1 Знакомство с Spring Framework 5.0 и паттернами проектирования

Эта глава даст вам представление о фреймворке Spring, его модулях и об использовании паттернов проектирования, обусловивших успех Spring. Здесь будут описаны все главные модули фреймворка. Мы начнем с его основ, рассмотрим новые возможности, появившиеся в Spring 5.0, а также разберемся в паттернах проектирования главных модулей фреймворка.

Прочитав эту главу, вы поймете, как Spring работает и решает распространенные проблемы проектирования корпоративных приложений с помощью паттернов проектирования. Вы узнаете, как улучшить слабую связанность между компонентами приложения и упростить создание приложений, используя Spring и паттерны проектирования.

В этой главе:

- знакомство с фреймворком Spring;
- упрощение разработки приложений благодаря применению Spring и паттернов;
  - использование широчайших возможностей паттерна POJO;
  - внедрение зависимостей;
  - применение аспектов в сквозных задачах;
  - использование паттерна «Шаблонный метод» для избавления от стереотипного кода;
- создание контейнера Spring для компонентов с помощью паттерна «Фабрика»;
  - разработка контейнера с контекстом приложения;
  - жизнь компонента в контейнере;
- модули Spring;
- новые возможности Spring Framework 5.0.

Знакомство с фреймворком Spring

В первые годы существования Java было разработано множество тяжеловесных технологий для создания корпоративных приложений. Однако поддерживать последние было нелегко ввиду их тесной связи с конкретным фреймворком. Пару лет

назад все Java-технологии, кроме Spring, были достаточно тяжеловесны, как EJB. В то время Spring предлагался как альтернативная технология, специально созданная для EJB, так как Spring предоставлял очень простую, гораздо более гибкую и легкую, по сравнению с другими существовавшими Java-технологиями, модель программирования.

Широкие возможности Spring достигаются благодаря использованию множества паттернов проектирования, но главной стала *POJO-модель программирования* (Plain Old Java Object, «старый добрый объект Java»). Она обеспечила простоту фреймворка Spring и, кроме того, предоставила функционал таких концепций, как паттерн *внедрения зависимостей* (DI) и *аспектно-ориентированное программирование* (AOP), благодаря использованию паттернов «Заместитель» и «Декоратор».

Spring Framework — это фреймворк с открытым исходным кодом и основанная на Java платформа, предоставляющая полную поддержку инфраструктуры для создания корпоративных Java-приложений. Таким образом, разработчики не должны думать об инфраструктуре приложения и могут сконцентрироваться на его бизнес-логике, а не конфигурации. Все файлы инфраструктуры, конфигурации и метаконфигурации, использующие Java или XML, обрабатываются фреймворком Spring. Так, при создании приложения с помощью модели программирования POJO он обеспечивает бóльшую гибкость, чем при использовании неагрессивной модели программирования.

IoC-контейнер Spring (Inversion of Control, инверсия управления) — ядро всего фреймворка. Он позволяет объединять в единое целое разные части приложения, обеспечивая согласованность архитектуры. Компоненты Spring MVC могут использоваться для формирования очень гибкого веб-уровня. IoC-контейнер облегчает разработку на бизнес-уровне с помощью POJO.

Spring упрощает создание приложений и часто устраняет зависимость от других API. Рассмотрим несколько ситуаций, в которых вы как разработчик можете выиграть от использования платформы Spring.

- Все классы в приложении являются простыми POJO-классами — Spring неагрессивен. В большинстве ситуаций он не требует от вас наследоваться от классов фреймворка или реализовывать его интерфейсы.
- Приложения, использующие Spring, не требуют наличия сервера приложений Java EE, но могут быть развернуты на нем.
- В транзакции с обращением к базе данных методы могут выполняться с помощью управления транзакциями Spring, без использования сторонних транзакционных API.
- Благодаря Spring можно использовать методы Java как обработчики запросов или удаленные методы, такие как метод `service()` из API сервлетов, без взаимодействия с API контейнера сервлетов.
- Spring позволяет использовать локальные методы `java` как обработчики сообщений без применения в приложении API сервиса сообщений *Java Message Service (JMS)*.

- Spring также позволяет использовать локальные методы `java` как операции управления без применения в приложении API управленческих расширений *Java Management Extensions (JMX)*.
- Spring выступает в роли контейнера для объектов приложения. Объекты не должны сами находить и *устанавливать* связи между собой.
- Spring создает компоненты и внедряет зависимости между объектами приложения, таким образом управляя жизненным циклом компонентов.

Упрощение разработки приложений благодаря применению Spring и паттернов

При разработке коммерческого приложения на традиционной платформе Java, когда речь заходит об организации повторного использования «блоков»-компонентов, возникает большое количество ограничений. Нельзя не учитывать, что повторное применение компонентов основной и общей функциональности — общепринятый стандарт проектирования. Для решения задачи повторного использования кода можно задействовать различные паттерны проектирования, такие как «Фабрика» (Factory), «Абстрактная фабрика» (Abstract Factory), «Строитель» (Builder), «Декоратор» (Decorator), «Локатор служб» (Service Locator), и составлять из базовых «кирпичиков» согласованное целое, класс или объект, таким образом обеспечивая повторное использование составляющих частей. Эти паттерны, решающие общие и рекурсивные задачи, реализованы внутри фреймворка Spring. Таким образом, он предоставляет инфраструктуру с четко определенным способом применения.

Написание корпоративных приложений — непростое дело, но Spring, специально созданный для борьбы с этими сложностями, упрощает задачу разработчиков. Применение Spring не ограничивается стороной сервера — он упрощает также сборку проекта, тестирование и слабое сцепление. Spring использует концепцию POJO, то есть компонент Spring может быть Java-объектом любого типа. Компонент — самодостаточный участок кода, который в идеале можно использовать повторно в разных приложениях.

Поскольку эта книга в основном посвящена всем *паттернам проектирования*, применяемым в Spring для упрощения разработки на Java, я объясняю или хотя бы описываю реализацию и задачи паттернов и наилучшие способы использования инфраструктуры для создания приложений. Фреймворк Spring использует следующие методы для упрощения разработки и тестирования:

- применяет *паттерн POJO* для легкой и минимально агрессивной разработки корпоративных приложений;
- задействует *паттерн внедрения зависимостей (DI)* для слабого сцепления и делает систему интерфейс-ориентированной;

- прибегает к *паттернам «Декоратор» и «Заместитель»* для декларативного программирования с помощью аспектов и общепринятых соглашений;
- применяет *паттерн «Шаблонный метод»* для устранения стереотипного кода с помощью аспектов и шаблонов.

В текущей главе я объясню все эти понятия и приведу конкретные примеры того, как Spring упрощает разработку на Java. Сначала разберемся, как Spring остается минимально агрессивным, поощряя проектирование, ориентированное на использование паттерна POJO.

Использование широчайших возможностей паттерна POJO

Существует множество фреймворков для разработки на Java, ограничивающих вас, заставляя расширять некоторые имеющиеся классы или реализовывать определенные интерфейсы. В частности, такого подхода придерживались Struts, Tapestry и ранние версии EJB. Эти фреймворки основаны на агрессивной модели программирования, которая усложняет поиск ошибок в системе, а порой делает код просто нечитаемым. Однако при работе с фреймворком Spring нет необходимости расширять готовые классы (реализовывать готовые интерфейсы) — реализация основана на паттерне POJO и следует неагрессивной модели программирования. Таким образом, искать ошибки в системе легче, а код остается понятным.

Spring позволяет программировать, задействуя простые не-Spring классы, и не заставляет применять специфичные для этого фреймворка классы и интерфейсы, так что все классы в Spring-приложении будут POJO. Таким образом, файлы могут быть скомпилированы и исполнены независимо от установленных библиотек Spring. Невозможно даже понять, что эти классы используются этим фреймворком. В худшем случае — при конфигурировании, основанном на Java, — вам придется прибегнуть к аннотациям Spring.

Поясню сказанное на следующем примере:

```
package com.packt.chapter1.spring;
public class HelloWorld {
  public String hello() {
    return "Hello World";
  }
}
```

Это просто POJO-класс без каких-либо указаний или особенностей реализации, относящихся к фреймворку и делающих его компонентом Spring. Он будет одинаково работать в приложениях, использующих и не использующих Spring. Этим и прекрасна неагрессивная модель программирования, применяемая Spring. Spring расширяет возможности POJO другим способом, организуя их взаимодействие с другими POJO с помощью паттерна внедрения зависимостей. Рассмотрим, как внедрение зависимостей позволяет разделить компоненты.

Внедрение зависимостей между POJO

Термин *«внедрение зависимостей»* не нов — он использовался еще в PicoContainer. Внедрение зависимостей — это паттерн проектирования, обеспечивающий слабую связанность между компонентами Spring, то есть между различными взаимодействующими POJO. Применение внедрения зависимостей к вашим сложным задачам позволит упростить код для написания, понимания и тестирования.

В вашем приложении множество объектов совместно обеспечивают некую требуемую вам функциональность. Подобная совместная работа объектов известна как внедрение зависимостей. Такое внедрение между работающими компонентами помогает при модульном тестировании каждого компонента приложения без сильной связанности.

В реальном приложении конечный пользователь хочет видеть результат. Для получения результата несколько объектов приложения работают совместно и иногда связаны между собой. Поэтому при написании классов в сложных приложениях подумайте о повторном использовании этих классов и сделайте их максимально независимыми. Это наилучшая практика в программировании, позволяющая модульно тестировать такие классы по отдельности.

Как внедрение зависимостей работает и как упрощает разработку и тестирование

Рассмотрим реализацию паттерна внедрения зависимостей в вашем приложении. Он способствует понятности, слабой связанности и тестируемости всего приложения. Предположим, у нас имеется простое приложение (но посложнее, чем пример *Hello World*, который вы писали на первом курсе). Все классы работают совместно для ожидаемого от них решения некой бизнес-задачи. Это значит, что у каждого класса в приложении есть своя доля ответственности за задачу, которую он делит с сотрудничающими объектами (зависимостями). Посмотрите на рис. 1.1. Такая зависимость между объектами может усложнять приложение и создавать сильную связанность.

Типичное приложение состоит из нескольких частей, работающих совместно для выполнения пользовательской задачи. Например, рассмотрим класс `TransferService`, показанный ниже.

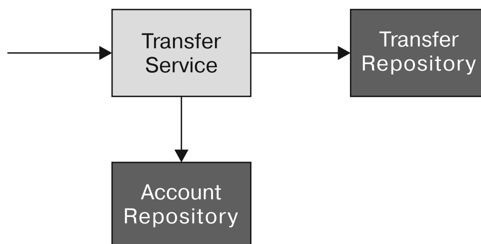

Рис. 1.1. Компонент TransferService традиционно зависит от двух других компонентов: TransferRepository и AccountRepository

Класс `TransferService` использует прямое инстанцирование.

```
package com.packt.chapter1.bankapp.transfer;
public class TransferService {
  private AccountRepository accountRepository;
  public TransferService () {
    this.accountRepository = new AccountRepository();
  }
  public void transferMoney(Account a, Account b) {
    accountRepository.transfer(a, b);
  }
}
```

Объекту `TransferService` нужен объект `AccountRepository`, чтобы перевести деньги со счета `a` на счет `b`. Поэтому он создает экземпляр `AccountRepository` и использует его. Но прямое инстанцирование усиливает связанность и приводит к тому, что создающий объекты код разбросан по всему приложению. При этом становится труднее поддерживать код и писать модульные тесты для `TransferService`, так как в этом случае при необходимости протестировать метод `transferMoney()` класса `TransferService`, используя для модульного тестирования `assert`, метод `transfer()` класса `AccountRepository` вряд ли вызовется тестом. Но разработчик не знает о зависимости `AccountRepository` от класса `TransferService`; по крайней мере он не может протестировать метод `transferMoney()` класса `TransferService` с помощью модульного тестирования.

В корпоративных приложениях связанность очень опасна и приводит к ситуации, когда невозможно развивать приложение в будущем, поскольку любые дальнейшие изменения приводят к появлению большого количества ошибок, а при их исправлении возникают новые ошибки. Сильно связанные компоненты — одна из причин серьезных проблем в подобных приложениях. Сильно связанный код, когда в нем нет необходимости, делает приложение неподдерживаемым, и с течением времени код перестанет использоваться повторно, поскольку не будет понятен другим разработчикам. Однако порой некая связанность необходима в корпоративном приложении, так как полностью не связанные компоненты в реальных задачах применяться не могут. Каждый компонент в приложении несет некую ответственность за свою роль и бизнес-требования, вплоть до того, что все компоненты должны знать об ответственности других приложений. То есть связанность иногда необходима, но мы должны быть очень осторожны при управлении ею, когда она нужна.

Использование вспомогательного паттерна «Фабрика» для зависимых компонентов

Попробуем другой способ обращения с зависимыми объектами, использующий паттерн «Фабрика». Он основан на паттерне «Фабрика» «банды четырех», создающем с помощью фабричного метода экземпляры класса. Таким образом, этот метод централизует применение оператора `new`. Он создает экземпляры класса, основываясь на информации, предоставленной клиентским кодом. Данный паттерн широко используется в стратегии внедрения зависимостей.

Класс `TransferService` использует вспомогательный паттерн «Фабрика»:

```
package com.packt.chapter1.bankapp.transfer;
public class TransferService {
  private AccountRepository accountRepository;
  public TransferService() {
    this.accountRepository =
      AccountRepositoryFactory.getInstance("jdbc");
  }
  public void transferMoney(Account a, Account b) {
    accountRepository.transfer(a, b);
  }
}
```

В данном коде мы используем паттерн «Фабрика» для создания объекта `AccountRepository`. При создании программного обеспечения одной из лучших практик при проектировании и разработке является *program-to-interface (P2I)*. Согласно этой практике конкретные классы должны реализовывать интерфейс, применяемый в клиентском коде для вызывающей функции, а не конкретный класс. Используя P2I, можно улучшать имеющийся код. Мы можем легко заменить его другой реализацией интерфейса, не затрагивая клиентский код. Таким образом, program-to-interface обеспечивает слабую связанность. Иначе говоря, отсутствует зависимость от конкретной реализации, приводящая к зависимости на низком уровне. Рассмотрим следующий код. В нем `AccountRepository` — интерфейс, а не класс:

```
public interface AccountRepository{
  void transfer();
  // другие методы
}
```

Мы можем реализовать его в соответствии с нашими требованиями, и он зависит от инфраструктуры клиента. Предположим, что `AccountREpository` понадобился нам в фазе разработки с использованием JDBC API. Мы можем предоставить конкретную реализацию `JdbcAccountRepository` интерфейса `AccountRepository`, как показано ниже:

```
public class JdbcAccountRepository implements AccountRepository{
  // ...реализация методов, определенных в AccountRepository
  // ...реализация остальных методов
}
```

В данном паттерне объекты создаются фабричными классами, чтобы код было легче поддерживать. Это предотвращает разбросанный среди других бизнес-компонентов код, создающий объекты. Вспомогательный класс фабрики также позволяет сделать создание объектов конфигурируемым. Эта техника решает проблему сильной связанности, но мы по-прежнему добавляем фабричные классы к бизнес-компоненту для получения сотрудничающих компонентов. Так что рассмотрим в следующем подразделе паттерн внедрения зависимостей и то, как он решает указанную проблему.

Использование паттерна внедрения зависимостей для зависимых компонентов

Согласно паттерну внедрения зависимостей зависимости присваиваются объектам при их создании фабрикой или третьей стороной. Эта фабрика координирует объекты в системе так, что от зависимого объекта не ожидается создание своих зависимостей. Таким образом, нам надо сосредоточиться на определении зависимостей вместо разрешения зависимостей сотрудничающих объектов в корпоративном приложении. Посмотрим на рис. 1.2. Обратите внимание, что зависимости внедряются в объекты, которым они необходимы.

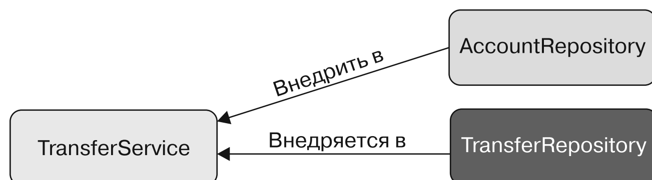

Рис. 1.2. Внедрение зависимостей между различными сотрудничающими компонентами приложения

Для иллюстрации этого утверждения рассмотрим класс `TransferService` — у него есть зависимости от `AccountRepository` и `TransferRepository`. Здесь `TransferService` способен переводить деньги при использовании любой реализации интерфейса `TransferRepository`, то есть мы можем применить `JdbcTransferRepository` или `JpaTransferRepository` в зависимости от того, что получаем на входе из окружающей среды.

Класс `TransferServiceImpl` достаточно гибок, чтобы принимать любой данный ему `TransferRepository`:

```
package com.packt.chapter1.bankapp;
public class TransferServiceImpl implements TransferService {
  private TransferRepository transferRepository;
  private AccountRepository accountRepository;
  public TransferServiceImpl(TransferRepository transferRepository,
   AccountRepository accountRepository) {
     this.transferRepository =
      transferRepository;//TransferRepository is injected
     this.accountRepository = accountRepository;
     // AccountRepository внедрен
   }
   public void transferMoney(Long a, Long b, Amount amount) {
     Account accountA = accountRepository.findByAccountId(a);
     Account accountB = accountRepository.findByAccountId(b);
     transferRepository.transfer(accountA, accountB, amount);
   }
}
```