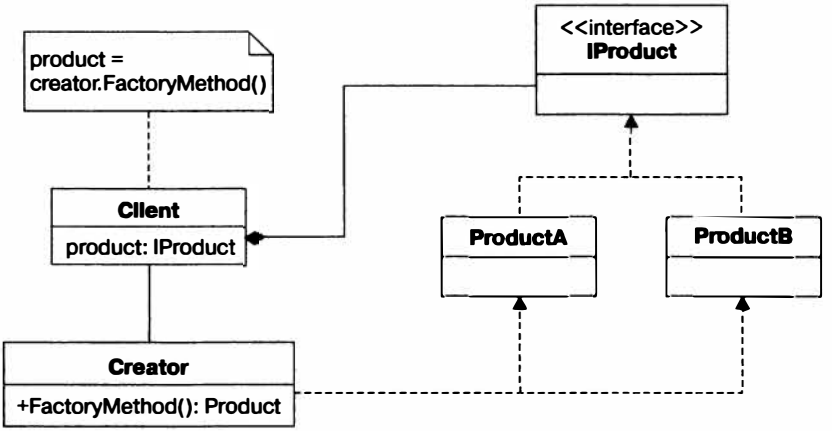


Рис. 6.1. Диаграмма классов показывает структуру паттерна «Фабричный метод». Вы можете видеть, как создание объекта инкапсулировано в подклассах

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Когда-то мы с другом разрабатывали приложение для настольной системы. Поначалу мы не были уверены, нужна ли нам развитая база данных. Это было задолго до NoSQL и документоориентированных хранилищ данных, так что нашим единственным вариантом было использование XML. Тем не менее мы не были уверены, будет ли достаточно для хранения наших данных XML-файлов. Однако тем временем мы уже начали разработку приложения и нуждались в конкретной реализации механизма хранения данных. Поскольку нам хотелось гибкости, чтобы иметь возможность перейти от XML к SQL, мы решили выполнить все объекты доступа к данным (DAO) в виде реализации паттерна «Фабрика». Таким образом, мы могли легко переключиться с XML на SQL или наоборот. Через несколько недель мы осознали, что сильно недооценили потребности нашей системы в данных. XML определенно не отвечал требованиям, и о нем теперь не могло идти и речи, так что мы до конца проекта были связаны с основанной на SQL базой данных. В конечном счете мы так и не использовали по-настоящему нашу фабрику-DAO.

Однако когда мы завершали разработку приложения, наши заказчики попросили о демонстрационной платформе. Было недостаточно показать возможности приложения и позволить им «поиграться» день или два. Им нужно было больше времени, чтобы оценить приложение. Это значило, что от нас требовалось установить работающее приложение в Сети, чтобы дать заказчикам время должным образом оценить его и убедиться, что оно работает. Мы не хотели устанавливать полнофункциональное приложение, так как у нас не было способа гарантировать, что заказчики не сделают копию, и мы определенно не хотели создавать демонстрационное приложение с нуля. Внезапно мне пришла в голову гениальная мысль. Демонстрационное приложение должно хранить данные достаточно хорошо, чтобы заказчик мог его оценить, но недостаточно хорошо для того, чтобы заказчик мог сделать пиратскую копию приложения. Идея заключалась во временном хранении данных в оперативной памяти. Если бы мы смогли легко поменять наши DAO для хранения данных в оперативной памяти вместо сохранения их в базе данных, то могли бы позволить заказчикам пробовать демоверсию столько, сколько они захотят. (Без постоянного хранилища данных приложение вообще не имело бы смысла!) Поскольку у нас уже была фабрика-DAO, нам нужно было только реализовать классы DAO для хранения в оперативной памяти и модифицировать код фабрики, чтобы они возвращались при отсутствии базы данных.

Результат был настолько удачен, что я реализовал еще одну фабрику для управления заданиями на печать, чтобы для демоверсии печатать в неформатированный текстовый файл вместо реального принтера. Эти изменения, использующие выгоды паттерна «Фабрика», означали, что мы могли легко разрешить заказчикам оценивать наше приложение столько, сколько они хотели, и, поскольку

заказчики не могли печатать форматированные копии и сохранять финансовую информацию, приложение было бесполезно при коммерческой эксплуатации.

Проектирование системы с использованием фабрик не выглядело на первых порах большой победой, но в конечном счете стало настоящим «спасательным кругом». Паттерны проектирования имеют тенденцию решать будущие проблемы, если, конечно, они используются в правильном контексте.

Реализация фабричного метода в простом коде. Фабричный метод не имеет шаблонного кода для своих реализаций. Листинги 6.1–6.6 демонстрируют его реализации с использованием автомата по розливу напитков DrinksMachine, который наливает различные виды напитков в зависимости от реализации его подклассов.

Листинг 6.1. Абстрактный класс DrinksMachine, расширяемый конкретными реализациями

```
public abstract class DrinksMachine {

    public abstract Drink dispenseDrink();

    public String displayMessage(){
        return "Thank for your custom.";
    }
}
```

Листинг 6.2. Реализация CoffeeMachine абстрактного класса DrinksMachine

```
public class CoffeeMachine extends DrinksMachine {

    public Drink dispenseDrink() {
        return new Coffee();
    }
}
```

Листинг 6.3. Реализация SoftDrinksMachine абстрактного класса DrinksMachine

```
public class SoftDrinksMachine extends DrinksMachine {

    public Drink dispenseDrink() {
        return new SoftDrink();
    }
}
```

Листинг 6.4. Интерфейс Drink

```
public interface Drink {}
```

Листинг 6.5. Реализация SoftDrink интерфейса Drink

```
public class SoftDrink implements Drink {
    SoftDrink() {
        System.out.println("Soft drink");
    }
}
```

Листинг 6.6. Реализация Coffee интерфейса Drink

```
public class Coffee implements Drink {
    Coffee() {
        System.out.println(„Coffee");
    }
}
```

Эта реализация демонстрирует, как подклассы абстрактного класса DrinksMachine определяют, какой напиток наливается. Это позволяет любой реализации класса DrinksMachine «наливать» любой объект типа Drink. Каждый подкласс абстрактного класса DrinksMachine определяет, какие напитки наливаются.

Это была простая реализация, в которой метод dispenseDrink наливает только один вид напитка. Более наглядным был бы пример, демонстрирующий получение методом dispenseDrink наименования напитка и затем конструирующим и возвращающим объект запрошенного напитка. Листинг 6.7 показывает, как этого добиться.

Листинг 6.7. Перечислимый тип CoffeeType

```
public enum CoffeeType {EXPRESSO. LATTE}

public Drink dispenseDrink(CoffeeType type) {
    Drink coffee = null;
    switch (type) {
        case EXPRESSO: coffee = new Expresso();
        case LATTE: coffee = new Latte();
    }
    return coffee;
}
```

Для краткости этот раздел показывает только фрагмент кода перечислимого типа CoffeeType, определяющего разновидности кофе, и метод dispenseDrink конкретного класса Coffee.

Абстрактная фабрика

Паттерн «Фабричный метод» прямолинеен и полезен в реализации, но в более сложных системах вам нужно будет его упорядочивать. Эта проблема приводит вас к новому паттерну, который называется «Абстрактная фабрика».

Паттерн «Абстрактная фабрика» описан как в книге GoF, так и «Паттернах проектирования» как «предоставляющий интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов».

Что абстрактная фабрика предлагает, так это инкапсуляцию группы фабрик и контроль над обращением к ним клиента. Эта глава не содержит все подробности того, как реализовывать абстрактные фабрики, но взамен предлагает краткое введение для понимания сути.

Диаграмма классов абстрактной фабрики показана на рис. 6.2.

Реализация абстрактной фабрики в простом коде. Для демонстрации паттерна проектирования «Абстрактная фабрика» в этом разделе мы расширим пример автомата по разливу напитков путем добавления фабрики, порождающей два различных типа автоматов: базовый и для гурманов.

«Семейства взаимосвязанных или взаимозависимых объектов», создаваемые абстрактной фабрикой, — это кофейный автомат и автомат по розливу безалкогольных напитков. Вам необходим интерфейс, который фабрики будут реализовывать. В листинге 6.8 вы создаете интерфейс AbstractDrinksMachineFactory.

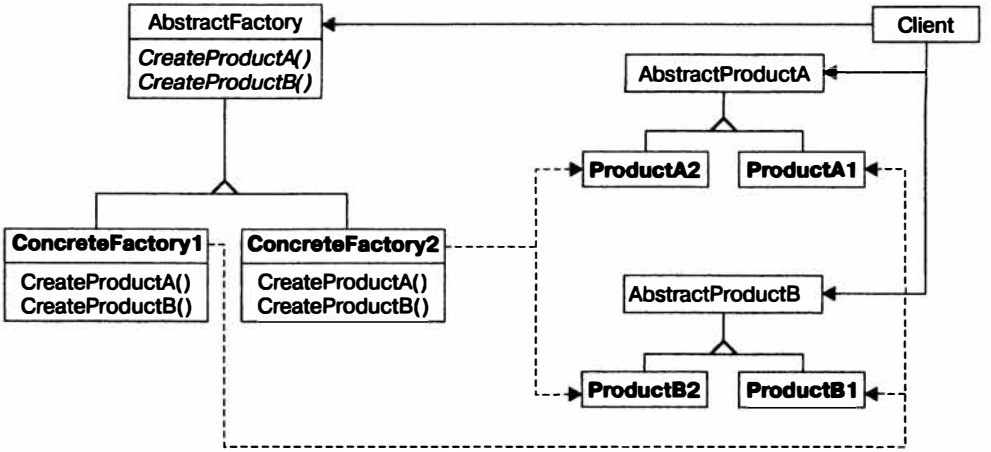


Рис. 6.2. Вы можете использовать паттерн «Абстрактная фабрика» для группировки существующих фабрик и инкапсуляции того, как вы к ним обращаетесь

Листинг 6.8. Интерфейс для абстрактной фабрики

```
public interface AbstractDrinksMachineFactory {
    public DrinksMachine createCoffeeMachine();
    public DrinksMachine createSoftDrinksMachine();
}
```

Конкретные реализации этого интерфейса — классы `GourmetDrinksMachineFactory` и `BasicDrinksMachineFactory`. Для краткости в листинге 6.9 показан только `GourmetDrinksMachineFactory`.

Листинг 6.9. Реализация AbstractDrinksMachineFactory

```
public class GourmetDrinksMachineFactory implements AbstractDrinksMachineFactory{
    public DrinksMachine createCoffeeMachine() {
        return new GourmetCoffeeMachine();
    }

    public DrinksMachine createSoftDrinksMachine() {
        return new GourmetSoftDrinksMachine();
    }
}
```

Каждая фабрика реализует метод `create` абстрактной фабрики по-разному, и в зависимости от того, экземпляр какой фабрики создается, получается различная реализация кофейного автомата и автомата по разливу безалкогольных напитков.

```
AbstractDrinksMachineFactory factory = new GourmetDrinksMachineFactory();
DrinksMachine CoffeeMachine = factory.createCoffeeMachine();
CoffeeMachine.dispenseDrink(CoffeeType.EXPRESSO);
```

Здесь демонстрируется создание экземпляра GourmetDrinksMachineFactory. Вызывается его метод создания кофейного автомата, чтобы сформировать нужный этой реализации объект кофейного автомата.

Полный код этой реализации можно найти в загружаемых файлах для главы 6.

Реализация абстрактной фабрики в Java EE

Паттерн «Фабрика» несложен в реализации, как вы уже видели в предыдущих примерах. Платформа Java EE предлагает простой и изящный способ реализации этого паттерна с помощью аннотаций и внедрения зависимостей. В мире Java EE вы используете аннотацию @Produces для создания объекта и @Inject для внедрения созданного объекта (или ресурса) там, где он требуется. Простейшая реализация фабричного метода на платформе Java EE показана в листинге 6.10.

Листинг 6.10. Простая реализация фабричного метода с использованием методов производителя данных

```
package com.devchronicles.producer;

import javax.enterprise.inject.Produces;

public class EventProducer {

    @Produces
    public String getMessage() {
        return "Hello World";
    }
}
```

Метод getMessage аннотирован аннотацией @Produces и возвращает строковые объекты, содержащие текст Hello world!. Хотя произведенный в этом примере тип — строка, вы можете производить все, что вам требуется, включая интерфейсы, классы, простые типы данных, Java-массивы и базовые типы языка Java.

Чтобы использовать произведенный объект, вам нужно внедрить тот же тип в класс, где вы собираетесь его применять, как показано в листинге 6.11.

Листинг 6.11. Внедрение созданной фабрикой строки

```
package com.devchronicles.factory;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.inject.Inject;

@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class EventService {

    @Inject
```

```
    private String message;

    public void startService() {
        System.out.println("Start service call " + message);
    }
}
```

Когда вы запускаете и вызываете метод startService, строковое значение в методе производителя данных внедряется в член message класса EventService и выводится в консоль. Это простейшая возможная реализация фабричного метода на платформе Java EE. Однако возникает один важный вопрос: откуда CDI-контейнер «знает», что он должен внедрить строку, произведенную методом getMessage, в член message класса EventService?

Ответ: CDI-контейнер полагается на типы для определения, куда внедрить произведенный тип. В этом примере произведенный тип — строка, как и внедряемый тип. Так что CDI-контейнер сопоставляет произведенный тип с внедряемым типом и внедряет его.

Вы можете возразить, что в реальной системе необходимо производить и внедрять различные экземпляры одного и того же объектного типа. Как CDI-контейнер «знает», куда внедрять каждый из произведенных типов? А знает он благодаря использованию конфигурации аннотации, которая называется *квалификатором*.

В реальных проектах вы, вероятно, захотите возвращать различные объектные типы вместо простой строки, так что можете создавать различные объекты по типу. Рассмотрим листинги 6.12–6.15.

Листинг 6.12. Компонент MessageA

```
package com.devchronicles.factory;

import javax.enterprise.inject.Alternative;

@Alternative
public class MessageA {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Листинг 6.13. Компонент MessageB

```
package com.devchronicles.factory;

import javax.enterprise.inject.Alternative;

@Alternative
public class MessageB {
```

```
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Листинг 6.14. Реализация фабрики, создающей компоненты-сообщения

```
package com.devchronicles.factory;

import javax.enterprise.inject.Produces;

public class EventProducer {
    @Produces
    public MessageA messageAFactory() {
        return new MessageA();
    }

    @Produces
    public MessageB messageBFactory() {
        return new MessageB();
    }
}
```

В этом примере вы создали два компонента: MessageA в листинге 6.12 и MessageB в листинге 6.13. Вы аннотировали их аннотацией @Alternative, которая отключает их, так что контейнер не пытается внедрить их экземпляры при нахождении подходящей точки внедрения. Вы аннотируете их таким образом, что фабрика в листинге 6.14 произведет их экземпляры. Если бы вы не указали эту аннотацию, контейнер сгенерировал бы исключение при загрузке приложения. Оно выглядело бы приблизительно так:

```
CDI deployment failure:WELD-001409 Ambiguous dependencies for type [MessageA]
```

Неоднозначность в том, что созданы два экземпляра MessageA: один — контейнером и другой — методом @Produces. Контейнер «не знает», какой экземпляр внедрять в член message класса EventService. Далее в этом разделе вы увидите способ разрешения этой неоднозначности.

Листинг 6.15. Внедрение компонентов, созданных фабрикой, с использованием аннотации @Inject

```
package com.devchronicles.factory;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.enterprise.event.Event;
import javax.inject.Inject;
```

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class EventService {

    @Inject
    private MessageA messageA;

    @Inject
    private MessageB messageB;

    public void startService() {
        messageA.setMessage("This is message A");
        messageB.setMessage("This is message B");

        System.out.println("Start service call " + messageA.getMessage());
        System.out.println("Start service call " + messageB.getMessage());
    }
}
```

В показанном в листинге 6.15 классе EventService контейнеры внедряют два компонента, произведенных фабрикой, в переменные экземпляра messageA и messageB класса EventService. Вы можете использовать эти объекты, как делали бы это при обычных условиях.

Альтернативная реализация — использование аннотаций @Qualifier и @interface, чтобы пометить тип, который вы хотите внедрить. В следующем примере задействованы пользовательские аннотации для создания двух квалификаторов: @LongMessage в листинге 6.16 и @ShortMessage в листинге 6.17.

Листинг 6.16. Квалификатор @ShortMessage

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD, ElementType.FIELD })
public @interface ShortMessage {}
```

Листинг 6.17. Квалификатор @LongMessage

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD, ElementType.FIELD })
public @interface LongMessage {}
```

Далее вы используете эти квалификаторы для аннотирования методов производителя данных, как показано в листинге 6.18, и их подходящих точек внедрения, как показано в листинге 6.19.

Листинг 6.18. Использование квалификаторов для разрешения неоднозначности между компонентами

```
public class EventProducer {

    @Produces @ShortMessage
```

```
    private MessageA messageAFactory(){
        return new MessageA();
    }

    @Produces @LongMessage
    private MessageB messageBFactory(){
        return new MessageB();
    }

}
```

Листинг 6.19. Внедрение созданных компонентов с использованием квалификаторов для разрешения неоднозначности

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class ClientMessage {

    @Inject @ShortMessage
    private MessageA messageA;

    @Inject @LongMessage
    private MessageB messageB;

    public void doEvent() {
        messageA.setMessage("This is a long email message.");
        messageB.setMessage("This is a short SMS message.");
        System.out.println(messageA.getMessage());
        System.out.println(messageB.getMessage());
    }
}
```

Аннотация `@Target`, заданная в интерфейсе квалификатора, определяет, где вы можете использовать квалификатор. Значение может быть одним из следующих: `TYPE`, `METHOD`, `FIELD` и `PARAMETER` — и их смысл понятен без пояснений.

В качестве другого варианта вы можете получить ту же реализацию, используя перечислимый тип, определенный в классе `@interface`. Листинг 6.20 демонстрирует эту реализацию.

Листинг 6.20. Пользовательский тип аннотации

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD })
public @interface MyEvent {
    Type value();
    enum Type { LOGGING, MESSAGE }
}
```

С помощью этой пользовательской аннотации вы можете применять различные методы для создания строковых объектов, помеченных вашей аннотацией. В листинге 6.21 строки производятся методами `messageAFactory` и `messageBFactory`.

Листинг 6.21. Применение пользовательских аннотаций для разрешения неоднозначности между компонентами

```
public class EventProducer {

    @Produces
    @MyEvent(MyEvent.Type.LOGGING)
    public String messageAFactory() {
        return "A message";
    }

    @Produces
    @MyEvent(MyEvent.Type.MESSAGE)
    public String messageBFactory() {
        return "Another message";
    }
}
```

Теперь вы используете эти аннотации для аннотирования методов производителя данных и их подходящих точек внедрения, как показано в листинге 6.22.

Листинг 6.22. Внедрение созданных компонентов с помощью пользовательских аннотаций для разрешения неоднозначности

```
package com.devchronicles.observer;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import com.devchronicles.observer.MyEvent;
import javax.inject.Inject;

@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class EventService {

    @Inject
    @MyEvent(MyEvent.Type.LOGGING)
    private String messageA;

    @Inject
    @MyEvent(MyEvent.Type.MESSAGE)
    private String messageB;

    public void startService() {
        System.out.println("Start service call " + messageA);
        System.out.println("Start service call " + messageB);
    }
}
```

Проще было бы использовать аннотацию @Named, а не создавать свой собственный тип аннотации. Это реализовано в листинге 6.23.

Листинг 6.23. Использование аннотаций @Named для разрешения неоднозначности

```
package com.devchronicles.factory;

import javax.enterprise.inject.Produces;
import javax.inject.Named;

public class EventProducer {
    @Produces
    @Named("Logging")
    public String messageAFactory() {
        return "A message";
    }

    @Produces
    @Named("Message")
    public String messageBFactory() {
        return "Another message";
    }
}
```

Далее вы используете @Named для аннотирования методов производителя данных и их подходящих точек внедрения, как показано в листинге 6.24.

Листинг 6.24. Внедрение с использованием аннотаций @Named

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class EventServiceName {

    @Inject
    @Named("Logging")
    private String messageA;

    @Inject
    @Named("Message")
    private String messageB;

    public void startService() {
        System.out.println("Start service call " + messageA);
        System.out.println("Start service call " + messageB);
    }
}
```

Хотя это кажется более простым, чем создание вашего собственного типа аннотации, в сложных системах это может оказаться не самым разумным или не самым удачным с точки зрения безопасности типов вариантом. Аннотация @Named работает со строками, указываемыми в кавычках, и далека от обеспечения безопасности типов. Компилятор не сможет предупредить вас о возможных ошибках.

Обуздание мощи CDI. Если у вашего приложения много реализаций интерфейса и вы хотите реализовать паттерн «Фабрика» для производства требуемых экземпляров класса, то можете в итоге получить класс фабрики с множеством

методов, аннотированных аннотацией @Produces. Такой код будет слишком «многословным» и сложным для обслуживания. К счастью, платформа Java EE обеспечивает решение в виде аннотации @Any и оригинального использования перечислимых типов, символьных констант в аннотациях и класса Instance.

То, что потребовало бы многих десятков, если не сотен строк кода для производства каждого экземпляра, вы можете сделать лишь четырьмя строкам кода. Вы можете достичь этого собиранием всех экземпляров данной реализации интерфейса и выбором того, который хотите использовать, с помощью аннотации @Any.

Аннотация @Any указывает контейнеру, что все компоненты, реализующие данный интерфейс, должны быть внедрены в этой точке внедрения. В листинге 6.30 код private Instance<MessageType>. messages внедряет экземпляры всех зависимостей, реализующих интерфейс MessageType, в переменную экземпляра messages.

Как только все зависимости внедрены, вам понадобится способ отличать их друг от друга и выбирать ту, которую вы хотите применять. Именно тут дает знать о себе польза от символьных констант в аннотациях и перечислимых типов. В листинге 6.31 вы определяете квалификатор @Message и перечислимые символьные константы SHORT и LONG, чтобы различать реализации интерфейса MessageType.

Чтобы выбрать зависимость, сравните ее с перечислимым типом квалификатора каждой реализации путем создания AnnotationLiteral типа, который вы ищете, получите его и верните клиенту.

А сейчас вы увидите, как это реализовано в коде (листинги 6.25–6.31). Воспользуемся примером фабрики, производящей объекты ShortMessage и LongMessage, а каждая реализация интерфейса MessageType будет аннотирована как SHORT или как LONG.

Листинг 6.25. Интерфейс MessageType

```
public interface MessageType {
    public String getMessage();
    public void setMessage(String message);
}
```

Листинг 6.26. ShortMessage-реализация интерфейса сообщений

```
@Message(Message.Type.SHORT)
@Dependent
public class ShortMessage implements MessageType {

    private String message;

    @Override
    public String getMessage() {
        return message;
    }

    @Override
    public void setMessage(String message) {
        this.message = message;
    }
}
```

Листинг 6.27. LongMessage-реализация интерфейса сообщений

```
@Message(Message.Type.LONG)
@Dependent
public class LongMessage implements MessageType {

    private String message;

    @Override
    public String getMessage() {
        return message;
    }

    @Override
    public void setMessage(String message) {
        this.message = message;
    }
}
```

Каждая конкретная реализация интерфейса MessageType, как показано в листинге 6.25, аннотирована квалификатором @Message, указывающим тип сообщения как Message.Type.SHORT или Message.Type.LONG, что реализовано в листингах 6.26 и 6.27 соответственно. Как можно видеть в листинге 6.28, квалификатор @Message реализован таким же образом, как и квалификатор, задействованный в показанном ранее примере пользовательского типа аннотации.

Листинг 6.28. Пользовательская аннотация сообщения

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE})
public @interface Message {
    Type value();
    enum Type{ SHORT, LONG }
}
```

Для создания символьных констант аннотаций, которые вы используете для сравнения между нужным вам типом и типом зависимости, придется расширить абстрактный класс AnnotationLiteral и реализовать Message как пользовательский квалификатор сообщения. Листинг 6.29 демонстрирует, как это сделать.

Листинг 6.29. Символьная константа аннотации, используемая для получения требуемого типа сообщения

```
public class MessageLiteral extends AnnotationLiteral<Message> implements Message {
    private static final long serialVersionUID = 1L;
    private Type type;

    public MessageLiteral(Type type) {
        this.type = type;
    }

    public Type value() {
        return type;
```

```
    }
}
```

Теперь, когда у вас есть все части головоломки, вы можете собрать их воедино в классе MessageFactory, показанном в листинге 6.30.

Листинг 6.30. Реализация фабрики

```
@Dependent
public class MessageFactory {

    @Inject
    @Any
    private Instance<MessageType> messages;

    public MessageType getMessage(Message.Type type) {
        MessageLiteral literal = new MessageLiteral(type);
        Instance<MessageType> typeMessages = messages.select(literal);
        return typeMessages.get();
    }
}
```

В классе фабрики все зависимости, реализующие интерфейс MessageType, внедряются в переменную экземпляра messages. Затем из метода getMessage вы берете параметр типа Message.Type для создания новой символьной константы MessageLiteral, используемой вами для указания реализации MessageType, которую вы хотите получить от messages и которая, в свою очередь, возвращается клиенту.

Клиент внедряет фабрику и вызывает метод getMessage, передавая требуемое им в Message.Type, как можно видеть в листинге 6.31.

Листинг 6.31. Клиент, использующий реализацию фабрики

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
@ApplicationScoped
public class Client {

    @Inject
    MessageFactory mf;

    public void doMessage(){
        MessageType m = mf.getMessage(Message.Type.SHORT);
        m.setMessage("This is a short message");
        System.out.println(m.getMessage());

        m = mf.getMessage(Message.Type.LONG);
        m.setMessage("This is a long message");
        System.out.println(m.getMessage());
    }
}
```

В этом разделе мы значительно отошли от первоначальной GoF-реализации паттерна «Фабрика». По сути, вы можете утверждать, что это на самом деле не настоящий паттерн «Фабрика», а скорее паттерн «Выбор и внедрение». Тем не

менее новая динамическая функциональность CDI позволяет вам быть творческим в отношении реализации традиционных паттернов, превосходя классические образцы проектирования.

Когда и где использовать паттерны «Фабрика»

Традиционная реализация паттерна «Фабрика» была существенно изменена с тех пор, как GoF впервые познакомили нас с их использованием.

Применение абстрактных фабрик — действенный путь сокрытия создания объекта, особенно если оно сложно. И чем сложнее создать объект, тем обоснованней использовать для этого фабрику. Если важно, чтобы объекты создавались единообразно и их создание строго контролировалось, вам тем более стоит подумать о реализации паттерна «Фабрика».

Однако в дивном новом мире среды CDI, где контейнеры инстанцируют управляемые объекты, польза от абстрактных фабрик довольно спорна. Ваш наилучший вариант реализации паттерна «Фабрика» заключается в использовании аннотации `@Produces`, которая все-таки позволяет вам скрывать сложную порождающую логику в методах производителя данных и внедрять результирующий объект в клиент.

В качестве альтернативы вы можете использовать мощь среды CDI и позволить контейнеру создать объекты, а затем выбрать экземпляр, который вы хотите использовать, из пула сходных объектов. Однако при этом вы ограничены простыми объектами, которые могут быть удовлетворительно получены путем вызова конструктора по умолчанию.

Резюме

В этой главе вы увидели, как реализовать различные варианты паттерна «Фабрика» в не-CDI-среде. А в CDI-среде вы увидели, как методы производителя данных и аннотация `@Inject` в корне изменили способ реализации и использования паттерна «Фабрика» в Java EE.

Вы открыли для себя, как обуздать мощь автоматического создания контейнером объектов компонентов и как выбрать и использовать их в вашем коде.

Будем надеяться, что у вас не осталось сомнения в том, что реализация паттерна «Фабрика» в Java EE — изящный, простой и гладкий способ создания объектов.

Упражнения

1. Создайте автомобильную фабрику, которая производит различные виды машин и фургонов, используя паттерн «Абстрактная фабрика».
2. Реализуйте ту же самую автомобильную фабрику, что и в предыдущем упражнении, но применяя `@Produces`, квалификаторы и перечислимые типы.
3. Воспользовавшись мощью CDI-контейнера, реализуйте способ иметь множество объектов одного типа и выбирать нужный вам тип, основываясь на логике, учитывающей безопасность типов.

7 Паттерн «Декоратор»

В этой главе:

- как реализовать паттерн «Декоратор» в простом коде;
- как паттерн «Декоратор» решил дилемму на практике;
- как реализовать паттерн «Декоратор», используя аннотации `@Decorator` и `@Delegate`;
- как сделать паттерн «Декоратор» подключаемым с помощью дескрипторов развертывания;
- как использовать квалификаторы для достижения «дробного» контроля над использованием декораторов.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 07.zip, каждый из них назван в соответствии с наименованиями в тексте.

Книга GoF описывает паттерн «Декоратор» как «динамически возлагающий на объект новые функции» и приводит в качестве примера библиотеку для построения графических интерфейсов пользователя. Это превосходный практический пример, поскольку библиотека, добавляющая новые стили или варианты поведения к интерфейсу пользователя (UI), — идеальная задача для паттерна «Декоратор».

Книга «Паттерны проектирования» приводит кофейню в качестве примера различных дополнительных функций, таких как прибавляемые к товару взбитые сливки. Каждая новая добавка оборачивает объект-напиток и добавляет описанию и цене новое поведение. Этот пример подходит лучше всего, поскольку у нас был аналогичный случай на практике (см. следующую «Историю из практики»).

Паттерн «Декоратор» основан на объектах-компонентах и объектах-декораторах, реализущих один интерфейс. У декоратора есть переменная экземпляра, реализующая тот же интерфейс, так что он может оборачивать как объекты-компоненты, так и другие декораторы. Совместное использование одного интерфейса позволяет этим паттернам декорировать основные компоненты или другие декораторы. При соответствующей реализации несложно вызвать все подходящие реализации функций по порядку: от последнего декоратора до внутреннего целевого объекта-компонента. В большинстве случаев должно быть нетрудно адаптировать существующую систему к использованию паттерна «Декоратор».

Что такое декоратор

Паттерн «Декоратор» — один из структурных паттернов, описанных в книге GoF. Его суть — в оборачивании целевого объекта так, чтобы вы могли динамически добавлять новые функции во время выполнения. Каждый декоратор может обернуть еще один, благодаря чему возможно теоретически неограниченное количество декорирований целевых объектов.

Хотя такое поведение во время выполнения намного гибче, чем наследование путем создания производных классов, оно повышает степень сложности при создании конкретных производных классов, поскольку затрудняет определение типов и поведения объектов до запуска приложения.

Декораторы применяются почти во всех языках программирования и на всех платформах, от UI до прикладной части систем. Большинство фреймворков и сред выполнения использует паттерн «Декоратор» для добавления гибкости и специфического для времени выполнения поведения.

На платформе Java EE вы реализуете паттерн «Декоратор» без шаблонного кода. Однако, в отличие от большинства паттернов, в этой книге вы часто будете добавлять XML-конфигурации в файл `bean.xml`.

ИСТОРИЯ ИЗ ПРАКТИКИ

Несколько лет тому назад мы получили подряд на завершение разработки системы заказа и оплаты еды и напитков для компании, которая затем предоставляла ее в качестве сервиса платежей в местах совершения покупок (POS) для своих клиентов[1]. Клиентами были рестораны, кафе и бары. У нас не было знания предметной области, так что мы сделали кое-какие обоснованные предположения на основе ограниченных знаний и информации, которыми на то время обладали. К счастью, большинство наших предположений оказались верными.

Одним из наших правил проектирования было: если дополнительная возможность меняет цену товара, то она должна быть добавлена в виде нового товара. Так что, если ресторан подает лишние порции за дополнительную цену, в меню должен быть добавлен новый элемент. Однако, если такая возможность (например, дополнительный сыр) была бесплатной, эта информация должна быть добавлена к заказу в виде заметки на полях.

Правило работало нормально для всех клиентов до того дня, как мы столкнулись с владельцем кафе, чей бизнес функционировал немного по-другому. Кафе специализировалось на продаже десертов, но также предлагало пиццу в качестве пикантного дополнения. Пицца была единственным относящимся к еде пунктом во всем меню. Поскольку кафе не специализировалось на пицце, оно не предлагало определенные виды пиццы, а вместо этого позволяло посетителям кафе создавать их собственные пиццы на основе длинного списка начинок и брало плату за каждую начинку. Для кафе это был вполне практичный способ предложения пиццы своим посетителям, поскольку только немногие из них захотели бы съесть пиццу. Однако это была катастрофа для нашей системы из-за нашего правила проектирования: если дополнительная возможность меняет цену товара, она должна быть добавлена как новый товар. Поскольку у каждой начинки была своя стоимость, нам нужно было рассчитать все комбинации начинок и ввести в меню новую пиццу для каждой комбинации.

Как вы знаете, алгоритмы с факториальной сложностью дают быстрый рост, что в данном случае означало бы очень длинный список пицц после всего нескольких комбинаций. А так как это было неприемлемо, мы предложили клиенту, чтобы он ввел несколько пицц с фиксированным количеством начинок (1 начинка, 2 начинки, 3 начинки) и мог добавлять примечание к заказу для записи выбранных посетителем начинок. При таком решении мы могли сократить размеры списка с n! до n.

[1] Pyro: http://muse.com.tr/pyro.html.

Тем не менее это решение все еще не было наилучшим. Поскольку система уже была запущена и работала, нам требовалось найти путь решения проблемы, не ломающий другие части системы. Нам нужен был способ добавления функциональности во время выполнения. Требовалось «украсить» существующие объекты пицц начинками. Очевидно, решение заключалось в реализации паттерна «Декоратор». И именно это мы и сделали. Каждая выбранная посетителем начинка оборачивала объект пиццы подобно приведенному в книге «Паттерны проектирования» примеру.

Диаграмма классов декоратора. Как можно видеть на диаграмме классов на рис. 7.1, паттерн «Декоратор» вносит дополнительный шаблонный код в существующую иерархию классов. Он вводит совместно используемый интерфейс между целевым классом и декоратором. У декоратора должна быть ссылка на экземпляр этого интерфейса.

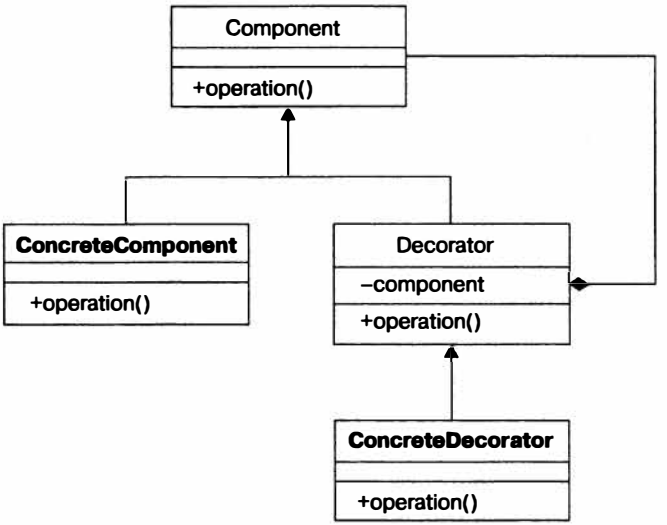


Рис. 7.1. Диаграмма классов паттерна «Декоратор»

Реализация паттерна «Декоратор» в простом коде

Если классы находятся на стадии проектирования, будет несложно добавить декораторы. Однако если необходимо снабдить декоратором существующую систему, вам может понадобиться переделать некоторые классы. Например, целевой класс должен реализовывать тот же интерфейс, что реализует декоратор.

Этот раздел демонстрирует применение паттерна «Декоратор» при разработке упрощенной POS-системы для пиццерии. Каждая пицца может быть «украшена» дополнительными начинками, такими как двойной сыр и бесплатный чили.

Во-первых, вам предстоит создать интерфейс `Order`, реализуемый с помощью класса `Pizza` и абстрактного класса декоратора `Extra`. Класс `Extra` расширяется классами добавочных начинок: `DoubleExtra`, `NoCostExtra` и `RegularExtra`.

Вы начнете с создания интерфейса Order в листинге 7.1.

Листинг 7.1. Интерфейс Order

```
public interface Order {
    public double getPrice();
    public String getLabel();
}
```

В листинге 7.2 вы создадите класс, представляющий в меню пиццу («Четыре сезона», «Маргарита», «Гавайская» и т. д.). Это целевой объект для декорирования.

Листинг 7.2. Предназначенный для декорирования класс, реализующий интерфейс Order

```
public class Pizza implements Order {

    private String label;
    private double price;

    public Pizza(String label, double price){
        this.label=label;
        this.price=price;
    }

    public double getPrice(){
        return this.price;
    }

    public String getLabel(){
        return this.label;
    }
}
```

Следующий код создает пиццу «Четыре сезона».

```
Order fourSeasonsPizza = new Pizza("Four Seasons Pizza", 10);
```

Далее вам необходимо создать декораторы, которые будут «украшать» пиццу добавочными начинками. Используем абстрактный класс таким образом, чтобы конкретным классам не пришлось реализовать все бизнес-методы интерфейса. Абстрактный декоратор создаст шаблон, который смогут расширять другие декораторы.

Пусть у вас есть различные типы начинок (сыр, перец чили, ананас и т. д.). Представьте, что посетитель хочет заказать немного более острое блюдо и ресторан не берет отдельную плату за эту дополнительную начинку. Таким образом, вам нужен декоратор, не добавляющий ничего к цене пиццы, но обеспечивающий соответствующую маркировку (что был заказан дополнительный чили). Кроме того, посетитель может попросить две дополнительные порции сыра и, если система напечатает «сыр» дважды, шеф-повар может подумать, что это ошибка в программе, и добавить только одну порцию сыра. Поэтому вам нужен дополнительный конкретный декоратор, чтобы выполнять правильную маркировку двойных начинок. Эти цели достигаются в листинге 7.3.

Листинг 7.3. Абстрактный декоратор, добавляющий дополнительные начинки

```
public abstract class Extra implements Order {

    protected Order order;
    protected String label;
    protected double price;

    public Extra(String label, double price, Order order) {
        this.label=label;
        this.price=price;
        this.order=order;
    }

    // Цена может оказаться основной проблемой,
    // так что поручим это конкретной реализации
    public abstract double getPrice();

    // Стандартной маркировки должно быть достаточно
    public String getLabel() {
        return order.getLabel()+", "+this.label;
    }
}
```

Теперь, когда у вас есть абстрактный декоратор, вы можете добавлять конкретное поведение и создавать конкретные декораторы. Вы начнете с декоратора `RegularExtra`, добавляющего цену и этикетку к целевому объекту (пицце). Поскольку функция маркировки уже есть в абстрактном декораторе и наследуется всеми расширяющими его подклассами, вам только нужно реализовать функциональность формирования цены. Листинг 7.4 об этом позаботится.

Листинг 7.4. Абстрактный декоратор, добавляющий дополнительные начинки

```
public class RegularExtra extends Extra {

    public RegularExtra(String label, double price, Order order) {
       super(label, price, order);
    }

    public Double getPrice() {
        return this.price+order.getPrice();
    }
}
```

Далее вам нужно создать `NoCostDecorator`, который меняет строку `label`, но не добавляет ничего к цене пиццы (листинг 7.5).

Листинг 7.5. Декоратор, добавляющий дополнительные начинки бесплатно

```
public class NoCostExtra extends Extra {
    public NoCostExtra(String label, double price, Order order) {
        super(label, price, order);
    }
    public Double getPrice() {
```

```
        return order.getPrice();
    }
}
```

И наконец, в листинге 7.6 вы реализуете декоратор `DoubleExtra`, чтобы избежать двукратной печати начинки на этикетке. Он удваивает цену и добавляет ключевое слово `double` перед целевой этикеткой.

Листинг 7.6. Декоратор, добавляющий двойные начинки

```
public class DoubleExtra extends Extra {

    public DoubleExtra(String label, double price, Order order) {
        super(label, price, order);
    }

    public Double getPrice() {
        return (this.price*2)+order.getPrice();
    }

    public String getLabel() {
        return order.getLabel()+ ", Double " + this.label;
    }
}
```

Теперь, когда паттерн «Декоратор» для добавления дополнительных начинок к вашей пицце реализован, можете протестировать реализацию.

```
Order fourSeasonsPizza = new Pizza("Four Seasons Pizza", 10);
fourSeasonsPizza = new RegularExtra("Pepperoni", 4, fourSeasonsPizza );
fourSeasonsPizza = new DoubleExtra("Mozzarella", 2, fourSeasonsPizza );
fourSeasonsPizza = new NoCostExtra("Chili", 2, fourSeasonsPizza );

System.out.println(fourSeasonsPizza.getPrice());
System.out.println(fourSeasonsPizza.getLabel());
```

Вывод в консоли будет следующим:

```
18.0
Pizza, Pepperoni, Double Mozzarella, Chili
```

Но погодите! Здесь есть потенциальная ошибка! Чили не бесплатен, если вы заказываете его как гарнир, но шеф-повар с радостью подаст его бесплатно в пицце. Вам нужно позаботиться, чтобы система учитывала это различие. Только представьте себе, что эти значения и этикетки берутся из базы данных. Что бы вы сделали для создания различных видов поведения для чили? Одним вариантом могло бы быть создание двух объектов для чили: один был бы маркирован как «с пиццей». Конечно, это была бы халтура, оставляющая любому официанту лазейку для заказа бесплатного чили для его друзей. Другим вариантом стало бы создание дополнительного конструктора в абстрактном классе, не принимающего цену в качестве параметра. Любой не взимающий плату за добавки декоратор мог бы реализовать это.

Реализация паттерна «Декоратор» в Java EE

В отличие от большинства других паттернов, описанных в этой книге, вы реализуете паттерн «Декоратор», описывая классы декоратора в дескрипторе развертывания `bean.xml` (за исключением случая, когда он аннотирован `@Priority`; см. далее в этом разделе). К счастью, эта конфигурация проста и дает вам преимущества легкой подключаемости и контроля за порядком вызова декораторов.

Реализация декоратора в Java EE вводит две новые аннотации: `@Decorator` и `@Delegate`. Первая аннотирует класс декоратора, а вторая аннотирует точку внедрения делегата, в которую внедряется декорируемый класс.

В качестве примера вы возьмете магазин, желающий сделать скидку на некоторые из товаров. Он будет использовать декоратор для применения этой скидки к обычной розничной цене. В листинге 7.7 вы начнете с создания интерфейса, который затем используете для связи декоратора с предназначенным для декорирования объектом.

Листинг 7.7. Интерфейс Product

```
public interface Product {
    public void setLabel(String label);
    public void setPrice(double price);
    public String getLabel();
    public double getPrice();
    public String generateLabel();
}
```

Интерфейс вводит метод `generateLabel`, который декоратор реализует для добавления поведения по предоставлению скидки. В листинге 7.8 вы создаете класс `Table`. Этот товар вы хотите декорировать, поэтому он реализует интерфейс `Product`.

Листинг 7.8. Предназначенный для декорирования класс реализует интерфейс Product

```
public class Table implements Product {

    private String label = "Dining Table";
    private double price = 100.00;

    public void setLabel(String label) {
        this.label = label;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getLabel() {
        return label;
    }

    public double getPrice() {
```

```
        return price;
    }

    public String generateLabel() {
        return price + ". " + label;
    }
}
```

Путем реализации интерфейса Product вы создаете декоратор PriceDiscountDecorator. Этот класс реализует метод generateLabel и добавляет ему поведение по предоставлению скидки. Декоратор снижает цену товара на 50 % и добавляет к этикетке товара текст (Discounted).

Чтобы контейнер мог распознавать этот класс как декоратор, вы должны аннотировать его с помощью @Decorator. Точка внедрения делегата (экземпляра, который будет декорирован) аннотирована @Delegate и должна быть внедренным полем, параметром метода-инициализатора или параметром метода-конструктора компонента. Тип делегата должен быть интерфейсом, реализованным предназначенными для декорирования классами, в данном случае Product. CDI-контейнер внедряет любой доступный экземпляр интерфейса Product в переменную экземпляра product, как показано в листинге 7.9.

Листинг 7.9. Декоратор PriceDiscountDecorator

```
@Decorator
public class PriceDiscountDecorator implements Product {

    @Any
    @Inject
    @Delegate
    private Product product;

    public String generateLabel() {
        product.setPrice(product.getPrice() * 0.5);
        product.setLabel(product.getLabel() + " (Discounted)");
        return product.generateLabel();
    }

    // Показаны не все методы

}
```

Наконец, вы должны описать декоратор в bean.xml. Хотя большая часть конфигурационных настроек уже была выполнена посредством аннотаций, вам все еще нужно добавить кое-какие XML-конфигурации, чтобы заставить декоратор работать.

Необходимость конфигурационных настроек может вас разочаровать, поскольку вы уже аннотировали свой декоратор. Тем не менее эти конфигурационные настройки просты и необходимы для того, чтобы вы могли описать порядок выполнения декораторов (если их более одного). Добавьте следующие строки в bean.xml:

```
<decorators>
    <class>com.devchronicles.decorator.PriceDiscountDecorator</class>
</decorators>
```

Ваша работа сделана. Теперь вы можете использовать ваш декоратор.

```
@Any
@Inject
Product product;

public void createPriceList(){
    System.out.println("Label: " + product.generateLabel());
}
```

Экземпляр `Table` внедрен в переменную экземпляра `product`, и метод `generateLabel` вызван. Вывод в консоли будет следующим:

```
Label: 12.5, Dining Table (Discounted)
```

Когда выполняется обращение к методу `generateLabel` любого экземпляра `Product`, контейнер его перехватывает. Обращение делегируется соответствующему методу декоратора `PriceDiscountDecorator`, где он снижает цену товара и передает вызов в исходное место назначения посредством вызова метода `generateLabel` объекта `Table`.

Формируется цепочка вызовов, включающая все декораторы, которые описаны как реализующие интерфейс `Product` декорирующие классы. Порядок, в котором вызываются декораторы, определяется порядком, в котором они описаны в дескрипторе развертывания `bean.xml`.

Вы увидите все это в действии в листинге 7.10, где определите еще один декоратор. Вы создаете декоратор `BlackFridayDiscountDecorator`, реализуете интерфейс `Product` и добавляете аннотации `@Decorator` и `@Delegate`.

Листинг 7.10. Декоратор `BlackFridayDiscountDecorator`

```
@Decorator
public class BlackFridayDiscountDecorator extends AbstractDiscountDecorator {

    @Any
    @Inject
    @Delegate
    private Product product;

    public String generateLabel() {
        product.setPrice(product.getPrice() * 0.25);
        product.setLabel(product.getLabel());
        return product.generateLabel();
    }

    // Показаны не все методы

}
```

Вы должны добавить декораторы в содержащий bean.xml архив в том порядке, в котором они должны, по-вашему, вызываться. В следующем фрагменте кода вы объявляете, что декоратор PriceDiscountDecorator должен вызываться перед декоратором BlackFridayDiscountDecorator.

```
<decorators>
    <class>com.devchronicles.decorator.PriceDiscountDecorator</class>
    <class>com.devchronicles.decorator.BlackFridayDiscountDecorator </class>
</decorators>
```

Когда вызывается метод generateLabel, формируется цепочка вызовов, включающая два декоратора. Обращение к generateLabel перехватывается и делегируется методу generateLabel декоратора PriceDiscountDecorator. Он вызывает getPrice, который будет перехвачен и делегирован методу getPrice декоратора BlackFridayDiscountDecorator, который, в свою очередь, вызывает метод getPrice своего внедренного объекта Product. (Это тот же экземпляр, который вы внедрили в декоратор PriceDiscountDecorator.) Этот вызов не перехватывается, поскольку больше нет объявленных для интерфейса декораторов, и обращается к методу getPrice объекта Table. Сразу по завершении этого вызова происходит возвращение вниз по стеку вызовов к первому методу getPrice. Он вызывается, возвращая цену Table. Декоратор снижает цену на 50 % и вызывает метод setPrice. Этот вызов делегируется вверх по цепочке вызовов до тех пор, пока не достигает объекта Table, где устанавливается новая цена. Затем вызов возвращается вниз по цепочке.

Выполняется обращение к методу getLabel, и создается цепочка вызовов, подобная цепочке метода getPrice.

И наконец, вызывается и перехватывается декоратором BlackFridayDiscountDecorator метод generateLabel. Цена снижается еще на 25 %, и запускается подобная сформированной декоратором PriceDiscountDecorator цепочка вызовов.

Выполняется следующий вывод в консоль:

```
Label: 6.25. Dining Table (Discounted)
```

Чтобы цепочка не разорвалась, метод generateLabel должен делегировать методу generateLabel внедренного экземпляра делегата, в противном случае цепочка разрывается и вызывается только первый декоратор.

Все классы, реализующие тот же интерфейс, что и реализованный точкой внедрения делегата, декорируются, но только если эти декораторы объявлены в bean.xml. Из этого следует два основных вывода.

- Декораторы могут быть активизированы/отключены во время развертывания путем редактирования файла bean.xml. Это дает удивительную гибкость управления тем, где и какие декораторы будут вызваны. Например, вы можете реализовать декоратор снижения цены только на период распродажи и отключить его, когда период закончится. Гибкость объявлений в дескрипторе развертывания означает, что этот декоратор может быть легко активизирован снова, если позднее потребуется отладочная информация.
- Декоратор автоматически применяется к классам, которые реализуют тот же интерфейс. Это выгодно при добавлении новых классов, поскольку они декорируются без написания дополнительного кода. Однако это может оказаться

неудобным, если нужно, чтобы не все классы одного типа были декорированы. К счастью, есть решение для этой проблемы, включающее использование квалификаторов для аннотации только тех классов, которые должны быть декорированы.

Чтобы не декорировать все классы одного типа, вам нужно создать пользовательский квалификатор и аннотировать точку внедрения делегата и все классы, которые, по-вашему, должны быть декорированы. Вы создадите товар `Plate`, реализующий интерфейс `Product`. Только на этот товар будет предоставляться скидка. Чтобы реализовать это требование, вы аннотируете его пользовательским квалификатором, таким образом не допуская декорирования других товаров.

Итак, вы создаете пользовательский квалификатор и называете его `@ClearanceSale`.

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, PARAMETER, TYPE})
public @interface ClearanceSale {}
```

В листинге 7.11 вы создаете новую реализацию интерфейса `Product` и аннотируете ее с помощью своего пользовательского квалификатора.

Листинг 7.11. Предназначенный для декорации класс аннотируется пользовательским квалификатором

```
@ClearanceSale

public class Plate implements Product {

    private String label = "Plate";
    private double price = 50.00;

    public void setLabel(String label) {
        this.label = label;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getLabel() {
        return label;
    }

    public double getPrice() {
        return price;
    }

    public String generateLabel() {
        return price + ", " + label;
    }
}
```

И наконец, вы аннотируете точку внедрения делегата в декораторе, который хотите вызвать. В данном случае вам необходимо выбрать декоратор PriceDiscountDecorator.

```
@ClearanceSale
@Any
@Inject
@Delegate
private Product product;
```

Только классы, аннотированные @ClearanceSale и реализующие интерфейс Product, внедряются в точку внедрения делегата декоратора PriceDiscountDecorator, поэтому только ваш класс Plate будет декорирован. Точка внедрения делегата может иметь столько квалификаторов, сколько требуется, и будет «привязана» только к компонентам с тем же квалификатором.

Декораторы без XML-конфигурации. Во время развертывания CDI-контейнер просматривает все файлы JAR и WAR в приложении в поисках дескрипторов развертывания bean.xml. Найденные он обрабатывает по очереди, выполняя соответствующие конфигурационные настройки. Встретив дескриптор <decorator/>, он активизирует декораторы для архива, в котором был найден файл bean.xml. Это не активизирует их для всего приложения, представляя собой проблему для тех разработчиков, кто хочет, чтобы декораторы применялись ко всем классам, реализующим один и то же интерфейс, независимо от того, где в приложении они находятся. Начиная с CDI 1.1[1], стало возможно активизировать декораторы для всего приложения путем аннотации класса декоратора аннотацией @Priority с указанием значения Interceptor.Priority. Вот пример того, как активизировать два ваших декоратора для всего приложения.

```
@Priority(Interceptor.Priority.APPLICATION)
@Decorator
public class PriceDiscountDecorator extends AbstractDiscountDecorator

@Priority(Interceptor.Priority.APPLICATION+10)
@Decorator
public class BlackFridayDiscountDecorator extends AbstractDiscountDecorator
```

Первыми вызываются декораторы, аннотированные с более низким значением приоритета. В предыдущем примере PriceDiscountDecorator вызывается перед BlackFridayDiscountDecorator.

Декораторы, аннотированные @Priority, вызываются перед декораторами в дескрипторе развертывания. Если декоратор активизирован и там и там, он вызывается дважды. Это может привести к нежелательным последствиям, так что вам необходимо убедиться, что декораторы активизированы только одним способом.

---

[1] CDI Specifications 1.1: http://docs.jboss.org/cdi/spec/1.1/cdi-spec.html#decorators.

Где и когда использовать паттерн «Декоратор»

Паттерн «Декоратор» динамически добавляет объекту поведение во время выполнения или тогда, когда невозможно или нецелесообразно создавать производные классы (возможно, потому, что при этом создаются множественные подклассы). Пример с пиццерией показывает, как добавить поведение к объекту пиццы во время выполнения на основе сделанного посетителем выбора.

Функциональность интерфейса программирования приложений (API) может быть расширена и усовершенствована посредством оборачивания в декоратор. Подобным образом часто декорируются потоки данных. `java.io.BufferedInputStream` — хороший пример декоратора, оборачивающего низкоуровневое API и добавляющего функциональность буферу потока ввода.

В платформе Java EE декораторы реализованы с помощью контекста и внедрения зависимостей (CDI). Вы можете использовать декораторы для добавления нового бизнес-поведения или любой другой функциональности, которая может быть обернута вокруг исходного объекта. Однако подобный проект должен быть хорошо документирован и четко реализован ради удобства сопровождения.

Подключаемость декораторов путем описания в дескрипторе развертывания облегчает активизацию/отключение декораторов без перекомпиляции и повторного развертывания. В среде, где требуется «горячее» развертывание, нет необходимости перезагружать сервер для вступления в силу изменений в `bean.xml`. Это делает исключительно легким изменение поведения приложения в условиях эксплуатации, без остановки на обслуживание.

Использование квалификаторов в большей степени обеспечивает пошаговый контроль за выполнением декораторов, чем их активизация/отключение в дескрипторе развертывания `bean.xml`. Вы можете применять квалификаторы для недопущения декорирования отдельных реализаций интерфейса или для добавления различных декораторов к реализациям одного интерфейса.

Декоратор перехватывает обращения только к определенным типам языка Java. Он «знает» обо всей семантике данного интерфейса и может реализовывать бизнес-логику. Это делает его идеальным для моделирования бизнес-функциональности, отождествляемой с определенным типом интерфейса.

Декораторы часто противопоставляют перехватчикам. Вторые перехватывают вызовы любого типа языка Java, но «не осведомлены» о семантике и поэтому не подходят для моделирования бизнес-функциональности. Перехватчики используются для реализации не связанной с бизнес-логикой сквозной функциональности, такой как журналирование, безопасность и аудит.

Интенсивное использование декораторов может привести к ошибкам времени выполнения, более сложному для понимания коду и потере преимущества строго типизированного статического полиморфизма. Оно также может потребовать дополнительных контрольных примеров. Тем не менее декораторы

могут обеспечить практически неограниченные возможности масштабирования приложения и отличный интерфейс для будущих реализаций без нарушения старого кода.

Резюме

В этой главе вы увидели, как реализация паттерна «Декоратор» в Java EE практически неотличима от ее до-Java EE-предка. Подлежащий декорированию объект инстанцируется и внедряется контейнером, а декораторы, которые должны быть применены, определяются по описаниям в дескрипторе развертывания `bean.xml` или путем стратегического применения пользовательских квалификаторов.

Использование аннотаций и внедрения зависимостей уменьшило количество строк кода, которые необходимо написать для реализации декоратора, и облегчило ввод дополнительных новых классов, которые автоматически декорируются благодаря реализуемому ими интерфейсу.

Вы увидели, как паттерн «Декоратор» эволюционировал в действительно легко подключаемый паттерн, который может быть активизирован/отключен во время работы приложения без потерь времени на сервисное обслуживание. Тем не менее он сохранил свои первоначальные принципы добавления поведения или функциональности к декорируемым им объектам.

Упражнения

1. Расширьте приведенный ранее пример с магазином, добавив больше осуществляющих скидки декораторов и введя больше квалификаторов для достижения лучшего контроля над тем, какие декораторы вызываются для каких конкретных реализаций.
2. Реализуйте паттерн «Декоратор» на существующем API для добавления новой функциональности. Например: `java.io.FileInputStream`.
3. Создайте декоратор, добавляющий следующее поведение системе банковской бухгалтерии: когда клиент снимает со счета более определенной суммы наличными, ему посылается текстовое SMS, извещающее о снятии.

8 Аспектно-ориентированное программирование (перехватчики)

В этой главе:

- введение в аспектно-ориентированное программирование;
- аспекты в языке Java;
- использование сервлетных фильтров в качестве аспектов;
- аспекты в Java EE, перехватчики;
- EJB- и CDI-перехватчики.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 08.zip, каждый из них назван в соответствии с наименованиями в тексте.

Аспектно-ориентированное программирование (АОП) — идея отнюдь не новая. Его место в языке Java и сторонних фреймворках было надежно закреплено с первых дней разработки корпоративных приложений. Несмотря на это, оно не было одним из классических паттернов проектирования, перечисленных в книге GoF.

АОП привнесло в программирование новую концепцию и парадигму. Суть идеи в том, что порядок выполнения кода определяется аспектами. Каждый аспект перехватывает выполнение программы и добавляет свое собственное поведение перед продолжением выполнения вызова.

Аспекты действуют исключительно эффективно, добавляя дополнительные логику и поведение к коду во время выполнения. Однако они также становятся причиной неоднозначного и сложного для понимания порядка выполнения кода, что часто может привести к практически не поддающемуся отладке коду. У АОП есть много сторонников и почитателей, но много и ненавистников.

К счастью, на платформе Java EE существует отличная понятная реализация, которая может быть весьма полезной, если используется должным образом и в соответствующем контексте.

Что такое аспектно-ориентированное программирование

Аспектно-ориентированное программирование (АОП) нацелено на решение вопросов совместной функциональности путем добавления нового поведения для существующего кода или приложения. Нет ничего необычного в том, чтобы получить в середине цикла разработки новую заявку на добавление журналирования или исправление уязвимости. Подобные заявки могут поглощать массу времени на переписывание существующего кода, даже если код журналирования — просто набор повторяющихся строк. Такие вопросы совместной функциональности, появляются ли они в середине цикла разработки или на этапе проектирования, называются *сквозной функциональностью*. На решение таких вопросов может быть направлено АОП.

За последнее десятилетие АОП стало популярной парадигмой программирования. Хотя Java не предлагает полностью законченное и готовое для использования решение, парадигма АОП обеспечивается в некоторых развитых фреймворках. AspectJ и Spring широко распространены и в течение уже длительного времени применяются в проектах на языке Java. В Java также существует простой, хотя и более фундаментальный подход с сервлетными фильтрами, пусть он и ограничивается веб-запросами. При использовании сервлетных фильтров любой запрос или ответ может быть перехвачен с добавлением любого дополнительного поведения.

Платформа Java EE приняла АОП на вооружение и представила концепцию *перехватчика (Interceptor)*. Каждое обновление Java EE приносит новую функциональность, реализуя полный потенциал АОП на платформе Java EE.

АОП не классифицируется как паттерн проектирования, но принято в качестве парадигмы программирования. Ни книга GoF, ни «Паттерны проектирования» не рассматривают аспекты. Однако если бы хоть одна из них рассматривала, то описывала бы их так: «Аспект обеспечивает путь для изменения динамического поведения во время выполнения (или компиляции) программы для решения вопросов сквозной функциональности в существующей базе исходного кода».

АОП основывается на внедрении кода во время компиляции или выполнения для добавления желаемого поведения или функциональности к каждой точке существующей базы кода, которая соответствует заданным условиям внедрения. Фреймворки, осуществляющие внедрение на этапе компиляции, обычно более работоспособны, но они создают файлы классов, которые не совпадают с исходными текстами построчно из-за вставленного кода. Инъекции времени выполнения не изменяют исходный код или файлы классов и осуществляются путем перехвата вызовов и выполнения желаемого кода перед исходной последовательностью команд или после нее.

АОП может доказать свою полезность, если нужно добавить к базе исходного кода повторяющееся действие, такое как журналирование или событие безопасности. Аспекты могут быть включены или отключены в зависимости от окружения или этапа проекта. Аспекты могут динамически добавлять желаемое поведение к выполняемому в текущий момент коду. Они динамически

декорируют вызовы методов подобно тому, как декорирует объекты паттерн «Декоратор».

ИСТОРИЯ ИЗ ПРАКТИКИ

Мы только что закончили разработку веб-приложения и завершали финальный этап перед запуском в эксплуатацию. После завершения функционального тестирования и проверки на соответствие требованиям заказчика нам нужно было передать приложение для проверки на защищенность. Была нанята команда экспертов по безопасности для тестирования нашей системы на уязвимости. Поскольку предшествовавшее нашему приложение было взломано и произошла утечка важных данных, тестирование на защищенность воспринималось весьма серьезно.

Мы были вполне уверены в своем приложении, так что все захватили попкорн и наблюдали за этапами тестирования. После огромного количества успешных тестов один в итоге был провален. Специалисты по безопасности сумели перехватить HTTP-запрос (HyperText Transfer Protocol, протокол передачи гипертекста) и изменить некоторые параметры для получения ответа от приложения. Проблема была не слишком серьезной, поскольку у промежуточного уровня была собственная система авторизации. Тем не менее модифицированный запрос мог получить доступ к авторизованному ответу.

Подытоживая: клиент должен вызвать несколько сервисов для доступа к ресурсу. Пусть сервис A возвращает некоторые ID и сервис B может быть вызван с ID, возвращенными сервисом A. Аналогично сервис C может быть вызван с одним из ID, возвращенных сервисом B. Это значит, что взломщик может осуществить перехват и вставить случайное ID B, которое пользователь должен был запросить, но не сделал этого. В подобном случае клиент обошел бы стандартное прохождение вызова и получил доступ к ресурсу.

Поскольку информация, к которой предоставлялся доступ, была авторизованным пользователем ресурсом, то проблема была не слишком серьезной. Тем не менее о ней было доложено как о слабом месте в безопасности, позволяющем обойти стандартное прохождение вызова, что привлекло внимание.

Приложение было уже закончено и протестировано, и мы очень не хотели проводить его рефакторинг. Вместо этого мы предложили гениальную идею: поскольку в каждом запросе клиенту необходимо использовать ID из предыдущего ответа, мы можем перехватывать все возвращаемые ID. Если требуемый ID был из списка запрошенных, мы легко можем пропустить его или сделать сессию недействительной и заставить пользователя подключиться заново.

Идея была простой и эффективной, но мы все еще не знали, как реализовать ее с минимальными изменениями. Поскольку все, что мы хотели сделать, имело отношение к веб-запросам, перехват и проверка их казались хорошей идеей. К счастью, Java уже предлагал встроенное решение, и нам не нужно было использовать причудливый сторонний фреймворк.

Решением проблемы стала реализация сервлетного фильтра. Это позволяло закэшировать требуемые ID в ответе и проверить, имеет ли следующий запрос действительный ID из списка. Нам нужно было только добавить файл класса, который бы играл роль сервлетного фильтра и XML-описание, чтобы привести его в действие. Решение было подключаемым и могло быть интегрировано без проблем. Кроме того, оно давало возможность отключить его в среде разработки.

В итоге система не только прошла все тесты на защищенность, но и сделала это лучше, чем можно было ожидать. Мы могли легко протоколировать и извлекать статистические данные из пар «запрос/ответ». И, что лучше всего, решение никак не влияло на общую архитектуру и сложность системы.

АОП может быть отличным инструментом для инкапсуляции совместной небизнес-функциональности. Однако АОП может служить и источником путаницы при добавлении нового поведения к бизнес-логике. Подобные реализации приводят к децентрализованной, распределенной и сложной для тестирования и отладки бизнес-логике. Получающийся код сложно поддерживать.

Реализация АОП в простом коде

Java SE не предлагает готовую поддержку АОП. Вы можете получить «чистый» АОП, используя сторонние фреймворки, такие как AspectJ или Spring. Они обычно зависят от конфигурации на чистом XML; однако вы можете реализовать АОП, используя аннотации. Реализация и конфигурация обоих фреймворков выходит за рамки этой книги, но они хорошо себя проявили и легко могут быть реализованы. Оба являются обоснованной альтернативой реализации Java EE.

Тем не менее веб-приложения Java обладают преимуществом использования сервлетов для перехвата запросов или ответов, что работает аналогично аспектам. Для реализации сервлетного фильтра создадим новый файл класса и реализуем интерфейс сервлетного фильтра. Затем обеспечим реализацию метода `doFilter()`, как показано в листинге 8.1.

Листинг 8.1. Простая реализация сервлетного фильтра

```
package com.devchronicles.interceptor.filter;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SecurityFilter implements Filter {

    @SuppressWarnings("unused")
    private FilterConfig filterConfig = null;

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain filterChain) throws IOException, ServletException {
        Log.info(((HttpServletRequest) request).getRemoteAddr());
        // Выполняем проверки безопасности
}

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
    }

}
```

Для активизации сервлетного фильтра на заданных URL (Uniform Resource Locator, унифицированные указатели ресурсов) контейнеру сервлетов нужна конфигурация, показанная в листинге 8.2. Она была помещена в файл web.xml веб-приложения.

Листинг 8.2. Описание сервлетного фильтра

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

<filter>
    <filter-name>LineSsoFilter</filter-name>
    <filterclass>com.devchronicles.interceptor.filter</filter-class>
</filter>

<filter-mapping>
    <filter-name>SecurityFilter</filter-name>
    <urlpattern>/*</url-pattern>
</filter-mapping>

</web-app>
```

Реализовать фильтры посредством Servlet 3.0, подобно показанному в листинге 8.3, даже проще, поскольку тут используются аннотации и нет необходимости в XML-конфигурации.

Листинг 8.3. Простая реализация сервлетного фильтра в Servlet 3.0

```
package com.devchronicles.interceptor.filter;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import javax.servlet.Filter;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter", urlPatterns = {"/*"})
public class SecurityFilter implements Filter {

    @Override
```

```
    public void doFilter(ServletRequest request, ServletResponse response,
      FilterChain filterChain) throws IOException, ServletException {
        Log.info(((HttpServletRequest) request).getRemoteAddr());
        // Выполняем проверки безопасности
    }

}
```

Сервлетные фильтры — простые в реализации инструменты, тем не менее они также весьма действенны. Однако функциональность все еще ограничена клиент-серверными веб-запросами. Чтобы перехватить обращения к другим методам или отладить перехват, вам понадобится более передовой подход.

Аспекты в Java EE, перехватчики

Платформа J2EE не предоставляла готового для использования решения для АОП, но гармонично сочеталась со сторонними фреймворками. Платформа Java EE 5 представила перехватчики, что привело к удобному в использовании встроенному аспектному подходу. Однако концепция перехватчиков была ограничена компонентами Enterprise JavaBeans (EJB) вплоть до появления контекста и внедрения зависимостей (CDI).

Перехватчики на платформе Java EE работают схожим с аспектами образом. Каждый перехватчик относится к определенной функциональности и «владеет» блоком кода, содержащим предназначенную для добавления функциональность. Целевой объект для декорирования называется *советом (Advice)*. Каждое обращение к совету внутри области видимости перехватчика перехватывается. Точное место расположения предназначенного для выполнения аспекта именуется *срез точек внедрения*.

Базовые перехватчики платформы Java EE могут работать только с EJB-компонентами. Представьте себе приложение, состоящее из сотен EJB-компонентов. Приложение в целом может быть конфигурировано для регистрации вызовов всех EJB-компонентов посредством применения одного предназначенного для всех этих EJB-компонентов перехватчика.

Реализация перехватчиков на платформе Java EE достаточно проста. Во-первых, необходимо создать новый класс перехватчика и аннотировать его с помощью `@Interceptor`. Этот класс «владеет» кодом совета. Любой метод, аннотированный `@AroundInvoke`, выполняется в срезе точек внедрения. Тем не менее есть несколько синтаксических правил относительно сигнатуры метода среза точек внедрения:

- должен возвращать объект типа `Object` и иметь параметр типа `InvocationContext`;
- должен объявлять о возможной генерации исключения.

Вы можете использовать параметр `InvocationContext` для осуществления доступа к информации о текущем контексте, как показано в листинге 8.4.

Листинг 8.4. Простая реализация перехватчика

```
package com.devchronicles.interceptor;

import javax.interceptor.AroundInvoke;
```

```
import javax.interceptor.InvocationContext;

@Interceptor
public class SecurityInterceptor {

    @AroundInvoke
    public Object doSecurityCheck(InvocationContext context) throws Exception{

        // Выполняем проверки безопасности!

        Logger.getLogger("SecurityLog").info(context.getMethod().getName()+
                                        "is accessed!");

        return context.proceed();
    }
}
```

Чтобы привести в действие класс перехватчика, вам необходимо аннотировать целевой совет с помощью `@Interceptors`, как в листинге 8.5. Аннотация `@Interceptors` может быть использована только для EJB- или MDB-компонентов (Message Driven Bean, управляемые сообщениями компоненты).

Листинг 8.5. Простая реализация целевого совета

```
package com.devchronicles.interceptor;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.interceptor.Interceptors;

@Interceptors(SecurityInterceptor.class)
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class SomeBusinessService {

    public void startService(){
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done...");
    }

    public void startAnotherService(){
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done again...");
    }
}
```

Аннотация `@Interceptors` весьма гибка. К тому же вы можете использовать ее на уровне как класса, так и метода. Аннотация `@Interceptors` поддерживает также указание множественных перехватчиков, что делает возможными несколько

перехватчиков на целевом совете. Листинг 8.5 использует перехватчики уровня класса, означающие, что SecutiryInterceptor будет перехватывать любое обращение к сервису. Если вы не хотите, чтобы перехватчик охватывал все вызовы методов класса, вы можете использовать аннотации *уровня метода*, как показано в листинге 8.6.

Листинг 8.6. Реализация перехватчиков уровня метода

```
package com.devchronicles.interceptor;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.interceptor.Interceptors;

@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class SomeBusinessService {

    @Interceptors(SecurityInterceptor.class)
    public void startService(){
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done...");
    }

    public void startAnotherService(){
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done again...");
    }
}
```

На этот раз перехватываются только обращения к методу startService(), в отличие от показанного в листинге 8.5, где перехватываются все методы класса. При этом вы должны аннотировать каждый метод отдельно.

Использование @Interceptor, @Interceptors с @AroundInvoke предоставляет мощный инструмент для решения вопросов сквозной функциональности методами АОП. Вдобавок перехватчики предлагают основанную на аннотациях удобную реализацию без шаблонного кода.

Вы можете использовать интерфейс InvocationContext для извлечения информации о контексте или для взаимодействия с контекстом *совета*. В табл. 8.1 перечислено несколько полезных методов.

Таблица 8.1. Методы АОП

Метод	Описание
public Object getTarget();	Возврат в целевой совет
public Method getMethod();	Возврат выполняемого метода из совета
public Object[] getParameters();	Получение параметров метода целевого совета

Метод	Описание
public void setParameters (Object[]);	Задание параметров метода целевого совета
public java.util.Map<String,Object> getContextData();	Получение данных контекста
public Object proceed() throws Exception;	Продолжение выполнения

В листинге 8.7 вы можете получить доступ к имени метода. Вы также можете проверить, был ли доступ ранее авторизован перехватчиком. Если нет, можете авторизовать пользователя на осуществление доступа к данному методу.

Листинг 8.7. Получение доступа к информации в InvocationContext

```
package com.devchronicles.interceptor;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Interceptor
public class SecurityInterceptor {

    @AroundInvoke
    public Object doSecurityCheck(InvocationContext context) throws Exception{

        // Выполняем проверки безопасности!
        Logger.getLogger("SecurityLog").info(context.getMethod()
                                        .getName()+ "is accessed!");
        String user = context.getContextData.get("user");

        if (user==null){
            user=(String)context.getParameters()[0];
            context.getContextData.put("user", user)'
        }

        return context.proceed();
    }

}
```

Жизненный цикл перехватчика

Вы можете легко захватывать фазы жизненного цикла перехватчика с помощью аннотаций жизненного цикла. В отличие от расширения (класса. — *Примеч. пер.*) и подмены (производным классом виртуальных функций базового класса. — *Примеч. пер.*), использование аннотаций жизненного цикла привязывает любую функцию к соответствующей фазе. Доступны следующие аннотации жизненного цикла: `@PostConstruct`, `@PrePassivate`, `@PostActivate` и `@PreDestroy`. Листинг 8.8 показывает, как осуществить привязку с помощью методов жизненного цикла перехватчика.

Листинг 8.8. Привязка к фазам жизненного цикла перехватчика

```
package com.devchronicles.interceptor;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Interceptor
public class SecurityInterceptor {

    @AroundInvoke
    public Object doSecurityCheck(InvocationContext context)
                                  throws Exception{
        // Выполняем проверку безопасности!
        Logger.getLogger("Security Log").info(context.getMethod()
                                        .getName()+ "is accessed!");
        String user = context.getContextData.get("user");
        if (user==null){
            user=(String)context.getParameters()[0];
            context.getContextData.put("user", user);
        }

        return context.proceed();
    }

    @PostConstruct
    public void onStart(){
        Logger.getLogger("SecurityLog").info("Activating");
    }

    @PreDestroy
    public void onShutdown(){
        Logger.getLogger("SecurityLog").info("Deactivating");
    }
}
```

Поскольку привязки основаны на аннотациях, имена методов несущественны и вы можете использовать любое имя.

Перехватчики уровня по умолчанию

Аннотирование целевого совета аннотацией `@Interceptors` обеспечивает легкую реализацию и настройку, но сама сущность АОП обычно требует большего. Большинству сценариев требуется, чтобы перехватчик для выполнения его функций имел в качестве целей все советы. Представьте себе добавление перехватчиков для журналирования или событий безопасности — использование только какого-то подмножества EJB-компонентов не даст результата. Кроме того, аннотирование всех EJB-компонентов может быть обременительным и с легкостью приводить к ошибкам из-за «человеческого фактора».

Платформа Java EE обеспечивает перехватчики уровня по умолчанию для обработки всех EJB-компонентов или их подмножеств в соответствии с предусмотренной схемой именования. В отличие от предыдущего примера, для реализации перехватчиков уровня по умолчанию вам понадобится XML-конфигурация:

```
<ejb-jar...>
    <interceptors>
        <interceptor>
            <interceptor-class>
                com.devchronicles.SecurityInterceptor
            </interceptor-class>
        </interceptor>
    </interceptors>
    <assembly-descriptor>
        <interceptor-binding>
          <ejb-name>*</ejb-name>
          <interceptor-class>
              <interceptor-class>
                  com.devchronicles.SecurityInterceptor
              </interceptor-class>
          </interceptor-class>
      </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

Первая часть этого XML-файла перечисляет перехватчики, после чего должны быть описаны привязки перехватчиков. Это выполняется в разделе описания компоновки, допускающем использование метасимвола (*), который распространяется на все имена или конкретное имя для создания привязки между перехватчиками и EJB-компонентами. Порядок перечисления перехватчиков также определяет порядок выполнения. Перехватчики, перечисленные в файле EJB-JAR, применяются только к EJB-компонентам в том же модуле.

Порядок выполнения перехватчиков

Если для совета был объявлен более чем один перехватчик, порядок выполнения будет такой: от наиболее общего к наиболее частному. Это значит, что перехватчики уровня по умолчанию будут выполняться перед перехватчиками уровня класса, за которыми будут следовать перехватчики уровня метода.

Такое поведение вполне ожидаемо; однако порядок перехватчиков одного уровня может оказаться чуть более запутанным. Если имеется несколько перехватчиков уровня по умолчанию, то порядок выполнения перехватчиков определяется порядком в файле EJB-JAR.

```
<ejb-jar...>
    <interceptors>
        <interceptor>
            <interceptor-class>
                com.devchronicles.SecurityInterceptor
```

```
            </interceptor-class>
            <interceptor-class>
                com.devchronicles.AnotherInterceptor
            </interceptor-class>
        </interceptor>
    </interceptors>
    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>OrderBean</ejb-name>
            <interceptor-order>
                <interceptor-class>
                    com.devchronicles.SecurityInterceptor
                </interceptor-class>
            </interceptor-order>
            <interceptor-class>
                com.devchronicles.AnotherInterceptor
            </interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

Если имеется несколько перехватчиков уровня класса, то перехватчики следуют порядку, в котором они перечислены в аннотации `@Interceptors`:

```
@Interceptors(SecurityInterceptor.class, AnotherInterceptor.class)
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class SomeBusinessService {
    public void startService(){
    // ...
```

И наконец, если есть несколько перехватчиков уровня метода, то опять-таки перехватчики следуют порядку, в котором они перечислены в аннотации `@Interceptors`:

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class SomeBusinessService {
    @Interceptors(SecurityInterceptor.class, AnotherInterceptor.class)
    public void startService(){
    // ...
```

Если вам необходимо модифицировать порядок выполнения по умолчанию, можете сделать это с помощью пользовательских конфигурационных настроек внутри XML-файла EJB-JAR. Следующий фрагмент переопределяет порядок перехватчиков и обеспечивает пользовательский порядок выполнения:

```
<ejb-jar...>
    <interceptors>
        <interceptor>
            <interceptor-class>
                com.devchronicles.SecurityInterceptor
            </interceptor-class>
```

```
        </interceptor>
    </interceptors>
    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>OrderBean</ejb-name>
            <interceptor-order>
                <interceptor-class>
                    com.devchronicles.SecurityInterceptor
                </interceptor-class>
            </interceptor-order>
            <interceptor-class>
                com.devchronicles.AnotherInterceptor
            </interceptor-class>
            <method>
                <method-name>startService</method-name>
            </method>
        </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

В редких случаях может понадобиться отключить перехватчики. Можете сделать это с помощью аннотаций, как видно в листинге 8.9. Платформа Java EE предоставляет две различные аннотации для отключения отдельно перехватчиков уровня по умолчанию и уровня класса.

Листинг 8.9. Отключение перехватчиков

```
package com.devchronicles.interceptor;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.interceptor.Interceptors;

@ExcludeDefaultInterceptors
@ExcludeClassInterceptors
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class SomeBusinessService {

    public void startService(){
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done...");
    }

    public void startAnotherService() {
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done again...");
    }
}
```

Тем не менее приведенный пример действует только в EJB- и MDB-компонентах, чего может оказаться недостаточно для всех случаев. Благодаря CDI достичь большего совсем не сложно.

CDI-перехватчики

До появления CDI перехватчики были применимы только к EJB- и MDB-компонентам. CDI реализовало огромный потенциал, преобразовав перехватчики в АОП-совместимый функционал, работающий на любом объекте.

Реализация CDI-перехватчиков простая и весьма гибкая. Во-первых, вам понадобится указать привязку. Привязка — пользовательская аннотация, указанная с помощью `@InterceptorBinding`.

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Secure {}
```

`@InterceptorBinding` используется для привязки перехватчиков к целевому коду. Далее вы можете реализовать и аннотировать перехватчик с помощью пользовательской привязки. CDI-перехватчики реализуются аналогично EJB-перехватчикам, единственное значимое различие заключается в использовании осуществляющей привязку аннотации, как можно видеть в листинге 8.10.

Листинг 8.10. Привязка перехватчика с помощью `@Secure`

```
package com.devchronicles.interceptor;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Secure
@Interceptor
public class SecurityInterceptor {

    @AroundInvoke
    public Object doSecurityCheck(InvocationContext context)
                throws Exception {
        // Выполняем проверки безопасности!
        Logger.getLogger("SecurityLog").info(context.getMethod()
                                    .getName()+ "is accessed!");
        String user = context.getContextData.get("user");
        if (user == null) {
            user = (String)context.getParameters()[0];
            context.getContextData.put("user", user)'
        }

        return context.proceed();
    }

    @PostConstruct
```

```
    public void onStart(){
        Logger.getLogger("SecurityLog").info("Activating");
    }
    @PreDestroy
    public void onShutdown(){
        Logger.getLogger("SecurityLog").info("Deactivating");
    }
}
```

Подобно EJB-перехватчикам, необходимо использовать аннотацию `@Interceptor` для преобразования файла класса в перехватчик. Аннотация `@Secure` привязывает перехватчик. И наконец, аннотация `@AroundInvoke` отмечает метод как подлежащий выполнению во время выполнения перехваченных вызовов.

Следующий шаг — реализация аннотации на совете, как показано в листинге 8.11.

Листинг 8.11. Реализация аннотации `@Secure` на совете

```
package com.devchronicles.interceptor;

import javax.interceptor.Interceptors;

@Secure
public class SomeBusinessBean {

    public void startService(){
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done...");
    }

    public void startAnotherService(){
        // Сложная бизнес-логика
        Logger.getLogger("AppLog").info("done again...");
    }
}
```

Для CDI-перехватчиков требуется одна дополнительная стадия описания перехватчиков в файле `beans.xml`. Это один из тех редких случаев, когда вам необходима XML-конфигурация, — она используется для определения порядка выполнения перехватчиков.

Привязки перехватчиков могут включать привязки других перехватчиков, оборачивая таким образом множественные привязки вместе. CDI-контейнер не запустится, если отсутствует файл `beans.xml`:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xsi:schemaLocation=" http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <interceptors>
        <class> com.devchronicles.interceptor.SecurityInterceptor</class>
        <class> com.devchronicles.interceptor.SomeOtherInterceptor</class>
    </interceptors>
</beans>
```

Хотя может показаться, что порядок описания осуществляющих привязку аннотаций влияет на порядок выполнения, в действительности он не имеет значения. Порядок выполнения перехватчиков зависит от порядка описания в файле `beans.xml`.

Совместное использование EJB- и CDI-перехватчиков может вызывать неоднозначность в очередности выполнения. Как правило, EJB-перехватчики выполняются перед выполнением CDI-перехватчиков.

Перехватывающие методы можно назвать источником усложнения системы, но создание множественных привязок и совместное использование EJB- и CDI-перехватчиков выводит сложность системы на новый уровень. Сложные структуры перехватчиков могут наглядно показать незнакомым с этим кодом разработчикам, насколько сложна архитектура приложения.

Где и когда использовать перехватчики

АОП — популярная парадигма программирования, которая может быть полезной при реализации и инкапсуляции сквозной функциональности. Во многих случаях АОП может великолепно проявить себя. Журналирование, аудит, безопасность и другие повторяющиеся события, относящиеся к небизнес-поведению, — отличные кандидаты на его использование.

Перехватчики в Java EE — мощные инструменты, позволяющие вам реализовать АОП, не прибегая к сторонним фреймворкам. С появлением CDI-перехватчиков платформа Java EE стала более совершенной и приобрела большие возможности. Для реализации перехватчика может — в отличие от других перечисленных в книге паттернов — потребоваться XML-конфигурация. Однако эта конфигурация ограничивается лишь обеспечением нужного порядка выполнения, что необходимо и для других паттернов, таких как декораторы.

Перехватчики могут служить для решения многих вопросов сквозной функциональности. Они обеспечивают «чистую» реализацию наряду с инкапсуляцией совместной функциональности. Однако перехватчики, изменяющие бизнес-поведение, могут стать источником проблем. В подобных случаях бизнес-логика распределяется между классом и перехватчиком. Бизнес-методы становятся неудобочитаемыми и вводящими разработчика в заблуждение, поскольку делают невидимой часть логики. Кроме того, это излишне запутывает архитектуру и последовательность выполняемых приложением действий. Вдобавок отладка становится очень запутанной и практически невыполнимой.

При разработке важно сделать код удобочитаемым и самодокументированным, поэтому неправильное употребление перехватчиков может принести большой вред, если оно применено к бизнес-логике. Однако использование перехватчиков для небизнес- и повторяющегося поведения может упростить бизнес-методы и оказать немалую помощь.

Как правило, следует избегать использования перехватчиков для внедрения бизнес-логики или изменения динамического поведения программы. Перехватчики хороши тогда, когда вам нужно выполнять повторяющиеся действия, охватывающие часть методов или классов.

Резюме

АОП — популярная тематика, имеющая много сторонников, но и много врагов. Как и следовало ожидать, это не панацея, решающая все проблемы. Аспекты, если не использовать их должным образом, могут существенно снизить удобочитаемость кода и усложнить общую последовательность выполняемых приложением действий.

Однако они могут стать чудесными инструментами, реализующими дополнительное поведение для существующей базы кода с минимальными усилиями. Вы можете легко включить или отключить их в зависимости от среды выполнения. Например, можно выключить журналирующий аспект во время разработки и активизировать его в тестовой среде.

Платформа Java EE предоставляет простые перехватчики, поддерживающие аннотации и требующие, за исключением особых случаев, лишь небольшой XML-конфигурации. Вы можете использовать перехватчики в контексте как EJB, так и MDB, на уровне как класса, так и метода. Вы можете также объявить перехватчики уровня по умолчанию, которые нацелены на обработку всех EJB-компонентов, удовлетворяющих заданным условиям. Уровень по умолчанию и установление порядка выполнения требуют внесения небольших конфигурационных настроек в XML-файле EJB-JAR.

CDI придает перехватчикам великолепную масштабируемость и функциональные возможности. Вы можете легко модифицировать CDI-перехватчики в соответствии с требованиями заказчика и использовать их более удобным способом с помощью аннотации `@InterceptorBinding`. Можете использовать привязки перехватчиков для оборачивания других привязок, формируя цепочку подлежащих выполнению перехватчиков. CDI-перехватчики требуют только минимальной XML-конфигурации, чтобы помочь CDI-контейнеру определить порядок выполнения.

EJB- и CDI-перехватчики могут работать по отдельности или вместе, согласованно. Они предоставляют всю необходимую для реализации АОП функциональность без необходимости применения сторонних фреймворков.

Надлежащее использование перехватчиков создает прекрасно построенные приложения со стройными потоками выполнения. Когда появляется необходимость реализовать перехватчики, убедитесь, что они не меняют бизнес-логику и содержат логику приложения.

9 Асинхронность

В этой главе:

- введение в асинхронное программирование;
- асинхронное программирование с применением потоков;
- использование асинхронного программирования в компонентах;
- когда и где лучше использовать асинхронное программирование.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 09.zip, каждый из них назван в соответствии с наименованиями в тексте.

Хотя асинхронное программирование не всегда рассматривают как паттерн проектирования, на протяжении последнего десятилетия оно было популярной моделью программирования, имеющей большое значение. Модель асинхронного программирования основывается на многопоточности и выполнении определенной функциональности в отдельных потоках. Преимуществами методов асинхронного программирования пользуются отнюдь не только многопоточные среды и языки программирования, но и однопоточные платформы, такие как популярная серверная JavaScript-платформа NodeJS, которые успешно применяют принципы асинхронного программирования.

ПРИМЕЧАНИЕ

Паттерн «Асинхронность» также называют неблокирующим выполнением метода, поскольку вызываемый метод не блокирует вызывающий. .

Платформа Java с самого начала была спроектирована с учетом поддержки многопоточности. Однако она не сумела обеспечить простой способ осуществления асинхронных обращений. Интерфейс `Future<T>`, появившийся в платформе Java EE 5, стал первой попыткой языка Java реализовать асинхронное программирование, но он был громоздким и сложным в использовании. Последующие версии Java представили аннотацию `@Asynchronous`. Асинхронный сервлет предоставил гораздо лучший набор инструментов для асинхронного программирования.

Что такое асинхронное программирование

Паттерн программирования «Асинхронность» — особый, хорошо интегрированный случай множественных потоков. Вследствие самой сущности потоков многопоточные модели нуждаются в системах уведомления и зависят от шаблонного кода для запуска потоков.

Асинхронные обращения используются даже в одногопоточных средах, таких как NodeJS. Почти все пользовательские интерфейсы поддерживают асинхронное выполнение для удержания UI в активном, реагирующем на действия пользователя состоянии. Первая буква «A» в аббревиатуре AJAX[1] — движущей силе Web 2.0 — означает «асинхронный».

Тем не менее асинхронное программирование может быть полезным и в других местах, помимо пользовательских интерфейсов, обычно на серверной стороне. Ни J2SE, ни J2EE не предоставляли встроенной «легкой» реализации для асинхронного программирования. С появлением платформы Java EE 5 был выпущен фреймворк для параллелизма (Concurrency Framework), основанный на JSR166. JSR166 включал множество утилит, делавших асинхронное программирование не только возможным, но и более легким и лучше управляемым. Интерфейс `Future<T>` также предоставил разработчикам способ реализации асинхронного выполнения метода.

Тем временем Spring представил вниманию разработчиков асинхронные вызовы методов, активизируемые с помощью аннотаций. Платформа Java EE не включала такое удобное решение вплоть до версии 6. Аннотация `@Asynchronous` появилась с выходом платформы Java EE 6 и предоставила удобную возможность реализации асинхронного выполнения метода.

Паттерн «Асинхронность». Асинхронное программирование не указано в числе паттернов проектирования ни в книге GoF, ни в «Паттернах проектирования». Если бы оно там присутствовало, его описание могло бы быть таким: «Обеспечивает способ вызова метода без блокирования вызывающего метода».

Сама сущность выполнения методов заключается в блокировании вызывающего вплоть до завершения выполнения вызванного метода. Такое поведение очевидно и вполне ожидаемо, однако не во всех случаях желательно. Почти все UI-фреймворки и веб-платформы основаны на неблокирующих запросах.

ИСТОРИЯ ИЗ ПРАКТИКИ

Мы получили задачу разработать для одной телекоммуникационной компании веб-портал по обслуживанию клиентов. При разработке портала мы реализовали инфраструктуру подробного журналирования. Мы не использовали сохранение журнала в базе данных, чтобы гарантировать, что журналирование будет быстрым и отказоустойчивым в моменте недоступности базы данных. У нас получилась надежная система с низким временем отклика, и мы были весьма довольны достигнутым.

Позднее нас попросили регистрировать каждое действие пользователя в таблицу базы данных наряду с определенной, относящейся к пользователю информацией. Предложенная нам для работы база данных имела репутацию медленной и постоянно перезагружающейся из-за аварийных сбоев. Это были плохие новости для нашей быстрой и надежной системы журналирования. Нам теперь нужно было проводить рефакторинг своей системы с учетом ненадежности базы данных.

---

[1] *Garret Jessie James.* AJAX (Asynchronous JavaScript and XML): A New Approach to Web Applications. — http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/.

Представьте, что произошло бы, если бы действия пользователя регистрировались в момент сбоя базы данных. Синхронное обращение к системе журналирования заблокировало бы ответ пользователю до восстановления соединения или истечения времени ожидания. Пользователю пришлось бы ждать, что было неприемлемым.

Мы не хотели, чтобы пользователь ждал ответа базы данных, и не хотели демонстрировать ошибку базы данных в клиентской части, особенно учитывая то, что мы всего лишь регистрировали статистические данные. После реализации и тестирования всех классов DAO мы добавили аннотацию @Asynchronous и были готовы начинать эксплуатацию системы.

Как это обычно бывает, мы были вполне уверены в своем тщательно протестированном инсталляционном пакете и предпочли пойти домой спать, а не проводить ночь с администраторами сервера, выполнявшими развертывание. На следующее утро мы получили по электронной почте письмо, уведомлявшее нас, что приложение было запущено.

Вскоре мы обнаружили, что файлы журналов нашего сервера полны ошибок, показывающих, что соединение с базой данных было недоступно. Мы связались с администраторами сервера и быстро выяснили, что администраторы базы данных забыли создать журнальные таблицы в действующей базе данных. Таблицы быстро были созданы, и все было хорошо до момента, когда база данных начала испытывать проблемы с производительностью и частыми перезагрузками сервера (как и можно было ожидать, исходя из ее репутации).

Наше приложение иногда не сохраняло некоторые некритические данные, но ни разу не снизило производительности. Во всех случаях, когда возникала проблема с базой данных и происходил сбой журналирования, действия пользователя уже были завершены и он не замечал сбоя благодаря асинхронным обращениям к журналирующей функциональности.

Асинхронное программирование — замечательный инструмент для разделения задач, которым не требуется взаимодействовать друг с другом.

Паттерн «Асинхронность» основан на подходе «самонаведения», когда операция выполняется параллельно или таким образом, при котором не блокируется выполняющий поток, а результат проверяется по мере готовности. Обычно асинхронный подход использует параллельное выполнение. Диаграмма классов не вполне точно отражает суть такого подхода, лучше будет продемонстрировать его с помощью блок-схемы (рис. 9.1).

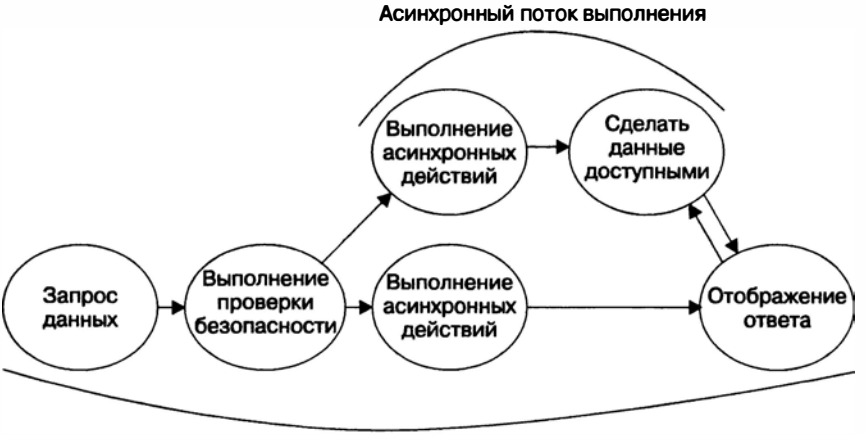


Рис. 9.1. Блок-схема асинхронности

Реализация паттерна «Асинхронность» в простом коде

Язык Java с самого начала поддерживал потоки, которые вы легко можете использовать для выполнения асинхронного кода:

```
public class AsyncRunnable implements Runnable {
    public void run() {
        System.out.println("Running!");
    }
}
```

Для выполнения класса Runnable инициализируйте его в потоке и вызовите метод run, обратившись к методу start() только что созданного потока.

```
(new Thread(new AsyncRunnable())).start();
```

Хотя предпочтительнее запускать поток по предыдущему примеру, другой подход состоит в расширении класса thread и переопределении метода run():

```
public class AsyncThread extends Thread {
    public void run() {
        System.out.println("Running!");
    }
}
```

Для выполнения класса инстанцируйте его и затем вызовите метод start():

```
(new HelloThread()).start();
```

При работе с потоками часто используются две основные операции: sleep() и join(). Обе генерируют исключение InterruptedException.

Метод sleep() дает потоку возможность бездействовать определенный период, задаваемый в миллисекундах. Следующий фрагмент кода переводит текущий поток в режим ожидания на одну секунду.

```
Thread.sleep(1000);
```

Метод join() заставляет один поток ожидать завершения выполнения второго потока. Представьте себе поток t1, которому необходим ресурс другого потока, t2. Чтобы заставить t1 ожидать завершения t2, присоедините его к потоку t2, как показано в следующем фрагменте кода:

```
t2.join();
```

Один из наиболее известных и широко используемых подходов к асинхронному программированию в языке Java — применение интерфейса Future<T>. Этот интерфейс делает возможным использование объекта-заместителя, предоставляющего ссылку на будущий объект. Поскольку фреймворки, обеспечивающие параллелизм, не предоставляют основанную на аннотациях поддержку асинхронного выполнения, интерфейс Future работает в сочетании с ExecutorService — составной частью вышеупомянутых фреймворков.

Следующий пример использует сервис выполнения для завершения задания, в то время как он возвращает ссылку на интерфейс Future с соответствующим родовым типом:

```
ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String> reference = executor.submit(
    new Callable<String>() {
        public String call() {
            return "Hello!!";
        }
    }
);
//..
if (reference.isDone())
    System.out.println(reference.get());
```

Класс FutureTask — реализация интерфейса Future<T>, которая реализует также интерфейс Runnable и доступна для непосредственного выполнения:

```
FutureTask<String> reference = new FutureTask<String>(
    new Callable<String>() {
        public String call() {
            return "Hello!!";
        }
    }
);

executor.execute(reference);
```

Вы можете отменить это выполнение, вызвав метод cancel(boolean mayInterruptIfRunning). Если параметру mayInterruptIfRunning присвоено значение true, то обращение к методу SessionContext.wasCancelled() возвращает true. В противном случае обращение к методу SessionContext.wasCancelled() возвращает false. Для проверки состояния отмены вызова вы можете использовать метод isCancelled(), возвращающий true при успешной отмене.

Фреймворки, основанные на JSR133[1] и реализующие параллелизм, предоставляют замечательные инструменты для работы с потоками и для параллельного программирования, например BlockingQueues. Эти темы выходят за рамки данной главы. Для дальнейшего изучения обратитесь к книге *Java Concurrency in Practice*[2]. Фреймворк Fork/Join, появившийся в Java EE 7, также воплощает серьезные изменения для асинхронного и параллельного программирования на Java.

---

[1] Java Memory Model and Thread Specification («Модель памяти Java и спецификация потоков»). — https://jcp.org/aboutJava/communityprocess/final/jsr133/index.html. — *Примеч. пер.*

[2] *Goetz Brian, Holmes David, Lea Doug, Peierls Tim, Bloch Joshua.* Java Concurrency in Practice. — Addison-Wesley Professional, 2006.

Асинхронное программирование в Java EE

Поскольку платформа J2EE не сумела предоставить встроенную поддержку парадигмы асинхронного программирования (за исключением `Timer`), сторонние фреймворки, такие как Spring и Quartz, вмешались и заполнили этот пробел. Недочет был исправлен в Java EE 5, это была первая версия Java с готовой для использования поддержкой паттерна программирования «Асинхронность».

Асинхронные компоненты

Платформа Java EE поддерживает асинхронное программирование несколькими способами. Простейший способ реализовать паттерн «Асинхронность» в Java EE, как и следовало ожидать, — применить аннотацию. Аннотирования метода с помощью `@Asynchronous` достаточно, чтобы указать контейнеру Java EE асинхронно выполнить вызванный метод в отдельном потоке. Чтобы увидеть асинхронную аннотацию в действии, вернемся к примеру журналирующего компонента-одиночки из главы 4 и добавим асинхронную аннотацию для изменения его поведения по умолчанию. Листинг 9.1 демонстрирует пример асинхронного компонента.

Листинг 9.1. Пример асинхронного компонента

```
package com.devchronicles.asynchronous;

import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import java.util.logging.Logger;
import javax.ejb.Asynchronous;

@Startup
@Singleton
public class MyLoggingBean {

    private Logger logger;

    @PostConstruct
    public void start(){
        logger = Logger.getLogger("MyGlobalLogger");
        logger.info("Well, I started first!!!");
    }

    public void logInfo(String msg){
        logger.info(msg);
    }

    @Asynchronous
    public void logAsync(String msg){
        logger.info(msg);
    }
}
```

Метод `logAsync()`, в отличие от его аналога `logInfo()`, выполняется асинхронно. Чтобы наблюдать асинхронное поведение, добавим вызовы `Thread.sleep()`:

```
public void logInfo(String msg){
    logger.info("Entering sync log");

    try{
        Thread.sleep(1000);
    } catch (InterruptedException e){}

    logger.info(msg);
}

@Asynchronous
public void logAsync(String msg){
    logger.info("Entering async log");
    try{
        Thread.sleep(13000);
    } catch (InterruptedException e){}

    logger.info(msg);
}
```

Наконец, создадим новый компонент для поочередного вызова обеих функций, как показано в листинге 9.2.

Листинг 9.2. Рефакторинг листинга 9.1 для учета обеих функций

```
package com.devchronicles.asynchronous;

import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Startup
@Singleton
public class TestLogging {

    @EJB
    MyLoggingBean logBean;

    @PostConstruct
    public void testLoggers(){
        System.out.println("call async");
        logBean.logAsync("Log Async");

        System.out.println("call sync");
        logBean.logInfo("Log Sync");

        System.out.println("finished");
    }
}
```

Типичный вывод в консоли будет таким:

```
> call async
> Entering async log
> call sync
> Entering sync log
> Log Sync
> finished
> Log Async
```

Когда вы вызываете метод `testLoggers()`, происходят вызовы методов `logAsync()` и `logSync()`. Оба метода дают их потокам выполнения возможность бездействовать заданное время. Как можно видеть из вывода в консоли, асинхронный метод был вызван, после чего на длительное время перешел в спящий режим, но не заблокировал при этом выполнение синхронного метода. Синхронный метод ненадолго перешел в режим ожидания, но вернул управление в вызывающий метод и напечатал `finished`. И наконец, асинхронный метод «проснулся» и завершил журналирование печатью `Log Async` в консоли.

Этот пример ясно показывает, что асинхронный вызов не останавливает поток вызывающей программы, а также не останавливает синхронный метод. Однако когда синхронный метод переходит в режим ожидания, вызывающий метод ждет его окончания. Аннотация `@Asynchronous` представляет собой простой способ реализации асинхронного поведения и может быть добавлена практически к любому методу в любой момент во время или после разработки.

Асинхронные сервлеты

До сих пор вы видели, что можете преобразовать любой метод компонента в асинхронный метод. Теперь вы увидите, как заставить асинхронно функционировать сервлет. Без наличия асинхронной поддержки в сервлетах нелегко отвечать требованиям асинхронности при веб-разработке.

Спецификация Servlet 3.0 (JSR 315) внесла серьезные усовершенствования в интерфейсы программирования веб-приложений (API) языка Java. С появлением JSR 315 спецификации сервлетов были обновлены (после длительного ожидания) для поддержки асинхронной модели выполнения, удобной конфигурации, подключаемости и других мелких улучшений.

Асинхронные сервлеты основываются на ключевом усовершенствовании в HyperText Transfer Protocol (HTTP) 1.1, сделавшем возможными постоянные соединения. В HTTP 1.0 каждое соединение использовалось для отправки и получения только одной пары «запрос/ответ»; в то же время HTTP 1.1 позволяет веб-приложениям поддерживать соединение в активном состоянии и посылать множественные запросы. При стандартной реализации прикладная часть Java потребовала бы отдельного потока, постоянно прикрепленного к HTTP-соединению. Однако неблокирующее API ввода-вывода языка Java (Java Nonblocking I/O, NIO) возвращает между активными запросами потоки «в оборот» благодаря новым возможностям NIO. На сегодняшний день все совместимые со спецификациями Servlet 3.0 веб-серверы имеют встроенную поддержку Java NIO.

Для чего вам может понадобиться от сервлетов подобное поведение? Для серверных систем характерны длительные операции, такие как соединение с другими серверами, выполнение сложных вычислений и осуществление операций транзакционных баз данных. Однако сущность веб-страниц требует как раз обратного. Веб-пользователи ожидают короткого времени отклика и UI, функционирующего даже в случае незавершенных операций серверной части. AJAX взял на себя решение этой проблемы для браузеров и начал революцию Web 2.0.

Спецификация Servlet 3.0 предоставила метод `startAsync()`, сделавший доступными асинхронные операции. Листинг 9.3 показывает пример этого.

Листинг 9.3. Пример использования метода startAsync()

```
package com.devchronicles.asynchronous;

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet(urlPatterns={"/async"}, asyncSupported=true)
public class AsyncServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                        throws IOException, ServletException {

        final AsyncContext asyncContext = req.startAsync();
        final String data;

        asyncContext.addListener(new AsyncListener() {

            @Override
            public void onComplete(AsyncEvent event) throws IOException {
                AsyncContext asyncContext = event.getAsyncContext();
                asyncContext().getWriter().println(data);
            }

            @Override
            public void onTimeout(AsyncEvent event) throws IOException {
                // Код опущен для краткости
            }

            @Override
            public void onError(AsyncEvent event) throws IOException {
                // Код опущен для краткости
            }

            @Override
            public void onStartAsync(AsyncEvent event) throws IOException {
                // Код опущен для краткости
            }
```

```
        });

        new Thread() {
            @Override
            public void run() {
                asyncContext.complete();
            }
        }.start();

        res.getWriter().write("Results:");
        // Чтение данных из базы данных
        data = "Queried data...";
        // Переводим поток в режим ожидания на некоторое время...
    }
}
```

Сервлет выводит `Results:` и далее выводит полученные из базы данные, которыми в этом сценарии является простая строка. Вам необходимо инициализировать отдельный поток. Метод `onComplete` класса `AsyncListener` выполняется только после завершения выполнения. В классе `AsyncListener` есть еще несколько методов жизненного цикла:

- `OnStartAsync` — выполняется при запуске асинхронного контекста;
- `OnTimeOut` — выполняется, только если истекает время ожидания;
- `onError` — выполняется, только если была получена ошибка.

Спецификация Servlet 3.1 предоставляет более простой способ реализации асинхронных сервлетов путем использования управляемых пулов потоков и сервиса выполнения. В листинге 9.4 задействуется `ManagedThreadFactory` для создания нового потока.

Листинг 9.4. Пример, использующий `ManagedThreadFactory`

```
package com.devchronicles.asynchronous;

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet(urlPatterns="/async",asyncSupported=true)
public class AsyncServlet extends HttpServlet {

    @Resource
    private ManagedThreadFactory factory;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException {

        final AsyncContext asyncContext = req.startAsync();
        final PrintWriter writer = res.getWriter();
```

```
        Thread thread = factory.newThread(new Runnable() {

            @Override
            public void run() {
                writer.println("Complete!");
                asyncContext.complete();
            }
        });
        thread.start();
    }
}
```

Этот пример создает новый поток с процессом, требующим больших затрат времени, и наконец вызывает функцию complete из asyncContext. ManagedThreadFactory служит в качестве доступного потока из пула, который вам необходимо запустить явным образом.

Другой подход состоит в передаче ManagedExecutorService асинхронного Runnable вместо создания и последующего запуска потока в сервлете. Делегирование ExecutorService вопросов организации поточной обработки обеспечивает более «чистый» код, как вы увидите в листинге 9.5.

Листинг 9.5. Пример делегирования ExecutorService

```
package com.devchronicles.asynchronous;

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet(urlPatterns="/async", asyncSupported=true)
public class AsyncServlet extends HttpServlet {
    @Resource
    private ManagedExecutorService executor;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException {

        final AsyncContext asyncContext = req.startAsync(); final
        PrintWriter writer = res.getWriter();
        executor.submit(new Runnable() {
            @Override
            public void run() {
                writer.println("Complete!");
                asyncContext.complete();
            }
        });
    }
}
```

Хотя это всего лишь на одну строку меньше, чем в предыдущем листинге, листинг 9.5 делегирует создание и запуск потока `ExecutorService` и имеет дело только с сервлетным кодом.

Асинхронные сервлеты легче для понимания и программирования и оказывают немедленный эффект на динамическое поведение, поскольку напрямую «переключают» на модель асинхронного выполнения. Асинхронные сервлеты обеспечивают «чистую» реализацию без большого количества шаблонного кода.

Где и когда применять асинхронное программирование

Вы можете использовать паттерн «Асинхронность» почти везде, где необходимо вернуть ответ до полного завершения выполнения. Этот подход может отличаться от асинхронного выполнения менее важных функций приложения, таких как журналирование или уведомление пользователя о требующих больших затрат времени операциях. Асинхронное программирование делает критический путь выполнения более коротким за счет делегирования подзадач другим потокам. Результатом будет более короткое время отклика.

Асинхронные аннотации — простой способ реализовать асинхронные методы или преобразовать существующие. Каждый метод, отмеченный аннотацией `@Asynchronous`, запускается в отдельном потоке без блокирования выполнения текущего потока. Такое поведение идеально подходит для обстоятельств, не влияющих на главный исполнительный цикл, а требующих выполнения в серверной части системы. Примеры этого — журналирование и средства сопровождения.

Вы можете использовать асинхронные сервлеты почти во всех современных веб-приложениях. Они обеспечивают неблокирующее асинхронное поведение без особой необходимости в AJAX. Асинхронные сервлеты могут быть полезны, когда необходима серверная операция размещения данных, такая как обновление информации или доставка сообщения.

Поскольку каждое асинхронное выполнение операции требует нового потока, виртуальной машине Java (Java Virtual Machine, JVM) приходится выполнять тем больше переключений контекста, чем больше реализуется асинхронных методов. Большое количество переключений контекста вызывает «голодание»[1] потоков, что приводит к худшей производительности, чем у синхронной реализации.

Представьте, что вы читаете книгу. В промежутках между чтением вам необходимо помнить историю, персонажей и номер последней страницы, которую вы читали. Если вы читаете две книги в одно и то же время, то можете завершить чтение второй, более короткой книги без необходимости дочитывать начатую вами более длинную. Время, потраченное на смену контекста с одной книги на другую, вполне приемлемо.

Чтение шести книг одновременно потребовало бы немалых усилий. Оно могло бы потребовать такого количества смен контекста, что вы не смогли бы дочитать

[1] См.: https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html. — *Примеч. пер.*

ни одну из книг в ожидаемый срок и переключались бы с одной книги на другую, не продвигаясь ни в одной из них.

Асинхронное программирование решительно изменило порядок выполнения, а значит, и отладки. Поскольку отладка основывается на приостановке выполнения и затем пошаговом, строка за строкой, выполнении, стало намного сложнее понимать динамическое поведение программы и имитировать то, что происходит на самом деле.

JVM определяет порядок выполнения потоков непосредственно во время этого процесса. Практически невозможно смоделировать такое же поведение вследствие различий в доступности ресурсов в среде тестирования и разработки. Если оно на самом деле вам не требуется, асинхронное выполнение только прибавит нежелательной сложности.

Организация поточной обработки и асинхронное выполнение может быть замечательным инструментом, но только если используется должным образом, без ресурсного «голодания». Выполнять не блокирующие друг друга части программы асинхронно — хорошая идея, но не для каждого метода.

Резюме

В эпоху многоядерности и Web 2.0 асинхронное программирование использует вычислительные ресурсы, делегирует неблокирующие задания — и приводит к более быстродействующим и быстрее откликающимся пользовательским интерфейсам. Даже если ваше приложение не реализует паттерн «Асинхронность», большинство серверов приложений и виртуальных машин Java используют внутри себя для многих операций асинхронное выполнение посредством пулов потоков. Использование этих доступных потоков и ресурсов существенно влияет на производительность и время отклика вашего приложения.

Организация поточной обработки была «полноправным гражданином» с самых первых дней появления Java, но применение потоков для запуска асинхронных заданий было сложным и не всегда безопасным в управляемых сервером контейнерах. С выходом фреймворка для параллелизма (Concurrency Framework) Java передал в руки разработчиков огромный набор инструментов.

Платформа Java EE последовала этой тенденции, предоставив удобную в использовании и реализации асинхронную модель программирования, основанную на аннотациях. Добавление аннотации `@Asynchronous` указывает контейнеру выполнять функцию асинхронно.

API сервлетов представило важные изменения в версии 3.0 и дальнейшие усовершенствования в версии 3.1. Новое API сервлетов использует новый неблокирующий ввод-вывод Java для эффективной поддержки асинхронного веб-программирования. Хотя предыдущим подходам для привязки к потоку была необходима пара «запрос/ответ», новая модель может использовать или освобождать потоки с помощью внутреннего пула потоков, обеспечиваемого контейнером.

Сегодня платформа Java EE предоставляет все требуемые для работы асинхронного кода инструменты без необходимости в сторонних фреймворках, таких как Spring или Quartz. Это делает паттерн «Асинхронность» великолепным средством

для реализации, если вы хотите выполнять неблокирующий код асинхронно и практически без шаблонного кода.

Упражнения

1. Создайте реализацию сеансового компонента, имеющего асинхронный метод.
2. Разработайте простую функциональность, использующую асинхронную методологию, для сохранения журнала приложения в базе данных.
3. Используйте асинхронные функциональные возможности Servlet 3.0 для проектирования асинхронного веб-сервиса.

10 Сервис таймера

В этой главе:

- развитие сервисов таймеров;
- автоматические таймеры;
- программные таймеры;
- задание расписания с помощью планировочных выражений;
- таймеры и транзакции.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 10.zip, каждый из них назван в соответствии с наименованиями в тексте.

Бизнес-приложениям приходится выполнять задачи, основываясь как на событиях в расписании, так и на календарном плане, генерируются ли еженедельные отчеты о деятельности пользователей, повторно заполняется кэш после очистки или клиенту отправляется электронное письмо с напоминанием. Существует множество сценариев использования. Сервис таймера дает вам возможность программировать события таймера в заданные моменты или через постоянные интервалы времени.

Эта глава продемонстрирует вам, как выполнить конфигурацию сервиса таймеров с помощью как автоматических, так и программных методик и как запланировать задания посредством cron-подобных планировочных выражений.

Что такое сервис таймера

Можете ли вы представить себе необходимость просыпаться каждое утро для проверки по часам — не время ли вставать? Вероятно, нет. Даже до изобретения будильников люди использовали солнечный свет или петухов, чтобы проснуться вовремя. Но петухов и солнце нельзя настроить. Отсутствие возможности настройки привело к изобретению одного из самых важных приспособлений в современной жизни — будильника. Сегодня даже простейшие мобильные телефоны и трекеры для занятий фитнесом оснащены будильником, который можно настроить на различное время и различные дни, и даже функцией включения по таймеру.

Длительное время ни платформа Java SE, ни Java EE не предоставляли встроенного решения для контролируемых по времени операций. Этот недостаток поддержки был заполнен разработками Java-сообщества, основанными на открытом исходном коде.

Обычно контролируемые по времени задания планируются с помощью сторонних инструментов, таких как Quartz[1], но подобные инструменты обычно непросты в использовании. Сторонние инструменты требуют от вас скачивания и установки библиотек, реализации интерфейсов и выполнения настроек в XML-файлах. Они далеко не просты.

К счастью для вас, частично из-за сложностей, с которыми столкнулись разработчики, пытавшиеся применять сторонние библиотеки, в спецификации EJB 2.1 появилось средство планирования. Этот сервис таймера удовлетворял простейшие сценарии использования. А для более сложных случаев по-прежнему был Quartz. На самом деле Quartz практически стал де-факто стандартом контролируемых по времени операций в языке Java.

В Java SE отсутствует реализация таймеров по умолчанию. Вы можете использовать Quartz как на платформе Java SE, так и на платформе Java EE, но применение Quartz — отдельная тема, выходящая за рамки этой книги. Так что в текущей главе будет пропущена реализация для Java SE и мы перейдем прямо к Java EE.

В спецификации EJB 3.2 (текущая редакция) сервисы таймеров получили дальнейшее развитие. Были представлены аннотации `@Schedule` и `@Schedules` и cron-подобные календарные выражения. На сегодняшний день удовлетворяются все сценарии использования, кроме разве что самых экзотических. Сервис таймера запускается в контейнере как сервис и регистрирует EJB-компонент для обратных вызовов. Он отслеживает существующие таймеры и их расписания и даже заботится о сохранении таймера в случае отключения или сбоя сервера. Единственное, что теперь нужно сделать разработчику, — задать расписание таймера.

Сервисы таймеров прошли длительный цикл разработки. Обзор усовершенствований подытожен в табл. 10.1.

Таблица 10.1. Цикл разработки сервисов таймеров

Версии Java и EJB	Разработка
EJB 2.1 Java 1.4 (ноябрь 2003 года)	ejbTimer реализует интерфейс TimedObject. TimerService доступен через метод EJBContext. Бизнес-логика должна содержаться в методе ejbTimeout
EJB 3.0 Java 5 (май 2006 года)	Объект TimerService внедряется посредством прямой инъекции с помощью аннотации @Resource. Бизнес-логика должна быть помещена в метод, аннотированный @Timeout. Расписание устанавливается с помощью даты, длительности, объекта ScheduleExpression или конфигурационного XML-файла. В Java EE 6 на них ссылаются как на программные таймеры

Продолжение

[1] Quartz Job Scheduler: http://www.quartz-scheduler.org/.

Таблица 10.1 *(продолжение)*

Версии Java и EJB	Разработка
EJB 3.1 Java 6 (декабрь 2009 г.)	Контейнер инициализирует объект TimerService автоматически, выполнение инъекции не требуется. Бизнес-логика должна быть помещена в метод, аннотированный @Schedule или @Schedules. Расписание устанавливается в атрибутах аннотаций и с помощью календарных выражений EJB-таймера
EJB 3.2 Java 7 (июнь 2013 г.)	Группа EJB Lite расширена за счет включения сервиса непостоянных EJB-таймеров. Включены усовершенствования API TimerService, делающие возможным доступ ко всем активным таймерам в EJB-модуле. Убраны ограничения на Timer и TimerHandle, заставлявшие использовать ссылки только внутри компонента

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Недавно меня пригласили консультантом в веб-проект, страдавший от периодических проблем с производительностью. Эти проблемы появились недавно — как раз тогда, когда начало увеличиваться количество посетителей сайта.

Разработчики выбрали базу данных NoSQL для хранения GPS-данных о местоположении посетителей сайта. Это было неплохим решением, поскольку такой конкретный NoSQL-склад данных был хорошо приспособлен к обработке геолокационных запросов.

Запросы регулярно выполнялись в базе данных, объединявшей информацию из наборов данных о местоположении и агрегирующей ее для формирования отчетов. Эти отчеты включали ежедневную статистику посещений и выполнялись каждый день кем-то из младшего персонала.

После проверки я обнаружил, что возникавшие у приложения проблемы с производительностью совпадали с моментом запуска формирования отчетов. Дополнительная нагрузка на базу данных, вызванная формированием отчетов, была причиной ухудшения производительности.

Решение было очень простым: запускать отчеты, когда база данных использовалась меньше всего. После просмотра отчетов о загрузке базы данных я решил, что оптимальным временем для запуска отчетов будет 03:00 GMT. Понятно, что я не мог просить кого-то из младшего персонала приходить на работу в 3 часа утра, так что решил автоматизировать формирование отчетов и настроить timerservice для запуска формирования отчетов.

Многие задания лучше всего запускать в нерабочее время по той же, изложенной выше, причине. Повторное заполнение кэшей данных — обычный пример «тяжелого» процесса, который следует запускать в тот момент, когда влияние на производительность сайта будет минимальным.

Реализация таймеров в Java EE

В платформе Java EE 7 существует два типа таймеров: автоматический и программный. Автоматические таймеры устанавливаются при развертывании корпоративного компонента Java (EJB), содержащего метод, аннотированный или `@Schedule(...)`, или `@Schedules(...)`. Аннотированный метод вызывается планировщиком контейнера в заданные моменты времени или через интервалы времени, описываемые в аргументах аннотаций. Такие методы именуются *методами обратного вызова*. Таймер начинает «тикать» сразу же, как только будет развернут EJB-компонент. Программный таймер устанавливается во время выполнения при вызове метода изнутри кода бизнес-логики. Таймер может быть настроен во

время работы и вызван в любое время (или не вызван вообще). Он начинает «тикать», когда этого потребует логика программы.

РЕАЛИЗАЦИЯ СЕРВИСА ТАЙМЕРА

Контейнер EJB реализует сервис таймера. Корпоративный компонент может обратиться к этому сервису тремя способами: посредством внедрения зависимости, через интерфейс EJBContext или путем поиска в пространстве имен интерфейса каталогов и служб именования Java (Java Naming and Directory Interface, JNDI). Мы рассмотрим только способ обращения посредством внедрения зависимости, поскольку он самый новый и наиболее эффективный.

Автоматические таймеры

Контейнер вызывает любой метод, соответствующе аннотированный `@Schedule`, и применяет конфигурацию планировщика, указанную в атрибутах аннотации. Атрибуты аннотации устанавливаются в соответствии с описанными ниже, в подразделе «Выражения таймеров», календарными атрибутами таймеров. Вот простой пример:

```
@Schedule(second="*/1", minute="*", hour="*")
public void executeTask(){
    System.out.println("Task performed");
}
```

В этом фрагменте кода метод `executeTask` аннотируется `@Schedule`; это указывает контейнеру установить во время развертывания таймер, основываясь на значениях времени, указанных в атрибутах аннотации. В этом примере контейнер вызывает метод `executeTask` один раз в секунду.

По умолчанию все состояния таймеров сохраняются и восстанавливаются после отключения или сбоя сервера. Если вы установите необязательный атрибут `persistent` в значение `false`, то таймер будет сбрасываться при перезагрузке сервера. Вы можете указать два дополнительных атрибута: `info` и `timezone`. Если вы укажете `timezone`, то при выполнении таймера будет учитываться часовой пояс, в противном случае будет использоваться часовой пояс сервера. Атрибут `info` позволяет разработчику предоставлять описание момента времени, которое вы можете получить, обратившись к методу `getInfo` интерфейса `Timer`.

```
@Schedule(hour = "23", minute = "59", timezone = "CET",
    info = "Generates nightly report")
public void executeTask(){
    System.out.println("Task performed");
}
```

В предыдущем фрагменте кода метод `executeTask` вызывается в 23:59 по центральноевропейскому времени независимо от часового пояса на сервере, на котором было развернуто приложение. Обращение к методу `getInfo` вернет текст `Generates nightly report`.

Вы можете настроить более сложные таймеры с помощью аннотации `@Schedules` (обратите внимание на множественное число) с множественными выражениями таймеров.

```
@Schedules({
    @Schedule(dayOfMonth = "1"),
    @Schedule(dayOfWeek = "Mon,Tue,Wed,Thu,Fri", hour = "8")
})
public void executeTask() {
    System.out.println("Task performed");
}
```

Этот таймер срабатывает первого числа каждого месяца и в каждый рабочий день в 08:00. Листинг 10.1 показывает законченный пример автоматического таймера.

Листинг 10.1. Простейшая реализация автоматического таймера

```
package com.devchronicles.timer;

import javax.ejb.Schedule;
import javax.ejb.Schedules;

public class PeriodicTimer {

    @Schedules({
        @Schedule(dayOfMonth = "1"),
        @Schedule(dayOfWeek = "Mon,Tue,Wed,Thu,Fri", hour = "8")
    })
    public void executeTask() {
        System.out.println("Task performed");
    }
}
```

Один из недостатков автоматического таймера состоит в том, что его расписание задается во время развертывания и не может быть изменено во время выполнения приложения. К счастью, из этой ситуации есть выход в виде программного таймера, который вы можете установить в любой момент во время выполнения.

Программные таймеры

Программные таймеры создаются во время выполнения путем вызова одного из методов `create...` интерфейса `TimerService`. Вот простой пример:

```
public void setTimer(){
    timerService.createTimer(30000, "New timer");
}
```

Когда код приложения вызывает метод `setTimer`, он создает одноразовый таймер, который после заданной продолжительности времени 30 000 миллисекунд вызывает метод «превышение лимита времени» того же компонента. Метод «превышение лимита времени» распознается по аннотации `@Timeout` и обязан соответствовать определенным требованиям. Он не должен генерировать исключения или возвращать значение. Он также избавлен от необходимости иметь параметры, но если имеет, то параметр должен быть типа `javax.ejb.Time`. Может существовать только один метод «превышение лимита времени».

```
@Timeout
public void performTask() {
    System.out.println("Simple Task performed");
}
```

CDI-контейнер внедряет ссылку на TimerService в переменную экземпляра, аннотированную @Resource. Вот тут контейнер выполняет внедрение в переменную экземпляра timerService:

```
@Resource
TimerService timerService;
```

Если вы соберете предыдущие три фрагмента кода в единый компонент и код приложения обратится к методу setTimer, то вы создадите таймер, который через 30 секунд вызовет метод «превышение лимита времени» performTask. Листинг 10.2 показывает простейшую возможную реализацию программного таймера на платформе Java EE 7.

Листинг 10.2. Простейшая реализация программного таймера

```
package com.devchronicles.timer;

import javax.annotation.Resource;
import javax.ejb.Timeout;
import javax.ejb.TimerService;

public class SimpleProgrammaticTimer {

    @Resource
    TimerService timerService;

    public void setTimer(){
        timerService.createTimer(30000, "New timer");
    }

    @Timeout
    public void performTask() {
        System.out.println("Simple Task performed");
    }
}
```

Существует четыре метода для создания таймеров с десятью сигнатурами в интерфейсе TimerService. Таблица 10.2 показывает пример каждого из них.

Таблица 10.2. Примеры четырех методов для создания таймеров

Пример	Описание
createIntervalTimer(new Date(), 10000, new TimerConfig());	Создает таймер, срабатывающий в заданную дату и после этого каждые 10 секунд
createSingleActionTimer(1000, new TimerConfig());	Создает таймер, срабатывающий через 1 секунду
createTimer(30000, "Created new programmatic timer");	Создает таймер, срабатывающий через 30 секунд
createCalendarTimer(new ScheduleExpression(). second("*/10").minute("*").hour("*"));	Создает таймер, срабатывающий каждые 10 секунд

Все методы, кроме createCalendarTimer, могут принимать в качестве первого параметра как длительность в миллисекундах, так и дату. Он устанавливает момент, в который срабатывает таймер.

Вот пример:

```
SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy 'at' HH:mm");
Date date = formatter.parse("26/01/2015 at 17:56");
timerService.createSingleActionTimer(date, new TimerConfig());
```

В этом фрагменте кода метод «превышение лимита времени» срабатывает в 17:56 26 января 2015 года.

Если необходим таймер по расписанию, вы можете использовать метод createCalendarTimer. Он принимает в качестве параметра объект ScheduleExpression, в котором расписание устанавливается с помощью значений, описанных в следующем разделе «Выражения таймеров».

```
ScheduleExpression expression = new ScheduleExpression();
expression.second("*/10").minute("*").hour("*");
timerService.createCalendarTimer(expression);
```

В этом фрагменте кода расписание установлено на срабатывание таймера каждые 10 секунд каждой минуты каждого часа.

Все методы для создания таймеров возвращают объект Timer, представляющий собой таймер. У этого объекта есть метод getHandle, возвращающий сериализуемый дескриптор таймера. Объект дескриптора может храниться в базе данных или оперативной памяти.

Далее вы можете получить объект дескриптора и вернуть ссылку на таймер, вызвав метод getTimer. Имея этот объект, вы можете извлекать полезную информацию относительно таймера.

Несложно получить информацию о поведении таймера. Вы можете извлечь подробности относительно расписания таймера, вызвав метод getSchedule. Он возвращает объект ScheduleExpression, у которого для каждого атрибута имеется метод чтения.

Например, getMinute() возвращает значение, установленное для атрибута минуты. Метод getNextTimeout возвращает момент времени, когда таймер сработает в следующий раз, тогда как getTimeRemaining возвращает количество миллисекунд до окончания функционирования таймера.

Метод isCalendarTimer возвращает true, если таймер был установлен путем конструирования объекта ScheduleExpression. Вы должны вызвать его перед методом getSchedule, чтобы убедиться, что таймер был создан именно таким образом, в противном случае getSchedule сгенерирует исключение IllegalStateException.

Вы можете выяснить информацию о состоянии сохранения таймера с помощью метода isPersistent. Аналогично вы можете получить информацию о времени, вызвав метод getInfo.

Таймеры автоматически отключаются, когда истекает срок их действия. Контейнер отключает одноразовые таймеры, и вы можете отключить таймер по расписанию, вызвав метод cancel объекта Timer.

Выражения таймеров

Как программные, так и автоматические таймеры могут использовать календарные атрибуты таймеров. Таблица 10.3 показывает диапазоны атрибутов, применяемых для настройки таймеров. Для автоматических календарных таймеров вы устанавливаете атрибуты в аннотации, тогда как программные календарные таймеры используют для задания значений календарных атрибутов методы класса `ScheduleExpression`.

Таблица 10.3. Календарные выражения

Атрибут	Описание	Допустимые значения
second	Одна секунда или более в пределах минуты	От 0 до 59
minute	Одна минута или более в пределах часа	От 0 до 59
hour	Один час или более в пределах дня	От 0 до 23
dayOfWeek	Один день или более в пределах недели	От 0 до 7 (0 и 7 относятся к воскресенью). Sun, Mon, Tue, Wed, Thu, Fri, Sat
dayOfMonth	Один день или более в пределах месяца	От 1 до 31. От –7 до –1 (дней до конца месяца). Last (последний). 1st, 2nd, 3rd — nth. От Sun до Sat
month	Один месяц или более в пределах года	От 1 до 12. От Jan до Dec
year	Конкретный календарный год	2014, 2015...

Заслуживает упоминания, что значение по умолчанию для атрибутов времени — `0` (ноль), а для нечисловых значений — `*` (звездочка).

Эта таблица была взята из руководства Oracle Java EE 7[1]. Синтаксис здесь cron-подобный и должен быть знаком большинству программистов. Есть несколько интересных особенностей.

Символ звездочки является «заполнителем» для всех возможных значений данного атрибута. Например, чтобы запланировать срабатывание каждый час, вы бы использовали выражение `hour="*"` для настраиваемых с помощью аннотаций таймеров. Для программных таймеров вы бы вызвали метод `hour("*")` для экземпляра класса `ScheduleExpression`.

Вы можете выражать значения каждого атрибута в виде списка или диапазона. Например, выражение `dayOfMonth="1. 15. last"` устанавливает таймер на срабатывание в первый, пятнадцатый и последний день каждого месяца, в то время как выражение `hour="8-18"` означает каждый час с 08:00 до 18:00.

Допустимо задавать промежутки времени, прибавляя их к начальному моменту времени. Выражение `hour="8/1"` срабатывает каждый час, начиная с 08:00, в то же время `hour="*/12"` срабатывает каждые 12 часов. Однако вы можете указывать промежутки времени только для секунд, минут и часов.

---

[1] Руководство Oracle Java EE 7: http://docs.oracle.com/javaee/7/tutorial/doc/ejb-basicexamples004.htm#GIQLY.

Таблица 10.4 предлагает несколько примеров календарного расписания в действии.

Таблица 10.4. Примеры выражений в действии

Выражение	Действие
Second = "10"	Каждые десять секунд
hour = "2"	Каждые два часа
Minute = "15"	Каждые 15 минут
dayOfWeek = "Mon, Fri"	Каждый понедельник и пятницу в полночь
dayOfWeek = "0–7", hour = "8"	Каждый день в 8:00
dayOfMonth = "–7"	За пять дней до конца каждого месяца в полночь
dayOfMonth = "1st Mon", hour = "22"	В первый понедельник каждого месяца в 22:00
Month = "Mar", dayOfMonth = "15"	15-го числа следующего марта
year = "2016", month = "May"	1 мая 2016 года в полночь

Новым в реализации EJB 3.2 является усовершенствование в API сервисов таймеров, предоставляющее доступ ко всем активным таймерам в EJB-модуле. Сюда включаются как программно, так и автоматически созданные таймеры (листинг 10.3).

Листинг 10.3. Поиск всех таймеров и управление ими

```
package com.devchronicles.timer;

import java.util.Collection;
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.Singleton;
import javax.ejb.Startup;

import javax.ejb.Timer;
import javax.ejb.TimerService;

@Singleton
@Startup
public class AllTimers {

    @Resource
    TimerService timerService;

    @PostConstruct
    public void manageTimer(){

        Collection<Timer> timers = timerService.getAllTimers();

        for(Timer t : timers){
```

```
            System.out.println("Timer Info: " + t.getInfo());
            System.out.println("Time Remaining: " + t.getTimeRemaining());
            t.cancel();
        }
    }
}
```

В листинге 10.3 компонент инстанцируется при запуске и вызывается метод manageTimer. Вы получаете коллекцию, содержащую все активные таймеры, и в цикле по коллекции выводите информацию о таймерах и количество миллисекунд, которое пройдет до срабатывания первого запланированного таймера. И наконец, вы отключаете таймер.

Транзакции

Компоненты создают таймеры внутри управляемой контейнером транзакции. Если выполняется откат этой транзакции, то же самое происходит и с таймером. Если эта транзакция откатывается, таймер затем тоже откатывается. Это означает, что его создание откатывается и, если он был отключен, отключение тоже аннулируется и таймер восстанавливается. В листинге 10.4 мы показываем пример метода таймера, отмеченного аннотацией транзакции.

Листинг 10.4. Таймер может устанавливать атрибут транзакции

```
package com.devchronicles.timer;

import javax.annotation.Resource;
import javax.ejb.Timeout;
import javax.ejb.TimerService;

public class SimpleProgramaticTimer {

    @Resource
    TimerService timerService;

    public void setTimer(){
        ScheduleExpression expression = new ScheduleExpression();
        expression.second("*/10").minute("*").hour("*");
        timer = timerService.createCalendarTimer(
            new ScheduleExpression().second("*/10").minute("*").hour("*"));
    }

    @Timeout
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void performTask() {
        System.out.println("Simple Task performed");
    }
}
```

Компоненты, использующие контейнерно-управляемые транзакции, устанавливают атрибут транзакции на метод, аннотированный @Timeout. Транзакции

могут быть назначены атрибуты `Required` или `RequiresNew`. Транзакция начинается до вызова метода. Если транзакция откатывается, то метод `@Timeout` вызывается снова.

Резюме

В этой главе вы увидели, как создавать автоматические и программные таймеры и как они ведут себя внутри транзакций. Таймеры могут быть весьма полезны, когда необходимо запустить cron-подобное задание без нарушения основной бизнес-логики. Вы можете видеть примеры таймеров во многих проектах и почти во всех языках программирования. Автоматические таймеры создаются путем аннотирования метода с помощью как аннотации `@Schedule`, так и `@Schedules`, а также жесткого задания значений таймера как атрибутов аннотации путем описания их в дескрипторе развертывания `ejb-jar.xml`. Программные таймеры создаются в коде приложения и могут менять значения во время выполнения.

Тип выбираемого вами для решения своих задач таймера зависит в основном от того, будет частота события меняться на основе бизнес-логики (клиентские сервисы) или технических требований (повторное заполнение кэша). Последний случай лучше обслуживать с помощью программного таймера, в то время как для первого полезнее автоматический таймер.

Таймеры по умолчанию сохраняются для защиты от отключений или сбоев сервера и могут быть сериализованы в базе данных и позднее получены оттуда. Таймеры принимают участие в транзакциях и полностью откатываются вместе с транзакцией. В EJB 3.2 стало несколько легче управлять таймерами; вы можете получить все активные таймеры в коллекции и вызывать методы таймеров для каждого экземпляра.

С новыми усовершенствованиями в платформе Java EE таймеры стали более полноценными и способными на многое, сделав большинство сторонних фреймворков устаревшими.

Упражнения

1. Напишите кэш, повторно заполняющий карту из базы данных. Установите сервис таймера на срабатывание в 3 часа ночи для вызова метода повторного заполнения кэша.
2. Разработайте программный таймер, посылающий уведомление клиенту, когда его подписка становится доступна для возобновления.

11 Паттерн «Наблюдатель»

В этой главе:

- как реализовать паттерн «Наблюдатель» в простом коде;
- как паттерн «Наблюдатель» работает на практике;
- как реализовать паттерн «Наблюдатель» с помощью аннотации `@Observes` и возбуждения событий;
- как использовать квалификаторы для достижения «дробного» контроля над использованием наблюдателей;
- как реализовать восприимчивые к транзакциям наблюдатели и откаты.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 11.zip, каждый из них назван в соответствии с наименованиями в тексте.

Паттерн «Наблюдатель» — один из наиболее широко используемых и общепринятых паттернов проектирования в современных языках программирования, программном обеспечении и UI-фреймворках. Большинство языков программирования использует наблюдателей во внутренних интерфейсах программирования приложений (API), и язык Java не исключение. Но платформа Java EE идет дальше других и предоставляет для паттерна «Наблюдатель» реализацию по умолчанию, так что разработчики могут использовать его, не реализуя с нуля. Данная глава рассматривает реализацию паттерна «Наблюдатель» по умолчанию для языка Java: где он используется, как наблюдатели реализованы с помощью аннотаций в Java EE и как их сделать восприимчивыми к транзакциям.

Что такое наблюдатель

Главная идея паттерна «Наблюдатель» заключается в том, что объект при изменении его состояния может сообщать другим объектам об этом изменении. На языке паттернов проектирования объект, изменяющий состояние, называется *субъектом*, в то время как получающие извещения об изменении называются *наблюдателями*. Отношение здесь «один ко многим», у субъекта может быть много наблюдателей.

Представьте себе приложение для интерактивной переписки, автоматически обновляющееся каждую секунду для проверки наличия новых сообщений. Представьте также, что у него есть функция комнаты для общения, позволяющая переписываться сразу многим людям. Каждый из клиентов регулярно проверяет на сервере, не появились ли новые сообщения, отправленные другими клиентами. Как вы легко можете догадаться, это не очень-то хорошо сказывается на производительности. Не будет ли намного разумнее «проталкивать» новое отправленное сообщение всем «подписанным» клиентам? Это определенно было бы эффективнее. Паттерн «Наблюдатель» может решить эту задачу. А именно, наблюдатель был бы сервером интерактивной переписки, а каждый клиент был бы субъектом. Каждый клиент регистрировался бы на сервере, и, когда пользователь отправлял бы новое сообщение (изменял состояние субъекта), субъект обращался бы к методу на сервере для извещения его о новом сообщении. Затем сервер вызывал бы методы на всех зарегистрированных клиентах и отправлял сообщение каждому из них.

ПРИМЕЧАНИЕ

Паттерн «Наблюдатель» также называют принципом Голливуда, чей девиз — «Не звоните нам, мы сами вам позвоним». Это неудивительно, большинство голливудских агентов предпочли бы звонить клиентам насчет новой роли, а не быть преследуемыми звонящими им клиентами. Эта система работает отлично, поскольку не бывает идеального времени для звонка агенту насчет имеющихся вакансий. Вероятнее всего, вы или упустите работу, если вакансии появляются чаще, чем вы звоните, или вас посчитают надоедливым, если вы будете звонить чаще, чем появляются вакансии.

С помощью паттерна «Наблюдатель» агент звонит подходящим клиентам, как только открывается новая вакансия, без потерь времени и бесполезной траты ресурсов.

Описание

Книга GoF утверждает, что паттерн «Наблюдатель» «определяет между объектами зависимость типа "один ко многим", так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются». «Паттерны проектирования» приводит пример приложения для наблюдения за погодой, отправляющего уведомление при изменении температуры. Паттерн «Наблюдатель» — один из паттернов поведения и основан на принципе наследования.

Чтобы быть наблюдателем, каждая конкретная реализация паттерна должна разделять схожий интерфейс. Субъект разрешает каждому наблюдателю добавлять себя в реестр. Когда субъект меняется, наблюдатель обращается к каждой зарегистрированной реализации для оповещения паттерна об изменениях.

Такая реализация эффективна, поскольку только одно обращение выполняется к каждому наблюдателю в момент изменения. «Наивное» решение вроде регулярных проверок субъекта может привести к бесконечному количеству обращений от разных наблюдателей даже при отсутствии изменений в наблюдаемом объекте.

Паттерн «Наблюдатель» не сильно отличается от подписки на новости. Объекты, желающие «подписаться» на изменения другого объекта, регистрируются для получения новостей об этих изменениях. Вместо проверки целевого объекта выполняется обращение к этим объектам при появлении изменений.

UI-фреймворки — еще одно место, где интенсивно используются наблюдатели, хотя это скорее относится к приложениям для настольных систем, а не к корпоративным приложениям. В контексте UI-фреймворков паттерн «Наблюдатель» часто называется паттерном «Прослушиватель». По сути, эти паттерны — одно и то же. Прослушиватели нажатий клавиш, дескрипторы «перетаскивания» и прослушиватели изменений значения — все они основаны на реализации паттерна «Наблюдатель».

Почти все веб-фреймворки построены на паттерне «Модель — представление — контроллер» (Model — View — Controller, MVC), который внутри также использует паттерн «Наблюдатель» (см. главу 14 для получения более подробной информации).

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Долгое время частью моей ежедневной работы было руководство практикантами и недавними выпускниками. Здесь я хочу рассказать об одной одаренной практикантке, с которой мне случилось работать. Подававшая надежды выпускница факультета электроники имела больше опыта в аппаратном обеспечении и структурном программировании, чем в объектно-ориентированных языках; вследствие этого у нее практически не было знаний о паттернах проектирования. Она только что завершила успешный проект, основанный на Arduino[1].

Мы начали разработку приложения для платформы Android, которое использовало встроенную в эту ОС возможность распознавания лиц для обнаружения, находится ли пользователь перед устройством. Практикантка сразу предложила воспользоваться основанным на Arduino проектом, чтобы создать цикл для опроса камеры на предмет обнаружения нового лица. Этот цикл запускался в основном потоке приложения, поэтому блокировал все приложение.

Осознав, что заблокировала поток пользовательского интерфейса, она решила создать отдельный поток для выполнения задачи распознавания лиц. Девушка использовала подход «если единственный имеющийся у вас инструмент — молоток, вы склонны считать любую проблему гвоздем». Тогда мы немного поговорили с ней о структуре приложения для Arduino. На Arduino все приложение представляло собой цикл, работу которого мы хотели поддерживать до тех пор, пока мы его не остановим, и управление всеми программными функциями происходило в этом цикле. Однако структура нашего приложения для Android была другой. Приложению скорее требовались извещения об обнаруженном лице, чем опрос камеры на предмет того, не было ли обнаружено лицо. Как только эта практикантка поняла, как работают наблюдатели, она без труда реализовала паттерн, поскольку система Android была уже построена на паттерне «Наблюдатель». Ей пришлось лишь добавить соответствующий класс прослушивателя и реализовать те функции, которые необходимо было выполнить при обнаружении лица.

Диаграмма классов наблюдателя

Как можно видеть на рис. 11.1, паттерн «Наблюдатель» предоставляет интерфейс `Observer`, который должны реализовывать все конкретные наблюдатели. В этом интерфейсе есть ровно один метод, вызываемый субъектом для оповещения наблюдателей о произошедшем изменении состояния. У каждого субъекта есть список зарегистрированных наблюдателей, и он обращается к методу `notifyObservers` для извещения зарегистрированных наблюдателей о любых обновлениях или изменениях субъекта. У него также есть методы для регистрации и аннулирования регистрации наблюдателей.

[1] Маленькая аппаратная плата для проектов сборки: http://www.arduino.cc.

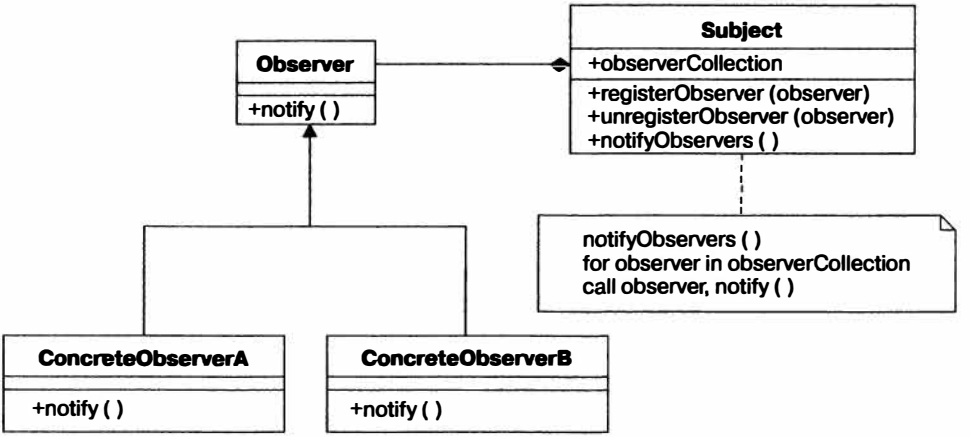


Рис. 11.1. Диаграмма классов паттерна «Наблюдатель»

Реализация паттерна «Наблюдатель» в простом коде

Язык Java обеспечивает готовую для использования реализацию паттерна «Наблюдатель». Разработчики легко могут реализовать этот паттерн с помощью интерфейса `Observer` и расширения класса `Observable`.

Первое, что вам необходимо сделать, — это создать класс, расширяющий класс `Observable`. В листинге 11.1 новостное агентство оповещает несколько типов подписчиков в момент публикации нового материала. Подписчик может добавить собственное поведение после получения обновления. Листинг 11.2 обеспечивает интерфейс для «публикации» класса `Observable`.

Листинг 11.1. Новостное агентство (NewsAgency), реализующее интерфейс Observable

```
package com.devchronicles.observer;

import java.util.ArrayList;
import java.util.List;
import java.util.Observable;
import java.util.Observer;

public class NewsAgency extends Observable implements Publisher {

    private List<Observer> channels = new ArrayList<>();

    public void addNews(String newsItem) {
        notifyObservers(newsItem);
    }

    public void notifyObservers(String newsItem) {
        for (Observer outlet : this.channels) {
            outlet.update(this, newsItem);
```

```
        }
    }

    public void register(Observer outlet) {
        channels.add(outlet);
    }
}
```

Листинг 11.2. Интерфейс Publisher

```
package com.devchronicles.observer;

public interface Publisher {}
```

Далее вам необходимо создать класс для наблюдения за изменениями NewsAgency. Наблюдатель должен реализовывать интерфейс Observer, как в листинге 11.3.

Листинг 11.3. Конкретный наблюдатель

```
package com.devchronicles.observer;

import java.util.Observable;
import java.util.Observer;

public class RadioChannel implements Observer {

    @Override
    public void update(Observable agency, Object newsItem) {

        if (agency instanceof Publisher) {
            System.out.println((String)newsItem);
        }
    }
}
```

Наконец, вы должны зарегистрировать наблюдатель RadioChannel на наблюдаемом NewsAgency и создать несколько новостных материалов.

```
// Создание наблюдателя и субъекта
NewsAgency newsAgency = new NewsAgency();
RadioChannel radioChannel = new RadioChannel();

// Регистрация наблюдателя на субъекте
newsAgency.register(radioChannel);

// Добавление новостных заголовков
newsAgency.addNews("Срочные новости: на Марсе найдена жизнь");
newsAgency.addNews("Последние известия: вторжение на Землю неизбежно");
newsAgency.addNews("Срочно в номер:
                    приветствуем наших новых марсианских повелителей");
```

Вывод в консоли будет примерно следующим:

```
Срочные новости: на Марсе найдена жизнь
Последние известия: вторжение на Землю неизбежно
Срочно в номер: приветствуем наших новых марсианских повелителей
```

Заметьте, что вы можете регистрировать много наблюдателей на newsAgency и получать с их помощью обновления. Например, можно зарегистрировать наблюдатель TVChannel или InternetNewsChannel для получения обновлений от newsAgency. Помимо этого, у вас могут быть другие Publisher (или любые иные типы объектов, реализующие Observable), выдающие обновления любому наблюдателю, пожелавшему зарегистрировать себя для получения новостей. Эти наблюдатели могут осуществлять проверку типа Observable и обрабатывать обновления в соответствии с источником.

Один существенный недостаток подобной реализации паттерна «Наблюдатель» в том, что вам приходится расширять класс Observable. Это принуждает использовать иерархию классов, которая может быть нежелательной. Поскольку вы не можете расширить сразу несколько классов в мире единичного наследования[1] языка Java, такой способ реализации паттерна «Наблюдатель» ограничивает проектирование наследования. Вы не можете добавить поведение класса Observable к существующему классу, который уже расширяет другой базовый класс, ограничивая тем самым потенциал его многократного использования.

Но не отчаивайтесь. Вы можете реализовать паттерн «Наблюдатель» «вручную», без использования внутренних интерфейсов Observer и Observable, следуя приведенной выше диаграмме классов. Однако, поскольку эта книга посвящена платформе Java EE, такая реализация оставляется вам для самостоятельной проработки.

Реализация паттерна «Наблюдатель» в Java EE

Хотя в языке Java с самого начала есть встроенная поддержка паттерна «Наблюдатель», платформа Java EE предоставляет более удобную реализацию с помощью аннотации @Observes и интерфейса javax.enterprise.event.Event<T>. Любой метод, аннотированный @Observes, прослушивает события определенного типа. Когда он слышит подобное событие, параметр метода наблюдателя получает в качестве значения экземпляр соответствующего типа, после чего выполняется метод.

Аннотация @Observes позволяет каждому методу прослушивать любые события, возбужденные объектом отмеченного типа. В листинге 11.4 приводится простой пример компонента, возбуждающего событие типа String, и другого компонента, слушающего события этого типа от нашего компонента.

Листинг 11.4. Компонент наблюдаемого сервиса

```
package com.devchronicles.observer;

import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
```

[1] Речь идет о наследовании классов. Множественное наследование интерфейсов в языке Java разрешено. — *Примеч. пер.*

```
import javax.ejb.TransactionAttributeType;
import javax.enterprise.event.Event;
import javax.inject.Inject;

@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class EventService {

    @Inject
    private String message;

    @Inject
    Event<String> event;

    public void startService(){
        event.fire("Starting service " + message);
    }
}
```

Контейнер внедряет объект Event типа String в переменную event экземпляра класса EventService. Это формирует часть сообщения, когда срабатывает строковый объект «возбудить событие». Этот объект, переменная экземпляра Message, — строка, которая могла быть произведена фабрикой (см. главу 6 для получения более полной информации о паттерне проектирования «Фабрика», внедренном в класс EventService). Чтобы заставить этот пример работать без создания фабрики, вы можете просто присвоить любую строковую константу переменной с именем message и убрать аннотацию @Inject следующим образом:

```
private String message = "produced message";
```

Теперь наблюдаемая часть завершена и наступило время создать наблюдатель, который будет слушать события вашего String. Добавление аннотации @Observes к сигнатуре метода указывает, что метод — наблюдатель событий типа, которому она предшествует. В данном случае аннотация @Observes предшествует типу String и, соответственно, будет слушать события этого типа. Аннотация @Observes, за которой следует тип объекта, творит чудеса и позволяет аннотированному методу наблюдать произошедшее событие заданного типа.

В листинге 11.5 к сигнатуре метода serviceTrace была добавлена аннотация @Observes, отмечающая этот метод как наблюдатель событий типа String. Когда происходит событие типа String, метод serviceTrace получает объект, произведенный событием через его параметр. serviceTrace может поступить с объектом String так, как ему захочется. В данном случае он выводит сообщение в консоль.

Листинг 11.5. Компонент-наблюдатель

```
package com.devchronicles.observer;

import javax.ejb.Stateless;
import javax.enterprise.event.Observes;

@Stateless
```

```
public class TraceObserver {

    public void serviceTrace(@Observes String message){
        System.out.println("Service message: " + message);
    }
}
```

Если вы запустите сервер и вызовете метод startService, то осознаете, каким удивительным образом строка внедряется в класс EventService, затем событие String происходит «там, где его будет наблюдать» метод serviceTrace класса TraceObserver и сообщение выводится в консоль. Поразительно, что это все, что от вас требуется в Java EE для реализации паттерна «Наблюдатель», без дальнейших конфигурационных настроек.

Хотя в сценариях реального мира вы, вероятно, будете использовать в качестве событий не простые строки, а скорее ваши собственные объекты, которые будут наблюдаться в соответствии с их типом, по-прежнему нет ничего сложного в том, чтобы создать различия между одинаковыми типами объектов и установить различные наблюдатели для их прослушивания.

Сейчас вы увидите пример, в котором вы будете использовать квалификаторы для разрешения неоднозначности между объектами String. Вы уже видели, как эффективно это может работать при реализации паттерна «Фабрика», производящего различные реализации одного типа объектов.

В листинге 11.6 вы начнете с кода, разрешающего неоднозначность между вашими строками.

Листинг 11.6. Интерфейс аннотации @Qualifier

```
package com.devchronicles.observer;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD,ElementType.PARAMETER})
public @interface MessageEvent {

    Type value();
    enum Type{ SERVICE, PARAMETER }
}
```

Предшествующий классу интерфейс описывает квалификатор MessageEvent и два элемента перечислимого типа (SERVICE и PARAMETER), которые вы будете использовать в аннотации, чтобы отметить строки, подлежащие возбуждению экземплярами события.

```
import com.devchronicles.observer.MessageEvent.Type;

@Stateless
```

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class EventService {

    @Inject
    private String message;

    @Inject @MessageEvent(Type.SERVICE)
    Event<String> serviceEvent;

    @Inject @MessageEvent(Type.PARAMETER)
    Event<String> parameterEvent;

    public void startService() {
        serviceEvent.fire("Starting service " + message);
        parameterEvent.fire("-d -p");
    }
}
```

При использовании квалификаторов вы просто добавляете аннотацию MessageEvent к соответствующему внедренному экземпляру с нужным элементом перечислимого типа в круглых скобках. Тогда вы сможете позже возбудить события из метода startService, как вы уже делали ранее в предыдущем примере. Выделенные жирным шрифтом части строк кода — это все, что было добавлено к предыдущему примеру из листинга 11.6.

Теперь вы добавите аннотации к части, относящейся к наблюдателю. Вам всего лишь нужно, как и ранее, добавить квалификаторы к соответствующей аннотации @Observes.

```
import com.devchronicles.observer.javaee.MessageEvent.Type;

@Stateless
public class TraceObserver {

    public void serviceTrace(
        @Observes @MessageEvent(Type.SERVICE) String message) {
            System.out.println("Service message: " + message);
    }

    public void parameterTrace(
        @Observes @MessageEvent(Type.PARAMETER) String message) {
            System.out.println("with parameters: " + message);
    }
```

Возбуждение ваших собственных объектных типов и наблюдение за ними проходит еще проще. Объектный тип уникален, и не нужно создавать ваши собственные квалификаторы аннотаций — вместо этого можно использовать объект.

Наблюдаемые события транзакционны и происходят в определенной вами фазе транзакции для этого события. Это может происходить до или после завершения транзакции или после удачной или неудачной транзакции.

А сейчас вы увидите все это в действии. В листинге 11.7 вы задаете три метода-наблюдателя, определяющие фазы транзакций, во время которой наблюдатели слушают события типа String.

Листинг 11.7. Наблюдатель событий транзакций

```
package com.devchronicles.observer;

import javax.enterprise.event.Observes;
import javax.enterprise.event.TransactionPhase;

public class TransactionEventObserver {

    public void onInProgress(@Observes String message) {
        System.out.println("In progress message: " + message);
    }

    public void onSuccess(
        @Observes(during = TransactionPhase.AFTER_SUCCESS) String message) {
        System.out.println("After success message: " + message);
    }

    public void onFailure(
        @Observes(during = TransactionPhase.AFTER_FAILURE) String message) {
        System.out.println("After failure message: " + message);
    }

    public void onCompletion(
        @Observes(during = TransactionPhase.AFTER_COMPLETION) String message) {
        System.out.println("After completion message: " + message);
    }
}
```

Существует пять фаз транзакций: BEFORE_COMPLETION, AFTER_COMPLETION, AFTER_SUCCESS, AFTER_FAILURE и IN_PROGRESS (по умолчанию). В листинге 11.7 мы не реализовывали BEFORE_COMPLETION. В листинге 11.8 мы реализуем класс, демонстрирующий события, которые происходят в случае успеха и в случае неудачи транзакции.

Листинг 11.8. Вызываем сценарии успеха и неудачи

```
package com.devchronicles.observer;

import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.enterprise.event.Event;
import javax.inject.Inject;

@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
```

```
public class Children {

    @Resource
    SessionContext sc;

    @Inject
    Event<String> message;

    int[] children = new int[3];

    public void getSixthChild() {
        try {
            int sixthChild = children[5];
            // Генерирует IndexOutOfBoundsException
        } catch (Exception e) {
            message.fire("Rollback event occurred.");
            System.out.println("Exception caught.");
            sc.setRollbackOnly();
        }
    }

    public void getThirdChild() {
        int thirdChild = children[2]; // Успех
        message.fire("Successful event");
    }

}
```

Класс `Children` моделирует успешную транзакцию в методе `getThirdChild` и неудачную — в методе `getSixthChild` путем искусственного создания ситуации, при которой генерируется исключение `IndexOutOfBounds`.

Изучите каждый метод, чтобы увидеть, как происходит наблюдение за событиями. Метод `getThirdChild` возбуждает событие `String`, передает ему сообщение `Successful event` и затем успешно завершается. Вывод вследствие вызова этого метода будет таким:

```
In progress: Successful event
After completion message: Successful event
After success message: Successful event
```

Метод `onInProgress` вызывается сразу же, как только происходит событие, в то время как транзакция все еще работает. Остальные два метода — `onCompletion` и `onSuccess` — должны ждать, пока транзакция достигнет фаз `AFTER_COMPLETION` и `AFTER_SUCCESS` соответственно, прежде чем они смогут выполниться.

Далее посмотрите на метод `getSixthChild`, который завершается неудачно, вызывая исключение `IndexOutOfBounds`. Вывод вследствие вызова этого метода будет таким:

```
In progress: Rollback event occurred.
Exception caught.
After completion message: Rollback event occurred.
After failure message: Rollback event occurred.
```

Как и ранее, метод onInProgress вызывается сразу, а методы onCompletion и onSuccess должны ждать, пока он завершится. Как только метод onInProgress выводит сообщение Exception caught и транзакция помечается для отката путем вызова метода setRollbackOnly класса SessionContext, метод onInProgress завершается, и вы можете выполнять своих наблюдателей. Выполняется метод onCompletion, за которым следует метод OnFailure.

Метод setRollbackOnly помечает текущую транзакцию для отката, поэтому она никогда не может быть зафиксирована. Это действие переводит транзакцию в фазу AFTER_FAILURE и вызывает метод OnFailure.

У наблюдателей также может быть условное поведение, хотя оно ограничено уведомлением о том, существует ли уже экземпляр наблюдателя компонента, определяющего методы в данном контексте. Методы наблюдателя вызываются только в том случае, когда экземпляр существует. Для определения метода как условного добавьте notifyObserver = Reception.IF_EXISTS в качестве аргумента аннотации @Observes.

```
import javax.enterprise.event.Reception;

public void addMember (
    @Observes(notifyObserver = Reception.IF_EXISTS) String message){
    // Код реализации
}
```

Поведением по умолчанию является создание экземпляра, если он не существует.

Где и когда использовать паттерн «Наблюдатель»

Паттерн «Наблюдатель», который может принести огромный прирост производительности, — эффективный способ добиться слабой связи и изменить направление обращения/прослушивания.

При проектировании ваших приложений или рефакторинге чужого кода старайтесь избегать ненужных выполнений методов, которые можно заменить реализацией паттерна «Наблюдатель».

В царстве Java EE вы можете перевести существующий код на использование паттернов «Наблюдатель» без особых трудностей. Наблюдатели Java EE обычно сопровождаются внедрением зависимостей, использующим @Inject, и фабриками, использующими @Produces.

Наиболее сильная сторона паттерна «Наблюдатель» — расцепление классов — является и его уязвимостью. Как только управление переходит к наблюдателю, вы теряете нить последовательности выполняемых приложением действий. «Видимость» ухудшается по мере того, как одно событие запускает другое. Запутанная реализация паттерна «Наблюдатель» может стать кошмаром для отладки, так что старайтесь делать реализацию максимально более простой. Избегайте множе-

ственных слоев наблюдателей, идеально будет использовать один слой или совсем небольшое их количество.

Чтобы помочь будущим и текущим разработчикам понять назначение вашего кода, задавайте такое имя наблюдателя, которое будет отражать его назначение. Помимо этого, вам лучше отразить причину наблюдения в именах его методов, выражая явным образом назначение класса.

В царстве Java EE существующий код может быть переведен на использование паттернов «Наблюдатель» без особых трудностей. Наблюдатели Java EE обычно сопровождаются внедрением зависимостей (`@Inject`) и фабриками (`@Produces`). Излишнее и ненужное использование наблюдателей может привести к трудным для понимания и отладки системам. Однако, поскольку большинство разработчиков привыкли к наблюдателям благодаря UI и веб-фреймворкам, они почти всегда инстинктивно используют их в правильном контексте.

Когда бы вы ни увидели подверженный изменениям ресурс и вызывающие программы, которые пытаются собирать информацию о субъекте, не колеблясь используйте паттерн «Наблюдатель». Восприимчивые к транзакциям наблюдатели предоставляют функциональность, которую нелегко было реализовать в предыдущих версиях. В фазе `BEFORE_COMPLETION` вы можете отменить текущую транзакцию, вызвав метод `setRollbackOnly`, позволяя, таким образом, нетранзакционным операциям выполняться внутри фазы транзакции. При генерации исключения есть возможность выполнить откат всей транзакции.

Во время фазы `IN_PROGRESS`, охватывающей время всей транзакции, наблюдаемые события могут возбуждаться и немедленно подвергаться наблюдению. Это можно реализовать в виде типа монитора выполнения или регистратора.

Обращение к методу наблюдателя блокирует источник события и является синхронным, но может быть сделано асинхронным, если аннотировать метод наблюдателя с помощью `@Asynchronous` (см. главу 9 для получения более подробной информации об использовании этой аннотации). Делать наблюдателей фазы `BEFORE_COMPLETION` асинхронными лучше с осторожностью, поскольку метод `setRollbackOnly` не будет действовать и транзакцию нельзя будет откатить. Асинхронные методы попадаются в новых транзакциях.

Резюме

В этой главе вы увидели, как базовая реализация паттерна «Наблюдатель» языка Java была усовершенствована в платформе Java EE 7 и как он может быть сделан восприимчивым к фазе транзакции наблюдаемых им событий. Его реализация полностью расцепляет бизнес-логику с наблюдателем, оставляя для их соединения только тип события и квалификатор. Можно побеспокоиться об отсутствии обзора связей, но оно может быть смягчено соответствующим именованием класса и методов и иллюстрированием связей в документации класса.

Восприимчивость к фазам транзакции придала паттерну «Наблюдатель» новую функциональность. Она обеспечивает интеграцию между методами наблюдателя и транзакцией, позволяя выполнять откаты.

Упражнения

1. Перечислите все известные вам реализации паттерна «Наблюдатель», которые можно встретить в языке Java.
2. Создайте пример, использующий `notifyObserver = Reception.IF_EXISTS` в качестве аргумента аннотации `@Observes`.
3. Используйте восприимчивость наблюдателей к фазам транзакции для отслеживания течения транзакции и зарегистрируйте результат транзакции (успех или неудачу).

12 Паттерн «Доступ к данным»

В этой главе:

- история возникновения паттерна «Доступ к данным»;
- исследование связанного паттерна «Объект передачи данных»;
- как DAO и паттерн «Фабрика» работают вместе;
- введение в JPA и ORM;
- простая реализация DAO;
- усовершенствованная реализация с применением обобщений;
- обсуждение роли DAO в современной платформе Java EE.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 12.zip, каждый из них назван в соответствии с наименованиями в тексте.

Невозможно представить себе корпоративное приложение, которое не взаимодействовало бы тем или иным образом с источником данных. Источник данных может быть реляционной, объектно-ориентированной или NoSQL-базой данных, хранилищем Lightweight Directory Access Protocol (LDAP), файловой системой, веб-сервисом или внешней системой. Из какого бы источника ни приходили данные, корпоративное приложение должно с ним взаимодействовать и выполнять простейшие операции создания, чтения, обновления и удаления (CRUD). Почти все серверы используют подобные источники данных для непрерывного сохранения сеансов или долго выполняющихся процессов.

Способы использования вами источников данных могут сильно различаться, и их реализация тоже может варьироваться в широком диапазоне. Существуют различные «диалекты» SQL, такие как Postgre SQL и Oracle SQL. Основная задача паттерна «Объект доступа к данным» (Data Access Object, DAO) — инкапсуляция доступа к источнику данных путем обеспечения интерфейса, через который различные слои могут обмениваться информацией с источником данных.

Эта глава обсуждает решенную DAO исходную проблему и его уместность в современных приложениях для платформы Java EE. Исследуется также роль связанного объекта передачи данных (Data Transfer Object, DTO) и то, как он и паттерн «Фабрика» сочетаются с DAO. Кроме того, глава охватывает использование

JPA и ORM в контексте DAO. Вы увидите реализацию DAO и пути усовершенствования его с помощью обобщений. И наконец, вы прочитаете об изменившейся роли паттерна и о том, почему он все еще считается эффективным паттерном проектирования.

Что такое паттерн «Доступ к данным»

Первоначальное решение, предлагавшееся паттерном DAO, было определено в книге «Образцы J2EE. Лучшие решения и стратегии проектирования»[1] *(Core J2EE Patterns: Best Practices and Design Strategies)* следующим образом.

«Используйте Data Access Object (DAO) для абстрагирования и инкапсулирования доступа к источнику данных. DAO управляет соединением с источником данных для получения и записи данных».

Задача, которая была решена путем абстрагирования и инкапсуляции источника данных, состояла в защите приложения от зависимости от реализации источника данных. Это расцепляло бизнес-слой с источником данных. Существовало мнение, что, если источник данных изменится, расцепление снизит или сведет на нет любое влияние на бизнес-слой. Однако в действительности источник данных редко меняется — даже от одного поставщика однотипных источников к другому, например от Postgre к MS SQL. Сложно представить себе, чтобы было принято решение поменять SQL-источник данных на систему неструктурированных XML-файлов, хранилище LDAP или веб-сервис. Такого просто не бывает. Так какое же значение имеет паттерн DAO в современной Java EE? Действительно ли он вам нужен?

Паттерн DAO по-прежнему остается полезным шаблоном, и его первоначальное решение все еще актуально, хотя мотивы для его реализации немного изменились. Ценность его скорее не в защите от влияния маловероятного изменения типа источника данных, а в тестируемости и возможности проверки с помощью объектов-имитаций, а также в его пользе при структурировании кода и освобождении от кода доступа к данным. Кроме того, все еще остается смысл в его использовании в качестве метода инкапсуляции устаревших систем хранения данных и упрощения доступа к сложным реализациям источников данных. Однако это, вероятнее всего, «тупиковые» и редкие случаи.

Паттерн DAO инкапсулирует операции CRUD в интерфейсе, реализованном конкретным классом. Этот интерфейс может быть сымитирован, а значит, легко протестирован без необходимости связи с базой данных. Тестирование стало проще, поскольку писать тесты с помощью объектов-имитаций легче, чем интегрировать тесты с действующей базой данных. Конкретные реализации DAO используют для выполнения операций CRUD низкоуровневые API, такие как JPA и Hibernate.

---

[1] *Алур Д., Крупи Дж., Малкс Д.* Образцы J2EE. Лучшие решения и стратегии проектирования. — М.: Лори, 2004. — 400 с.

Диаграмма классов доступа к данным. На рис. 12.1 показана диаграмма классов DAO, демонстрирующая взаимодействие между клиентом, DAO и DTO. На рисунке не показана дополнительная фабрика, производящая экземпляр DAO.

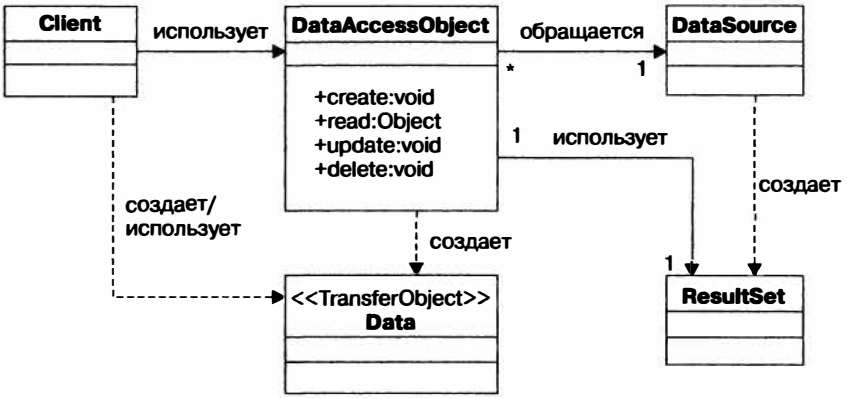


Рис. 12.1. Диаграмма классов паттерна «Доступ к данным»

Обзор паттерна «Доступ к данным»

Реализация паттерна DAO включает четыре составляющих:

- интерфейс DAO;
- конкретную реализацию интерфейса DAO;
- фабрику DAO;
- DTO.

Фабрика, интерфейс и DTO — дополнительные необязательные компоненты. Вы увидите, как эти два паттерна используются вместе с паттерном DAO. Паттерн «Фабрика» обсуждается более подробно в главе 6.

Паттерн «Объект передачи данных»

DTO переносит между логическими слоями данные, извлеченные из базы данных или сохраняемые в ней. Например, при передаче списка объектов `User` от слоя доступа к данным в веб-слой слой сервисов будет выполнять передачу от DAO к DTO.

ПРИМЕЧАНИЕ

DTO называют также объект-значение (Value Object)[1].

[1] Частая путаница между названиями этих различных по сути паттернов вызвана тем, что паттерн, носивший в первом издании книги Core J2EE Patterns: Best Practices and Design Strategies название Value Object, во втором был переименован в DTO. Мартин Фаулер описывает объект-значение как «маленький объект для хранения, например, денежных величин или диапазонов дат». См.: http://martinfowler.com/bliki/ValueObject.html. — *Примеч. пер.*

Решение, предлагаемое паттерном DTO, описывается в Core J2EE Patterns: Best Practices and Design Strategies следующим образом.

«Используйте объект передачи данных для переноса различных элементов данных между уровнями».

DTO снижает количество удаленных запросов по сети в приложениях, выполняющих многочисленные обращения к корпоративным компонентам, приводя тем самым к улучшению производительности. Иногда не все данные, извлеченные из базы, требуются в веб-слое или каком-то другом слое, нуждающемся в использовании данных. Итак, DTO сокращает объем данных только до тех, которые требуются слою, таким образом оптимизируя передачу данных между уровнями. Данная глава не углубляется в подробности DTO. Рекомендуем вам прочитать соответствующий раздел в книге Core J2EE Patterns: Best Practices and Design Strategies.

Java Persistence Architecture API и объектно-реляционное отображение

Интерфейс программирования приложений Java Persistence (Java Persistence API, JPA) управляет взаимодействием приложения с источником данных. Он определяет, как обращаться к данным, сохранять их и управлять их перемещением между объектами приложения и источниками данных. Сам по себе JPA не выполняет CRUD или другие относящиеся к данным операции, это просто набор интерфейсов и технических требований к реализации. Тем не менее совместимые с платформой Java EE приложения должны обеспечивать поддержку его использования.

Спецификация JPA заменила имевшуюся в EJB 2.0 спецификацию контейнерно-управляемой сохраняемости (Container-Managed Persistence, CMP) компонентов-сущностей — тяжеловесную и сложную. Отрицательная реакция на CMP многих членов сообщества разработчиков привела к широкому распространению проприетарных решений, таких как Hibernate и TopLink. Это, в свою очередь, стало поводом к разработке JPA (выпущенного вместе с EJB 3.0), предназначенного для объединения CMP, Hibernate и TopLink и, похоже, весьма преуспевшего в этом.

В основе JPA лежит концепция сущности. Для знакомых с CMP укажем: это то, что называлось там *компонент-сущность*. *Сущность* — это недолго живущий объект, который может быть сохранен в базе данных не как сериализуемый объект, а как данные. Это простой Java-объект в старом стиле (POJO), члены которого аннотированы и отображены на поле в источнике данных. Для того чтобы лучше понять, как это выглядит в коде, просмотрите приведенный ниже фрагмент. Здесь класс сущности `Movie` представлен в виде соответствующим образом аннотированного POJO:

```
@Entity
public class Movie {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
```

```
    private String description;
    private Float price;

    public Movie() {
    }
    // Для краткости методы чтения и устанавливающие методы опущены.
}
```

Как вы можете видеть, это простой класс всего с тремя аннотациями. Аннотация уровня класса `@Entity` указывает, что с этим классом следует обращаться как с классом сущности, и аннотации `@Id` и `@GeneratedValue` отмечают член класса `id` как автоматически сгенерированное идентификационное поле. Это значит, что при сохранении сущности поле `id` автоматически генерируется в соответствии с устанавливаемыми источником данных правилами автоматической генерации полей. Если источник данных — база данных, то все поля в этой сущности сохраняются в таблице `Movie`. Не требуется больше никаких аннотаций для указания сохраняемых полей. Благодаря соглашениям по конфигурации все поля сохраняются, если не аннотировано обратное. Такое отображение называется *объектно-реляционным* (Object Relational Mapping, ORM). Подробное обсуждение JPA и ORM выходит за рамки данной главы, так что рекомендуем вам прочитать *The Java EE 7 Tutorial: Part VIII Persistence*[1].

Реализация паттерна «Доступ к данным» в Java EE

Сейчас вам предстоит разобрать пример, чтобы увидеть, как реализовывать DAO в Java EE. В качестве предметной области возьмем прокат фильмов, а в качестве источника данных — реляционную БД. Вы начнете с создания класса сущности фильма и аннотируете его соответствующими аннотациями JPA, как показано в листинге 12.1.

Листинг 12.1. Класс сущности фильма

```
package com.devchronicles.dataaccessobject;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Transient;

@Entity
public class Movie2 implements Serializable {
    private static final long serialVersionUID = -6580012241620579129L;
```

[1] http://docs.oracle.com/javaee/7/tutorial/doc/.

```
@Id @GeneratedValue
private int id;
private String title;
private String description;
private int price;
// Переменные времени выполнения,
// которые нет необходимости сохранять
@Transient
private int runtimeId;

public Movie2() {
}

public int getId() {
    return this.id;
}

public void setId(int id) {
    this.id = id;
}

public String getTitle() {
    return this.title;
}

public void setTitle(String title) {
    this.title = title;
}

public String getDescription() {
    return this.description;
}

public void setDescription(String description) {
    this.description = description;
}

public int getPrice() {
    return this.price;
}

public void setPrice(int price) {
    this.price = price;
}

public int getRuntimeId() {
    return this.runtimeId;
}

public void setRuntimeId(int runtimeId) {
```

```
        this.runtimeId = runtimeId;
    }
}
```

Класс в листинге 12.1 — простой POJO с соответствующими JPA-аннотациями. Как было кратко упомянуто ранее, аннотация уровня класса `@Entity` указывает, что с этим классом следует обращаться как с классом сущности, и он должен управляться поставщиком сохраняемости. У класса сущности должен быть общедоступный или защищенный конструктор без аргументов, хотя у него могут быть и другие конструкторы. Это должен быть класс верхнего уровня, значит, он не может быть перечислением или интерфейсом и он не должен быть `final`. Кроме того, не могут быть объявленными как `final` ни одна из сохраняемых переменных экземпляра и ни один из методов чтения/устанавливающих методов. Класс сущности должен реализовывать интерфейс `Serializable`.

Вы аннотировали член `id` аннотациями `@Id` и `@GeneratedValue`, которые пометили его как автоматически сгенерированный первичный ключ. Все сущности обязаны иметь первичный ключ, который может быть одним членом класса или их комбинацией.

Для первичного ключа можно использовать один из следующих типов:

- простые типы данных языка Java (`byte`, `char`, `short`, `int`, `long`);
- классы-обертки простых типов данных Java (`Byte`, `Character`, `Short`, `Integer`, `Long`);
- массивы простых типов или типов-оберток (`long[]`, `Long[]`);
- типы данных языка Java (`String`, `BigInteger`, `Date`).

Все члены класса сущности автоматически отображаются на поля с таким же именем в таблице `movie`, если только они не аннотированы `@Transient`. Это означает, что член [класса] `id` отображается на поле `id` в таблице `movie`, член `title` отображается на поле `title` в таблице `movie` и т. д.

Затем в листинге 12.2 вы создаете интерфейс DAO. Он должен определять основные методы CRUD и любые другие методы, которые могут оказаться полезными.

Листинг 12.2. Интерфейс DAO

```
package com.devchronicles.dataaccessobject;

import java.util.List;

public interface MovieDAO {
    public void addMovie(Movie movie);
    public Movie getMovie(int id);
    public void deleteMovie(int id);
    public void updateMovie(Movie movie);
    public List<Movie> getAllMovies();
}
```

Теперь перейдем к конкретной реализации интерфейса DAO, показанной в листинге 12.3. Здесь вы реализуете операции CRUD. Обратите внимание, что конструктору передается экземпляр EntityManager. Он связан с контекстом сохраняемости, описанным в persistence.xml. API EntityManager обеспечивает функциональность создания, удаления и сохранения, равно как и возможность создания запросов. Никакие кратковременные поля не будут сохраняться или извлекаться из базы данных, поэтому рассчитывайте на то, что данные в кратковременных полях будут сбрасываться при каждом пересоздании объекта.

Листинг 12.3. Реализация интерфейса DAO

```
package com.devchronicles.dataaccessobject;

import java.util.List;
import javax.persistence.EntityManager;

public class MovieDAOImpl implements MovieDAO {

    private EntityManager em;

    public MovieDAOImpl(EntityManager em) {
        this.em = em;
    }

    @Override
    public void addMovie(Movie movie) {
        em.persist(movie);
    }

    @Override
    public Movie getMovie(int id) {
        return getAllMovies().get(id);
    }

    @Override
    public void deleteMovie(int id) {
        em.remove(getMovie(id));
    }

    @Override
    public void updateMovie(Movie movie) {
        em.merge(movie);
    }

    @Override
    public List<Movie> getAllMovies() {
        return em.createQuery("SELECT m FROM Movie m". Movie.class)
        .getResultList();
    }
}
```

В листинге 12.4 вы создаете фабрику DAO. EntityManager создается и внедряется в этот класс, после чего передается в качестве аргумента конструктора методу createMovieDAO, создающему объект DAO. Паттерн «Фабрика» подробнее рассматривается в главе 6, поэтому обратитесь, пожалуйста, к ней за расширенной информацией.

Листинг 12.4. Фабрика DAO

```
package com.devchronicles.dataaccessobject;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@ApplicationScoped
public class MovieDAOFactory {

    @PersistenceContext(unitName = "moviePU")
    private EntityManager em;

    @Produces
    public MovieDAO createMovieDAO() {
        return new MovieDAOImpl(em);
    }
}
```

Список сущностей вашего приложения называется *модулем сохраняемости*. Модуль сохраняемости приложения описывается в конфигурационном файле persistence.xml. Он должен располагаться в каталоге **META-INF** вашего приложения. Перечислим основные значимые элементы persistence.xml.

- **Имя модуля сохраняемости.** Вы можете присвоить модулю сохраняемости имя, так что можно определить несколько модулей сохраняемости и затем выбирать их во время выполнения.
- **Тип транзакции модуля сохраняемости.** В приложениях Java SE тип транзакции по умолчанию — RESOURCE_LOCAL, в то время как в Java EE тип транзакции — JTA. Это значит, что менеджер сущностей участвует в транзакции.
- **Поставщик.** Этот элемент идентифицирует класс, предоставляющий фабрику для создания экземпляра EntityManager.
- **Класс.** Классы сущности, используемые в приложении, должны быть перечислены в элементах <class>.
- **Свойство.** Можно указать дополнительные свойства, такие как свойства соединения с базой данных и свойства поставщика сохраняемости, например возможность создавать новые таблицы через удаление.

EntityManager связан с контекстом сохраняемости, описанным в persistence.xml в листинге 12.5.

Листинг 12.5. persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
     xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="moviePU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <jta-data-source>jdbc/sample</jta-data-source>
        <class>com.devchronicles.dataaccessobject.Movie</class>
    </persistence-unit>
</persistence>
```

Конкретный источник данных определяется в файле persistence.xml. В этом случае вы определили базу данных Derby с помощью поставщика EclipseLink. Вы определили тип транзакции как JTA, поскольку это реализация приложения для платформы Java EE, и указали в качестве класса сущности com.devchronicles.data-accessobject.Movie.

Наконец, вам необходимо внедрить созданный вами DAO и использовать его. Клиент в листинге 12.6 получает экземпляр внедренного DAO и использует его для извлечения всех фильмов.

Листинг 12.6. Клиент

```
package com.devchronicles.dataaccessobject;

import javax.ejb.Stateless;
import javax.inject.Inject;
import java.util.List;

@Stateless
public class Client {

    @Inject
    MovieDAO movieDAO;

    public List<Movie> getAllMovies() {
        return movieDAO.getAllMovies();
    }

}
```

Это упрощенная реализация DAO, и она может быть во многих отношениях усовершенствована.

Обеспечивающая безопасность типов реализация DAO. Один из способов усовершенствования приведенной выше реализации DAO — обеспечить безопасность типов в интерфейсе DAO. Это делает возможным типобезопасный DAO, который может быть реализован подынтерфейсом для каждого типа сущности, который требуется сохранить. Как может выглядеть базовый DAO, показано в листинге 12.7.

Листинг 12.7. Типобезопасный DAO

```
package com.devchronicles.dataaccessobject;

public interface BaseDAO<E, K> {
    public void create(E entity);
    public Movie retrieve(K id);
    public void update(E entity);
    public void delete(K id);
}
```

Первый параметр типа E представляет сущность, тогда как параметр типа K используется в качестве ключа. Позднее интерфейс BaseDAO может быть расширен подынтерфейсом, который бы определял специфические для этой сущности методы.

В листинге 12.8 вы создаете интерфейс, расширяющий BaseDAO и определяющий метод, который возвращает список всех фильмов.

Листинг 12.8. Специальная реализация базового DAO для фильмов

```
package com.devchronicles.dataaccessobject;

import java.util.List;

public interface MovieDAO2 extends BaseDAO<Movie, Integer> {

    public List<Movie> findAllMovies();

}
```

Конкретный класс может реализовать этот интерфейс и обеспечить код для каждого метода.

Где и когда использовать паттерн «Доступ к данным»

Иные могли бы поспорить, что DAO перестал быть полезным паттерном проектирования, поскольку вы легко можете вызвать EntityManager напрямую. Это разумный довод, ведь EntityManager предоставляет «чистое» API, абстрагирующее базовый слой доступа к данным. Кроме того, обоснованным будет предположение, что вероятность смены поставщика данных весьма низка, что делает обеспечиваемую DAO абстракцию менее осмысленной. Хотя эти аргументы и заслуживают внимания, все еще можно поспорить, что для DAO найдется место в хорошо спроектированном приложении для платформы Java EE (и это может быть не то место, которое ему предназначалось первоначально).

Значение расширения BaseDAO, как показано в листинге 12.7, для каждого типа сущности — в масштабируемости каждой реализации. Методы, специфичные для сущности, могут быть написаны при сопровождении общего интерфейса. Вы один раз выбираете реализацию DAO для каждой сущности, вместо того чтобы выбирать нужный метод EntityManager каждый раз, когда требуется сохранить или извлечь данные.

Именованные запросы могут быть расположены в сущности, к которой они относятся. Это позволяет хранить запросы в логичном для них месте, что облегчает сопровождение. DAO делает возможными унифицированные и управляемые стратегии доступа к данным, поскольку все обращения к данным сущности обязательно проходят через DAO сущности. Это происходит благодаря принципу персональной ответственности, поскольку только DAO обращается к данным приложения и управляет ими.

Не забудьте, что, хотя это и маловероятно, источник данных может измениться. Если это произойдет, вы будете рады, что на слое данных есть абстракция.

Резюме

У DAO есть свои поклонники и ненавистники. Решение, использовать ли этот паттерн в вашем приложении, должно основываться на проектных требованиях приложения. Как и в случае с другими паттернами, его использование ради использования потенциально опасно и может вызвать путаницу, поскольку абстракция скрывает назначение кода. Убедитесь, что вы хорошо понимаете различные реализации паттерна и то, как он взаимодействует с DTO и паттерном «Фабрика».

Упражнения

1. Напишите интерфейс и его реализацию для DAO заказа фильмов. Вы можете взять за основу примеры в тексте.
2. Напишите сервисный фасад и DTO, использующий MovieDAO.

13 Веб-сервисы, воплощающие REST

В этой главе:

- что такое REST и какие функции он выполняет;
- шесть ограничений REST;
- модель зрелости Ричардсона REST API;
- как спроектировать API, воплощающее REST;
- показываем REST в действии;
- как реализовать REST на платформе Java EE;
- HATEOAS — наивысший уровень модели зрелости Ричардсона.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 13.zip, каждый из них назван в соответствии с наименованиями в тексте.

Нет сомнений, что вы слышали термин REST. Но мы не уверены в том, что в точности понимаете, что он означает и как реализован. Многие люди, не знающие или знающие очень мало о REST, будут говорить вам, что ваш сайт должен быть «совместимым» с REST и что без REST ваш сайт, вероятно, не сможет сохранить работоспособность. REST для таких людей — модное словечко, но для тех, кто знает, что такое REST и какие выгоды он может принести, это гораздо больше, чем еще одно модное словечко. Так что же REST на самом деле означает и откуда он появился?

REST расшифровывается как «передача состояния представления» (REpresentational State Transfer) и является архитектурным стилем представления и передачи данных. Он состоит из набора в шесть ограничений, устанавливаемых на данные, компоненты и их взаимодействия в распределенной системе гипермедиа (Интернет). Он не привязан к конкретному протоколу (хотя почти во всех случаях используется с протоколом передачи гипертекста (HTTP)) и для него не существует стандарта консорциума Всемирной паутины (W3C). Это набор соглашений, стилей и подходов, относительно которых договорились за время использования.

Термин REST был придуман Роем Филдингом в его диссертации 2000 года на соискание степени PhD. Диссертация называлась «Архитектурные стили и проектирование сетевых архитектур программного обеспечения»[1]. С того времени концепция REST была настолько широко признана разработчиками и архитекторами, что стала неотъемлемой частью многих языков и фреймворков. Например, язык Ruby предоставляет «естественный» способ использования воплощающих REST маршрутов, а фреймворк Spring обеспечивает упрощенный способ реализации HATEOAS, являющегося третьим уровнем модели зрелости Ричардсона (более подробно об этом вы прочтете далее)[2].

О REST обычно говорят скорее как об архитектурном подходе, чем как о паттерне проектирования. Тем не менее REST был разработан для решения распространенных проблем, с которыми встречаются корпоративные приложения, что соответствует понятию паттерна проектирования.

Что такое REST

REST для разных людей означает разные вещи. Эта глава будет обсуждать его с точки зрения разработчика, желающего реализовать воплощающий REST интерфейс программирования приложений (API) для сайта форума, на котором обсуждают фильмы.

СЛОВАМИ РОЯ ФИЛДИНГА

REST подчеркивает масштабируемость взаимодействия компонентов, универсальность интерфейсов, независимое развертывание компонентов и их посредничество для снижения задержек, укрепления безопасности и инкапсуляции устаревших систем.

Целесообразнее всего думать о REST как о стиле форматирования URI, представляющих ресурсы (данные), которые ваше приложение может предоставить, и ресурсы, которые оно может хранить. Что подразумевается под ресурсами? На сайте форума у вас есть множество ресурсов, таких как пользователи сайта и их сообщения. Эти ресурсы представляют собой «имена существительные» и в сочетании с методами HTTP формируют унифицированные идентификаторы ресурсов (Uniform Resource Identifier, URI). Например, вы можете представить ресурс учетной записи с URI `/accounts` и соединить его с HTTP-методом `GET` так, что запрос к этому URI вернет все учетные записи. Подобным образом можно представить и ресурс идентифицируемой учетной записи путем добавления ID сообщения к URI вот так: `/accounts/:id`. Запрос `GET` к этому URI вернет подробности учетной записи с заданным ID. С помощью URI, воплощающего REST, вы не только можете получить представления ресурсов, но и создать ресурсы. Для этого вам необходимо создать URI с помощью HTTP-метода `POST`. Например, для создания нового ресурса учетной записи вам нужно будет отправить запрос `POST`

[1] *Roy Fielding*. Architectural Styles and the Design of Network-Based Software Architectures. — 2000. — Chapter 5. — http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[2] Леонард Ричардсон изложил свою модель воплощающей REST зрелости во время конференции QCon в 2008 году. — www.crummy.com/writing/speaking/2008-QCon/act3.html.

на URI `/accounts`, содержащий в теле HTTP-запроса необходимые для создания ресурса данные.

Как вы поняли, URI представляет ресурс на удаленном сервере и метод выполнения запросов к ресурсам (HTTP-метод) предполагает определенные действия над этим ресурсом.

Было бы заманчиво отобразить HTTP-методы на действия создания, извлечения, обновления и удаления (CRUD). Например, `POST` на создание и `GET` на чтение[1]. Однако это не в духе REST и не поможет в понимании представлений ресурсов. В действительности это ближе к реализации паттерна «Удаленный вызов процедур» (Remote Procedure Call, RPC), находящейся на 0-м уровне модели зрелости Ричардсона. Вы же заинтересованы лишь в реализации REST на наивысшем, 3-м, уровне (обратитесь к разделу «Модель зрелости Ричардсона API REST» за всеми подробностями).

Воплощающее REST API имеет дело не с действиями, а с «существительными». Эти «существительные» представляют ресурсы. Так, вы узнаете о ресурсах сообщений, о пользовательских ресурсах и об адресных ресурсах в противоположность «глаголам», таким как `getUser`, `addPost` и `deleteAddress`. REST отличается от простого протокола доступа к объектам (Simple Object Access Protocol, SOAP) и RPC, имеющим дело с действиями, которые вы хотите выполнить над данными приложения. В воплощающем REST смысле к URI обращаются с помощью соответствующего HTTP-метода.

Каждый ресурс идентифицируется с помощью URI. Может существовать множество способов сослаться на одни и те же ресурсы, так что вы сможете получить доступ к одному ресурсу с разных начальных точек. Например, вы можете получить ресурс, представляющий пользователя, обратившись к пользователю напрямик через URI `GET /users/:id` или перейдя от одного пользователя к его подписчикам и затем к нему: `GET /user/:id1/followers/:id1`. Представление ресурса не является реальным ресурсом, и для одного и того же ресурсадопустимо быть представленным несколькими способами.

Поток представлений ресурсов между клиентом и сервером двунаправленный и представляет по меньшей мере часть состояния ресурса. Когда это происходит, он содержит как раз достаточно данных, чтобы создавать, изменять или удалять такой ресурс на сервере.

Представления ресурсов обычно задаются в нотации объектов JavaScript (JavaScript Object Notation, JSON) или на расширяемом языке разметки (Extensible Markup Language, XML), но они могут быть в любом формате, включая пользовательское представление.

Шесть ограничений REST

Согласно диссертации Роя Филдинга, по-настоящему воплощающая REST архитектура соответствует всем, кроме одного, сформулированным ограничениям. Эта группа ограничений носит название «стиль REST».

---

[1] Относящийся к HTTP/1.1 стандарт RFC определяет следующие методы: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE и CONNECT.

Клиент-сервер

Клиент-серверное ограничение основано на принципе разделения ответственности и четко определяет разделение между клиентом и сервером. Ограничение несложно и требует от клиента отправлять запрос, а от сервера — получать запрос. Сервер может реагировать на запрос клиента.

Унифицированный интерфейс

Это ограничение определяет интерфейс между клиентом и сервером, заявляя, что он должен быть настолько общим и простым, насколько это только возможно. Вы уже узнали, что ресурс — представление данных и у клиента нет непосредственного доступа к данным. Это ограничение определяет, каким образом «существительные» представляют ресурс и то, что интерфейс поддерживается создателем действующей информационной системы. Это гарантирует некоторую степень неизменности интерфейсов на протяжении длительного времени. Ограничение не указывает, что реализация должна использовать протокол HTTP, но на практике она почти всегда основывается на HTTP. Как вы уже видели, при использовании вами спецификации HTTP URI составляется из ресурсов-«существительных» и «глаголов» HTTP.

Отсутствие сохранения состояния

Сервер не должен сохранять состояние клиента. Другими словами, каждое сообщение (запрос) само себя описывает, в нем есть достаточно информации или контекста с сервера для обработки сообщения. Отсюда следует, что, если состояние есть, его сохранение обеспечивается на стороне клиента. Преимущество этого ограничения состоит в том, что оно повышает масштабируемость, надежность и прозрачность. Недостаток — снижение производительности, поскольку сообщениям приходится быть больше в размерах для поддержания связи без сохранения состояния.

Кэшируемость

Ответы сервера должны быть кэшируемыми. Воплощающий REST интерфейс должен обеспечивать механизм для обозначения сообщений как кэшируемых или некэшируемых. Это может выполняться явным или неявным образом либо согласовываться, позволяя клиенту при необходимости повторно отправлять сообщение.

Многослойность системы

Клиент не может предполагать наличие прямого доступа к серверу. Ответ может быть кэшируемым, а может и извлекать ресурс прямо с сервера. Это служит улучшению масштабируемости, так как между клиентом и сервером может быть дополнительное программное или аппаратное обеспечение.

Код по запросу

Это ограничение определяет архитектуру REST как состоящую из иерархических слоев, чей обмен информацией ограничен их непосредственными соседями. Это разделение «забот» упрощает архитектуру и изолирует несхожие и устаревшие компоненты. Получаемые вами от этого принципа выгоды — в росте масштабируемости, поскольку новые компоненты легко могут быть внесены, а устаревшие — изъяты или замещены. К недостатку этого ограничения можно отнести снижение производительности системы из-за увеличения количества косвенных обращений, зависящих от многослойной структуры.

Такое ограничение позволяет клиенту скачивать и выполнять с сервера код, разрешающий серверу временно расширять возможности клиента за счет передачи дополнительной логики. Этот код может представлять собой фрагмент на языке JavaScript. Данное ограничение необязательно.

Нарушение любого из вышеприведенных ограничений (кроме кода по запросу) означает, что сервис не строго воплощает REST. Однако нарушение ограничения не значит, что сервис — нежизнеспособная и бесполезная реализация.

Модель зрелости Ричардсона API REST

Вы уже прочли о том, как по-настоящему воплощающий REST интерфейс программирования приложений достигает 3-го уровня модели зрелости Ричардсона. Теперь заглянем немного глубже и рассмотрим каждый из уровней модели.

Модель, разработанная Леонардом Ричардсоном, пытается классифицировать интерфейсы программирования приложений в соответствии с соблюдением налагаемых REST ограничений. Чем более совместимо ваше приложение, тем лучше оно работает. Существует четыре уровня. Нижний уровень — 0-й, который означает наименее совместимую реализацию, и высший — 3-й, который наиболее совместим и поэтому в наилучшей степени воплощает REST[1].

Уровень 0. «Болото» POX[2]

Эта модель использует HTTP в качестве транспортного протокола для активизации удаленных взаимодействий. Модель не пользуется протоколом для индикации состояния приложения, обычно он применяется просто для туннелирования запросов и ответов на один URI, такой как `/getUser`, используя при этом только один метод HTTP. Это классический пример модели RPC, который больше напоминает SOAP и XML-RPC, чем REST.

---

[1] Мартин Фаулер на своем сайте приводит отличный обзор: www.martinfowler.com/article/richardsonMaturityModel.html.

[2] Plain Old XML (простой старый XML). — *Примеч. пер.*

Уровень 1. Ресурсы

На этом уровне модель способна отличать разные ресурсы друг от друга. Она будет взаимодействовать с различными конечными точками, поскольку каждая конечная точка представляет другой ресурс. Применяются URI вида `POST resources/123`, но модель все еще использует только один HTTP-метод.

Уровень 2. «Глаголы» HTTP

На этом уровне вы реализуете использование в полном объеме «глаголов» HTTP и сочетаете их с вашими ресурсами-«существительными» для обеспечения того типа REST, который обсуждался ранее в этой главе. Вы пользуетесь всеми преимуществами тех возможностей, который HTTP предоставляет для реализации вашего воплощающего REST API.

Уровень 3. Управляющие элементы гипермедиа

На этом уровне модель использует HATEOAS (Hypermedia as the Engine of Application State, «гипермедиа как движок состояния приложения») для управления состоянием приложения. Задача управляющих элементов гипермедиа заключается в том, чтобы «советовать» клиенту, что может быть выполнено в следующую очередь, и снабжать его URI, необходимыми для выполнения следующего действия. Вы увидите, как это работает и как реализовать HATEOAS, дальше в этой главе.

Проектирование воплощающего REST API

Хорошо спроектированное воплощающее REST API подразумевает хорошо описанный унифицированный интерфейс. Для его создания важно досконально понимать методы HTTP и коды ответа и необходимо максимально полно знать структуры данных вашего приложения. Задача — объединить их в простой, «чистый» и изящный URI ресурса.

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

В одной компании, на которую я работал, была традиция: когда группа разработчиков завершала проект, она представляла его другим группам. Это было примерно в то же время, когда набирал популярность REST. Одна из групп решила построить прикладную часть на основе REST для обслуживания как мобильных, так и веб-клиентов. Мы были в восхищении и внимательно слушали, как они успешно построили прекрасно спроектированную прикладную часть на основе REST, которая была в состоянии обрабатывать данные двух разновидностей систем. Когда руководитель группы начал рассказывать технические детали системы, мы осознали, что они сохраняют состояние клиента на стороне сервера. Отнюдь не воплощение REST!

Я поднял этот вопрос, спросив, действительно ли проект следовал принципам воплощения REST? Само собой, архитектор проекта был оскорблен моими словами и начал демонстрировать нам документацию API REST и то, как каждый клиент будет работать с OAuth и передавать параметры через унифицированные указатели ресурса (Uniform Resource Locator, URL). Тем не менее система основывалась скорее на сохранении состояния, а не его передаче.

Подобно модникам, разработчики и проектировщики систем обожают тенденции и хотят выглядеть ультрамодно. Однако без понимания базовых принципов и выяснения, действительно ли они отвечают вашим нуждам и задачам, вы можете показаться нелепо выглядящим человеком, который хочет быть ультрамодным, но не сумел полностью разобраться, как это сделать. В их случае они построили прикладную часть, воплощающую «сохранение состояния представления» (RESP?) вместо REST.

Скоро вы узнаете, из каких элементов состоит URI.

Именование ресурсов

Воплощающие REST API написаны для клиентов и должны быть осмысленными для клиентов этих API. При выборе существительных для именования ресурсов вы должны быть знакомы со структурой данных приложения и тем, как ваши клиенты, вероятнее всего, будут их использовать. Не существует четко описанных правил о том, как вам следует именовать ваши ресурсы, но имеются соглашения, следование которым может помочь вам создать информативные имена ресурсов, интуитивно понятные остальным.

Существительные, а не глаголы

Вы должны именовать ресурсы «в честь» существительных, а не глаголов или действий. Цель имени ресурса — представлять ресурс. Метод HTTP описывает подлежащее выполнению действие. Следующий раздел рассматривает методы HTTP подробнее. Для представления ресурса отдельного пользователя можно указывать существительное `users` («пользователи») для представления всех пользователей и ID для идентификации конкретного пользователя, вот так: `users/123456`. Примером несоответствующего REST и неправильно составленного URI может быть `users/123456/update` (или URI, включающий действие в строку запроса вот так: `users/123456?action=update`).

Сама сущность данных состоит в том, что они иерархичны. Допустим, вы хотите представить все сообщения пользователя с ID `123456`. Вы могли бы использовать существительное `noun` для представления всех сообщений и создать URI `users/123456/posts`. Ранее было упомянуто, что представление ресурса — это не сам реальный ресурс и один и тот же ресурс может быть представлен разными способами. Чтобы представить все сообщения конкретного пользователя, вы можете указывать URI `posts/users/123456`. Когда у вас уже есть представление ресурса, вы можете решить, что хотите с ним сделать, с помощью одного из четырех методов HTTP. Для извлечения ресурса можно воспользоваться методом `GET`, а для создания ресурса — методом `POST`. Больше об этом читайте в следующем разделе.

Информативность

Как вы уже видели, выбранные существительные должны отражать представляемый ими ресурс. Сочетание этих представлений с идентификаторами облегчает интерпретацию ресурса и делает его интуитивно понятным. Если вы читаете URI

в сочетании с его методом HTTP и вам сразу не становится очевидно, какой ресурс он представляет, то этот URI — неудачное воплощение REST.

Множественное, а не единственное число

Имена ресурсов должны быть во множественном числе, поскольку они представляют множества данных. Имя ресурса `users` представляет множество пользователей, а имя ресурса `posts` — множество сообщений. Идея в том, что эти существительные во множественном числе представляют собой множество, а ID ссылается на один элемент этого множества. Использование существительного в единственном числе может быть оправданно, если во всем приложении существует только один экземпляр такого типа данных, однако это не слишком распространенная практика.

Методы HTTP

В спецификации HTTP 1.1 определено восемь методов HTTP, однако только четыре из них широко используются при проектировании воплощающих REST API. Это методы `GET`, `POST`, `PUT` и `DELETE`. У них есть четко определенные смыслы и способы применения в контексте REST. При рассмотрении методов HTTP особое значение имеет концепция идемпотентности. Смысл идемпотентности с точки зрения воплощения REST API в том, что повторные обращения клиента к одному URI всегда будут возвращать один и тот же результат. Соответственно, выполнение запроса на сервере один раз приводит к тому же результату, что и выполнение того же запроса много раз (предполагается, что различные отдельные операции не изменили состояния ресурса).

Только один из четырех чаще всего используемых методов HTTP идемпотентен: `GET`. Это означает, что никакой выполняемый с помощью этого метода URI ресурса не вызовет изменений на сервере. Вы не можете применять его для создания, обновления или удаления ресурса. Спецификация HTTP 1.1 относит этот метод к безопасным, поскольку он «не должен иметь иного значения, кроме извлечения [данных]». В контексте REST этот метод применяется для получения от сервера представления ресурса. Он никогда не должен использоваться для выполнения изменения данных.

Остальные три метода — `POST`, `PUT` и `DELETE` — не являются идемпотентными и ожидаемо могут выполнять изменения на сервере. Далее вы узнаете о каждом методе и их использовании в контексте сайта форума. Вы также узнаете о кодах ответа HTTP (**http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html**), которые могут быть возвращены вместе с ответами клиенту, и о том, что они означают.

GET

Этот метод используется для получения от сервиса представления ресурса. Вам ни при каких обстоятельствах не следует применять его для создания, обновления или удаления ресурса. Обращение к нему однократно должно иметь такой же результат, как и обращение к нему 100 раз. Если запрос ресурса успешен, представление ресурса возвращается в теле HTTP-ответа в запрошенном формате данных, чаще все-

го JSON или XML. Возвращаемый при этом код ответа HTTP — 200 (OK). Если ресурс не найден, должно быть возвращено 404 (NOT FOUND), а если запрос ресурса составлен неправильно, должно быть возвращено 400 (BAD REQUEST). Правильно составленный URI, который вы можете использовать в вашем приложении для форума, может быть GET users/123456/followers. Он представляет всех пользователей-подписчиков пользователя 123456.

POST

Этот метод используется для создания нового ресурса в заданном контексте. Например, для создания нового пользователя вы можете отправить на ресурс users необходимые для этого данные. Сервис позаботится о создании нового ресурса, связывании его с контекстом и присвоении ID. При успешном создании ответ HTTP будет 201 (CREATED) и будет возвращена ссылка на вновь созданный ресурс — или в заголовке Location ответа, или в содержимом JSON в теле ответа. Представление ресурса может быть возвращено в теле ответа, что часто предпочтительнее во избежание дополнительных обращений к API для извлечения представления только что созданных данных. Это снижает количество операций обмена информацией API.

PUT

Метод PUT в большинстве случаев применяется для обновления известного ресурса. URI содержит достаточно информации для идентификации ресурса, такой как контекст и идентификатор. Тело запроса содержит обновленную версию ресурса и, если обновление прошло успешно, возвращается код HTTP-ответа 200. URI, обновляющий пользовательскую информацию, таков: PUT users/123456. Реже для создания ресурса используется метод PUT, если клиент создает идентификатор ресурса. Однако этот способ создания ресурса несколько смущает. Зачем использовать PUT, если POST работает ничуть не хуже и при этом общеизвестен? Нельзя не отметить важный момент относительно обновления ресурса: сервису в теле HTTP-запроса передается все представление ресурса, а не только изменившаяся информация.

DELETE

Как ни удивительно, этот метод применяется для удаления ресурса с сервиса. URI содержит контекст и идентификатор ресурса. Чтобы удалить пользователя с ID 123456, используется URI DELETE users/123456. Тело ответа может включать представление удаленного ресурса. Успешное удаление приводит к возвращению кода HTTP-ответа 200 (OK). Если ресурс не найден, возвращается код 400.

REST в действии

Итак, вы собираетесь спроектировать воплощающее REST API для сайта форума, используя всю полученную выше иформацию.

Вы начнете с изучения структуры данных сайта и определения предметных областей. Две главные предметные области — это пользователи и сообщения. Пользователи могут быть далее разбиты по подписчикам, а сообщения обычно группируются по темам. Итак, имея эти предметные области, начнем думать об URI, необходимом для представления этих ресурсов.

Существительное users

Вы знаете, что для создания нового пользователя необходимо применять POST и контекст `users`, так что создающий пользователя URI будет выглядеть таким образом:

```
POST /users
```

Тело запроса содержит все необходимое вам для создания нового пользователя. Ответ включает URI для представления пользователя. Это запрос метода `GET`, он выглядит вот так:

```
GET /users/123456
```

Он запрашивает подробности о пользователе с ID 123456.

Если вы хотите обновить этого пользователя, вам нужно будет добавить метод `PUT`, вот так:

```
PUT /users/123456
```

Если вы хотите удалить этого пользователя, применяйте метод `DELETE`, вот так:

```
DELETE /users/123456
```

Если хотите выполнить пакетное удаление, можно передать все ID пользователей, которых вы хотите удалить, в теле обращения к `DELETE /users`. Это потребует меньше операций обмена информацией, чем множество обращений к ресурсам каждого пользователя.

Вы можете захотеть получить всех пользователей сервиса вот так: `GET /users`. Такое обращение будет, конечно же, ограничиваться из соображений безопасности, так что будут возвращены только те пользователи, просматривать которых вызывающий авторизован.

Это все, что касается контекста пользователя. Теперь посмотрим на *подписчиков*. Подписчик — это пользователь, который читает другого пользователя, поскольку интересуется его сообщениями.

Чтобы получить всех подписчиков данного пользователя, применяйте метод `GET`, вот так:

```
GET /users/123456/followers
```

Чтобы создать нового подписчика пользователя, передайте ID этого подписчика в теле запроса к `POST /users/123456/followers`. Вы можете получить подробности подписчика одним из двух способов:

```
GET /users/123456/followers/456789
```

или

```
GET /users/456789
```

Это пример того, как можно представить ресурс двумя различными образами. Чтобы удалить подписчика данного пользователя, вы можете сделать следующее:

```
DELETE /users/123456/followers/456789
```

Это действие удалит пользователя 456789 из числа подписчиков пользователя 123456, но не удалит его на самом деле. А следующее действие все-таки удалит его полностью:

```
DELETE /users/456789
```

Вы прочитали про контекст подписчиков. Теперь посмотрим на темы и сообщения.

Существительное topics и существительное posts

Вы уже видели, как создать пользователя с помощью метода POST. То же самое справедливо относительно создания темы и сообщения.

Для создания темы используйте URI:

```
POST /topics
```

Для создания сообщения в теме используйте:

```
POST /topics/123/posts
```

Обратите внимание на то, что нельзя создать сообщение с помощью такого URI:

```
POST /posts
```

поскольку у вас нет контекста. Здесь недостаточно информации для сервиса, чтобы создать сообщение, поскольку неизвестно, с какой темой это сообщение должно быть связано.

Чтобы получить тему и сообщение, используйте:

```
GET /topics/123
```

Чтобы получить конкретное сообщение в теме, используйте:

```
GET /topics/123/posts/456
```

или

```
GET /posts/456
```

Чтобы удалить тему или сообщение, используйте:

```
DELETE /topics/123
```

Чтобы удалить сообщение, используйте:

```
DELETE /topics/123/posts/456
```

или

```
DELETE /posts/456
```

Для внесения изменений в тему или сообщение можно воспользоваться любым из следующих трех способов:

```
PUT /topics/123
PUT /topics/123/posts/456
```

или

```
PUT posts/456
```

Теперь, когда у вас определены простейшие URI и контексты, можно начать развлекаться и комбинировать пользователей, темы и сообщения различными образами, чтобы усложнить их.

Чтобы получить представление всех тем, размещенных данным пользователем, укажите следующее:

```
GET /users/123456/posts
```

Если хотите получить все сообщения в конкретной теме для данного пользователя, введите следующее:

```
GET /users/123456/topics/123/posts
```

Чтобы получить все сообщения, размещенные подписчиком данного пользователя в данной теме, укажите следующее:

```
GET /users/123456/followers/456789/topics/123/posts
```

Вы можете творчески подойти к комбинациям ресурсов. Несмотря на кажущуюся сложной сущность, вы никогда не должны забывать, что клиенты воплощающего REST API используют URI, так что URI должны быть понятными и вложенность должна быть сведена к минимуму. Если вы чувствуете, что ваш URI настолько сложен, что его лучше сопроводить пояснением, вам лучше подумать о его рефакторинге.

Пример хорошо спроектированного и продуманного API, воплощающего REST, был разработан компанией Sugarsync (**https://www.sugarsync.com/developer**), которая занимается хранением данных. Рекомендуем вам просмотреть их ссылки на ресурсы, чтобы увидеть, как они понятно определили ресурсы папки, адреса и рабочей области. Обратите внимание, как методы HTTP используются для создания, чтения и удаления ресурсов.

Реализация REST в Java EE

В этой главе была подробно рассмотрена теоретическая часть хорошо спроектированного воплощающего REST API. Теперь, когда вы увидели, как можно конструировать URI, вы можете посмотреть, как это все будет выглядеть в коде.

Платформа Java EE 7 предоставляет некоторые полезные аннотации, упрощающие задачу создания воплощающего REST API. Полезнее всего аннотация `@Path`. Она определяет контекст URI и класс или метод, который будет обрабатывать выполняемые к этому URI запросы. Кроме того, есть аннотации для каждого HTTP-метода: `@GET`, `@POST`, `@PUT`, `@DELETE` и т. д. Эти аннотации отмечают методы, обрабатывающие запросы, которые были выполнены с помощью соответствующего HTTP-метода.

Ваше приложение может иметь несколько воплощающих REST контекстов. Чтобы позаботиться об этом, аннотация `@ApplicationPath` принимает параметр, указывающий пространство, в котором будет существовать подобное API. Благо-

даря всего лишь двум этим типам аннотаций у вас есть все необходимое для реализации простого воплощающего REST API.

А теперь в листинге 13.1 вы реализуете URI `GET /users`.

Листинг 13.1. Простейшая реализация воплощающего REST API в Java EE

```
package com.devchronicles.forum;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Application;

@ApplicationPath("/")
@Path("users")
public class Users1 extends Application {
    @GET
    public String getUsers() {
        return "Тут мы возвращаем представление всех пользователей";
    }
}
```

Если вы развернете это приложение на своей локальной машине и ваше приложение будет называться `forum`, то сможете проверить этот URI, посетив страницу **http://localhost/forum/users**. Обратите внимание на текстовое сообщение `Тут мы возвращаем представление всех пользователей`, выведенное в окне браузера.

Обратите внимание в листинге 13.1 на то, как вы аннотировали класс с помощью `@Path` и передали ей контекст `users`. Передаваемой вами строке не должен предшествовать прямой слеш и за ней не должен следовать обратный слеш. Пространство, в котором будет существовать воплощающий REST интерфейс, было определено как корневой каталог («/») приложения. Вызываемый при выполнении к URI `users` запроса `GET` метод аннотирован с помощью `@GET`. В этом простом примере возвращается строка, но настоящая цель — вернуть извлеченные из БД данные в формате JSON или XML наряду с кодом статуса HTTP. Это то, что происходит в листинге 13.2.

Листинг 13.2. Ответ клиенту с содержимым JSON

```
package com.devchronicles.forum;

import java.util.ArrayList;

import javax.json.Json;
import javax.json.JsonArrayBuilder;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Application;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@ApplicationPath("/")
```

```
@Path("users")
public class Users2 extends Application {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getUsers() {

        ArrayList<User> allUsers = this.findAllUsers();
        JsonArrayBuilder jsonArrayBuilder = Json.createArrayBuilder();

        for (User user : allUsers) {
            jsonArrayBuilder.add(Json.createObjectBuilder()
                            .add("id", user.getId())
                            .add("firstname", user.getFirstname())
                            .add("lastname", user.getFirstname())
            );
        }

        return Response.ok(jsonArrayBuilder.build()).build();
    }

    public ArrayList<User> findAllUsers() {
        ArrayList<User> allUsers = new ArrayList<>();
        allUsers.add(new User(123456, "Alex", "Theedom"));
        allUsers.add(new User(456789, "Murat", "Yener"));
        return allUsers;
    }
}
```

В коде листинга 13.2 формируется объект JSON из пользовательских данных, находящихся в базе данных (для краткости: метод применяется для возвращения пользовательских данных), и отправляется обратно клиенту с кодом ответа HTTP 200. Первая вещь, на которую вы можете обратить внимание, — то, что метод `getUsers()` теперь аннотирован `@Produces(MediaType.APPLICATION_JSON)`. Это определяет тип MIME, который этот метод может производить и возвращать клиенту. Для построения JSON и оборачивания его в объект `javax.ws.rs.core.Response` перед возвратом клиенту применяются классы `javax.json.Json` и `javax.json.JsonArrayBuilder`. Если все пройдет успешно, вы увидите в браузере следующий вывод:

```
[
    {"id":123456,"firstname":"Alex","lastname":"Theedom"},
    {"id":456789,"firstname":"Murat","lastname":"Yener"}
]
```

Пока вы увидели, как извлечь представление ресурса всех пользователей системы, но что, если вас интересует только один пользователь и вы знаете его идентификационный номер? Что ж, это столь же просто. В URI вы передаете ID пользователя вот так:

```
GET /users/123456
```

И в управляющем классе REST вы восстанавливаете номер ID, ссылаясь на путь REST и используя переменную URI для передачи ID методу. Вы аннотируете метод, который будет получать вызов от воплощающего REST API, — `@Path("/{id}")` и в сигнатуре метода аннотируете параметр, через который должно быть передано значение ID:

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getUser(@PathParam("id") String id){
    User user = this.findUser(id);
    JsonArrayBuilder jsonArrayBuilder = Json.createArrayBuilder();
    jsonArrayBuilder.add(
        Json.createObjectBuilder()
            .add("id",user.getId())
            .add("firstname", user.getFirstname())
            .add("lastname", user.getLastname())
    );

    return Response.ok(jsonArrayBuilder.build()).build();
}

public User findUser(String id){
    return new User("123456", "Alex","Theedom");
}
```

Как вы можете видеть в предыдущем фрагменте кода, строковый параметр ID аннотирован `@PathParam("id")` так, что восстановленный с помощью аннотации `@Path("/{id}")` из URI ID передается в метод. Нет необходимости включать полный путь URI в аннотацию `@Path`, поскольку базовый URI установлен в аннотации `@Path` для класса. Все установленные для методов пути являются относительными и отсчитываются от базового пути, устанавливаемого для класса.

Переменная URI может быть регулярным выражением. Например, аннотация пути `@Path("/{id: [0-9]*}")` будет соответствовать только числовым ID. Любые несоответствующие ID будут приводить к возврату клиенту ответа HTTP 404.

Вы увидели простые конструкции URI, состоящие из одного ресурса-«существительного» и переменной URI. А что делать с более сложными URI, такими как `GET /users/123456/followers/456789`? То же, что и раньше, только с немного более сложными `@Path` и `@PathParam`.

```
@GET
@Path("/{user_id}/followers/{follower_id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getUser(
@PathParam("user_id") String user_id,
@PathParam("follower_id") String follower_id)
```

Вы обстоятельно изучили HTTP-метод `GET`. А что насчет методов `POST`, `PUT` и `DELETE`? Чтобы написать отвечающий на HTTP-запрос метод `POST`, вы поступаете так же,

как поступили бы с GET, но меняете два компонента. Вы аннотируете метод аннотацией @POST вместо @GET и @Consumes вместо @Produces. Вот простой пример:

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Path("/{user_id}/followers/")
public Response createUser(@PathParam("user_id") String user_id. String body)
```

Этот пример работает аналогично таковому для метода GET, однако обратите внимание, что здесь нет явного отображения тела HTTP-запроса на параметр метода. Это отображение выполняется неявным образом. Содержимое тела HTTP-сообщения передается только неаннотированному параметру, найденному в сигнатуре метода. Во избежание путаницы допускается не более одного такого параметра.

HTTP-методы PUT и DELETE работают аналогично методу POST.

URI может содержать параметры HTTP-запроса. Их можно извлечь из URI посредством аннотирования параметра в сигнатуре метода аннотацией @QueryParam("page"). Она извлекает параметр HTTP-запроса page из URI /users?page=10.

В интерфейсе программирования приложений JAX-RS есть множество других аннотаций, облегчающих проектирование хорошо воплощающих REST API. Рекомендуем вам ознакомиться с ними.

HATEOAS

Как уже обсуждалось, HATEOAS — это наивысший уровень реализации REST в модели зрелости Ричардсона, который должен рассматриваться в качестве нирваны воплощенности REST.

Допустим, клиент запрашивает ресурс, представляющий все разрешенные пользователю для просмотра сообщения в системе. URI в таком случае будет GET /posts, а ответ, в случае успешного выполнения, вернется со следующим телом HTTP-сообщения:

```
{
  "posts": [
    {
      "id": 71892,
      "title": "Best movie of 2015",
      "content": "I think the best movie of 2015 is the Golden Egg of Siam.",
      "links": [
        {
          "rel": "self",
          "href": "http://localhost:8080/rest/posts/71892",
          "method": "GET"
        },
        {
          "rel": "replies",
          "href": "http://localhost:8080/rest/posts/71892/posts",
          "method": "GET"
        },
        {
```

```
          "rel": "follower",
          "href": "http://localhost:8080/rest/posts/71892/followers",
          "method": "GET"
        },
        {
          "rel": "owner",
          "href": "http://localhost:8080/rest/posts/71892/users",
          "method": "GET"
        }
      ]
    },
    {
      "id": 71893,
      "title": "Worst movie of 2015",
      "content": "The worst movie has gotta be Just Being Me.",
      "links": [
        {
          "rel": "self",
          "href": "http://localhost:8080/rest/posts/71893",
          "method": "GET"
        },
        {
          "rel": "owner",
          "href": "http://localhost:8080/rest/posts/71893/users",
          "method": "GET"
        }
      ]
    }
  ]
}
```

Здесь многое происходит. Ответ находится в файле в формате JSON и сформирован из одного объекта JSON с именем `post`, содержащего массив сообщений:

```
{
  "posts": [
    {
      ... первый элемент массива
    },
    {
      ...второй элемент массива
    }
  ]
}
```

В этом примере в массиве только два элемента, представляющие два сообщения. Каждое сообщение — объект JSON в том же формате:

```
{
  "id": 71892,
  "title": "Best movie of 2015",
  "content": "I think the best movie of 2015 is the Golden Egg of Siam.",
}
```

Как можно видеть из фрагмента JSON, он содержит ID, идентифицирующий ресурс сообщения, за которым следует название и содержимое сообщения. Это тот минимум, который можно ожидать в ответе на запрос ресурса сообщения, вне зависимости от зрелости интерфейса REST. Отличает этот ответ элемент `links`.

```
"links": [
  {
    "rel": "self",
    "href": "http://localhost:8080/rest/posts/71892",
    "method": "GET"
  },
  ...
]
```

Именно тут появляется на свет HATEOAS. В каждой ссылке в массиве ссылок имеются три составные части: `rel` — это отношение, которое имеет ссылка `href` к текущему ресурсу; `href` — ссылка на дополнительные ресурсы, а `method` — метод, который должен использоваться для получения ресурса. Элемент `rel` может принимать любое значение и не обязан следовать соглашениям, хотя `'self' rel` по традиции относится к ссылке, представляющей дополнительную информацию относительно текущего ресурса. В приведенном примере он просто ссылается на самого себя. Другие ссылки относятся к ответам других пользователей на данное сообщение (ответы или отклики), наблюдающих за этим сообщением пользователей (подписчиков) и разместившего сообщение пользователя (владельца). Любой ресурс, связанный с ресурсом основного сообщения, может быть представлен ссылкой в массиве ссылок.

Как вы могли видеть из примера, первое сообщение в массиве имеет четыре ссылки в его массиве ссылок, в то время как второе — только две. Это потому, что у второго сообщения нет наблюдающих за ним пользователей или каких-либо ответов.

Такой способ предоставления ссылок дает клиенту необходимую ему для нахождения пути к другим ресурсам информацию и позволяет вам легко расширять и наращивать API без особых трудностей.

В качестве примера хорошо спроектированной реализации HATEOAS можно привести Paypal.com (**https://developer.paypal.com/webapps/developer/docs/integration/direct/paypal-rest-payment-hateoas-links/**). Разработчики использовали HATEOAS, чтобы предоставить возможность построения API, взаимодействующего с их платежной системой посредством простого перехода по предусмотренным в массиве ссылкам.

Где и когда использовать REST

REST — простой и хорошо отработанный подход, не скованный стандартами. Можно было бы поспорить, что это явный недостаток по сравнению с SOAP, являющимся отраслевым стандартом с его собственным четко определенным протоколом и правилами реализации. Однако легкость реализации и использования перевешивают любые сложности, создаваемые отсутствием стандарта. Создание воплощающего REST ресурса столь же просто, как предоставление URI, а применение HTTP-про-

токола делает простым «межъязыковое» взаимодействие. Общим языком является HTTP, язык Интернета, простой и понятный для всех.

Ситуации с ограниченной пропускной способностью не представляют проблем для «легковесного» подхода REST, что особенно привлекательно для мобильных устройств. Выполнение запроса от воплощающего REST API обходится «недорого». Вспомните, что это просто HTTP-запрос и что возвращаемые данные могут быть в любом подходящем формате. Формат не обязательно должен быть JSON или XML, это может быть формат Atom Syndicate (ATOM) или какой-то пользовательский формат. Гибкость, приносимая использованием всего лишь простого URI для представления ресурса, позволяет разработчикам клиентской части проявлять их творческие способности. Вы можете подключать асинхронный JavaScript или XML (AJAX) для обращения к одному или нескольким URI, возможно, от различных поставщиков REST. Вы можете объединять ответы на эти обращения для обеспечения более «богатого» содержимого для посетителей сайта.

Если ваша реализация REST использует протокол HTTP (что почти наверняка), то в виде бесплатного бонуса вы получаете возможность кэширования. Протокол HTTP поддерживает кэш в качестве базовой возможности, которую вы можете использовать в приложении, устанавливая значения заголовков HTTP.

Резюме

REST API вашего приложения должно быть простым и интуитивно понятным для применения другими разработчиками, и ваша обязанность как разработчика — спроектировать API, которое не только удовлетворяет требованиям вашего приложения, но и гарантирует, что пользователи API смогут обращаться к ресурсам, необходимым для надлежащего функционирования их приложения.

В этой главе мы коснулись только основ правильного проектирования воплощающего REST API. Есть много других соображений, нами даже не упомянутых, таких как безопасность, использование строк запросов и то, как запускать второстепенные серверные задачи.

К счастью, REST вездесущ и существует масса ресурсов, которые могут помочь вам разработать API, действительно воплощающее REST. Просто поищите в Интернете, и вы не будете знать, что делать с обилием статей, форумов и книг, посвященных теме REST.

Упражнения

1. Поищите в Интернете общедоступные воплощающие REST API и напишите простые программы, использующие эти API.
2. Реализуйте URI для сайта форума, подробно описанного в предшествующем тексте, и напишите обращающуюся к ним клиентскую часть.
3. Разработайте перемещение по сайту, используя подход, полностью соответствующий HATEOAS-стилю.

14 Паттерн «Модель — представление — контроллер»

В этой главе:

- введение в паттерн MVC;
- происхождение паттерна MVC;
- как реализовать паттерн MVC с помощью составных паттернов;
- реализация паттерна MVC в Java EE;
- когда и где использовать паттерн MVC.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 14.zip, каждый из них назван в соответствии с наименованиями в тексте.

Паттерн «Модель — представление — контроллер» (Model — View — Controller, MVC) — один из распространенных архитектурных паттернов проектирования в современной разработке приложений из числа перечисленных в книге «Банды четырех». Он основывается на философии разделения обязанностей и инкапсуляции обработки данных приложения от представления этих данных. Если не инкапсулировать обработку данных приложения от их представления, можно получить сильно связанную систему, неудобную для сопровождения и расширения. Разделение обязанностей, обеспечиваемое паттерном «Модель — представление — контроллер», дает возможность выполнять изменения как в бизнес-логике, так и в пользовательском интерфейсе более независимо друг от друга.

Использование паттерна MVC не сильно отличается от покупки подписки у поставщика кабельного телевидения и телевизора в магазине электроники. Один обеспечивает контент, а другой заботится о том, чтобы этот контент правильно отображался. Никто из них не беспокоится насчет изменения технологий по ходу дела. Вы всегда можете купить новый телевизор при появлении лучших образцов или подписаться на дополнительные каналы без покупки нового оборудования.

Паттерн MVC широко используется при разработке веб-приложений, и мы будем обсуждать его реализацию именно в этом контексте.

Что такое паттерн проектирования MVC?

В паттерне «Модель — представление — контроллер» модель представляет данные приложения и связанную с ними бизнес-логику. Модель может быть представлена одним объектом или сложным графом связанных объектов. В приложении для платформы Java EE данные инкапсулируются в объектах предметной области, часто развертываемых в EJB-модуле. Данные передаются в БД и из нее в объектах передачи данных (DTO), и к ним обращаются с помощью объектов доступа к данным (DAO) (см. главу 12).

Представление — это наглядное отображение содержащихся в модели данных. Подмножество модели содержится в отдельном представлении, таким образом, представление действует в качества фильтра для данных модели. Пользователь взаимодействует с данными модели с помощью предлагаемого представлением наглядного отображения и обращается к бизнес-логике, которая, в свою очередь, воздействует на данные модели.

Контроллер связывает представление с моделью и управляет потоками данных приложения. Он выбирает, какое представление визуализировать для пользователя в ответ на вводимые им данные и в соответствии с выполняемой бизнес-логикой. Контроллер получает сообщение от представления и пересылает его модели. Модель, в свою очередь, подготавливает ответ и отправляет его обратно контроллеру, где происходит выбор представления и отправка его пользователю.

Паттерн MVC логически охватывает клиента и промежуточный уровень многоуровневой архитектуры. В среде Java EE модель располагается в бизнес-слое, обычно в виде EJB-модуля. Контроллер и представление расположены на веб-уровне[1]. Представление, вероятнее всего, будет создано из JavaServer Faces (JSF) или JavaServer Pages (JSP) с помощью языка выражений (EL). Контроллер обычно представляет собой сервлет, получающий HTTP-запросы от пользователя (см. главу 2, в которой обсуждаются многоуровневая архитектура и различные слои приложения).

Часто MVC сочетается с другими паттернами, такими как «Команда» (или «Действие»), «Стратегия», «Компоновщик» и «Наблюдатель». Данная глава не углубляется в детали этих паттернов, но паттерна «Действие» мы коснемся далее в одном из примеров.

ПРЕДПОСЫЛКИ

Впервые этот паттерн упоминался еще до создания Интернета в современном виде, в статье, опубликованной в декабре 1979 года работавшим тогда в компании Xerox SmallTalk-программистом Трюгве Ринскаугом[2].

И хотя элементы MVC этого паттерна были описаны более 35 лет назад, они удивительно точно соответствуют современному их использованию в веб-приложениях.

---

[1] По-видимому, авторы имеют в виду веб-слой. — *Примеч. пер.*

[2] Специализированный сайт Трюгве М. Х. Ринскауга: https://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

Рисунок 14.1 показывает пользователя, выполняющего запрос к контроллеру. Контроллер обрабатывает запрос путем обновления модели и визуализации нового представления, которое затем отправляется пользователю.

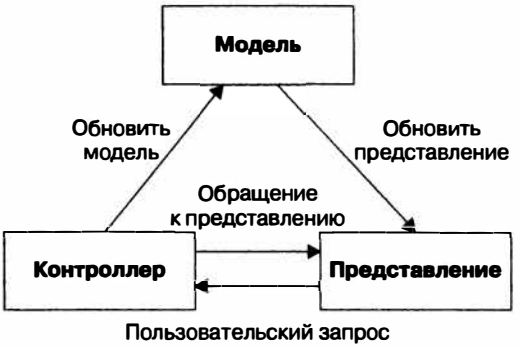


Рис. 14.1. Диаграмма паттерна «Модель — представление — контроллер»

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Вернемся в те времена, когда JSF еще был последним писком моды, а «проблема 2000» не привела к ядерному апокалипсису, как было обещано. Я работал в маленьком веб-стартапе. Наш стартап состоял всего лишь из небольшого количества разработчиков JSP/Java и нескольких разработчиков флеш-анимации. Мы хотели создать веб-портал, который бы предоставлял динамический контент в зависимости от конкретных нужд клиента, заключившего с нами договор.

Итак, мы начали проект с полным воодушевлением и разработали более чем динамический сайт, предоставлявший различные возможности разным клиентам. Мы искренне гордились своим творением, и было немало клиентов, покупавших доступ к нашему замечательному сайту. Фактически мы стали вполне преуспевающими и к тому же довольно быстро. Казалось, что все по-настоящему довольны нашим продуктом, и мы были рады нашему успеху. Однако эта радость оказалась кратковременной. По мере того как все новые и новые клиенты покупали наш продукт, сайт становился все более и более сложным в управлении. Что мы сделали, так это смешали бизнес-логику с логикой представления, поэтому для каждого нового клиента нам приходилось вносить изменения во все JSP сайта. Вскоре JSP превратились в чудовищную мешанину бизнес-логики и логики отображения и мы получили неуправляемый спагетти-код. Это превратилось в кошмар, и у нас не было другого выхода, кроме как переписать все приложение, но на этот раз реализуя паттерн MVC.

Мы переписали приложение, и оно стало управляемым, но лишь после множества долгих вечеров и выходных, проведенных в офисе. Мораль этой истории: паттерн MVC благотворно влияет не только на сопровождение вашего приложения, но и на равновесие между вашей работой и личной жизнью.

Типы MVC. Паттерн MVC существует в множестве разных форм. Две наиболее известные обычно называются тип I и тип II.

- **MVC тип I.** Этот тип представляет собой странично-ориентированный подход, при котором представление и контроллер существуют в виде одной сущности, именуемой «представление — контроллер». При этом подходе логика контроллера реализуется в представлении, таком как JSF. Все выполняемые контроллером задания, включая извлечение атрибутов и параметров HTTP-запроса, вызов бизнес-логики и управление HTTP-сеансом, встроены в представление с помощью скриптлетов и библиотек тегов. Тип I сильно связывает формиро-

вание представления с последовательностью выполняемых приложением действий, затрудняя тем самым сопровождение.

- **MVC тип II.** Проблемы с сопровождением в типе I преодолены в типе II благодаря вынесению логики контроллера из представления в сервлет, при этом визуализация данных остается представлению.

АЛЬТЕРНАТИВА MVC. ПОЗНАКОМЬТЕСЬ С MVP

Извините за разрушение иллюзий, но MVP не расшифровывается как «самый полезный паттерн» (most valuable pattern). MVP — аббревиатура, означающая паттерн «Модель — представление — презентатор» (Model — View — Presenter) — альтернативу паттерну «Модель — представление — контроллер». Вместо создания трехсторонних отношений между контроллером, представлением и моделью, подобно MVC, MVP предлагает единый способ связи между всеми сторонами — презентатор берет на себя управление обменом информацией между представлением и моделью. Он весьма популярен на платформах .NET и Silverlight, в наборе инструментальных средств веб-разработчика Google Web Toolkit и наборе библиотек Vaadin.

Главное различие между типом I и типом II — в местонахождении логики контроллера: в типе I она в представлении, а в типе II — в сервлете.

Многие фреймворки, такие как Spring MVC, Struts, Grails и Wicket, реализуют свою собственную версию паттерна MVC типа II. Например, Spring MVC включает концепцию сервлета-диспетчера, взаимодействующего с HTTP-запросами и выполняющего делегирование контроллеру, а также содержит представление (и преобразователь представления) и обработчики. Рисунок 14.2 демонстрирует диаграмму реализации паттерна MVC в Spring.

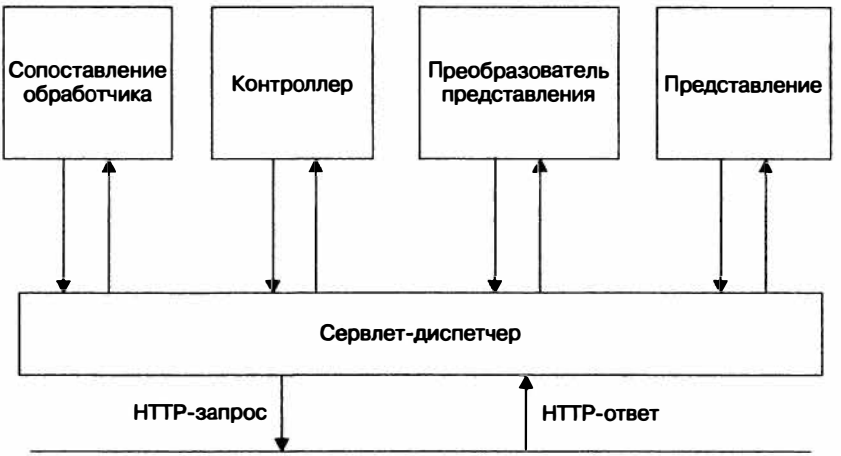


Рис. 14.2. Диаграмма реализации паттерна MVC в Spring

Реализация паттерна MVC в простом коде

Вы реализуете паттерн MVC с помощью паттерна «Действие». Он берет на себя ответственность за определение, куда переадресовывать пользователя на основании

пользовательского запроса. Это помогает обеспечивать персональную ответственность контроллера.

Вы начнете с класса контроллера. В листинге 14.1 вы реализуете простой контроллер, отвечающий на любой выполненный к пути /users/* HTTP-запрос GET. Отображение этого отношения описано в файле web.xml:

```
<servlet>
    <servlet-name>FrontController</servlet-name>
    <servlet-class>com.devchronicles.mvc.plain.FrontController</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>/users/*</url-pattern>
</servlet-mapping>
```

Здесь вы отображаете класс контроллера com.devchronicles.mvc.plain.FrontController на URL запроса /users/*. Следовательно, любой выполненный к этому URL запрос переадресуется классу FrontController для обработки.

SERVLETS 3.0

Как вариант, вы можете аннотировать класс контроллера с указанием URL запроса, вот так: @WebServlet({"/users/*"}). Эта аннотация избавляет от необходимости описывать отображение сервлета в web.xml.

Для подобных запросов вызывается метод doGet(), а из AbstractActionFactory извлекается объект Action, определяющий местонахождение *представления*, которое нужно будет вернуть пользователю.

Листинг 14.1. Переделанный класс UserService

```
package com.devchronicles.mvc.plain;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontController extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, IOException {

        Action action =
            AbstractActionFactory.getInstance().getAction(request);
        String view = action.execute(request, response);
        getServletContext().getRequestDispatcher(view).forward(request,
                                                                response);
    }
}
```

В листинге 14.2 имеется два класса: AbstractActionFactory и ActionFactory. Первый создает экземпляр второго. Метод getAction класса ActionFactory принимает объект HttpServletRequest, содержащий ссылку на URI требуемого расположения. Фабрика использует URI для определения, какой объект Action вернуть контроллеру. Вы поддерживаете карту соответствий URI путей запроса и объектов Action в действии Map. На основании URI пути запроса объект Action выбирается из карты и возвращается контроллеру.

Листинг 14.2. Класс Factory

```
package com.devchronicles.mvc.plain;

public class AbstractActionFactory {
    private final static ActionFactory instance = new ActionFactory();

    public static ActionFactory getInstance() {
        return instance;
    }
}

package com.devchronicles.mvc.plain;

import java.util.HashMap;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;

public class ActionFactory {

    private Map<String, Action> actions = new HashMap<>();
    private Action action;

    public ActionFactory() {
        actions.put("GET/users", new HomeAction());
        actions.put("GET/users/listusers", new ListUsersAction());
    }

    public synchronized Action getAction(HttpServletRequest request) {
        String path = request.getServletPath() + request.getPathInfo();
        String actionKey = request.getMethod() + path;
        System.out.println(actionKey);
        action = actions.get(actionKey);
        if(action == null){
            action = actions.get("GET/users");
        }

        return action;
    }
}
```

В объекте Action важно то, что конкретная реализация обеспечивает реализацию метода execute(). Этот метод выполняет часть бизнес-логики, требуемую

для формирования запрошенной пользователем страницы. Это может быть запрос к базе данных для получения информации, выполнение вычислений или формирование файла.

В листинге 14.3 метод `execute()` класса `ListUserAction` формирует список пользователей, добавляет его как атрибут в объект запроса. Затем он возвращает расположение *представления* для визуализации и отображает его пользователю. Страница `listuser.jsp` обращается к теперь хранящимся в объекте запроса данным и отображает их пользователю.

Для краткости был заполнен и возвращен объект `List`, но в реальном приложении именно здесь вы бы использовали EJB или другие объекты данных, осуществляющие соединение с базой данных.

Листинг 14.3. Класс Action

```
package com.devchronicles.mvc.plain;

import java.util.ArrayList;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ListUsersAction implements Action {
    public String execute(HttpServletRequest request,
                          HttpServletResponse response) {

        List<String> userList = new ArrayList<>();
        userList.add("Джон Леннон");
        userList.add("Ринго Старр");
        userList.add("Пол Маккартни");
        userList.add("Джордж Харрисон");
        request.setAttribute("listusers", userList);
        return "/WEB-INF/pages/listusers.jsp";
    }
}
```

Объект `Action` возвращается контроллеру, получающему адрес страницы, на который он должен переадресовать объекты запроса и ответа.

```
String view = action.execute(request, response);
getServletContext().getRequestDispatcher(view).forward(request, response);
```

В листинге 14.4 JSP обращается к переменной `requestScope` страницы и извлекает объект списка `userList`, созданный в `ListUserAction`. Затем он проходит по набору в цикле и отображает имена пользователей.

Листинг 14.4. Обращение к `listuser.jsp` формирует запрошенную пользователем страницу

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-
8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix ="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/
TR/html4/loose.dtd">
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>List of users</title>
</head>
<body>
<h1>Our users are:</h1>
<c:forEach items="${requestScope.listusers}" var="listusers">
    <br> ${listusers}
</c:forEach>
</body>
</html>
```

Продемонстрированный пример — простая реализация паттерна MVC. Теперь вам предстоит посмотреть, как можно реализовать то же самое приложение, но используя преимущества платформы Java EE.

Реализация паттерна MVC в Java EE

Реализация паттерна MVC в простом коде требует от вас написания логики контроллера, отображения URL на классы контроллера и написания немалого количества «канализационного» кода. Однако в последней версии платформы Java EE «канализационный» код был написан за вас. Вы можете сосредоточиться на представлении и модели. Заботы по реализации контроллера возьмет на себя FacesServlet.

FacesServlet

FacesServlet берет на себя управление запросами пользователей и «доставкой» представления пользователю. Он управляет жизненным циклом веб-приложений, применяющих JSF для формирования пользовательского интерфейса. Все запросы пользователей проходят через FacesServlet. Сервлет является неотъемлемой частью JSF и может быть сконфигурирован соответствующим образом, если понадобится не соответствующее соглашениям поведение. Однако — спасибо концепции соглашений по конфигурации — вы обнаружите, что все веб-приложения, кроме разве что самых сложных, не требуют изменения настроек по умолчанию.

НАСТРОЙКА FACESSERVLET

Если вам все же понадобится поменять конфигурацию FacesServlet, необходимо будет внести изменения в файл faces-config.xml.

Начиная с версии FacesServlet 2.2, вы можете выполнить наиболее распространенные настройки с помощью аннотаций, не трогая файл `faces-config.xml`.

MVC с использованием FacesServlet

Сейчас вы перепишете предыдущий пример с использованием сервлета FacesServlet и JSF. Язык описания представлений для JSF называется Facelets. Он пришел на замену JSP и написан на XHTML с применением каскадных таблиц стилей (Cascading Style Sheets, CSS).

JSF включает концепцию управляемых компонентов, представляющих собой простые Java-объекты в старом стиле, аннотированные `@Named` и `@RequestScope`. Эти компоненты доступны через JSF-страницу на протяжении выполнения HTTP-запроса. Вы можете обращаться к их методам непосредственно в JSF. В листинге 14.5 вы перепишете класс `ListUsersAction`, чтобы превратить его в управляемый компонент.

Листинг 14.5. Класс `ListUserAction`, переписанный в виде управляемого компонента

```
package com.devchronicles.mvc.javaee;

import java.util.ArrayList;
import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@RequestScoped
@Named
public class ListUsersAction {

    private List<String> userList = new ArrayList<>();

    public List<String> getUserList() {
        return userList;
    }

    public String execute() {
        userList.add("Джон Леннон");
        userList.add("Ринго Старр");
        userList.add("Пол Маккартни");
        userList.add("Джордж Харрисон");
        return "/WEB-INF/pages/listusers.xhtml";
    }
}
```

Поскольку все управляемые компоненты аннотированы по крайней мере `@Named` и `@RequestScope`, существует аннотация стереотипа, позволяющая при своем применении придать классу все поведенческие характеристики управляемого компонента. Эта аннотация — `@Model`.

Далее вам нужно создать файл `index.xhtml`. Он замещает `home.jsp`, и обращение к нему выполняется непосредственно из браузера. Задача этого JSF — вызвать метод `execute` класса `ListUsersAction`, который подготовит данные для представления `listusers.xhtml`.

Листинг 14.6 демонстрирует вызов этого метода.

Листинг 14.6. Главная веб-страница простого примера MVC

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/
html">

<h:head><title>Добро пожаловать!</title></h:head>

<h:body>
<h1>Добро пожаловать на наш сайт!</h1>
<h:form>
    <h2>Нажмите, чтобы увидеть <h:commandLink value="список пользователей"
        action="#{listUsersAction.execute}"/>.</h2>
</h:form>
</h:body>
</html>
```

Вы используете тег `h:commandLink` для ссылки на управляемый компонент, а также метод `execute()` в элементе `action`. Обращение к методу `execute()` происходит непосредственно из JSF; он формирует список пользователей, возвращает местоположение представления, которое будет визуализировать список, а затем вызывает метод `getUserList()` и отображает список пользователей. Листинг 14.7 демонстрирует это.

Листинг 14.7. Представление, визуализирующее данные модели

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html">

<head>
<title>Список пользователей</title>
</head>
<body>
    <h1>Наши пользователи:</h1>
    <ui:repeat value="#{listUsersAction.userList}" var="listusers">
        <h:outputText value="#{listusers}" />
        <br/>
    </ui:repeat>
</body>
</html>
```

В управляемом компоненте на класс действия ссылаются как на `listUsersAction`, с первой буквой в нижнем регистре, а в названии метода `getUserList()` опущено `get`. Если название метода начинается с `get`, то вы можете это `get` опустить.

При развертывании приложения представление index.xhtml сформирует ссылку, которая, если на ней щелкнуть, отобразит список пользователей, вот так:

```
Наши пользователи:
Джон Леннон
Ринго Старр
Пол Маккартни
Джордж Харрисон
```

Вы только что успешно создали сайт в стиле MVC, используя новейшие возможности платформы Java EE 7.

Где и когда использовать паттерн MVC

Наилучшие результаты дает использование паттерна MVC в веб-приложениях, хотя вы можете использовать его везде, где может быть полезным разделить логику представления и бизнес-логику. На самом деле паттерн MVC применяется в веб-приложениях настолько повсеместно, что любое предложение *не* использовать его будет встречено с презрением и насмешкой.

Нет сомнений, что два основных преимущества использования паттерна MVC весьма серьезны. Его разделение обязанностей способствует созданию гибких и легко адаптируемых веб-приложений, а его разделение производства позволяет разрабатывать различные части приложения практически независимо друг от друга. Например, одна команда может работать над отображающей логикой, а другая — над бизнес-логикой и объектами предметной области.

Резюме

У паттерна MVC есть множество толкователей со своими точками зрения на его использование, реализацию и даже пригодность. Вы прочитали мою интерпретацию этого паттерна и посмотрели на его реализации. Увидеть настоящие его преимущества может быть непросто.

Рекомендуем вам не забывать, в чем сущность паттерна MVC: разделение логики представления с бизнес-логикой. Если вы написали код, удовлетворяющий этому требованию, значит, вы успешно реализовали паттерн MVC.

Основной замысел отделения логики представления от бизнес-логики состоит в обеспечении чистого разделения объектов предметной области, моделирующих вашу задачу, и представления этой логики. Разделение дает бизнес-данным возможность быть представленными любым количеством различных способов, одновременно не требуя, чтобы объект предметной области «знал» что-либо о способе, которым осуществляется его представление. Он может быть отображен на экране в множестве форматов, в том числе как файл Word или Excel.

Упражнение

Расширьте приведенный в этой главе пример, добавив различные представления для отображения списка пользователей.

15 Другие паттерны в Java EE

В этой главе:

- веб-сокеты;
- программное обеспечение промежуточного уровня, ориентированное на обработку сообщений;
- микросервисы и монолиты.

ЗАГРУЗКА ИСХОДНОГО КОДА С WROX.COM ДЛЯ ЭТОЙ ГЛАВЫ

Страница загрузки исходного кода для этой главы находится на вкладке Download Code по адресу www.wrox.com/go/projavaeedesignpatterns. Фрагменты исходного кода содержатся в файле Chapter 15.zip, каждый из них назван в соответствии с наименованиями в тексте.

В этой главе обсуждаются некоторые преимущества платформы Java EE и разработки под нее. Вы можете рассматривать ее как сборник всего того, что необходимо знать, но что не подходит для других глав.

Эта глава познакомит вас с веб-сокетами — восхитительной новой возможностью платформы Java EE. Затем вы узнаете про ориентированное на обработку сообщений программное обеспечение промежуточного уровня, а после перейдете к близкой теме архитектуры микросервисов.

Насладитесь этим эклектичным набором технологических деликатесов!

Что такое веб-сокеты

Веб-сокеты (WebSockets), возможно, самое интересное нововведение в веб-технологиях со времен появления «Асинхронного JavaScript и XML» (AJAX). Они стали популярными с выходом HTML5 и поддерживаются множеством веб-фреймворков. Однако потребовалось немало времени для создания стабильной и совместимой спецификации веб-сокетов.

Модель протокола передачи гипертекста (HTTP) была спроектирована задолго до того, как стал популярен Интернет, она основывается на простых спецификации и дизайне. В традиционной модели HTTP клиент открывает соединение

с сервером прикладной части, отправляет HTTP-запрос типа[1] GET, POST, PUT или DELETE, а HTTP-сервер возвращает соответствующий ответ.

Традиционная модель HTTP была обременительной практически для любого приложения, выходящего за пределы простой модели данных «получить и отправить контент». Представьте себе клиент приложения для интерактивной переписки, в котором участники могут отправлять сообщения в любом порядке и сотни участников могут общаться одновременно. Для подобных целей стандартный подход «запрос-ответ» налагает слишком сильные ограничения. Первыми попытками обойти эти ограничения стали AJAX и Comet. Оба были основаны на так называемых длинных опросах: открытии HTTP-соединения и поддержании его в активном состоянии (сохранении соединения открытым) посредством незавершения отправки ответа.

С помощью веб-сокетов клиент может создать «сырой» сокет для сервера и осуществлять полнодуплексную[2] связь. Поддержка веб-сокетов была введена в JSR-356. Пакет javax.websocket и его серверный подпакет содержат все относящиеся к веб-сокетам классы, интерфейсы и аннотации.

Чтобы реализовать веб-сокеты на платформе Java EE, вам необходимо создать класс конечной точки с методами жизненного цикла веб-сокета, как показано в листинге 15.1.

Листинг 15.1. Пример конечной точки

```
package com.devchronicles.websockets;
public class HelloEndpoint extends Endpoint {

    @Override
    public void onOpen(final Session session, EndpointConfig config) {

        session.addMessageHandler(new MessageHandler.Whole<String>() {

            @Override
            public void onMessage(String msg) {
                try {
                    session.getBasicRemote().sendText("Hello " + msg);
                } catch (IOException e) { }
            }
        });
    }
}
```

Класс Endpoint предоставляет три метода жизненного цикла: onOpen, onClose и onError. Расширяющий его класс должен реализовать как минимум метод onOpen.

Вы можете развернуть конечную точку двумя способами: с помощью конфигурации или программными средствами.

Чтобы выполнить развертывание кода на листинге 15.1 программно, ваше приложение должно вызвать следующее:

```
ServerEndpointConfig.Builder.create(HelloEndpoint.class, "/hello").build();
```

---

[1] Полный список HTTP-методов: GET, POST, DELETE, PUT, PATCH, OPTION, HEAD, TRACE и CONNECT.

[2] С одновременным приемом и передачей данных. — *Примеч. пер.*

Развернутый веб-сокет доступен на ws://<host>:<port>/<application>/hello. Однако лучше использовать конфигурацию с помощью аннотации. При этом та же конечная точка становится кодом из листинга 15.2.

Листинг 15.2. Пример конечной точки с аннотациями

```
package com.devchronicles.websockets;

@ServerEndpoint("/hello")
public class HelloEndpoint {

    @OnMessage
    public void onMessage(Session session, String msg) {
        try {
            session.getBasicRemote().sendText("Hello " + msg);
        } catch (IOException e) { }
    }
}
```

Такой подход позволяет вам использовать аннотации, придерживаясь подхода простых Java-объектов в старом стиле (POJO), поскольку вы не расширяете базовый класс. У аннотированной конечной точки те же методы жизненного цикла, что и в листинге 15.1, но она вводит дополнительный метод жизненного цикла onMessage. Вместо реализации OnOpen и добавления обработчика onMessage в основанном на аннотациях подходе достаточно реализовать аннотированный метод жизненного цикла onMessage. Вы можете аннотировать с помощью @OnMessage несколько методов, чтобы получать различные типы данных, такие как String или ByteBuffer для двоичных данных.

Реализация веб-сокета со стороны клиента зависит от используемого веб-фреймворка. Как бы то ни было, в следующем фрагменте показана простая версия на языке JavaScript:

```
var webSocket = new WebSocket('ws://127.0.0.1:8080/websockets/hello');
webSocket.send("world");
```

Лучшим примером будет отправка сложного объекта в формате нотации объектов JavaScript (JSON), который может быть организован в объект так, как показано в следующем фрагменте:

```
var msg = {
    type: "message",
    text: "World",
    date: Date.now()
};

webSocket.send(JSON.stringify(msg));

webSocket.onmessage = function(evt) { /* Должен получить hello world */ };
```

Веб-сокеты отлично подходят для создания веб-приложений, требующих асинхронного обмена сохраняемыми сообщениями между клиентом и сервером. Платформа Java EE предоставляет удобную реализацию веб-сокетов. Конфигурационных

настроек и вариантов реализации у них гораздо больше, чем обсуждается здесь. Если мы пробудили у вас интерес к веб-сокетам, рекомендуем заглянуть в руководство Oracle по платформе Java EE[1], детальнее рассказывающее о программировании веб-сокетов с помощью Java API.

Что такое ориентированное на обработку сообщений ПО промежуточного уровня

Связь между компонентами в системе Java EE синхронна. Цепочка вызовов начинается с обращения EJB-компонента к объекту доступа к данным (DAO), затем к сущности, и так далее до конечной цели. Все компоненты цепочки вызовов должны быть доступны и готовы к получению вызова, а вызывающий компонент обязан ждать ответа перед продолжением выполнения. Успех вызова зависит от доступности всех компонентов. Как вы видели в главе 9, вызов асинхронного метода не требует от вызывающего объекта ожидания ответа. Обычная последовательность выполнения может продолжаться, в то время как асинхронный метод создает свою собственную цепочку вызовов.

Ориентированное на обработку сообщений программное обеспечение промежуточного уровня (Message-Oriented Middleware, MOM) представляет собой своеобразный буфер между системами, позволяющий приостанавливать обмен сообщениями, если компонент не готов к работе. Сообщения доставляются, как только все компоненты становятся доступными. Вызовы транслируются в сообщения и отправляются через систему передачи сообщений целевому компоненту, который обрабатывает сообщение и, возможно, отвечает. Если целевой компонент недоступен, то сообщения образуют очередь, ожидая, когда система станет доступной. Когда компонент становится доступным, сообщения обрабатываются.

На одном конце цепочки производитель данных, транслирующий вызов в такую форму, которая может быть передана в качестве сообщения, а на другом конце — потребитель данных, получающий сообщение. Потребители и производители данных сильно расцеплены, поскольку они «не знают» ничего друг о друге. Они даже не обязательно должны быть написаны на одном языке программирования или располагаться в одной сети; они могут быть распределены по нескольким внешним серверам.

Систему MOM составляют четыре типа участников: сообщения, потребители данных, производители данных и посредники. Производители данных формируют сообщения и отправляют их посредникам, которые затем распространяют эти сообщения по местам назначения, где они хранятся, пока не подключится потребитель данных и не обработает их.

Существуют две архитектурные реализации MOM: «точка-точка» и «издатель/подписчик».

---

[1] Руководство Oracle по WebSocket API: http://docs.oracle.com/javaee/7/tutorial/doc/websocket.htm.

В реализации «точка-точка» производитель данных отправляет сообщение по месту назначения, которое называется *очередь*. В очереди сообщение ожидает, пока потребитель данных не извлечет его и не подтвердит, что оно было успешно обработано. Если это происходит, сообщение удаляется из очереди. Рисунок 15.1 показывает производителя данных, помещающего сообщение M1 в очередь, и далее потребителя данных, извлекающего сообщение из очереди и обрабатывающего его. В этой реализации сообщение обрабатывается только одним потребителем данных.

Рис. 15.1. Реализация «точка-точка»

В реализации «издатель/подписчик» место назначения называется *тема*. Производитель «публикует» сообщение в теме, и все потребители, «подписанные» на нее, получают копию сообщения. Это показано на рис. 15.2, где сообщения M1 и M2 публикуются в теме T1 и получаются потребителями C1 и C2, а сообщение M3 публикуется в теме T2 и получается потребителями C2 и C3.

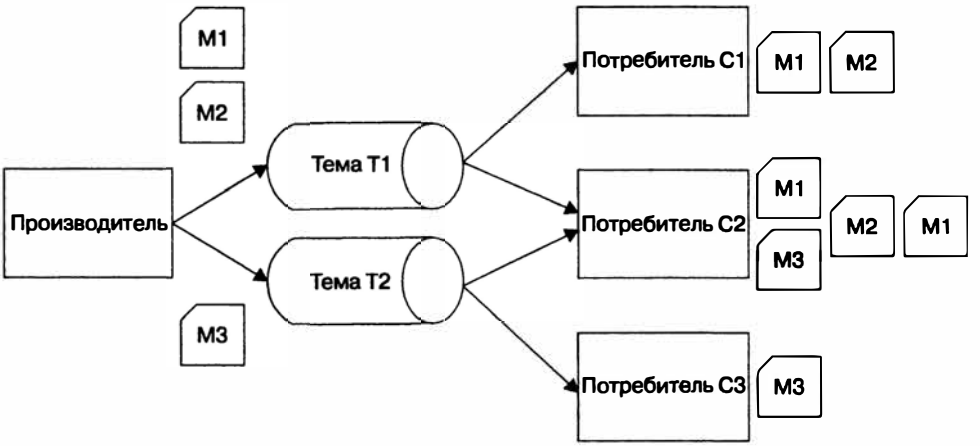


Рис. 15.2. Реализация «издатель/подписчик»

Платформа Java EE предоставляет удобный интерфейс программирования приложений для работы с этими реализациями, который называется интерфейсом передачи сообщений Java (Java Message Service, JMS). Это набор интерфейсов, описывающих создание сообщений, посредников и потребителей данных. При реализации в EJB-контейнере управляемые сообщениями компоненты (MDB) функционируют в качестве асинхронно вызываемых прослушивателей для сообщений JMS.

Что такое архитектура микросервисов

За последние несколько лет паттерн «Архитектура микросервисов» стал очень популярен. Его идея состоит в проектировании большого распределенного масштабируемого приложения, состоящего из маленьких связанных воедино сервисов, способных развиваться или даже быть полностью переписанными за время жизни приложения.

По своей идее он схож с уже давно используемым паттерном «Сервис-ориентированная архитектура» (SOA). Сущность идеи микросервисов в том, что каждый сервис должен быть маленьким — возможно, маленьким вплоть до размеров всего в несколько сотен строк кода. Цель — разбить большое монолитное приложение на гораздо меньшие приложения ради решения задач его разработки и развития.

В этой главе обсуждаются возможные причины для следования по пути микросервисов, их недостатки и достоинства, а также они сравниваются с более известной и привычной монолитной архитектурой.

Монолитная архитектура

Наиболее распространенным способом разработки и формирования пакетов веб-приложения всегда была сборка всех ресурсов, компонентов и файлов классов в единый файл архива веб-приложения (Web application ARchive, WAR) или корпоративного архива (Enterprise ARchive, EAR) с последующим его развертыванием на веб-сервере. Типовое приложение для книжного магазина может включать компоненты, управляющие учетными записями пользователей и обработкой оплат, контролирующие уровень запасов, администрирующие сервис по работе с покупателями и формирующие представления клиентской части. Все это разрабатывается в едином монолитном приложении и устанавливается на веб-сервер. Рисунок 15.3 демонстрирует упрощенную схему монолитного приложения.

Компоненты объединяются в пакет в логической модулярной форме и устанавливаются как одно единое монолитное приложение. Это простой способ разрабатывать и устанавливать приложения, поскольку тестировать приходится только одно приложение. IDE и другие инструменты разработчика проектируются с учетом монолитной архитектуры. Несмотря на эти преимущества монолитной архитектуры, приложения, построенные подобным образом, часто очень велики.

Разработчикам проще иметь дело с маленькими приложениями, разбираться в них и сопровождать их, а большие монолитные приложения могут быть сложны для понимания, особенно для недавно присоединившихся к команде разработчиков. Им потребуются недели или даже месяцы, чтобы полностью разобраться в приложении.

Частое выполнение установки непрактично, поскольку требует согласования действий многих разработчиков (а возможно, и других отделов). Организация установки может занять часы или дни, задерживая тестирование новых возможностей и исправление ошибок. Существенный недостаток монолитного проектирования — сложность смены технологии или фреймворка. Приложение уже разработано на основе технологических решений, принятых в самом начале реализации

проекта. Вы привязаны к этим решениям; и если даже найдется технология, решающая задачу более изящным или производительным способом, будет нелегко начать ее использовать. Не всегда приемлемо переписывать целое приложение. Паттерн «Монолитная архитектура» плохо подходит для хорошей масштабируемости.

Рис. 15.3. Монолитная архитектура

Масштабируемость

Масштабируемость означает способность приложения расти (и уменьшаться) в соответствии с требуемыми изменениями его сервисов, без заметного влияния на удобство его использования. Плохо функционирующий сайт электронной коммерции быстро теряет покупателей, что делает масштабируемость очень существенной. Первое напрашивающееся решение — масштабировать горизонтально, создав копии приложения на многих серверах и распределив нагрузку трафика с ориентацией на высокую доступность (High Availability, HA), с переходом пассивного узла в активное состояние при отказе активного узла. Масштабирование по оси *X* повышает пропускную способность и доступность приложения. Такой вариант не сказывается на стоимости разработки, но повышает расходы на хостинг и сопровождение[1].

[1] Это может привести к затратам на разработку, если система использует данные сеансов HTTP и не представляет собой созданное для репликации сеансов HTTP кластеризованное решение.

Вы можете масштабировать приложение по оси *Z*. Код приложения дублируется на нескольких серверах, подобно разбиению по оси *X*, но в данном случае каждый сервер отвечает только за часть данных. Создается механизм маршрутизации данных к нужному серверу, возможно, на основе типа пользователя или первичного ключа. Масштабирование по оси *Z* дает практически те же преимущества в производительности, что и масштабирование по оси *X*; однако оно предполагает дополнительные затраты на рефакторинг архитектуры приложения.

Ни одно из этих решений не устраняет повышения сложности приложения и его разработки. Для этого вам потребуется вертикальное масштабирование.

Приложение может масштабироваться вдоль оси *Y*. При этом оно подвергается декомпозиции по функциональности, сервисам или ресурсам. Каким способом вы это сделаете — целиком ваш выбор, который будет зависеть от обстоятельств, хотя общепринято разбивать по сценариям использования. Идея в том, чтобы каждая часть инкапсулировала небольшой набор связанных функций.

Чтобы визуализировать масштабирование по осям *X*, *Y* и *Z*, нарисуем куб масштаба AKF[1], как показано на рис. 15.4.

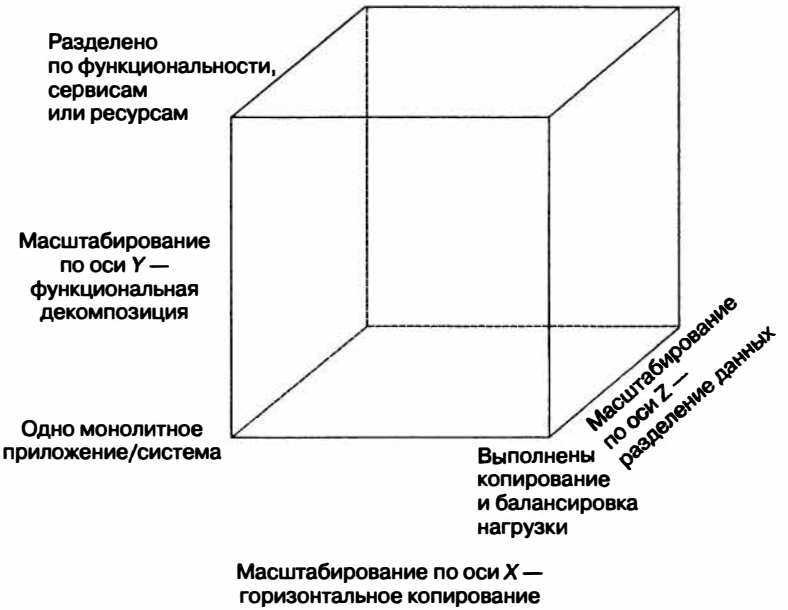


Рис. 15.4. В кубе AKF должно быть масштабирование по осям X, Y и Z

Декомпозиция на сервисы

При применении метода микросервисов монолитное приложение разбивается вдоль оси *Y* на сервисы, соответствующие одному сценарию использования или набору

[1] *Abbott Martin L., Fisher Michael T.* The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. — January 1, 2009.

связанной функциональности. Эти сервисы затем копируются на несколько серверов и размещаются за балансировщиком нагрузки (разделение по оси *X*). Хранение данных может масштабироваться вдоль оси *Z* посредством дробления данных на основе первичного ключа.

Если вы декомпозируете приложение на рис. 15.3 вдоль оси *Y*, то придете к показанной на рис. 15.5 архитектуре.

Рис. 15.5. Декомпозиция по оси Y

Представления клиентской части были разбиты на отдельные приложения, которые обращаются к функциональности нескольких сервисов прикладной части. Сервисы были выделены из монолитного приложения в автономные приложения, управляющие своими собственными данными. Выполнить разбиение вдоль оси *Z* можно путем дробления данных, а масштабирование вдоль оси *X* — с помощью кластеризации и выравнивания нагрузки.

Вы увидели, как выполняется декомпозиция на микросервисы монолитного приложения, и поняли важность масштабируемости для продолжения функционирования приложения.

Теперь вы посмотрите внимательнее на конкретные выгоды и издержки архитектуры микросервисов.

Выгоды микросервисов

С точки зрения разработки выгоды архитектуры микросервисов — это результаты размера и скорости работы составляющих ее маленьких приложений. Они легки в понимании для разработчиков и в управлении — для IDE. Большое приложение,

состоящее из многих сотен модулей, может требовать значительного времени для загрузки, что негативно влияет на производительность разработчиков. Приложение каждого микросервиса может быть развернуто быстрее и часто без согласования с другими командами. Поскольку каждый сервис автономен, локальные изменения в коде не влияют на другие микросервисы; следовательно, возможна непрерывная разработка. Каждый микросервис разрабатывается закрепленной за ним командой разработчиков, управляющих развертыванием и требованиями к ресурсам независимо от других команд.

Пользовательский интерфейс обычно отделен от разработки прикладной части. Ваша команда разработчиков может никогда не видеть ни одной строки кода UI. Если вы программируете API передачи состояния представления (REST) (см. главу 13), вам необходимо только соблюдать соглашения по предоставлению ресурсов, а не думать о способе реализации клиентской части. Это дает возможность по-настоящему разделить различные виды функциональности.

С точки зрения производительности приложения, главное преимущество состоит в возможности разворачивать каждый микросервис в его собственной, специально приспособленной среде. Требования к ресурсам вашего микросервиса могут отличаться от требований других, позволяя, таким образом, «дробное» выделение ресурсов. Монолитное приложение развертывается в единой среде с совместно используемыми ресурсами.

Вырастают отказоустойчивость и степень локализации неисправностей. Сбой в одном микросервисе не влияет на работоспособность остальных, позволяя приложению продолжать работу. В системе, использующей MOM для обмена информацией между сервисами, сообщения, предназначенные сбойному микросервису, ждут в очереди, пока ошибка не будет устранена, а микросервис (потребитель данных) не начнет получать сообщения. При горизонтальном масштабировании приложения обслуживание не прерывается, поскольку сообщения получает один из дублирующих микросервисов. В монолитном приложении подобный сбой привел бы к отказу всего приложения.

Среди всех преимуществ, свойственных архитектуре микросервисов, наиболее обсуждаемое — легкость, с которой можно изменять технологический стек. Поскольку каждый микросервис невелик, его несложно переписать. В сущности, новые микросервисы могут быть написаны на любом языке программирования, что позволяет выбирать наиболее подходящий для решения задачи язык. Технологические решения, принятые в начале реализации проекта, не указывают вам, какие технологии вы обязаны использовать на протяжении всего существования приложения.

Ничто в жизни не бывает бесплатно

Выгоды архитектуры микросервисов не даются бесплатно. С ними связаны определенные затраты.

Легкость, с которой можно разработать новый микросервис, приводит к столь же легкому росту их количества, причем росту очень быстрому. Пятнадцать микросервисов могут легко стать тридцатью и более, особенно если считать

различные версии одного микросервиса. Это приводит к некоторым сложностям.

Ответственность за функционирование переходит к команде разработчиков. При наличии всего лишь небольшого количества сопровождаемых сервисов это несложно, но по мере роста количества микросервисов работы по их сопровождению становится больше. Чтобы гарантировать, что все микросервисы развернуты и сопровождаются, требуются существенные капиталовложения. Процессы должны быть автоматизированы, чтобы можно было снизить накладные расходы по развертыванию и сопровождению большого количества сервисов. К общим эксплуатационным расходам может добавиться требующий заполнения пробел в знаниях.

Сквозные изменения в семантике означают, что все микросервисы обязаны обновлять свой код, чтобы работать согласованно. Это может отнимать много времени и означать существенные расходы на повторное тестирование. Причиной таких изменений, заставляющих все команды работать согласованно, могут быть изменения в соглашениях по интерфейсам и форматам сообщений. В равной мере неудачная попытка изменения интерфейса или формата сообщения на раннем этапе проекта может привести к существенному росту затрат по мере увеличения количества микросервисов.

Вас, вероятно, учили, что дублирование кода — нечто плохое, и это так и есть. В среде микросервисов риск дублирования кода весьма велик. Чтобы избежать сцепления и зависимостей, код иногда приходится дублировать, а значит, каждый его экземпляр должен быть протестирован и сопровождаем. Возможно, вам удастся выделить код в библиотеку общего пользования, но это не будет работать в многоязыковой среде.

Свойственные распределенным системам сложность и ненадежность в точности повторяются в среде микросервисов. Каждый сервис может размещаться распределенным способом, обмениваясь информацией через сети, страдающие из-за проблем с задержками, несовместимости версий, ненадежности провайдеров, проблем с оборудованием и т. п. Жизненно необходим постоянный мониторинг производительности сети.

Выводы

Монолитная архитектура многие годы использовалась для разработки приложений и верно служила маленьким приложениям и командам разработчиков. Она проста в разработке и тестировании, поскольку IDE проектируются для управления приложениями с подобным типом структуры. Но, как вы видели, она немасштабируема и затрудняет разработку. Рефакторинг оказывается дорогостоящим, а внедрение новых технологий — сложным.

Микросервисы представляют собой декомпозицию на логические сервисы, реализующие связанную функциональность. Их маленький размер облегчает понимание разработчиками. Разработка и развертывание непрерывны. Масштабируемость — это часть архитектуры, и вы не связаны с первоначальными технологическими решениями.

Наконец, несколько антипаттернов

Цель этой книги — заполнить брешь между «классическими» паттернами и платформой Java EE. Вы можете найти множество книг, обсуждающих антипаттерны, но не повредит обсудить некоторые из них здесь.

Антипаттерны обычно являются следствием неправильного использования одного или нескольких паттернов. Разработчик Java EE с достаточным опытом легко перечислит больше антипаттернов, чем паттернов. Приведем список нескольких наиболее распространенных (или таких, с которыми вы могли уже сталкиваться).

Сверхкласс

Наверное, нет ни одного проекта без огромного класса, служащего сразу многим целям и имеющего много обязанностей. Это не только нарушает принципы платформы Java EE: такие классы попирают базовые принципы объектно-ориентированного программирования, и их нужно избегать.

Сервисы, перегруженные многими обязанностями, относятся к той же категории. Если вы небольшой приверженец объектно-ориентированного программирования — ничего страшного, но если вы хотите продолжать писать код на объектно-ориентированном языке, то лучше, чтобы ваши классы были небольшими и сильно связанными.

Хотя об этом антипаттерне высказывались многие другие, первым дал ему название Реза Рахман.

Лазанья-архитектура

Платформа Java EE с первых дней поощряла использование слоев, что, вероятно, привело ко многим ненужным интерфейсам и пакетам. Хотя этот паттерн может показаться ответом на сверхкласс и монолитные приложения, обычно он излишне усложняет положение дел.

Лазанья-архитектура в мире ООП не сильно отличается от спагетти-программирования в структурном программировании. Слишком большое количество абстракций не нужно и ничуть не помогает. Интерфейсы и слабая связь — отличные средства только тогда, когда вы используете их в нужном количестве, нужном контексте и только при необходимости.

Этот антипаттерн упоминался многими разработчиками под множеством различных имен, например как «пахлава-код». Однако имя «лазанья» было впервые дано Адамом Бьеном[1], активно возражающим против излишнего использования интерфейсов.

Господин Колумб

Почти все опытные разработчики приложений для платформы Java EE хотели бы изобрести или реализовать свое собственное идеальное решение. Почти всегда это

[1] *Бьен Адам.* Автор и Java-чемпион. — www.adam-bien.com.

просто попытки абстрагировать и предоставить лучший интерфейс для распространенной библиотеки, например журналирования или тестирования, вплоть до впадания в крайности и переписывания важной функциональности, поддерживаемой сообществом свободного программного обеспечения годами (как, например, слой объектно-реляционного отображения (ORM)).

Хотя изобретение чего-то нового может быть интересно, изобретать что-то заново — просто потеря времени. Если вы собираетесь написать новый фреймворк журналирования или объектно-реляционного отображения, то у вас должна быть действительно веская причина делать это. Если же ее нет, то вы просто переписываете хорошо поддерживаемый готовый продукт и, вероятнее всего, закончите тем, что будете сами его сопровождать, обеспечивать всю поддержку, тестирование и дальнейшую разработку.

Всегда старайтесь убедиться, что вы провели достаточно тщательный информационный поиск по проектам с открытым исходным кодом, перед тем как начинать писать фреймворк с нуля.

Друзья с привилегиями

Огромной проблемой J2EE является привязка к поставщикам. Ко времени выхода J2EE 1.4 серверы большинства поставщиков работали только с инструментами и IDE соответствующего поставщика. Это на первый взгляд выглядело взаимовыгодными отношениями, так как поставщик обеспечивал профессиональную поддержку его собственных инструментов, а поддержка инструментов с открытым исходным кодом была оставлена сообществу открытого ПО. Однако в долгосрочной перспективе многие разработчики J2EE обратили внимание, что инструменты с открытым исходным кодом обеспечивают стандартизованное поведение и совместимость со спецификациями языка Java, а поставщикам это не удавалось.

В покупке профессиональной поддержки и услуг, а также покупке инструментов, серверов и IDE у поставщиков нет ничего плохого, до тех пор пока это не связано с привязкой проекта к конкретному поставщику. Привязка к поставщику может привести к проблемам, которые невозможно решить без новых версий и «заплат», тогда как создание приложений всегда оставляет вероятность, что поставщика можно будет сменить.

Дорогостоящие технологические новинки

Увлеченные разработчики любят использовать дорогостоящие технологические новинки. Например, веб-сокеты появились много лет назад, но они все еще страдают от проблем совместимости со старыми версиями браузеров. Никто не против испытать радость при изучении чего-то нового и реализации в проекте дорогостоящих технологических новинок. Однако поддержка такого проекта может оказаться обременительной, если вы ориентированы на серийный выпуск.

Прежде чем решить, какой фреймворк или технологию применять, хорошо посмотреть, вписываются ли они в вашу целевую пользовательскую базу. Если вы создаете банковское приложение для клиентов, которые до сих пор могут использовать

Internet Explorer 6, лучше всего применять веб-сокеты (хотя большинство фреймворков веб-сокетов поддерживают сценарии автоматического восстановления).

Необходимо убедиться, что внешняя библиотека или фреймворк хорошо поддерживаются, достаточно «зрелые» и подходят для вашего проекта, прежде чем начинать их использовать.

«Мастер на все руки»

Утилитные классы и пакеты — обычное дело в проектах. Никто не станет спорить, что может понадобиться класс для выполнения математических действий вроде округления или преобразования различных числовых типов. Если у вас действительно есть такие утилитные или вспомогательные классы, вероятно, вы захотите объединить их в пакете с именем `util` или `helper`, не правда ли? В действительности это лишь поможет вам собирать мусор. Поскольку `util` и `helper` называются слишком обобщенно, в эти пакеты будут перемещаться очень многие классы. Любой класс, который не удается легко отнести к какой-то категории, в итоге окажется в таком пакете. Обобщенное название не дает реальной информации, так что даже если какие-то классы уже не используются, никто не осмелится их удалить.

Если у вас есть замечательная, нужная всем утилита, просто поместите ее в место, соответствующее ее текущему использованию, и предоставьте требуемую документацию. При необходимости в будущем вы сможете переместить ее в какой-то более общий пакет.

Впервые, как и лазанью, этот паттерн описал Адам Бьен. Он же дал ему название.

ЧАСТЬ III

Подведем итоги

16 Паттерны проектирования: хорошие, плохие, ужасные

В этой главе:

- хорошие: как паттерны проектирования могут привести к успеху;
- плохие: как излишнее и неправильное использование паттернов проектирования может привести к проблемам;
- ужасные: как некоторые «неофициальные» стандарты могут привести к краху.

До сих пор эта книга охватывала многие «классические» паттерны проектирования из книги GoF, а также кое-какие дополнительные паттерны, которые могут стать «классикой» в будущем.

Как справедливо относительно всего на свете, паттерны проектирования не всегда приносят пользу. Они могут также и причинять вред, склоняя вас к реализации антипаттернов. Эта глава сосредотачивается на хороших, плохих и «злых» аспектах паттернов проектирования и обеспечивает, надеемся, наилучшие методы для вашей тяжелой артиллерии паттернов.

Хороший: паттерны для успеха

Как уже многократно упоминалось, паттерны проектирования представляют собой собранную воедино мудрость и опыт многих толковых программистов. Вы можете использовать их опыт для решения многих распространенных задач, встречающихся в разработке программного обеспечения. Уже в первые дни возникновения программирования, когда использование `goto` еще считалось разрешенным (и допустимым), многие проекты терпели крах. Одним из ранних важных источников по инженерии разработки ПО была книга *The Mythical Man-Month*, написанная Фредериком Бруксом, когда он руководил разработкой OS360 в компании IBM[1]. Хотя эта книга была опубликована в 1975 году, она до сих пор охватывает

[1] *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. – М.: Символ-Плюс, 2001. – 304 с.

многие проблемы современных программных проектов. В то время становился популярен один из первых паттернов проектирования в программном обеспечении — объектно-ориентированное программирование (ООП). ООП было набором правил проектирования и шаблонов, позволявших проще и эффективнее моделировать ситуации из реальной жизни в коде. Оно стало настоящей волшебной палочкой для проектирования, программирования и сопровождения программного обеспечения. Первопроходцами первых золотых лет ООП были языки Smalltalk, C++ и Objective-C. Хотя Эдсгер Дейкстра[1] заметил, что «объектно-ориентированное программирование — исключительно плохая идея, придумать которую могли только в Калифорнии», это была «первая ласточка», поменявшая стиль написания программ.

Однако объектно-ориентированное программирование тоже не было «серебряной пулей». Во-первых, использование объектно-ориентированных языков не означало на самом деле использования объектно-ориентированного подхода. Разработчикам позволялось (и до сих пор позволяется) писать процедурный код на любом объектно-ориентированном языке программирования. Во-вторых, применение сложных и плохо спроектированных объектов может привести к проблемам ничуть не меньшим, чем любая не-ООП-система.

В начале 1990-х годов знаменитая «Банда четырех» — Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес — опубликовала *Design Patterns: Elements of Reusable Object-Oriented Software*. Это была первая книга, в которой в качестве паттернов проектирования был собран набор решений для распространенных задач. Издание охватывает 23 паттерна проектирования, которые в нашей книге мы называли классическими, и включает примеры кода на языках C++ и Smalltalk. За прошедшие годы выдающиеся программисты, такие как Джим Коплин[2], изобрели и добавили в каталоги немало новых паттернов.

Как бы ни было, паттерны проектирования не зависят от платформы и языка программирования, так что могут быть реализованы в любом программном проекте. Паттерны проектирования решают общие задачи и предоставляют разработчикам общий словарь для общения. Вы можете сказать просто: «У нас наблюдатели на X.» — вместо того, чтобы описывать вашу реализацию механизма обратного вызова, срабатывающую только при изменении ресурса.

Когда в середине 1990-х был разработан язык Java, в его среду выполнения были интегрированы многие паттерны проектирования. Java удачно пользуется паттернами проектирования и делает многие из них доступными для использования в языке, предоставляя реализацию по умолчанию для интерфейсов программирования приложений (API).

С выходом платформы Java EE появилось еще больше паттернов, многие из которых были описаны в книге *Core J2EE Patterns: Best Practices and Design Strategies*.

---

[1] Эдсгер Вибе Дейкстра — голландский ученый, занимавшийся теорией вычислительной техники, получивший в 1972 году премию Тьюринга за фундаментальный вклад в теорию языков программирования.

[2] Джеймс О. Коплин — автор, преподаватель и исследователь в области компьютерных наук.

Чтение каталогов паттернов и изучение их сценариев использования расширяет ваши знания распространенных проблем и путей их решения, даже если они еще не появились в вашем проекте. В этой книге было приведено немало историй из практики, рассказывавших о влиянии конкретного паттерна проектирования на проект. Эти истории — из реального опыта. Чтение и запоминание паттерна не гарантирует вам решения как по мановению волшебной палочки, но, когда вы столкнетесь с подобным затруднением или проблемой, может подсказать вам ключи к решению задачи. Очень скоро, с приходом опыта, благодаря использованию соответствующего паттерна вы будете принимать меры для устранения проблемы еще до ее возникновения.

Первоначально модель программирования J2EE в значительной степени основывалась на XML-конфигурациях и тяжеловесных EJB-контейнерах. Для должной работы компонентов требовались расширение определенных классов и реализация каждого метода. Очень скоро этот подход был признан низкопроизводительным и практически стал антипаттерном. Хотя Spring[1] и реализовывал подход с облегченными контейнерами, конструкция вышедшей вскоре платформы Java EE отдавала предпочтение встроенным аннотациям кода перед конфигурационными файлами. Облегченные контейнеры и EJB-компоненты, основанные на простых Java-объектах в старом стиле (POJO), предлагали производительную и удобную для тестирования модель программирования. Последующие версии платформы Java EE предоставили множество желанных возможностей, большая часть которых описана в нашей книге. Наконец, контекст и внедрение зависимостей (CDI) обеспечили новый контейнер с великолепными адаптивными возможностями. С помощью CDI вы можете без особых сложностей реализовать многие паттерны, такие как «Наблюдатель» и «Декоратор».

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Когда мне выпала обязанность работать над проектом Eclipse Libra, у меня были только ограниченные знания о подключаемых модулях Eclipse. Прочитав единственную доступную книгу Eclipse: Building Commercial-Quality Plug-Ins («Eclipse: Создание подключаемых модулей общего назначения»), написанную Дэном Рубелем и Эриком Клейбергом (Addison-Wesley, 2008), я решил углубиться в существующую базу кода в репозитории Eclipse.

Вскоре я был просто очарован архитектурой Eclipse вообще и построением подключаемых модулей в частности. Повсюду были использованные в правильном контексте паттерны проектирования, такие как «Адаптер», «Декоратор», «Стратегия», «Хранитель» и многие другие, обеспечивавшие эффективную и прозрачную реализацию.

Репозиторий кода Eclipse — один из лучших действующих источников правильной реализации паттернов проектирования в реальных проектах, используемый миллионами разработчиков.

Плохой: излишнее и неправильное использование паттернов

Первый пройденный мной курс повышения квалификации по паттернам проектирования просто потряс меня. Я провел весь следующий месяц за чтением книги

[1] Основанный на языке Java фреймворк, предоставляющий множество возможностей, включая внедрение зависимостей и аспектно-ориентированное программирование.

«Паттерны проектирования», а затем, конечно, книги GoF. Я вооружился знанием паттернов и был готов их использовать. С течением времени я осознал, что мне даже не требуется обращаться к наследованию, я могу построить все модели и иерархии объектов с помощью декораторов. Я даже создал набор утилитных классов, состоящий из нескольких одиночек, шин передачи сообщений для наблюдателей, а также общих декораторов и наблюдателей. Я подключаю его при создании нового проекта.

Я всегда с гордостью показывал свой код, демонстрируя всем, каким я был выдающимся и искушенным программистом. Потребовалось не так уж много времени, чтобы понять: мой подход по использованию паттернов проектирования только чрезмерно усложнял код и добавлял слишком много слоев во время выполнения. Даже упрощение существующего кода приводило к лучшей производительности. Сложный и замысловатый код не делает вас лучшим программистом, и при этом его нельзя назвать оптимальным и удобным для сопровождения. Инженерия — искусство использования правильного инструмента в правильном месте и эффективного построения системы.

ИСТОРИЯ ИЗ ПРАКТИКИ ОДНОГО ИЗ АВТОРОВ

Однажды на собеседовании меня попросили реализовать структуру данных для обработки операций транзакционной базы данных. Я должен был написать код дома и отправить его в фирму для оценки по электронной почте. Система должна была уметь добавлять новые значения и выполнять сохранение при фиксации транзакции, а также возвращаться к предыдущему состоянию при выполнении отката. Внутренний голос кричал: «Хранитель!» Хотя до того мне никогда не приходилось применять хранитель, я знал, что он идеально подходит. Паттерн «Хранитель» позволил бы мне выполнить фиксацию транзакции в точке сохранения, которую позднее можно было откатить. Так что я начал ревизию своих познаний о реализации паттерна «Хранитель». Я создал свои классы Caretaker, Memento и Originator, разместив их во внутреннем пакете, и реализовал код логики базы данных, использовавший внутренние классы хранителя.

Я гордился собой и был уверен в написанном коде. К моему удивлению, фирма не захотела продолжать собеседование. Возможно, они искали кого-то, кто использовал бы простой стек для проталкивания и выталкивания значений, но написанный код придал мне уверенности в моих знаниях паттернов проектирования, даже тех, которые я редко использовал.

Вернувшись к исходному вопросу спустя годы, я осознал наличие ограничений производительности при использовании минимальных объектов, а также понял, что у моей программы производительность была порядка O (log N). Мой код был читабелен, понятен и прост в сопровождении, однако он не сумел решить основные вопросы, которые интересовали клиента в первую очередь.

Паттерны проектирования приносят больше вреда, чем пользы, если их знание мешает вам разглядеть хорошие решения.

...ужасные

Паттерны проектирования и платформа Java EE — старые приятели. Однако эта дружба не всегда приносила удачу. Когда J2EE был принят корпоративным миром и началось его применение в больших проектах, на помощь пришли паттерны проектирования. Многие из классических паттернов проектирования из книги GoF нашли свое место в приложениях J2EE. А вскоре за ними последовали корпоративные паттерны для решения распространенных задач платформы J2EE.

Платформа J2EE стала популярной и дала толчок многим новым концепциям, таким как сервис-ориентированная архитектура (SOA) и веб-сервисы. Однако сложная структура платформы J2EE обрекла многие проекты на крах. J2EE-компоненты основаны на расширении классов и требуют для своего выполнения тяжеловесный контейнер. Поскольку компоненты рассчитывают на контейнер, для процесса разработки необходимы полнофункциональные тяжеловесные серверы, замедляющие разработку и требующие для своей работы дорогостоящего оборудования. Кроме того, те корпоративные контейнеры были медленными, что приводило к замедлению перезапусков и обновлений. Практически невозможно было выполнять должным образом тестирование и модульное тестирование.

Вдобавок конфигурация J2EE основывалась на объемных XML-файлах. Хотя разделение конфигурации и кода казалось хорошей идеей, скоро оно стало настоящим XML-кошмаром. Для создания простого компонента требовалась «тяжелая» конфигурация.

Когда J2EE стала корпоративной платформой, консультанты, программные архитекторы и поставщики стали выпускать сложные, запутанные инструкции, что привело к чрезмерному усложнению, чрезмерному проектированию приложений и чрезмерному количеству слоев у них. Такие приложения было невозможно тестировать и трудно разрабатывать (из-за длительности перезапусков), отлаживать и устанавливать.

К счастью, у истории корпоративной платформы Java счастливый конец. Движение за POJO и облегченные контейнеры, возглавляемое Родом Джонсоном[1], обрело множество последователей и вскоре стало конкурентом J2EE. Фреймворк Spring предоставил облегченный контейнер и возможность запуска на простых Java-серверах. Подход POJO был очень удобен для тестирования и большую часть времени не нуждался в контейнере, но даже если контейнер был нужен, использовать его было удобно.

Успех Spring привел к возрождению процесса обсуждения в Java-сообществе (Java Community Process). Платформа Java EE 5 была спроектирована с нуля для поддержки POJO EJB и более «легких» контейнеров. Платформа Java EE эволюционировала и достигла зрелости.

Однако старые привычки и методики программирования не поменялись за одну ночь. Многие разработчики, используя облегченные контейнеры и серверы, все еще следуют шаблонам J2EE, создавая слишком многослойные, сложные приложения. Как английский язык изменился со времен Шекспира, изменились и платформы и языки программирования. Не застревайте в прошлом, сопротивляясь переменам.

ИСТОРИЯ ИЗ ПРАКТИКИ

Это было в самом начале существования J2EE, и нам нужно было реализовать банковскую систему нового поколения. Мы задействовали все лучшие методики, паттерны, инструкции и все остальное, что только сумели найти в книгах и онлайн-ресурсах.

Наше приложение существенно зависело от конкретного поставщика и не было переносимым. Нам приходилось использовать интегрированную среду разработки (IDE) этого поставщика и его же сервер, и все это происходило в 32-битную эпоху, когда Windows отказывалась адресовать более

[1] Род Джонсон — австралийский программист, создавший фреймворк Spring.

3 Гбайт оперативной памяти. Сервер и IDE были настолько медленными, что при запуске в режиме отладки нам даже не нужны были точки останова для приостановки выполнения.

Поставщик заверял нас, что операционная среда будет быстрой, и тем не менее жизненный цикл разработки напоминал ядро на цепи (на ноге каторжника). Во время перезапуска сервера мы могли легко выйти попить кофе.

Все стало еще «веселее», когда мы захотели начать промышленную эксплуатацию. Операционная среда оказалась столь же медлительной, как и среда разработки. Скоро у нас появилась привычка наблюдать за состоянием памяти как операционной среды, так и среды разработки.

Наконец, чтобы выяснить, что же мы делаем не так, мы наняли известного консультанта. Он был профи, к которому мы относились как к Гендальфу[1]. После длившегося несколько дней анализа нашего кода он предложил нам убрать почти все фасады (у нас были фасады практически для каждого компонента) и все лишние интерфейсы (опять-таки у нас были чуть ли не интерфейсы для интерфейсов). Он также попросил нас минимизировать количество наших похожих на лазанью слоев, сократив иерархии вызовов (EJB-компонент обращается к EJB-компоненту, который обращается к EJB-компоненту...).

Это были времена J2EE 1.4, с тяжеловесными серверами от поставщиков, так что чуда не произошло. Тем не менее мы получили небольшой прирост производительности и, по крайней мере, намного более удобочитаемый код.

Допущение, что все может измениться, разработка в расчете на гибкость не обещает светлого будущего, зато практически гарантирует печальное настоящее.

Резюме

Паттерны проектирования — один из самых важных, многообещающих и полезных вопросов программного обеспечения. Нельзя стать настоящим объектно-ориентированным программистом без должного знания распространенных паттернов проектирования.

Хорошо понимая суть паттернов, вы фактически будете владеть замечательным набором инструментов для распространенных задач, которые вам, вероятно, повстречаются. Платформа Java EE делает еще шаг вперед: в нее интегрирован намного более удобный способ использования паттернов проектирования в корпоративных проектах. Большинство паттернов появились в платформе Java EE после долгих мучительных споров, что дает гарантию их хорошей реализации и близости к совершенству.

Все описанные в данной книге паттерны основаны на стандартах Java EE, так что они практически обречены на успешную работу.

Тем не менее паттерны не серебряные пули и не волшебные палочки. Если интенсивно использовать их без причины, они будут лишь чрезмерно усложнять проект. Знание вами паттерна не всегда должно означать, что вы должны его использовать, разве что вы уверены, что он подходит (для данной ситуации) и решает потенциальную проблему.

Читайте и изучайте паттерны проектирования и старайтесь периодически освежать память насчет того, где они могут пригодиться и какие задачи решают. Это сбережет вам немало строк кода и принесет всеобщее уважение.

---

[1] Волшебник из романа-эпопеи Дж. Р. Р. Толкина «Властелин колец». — *Примеч. пер.*