Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект лекции №9
Spring AOP
Транзакции

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Аспектно-ориентированный Spring

Некоторые операции, выполняемые программами, являются общими для большинства приложений. Регистрация событий, обеспечение безопасности и управление транзакциями – все это важные операции, но должны ли прикладные объекты принимать активное участие в них? Или для прикладных объектов лучше сосредоточиться на решении проблем предметной области, а управление некоторыми аспектами предоставить кому-то другому?

Функции, охватывающие несколько точек приложения, в разработке программного обеспечения называются сквозными . Как правило, сквозные функции концептуально отделены (но часто встроены) от основной логики приложения. Отделение основной логики от сквозных функций – это именно то, для чего предназначено аспектно-ориентированное программирование (AOP).

Мы уже знаем как использовать прием внедрения зависимостей (DI) для настройки и управления прикладными объектами. Подобно тому, как внедрение зависимостей помогает отделить прикладные объекты друг от друга, аспектно-ориентированное программирование помогает отделить сквозные функции от объектов, на которые они влияют.

Реализация регистрации событий является типичным примером применения аспектов, но это далеко не единственная область, где они могут пригодиться. Далее в книге будет представлено несколько практических примеров применения аспектов, включая декларативные транзакции, обеспечение безопасности и кеширование.

В этой главе исследуется поддержка аспектов в Spring, включая объявление аспектами обычных классов и особенности приме нения аннотаций для создания аспектов. Кроме того, здесь будет представлено расширение AspectJ, еще одна популярная реализация AOP, способная дополнить реализацию AOP в Spring.

1.1 Знакомство с AOP

Как отмечалось ранее, аспекты помогают отделить сквозные функции. Проще говоря, сквозные функции могут быть описаны как некоторая функциональность, затрагивающая множество мест в приложении. Обеспечение безопасности, например, является такой сквозной функцией – правила соблюдения безопасности могут затрагивать множество методов в приложении. Рисунок 1 дает визуальное представление сквозных функций.

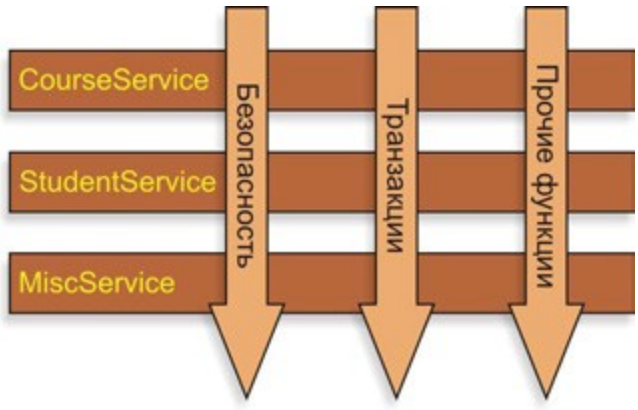

Рисунок 1 — Аспекты отделяют сквозные функции, реализуя логику, охватывающую множество объектов в приложениях

Рисунок 1 представляет типичное приложение, разбитое на модули. Главная задача каждого из них – предоставление услуг в конкретной предметной области. Но каждый из этих модулей также требует выполнения вспомогательных функций, таких как обеспечение безопасности и управление транзакциями.

Общая объектно-ориентированная методика повторного использования общей функциональности заключается в применении наследования или делегирования. Однако наследование может привести к созданию хрупкой иерархии объектов, если по всему приложению используется один и тот же базовый класс, а реализация делегирования может оказаться слишком громоздкой, поскольку может потребоваться выполнять сложные вызовы объекта-делегата.

Аспекты предлагают альтернативу наследованию и делегированию, более простую во многих случаях. При использовании аспектно-ориентированного программирования общая функциональность так же определяется в одном месте, но порядок и место применения этой функциональности можно определять декларативно, без изменения класса, к которому применяются новые возможности. Сквозные функции могут быть выделены в специальные классы, называемые аспектами . Это дает два преимущества. Во-первых, логика каждой из них теперь сосредоточена в одном месте, в отличие от случаев, когда она разбросана по всей программе. Во-вторых, упрощаются прикладные модули, поскольку содержат только реализацию своей главной задачи (или базовой функциональности), а вторичные проблемы вынесены в аспекты.

1.1.2 Определение терминологии AOP

Как и большинство технологий, в AOP сформировался свой собственный жаргон. Аспекты часто описываются в терминах «советов», «срезов множества точек сопряжения» и «точек сопряжения». Взаимосвязь этих понятий иллюстрирует рисунке 2.

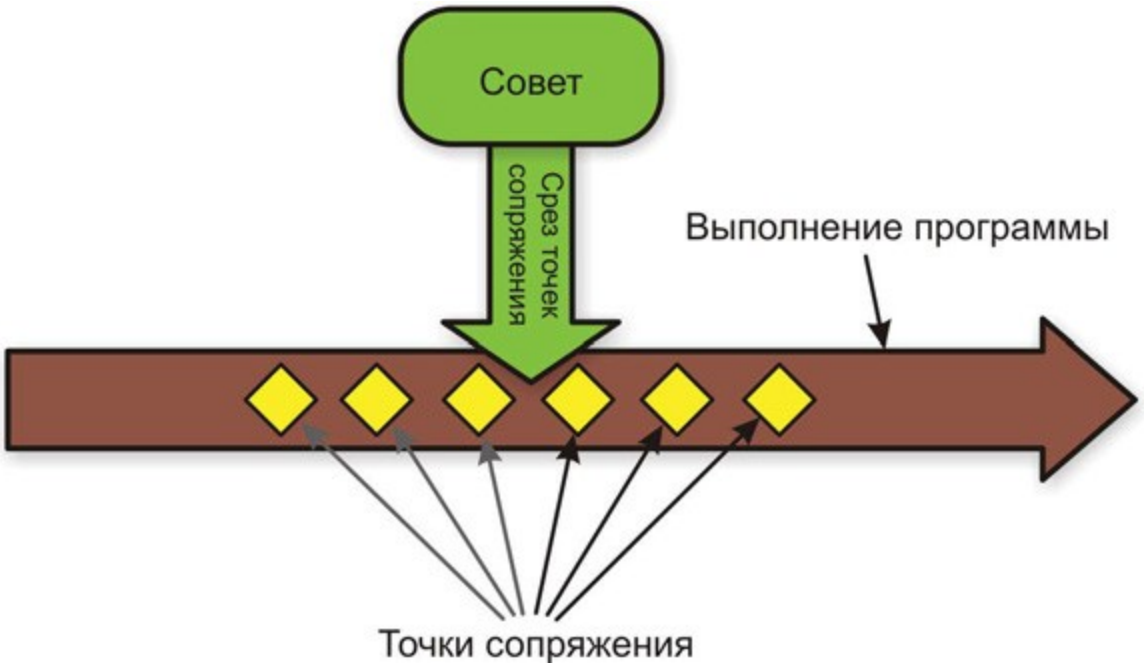

Рисунок 2 — Функциональность аспектов (советов) вплетается в поток выполнения программы в одной или нескольких точках сопряжения

К сожалению, многие термины, используемые для описания AOP, непонятны для непосвященных. Тем не менее они являются частью идиомы AOP, и знать их совершенно необходимо для понимания AOP. Прежде чем пускаться в путь, нужно научиться говорить.

Совет

Аспекты имеют свою цель – работу, которую они призваны делать. В терминах AOP работа аспекта называется совет. Совет определяет, что и когда делает аспект. В дополнение к описанию работы, выполняемой аспектом, совет учитывает, когда следует ее выполнять. Будет ли она выполняться перед вызовом метода? После его вызова? Или и в том, и другом случаях? Или только когда метод возбудит исключение?

Аспекты Spring могут работать с пятью типами советов:

- *до* – работа выполняется перед вызовом метода;
- *после* – работа выполняется после вызова метода, независимо от результата;
- *после успешного вызова* – работа выполняется после вызова метода, если его выполнение завершилось успешно;
- *после исключения* – работа выполняется после того, как вызванный метод возбудит исключение;
- *вокруг* – аспект обертывает метод, обеспечивая выполнение некоторых операций до и после вызова метода.

Точки сопряжения

Приложение может иметь тысячи точек применения совета. Эти точки известны как точки сопряжения (join points). Точка сопряжения – это точка в потоке выполнения приложения, куда может быть внедрен аспект. Это может быть вызов метода, возбуждение исключения или даже изменение поля. Все это – точки, куда может быть внедрен аспект для добавления новой особенности поведения.

Срезы множества точек сопряжения

От аспекта не требуется воздействовать на все точки сопряжения в приложении. Срезы множества точек сопряжения помогают сузить множество точек для внедрения аспекта.

Если совет отвечает на вопросы что и когда, то срезы множества точек сопряжения отвечают на вопрос где. Срез содержит одну или более точек сопряжения, куда должны быть вплетены советы. Часто срезы множества точек сопряжения определяются за счет явного указания имен классов и методов или через регулярные выражения, определяющие шаблоны имен классов и методов. Некоторые фреймворки, поддерживающие AOP, позволяют создавать срезы множества точек сопряжения динамически, определяя необходимость применения совета, опираясь на решения, принимаемые во время выполнения, такие как значения параметров метода.

Аспекты

Аспект объединяет в себе совет и срез множества точек сопряжения. Взятые вместе, они определяют все, что нужно знать об аспекте, – что он делает, где и когда.

Внедрение

Внедрение позволяет добавлять новые методы или атрибуты в существующие классы. Например, можно создать класс-совет Auditable, хранящий информацию о том, когда объект был изменен в последний раз. Это может быть очень простой класс, состоящий из единственного метода, например setLastModified(Date), и переменной экземпляра для хранения этой информации. В дальнейшем новый метод и переменная могут быть внедрены в существующие классы без их изменения, добавляя новые черты поведения и информацию.

Вплетение

Вплетение – это процесс применения аспектов к целевому объекту для создания нового, проксированного объекта. Аспекты вплетаются в целевой объект в указанные точки сопряжения. Вплетение может происходить в разные моменты жизненного цикла целевого объекта.

- *Во время компиляции* – аспекты вплетаются в целевой объект, когда тот компилируется. Это требует специального компилятора, такого как AspectJ, вплетающего аспекты на этапе компиляции.
- *Во время загрузки класса* – вплетение аспектов выполняется в процессе загрузки целевого класса виртуальной машиной JVM. Это требует специального загрузчика, который дополняет байт-код целевого класса перед внедрением его в приложение, например механизм load-time weaving (LTW) в AspectJ 5.
- *Во время выполнения* – вплетение аспектов производится во время выполнения приложения. В этом случае контейнер AOP обычно динамически генерирует объект с вплетенным аспектом, представляющий целевой объект.

Вы познакомились с довольно большим количеством терминов. Если теперь вернуться к рисунку 2, можно увидеть, как совет, содержащий реализацию сквозной функции, может применяться к объектам в приложении. Точки сопряжения – это все точки в потоке выполнения приложения, к которым при необходимости можно было бы применить совет. Срез множества точек сопряжения определяет, куда (к каким точкам сопряжения) применяется этот совет. Здесь важно понять, что срез определяет точки сопряжения для применения совета.

Теперь, после знакомства с терминологией AOP, посмотрим, как эти основные концепции AOP реализованы в Spring.

1.1.2 Поддержка AOP в Spring

Не все фреймворки AOP равноценны. Они могут отличаться богатством моделирования точек сопряжения. Некоторые позволяют применять советы на уровне изменения полей, тогда как другие предлагают точки сопряжения на уровне вызовов методов. Они могут также отличаться порядком и особенностями вплетения аспектов. Но не эти особенности делают фреймворки фреймворками с поддержкой AOP, а возможность определения срезов множества точек сопряжения аспектов.

За последние несколько лет в AOP многое изменилось. В результате наведения порядка в этой области некоторые фреймворки объединились, а некоторые исчезли со сцены. В 2005 году произошло слияние проектов AspectWerkz и AspectJ, ознаменовавшее самое значительное событие в мире AOP и оставившее нам три доминирующих фреймворка AOP:

- AspectJ (http://eclipse.org/aspectj);
- JBoss AOP (http://www.jboss.org/jbossaop);
- Spring AOP (http://www.springframework.org).

Мы сосредоточимся на Spring AOP. Тем не менее проекты Spring и AspectJ тесно взаимодействуют друг с другом, и реализация поддержки AOP в Spring многое заимствует из AspectJ.

Фреймворк Spring поддерживает четыре разновидности AOP:

- классическое аспектно-ориентированное программирование на основе промежуточных объектов;
- аспекты, создаваемые с применением аннотаций @AspectJ;
- аспекты на основе POJO;
- внедрение аспектов AspectJ (доступно во всех версиях Spring).

Первые три разновидности являются вариантами аспектно-ориентированного программирования на основе промежуточных объектов. Соответственно, поддержка AOP в Spring ограничивается перехватом вызовов методов. Если для работы аспекта потребуется нечто более сложное, чем простой перехват вызовов методов (например, перехват вызова конструктора или определение момента изменения значения свойства), следует подумать о возможности реализации аспектов в AspectJ , возможно, через внедрение компонентов Spring в аспекты AspectJ посредством механизма внедрения зависимостей.

В этой главе мы исследуем приемы аспектно-ориентированного программирования с использованием фреймворка Spring, но прежде разберемся с некоторыми ключевыми понятиями фреймворка AOP.

Советы в Spring реализуются на языке Java

Все советы , которые вы будете создавать с использованием фреймворка Spring, будут написаны как стандартные Java-классы. То есть при создании аспектов можно пользоваться той же самой интегрированной средой разработки, которая используется для разработки обычного программного кода на Java. К тому же срезы множества точек сопряжения, указывающие, где должны применяться советы, обычно определяются в XML-файле конфигурации Spring.

Сравните это с AspectJ. Сейчас AspectJ поддерживает возможность определения аспектов на основе аннотаций, но это – расширение языка Java. В таком подходе есть свои преимущества и недостатки. Наличие языка AOP предполагает широкие возможности и богатый набор инструментов AOP, однако прежде чем все это использовать, необходимо изучить инструменты и синтаксис.

Советы в Spring – это объекты времени выполнения

При использовании фреймворка Spring, аспекты вплетаются в компоненты во время выполнения посредством обертывания их прокси-классами. Как показано на рисунке 3, прокси-класс играет роль целевого компонента, перехватывая вызовы методов и передавая эти вызовы целевому компоненту.

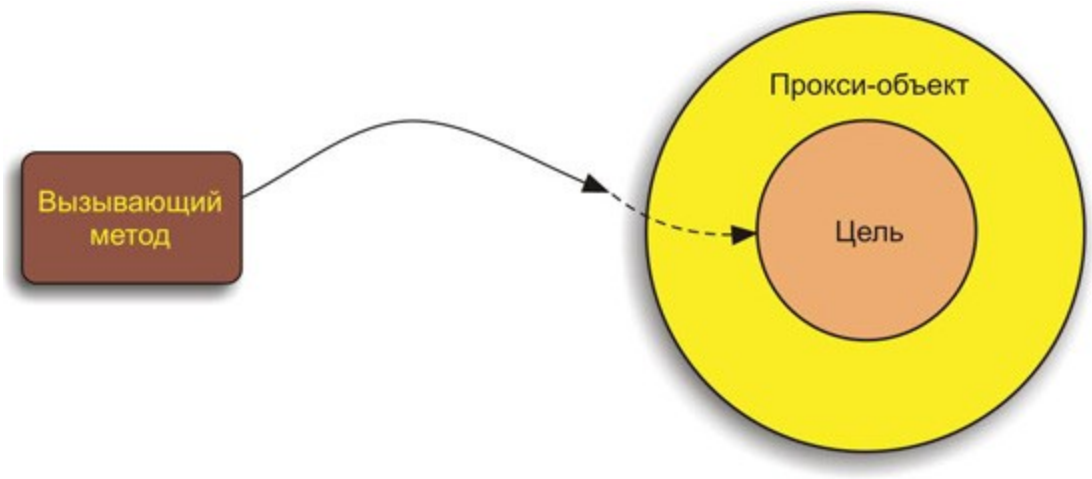

Рисунок 3 — Аспекты Spring реализуются в виде прокси-объектов, обертывающих целевые объекты. Прокси-объект обрабатывает вызовы методов, выполняет дополнительную логику аспекта и затем вызывает целевой метод

Между моментами времени, когда прокси-объект перехватывает вызов метода, и вызовом метода целевого компонента выполняется логика аспекта.

Фреймворк Spring не создает проксированные объекты, пока в них нет необходимости. При использовании контекста приложения ApplicationContext проксированные объекты создаются после загрузки всех компонентов из BeanFactory. Поскольку Spring создает прокси-объекты во время выполнения, для вплетения аспектов в Spring AOP не требуется специальный компилятор.

Spring поддерживает точки сопряжения только для методов

Как упоминалось выше, различные реализации AOP обеспечивают несколько моделей точек сопряжения. Так как поддержка AOP в Spring основана на использовании динамических прокси-объектов, точками сопряжения могут служить только методы, в отличие от некоторых других фреймворков AOP, таких как AspectJ и JBoss, где помимо методов роль точек сопряжения могут играть поля и конструкторы. Отсутствие в Spring возможности применения аспектов к полям препятствует созданию высокоизбирательных аспектов, например для отслеживания изменений полей в объектах. Невозможность применить аспект к конструктору также препятствует реализации логики, которая должна выполняться в момент создания экземпляра компонента.

Однако возможности перехватывать вызовы методов должно быть достаточно для удовлетворения если не всех, то большинства потребностей. Если потребуется нечто большее, чем возможность перехвата вызовов методов, всегда можно воспользоваться расширением AspectJ.

Теперь, получив общее представление о возможностях AOP и особенностях его поддержки в Spring, можно перейти к практическому

изучению принципов создания аспектов в Spring. Сначала познакомимся поближе с декларативной моделью AOP.

1.2 Выбор точек сопряжения в описаниях срезов

Как упоминалось выше, срезы множества точек сопряжения – это множества точек применения советов аспектов в программном коде. Вместе с советами аспектов срезы являются одними из наиболее фундаментальных элементов аспектов. Поэтому важно знать и понимать, как они определяются.

В Spring AOP срезы множества точек сопряжения определяются на языке выражений AspectJ. Знакомые с расширением AspectJ не будут испытывать затруднений при определении срезов в Spring. Но если вы незнакомы с AspectJ, этот раздел послужит вам кратким учебником по определению срезов в стиле AspectJ.

Самое важное, что следует помнить о механизме формирования срезов множества точек сопряжения в стиле AspectJ, входящем в состав Spring AOP, – он поддерживает ограниченное подмножество конструкций описания, доступных в AspectJ. Напомним, что модель Spring AOP основана на использовании прокси-объектов, а некоторые элементы выражений неприменимы в этой модели AOP.

В списке ниже перечислены указатели для использования в описаниях срезов в AspectJ, поддерживаемые в Spring AOP.

Конструкции языка выражений описания срезов множества точек сопряжения в AspectJ, поддерживаемые аспектами в Spring:

args() Ограничивает срез точек сопряжения вызовами методов, чьи аргументы являются экземплярами указанных типов

@args() Ограничивает срез точек сопряжения вызовами методов, чьи аргументы аннотированы указанными типами аннотаций

execution() Соответствует точкам сопряжения, которые являются вызовами методов

this() Ограничивает срез точек сопряжений точками, где ссылка на компонент является ссылкой на прокси-объект указанного типа

target() Ограничивает срез точек сопряжений точками, где целевой объект имеет указанный тип

@target() Ограничивает срез точек сопряжений точками, где класс выполняемого объекта снабжен аннотацией указанного типа

within() Ограничивает срез точек сопряжений точками только внутри указанных типов

@within() Ограничивает срез точек сопряжений точками внутри указанных типов, снабженных указанной аннотацией (в Spring AOP соответствует вызовам методов в указанном типе, отмеченных указанной аннотацией)

@annotation Ограничивает срез точек сопряжений точками, помеченными указанной аннотацией

Попытки использовать любые другие указатели AspectJ приведут к исключению IllegalArgumentException.

Обратите внимание, что среди поддерживаемых указателей только execution фактически выполняет сопоставление – все остальные используются для ограничения множества совпадений. Это означает, что execution является основным указателем, который должен использоваться во всех определениях срезов множества точек сопряжения. Остальные указатели применяются только для ограничения точек сопряжения в срезе.

1.2.1 Определение срезов множества точек сопряжения

На рисунке 4 представлено выражение, определяющее срез множества точек сопряжения, который можно использовать для применения совета к вызову метода play() интерфейса Instrument.

Рисунок 4 — Выражение, определяющее срез множества точек сопряжения AspectJ и выбирающее метод play() интерфейса Instrument

Указатель execution() используется здесь для выбора метода play() интерфейса Instrument. Описание метода начинается со звездочки, указывающей, что тип возвращаемого методом значения не должен учитываться. Далее определяются полное имя класса и имя метода, который требуется выбрать. В списке параметров метода указаны две точки, идущие подряд (..), указывающие, что в срез может быть включена любая точка сопряжения, соответствующая любому вызову метода play(), независимо от конкретного списка передаваемых ему аргументов.

Теперь предположим, что необходимо ограничить множество точек сопряжения границами пакета com.springinaction.springidol. В этом случае можно задействовать указатель within(), как показано на рисунке 5.

Рисунок 5 — Ограничение множества точек сопряжения с помощью указателя within()

Обратите внимание на оператор &&, объединяющий указатели execution() и within() логической операцией «И» (то есть использоваться будут только точки сопряжения, где будут выполнены требования обоих указателей). Аналогично, чтобы объединить указатели логической операцией «ИЛИ», можно было бы использовать оператор ||. А чтобы инвертировать смысл указателя – использовать оператор !.

Поскольку в языке разметки XML амперсанды имеют специальное значение, при определении срезов множества точек сопряжения в конфигурационном XML-файле вместо оператора && можно использовать оператор and. Аналогично можно использовать операторы or и not вместо || и ! (соответственно).

1.2.2 Использование указателя bean()

Помимо указателей, перечисленных в списке выше, в версии Spring 2.5 появился новый указатель bean(), позволяющий идентифицировать компоненты внутри выражений определения срезов по их идентификаторам. Указатель bean() принимает идентификатор или имя компонента в виде аргумента и ограничивает срез множества точек сопряжения, оставляя в нем только точки, соответствующие указанному компоненту.

Например, взгляните на следующее определение среза:

```
execution(* com.springinaction.springidol.Instrument.play())
        and bean(eddie)
```

Это определение говорит, что совет аспекта должен применяться к вызовам метода Instrument.play(), но только внутри компонента с идентификатором eddie. Возможность ограничения доступных точек сопряжения границами определенного компонента может оказаться ценной в некоторых случаях, однако имеется также возможность инвертировать условие и обеспечить применение аспекта ко всем компонентам, кроме имею щего определенный идентификатор:

```
execution(* com.springinaction.springidol.Instrument.play())
        and !bean(eddie)
```

В данном случае совет аспекта будет вплетен во все компоненты, кроме компонента с идентификатором eddie.

Теперь, после знакомства с основами определения срезов множества точек сопряжения, можно перейти к созданию советов и объявлению аспектов, использующих эти срезы.

1.3 Объявление аспектов в XML

Знакомые с классической моделью аспектно-ориентированного программирования в Spring знают, что работать с ProxyFactoryBean очень неудобно. В свое время разработчики Spring осознали это и приступили к реализации более удобного способа объявления аспектов в Spring. В результате их усилий в пространстве имен aop появились новые элементы. Перечень элементов настройки механизма AOP приводится в списке ниже:

Элементы настройки механизма AOP в Spring упрощают объявление аспектов, основанных на POJO:

<aop:advisor> Определяет объект-советник

<aop:after> Определяет AOP-совет, выполняемый после вызова метода (независимо от успешности его завершения)

<aop:after-returning> Определяет AOP-совет, выполняемый после успешного выполнения метода

<aop:after-throwing> Определяет AOP-совет, выполняемый после возбуждения исключения

<aop:around> Определяет AOP-совет, выполняемый до и после выполнения метода

<aop:aspect> Определяет аспект

<aop:aspectj-autoproxy> Включает поддержку аспектов, управляемых аннотациями, созданными с применением аннотации @AspectJ

<aop:before> Определяет AOP-совет, выполняемый до выполнения метода Объявление аспектов в XML

<aop:config> Элемент верхнего уровня настройки механизма AOP

<aop:declare-parents> Внедряет в объекты прозрачную реализацию дополнительных интерфейсов

<aop:pointcut> Определяет срез точек сопряжения

Дальнейшие примеры (*Пример взят из книги.*) будем рассматривать на примере реализации конкурса талантов «Spring Idol».В этом примере будем демонстрировать нескольких исполнителей в виде компонентов Spring, чтобы они могли продемонстрировать свои таланты. Это довольно забавный пример. Но для конкурсов, подобных этому, нужны зрители, иначе в их проведении нет никакого смысла.

Поэтому, чтобы продемонстрировать возможности Spring AOP, создадим класс Audience для нашего примера конкурса талантов. В листинге 1 представлен класс, определяющий функции зрителей.

Листинг 1. Класс Audience для конкурса талантов

```
package com.springinaction.springidol;
public class Audience {
        public void takeSeats() { // Перед выступлением
                System.out.println("The audience is taking their seats.");
        }
        public void turnOffCellPhones() { // Перед выступлением
```

```
            System.out.println("The audience is turning off their cellphones");
        }
        public void applaud() { // После выступления
            System.out.println("CLAP CLAP CLAP CLAP CLAP");
        }
        public void demandRefund() { // После неудачного выступления
            System.out.println("Boo! We want our money back!");
        }
}
```

Как видите, в классе Audience нет ничего примечательного. Это обычный Java-класс с горсткой методов. Его можно зарегистрировать в виде компонента в контексте приложения Spring, подобно любому другому классу:

```
<bean id="audience" class="com.springinaction.springidol.Audience" />
```

Несмотря на его неприметность, класс Audience обладает всем необходимым для создания аспекта. К нему нужно лишь добавить немного волшебства Spring AOP.

1.3.1 Объявление советов, выполняемых до или после

С помощью элементов настройки механизма Spring AOP компонент audience можно превратить в аспект, как показано в листинге 2.

Листинг 2. Определение аспекта audience с использованием элементов настройки Spring AOP

```
<aop:config>
    <aop:aspect ref="audience"> <!-- Ссылка на компонент audience -->
        <aop:before pointcut=
    "execution(* com.springinaction.springidol.Performer.perform(..))"
        method="takeSeats" /> <!-- Перед выступлением -->

    <aop:before pointcut=
    "execution(* com.springinaction.springidol.Performer.perform(..))"
    method="turnOffCellPhones" /> <!-- Перед выступлением -->

    <aop:after-returning pointcut=
     "execution(* com.springinaction.springidol.Performer.perform(..))"
        method="applaud" /> <!-- После выступления -->

    <aop:after-throwing pointcut=
"execution(* com.springinaction.springidol.Performer.perform(..))"
    method="demandRefund" /> <!-- После неудачного выступления -->
    </aop:aspect>
 </aop:config>
```

Первое, на что следует обратить внимание, – большинство элементов настройки должны находиться внутри элемента <aop:config> .

Есть некоторые исключения из этого правила, но когда дело доходит до объявления компонентов аспектами, конфигурация всегда должна начинаться с элемента <aop:config>.

Внутри <aop:config> можно объявить один или более объектов-советников, аспектов или срезов множества точек сопряжения. В листинге 2 был объявлен единственный аспект с помощью элемента <aop:aspect> . Атрибут ref ссылается на компонент POJO, реализующий функциональность аспекта, в данном случае audience. Компонент, на который ссылается атрибут ref, определяет методы, вызываемые советами в аспекте.

Сам аспект определяет четыре совета. Два элемента <aop:before> определяют советы, выполняемые перед вызовом метода, которые будут вызывать методы takeSeats() и turnOffCellPhones() компонента Audience перед вызовами любых методов, определяемых срезом с точками сопряжения. Элемент <aop:after-returning> определяет совет, выполняемый после успешного завершения метода и вызывающий метод applaud() после любых методов, определяемых срезом. И элемент <aop:after-throwing> определяет совет, выполняемый в случае возбуждения исключения и вызывающий метод demandRefund(), если было возбуждено какое-либо исключение. На рисунке 6 иллюстрирует, как логика советов вплетается в основную логику работы приложения.

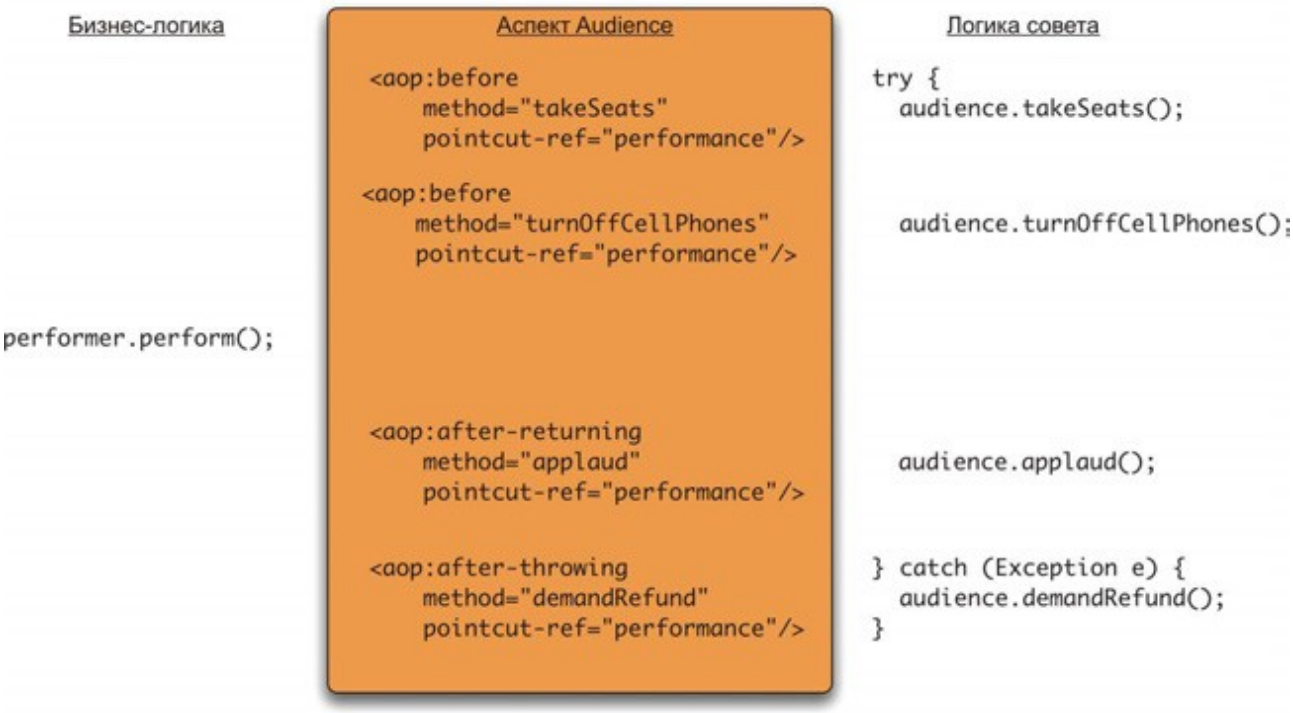

Рисунок 6 — Аспект Audience включает четыре совета, вплетаемых в логику вызова методов, определяемых срезом с точками сопряжения

Во всех элементах, объявляющих советы, атрибут pointcut – срез множества точек сопряжения, где будет применяться данный совет. Значением атрибута pointcut является определение среза с применением синтаксиса выражений AspectJ.

Обратите внимание, что во всех элементах объявления советов атрибут pointcut имеет одинаковое значение, потому что все советы применяются к одному и тому же срезу множества точек сопряжения.

Однако это является нарушением принципа DRY(don’t repeat your self – не повторяйся). Если впоследствии потребуется изменить определение среза, это придется сделать в четырех различных местах.

Чтобы избежать дублирования определений срезов множества точек сопряжения, можно определить именованный срез с помощью элемента <aop:pointcut> . В листинге 3 представлен фрагмент XML, демонстрирующий, как внутри элемента <aop:aspect> определить именованный срез с помощью элемента <aop:pointcut>, который затем можно использовать во всех элементах определения советов.

Листинг 3. Определение именованного среза множества точек сопряжения для устранения избыточных определений срезов

```
<aop:config>
    <aop:aspect ref="audience">
        <aop:pointcut id="performance" expression=
            "execution(* com.springinaction.springidol.Performer.perform(..))"
             /> <!-- Определение среза множества точек сопряжения -->

        <aop:before
            pointcut-ref="performance" <!-- Ссылка на именованный срез -->
            method="takeSeats" />

        <aop:before
            pointcut-ref="performance" <!-- Ссылка на именованный срез -->
            method="turnOffCellPhones" />

        <aop:after-returning
            pointcut-ref="performance" <!-- Ссылка на именованный срез -->
            method="applaud" />

        <aop:after-throwing
            pointcut-ref="performance" <!-- Ссылка на именованный срез -->
            method="demandRefund" />
    </aop:aspect>
</aop:config>
```

Теперь определение среза, ссылки на который используются в нескольких местах, существует в единственном экземпляре. Элемент <aop:pointcut> определяет срез множества точек сопряжения с идентификатором performance.

Кроме того, во всех элементах, объявляющих советы, определения срезов были заменены атрибутом pointcut-ref со ссылкой на именованный срез.

Как показано в листинге 3, элемент <aop:pointcut> определяет срез множества точек сопряжения, на который можно ссылаться из любых советов, объявленных внутри того же элемента <aop:aspect>. Однако имеется возможность определять срезы, на которые можно ссылаться из разных аспектов, для чего достаточно поместить элементы <aop:pointcut> в область действия элемента <aop:config> .

1.3.2 Объявление советов, выполняемых и до, и после

Текущая реализация класса Audience работает замечательно. Но простые советы, выполняемые до или после вызова метода, имеют некоторые ограничения. В частности, довольно сложно обеспечить совместное использование информации советами, выполняемыми до или после, не прибегая к использованию переменных экземпляра.

Например, предположим, что к выключению сотовых телефонов и аплодисментов в конце также было бы желательно, чтобы зрители посматривали на часы и отмечали продолжительность выступления. Единственный способ решить эту задачу – использовать совет, выполняемый до и после, чтобы перед выступлением засечь текущее время, а после выступления определить его продолжительность. Но при этом придется сохранить время начала в переменной экземпляра. Поскольку экземпляр класса Audience – единственный в приложении, он не может использоваться в многопоточной среде выполнения без риска повредить хранимые в нем данные.

В этом отношении совет, выполняемый и до, и после вызова целевого метода, имеет преимущество перед советами, выполняемыми до или после. Он позволяет решать те же задачи, что и советы, выполняемые до или после, но делает это в единственном методе. А так как совет целиком выполняется в рамках единственного метода, нет необходимости сохранять промежуточные данные в переменных экземпляра.

Например, взгляните на новый метод watchPerformance(), представленный в листинге 4.

Листинг 4 Метод watchPerformance() реализует совет AOP, выполняемый и до, и после вызова целевого метода

```
public void watchPerformance(ProceedingJoinPoint joinpoint) {
        try {
                System.out.println("The audience is taking their seats.");
                System.out.println("The audience is turning off their cellphones");
                long start = System.currentTimeMillis(); // Перед выступлением

                joinpoint.proceed(); // Вызов целевого метода

                long end = System.currentTimeMillis(); // После выступления
                System.out.println("CLAP CLAP CLAP CLAP CLAP");
```

```
            System.out.println("The performance took " + (end — start)
            + " milliseconds.");
        } catch (Throwable t) {
            System.out.println("Boo! We want our money back!");
        }
    }
```

Первое, на что следует обратить внимание в реализации нового совета, – он принимает параметр типа ProceedingJoinPoint . Этот объект необходим для вызова целевого метода внутри совета. Метод совета выполнит все необходимые предварительные операции и, когда будет готов передать управление целевому методу, вызовет метод ProceedingJoinPoint.proceed().

Имейте в виду, что крайне важно не забывать включать вызов метода proceed(). Если этого не сделать, совет фактически заблокирует доступ к целевому методу. Возможно, в некоторых случаях именно это и требуется, но чаще бывает желательно, чтобы целевой метод был выполнен в какой-то момент.

Интересно отметить, что, кроме блокирования доступа к целевому методу отказом от вызова метода proceed(), внутри совета его также можно вызвать несколько раз. Это может пригодиться для реализации повторных попыток вызова целевого метода, на случай если он может потерпеть неудачу.

В примере с аспектом audience метод watchPerformance() реализует все, что было реализовано в четырех прежних методах совета, но теперь все это сосредоточено в единственном методе, и он несет полную ответственность за обработку ошибок.

Обратите также внимание, что непосредственно перед вызовом метода proceed(), соответствующего точке сопряжения, в локальной переменной сохраняется текущее время. Сразу после возврата из метода выводится длительность выступления. Объявление совета, выполняющегося и до, и после целевого метода, мало отличается от объявлений советов других типов, достаточно просто использовать элемент <aop:around>, как показано в листинге 5.

Листинг 5. Определение аспекта audience с единственным советом, выполняемым и до, и после вызова целевого метода

```
<aop:config>
    <aop:aspect ref="audience">
    <aop:pointcut id="performance2" expression=
        "execution(* com.springinaction.springidol.Performer.perform(..))" />
    <!-- Совет, выполняемый и до, и после -->
    <aop:around pointcut-ref="performance2"
        method="watchPerformance()" />
    </aop:aspect>
</aop:config>
```

Подобно другим XML-элементам определения советов, в элементе <aop:around> указываются срез множества точек сопряжения и имя метода,

реализующего совет. Здесь используется тот же срез, что и прежде, но в атрибуте method указано имя нового метода watchPerformance().

1.3.3. Передача параметров советам

До сих пор наши аспекты отличались простотой реализации и не принимали параметров. Единственное исключение – метод watchPerformance(), реализующий совет, выполняемый и до, и после целевого метода, который принимает параметр типа ProceedingJoinPoint. Реализованный нами совет никак не заботится о параметрах для передачи целевому методу. Впрочем, в этом нет ничего страшного, потому что вызываемый нами метод perform() не принимает никаких параметров.

Тем не менее иногда бывает желательно, чтобы совет не только обертывал целевой метод, но также проверял значения передаваемых ему параметров.

Чтобы увидеть, как действует эта возможность, познакомимся с новым участником конкурса «Spring Idol». Наш новый участник обладает талантом чтения мыслей и определяется интерфейсом MindReader :

```
package com.springinaction.springidol;
public interface MindReader {
    void interceptThoughts(String thoughts);
    String getThoughts();
}
```

Человек, читающий мысли, выполняет два основных задания: он читает мысли добровольца и сообщает их публике. Ниже представлен простой класс Magician, реализующий интерфейс MindReader:

```
package com.springinaction.springidol;
public class Magician implements MindReader {
    private String thoughts;
    public void interceptThoughts(String thoughts) {
        System.out.println("Intercepting volunteer’s thoughts");
        this.thoughts = thoughts;
    }
    public String getThoughts() {
        return thoughts;
    }
}
```

Теперь нам нужен доброволец, чьи мысли будут прочитаны. Для этой цели определим интерфейс Thinker:

```
package com.springinaction.springidol;
public interface Thinker {
    void thinkOfSomething(String thoughts);
}
```

Класс Volunteer представляет собой простейшую реализацию интерфейса Thinker:

```
package com.springinaction.springidol;
```

```
public class Volunteer implements Thinker {
	private String thoughts;
	public void thinkOfSomething(String thoughts) {
		this.thoughts = thoughts;
	}

	public String getThoughts() {
		return thoughts;
	}
}
```

Детали реализации Volunteer не представляют особого интереса или важности. Что действительно интересно – как фокусник (объект Magician) будет читать мысли добровольца (объект Volunteer) с использованием Spring AOP.

Для осуществления телепатического контакта воспользуемся теми же элементами <aop:aspect> и <aop:before>, что и прежде. Но на этот раз настроим в них передачу совету параметров для целевого метода.

```
<aop:config>
	<aop:aspect ref="magician">
		<aop:pointcut id="thinking"
			expression="execution(*
			com.springinaction.springidol.Thinker.thinkOfSomething(String))
				and args(thoughts)" />
		<aop:before pointcut-ref="thinking"
			method="interceptThoughts"
			arg-names="thoughts" />
	</aop:aspect>
</aop:config>
```

Ключ к телепатическим возможностям фокусника Magician находится в определении среза множества точек сопряжения и в атрибуте arg-names элемента <aop:before>. Срез идентифицирует метод thinkOfSomething() интерфейса Thinker, указывая, что метод принимает строковый аргумент, и уточняя имя аргумента thoughts с помощью указателя args.

Элемент <aop:before> определения совета также ссылается на аргумент thoughts, указывая, что он должен передаваться методу interceptThoughts() класса Magician. Теперь всякий раз, когда будет вызываться метод thinkOfSomething() компонента volunteer, аспект Magician будет перехватывать его мысли.

Чтобы убедиться в этом, ниже представлен простой тестовый класс со следующим методом:

```
@Test
public void magicianShouldReadVolunteersMind() {
	volunteer.thinkOfSomething("Queen of Hearts");
	assertEquals("Queen of Hearts", magician.getThoughts());
}
```

1.3.4 Внедрение новых возможностей с помощью аспектов

В некоторых языках программирования, таких как Ruby и Groovy, есть понятие открытых классов. Они позволяют добавлять в объекты или классы новые методы без непосредственного изменения определения этих объектов/классов. К сожалению, язык Java не обладает такими динамическими возможностями. Как только класс будет скомпилирован, у вас остается совсем немного возможностей расширить его функциональные возможности.

Но если задуматься, разве аспекты, рассматривающиеся в этой главе, не являются такой возможностью? Конечно, мы не добавляли новых методов в объекты, но мы расширяли функциональные возможности уже существующих методов. Коль скоро аспекты позволяют обертывать существующие методы новыми функциональными возможностями, почему бы с их помощью не попробовать добавлять новые методы? В действительности концепция внедрения, существующая в аспектно-ориентированном программировании, позволяет присоединять новые методы к компонентам Spring.

Напомним, что аспекты в Spring – это всего лишь промежуточные объекты-обертки (прокси-объекты), реализующие те же самые интерфейсы, что и компоненты, которые они обертывают. Что, если кроме реализации этих интерфейсов прокси-объект будет содержать реализацию некоторого другого интерфейса? Тогда любой компонент, окруженный таким аспектом, будет выглядеть как объект, реа лизующий дополнительный интерфейс, даже если лежащий в его основе класс в действительности не реализует его. На рисунке 7 изобра жено, как действует этот механизм.

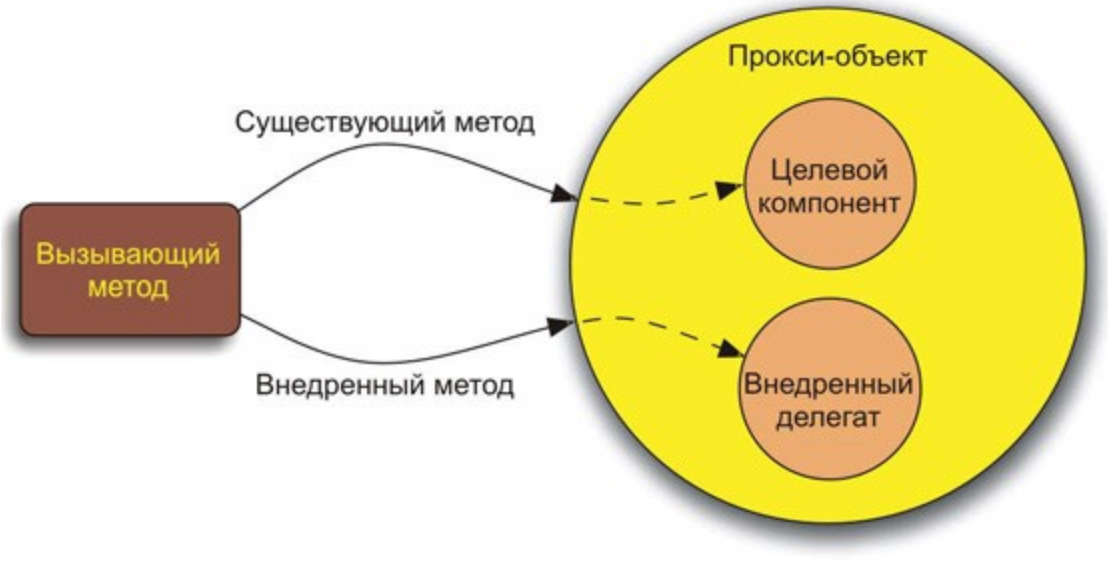

Рисунок 7 — С помощью Spring AOP можно внедрять новые методы в компоненты. Прокси-объект перехватывает вызовы и делегирует их другому объекту, реализующему новый метод

На рисунке 7 видно, что при вызове метода внедренного интерфейса прокси-объект делегирует вызов некоторому другому объекту, содержащему

реализацию нового интерфейса. Фактически это позволяет разбить реализацию компонента на несколько классов.

Попробуем воплотить эту идею на практике. Представим, что нам потребовалось внедрить следующий интерфейс Contestant во все компоненты, представляющие участников конкурса в примере:

```
package com.springinaction.springidol;
public interface Contestant {
    void receiveAward();
}
```

Можно было бы обойти все реализации интерфейса Performer и изменить их, добавив реализацию интерфейса Contestant (соперник). Но в общем и целом это может быть не самый благоразумный шаг (потому что Contestant (соперник) и Performer (исполнитель) – необязательно взаимоисключающие понятия). Более того, иногда может оказаться просто невозможным изменить все реализации интерфейса Performer, особенно когда в приложении используются сторонние реализации и отсутствуют их исходные тексты.

К счастью, возможность внедрения, поддерживаемая AOP, в состоянии помочь решить эту проблему без изменения архитектурных решений и не требуя изменять существующие реализации. Для осуществления описанной задачи необходимо задействовать элемент <aop:declare-parents>:

```
<aop:aspect>
<aop:declare-parents
        types-matching="com.springinaction.springidol.Performer+"
        implement-interface="com.springinaction.springidol.Contestant"
        default-impl="com.springinaction.springidol.GraciousContestant"
        />
</aop:aspect>
```

Как следует из его имени, элемент <aop:declare-parents> объявляет, что компоненты, к которым применяется описываемый аспект, приобретают новых родителей в иерархии наследования. В частности, в данном случае утверждается, что компоненты, чьи типы совместимы с интерфейсом Performer (определяется атрибутом types-matching), получают в иерархии интерфейс Contestant (определяется атрибутом implement-interface) в иерархии наследования. Последний атрибут элемента описывает, где находятся реализации методов интерфейса Contestant.

Существуют два способа определения реализации внедренного интерфейса. В данном случае был использован атрибут default-impl , явно определяющий реализацию посредством полного имени класса. Другой способ заключается в использовании атрибута delegate-ref :

```
<aop:declare-parents
        types-matching="com.springinaction.springidol.Performer+"
        implement-interface="com.springinaction.springidol.Contestant"
        delegate-ref="contestantDelegate"
```

/>

Атрибут delegate-ref ссылается на компонент Spring, играющий роль внедренного делегата. Он предполагает существование компонента с идентификатором contestantDelegate в контексте Spring:

```
<bean id="contestantDelegate"
 class="com.springinaction.springidol.GraciousContestant" />
```

Разница между двумя способами определения делегата с по мощью атрибутов default-impl и delegate-ref заключается в том, что во втором случае компонент Spring сам может быть субъектом внедрения, применения аспектов и воздействия других механизмов Spring.

1.4 Аннотирование аспектов

Важной особенностью, появившейся в AspectJ 5, стала возможность использовать аннотации для создания аспектов. До выхода версии AspectJ 5 для создания аспектов AspectJ требовалось знать синтаксис расширения языка Java. Однако появление возможности использования аннотаций AspectJ свело преобразование любых классов в аспекты к простому добавлению в них нескольких аннотаций. Эту новую возможность часто называют как @AspectJ .

Взглянув еще раз на класс Audience, можно заметить, что он содержит всю необходимую функциональность, реализующую поведение публики в зале, но в нем отсутствует что-либо, превращающее его в аспект. По этой причине нам пришлось объявлять совет и срез множества точек сопряжения в конфигурационном XML-файле.

С помощью аннотаций @AspectJ можно превратить класс Audience в аспект, не прибегая к созданию дополнительных классов или объявлению компонентов. В листинге 5.6 представлен новый класс Audience, на этот раз преобразованный в аспект с помощью аннотаций.

Листинг 6. Преобразование класса Audience в аспект с помощью аннотаций

```
package com.springinaction.springidol;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {
        @Pointcut( // Определение среза
                "execution(* com.springinaction.springidol.Performer.perform(..))")
        public void performance() {
        }
```

```
    @Before("performance()")
    public void takeSeats() { // Перед выступлением
        System.out.println("The audience is taking their seats.");
    }

    @Before("performance()") // Перед выступлением
    public void turnOffCellPhones() {
        System.out.println("The audience is turning off their cellphones");
    }

    @AfterReturning("performance()") // После успешного выступления
    public void applaud() {
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }

    @AfterThrowing("performance()")
    public void demandRefund() { // После неудачного выступления
        System.out.println("Boo! We want our money back!");
    }
}
```

Теперь новый класс Audience отмечен аннотацией @Aspect . Эта аннотация указывает, что Audience является не обычным простым Java-объектом, а аспектом.

Аннотация @Pointcut используется в @AspectJ для определения среза множества точек сопряжения. Значение параметра аннотации @Pointcut является выражением AspectJ. Здесь выражение указывает, что аспект будет подключаться к точкам сопряжения, соответствующим методу perform() интерфейса Performer. Имя среза определяется именем метода, к которому применяется аннотация. То есть данный срез имеет имя performance(). Фактическое тело метода performance() не имеет значения и в действительности должно быть пустым. Сам метод служит лишь точкой подключения аннотации @Pointcut.

Все методы класса Audience отмечены аннотациями, определяющими советы. Аннотация @Before применяется к методам takeSeats() и turnOffCellPhones(), указывая, что эти два метода являются советами, выполняемыми перед вызовом целевого метода. Аннотация @AfterReturning указывает, что метод applaud() является советом, выполняемым в случае благополучного завершения целевого метода. И аннотация @AfterThrowing перед методом demandRefund() указывает, что он будет вызываться в случае появления исключительных ситуаций в процессе выступления.

Всем аннотациям советов передается строка с именем среза множества точек сопряжения performance(), определяющая точки применения советов.

Обратите внимание, что кроме добавления аннотаций и пустого метода performance() реализация класса Audience осталась прежней. Это означает, что он все еще остается простым Java-объектом и может использоваться в этом его

качестве. Он также может быть связан в конфигурации Spring, как показано ниже:

```
<bean id="audience"
        class="com.springinaction.springidol.Audience" />
```

Поскольку класс Audience содержит собственные определения среза множества точек сопряжения и советов, больше нет необходимости объявлять их в конфигурационном XML-файле. Последнее, что осталось сделать, – это применить Audience как аспект. Для этого необходимо объявить в контексте Spring компонент, реализующий автоматическое проксирование, который знает, как превратить компоненты, отмеченные аннотациями @AspectJ, в советы.

Для этой цели в состав Spring входит класс AnnotationAwareAspectJAu toProxyCreator , создающий объекты автоматического проксирования. Класс AnnotationAwareAspectJAutoProxyCreator можно зарегистрировать в контексте Spring с помощью элемента <bean>, но это потребует большого объема ввода с клавиатуры (уж поверьте... мне приходилось делать это несколько раз). Чтобы упростить эту задачу, фреймворк Spring предоставляет собственный конфигурационный элемент в пространстве имен aop , гораздо более простой в запоминании, нежели это длинное имя класса:

```
<aop:aspectj-autoproxy />
```

Элемент <aop:aspectj-autoproxy/> создаст в контексте Spring компонент класса AnnotationAwareAspectJAutoProxyCreator и автоматически выполнит проксирование компонентов, имеющих методы, совпадающие с объявлениями точек сопряжения в аннотациях @Pointcut, которые присутствуют в компонентах, отмеченных аннотацией @Aspect.

Перед использованием элемента <aop:aspectj-autoproxy> нужно не забыть включить пространство имен aop в конфигурационном файле Spring:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

Вы должны понимать, что в качестве руководства к действию при создании аспектов элемент <aop:aspectj-autoproxy> использует только аннотации @AspectJ. В действительности эти аспекты все еще остаются аспектами Spring, а это означает, что, несмотря на использование аннотаций @AspectJ, советы могут применяться только к вызовам методов. Если необходимы дополнительные возможности AspectJ, следует использовать механизм AspectJ времени выполнения и не полагаться на возможность фреймворка Spring создавать аспекты на основе прокси-объектов.

Важно также отметить, что и элемент <aop:aspect>, и аннотации @AspectJ обеспечивают эффективные способы превращения POJO в аспекты. Но элемент <aop:aspect> обладает одним важным преимуществом перед аннотациями @AspectJ – он не требует изменения исходного программного кода класса для преобразования его в аспект. Для применения аннотаций @AspectJ к классам и методам необходимо иметь исходные тексты, тогда как элемент <aop:aspect> может ссылаться на любой компонент.

Теперь посмотрим, как с помощью аннотаций @AspectJ создать совет, выполняемый и до, и после вызова целевого метода.

1.4.1 Создание советов, выполняемых и до, и после

Как и в случае настройки аспектов в XML-файле конфигурации Spring, при использовании аннотаций @AspectJ можно создавать не только советы, выполняемые до или после вызова целевого метода, но и советы, выполняемые и до, и после вызова. Для этого следует использовать аннотацию @Around, как показано в следующем примере:

```
@Around("performance()")
public void watchPerformance(ProceedingJoinPoint joinpoint) {
        try {
                System.out.println("The audience is taking their seats.");
                System.out.println("The audience is turning off their cellphones");

                long start = System.currentTimeMillis();
                joinpoint.proceed();
                long end = System.currentTimeMillis();

                System.out.println("CLAP CLAP CLAP CLAP CLAP");

                System.out.println("The performance took " + (end — start)
                        + " milliseconds.");
        } catch (Throwable t) {
                 System.out.println("Boo! We want our money back!");
        }
}
```

Здесь аннотация @Around указывает, что метод watchPerformance() используется как совет, выполняемый и до, и после вызова целевого метода, соответствующего срезу множества точек сопряжения performance(). Этот фрагмент может показаться до боли знакомым, что неудивительно, так как это тот же самый метод watchPerformance(), который мы видели выше. Только на этот раз он отмечен аннотацией @Around.

Как вы помните, методы советов, выполняемых и до, и после вызова целевого метода, должны явно вызывать метод proceed(), чтобы выполнить целевой метод. Но простого применения аннотации @Around к методу недостаточно, чтобы обеспечить вызов метода proceed(). Поэтому методы, реализующие советы, выполняемые и до, и после вызова целевого метода,

должны принимать в виде аргумента объект типа ProceedingJoinPoint и вызывать метод proceed() этого объекта.

1.4.2 Передача аргументов аннотированным советам

Передача параметров советам, созданным с помощью аннотаций @AspectJ, мало чем отличается от случая, когда аспекты объявляются в конфигурационном XML-файле Spring. В действительности XML-элементы, использовавшиеся выше, имеют прямые эквиваленты в виде аннотаций @AspectJ, как показано ниже, на примере нового класса Magician .

Листинг 7 Превращение класса Magician в аспект с помощью аннотаций @AspectJ

```
package com.springinaction.springidol;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
	public class Magician implements MindReader {
		private String thoughts;

// Объявление параметризованного среза множества точек сопряжения
			@Pointcut("execution(* com.springinaction.springidol."
				+ "Thinker.thinkOfSomething(String)) && args(thoughts)")
	public void thinking(String thoughts) {
	}

	@Before("thinking(thoughts)") // Передача параметра в совет
	public void interceptThoughts(String thoughts) {
		System.out.println("Intercepting volunteer’s thoughts : "
			+ thoughts);
		this.thoughts = thoughts;
	}

	public String getThoughts() {
		return thoughts;
	}
}
```

Элемент <aop:pointcut> является эквивалентом аннотации @Pointcut, а элемент <aop:before> – эквивалентом аннотации @Before. Единственное важное отличие состоит в том, что аннотации @AspectJ могут опираться на синтаксис

языка Java для объявления значений параметров, передаваемых советам. Поэтому здесь нет никакой потребности в наличии прямого эквивалента атрибута arg-names элемента <aop:before>.

1.4.3 Внедрение с помощью аннотаций

Выше было показано, как с помощью элемента <aop:declareparents> внедрить реализацию интерфейса в существующий компонент, не изменяя исходного программного кода. Теперь посмотрим на этот же пример с другой стороны, но на этот раз задействуем аннотации AOP.

Эквивалентом элемента <aop:declare-parents> в множестве аннотаций @AspectJ является аннотация @DeclareParents . Она действует практически так же, как и родственный ей XML-элемент, когда используется внутри класса, отмеченного аннотацией @Aspect. Листинг 5.8 демонстрирует использование аннотации @DeclareParents.

Листинг 8 Внедрение интерфейса Contestant с использованием аннотаций @AspectJ

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class ContestantIntroducer {

        @DeclareParents( // Внедрение интерфейса Contestant
                value = "com.springinaction.springidol.Performer+",
                defaultImpl = GraciousContestant.class)
        public static Contestant contestant;
}
```

Как видите, ContestantIntroducer – это аспект. Но в отличие от аспектов, создававшихся до сих пор, он не содержит никаких советов. Зато он внедряет интерфейс Contestant в компоненты Performer. Подобно элементу <aop:declare-parents>, аннотация @DeclareParents состоит из трех частей.

- Атрибут value является эквивалентом атрибуту types-matching элемента <aop:declare-parents>. Он определяет тип компонентов, в которые должен быть внедрен интерфейс.
- Атрибут defaultImpl является эквивалентом атрибуту defaultimpl элемента <aop:declare-parents>. Он определяет класс, реализующий внедряемый интерфейс.
- Статическое свойство, к которому применяется аннотация @DeclareParents, определяет внедряемый интерфейс.

Как и для любых других аспектов, класс ContestantIntroducer следует объявить компонентом в контексте приложения Spring:

<bean class="com.springinaction.springidol.ContestantIntroducer" />

Теперь, когда элемент <aop:aspectj-autoproxy> обнаружит компонент, отмеченный аннотацией @Aspect, он автоматически создаст прокси-объект, делегирующий вызовы методов либо целевому объекту, либо реализации внедренного интерфейса, в зависимости от принадлежности вызываемого метода.

Обратите внимание, что аннотация @DeclareParents не имеет эквивалента атрибуту delegate-ref элемента <aop:declare-parents>. Это обусловлено тем, что @DeclareParents является аннотацией @AspectJ. @AspectJ – это независимый от Spring проект, поэтому его аннотации не обеспечивают поддержку компонентов. Отсюда следует, что если реализация внедряемого интерфейса оформлена в виде компонента, объявленного в конфигурации Spring, аннотация @DeclareParents может оказаться бесполезной и вам придется прибегнуть к использованию элемента <aop:declare-parents>.

Поддержка AOP в Spring позволяет отделять сквозные задачи от основной логики приложения. Но, как было показано выше, аспекты Spring все еще основаны на применении прокси-объектов, и круг их применения ограничен вызовами методов. Если вам потребуется нечто большее, чем применение советов к методам, подумайте о возможности использовать расширение AspectJ. В следующем разделе будет показано, как в приложениях на основе Spring использовать традиционные аспекты AspectJ.

1.5 В заключение о AOP

AOP является мощным дополнением к объектно-ориентированному программированию. Аспекты позволяют группировать функциональные возможности в модули многократного использования, которые в противном случае оказались бы рассеяны по всему приложению. Вы можете точно указать, где и как должны использоваться эти функциональные возможности. Это позволяет избежать дублирования программного кода и при разработке прикладных классов сконцентрироваться на решении основной задачи.

Spring предоставляет фреймворк AOP, позволяющий окружать аспектами вызовы методов. Вы узнали, как вплетать советы до и/или после вызова метода, а также добавлять собственную обработку исключений.

Существуют несколько вариантов использования аспектов в приложениях на основе Spring. Определять советы и срезы множества точек сопряжения в Spring проще с помощью аннотаций @AspectJ и упрощенной схемы конфигурирования.

Наконец, бывают ситуации, когда для решения каких-либо задач возможностей Spring AOP оказывается недостаточно и приходится обращаться к расширению AspectJ для реализации более мощных аспектов. В таких ситуациях можно использовать Spring для внедрения зависимостей в аспекты AspectJ.

2 Знакомство с транзакциями

Транзакции позволяют объединить несколько операций в единый блок, в котором будут выполнены либо все операции, либо ни одной. Если все идет гладко, транзакция завершается успехом. Но если что-то пойдет не так, результаты операций, которые успеют выполниться к этому моменту, будут отброшены, как если бы ничего и не происходило.

Пожалуй, самым типичным примером транзакций в реальном мире может служить перевод денег. Перевод денег должен быть выполнен полностью или не выполнен вообще. Если операция списания будет выполнена успешно, а операция зачисления потерпит неудачу, вы потеряете свои деньги.

Ранее (в теме работы с БД) мы исследовали поддержку различных механизмов доступа к данным в Spring и познакомились с несколькими способами чтения и записи данных в базу данных. При записи данных необходимо гарантировать целостность данных, выполняя запись в рамках транзакции. Фреймворк Spring обладает широкими возможностями, обеспечивая как программное, так и декларативное управление транзакциями. В этой главе будет показано, как использовать транзакции в приложениях, чтобы в случае, когда все идет как надо, результаты работы не терялись.

Транзакции играют важную роль в программном обеспечении, гарантируя, что данные и ресурсы всегда будут оставаться в непротиворечивом состоянии. Без них было бы можно повредить данные или оставить их в состоянии, противоречащем логике работы приложения.

Прежде чем приступить к изучению поддержки транзакций в Spring, необходимо познакомиться с некоторыми ключевыми понятиями. Рассмотрим четыре фактора, которыми руководствуются транзакции, и посмотрим, как они действуют.

2.1 Описание транзакций в четырех словах

В лучших традициях разработки программного обеспечения была придумана аббревиатура, описывающая транзакции: ACID . Эта аббревиатура происходит от следующих слов.

Atomic (атомарность) – транзакции состоят из одной или более операций, объединенных в единицу работы. Атомарность гарантирует, что либо будут выполнены все операции, либо ни одна из них. Если все операции выполняются успешно, транзакция завершается успехом. Если какая-то операция терпит неудачу, вся транзакция терпит неудачу и отменяется.

Consistent (непротиворечивость) – после выполнения транзакции (независимо от успеха или неудачи) система остается в состоянии, не противоречащем бизнес-модели. Данные не должны повреждаться.

Isolated (изолированность) – транзакции должны позволять нескольким пользователям работать с одними и теми же данными, не мешая друг другу. То есть транзакции должны быть изолированы друг от друга, предотвращая

возможность одновременного чтения и записи одних и тех же данных. (Обратите внимание, что изолированность обычно связана с блокировкой строк и/или таблиц в базе данных.)

Durable (долговечность) – после выполнения транзакции результаты ее выполнения должны сохраняться, чтобы они не терялись в случае ошибок во время работы системы. Под долговечностью обычно понимается сохранение результатов в базе данных или каком-то другом хранилище. В примере с покупкой билета под гарантией атомарности транзакции понимается отмена всех результатов, если какая-то из операций потерпит неудачу. Атомарность обеспечивает непротиворечивость, гарантируя, что данные никогда не останутся в противоречивом, частично измененном состоянии. Изолированность также обеспечивает непротиворечивость, не позволяя другой, параллельной транзакции «украсть» у вас свободное место в ходе покупки билета.

Наконец, под долговечностью понимается сохранение результатов в некотором хранилище. В случае краха системы или других катастрофических событий вы не должны волноваться по поводу потери результатов транзакции.

2.1.1 Знакомство с поддержкой транзакций в Spring

Фреймворк Spring , как и EJB, предоставляет поддержку программного и декларативного управления транзакциями. Но возможности Spring в этом отношении намного шире, чем возможности EJB.

Поддержка программного управления транзакциями в Spring существенно отличается от аналогичной ей поддержки в EJB. В отличие от EJB, где используется реализация Java Transaction API (JTA), фреймворк Spring использует механизм обратных вызовов, изолирующий фактическую реализацию транзакций от программного кода, использующего ее. В действительности поддержка управления транзакциями в Spring даже не требует наличия реализации JTA. Если приложение использует только одно хранилище данных, Spring может использовать поддержку транзакций, предлагаемую самим механизмом хранения. В число поддерживаемых механизмов входят JDBC, Hibernate и Java Persistence API (JPA). Но если требования к транзакциям в приложении распространяются на несколько хранилищ, Spring может предложить поддержку распределенных транзакций на основе сторонней реализации JTA.

Программное управление транзакциями обеспечивает высочайшую гибкость и точность в определении границ транзакций, тогда как декларативное управление транзакциями (основанное на Spring AOP) помогает изолировать операции от правил применения транзакций. Поддержка декларативного управления транзакциями в Spring напоминает поддержку в EJB транзакций, управляемых контейнером (container-managed transactions, CMT). И та, и другая позволяют определять границы транзакций декларативно. Но в Spring декларативное управление транзакциями предоставляет более широкие

возможности, чем CMT, позволяя объявлять дополнительные атрибуты, например определяющие уровень изоляции и пределы времени ожидания.

Выбор между программным и декларативным управлением транзакциями в значительной степени определяется выбором между точностью управления и удобством использования. При программном управлении транзакциями приложение получает возможность точно определять границы транзакций, устанавливая начало и конец области действия транзакций. Обычно высокая точность определения границ транзакций не требуется, и поэтому чаще предпочтение отдается объявлению транзакций в файле определения контекста.

Независимо от выбранного способа управления транзакциями, программного или декларативного, в приложениях необходимо будет использовать диспетчер транзакций Spring, обеспечивающий интерфейс к конкретным реализациям транзакций. Посмотрим, насколько диспетчеры транзакций в Spring способны освободить программиста от необходимости взаимодействовать с конкретными реализациями транзакций.

2.2 Выбор диспетчера транзакций

Фреймворк Spring не осуществляет непосредственного управления транзакциями. Вместо этого в его состав входит набор диспетчеров транзакций , которые принимают на себя всю ответственность за управление конкретными реализациями транзакций, предоставляемых либо посредством JTA, либо механизмом хранения данных.

В состав Spring входят:

- jca.cci.connection.CciLocalTransactionManager — При использовании поддержки в Spring для работы с Java EE Connector Architecture (JCA) и Common Client Interface (CCI)
- jdbc.datasource.DataSourceTransactionManager —Для работы с поддержкой JDBC в Spring. Также можно использовать при работе с iBATIS
- jms.connection.JmsTransactionManager — При использовании JMS 1.1+
- jms.connection.JmsTransactionManager102 — При использовании JMS 1.0.2
- orm.hibernate3.HibernateTransactionManager — При использовании Hibernate 3
- orm.jdo.JdoTransactionManager — При использовании JDO
- orm.jpa.JpaTransactionManager — При использовании Java Persistence API (JPA)
- transaction.jta.JtaTransactionManager — При необходимости использовать распределенные транзакции или когда другие диспетчеры транзакций не соответствуют требованиям
- transaction.jta.OC4JJtaTransactionManager — При использовании контейнера Oracle OC4J JEE
- transaction.jta.WebLogicJtaTransactionManager — При необходимости использовать распределенные транзакции для работы с WebLogic

- transaction.jta.WebSphereUowTransactionManager — При необходимости использовать транзакции, управляемые компонентом UOWManager в WebSphere

Каждый из этих диспетчеров играет роль фасада для конкретной реализации. (Взаимосвязи между некоторыми диспетчерами транзакций и конкретными реализациями изображены на рисунке 8.) Это позволяет работать с транзакциями, не беспокоясь об особенностях каждой конкретной реализации.

Чтобы задействовать диспетчера транзакций, его необходимо объявить в контексте приложения. В данном разделе будет показано, как настраивать некоторые, наиболее широко используемые диспетчеры транзакций в Spring, начиная с диспетчера DataSourceTransactionManager, предоставляющего поддержку транзакций при работе с простыми механизмами хранения данных JDBC и iBATIS .

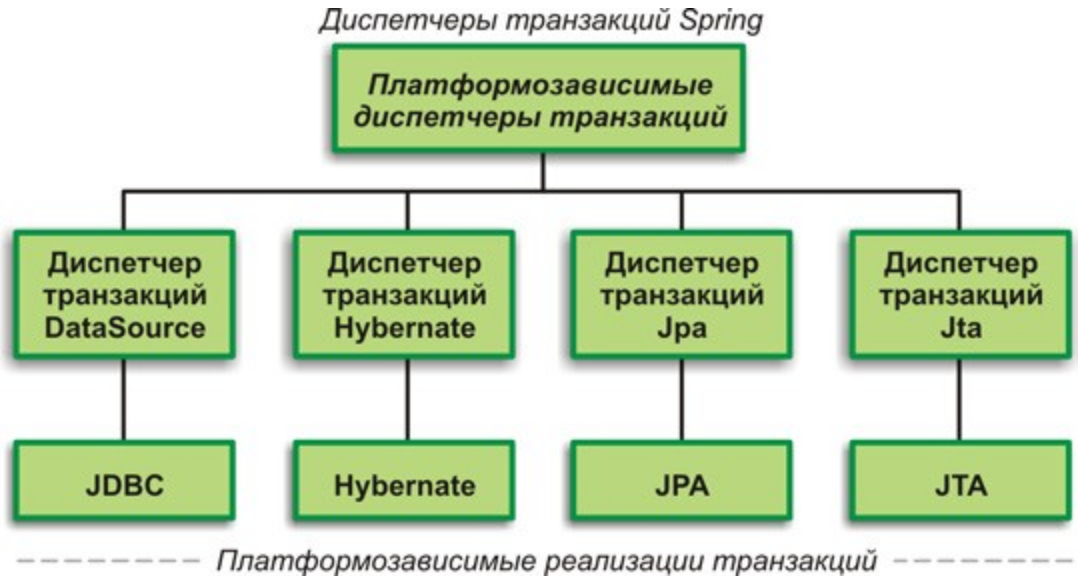

Рисунок 8 — Диспетчеры транзакций в Spring делегируют ответственность за управление транзакциями конкретным реализациям транзакций

2.2.1 Транзакции JDBC

Если для хранения данных в приложении предполагается использовать простой механизм JDBC, для управления транзакциями должен использоваться диспетчер DataSourceTransactionManager. Для этого необходимо добавить определение компонента DataSourceTransactionManager в контекст приложения, как показано ниже:

```
<bean id="transactionManager" class="org.springframework.jdbc.
                datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
</bean>
```

Обратите внимание, что в свойство dataSource записывается ссылка на компонент с именем dataSource. Очевидно, что компонент dataSource

представляет реализацию интерфейса javax.sql.DataSource и определен где-то в файле конфигурации контекста приложения.

За кулисами компонент DataSourceTransactionManager управляет транзакциями, выполняя вызовы методов объекта java.sql.Connection, полученного из компонента DataSource. Например, в случае успешного выполнения транзакция подтверждается вызовом метода commit() объекта соединения. Аналогично, в случае неудачи, отмена транзакции производится вызовом метода rollback() .

2.2.2 Транзакции Hibernate

Если для доступа к хранилищу данных приложение использует фреймворк Hibernate, тогда должен использоваться диспетчер HibernateTransactionManager. При работе с версией Hibernate 3 в определение контекста приложения необходимо добавить следующее определение элемента <bean>:

```
<bean id="transactionManager" class="org.springframework.
        orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Свойство sessionFactory должно быть связано с компонентом типа SessionFactory Hibernate, который в данном случае имеет недвусмысленное имя sessionFactory.

Диспетчер HibernateTransactionManager возлагает всю ответственность за управление транзакциями на объект org.hibernate.Transaction, который он получает из объекта сеанса Hibernate. В случае успешного выполнения транзакция подтверждается вызовом метода commit() объекта Transaction. Аналогично, в случае неудачи, отмена транзакции производится вызовом метода rollback() объекта Transaction .

2.2.3 Транзакции Java Persistence API

Фреймворк Hibernate уже в течение многих лет фактически является стандартным механизмом хранения данных в Java-приложениях, но совсем недавно на сцену вышла библиотека Java Persistence API (JPA), ставшая действительным стандартом в области хранения данных. Если вы готовы перейти на использование JPA, тогда для управления транзакциями вам потребуется диспетчер JpaTransactionManager . Ниже показано, как выполняется настройка компонента JpaTransactionManager в Spring:

```
<bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
<property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Диспетчеру JpaTransactionManager необходима только ссылка на фабрику диспетчера сущностей JPA (на любую реализацию интерфейса javax.persistence.EntityManagerFactory). Управление транзакциями

JpaTransactionManager будет осуществляться посредством объекта EntityManager, возвращаемого фабрикой. Помимо применения транзакций к операциям JPA, диспетчер JpaTransactionManager также поддерживает транзакции для простых операций JDBC в том же самом источнике данных DataSource, используемом фабрикой EntityManagerFactory. Чтобы воспользоваться этой поддержкой, диспетчера JpaTransactionManager необходимо также связать с реализацией интерфейса JpaDialect . Например, предположим, что в приложении используется компонент EclipseLinkJpaDialect, настроенный следующим образом:

```
<bean id="jpaDialect"
        class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect"  />
```

Тогда вам необходимо внедрить компонент jpaDialect в компонент JpaTransactionManager, как показано ниже:

```
<bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
<property name="entityManagerFactory" ref="entityManagerFactory" />
<property name="jpaDialect" ref="jpaDialect" />
</bean>
```

Важно отметить, что при этом реализация JpaDialect должна поддерживать смешанный JPA/JDBC доступ к данным. Все реализации JpaDialect (EclipseLinkJpaDialect, HibernateJpaDialect, OpenJpaDialect и TopLinkJpaDialect), входящие в состав Spring, обеспечивают такую смешанную поддержку механизмов JPA и JDBC. А реализация DefaultJpaDialect – нет.

2.3 Программное управление транзакциями в Spring

Существуют два типа людей: любители покомандовать и те, кто командовать не любит. Любители покомандовать стремятся контролировать все и вся, и никому не доверяют. Если вы – разработчик и любите покомандовать, вы наверняка относитесь к числу людей, предпочитающих командную строку, и пишете свои методы доступа к свойствам, не доверяя эту работу среде разработки.

Любителям командовать также нравится точно знать, что происходит в их программном коде. Когда дело доходит до транзакций, они предпочитают полностью контролировать точку запуска транзакции, точку ее подтверждения и завершения. Декларативное управление транзакциями оказывается недостаточно точным инструментом для них.

Впрочем, это не так уж и плохо. Любители покомандовать отчасти правы. Как будет показано ниже в этой главе, декларативное управление транзакциями ограничивается уровнем методов. Если потребуется более точное управление границами транзакций, этого можно будет добиться, используя только

программное управление транзакциями. В качестве примера метода, выполняемого в рамках транзакции, возьмем метод saveStudent() класса StudentServiceImpl, представленный ниже.

```
public void saveStudent(Student student) {
        studentDao.saveStudent(student);
}
```

Несмотря на кажущуюся простоту, в этом методе могут выполняться довольно сложные операции, невидимые на первый взгляд. В процессе сохранения объекта Student механизм хранения данных может выполнять множество действий. Даже если сохранение сводится к простой вставке строки в таблицу базы данных, важно убедиться, что все операции выполняются в рамках транзакции. Если все операции будут выполнены успешно, транзакцию можно подтвердить. Если что-то пойдет не так, ее можно отменить.

Один из подходов к включению транзакций в работу заключается в добавлении транзакции программно, непосредственно внутри метода saveStudent(), с использованием класса шаблона TransactionTemplate . Подобно другим классам шаблонов в Spring (таким как класс JdbcTemplate, обсуждавшийся в предыдущей главе), TransactionTemplate использует механизм обратных вызовов. В листинге ниже приводится измененная версия метода saveStudent(), демонстрирующая, как добавлять транзакции с помощью TransactionTemplate.

```
public void saveStudent(final Student student) {
        txTemplate.execute(new TransactionCallback<Void>() {
                public Void doInTransaction(TransactionStatus txStatus) {
                        try {
                                studentDao.saveStudent(student);
                        } catch (RuntimeException e) {
                                txStatus.setRollbackOnly();
                                throw e;
                        }
                        return null;
                }
        });
}
```

Чтобы иметь возможность использовать класс TransactionTemplate, необходимо сначала реализовать интерфейс TransactionCallback. Так как интерфейс TransactionCallback определяет единственный метод, часто бывает проще реализовать его в виде анонимного вложенного класса, как показано в коде выше. А программный код, который должен выполняться в рамках транзакции, должен находиться внутри метода doInTransaction().

Вызов метода execute() экземпляра класса TransactionTemplate выполнит программный код внутри экземпляра TransactionCallback. В случае появления проблемы будет вызван метод setRollbackOnly() объекта TransactionStatus, чтобы отменить транзакцию. В противном случае, если метод doInTransaction() благополучно вернул управление, транзакция будет подтверждена. А откуда здесь возьмется экземпляр TransactionTemplate? Хороший вопрос. Он должен быть внедрен в компонент StudentServiceImpl, как показано ниже:

```
<bean id="studentService"
        class="ru.sfu.StudentServiceImpl,">
...
        <property name="transactionTemplate ">
                <bean class=”org.springframework.transaction.support.
                        TransactionTemplate”>
                        <property name=”transactionManager”
                                ef=”transactionManager” />
                </bean>
        </property>
</bean>
```

Обратите внимание на наличие в компоненте TransactionTemplate свойства transactionManager. За кулисами компонент TransactionTemplate использует реализацию интерфейса PlatformTransactionManager для обслуживания особенностей транзакций для конкретной платформы. Здесь в свойство внедряется ссылка на компонент с именем transactionManager, который может быть любым диспетчером транзакций.

Программное управление транзакциями отлично подходит для ситуаций, когда требуется иметь полный контроль над границами транзакций. Но, как видно из кода выше, это достаточно утомительно. Необходимо изменить реализацию метода saveStudent() – использовать классы из фреймворка Spring, чтобы ввести в действие поддержку программного управления транзакциями в Spring.

Чаще всего в приложениях не требуется такая точность управления границами транзакций. Именно по этой причине обычно предпочитают объявлять транзакции за пределами программного кода (в конфигурационном файле Spring, например).

2.4 Декларативное управление транзакциями

Еще совсем недавно декларативное управление транзакциями было доступно только в контейнерах EJB. Но сейчас Spring предлагает аналогичную поддержку для POJO. Это – важная особенность Spring, потому что теперь для декларативного обеспечения атомарности операций не требуется создавать контейнеры EJB.

Поддержка декларативного управления транзакциями в Spring реализована посредством фреймворка Spring AOP. Это вполне естественно, потому что транзакции – это системная служба, стоящая уровнем выше приложения. Транзакции в Spring можно интерпретировать как аспекты, «обертывающие» методы.

Фреймворк Spring предоставляет три способа объявления границ транзакций. Исторически фреймворк Spring всегда обладал поддержкой декларативного управления транзакциями за счет проксирования компонентов с использованием Spring AOP и TransactionProxyFactoryBean. Но, начиная с версии Spring 2.0, появился более удобный способ объявления транзакций, основанный на использовании конфигурационного пространства имен tx и аннотации @Transactional.

Несмотря на то что класс TransactionProxyFactoryBean все еще доступен в современных версиях Spring, он считается устаревшим и потому не будет рассматриваться здесь. Вместо этого мы сконцентрируемся на использовании пространства имен tx и объявлении транзакций с помощью аннотации. Но сначала исследуем определение атрибутов транзакций.

2.4.2 Определение атрибутов транзакций

В Spring декларативные транзакции определяются с помощью атрибутов транзакций . Атрибуты транзакции – это описание особенностей применения транзакции к методу. Всего имеется пять различных атрибутов транзакции, как показано на рисунке 9.

Рисунок 9 — Декларативные транзакции определяются в терминах правил распространения, уровня изоляции, признака «только для чтения», предельного времени ожидания и правил отмены

Фреймворк Spring предоставляет несколько механизмов объявления транзакций, однако все они опираются на эти пять параметров, управляющих поведением транзакций. Поэтому, чтобы научиться объявлять транзакции в Spring, важно разобраться с этими параметрами.

Независимо от используемого механизма декларативных транзакций у вас всегда будет возможность определить данные атрибуты. Исследуем каждый атрибут отдельно и посмотрим, какое влияние они оказывают на транзакции.

Правила распространения

Первый атрибут, который мы рассмотрим, определяет правила распространения транзакции. Этот атрибут назначает границы транзакции. Всего фреймворк Spring определяет семь разных правил распространения:

- *PROPAGATION_MANDATORY* Указывает, что метод должен выполняться внутри транзакции. Если к моменту вызова метода не будет запущена транзакция, фреймворк возбудит исключение
- *PROPAGATION_NESTED* Указывает, что метод должен выполняться внутри вложенной транзакции, если к моменту вызова метода уже была запущена транзакция. Вложенную транзакцию можно подтвердить или отменить независимо от вмещающей транзакции. При отсутствии вмещающей транзакции это правило действует подобно правилу
- *PROPAGATION_REQUIRED.* Поддержка этого правила различными фреймворками хранения данных в лучшем случае является неполной. Чтобы выяснить наличие поддержки вложенных транзакций используемым фреймворком, обращайтесь к его документации
- *PROPAGATION_NEVER* Указывает, что метод не должен выполняться внутри транзакции. Если к моменту вызова метода будет запущена транзакция, фреймворк возбудит исключение
- *PROPAGATION_NOT_SUPPORTED* Указывает, что метод не должен выполняться внутри транзакции. Если к моменту вызова метода будет запущена транзакция, фреймворк приостановит ее на время выполнения метода. При использовании JTATransactionManager необходим доступ к TransactionManager
- *PROPAGATION_REQUIRED* Указывает, что метод должен выполняться внутри транзакции. Если к моменту вызова метода будет запущена транзакция, он будет выполняться в рамках этой транзакции. В противном случае будет запущена новая транзакция
- *PROPAGATION_REQUIRES_NEW* Указывает, что метод должен выполняться внутри собственной транзакции. Будет запущена новая транзакция, и, если к моменту вызова метода уже будет запущена другая транзакция, фреймворк приостановит ее на время выполнения метода. При использовании JTATransactionManager необходим доступ к TransactionManager
- *PROPAGATION_SUPPORTS* Указывает, что метод не требует наличия транзакции, но он может выполняться внутри имеющейся транзакции

Правила распространения транзакций, перечисленные выше, кому-то могут показаться знакомыми. И это неудивительно, потому что они отражают правила распространения транзакций, управляемых контейнером, в EJB (container-managed transactions, CMT). Например, правило PROPAGATION_REQUIRES_NEW в Spring эквивалентно правилу RequiresNew

в CMT. Однако в Spring добавлено одно дополнительное правило распространения, отсутствующее в CMT, – PROPAGATION_NESTED, обеспечивающее поддержку вложенных транзакций.

Правила распространения отвечают на вопрос: «должна ли быть запущена новая или приостановлена выполняющаяся транзакция» или «должен ли метод выполняться в контексте транзакции».

Например, если метод объявлен как выполняющийся в рамках транзакции с правилом PROPAGATION_REQUIRES_NEW, это означает, что границы транзакции совпадают с границами метода: перед вызовом метода запускается новая транзакция, а когда метод возвращает управление или возбуждает исключение, транзакция завершается. В случае использования правила PROPAGATION_REQUIRED границы транзакции будут зависеть от того, была ли запущена другая транзакция.

Уровни изоляции

Второй атрибут, описывающий поведение декларативной транзакции, определяет уровень изоляции. Уровень изоляции указывает, насколько транзакция подвержена влиянию других транзакций, выполняющихся параллельно. Уровень изоляции транзакции можно также представить как меру эгоизма транзакции по отношению к данным, используемым в рамках транзакции.

В типичных приложениях транзакции, выполняющиеся параллельно, часто работают с одними и теми же данными. Параллельное выполнение транзакций может привести к следующим проблемам.

Чтение неподтвержденных данных – когда одна транзакция читает данные, записанные, но не подтвержденные другой транзакцией. Если позднее другая транзакция будет отменена, данные, полученные первой транзакцией, окажутся недействительными.

Неповторимость получаемых результатов – когда транзакция несколько раз выполняет один и тот же запрос и каждый раз получает разные данные. Обычно это обусловлено действием других транзакций, успевающих внести изменения между выполнением запросов.

Чтение фантомных данных – напоминает неповторимость получаемых результатов. Эта ситуация может возникать, когда транзакция (T1) читает несколько строк, а затем вторая транзакция (T2), выполняющаяся параллельно, вставляет несколько строк. При выполнении последующих запросов первая транзакция (T1) будет обнаруживать дополнительные строки, отсутствовавшие прежде.

В идеале, чтобы избежать подобных проблем, транзакции должны быть полностью изолированы друг от друга. Но полная изоляция может отрицательно сказаться на производительности, потому что часто для этого необходимо блокировать доступ к строкам (а иногда и к таблицам целиком) в хранилище данных. Агрессивное использование блокировок может

воспрепятствовать параллельному выполнению транзакций, вынуждая их ждать, пока другие транзакции не выполнят свою работу.

Учитывая, что полная изоляция может влиять на производительность и она необходима далеко не во всех приложениях, иногда бывает желательно иметь более гибкую возможность изолирования транзакций. Поэтому транзакции поддерживают несколько уровней изоляции:

ISOLATION_DEFAULT Используется уровень изоляции по умолчанию, зависящий от конкретного хранилища данных

ISOLATION_READ_UNCOMMITTED Позволяет читать неподтвержденные изменения. Может привести к проблеме чтения неподтвержденных данных, неповторимости получаемых результатов и чтения фантомных данных

ISOLATION_READ_COMMITTED Позволяет читать подтвержденные изменения, выполненные в других, параллельно выполняющихся транзакциях. Устраняет проблему чтения неподтвержденных данных, но проблемы неповторимости получаемых результатов и чтения фантомных данных все еще могут возникать

ISOLATION_REPEATABLE_READ Многократные попытки чтения одного и того же поля будут возвращать одни и те же результаты, если эти поля не будут изменяться в пределах самой транзакции. Устраняет проблемы чтения неподтвержденных данных и неповторимости получаемых результатов, но проблема чтения фантомных данных все еще может возникать

ISOLATION_SERIALIZABLE Этот уровень изоляции полностью соответствует определению ACID и устраняет проблемы чтения неподтвержденных данных, неповторимости получаемых результатов и чтения фантомных данных. Этот уровень изоляции снижает производительность приложения в большей степени, чем все остальные, потому что такая изоляция достигается за счет полного блокирования доступа к таблицам, используемым в транзакции

Наибольшую производительность обеспечивает уровень изоляции ISOLATION_READ_UNCOMMITTED, но он в наименьшей степени изолирует транзакции друг от друга, оставляя возможность появления проблем чтения неподтвержденных данных, неповторимости получаемых результатов и чтения фантомных данных. Полной его противоположностью является уровень изоляции ISOLATION_SERIALIZABLE, предотвращающий все эти проблемы, но он оказывает самое сильное отрицательное влияние на производительность.

Помните, что не все источники данных поддерживают все уровни изоляции, перечисленные выше. Обязательно обращайтесь к соответствующей документации, чтобы выяснить, какие уровни изоляции поддерживаются.

Только для чтения

Третьим параметром объявления транзакции является признак доступа к данным только для чтения . На тот случай, если транзакция будет выполнять

только операции чтения данных, хранилище может предусматривать некоторые оптимизации, учитывающие природу таких транзакций. Объявляя транзакцию как выполняющую только чтение данных, вы даете хранилищу данных возможность применить эти оптимизации.

Поскольку оптимизация доступа к данным только для чтения выполняется хранилищем данных в начале транзакции, этот признак имеет смысл использовать лишь при объявлении транзакций для методов, для которых правило распространения допускает создание новой транзакции (PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW и PROPAGATION_NESTED).

Кроме того, при использовании механизма хранения данных Hibernate объявление транзакции как выполняющей только операции чтения, приведет к тому, что в Hibernate режим выталкивания получит значение FLUSH_NEVER, предписывающее фреймворку Hibernate избегать ненужной синхронизации объектов с базой данных, откладывая все обновления до конца транзакции.

Время ожидания

При нормальной работе приложения транзакции не могут выполняться в течение длительного времени. Поэтому следующей характеристикой в объявлениях транзакций является параметр, определяющий время ожидания .

Представьте, что транзакция неожиданно стала выполняться слишком долго. Поскольку выполнение транзакций может блокироваться хранилищем данных, долго выполняющиеся транзакции могут удерживать ресурсы базы данных неоправданно продолжительное время. Вместо того чтобы ждать, можно объявить, что транзакция должна автоматически отменять выполненные в ней операции по прошествии определенного количества секунд.

Поскольку отсчет времени ожидания начинается с момента запуска транзакции, объявлять предельное время ожидания имеет смысл только при объявлении транзакций для методов, для которых правило распространения допускает создание новой транзакции (PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW и PROPAGATION_NESTED).

Правила отмены

Последней гранью пятиугольника, изображенного на рисунке 2 является набор правил, определяющих, какие исключения должны приводить к отмене транзакции, а какие нет . По умолчанию транзакции отменяются только в случае появления исключений времени выполнения, но не в случае контролируемых исключений (checked exceptions). (Это соответствует правилам отмены, применяемым в EJB.)

Однако есть возможность объявить, что транзакция должна отменяться в ответ не только на исключения времени выполнения, но и на определенные контролируемые исключения. Аналогично можно объявить, что транзакция не должна отменяться в ответ на указанные исключения, даже если они являются

исключениями времени выполнения. Теперь, после знакомства с атрибутами транзакций, описывающими их поведение, можно посмотреть, как использовать эти атрибуты при объявлении транзакций в Spring.

2.4.1 Объявление транзакций в XML

В предыдущих версиях Spring объявление транзакций было связано с внедрением специального компонента TransactionProxyFactoryBean. Проблема с компонентом TransactionProxyFactoryBean состоит в том, что его использование ведет к разбуханию конфигурационных файлов Spring. К счастью, в настоящее время эта проблема была устранена, и теперь фреймворк Spring предлагает конфигурационное пространство имен tx, существенно упрощающее использование декларативных транзакций в Spring.

Чтобы использовать пространство имен tx, необходимо добавить его в конфигурационный XML-файл Spring:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx=”http://www.springframework.org/schema/tx”
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd http://
        www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd”>
```

Обратите внимание, что при этом также необходимо включить пространство имен aop . Это очень важно, потому что некоторые элементы настройки декларативных транзакций опираются на отдельные элементы настройки AOP.

Пространство имен tx содержит несколько новых конфигурационных XML-элементов, наиболее примечательным из которых является <tx:advice> . Следующий фрагмент XML-файла демонстрирует, как можно использовать элемент <tx:advice> для объявления транзакции, подобной той, что была определена ранее для службы Student:

```
<tx:advice id="txAdvice">
        <tx:attributes>
                <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
        </tx:attributes>
</tx:advice>
```

Атрибуты транзакции внутри элемента <tx:advice> определяются элементом <tx:attributes> с помощью одного или более элементов <tx:method> .

Элемент <tx:method> определяет атрибуты транзакции для метода (или методов), указанного в атрибуте name (допускается использование шаблонных символов). Элемент <tx:method> имеет несколько атрибутов, помогающих определять поведение транзакции.

Пять сторон пятиугольника поведения транзакции определяются атрибутами элемента <tx:method>:

- *isolation* Определяет уровень изоляции транзакции
- propagation Определяет правила распространения транзакции
- *read-only* Определяет режим доступа к данным
  Правила отмены:
- *rollback-for* определяет проверяемые исключения, при появлении которых транзакция должна отменяться.
- *no-rollback-for* определяет исключения, при появлении которых транзакция не должна отменяться
- *timeout* Определяет предельное время ожидания для транзакций, выполняющихся продолжительное время

Согласно определению совета txAdvice транзакции, методы, на которые распространяется эта транзакция, делятся на две категории: имена которых начинаются с add и все остальные. Метод saveStudent() попадает в первую категорию и объявляется как обязательно выполняющийся в рамках транзакции. Остальные методы определяются с правилом propagation="supports" – они не требуют выполнения в рамках транзакции, только если таковая существует.

При объявлении транзакции с помощью элемента <tx:advice> все еще необходим диспетчер транзакций, как при использовании компонента TransactionProxyFactoryBean. В случае с элементом <tx:advice> предполагается, что диспетчер транзакций будет объявлен как компонент с идентификатором transactionManager. Если в конфигурации диспетчеру транзакций присвоено другое имя (например, txManager), необходимо указать этот идентификатор в атрибуте transactionmanager:

```
<tx:advice id="txAdvice"
        transaction-manager="txManager">
 ...
</tx:advice>
```

Сам по себе элемент <tx:advice> определяет только совет аспекта для применения к методам, требующим выполнения в рамках транзакций. Но это лишь совет, а не полноценный аспект. Нигде в элементе <tx:advice> не указывается, к какому компоненту применяется этот совет, – для этого необходимо определить срез множества точек внедрения. Чтобы создать полное определение аспекта транзакции, следует определить объект-советник. Именно для этого и необходимо пространство имен aop. Следующий фрагмент

XMLфайла определяет объект-советник, использующий совет txAdvice для применения к любым компонентам, реализующим интерфейс StudentService:

```
<aop:config>
    <aop:advisor
        pointcut="execution(* *..StudentService.*(..))"
        advice-ref="txAdvice"/>
</aop:config>
```

В атрибуте pointcut используется выражение AspectJ выборки точек внедрения, указывающее, что этот объект-советник должен применить совет ко всем методам интерфейса StudentService. Перечень методов, выполняющихся в рамках транзакции, а также атрибуты транзакции определяются самим советом txAdvice, на который ссылается атрибут advice-ref совета.

Несмотря на то что элемент <tx:advice> несет немало удобств в создании декларативных транзакций, тем не менее в Spring 2 имеется еще одна возможность, которая выглядит еще более привлекательной для тех, кто работает в окружении Java 5. Посмотрим, как можно управлять транзакциями с помощью аннотаций.

2.4.2 Определение транзакций с помощью аннотаций

Применение элемента <tx:advice> позволяет значительно упростить объявление транзакций в конфигурационных XML-файлах Spring. А что, если я скажу, что можно добиться еще большего упрощения? Что, если я скажу, что достаточно добавить в определение контекста приложения всего одну строку, чтобы обеспечить объявление транзакций?

Помимо элемента <tx:advice>, пространство имен tx содержит также элемент <tx:annotation-driven>. Чтобы воспользоваться им, достаточно добавить в XML-файл всего одну строку:

```
<tx:annotation-driven />
```

Чтобы сделать пример немного интереснее, можно добавить определение компонента диспетчера транзакций с помощью атрибута transactionmanager (который по умолчанию ссылается на компонент с идентификатором transactionManager):

```
<tx:annotation-driven transaction-manager="txManager" />
```

Но это все, что можно добавить. Эта короткая строка в XML-файле обладает большими возможностями, позволяя определять транзакции там, где это более уместно: в методах, которые должны выполняться в рамках транзакций.

Аннотации – одна из самых крупных и самых обсуждаемых особенностей Java 5. Аннотации позволяют добавлять метаданные непосредственно в

программный код, а не во внешние конфигурационные файлы. Я полагаю, что они прекрасно подходят для объявления транзакций.

Элемент <tx:annotation-driven> сообщает фреймворку Spring проверить все компоненты в контексте приложения и отыскать отмеченные аннотацией @Transactional на уровне определения класса или на уровне методов. К каждому компоненту с аннотацией @Transactional элемент <tx:annotation-driven> автоматически применит совет с определением транзакции. Атрибуты транзакции в этом случае определяются параметрами аннотации @Transactional.

Например, далее демонстрируется объявление класса StudentServiceImpl, дополненное аннотациями @Transactional.

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class StudentServiceImpl implements SptudentService {
...
        @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
        public void addStudent(Student student {
 ...
        }
...
}
```

Аннотация @Transactional на уровне определения класса StudentServiceImpl сообщает, что все его методы поддерживают возможность выполнения в рамках транзакций, выполняющихся в режиме «только для чтения». А аннотация на уровне метода saveStudent() сообщает, что этот метод обязательно должен выполняться в контексте транзакции.

2.5 Заключение о транзакциях

В заключение Транзакции занимают важное место в корпоративных приложениях, повышая их надежность. Они обеспечивают бескомпромиссность операций, предотвращая повреждение данных в случае непредвиденных событий. Они также поддерживают возможность параллельного выполнения, исключая возможность взаимовлияния потоков выполнения в многопоточных приложениях, оперирующих одними и теми же данными.

Фреймворк Spring поддерживает и программное, и декларативное управление транзакциями. В обоих случаях Spring ограждает вас от необходимости работать непосредственно с конкретными механизмами управления транзакциями, скрывая конкретные реализации за универсальным API.

Для поддержки декларативного управления транзакциями Spring использует собственный фреймворк AOP. Декларативные транзакции в Spring обладают теми же возможностями, что и конкурирующая платформа EJB CMT, позволяя определять не только правила распространения в POJO, но также

уровень изоляции, задействовать оптимизации, подразумеваемые режимом доступа «только для чтения», и правила отмены в ответ на конкретные исключения.

Эта глава продемонстрировала, как применять декларативные транзакции с использованием аннотаций модели программирования Java 5. С введением аннотаций в Java 5, чтобы обеспечить выполнение метода в рамках транзакции, достаточно просто пометить его соответствующей аннотацией.

Как было показано выше, Spring распространяет мощь декларативных транзакций на POJO. Это самое захватывающее нововведение, так как прежде декларативные транзакции были доступны только в EJB. Но декларативные транзакции – это лишь верхушка айсберга из всего того, что Spring может предложить для работы с POJO. В следующей главе будет показано, как Spring распространяет на POJO декларативную поддержку безопасности.