

# Spring

## В ДЕЙСТВИИ

ТРЕТЬЕ ИЗДАНИЕ

Крейг Уоллс



ДМК  
ИЗДАТЕЛЬСТВО



MANNING

**УДК 004Spring**  
**ББК 32.973-018.2**  
**У62**

Уоллс К.

У62 Spring в действии. – М.: ДМК Пресс, 2013. – 752 с.: ил.

ISBN 978-5-94074-568-6

Фреймворк Spring Framework – необходимый инструмент для разработчиков приложений на Java.

В книге описана последняя версия Spring 3, который несет в себе новые мощные особенности, такие как язык выражений SpEL, новые аннотации для работы с контейнером IoC и поддержка архитектуры REST. Автор, Крейг Уоллс, обладает особым талантом придумывать весьма интересные примеры, сосредоточенные на особенностях и приемах использования Spring, которые действительно будут полезны читателям.

В русскоязычном переводе добавлены главы из 2-го американского издания, которые автор не включил в 3-е издание «Spring in Action».

Издание предназначено как для начинающих пользователей фреймворка, так и для опытных пользователей Spring, желающих задействовать новые возможности версии 3.0.

**УДК 004Spring**  
**ББК 32.973-018.2**

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-9351-8235-1 (анг.)

Copyright © 2011 by Manning  
Publications Co.

ISBN 978-5-94074-568-6 (рус.)

© Оформление, перевод  
ДМК Пресс, 2013



# Содержание

<b>Предисловие к русскому изданию .....</b>	<b>16</b>
<b>Предисловие .....</b>	<b>18</b>
<b>Благодарности .....</b>	<b>20</b>
<b>Об этой книге .....</b>	<b>22</b>
<b>Об иллюстрации на обложке .....</b>	<b>28</b>
 <b>Часть I. Ядро Spring .....</b>	 <b>29</b>
 <b>Глава 1. Введение в Spring .....</b>	 <b>30</b>
1.1. Упрощение разработки на языке Java .....	32
1.1.1. Свобода использования POJO .....	33
1.1.2. Внедрение зависимостей .....	35
1.1.3. Применение аспектно-ориентированного программирования .....	40
1.1.4. Устранение шаблонного кода с помощью шаблонов ....	47
1.2. Контейнер компонентов .....	50
1.2.1. Работа с контекстом приложения .....	51
1.2.2. Жизненный цикл компонента .....	53
1.3. Обзор возможностей Spring .....	54
1.3.1. Модули Spring .....	55
1.3.2. Дополнительные возможности Spring .....	59
1.4. Что нового в Spring .....	64
1.4.1. Что нового в Spring 2.5? .....	64
1.4.2. Что нового в Spring 3.0? .....	65
1.4.3. Что нового в экосистеме Spring? .....	66
1.5. В заключение .....	67
 <b>Глава 2. Связывание компонентов .....</b>	 <b>69</b>
2.1. Объявление компонентов .....	70

2.1.1. Подготовка конфигурации Spring .....	71
2.1.2. Объявление простого компонента .....	73
2.1.3. Внедрение через конструкторы .....	74
2.1.4. Область действия компонента .....	80
2.1.5. Инициализация и уничтожение компонентов .....	82
2.2. Внедрение в свойства компонентов .....	84
2.2.1. Внедрение простых значений .....	86
2.2.2. Внедрение ссылок на другие компоненты .....	87
2.2.3. Связывание свойств с помощью пространства имен p .....	91
2.2.4. Внедрение коллекций .....	92
2.2.5. Внедрение пустого значения .....	99
2.3. Внедрение выражений .....	99
2.3.1. Основы языка выражений SpEL .....	100
2.3.2. Выполнение операций со значениями SpEL .....	105
2.3.3. Обработка коллекций на языке SpEL .....	111
2.4. В заключение .....	116

### **Глава 3. Дополнительные способы связывания компонентов .....**

3.1. Объявление родителей и потомков компонентов .....	118
3.1.1. Абстрактные компоненты .....	119
3.1.2. Общие абстрактные свойства .....	122
3.2. Внедрение методов .....	124
3.2.1. Основы замещения методов .....	125
3.2.2. Использование внедрения методов чтения .....	130
3.3. Внедрение не-Spring компонентов .....	132
3.4. Пользовательские редакторы свойств .....	135
3.5. Специальные компоненты Spring .....	140
3.5.1. Компоненты постобработки .....	140
3.5.2. Постобработка контейнера .....	144
3.5.3. Внешние файлы с настройками свойств .....	145
3.5.4. Извлечение текстовых сообщений .....	147
3.5.5. Уменьшение связности с использованием событий ...	150
3.5.6. Создание «осведомленных» компонентов .....	153
3.6. Компоненты, управляемые сценариями .....	156
3.6.1. Добавляем лайм в кокос .....	158

3.6.2. Компонент, управляемый сценарием.....	159
3.6.3. Внедрение в свойства компонентов, управляемых сценариями .....	162
3.6.4. Обновление компонентов, управляемых сценариями .....	164
3.6.5. Создание компонентов, управляемых сценариями, непосредственно в конфигурации.....	165
3.7. В заключение.....	166

## **Глава 4. Сокращение размера**

<b>XML-конфигурации Spring .....</b>	<b>168</b>
4.1. Автоматическое связывание свойств компонентов .....	169
4.1.1. Четыре типа автоматического связывания .....	169
4.1.2. Смешивание автоматического и явного связывания .....	175
4.2. Связывание посредством аннотаций .....	177
4.2.1. Использование аннотации @Autowired .....	178
4.2.2. Автоматическое связывание с применением стандартной аннотации @Inject.....	183
4.2.3. Использование выражений в аннотациях внедрения зависимостей.....	186
4.3. Автоматическое определение компонентов.....	188
4.3.1. Аннотирование компонентов для автоматического определения .....	189
4.3.2. Включение фильтров в элемент component-scans ...	190
4.4. Конфигурирование Spring в программном коде на Java....	192
4.4.1. Подготовка к конфигурированию на языке Java .....	192
4.4.2. Определение конфигурационных классов .....	193
4.4.3. Объявление простого компонента .....	194
4.4.4. Внедрение зависимостей в конфигурации на языке Java .....	195
4.5. В заключение.....	196

## **Глава 5. Аспектно-ориентированный Spring .....**

5.1. Знакомство с AOP .....	200
5.1.1. Определение терминологии AOP .....	201
5.1.2. Поддержка AOP в Spring .....	204

5.2. Выбор точек сопряжения в описаниях срезов .....	207
5.2.1. Определение срезов множества точек сопряжения .....	209
5.2.2. Использование указателя bean() .....	210
5.3. Объявление аспектов в XML .....	211
5.3.1. Объявление советов, выполняемых до или после ....	213
5.3.2. Объявление советов, выполняемых до, и после ....	216
5.3.3. Передача параметров советам .....	218
5.3.4. Внедрение новых возможностей с помощью аспектов .....	221
5.4. Аннотирование аспектов .....	224
5.4.1. Создание советов, выполняемых и до, и после .....	227
5.4.2. Передача аргументов аннотированным советам ....	228
5.4.3. Внедрение с помощью аннотаций .....	230
5.5. Внедрение аспектов AspectJ .....	231
5.6. В заключение .....	235

## **Часть II. Основы приложений Spring .....**

**237**

### **Глава 6. Работа с базами данных .....**

**238**

6.1. Философия доступа к данным в Spring .....	239
6.1.1. Знакомство с иерархией исключений доступа к данным в Spring .....	241
6.1.2. Шаблоны доступа к данным .....	244
6.1.3. Использование классов поддержки DAO .....	247
6.2. Настройка источника данных .....	249
6.2.1. Использование источников данных из JNDI .....	249
6.2.2. Использование пулов соединений .....	250
6.2.3. Источник данных JDBC .....	252
6.3. Использование JDBC совместно со Spring .....	253
6.3.1. Борьба с разбуханием JDBC-кода .....	254
6.3.2. Работа с шаблонами JDBC .....	258
6.4. Интеграция Hibernate и Spring .....	265
6.4.1. Обзор Hibernate .....	267
6.4.2. Объявление фабрики сеансов Hibernate .....	268
6.4.3. Создание классов для работы с Hibernate, независимых от Spring .....	271

6.5. Spring и Java Persistence API .....	273
6.5.1. Настройка фабрики диспетчера сущностей.....	274
6.5.2. Объект DAO на основе JPA.....	280
6.6. Кеширование.....	282
6.6.1. Настройка кеширования.....	285
6.6.2. Настройка компонентов для кеширования.....	289
6.6.3. Декларативное кеширование с помощью аннотаций ....	292
6.7. В заключение.....	294

## **Глава 7. Управление транзакциями..... 296**

7.1. Знакомство с транзакциями .....	297
7.1.1. Описание транзакций в четырех словах.....	299
7.1.2. Знакомство с поддержкой транзакций в Spring .....	300
7.2. Выбор диспетчера транзакций.....	301
7.2.1. Транзакции JDBC .....	303
7.2.2. Транзакции Hibernate .....	303
7.2.3. Транзакции Java Persistence API .....	304
7.2.4. Транзакции Java Transaction API .....	305
7.3. Программное управление транзакциями в Spring .....	306
7.4. Декларативное управление транзакциями.....	309
7.4.1. Определение атрибутов транзакций.....	309
7.4.2. Объявление транзакций в XML.....	315
7.4.3. Определение транзакций с помощью аннотаций.....	318
7.5. В заключение.....	320

## **Глава 8. Создание веб-приложений с помощью Spring MVC..... 321**

8.1. Обзор Spring MVC .....	322
8.1.1. Путь одного запроса через Spring MVC.....	322
8.1.2. Настройка Spring MVC .....	324
8.2. Создание простого контроллера.....	327
8.2.1. Настройка поддержки аннотаций в Spring MVC.....	328
8.2.2. Контроллер главной страницы.....	329
8.2.3. Поиск представлений .....	334
8.2.4. Объявление представления главной страницы .....	340
8.2.5. Завершение определения контекста приложения	
Spring .....	342

8.3. Контроллер обработки входных данных .....	344
8.3.1. Создание контроллера, обрабатывающего входные данные.....	345
8.3.2. Представление, отображающее список сообщений .....	348
8.4. Обработка форм .....	349
8.4.1. Отображение формы регистрации .....	350
8.4.2. Обработка данных формы .....	353
8.4.3. Проверка входных данных .....	355
8.5. Выгрузка файлов .....	361
8.5.1. Добавление в форму поля выгрузки файла .....	361
8.5.2. Прием выгружаемых файлов .....	363
8.5.3. Настройка Spring для выгрузки файлов .....	367
8.6. Альтернативы JSP1 .....	368
8.6.1. Использование шаблонов Velocity .....	368
8.6.2. Использование шаблонов FreeMarker.....	377
8.7. Генерирование вывода, отличного от HTML.....	383
8.7.1. Создание электронных таблиц Excel .....	384
8.7.2. Создание документов PDF .....	388
8.8. В заключение.....	391

## **Глава 9. Использование Spring Web Flow .....**

9.1. Установка Spring Web Flow .....	393
9.1.1. Настройка расширения Web Flow в Spring.....	394
9.2. Элементы последовательности операций.....	397
9.2.1. Состояния .....	398
9.2.2. Переходы .....	402
9.2.3. Данные в последовательностях.....	404
9.3. Соединяем все вместе: последовательность pizza.....	407
9.3.1. Определение основной последовательности .....	407
9.3.2. Сбор информации о клиенте .....	412
9.3.3. Оформление заказа .....	420
9.3.4. Прием оплаты .....	424
9.4. Безопасность веб-последовательностей .....	427
9.5. Интеграция Spring Web Flow с другими фреймворками.....	427
9.5.1. JavaServer Faces.....	428
9.6. В заключение.....	430



<b>Глава 10. Безопасность в Spring</b> .....	432
10.1. Введение в Spring Security .....	433
10.1.1. Обзор Spring Security .....	434
10.1.2. Использование конфигурационного пространства имен Spring Security .....	435
10.2. Безопасность веб-запросов .....	436
10.2.1. Сервлет-фильтры .....	437
10.2.2. Минимальная настройка безопасности .....	438
10.2.3. Перехват запросов .....	443
10.3. Безопасность на уровне представлений .....	447
10.3.1. Доступ к информации об аутентификации .....	448
10.3.2. Отображение с учетом привилегий .....	449
10.4. Аутентификация пользователей .....	452
10.4.1. Настройка репозитория в памяти .....	453
10.4.2. Аутентификация с использованием базы данных .....	455
10.4.3. Аутентификация с использованием LDAP .....	457
10.4.4. Включение функции «запомнить меня» .....	462
10.5. Защита методов .....	463
10.5.1. Защита методов с помощью аннотации @Secured .....	464
10.5.2. Использование аннотации JSR-250 @RolesAllowed .....	465
10.5.3. Защита с помощью аннотаций, выполняемых до и после вызова .....	466
10.5.4. Объявление точек внедрения для защиты методов .....	472
10.6. В заключение .....	472
<b>Часть III. Интеграция Spring</b> .....	474
<b>Глава 11. Взаимодействие с удаленными службами</b> .....	476
11.1. Обзор механизмов удаленных взаимодействий в Spring .....	477
11.2. Использование RMI .....	481
11.2.1. Экспортирование службы RMI .....	481

11.2.2. Внедрение службы RMI .....	484
11.3. Экспортирование удаленных служб с помощью Hessian и Burlap .....	488
11.3.1. Экспортирование службы с помощью Hessian/Burlap .....	489
11.3.2. Доступ к службам Hessian/Burlap .....	492
11.4. Использование Spring Http Invoker .....	494
11.4.1. Экспортирование компонентов в виде служб HTTP Invoker .....	495
11.4.2. Доступ к службам HTTP Invoker .....	497
11.5. Экспортирование и использование веб-служб .....	498
11.5.1. Создание конечных точек JAX-WS с поддержкой Spring .....	500
11.5.2. Проксирование служб JAX-WS на стороне клиента .....	505
11.6. В заключение .....	507

## **Глава 12. Поддержка архитектуры REST в Spring .... 509**

12.1. Обзор архитектуры REST .....	510
12.1.1. Основы REST .....	510
12.1.2. Поддержка REST в Spring .....	511
12.2. Создание контроллеров, ориентированных на ресурсы .....	512
12.2.1. Устройство контроллера, противоречащего архитектуре REST .....	512
12.2.2. Обработка адресов URL в архитектуре RESTful .....	514
12.2.3. Выполнение операций в стиле REST .....	519
12.3. Представление ресурсов .....	523
12.3.1. Договоренность о представлении ресурса .....	524
12.3.2. Преобразование HTTP-сообщений .....	528
12.4. Клиенты REST .....	532
12.4.1. Операции класса RestTemplate .....	534
12.4.2. Чтение ресурсов .....	536
12.4.3. Изменение ресурсов .....	540
12.4.4. Удаление ресурсов .....	542
12.4.5. Создание новых ресурсов .....	543
12.4.6. Обмен ресурсами .....	546

12.5. Отправка форм в стиле RESTful .....	549
12.5.1. Отображение скрытого поля с именем метода .....	550
12.5.2. Преобразование типа запроса .....	552
12.6. В заключение .....	553

## **Глава 13. Обмен сообщениями в Spring** .....

13.1. Краткое введение в JMS .....	556
13.1.1. Архитектура JMS .....	557
13.1.2. Преимущества JMS .....	561
13.2. Настройка брокера сообщений в Spring .....	563
13.2.1. Создание фабрики соединений .....	563
13.2.2. Объявление приемников ActiveMQ .....	565
13.3. Работа с шаблонами JMS в Spring .....	566
13.3.1. Борьба с разбуханием JMS-кода .....	566
13.3.2. Работа с шаблонами JMS .....	568
13.4. Создание POJO, управляемых сообщениями .....	575
13.4.1. Создание объекта для чтения сообщений .....	576
13.4.2. Настройка обработчиков сообщений .....	578
13.5. Механизмы RPC, основанные на сообщениях .....	579
13.5.1. Механизм RPC, основанный на сообщениях, в фреймворке Spring .....	580
13.5.2. Механизм RPC, основанный на сообщениях, в Lingo .....	583
13.6. В заключение .....	586

## **Глава 14. Управление компонентами Spring с помощью JMX** .....

14.1. Экспортирование компонентов Spring как управляемых компонентов .....	589
14.1.1. Экспортирование методов по их именам .....	593
14.1.2. Определение экспортируемых операций и атрибутов с помощью интерфейсов .....	596
14.1.3. Объявление управляемых компонентов с помощью аннотаций .....	597
14.1.4. Разрешение конфликтов между управляемыми компонентами .....	599
14.2. Удаленные компоненты MBean .....	601

14.2.1. Экспортирование удаленного компонента	
MBean .....	602
14.2.2. Доступ к удаленным компонентам MBean.....	603
14.2.3. Проксирование управляемых компонентов .....	605
14.3. Обработка извещений .....	606
14.3.1. Прием извещений .....	609
14.4. В заключение .....	610

## **Глава 15. Создание веб-служб на основе модели contract-first .....**

<b>contract-first .....</b>	<b>611</b>
15.1. Введение в Spring-WS .....	613
15.2. Определение API службы (в первую очередь!).....	616
15.2.1. Создание примеров XML-сообщений .....	616
15.3. Обработка сообщений в веб-службе .....	623
15.3.1. Создание конечной точки на основе модели JDOM .....	625
15.3.2. Маршалинг содержимого сообщений.....	628
15.4. Связываем все вместе .....	632
15.4.1. Spring-WS: общая картина .....	633
15.4.2. Отображение сообщений в конечные точки .....	634
15.4.3. Настройка конечной точки службы.....	635
15.4.4. Настройка маршала сообщения .....	636
15.4.5. Обработка исключений в конечной точке .....	639
15.4.6. Создание WSDL-файлов .....	641
15.4.7. Развертывание службы.....	646
15.5. Использование веб-служб Spring-WS .....	647
15.5.1. Работа с шаблонами веб-служб.....	648
15.5.2. Использование поддержки шлюза веб-служб .....	656
15.6. В заключение .....	658

## **Глава 16. Spring и Enterprise JavaBeans .....**

16.1. Внедрение компонентов EJB в Spring.....	661
16.1.1. Проксирование сеансовых компонентов (EJB 2.x).....	663
16.1.2. Внедрение компонентов EJB в компоненты Spring .....	668
16.2. Разработка компонентов с поддержкой Spring (EJB 2.x) .....	670

16.3. Spring и EJB3 .....	673
16.3.1. Pitchfork.....	674
16.3.2. Введение в Pitchfork .....	676
16.3.3. Внедрение ресурсов с помощью аннотации .....	676
16.3.4. Объявление перехватчиков с помощью аннотаций.....	678
16.4. В заключение .....	680
<b>Глава 17. Прочее .....</b>	<b>682</b>
17.1. Импортирование внешних настроек.....	683
17.1.1. Подстановка переменных-заполнителей .....	684
17.1.2. Переопределение свойств .....	687
17.1.3. Шифрование внешних определений свойств.....	689
17.2. Внедрение объектов из JNDI .....	692
17.2.1. Работа с обычным JNDI API .....	692
17.2.2. Внедрение объектов из JNDI .....	695
17.2.3. Внедрение компонентов EJB в Spring .....	699
17.3. Отправка электронной почты .....	701
17.3.1. Настройка отправки электронной почты .....	701
17.3.2. Создание электронных писем.....	704
17.4. Выполнение заданий по расписанию и в фоновом режиме .....	711
17.4.1. Объявление методов, вызываемых по расписанию .....	712
17.4.2. Объявление асинхронных методов .....	714
17.5. В заключение .....	716
17.6. Конец? .....	717
<b>Предметный указатель.....</b>	<b>719</b>



# Глава 1. Введение в Spring

В этой главе рассматриваются следующие темы:

- ❑ обзор основных модулей Spring;
- ❑ разделение прикладных объектов;
- ❑ управление сквозными задачами с помощью AOP;
- ❑ контейнер компонентов в Spring.

Все началось с компонента.

В 1996 году язык программирования Java был еще новой, перспективной платформой, вызывающей большой интерес. Многие разработчики пришли в этот язык после того, как они увидели, как с помощью апплетов можно создавать полнофункциональные и динамические веб-приложения. Однако вскоре они увидели, что этот новый и странный язык способен на большее, нежели простое управление мультяшными героями. В отличие от других существующих языков, Java позволил писать сложные приложения, состоящие из отдельных частей. Они пришли ради апплетов, а остались ради компонентов.

В декабре 1996 года компания Sun Microsystems опубликовала спецификацию JavaBeans 1.00-A, определившую модель программных компонентов Java – набор приемов программирования, позволяющих повторно использовать простые Java-объекты и легко конструировать из них более сложные приложения. Несмотря на то что компоненты JavaBeans были задуманы как универсальный механизм определения повторно используемых прикладных компонентов, они использовались преимущественно в качестве шаблона для создания элементов пользовательского интерфейса. Они казались слишком простыми для «настоящей» работы. Промышленным программистам требовалось нечто большее.

Сложные приложения часто требуют таких услуг, как поддержка транзакций, безопасность и распределенные вычисления, которые не были предусмотрены спецификацией JavaBeans. Поэтому в марте 1998 года компания Sun Microsystems опубликовала новую спецификацию – Enterprise JavaBeans (EJB) 1.0. Эта спецификация

расширила понятие серверных Java-компонентов, обеспечив столь необходимые службы, однако она не смогла поддержать той простоты, что была заложена в оригинальную спецификацию JavaBeans. Кроме похожего названия, спецификация EJB имеет мало общего со спецификацией JavaBeans.

Несмотря на появление множества успешных приложений, созданных на основе спецификации EJB, она никогда не применялась по прямому назначению: для упрощения разработки корпоративных приложений. Это правда, что декларативная модель программирования EJB упрощает многие инфраструктурные аспекты разработки, такие как транзакции и безопасность. Но она привнесла другие сложности, требуя создания дескрипторов развертывания и использования шаблонного кода (домашние и удаленные/локальные интерфейсы). С течением времени многие разработчики разочаровались в EJB, что привело к спаду популярности данной технологии и поиску более простых путей.

Сегодня разработка Java-компонентов вернулась к своим истокам. Новые технологии программирования, включая аспектно-ориентированное программирование (AOP) и внедрение зависимостей (DI), дают JavaBeans дополнительные возможности, ранее заложенные в EJB. Эти технологии оснащают обычные Java-объекты (Plain-Old Java Objects, POJO) моделью декларативного программирования, напоминающей EJB, но без всей сложности спецификации EJB. Больше не надо создавать громоздкий компонент EJB, когда достаточно простого компонента JavaBean.

Следует отметить, что даже EJB эволюционировала к модели программирования на основе POJO. Заимствуя идеи AOP и DI, последняя спецификация EJB стала существенно проще, чем предшествующие. Однако многие разработчики расценили этот шаг как слишком маленький и сделанный достаточно поздно. К моменту выхода спецификации EJB 3 другие фреймворки на основе модели POJO уже зарекомендовали себя в сообществе пользователей Java как фактические стандарты.

Фреймворк Spring Framework, который будет изучаться на протяжении всей этой книги, является передовым средством разработки приложений на основе POJO. В этой главе Spring Framework будет рассматриваться с общей точки зрения, чтобы дать представление о том, что такое Spring. Данная глава позволит получить хорошее понимание круга задач, решаемых фреймворком Spring, и подготовит почву для остальной части книги. Для начала выясним, что такое Spring.

## 1.1. Упрощение разработки на языке Java

Spring – это свободно распространяемый фреймворк, созданный Родом Джонсоном (Rod Johnson) и описанный в его книге «Expert One-on-One: J2EE Design and Development». Он был создан с целью устранить сложности разработки корпоративных приложений и сделать возможным использование простых компонентов JavaBean для достижения всего того, что ранее было возможным только с использованием EJB. Однако область применения Spring не ограничивается разработкой программных компонентов, выполняющихся на стороне сервера. Любое Java-приложение может использовать преимущества фреймворка в плане простоты, тестируемости и слабой связанности.

---

**Термины, обозначающие компоненты.** Несмотря на то, что при обсуждении фреймворка Spring для обозначения программных компонентов слова «bean» и «JavaBean» используются как синонимы, это не означает, что компоненты в Spring должны точно соответствовать спецификации JavaBeans. Компоненты в Spring могут быть любыми простыми объектами модели POJO. В этой книге термин «JavaBeans» будет использоваться как синоним термина «POJO» (обычный Java-объект).

---

Как будет не раз показано на протяжении этой книги, фреймворк Spring обладает весьма широкими возможностями. Но в основе практически всех его особенностей лежат несколько фундаментальных идей, направленных на достижение главной цели – *упрощение разработки приложений на языке Java*.

Весьма смелое заявление! Разработчики многих фреймворков утверждают, что их продукты упрощают те или иные аспекты разработки. Но целью фреймворка Spring является упрощение разработки приложений Java вообще. Это требует некоторых пояснений. Так как же фреймворк Spring упрощает разработку на языке Java?

В своем устремлении на сложности, связанные с разработкой на языке Java, фреймворк Spring использует четыре ключевые стратегии:

- ❑ легковесность и ненасильственность благодаря применению простых Java-объектов (POJO);
- ❑ слабое связывание посредством внедрения зависимостей и ориентированности на интерфейсы;
- ❑ декларативное программирование через аспекты и общепринятые соглашения;
- ❑ уменьшение объема типового кода через аспекты и шаблоны.



Практически все возможности фреймворка Spring уходят корнями в эти стратегии. В остальной части главы каждая из этих идей будет рассматриваться более подробно на конкретных примерах, позволяющих увидеть, как фреймворк Spring действительно упрощает разработку приложений на языке Java. Для начала посмотрим, как он обеспечивает ненасильственность, поощряя использование простых объектов.

### 1.1.1. Свобода использования POJO

Те, кто имеет опыт достаточно продолжительной разработки на языке Java, вероятно, видели (и даже могли использовать) фреймворки, вынуждающие расширять свои классы или предусматривать реализацию своих интерфейсов. Классическим примером являются сеансовые компоненты эры EJB 2. Как показано в простейшем примере HelloWorldBean, спецификация EJB 2 предъявляет достаточно сложные требования:

#### Листинг 1.1. Спецификация EJB 2.1 вынуждает реализовывать ненужные методы

---

```
package com.habuma.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean { // Зачем все эти методы
    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }

    public void setSessionContext(SessionContext ctx) {
    }

    public String sayHello() { // Основная логика компонента EJB
        return "Hello World";
    }

    public void ejbCreate() {
    }
}
```

---

Интерфейс `SessionBean` обеспечивает возможность управления жизненным циклом компонента EJB за счет реализации различных методов обратного вызова (имена этих методов начинаются с последовательности символов `ejb`). Или, говоря другими словами, интерфейс `SessionBean` вынуждает вторгаться в жизненный цикл компонента EJB, даже если в этом нет необходимости. Масса программного кода в примере `HelloWorldBean` нужна только ради удовлетворения нужд фреймворка. В результате возникает вопрос: кто для кого работает?

Однако спецификация EJB 2 не единственная в своей насильственности. Другие популярные фреймворки, такие как ранние версии `Struts`, `WebWork` и `Tapestry`, накладывали свой отпечаток на иначе простые Java-классы. Эти тяжеловесные фреймворки вынуждали разработчиков создавать классы, захламленные ненужным программным кодом, часто сложные в тестировании.

Фреймворк `Spring` не вынуждает (насколько это возможно) захламлять приложения программным кодом для поддержки своего API. Он практически никогда не заставляет обеспечивать реализацию своих интерфейсов или наследовать свои классы. Напротив, в приложениях, основанных на фреймворке `Spring`, классы часто вообще не имеют никаких признаков, по которым можно было бы судить, что они используются фреймворком. В худшем случае класс может быть аннотирован одной из аннотаций `Spring`, но во всех остальных отношениях он будет обычным объектом `POJO`.

Для иллюстрации класс `HelloWorldBean`, представленный в листинге 1.1, можно преобразовать в компонент, управляемый фреймворком `Spring`, как показано в листинге 1.2.

---

**Листинг 1.2. Фреймворк `Spring` не выдвигает необоснованных требований к классу `HelloWorldBean`**

---

```
package com.habuma.spring;

public class HelloWorldBean {
    public String sayHello() {                // Это все, что необходимо
        return "Hello World";
    }
}
```

---

Так лучше? Исчезли все ненужные методы управления жизненным циклом. Класс `HelloWorldBean` не реализует, не наследует и не

импортирует ничего из Spring API. Класс `HelloWorldBean` мал, краток и во всех смыслах является простым Java-объектом.

Несмотря на свою простоту, простые Java-объекты могут обладать большими возможностями. Одним из механизмов Spring, увеличивающих мощь простых объектов, является их объединение с помощью внедрения зависимостей. Рассмотрим, как внедрение зависимостей помогает сохранить прикладные объекты независимыми друг от друга.

### 1.1.2. Внедрение зависимостей

Для кого-то фраза «внедрение зависимостей» может звучать устрашающе, вызывая в воображении сложные приемы программирования или шаблоны проектирования. Однако на самом деле DI не настолько сложно, как кажется. На самом деле применение DI в проектах позволяет существенно упростить программный код, облегчит его понимание и тестирование.

Любое нетривиальное приложение (более сложное, чем простое приложение `Hello World`) состоит из двух или более классов, которые взаимодействуют друг с другом, реализуя некоторую логику. Обычно каждый объект ответствен за получение собственных ссылок на объекты, с которыми он взаимодействует (его зависимости). Это может привести к сильной связности и сложностям при тестировании.

Например, взгляните на класс **рыцаря** в листинге 1.3 ниже.

#### Листинг 1.3. Класс `DamselRescuingKnight` может принимать только экземпляр класса `RescueDamselQuest`

---

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
    private RescueDamselQuest quest;

    public DamselRescuingKnight() {
        quest = new RescueDamselQuest(); // Тесная связь с классом
    }                                     // RescueDamselQuest

    public void embarkOnQuest() throws QuestException {
        quest.embark();
    }
}
```

---

Как показано в листинге 1.3, экземпляр `DamselRescuingKnight` создает собственный экземпляр `RescueDamselQuest` в конструкторе. Это обуславливает тесную связь между `DamselRescuingKnight` и `RescueDamselQuest` и существенно ограничивает возможности рыцаря. Если потребуется спасти даму, он готов будет совершить этот подвиг. Но если потребуется убить дракона или, например, стать рыцарем Круглого стола, он не сможет сделать этого<sup>1</sup>.

Но, что хуже всего, для класса `DamselRescuingKnight` очень сложно будет написать модульный тест. В процессе тестирования желательно было бы убедиться, что при вызове метода `embarkOnQuest()` класса `DamselRescuingKnight` вызывается метод `embark()` класса `RescueDamselQuest`. Но в данном случае нет очевидного способа реализовать такую проверку. К сожалению, класс `DamselRescuingKnight` останется непротестированным.

Связь классов – это «двуглавый зверь». С одной стороны, сильно связанный код сложен в тестировании, его трудно использовать повторно, он сложен в понимании, и, как правило, такой код оказывается очень хрупким при исправлении ошибок (исправление одной ошибки может привести к нескольким новым). С другой стороны, совершенно не связанный код ничего не делает. Чтобы делать что-то полезное, классы должны знать друг о друге. Связанность необходима, но должна тщательно контролироваться.

С другой стороны, благодаря DI объекты получают свои зависимости во время создания от некоторой третьей стороны, координирующей работу каждого объекта в системе. Объекты не создают и не получают свои зависимости самостоятельно – зависимости внедряются в объекты.

Для иллюстрации рассмотрим класс `BraveKnight`, представленный в листинге 1.4, реализующий рыцаря, который не только храбр, но и способен совершать любые подвиги.

---

**Листинг 1.4. Класс `BraveKnight`, достаточно гибкий, чтобы совершить любой подвиг**

---

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
```

---

<sup>1</sup> Здесь необходимо уточнить, что класс `DamselRescuingKnight` реализует «рыцаря, спасающего даму», а класс `RescueDamselQuest` реализует «сценарий спасения дамы». – *Прим. перев.*

```
private Quest quest;

public BraveKnight(Quest quest) {
    this.quest = quest;           // Внедрение сценария подвига
}

public void embarkOnQuest() throws QuestException {
    quest.embark();
}
}
```

Как видно из листинга, в отличие от класса `DamselRescuingKnight`, класс `BraveKnight` не создает собственного сценария подвига, а получает его извне, в виде аргумента конструктора. Такой способ внедрения зависимостей называется *внедрением через конструктор*.

Более того, сценарий подвига имеет тип интерфейса `Quest`, который реализуют все такие сценарии. Поэтому `BraveKnight` (храбрый рыцарь) сможет совершать любые подвиги, такие как `RescueDamselQuest` (спаси даму), `SlayDragonQuest` (убить дракона), `MakeRoundTableRounderQuest` (стать рыцарем Круглого стола) или любой другой, реализующий интерфейс `Quest`.

Фактически класс `BraveKnight` никак не связан с конкретной реализацией `Quest`. Для него не важно, какой подвиг будет поручен, при условии что он реализует интерфейс `Quest`. В этом состоит основное преимущество DI – слабая связанность. Если объект взаимодействует со своими зависимостями только через их интерфейсы (ничего не зная о конкретных реализациях или особенностях их создания), зависимости можно будет замещать любыми другими реализациями, без необходимости учитывать эти различия в самом объекте.

Прием замены зависимостей очень часто используется при тестировании, когда выполняется подстановка фиктивной реализации. Класс `DamselRescuingKnight` невозможно было протестировать в полной мере из-за тесной связи, но класс `BraveKnight` легко поддается тестированию за счет подстановки фиктивной реализации интерфейса `Quest`, как показано в листинге 1.5.

---

**Листинг 1.5. Протестировать класс `BraveKnight` можно с помощью фиктивной реализации интерфейса `Quest`**

---

```
package com.springinaction.knights;

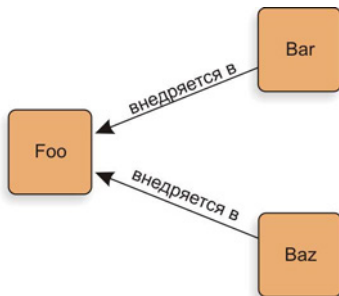
import static org.mockito.Mockito.*;
```

```
import org.junit.Test;

public class BraveKnightTest {
    @Test
    public void knightShouldEmbarkOnQuest() throws QuestException {
        Quest mockQuest = mock(Quest.class); // Создание фиктивного
                                              // объекта Quest
        BraveKnight knight = new BraveKnight(mockQuest); // Внедрение
        knight.embarkOnQuest();

        verify(mockQuest, times(1)).embark();
    }
}
```

В данном примере фиктивная реализация интерфейса `Quest` создана с помощью фреймворка `Mockito`. После получения фиктивного объекта создается новый экземпляр `BraveKnight`, в который через конструктор внедряется фиктивный объект `Quest`. После вызова метода `embarkOnQuest()` выполняется обращение к фреймворку `Mockito` с целью убедиться, что метод `embark()` интерфейса `Quest` фиктивного объекта был вызван только один раз.



**Рис. 1.1.** Механизм внедрения зависимостей основан на предоставлении объекту его зависимостей извне, а не на приобретении этих зависимостей самим объектом

### ***Передача сценария подвига рыцарю***

Теперь, когда класс `BraveKnight` может принимать любые задания, как определить, какой именно объект `Quest` был ему передан? Процесс создания связей между прикладными компонентами называется *связыванием* (wiring). Фреймворк `Spring` поддерживает множест-

во способов связывания компонентов, но наиболее общим из них является способ на основе XML. В листинге 1.6 показано содержимое простого конфигурационного файла Spring, `knights.xml`, который передает объекту `BraveKnight` задание `SlayDragonQuest`.

### Листинг 1.6. Внедрение сценария `SlayDragonQuest` в объект `BraveKnight` средствами Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" />    <!-- Внедрение компонента quest -->
  </bean>

  <!-- Создание SlayDragonQuest -->
  <bean id="quest"
    class="com.springinaction.knights.SlayDragonQuest" />
</beans>
```

Это простой способ связывания компонентов. Пока не стоит слишком беспокоиться о деталях. Подробнее о конфигурировании Spring и о том, что происходит на данном этапе, будет рассказываться в главе 2, где также будут показаны другие способы связывания компонентов в Spring.

Теперь, объявив отношения между `BraveKnight` и `Quest`, необходимо загрузить XML-файл и запустить приложение.

### ***Рассмотрим этот механизм в действии***

В приложении, созданном на основе Spring, *контекст приложения* загружает определения компонентов и связывает их вместе. За создание объектов, составляющих приложение, и их связывание полностью отвечает контекст приложения. В составе фреймворка Spring имеется несколько реализаций контекста приложения.

Поскольку *компоненты* приложения объявлены в XML-файле `knights.xml`, в качестве контекста приложения может использоваться класс `ClassPathXmlApplicationContext`. Реализация контекста в Spring загружает контекст из одного или более XML-файлов, находящихся в библиотеке классов (`classpath`). Метод `main()` в листинге 1.7 использует `ClassPathXmlApplicationContext`, чтобы загрузить `knight.xml` и получить ссылку на объект `Knight`.

### Листинг 1.7. KnightMain.java загружает контекст Spring, содержащий объект Knight

---

```
package com.springinaction.knights;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class KnightMain {
    public static void main(String[] args) {
        // Загрузка контекста Spring
        ApplicationContext context =
            new ClassPathXmlApplicationContext("knights.xml");

        // Получение компонента knight
        Knight knight = (Knight) context.getBean("knight");
        // Использование компонента knight
        knight.embarcOnQuest();
    }
}
```

---

Здесь метод `main()` создает контекст приложения Spring на основе файла `knights.xml`. Затем использует контекст как фабрику для извлечения компонента с идентификатором «knight». Получив ссылку на объект `Knight`, он вызывает метод `embarcOnQuest()`, чтобы отправить рыцаря выполнять задание. Обратите внимание, что этот класс ничего не знает о задании `Quest`, переданном рыцарю. Собственно говоря, он даже не знает, что имеет дело с классом `BraveKnight`. Только файл `knights.xml` имеет полную информацию о реализациях, участвующих в работе.

На этом мы закончим краткое знакомство с приемом внедрения зависимостей. Дополнительные примеры применения DI будут встречаться на протяжении всей книги. Желающим поближе познакомиться с этим приемом я рекомендую прочитать книгу Дханжи Прасанна (Dhanji R. Prasanna) «Dependency Injection».

А теперь познакомимся с другими стратегиями упрощения программирования на языке Java — декларативным программированием посредством *аспектов*.

### 1.1.3. Применение аспектно-ориентированного программирования

Хотя DI делает возможным ослабить связь между компонентами приложения, *аспектно-ориентированное программирование* позво-



ляет оформлять функциональность, используемую в приложении, в виде многократно используемых компонентов.

Аспектно-ориентированное программирование часто определяют как прием программирования, поддерживающий разделение задач в пределах программной системы. Системы конструируются из нескольких компонентов, каждый из которых отвечает за определенную часть функциональности. Зачастую эти компоненты также несут дополнительную ответственность сверх своей основной функциональности. Системные службы, такие как журналирование, управление транзакциями и безопасность, часто находят свое отражение в компонентах, основная задача которых заключается в чем-то другом. Такие системные службы обычно называют *сквозными задачами*, потому что в их работу может вовлекаться несколько компонентов системы.

Распространение этих задач на несколько компонентов влечет за собой увеличение сложности программного кода:

- ❑ программный код, реализующий решение общесистемных задач, дублируется в разных компонентах. Это означает, что в случае необходимости что-то изменить в этой реализации придется просмотреть множество компонентов. Даже если вынести решение задачи в отдельный модуль, чтобы компонентам оставалось только вызвать единственный метод, все равно вызов этого метода будет дублироваться в разных местах;
- ❑ компоненты захламляются программным кодом, не имеющим прямого отношения к их основной функциональности. Метод добавления записи в адресную книгу должен заботиться только о том, как добавить адрес, а не о безопасности или о необходимости использования транзакции.

Эта сложность иллюстрируется на рис. 1.2. Бизнес-объекты слева слишком тесно связаны с системными службами. Мало того, что каждый объект знает, что его операции должны фиксироваться в журнале, соответствовать требованиям безопасности и выполняться в рамках транзакции, они еще несут ответственность за использование этих служб.

АОР делает возможным отделение этих служб и декларативное их применение к необходимым компонентам. Благодаря этому компоненты могут сосредоточиться на решении собственных задач, полностью игнорируя системные службы, которые могут быть вовлечены в общий процесс. Проще говоря, аспекты позволяют сохранить простоту РОЮ.