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

Конспект лекции №2
Фреймворк Spring. Внедрение зависимостей.

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Предисловие

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. В отличие от других существующих языков, Java позволил писать сложные приложения, состоящие из отдельных частей.

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.

2 Введение

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 (Enterprise JavaBeans). 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 .

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.

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 большую 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.

3 Модули Spring

Фреймворк Spring Framework состоит из модулей, подразделяющихся на шесть категорий.

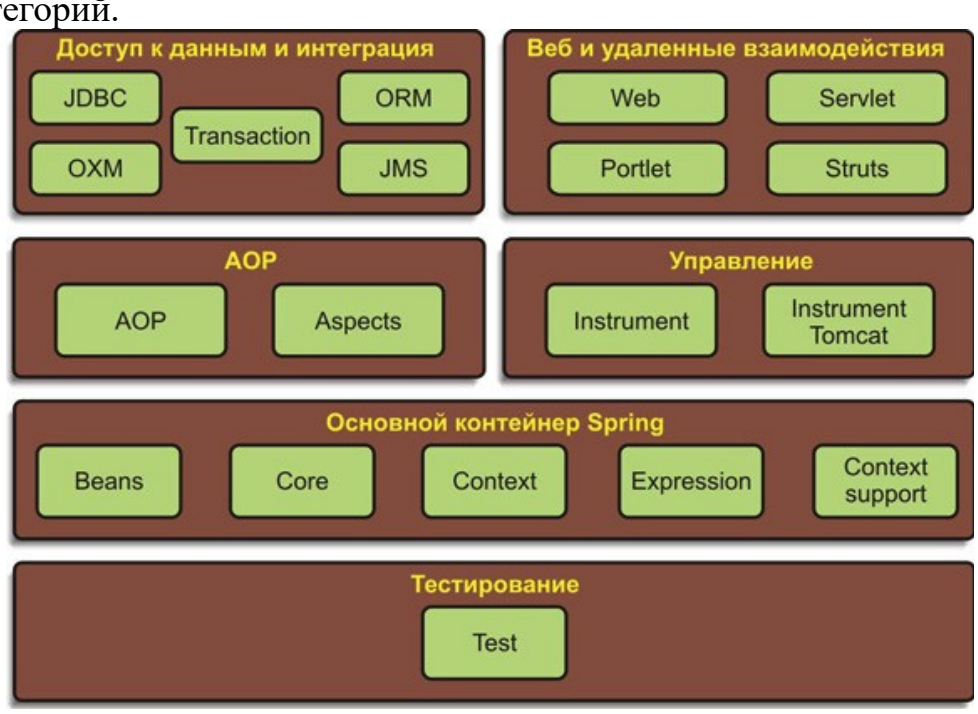

Рисунок 1 – Модули Spring

В совокупности эти модули-категории предоставляют все необходимое для разработки корпоративных приложений. Но никто не обязывает основывать приложение исключительно на фреймворке Spring. Вы свободны в выборе модулей, необходимых в приложении, и в поиске альтернатив, если Spring не отвечает всем требованиям. Фактически фреймворк Spring имеет точки интеграции с некоторыми другими фреймворками и библиотеками, что устраняет необходимость писать их самостоятельно.

Рассмотрим некоторые модули Spring по отдельности, чтобы увидеть, как они вписываются в общую архитектуру фреймворка.

3.1 Основной контейнер Spring

Центральное положение в фреймворке Spring занимает *контейнер*, управляющий процессом создания и настройки компонентов приложения. Этот модуль содержит фабрику компонентов, обеспечивающую внедрение зависимостей. На фабрике компонентов покоятся несколько реализаций контекста приложения Spring, каждый из которых предоставляет различные способы конфигурирования Spring.

3.2 Модуль AOP

Фреймворк Spring обеспечивает богатую поддержку аспектно-ориентированного программирования в своем модуле AOP. Данный модуль служит основой при разработке аспектов в приложении, построенном на основе Spring. Как и DI, AOP способствует ослаблению связей между прикладными объектами. Однако поддержка AOP позволяет отделять аспекты приложения (такие как транзакции и безопасность) от объектов, к которым они применяются.

3.3 Доступ к данным и интеграция

Работа с JDBC зачастую сводится к обширному использованию шаблонного кода, который устанавливает соединение, создает SQL-запрос, обрабатывает результаты запроса, а затем закрывает соединение. Модуль поддержки JDBC объектов доступа к данным (Data Access Objects, DAO) в Spring абстрагирует шаблонный код и позволяет сохранить простым и прозрачным программный код, реализующий операции с базами данных, а также предотвратить проблемы, возникающие в результате ошибки освобождения ресурсов. Этот модуль также образует слой важных исключений, основанных на сообщениях об ошибках, посылаемых некоторыми серверами баз данных. Благодаря этому вам больше не придется расшифровывать непонятные сообщения об ошибках SQL!

Для тех, кто предпочитает использовать инструменты объектно-реляционного отображения (Object Relational Mapping, ORM) поверх JDBC, фреймворк Spring предоставляет модуль ORM. Поддержка ORM в Spring основана на поддержке DAO, обеспечивающей удобный способ создания

объектов доступа к данным для некоторых ORM-решений. Фреймворк Spring не пытается реализовать свое собственное ORM-решение, а просто предоставляет рычаги управления некоторыми популярными фреймворками ORM, включая Hibernate , Java Persistence API, Java Data Objects и iBATIS SQL Maps.

3.4 Веб и удаленные взаимодействия

Парадигма *модель–представление–контроллер* (Model–View–Controller , MVC) часто используется при создании веб-приложений, в которых пользовательский интерфейс отделен от логики работы приложения. Язык Java не испытывает недостатка в фреймворках MVC, благодаря поддержке Apache Struts , JSF , WebWork и Tapestry, являющихся наиболее популярными реализациями MVC.

Несмотря на то что фреймворк Spring легко интегрируется с различными популярными фреймворками MVC, его модуль поддержки веб- и удаленных взаимодействий включает собственный фреймворк MVC, который использует приемы создания слабо связанных объектов в веб-слое приложения. Этот фреймворк имеет две разновидности: фреймворк на основе сервлетов, для создания обычных веб-приложений, и фреймворк на основе портлетов, для создания приложений на основе API Java-портлетов.

В дополнение к поддержке создания пользовательского интерфейса в веб-приложениях этот модуль также предоставляет поддержку удаленных взаимодействий для создания приложений, взаимодействующих с другими приложениями. В состав средств удаленных взаимодействий в Spring входят механизм *вызова удаленных методов* (Remote Method Invocation, RMI), Hessian, Burlap, JAX-WS и собственный механизм вызова через протокол HTTP.

Остальные модули Spring рассматривать не будем.

4 Контейнер компонентов

В приложениях на основе фреймворка Spring прикладные объекты располагаются внутри контейнера Spring. Как показано на Рисунке 2, контейнер создает объекты, связывает их друг с другом, конфигурирует и управляет их полным жизненным циклом, от за рождения до самой их смерти (или от оператора new до вызова метода finalize()).

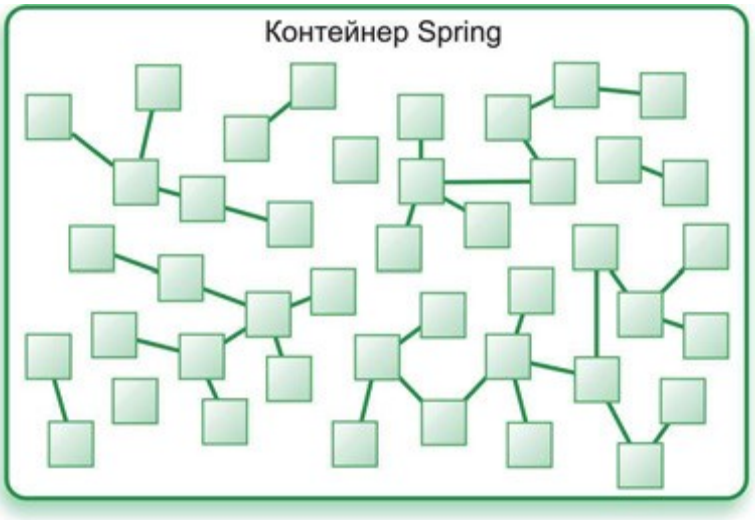

Рисунок 2 – Объекты в приложениях на основе фреймворка Spring создаются, связываются между собой и существуют внутри контейнера Spring

Контейнер находится в ядре фреймворка Spring Framework. Для управления компонентами, составляющими приложение, он использует прием внедрения зависимостей (DI). Управление включает создание взаимосвязей между взаимодействующими компонентами. Фактически эти объекты яснее и проще для понимания, поддерживают возможность повторного использования и легко поддаются тестированию.

Фреймворк Spring имеет не один контейнер. В его состав входят несколько реализаций контейнера, которые подразделяются на два разных типа. *Фабрики компонентов* (bean factories) (определяются интерфейсом org.springframework.beans.factory.BeanFactory) – самые простые из контейнеров, обеспечивающие базовую поддержку DI. *Контекст приложений* (application contexts) (определяется интерфейсом org.springframework.context.ApplicationContext) основан на понятии фабрик компонентов и реализует прикладные службы фреймворка, такие как возможность приема текстовых сообщений из файлов свойств и возможность подписывать другие программные компоненты на события, возникающие в приложении.

С фреймворком Spring можно работать, используя и фабрики компонентов, и контексты приложений, но для большинства приложений фабрики компонентов часто оказываются слишком низкоуровневым инструментом. Поэтому контексты приложений выглядят более предпочтительно, чем фабрики компонентов.

4.1 Работа с контекстом приложения

В составе Spring имеется несколько разновидностей *контекстов приложений*. Три из них используются наиболее часто:

ClassPathXmlApplicationContext – загружает определение контекста из XML-файла, расположенного в библиотеке классов (classpath), и обрабатывает файлы с определениями контекстов как ресурсы;

FileSystemXmlApplicationContext – загружает определение контекста из XML-файла в файловой системе;

XmlWebApplicationContext – загружает определение контекста из XML-файла, содержащегося внутри веб-приложения.

5 Внедрение зависимостей

Теперь, после обилия новых терминов и информации, обо всем по порядку. Давайте разберемся, что же такое Внедрение зависимостей.

Внедрение зависимости (англ. Dependency injection, DI) — процесс предоставления внешней зависимости программному компоненту. Является специфичной формой «инверсии управления» (англ. Inversion of control, IoC), когда она применяется к управлению зависимостями. В полном соответствии с принципом единственной обязанности объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

Приведем пример зависимостей. Например, у нас есть класс автомобиль и класс двигателя. Для того чтобы автомобилю поехать, ему необходим двигатель.

```
public class Engine{
// параметры двигателя
}

public class Car {
	private Engine engine;

	public void drive(){
		engine = new Engine();
		// код движения автомобиля
	}
}
```

Видна сильная связность, в данном случае автомобиль может поехать только с определенным двигателем. Как решить данную проблему, чтобы создавать автомобиль с любым двигателем? Одним из решений является использование интерфейсов.

```
interface Engine{
// общие параметры двигателя
}

class GasEngine implements Engine{
// параметры бензинового двигателя
}

class ElectricalEngine implements Engine{
// параметры электродвигателя
}
```

```
public class Car {
private Engine engine;

public void drive(){
	engine = new GasEngine();
	// или
	engine = new ElectricalEngine();

	// код движения автомобиля
}
}
```

Теперь мы получили слабую зависимость. Наш автомобиль может использовать любой двигатель, но мы все еще самостоятельно вручную задаем тип используемого двигателя. Здесь еще нет никакого внедрения зависимостей, мы лишь ослабили связь, но далее нам нужно поговорить о таком подходе как инверсия контроля.

5.1 Inversion of Control

Немного отвлечемся от примера и рассмотрим понятие инверсии управления.

Инверсия управления (англ. Inversion of Control, IoC) — важный принцип объектно-ориентированного программирования, используемый для уменьшения зацепления в компьютерных программах. Также архитектурное решение интеграции, упрощающее расширение возможностей системы, при котором поток управления программы контролируется фреймворком.

В обычной программе программист сам решает, в какой последовательности делать вызовы процедур. Но, если используется фреймворк, программист может разместить свой код в определенных точках выполнения (используя callback или другие механизмы), затем запустить «главную функцию» фреймворка, которая обеспечит все выполнение и вызовет код программиста тогда, когда это будет необходимо. Как следствие, происходит утеря контроля над выполнением кода — это и называется инверсией управления (фреймворк управляет кодом программиста, а не программист управляет фреймворком).

Инверсия управления бывает не только в фреймворках, но и в некоторых библиотеках (но обычно библиотеки не создают инверсии управления — они предоставляют набор функций, которые должен вызывать программист).

Принцип инверсии управления состоит в том, что сущность не сама задает свои зависимости, а когда этой сущности зависимости поставляются извне.

Как мы видим, в нашем предыдущем примере класс Car зависит от классов реализующих Engine. Вместо этого, мы хотим передавать любой объект типа Engine в класс Car, тем самым уменьшая зависимость, это и есть яркий пример инверсии управления. Применим данный архитектурный подход.

```
public class Car {
	private Engine engine;

	// зависимость внедряется извне
	public Car(Engine engine) {
		this.engine = engine;
	}

	public void drive(){
		// объекты больше не создаются

		// код движения автомобиля
	}
}
```

Таким образом мы можем передать любой двигатель автомобилю для движения в момент создания автомобиля.

6 Подготовка конфигурации Spring

Как уже отмечалось, основу фреймворка Spring составляет *контейнер*. Если не выполнить настройку Spring, у нас в руках окажется пустой контейнер, не имеющий никакой практической ценности. Поэтому необходимо сконфигурировать фреймворк Spring, чтобы сообщить ему, какие компоненты должны находиться в контейнере и как они должны быть связаны между собой.

Начиная с версии Spring 3.0, существуют два пути настройки компонентов в контейнере Spring. Первый путь – традиционный, когда конфигурация описывается в одном или более XML-файлах. Но версия Spring 3.0 предлагает еще один путь, основанный на программном коде Java. Сейчас мы рассмотрим традиционный способ, опирающийся на XML-файлы, а новый способ, основанный на программном коде Java, мы рассмотрим в следующей теме.

При объявлении компонентов в XML-файле роль корневого элемента конфигурационного файла играет элемент <beans> из схемы beans. Ниже показано, как выглядит типичный XML-файл с конфигурацией фреймворка 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"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd">

<!-- Здесь должны находиться объявления компонентов -->

</beans>
```

Внутрь элемента <beans> можно поместить полное описание конфигурации Spring, включая объявления <bean>. Однако пространство имен

beans не является единственным, поддерживаемым фреймворком Spring. Всего основной фреймворк Spring Framework поддерживает десять пространств имен для описания конфигураций.

7 Объявление простого компонента

Изменим класс GasEngine, добавим ему поле:

```
public class GasEngine implements Engine {

	private double volume = 1.6; // объем двигателя

	public double getVolume() {
		return volume;
	}
}
```

Объявление класса GasEngine в конфигурационном файле Spring выглядит следующим образом:

```
<bean id="engineBean"
	class="ru.firstapp.GasEngine">
</bean>
```

Элемент <bean> является самым основным в конфигурационном файле Spring. Он предписывает фреймворку Spring создать объект. Здесь мы объявили объект engineBean как компонент, управляемый фреймворком Spring, используя чуть ли не самое простейшее объявление <bean>, какое только возможно. Атрибут id присваивает компоненту имя, которое можно использовать для обращения к компоненту в контейнере Spring. Этот компонент будет известен как engineBean. А атрибут class сообщает фреймворку тип компонента. Компонент engineBean – это объект класса GasEngine.

В процессе загрузки компонентов контейнер Spring создаст компонент engineBean, используя конструктор по умолчанию.

8 Контекст приложения

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 (бины)* приложения объявлены в XML-файле componentContext.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() ниже использует ClassPathXmlApplicationContext, чтобы загрузить componentContext.xml, и получить ссылку на объект GasEngine.

```
public class TestSpring {
 public static void main(String[] args) {
```

```
            ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

            GasEngine engine = context.getBean("engineBean", GasEngine.class);

            System.out.println("Объем двигателя равен " + engine.getVolume());

            context.close();
      }
      }
```

Чтобы получить объект engineBean, мы вызываем метод getBean() контекста.

Результатом запуска класса TestSpring будет вывод в консоль сообщения: «Объем двигателя равен 1.6». Мы продемонстрировали как создавать компоненты, как загружать конфигурацию, теперь поговорим о непосредственно внедрении зависимостей.

9 Внедрение зависимостей в Spring

Теперь продемонстрируем как внедрение зависимостей реализуется в Spring.

Процесс создания связей между прикладными компонентами называется *связыванием* (wiring). Фреймворк Spring поддерживает множество способов связывания компонентов, но наиболее общим из них является способ на основе XML. Мы уже подготовили простую конфигурацию, теперь в ней будем прописывать зависимости.

9.1 Внедрение через конструктор

9.1.1 Внедрение простых значений

В предыдущем примере у нас было задано поле объема двигателя непосредственно в классе. Конечно же такой подход нас не устраивает. Изменим конструктор, чтобы он принимал величину объема двигателя в качестве параметра конструктора, а также создадим конструктор по умолчанию:

```
      public class GasEngine implements Engine {

      private double volume; // объем двигателя

      public GasEngine() {
            volume = 1.6;
      }

      public GasEngine(double volume){
            this.volume = volume;
      }

      public double getVolume() {
            return volume;
      }
      }
```

Если мы хотим передать параметр в конструктор наша конфигурация будет выглядеть следующим образом:

```
<bean id="engineBean"
	class="ru.firstapp.GasEngine">
	<constructor-arg value="3.0" />
</bean>
```

Элемент <constructor-arg> здесь используется для передачи фреймворку Spring дополнительной информации, касающейся создания компонента. Если элемент <constructor-arg> не указан, то будет использоваться конструктор по умолчанию. Но здесь присутствует элемент <constructor-arg> со значением в атрибуте value, поэтому фреймворк задействует другой конструктор.

Теперь при выполнении данного кода:

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

GasEngine engine = context.getBean("engineBean", GasEngine.class);

System.out.println("Объем двигателя равен " + engine.getVolume());
```

Будет выведено следующее:

```
Объем двигателя равен 3.0
```

9.1.2 Внедрение ссылок через конструкторы

Здесь мы наконец подошли к тому примеру внедрения зависимостей, который рассматривали в самом начале без использования фреймворка Spring.

Мы хотим внедрить ссылку на Engine в объект Car. Как же это сделать при помощи Spring? Для начала изменим наш код, чтобы он стал более информативен. Следите внимательно за кодом.

Интерфейс двигателя будет выглядеть следующим образом:

```
public interface Engine {
	public double getPower();
}
```

Внутри класса GasEngine для примера введем некоторый коэффициент (абстрактный, просто для примера) который будет использоваться при расчете мощности:

```
public class GasEngine implements Engine {

private double volume; // объем двигателя
private static final double koef = 57.5; // некоторый коэфициант преобразования
                                         объема к мощности

public GasEngine() {
	volume = 1.6;
}

public GasEngine(double volume){
	this.volume = volume;
```

```
}

public double getVolume() {
	return volume;
}

@Override
public double getPower() {
	return koef * volume;
}
}
```

Тогда реализация нашего электродвигателя будет выглядеть следующим образом:

```
public class ElectricalEngine implements Engine {

private double capacity; // емкость аккумулятора
private static final double koef = 0.0013; // некоторый коэфициент преобразования
                          емкости к мощности

public ElectricalEngine() {
	capacity = 30000;
}

public ElectricalEngine(double capacity) {
	this.capacity = capacity;
}

public double getCapacity() {
	return capacity;
}

@Override
public double getPower(){
	return koef * capacity;
}
}
```

Код автомобиля будет выглядеть следующим образом:

```
public class Car {

private Engine engine;

public Car(Engine engine) {
	this.engine = engine;
}

public void drive(){
	System.out.println("Мы едем с мощностью двигателя в " +
			engine.getPower());
}
}
```

Теперь в xml-конфигурации внедрим в Car ссылку на GasEngine следующим образом:

```
<bean id="engineBean"
	class="ru.firstapp.GasEngine">
```

```
            <constructor-arg value="3.0" />
    </bean>

    <bean id="carBean"
            class="ru.firstapp.Car">
            <constructor-arg ref="engineBean" />
       </bean>
```

Запустим наш тестовый класс со следующим кодом:

```
      ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

      Car car = context.getBean("carBean", Car.class);

      car.drive();

      context.close();
```

И получим сообщение:

```
Мы едем с мощностью двигателя в 172.5
```

Что же произошло в данном коде?

В тестовом классе создается `ClassPathXmlApplicationContext` контекст приложения Spring на основе файла componentContext.xml. Затем использует контекст как фабрику для извлечения компонента с идентификатором «carBean». Который, в свою очередь, создается не с помощью конструктора по умолчанию, а с помощью конструктора с параметром, который указан в конфигурации. Получив ссылку на объект Car мы вызываем метод drive(), чтобы вывести сообщение. Обратите внимание, что этот класс ничего не знает о двигателе GasEngine, переданном автомобилю. Только файл componentContext.xml имеет полную информацию о реализациях, участвующих в работе.

Таким легким способом мы можем автомобилю передать другой двигатель:

```
      <bean id="engineBean"
            class="ru.firstapp.ElectricalEngine">
            <constructor-arg value="30000.0" />
    </bean>

    <bean id="carBean"
            class="ru.firstapp.Car">
            <constructor-arg ref="engineBean" />
    </bean>
```

Запустим приложение ничего в нем не меняя, мы изменили только xml-конфигурацию. И тогда получим совсем другую мощность, рассчитываемую в методе уже класса ElectricalEngine. Результат выполнения будет следущий:

```
Мы едем с мощностью двигателя в 39.0
```

9.2 Внедрение в свойства компонентов

Зачастую нам не хочется передавать параметры через конструктор, мы выполняем это через set-функции (иногда будет использоваться сленговый термин сеттеры). Продемонстрируем как это можно выполнить с помощью фреймворка Spring. Очередной раз изменим наш GasEngine, уберем конструктор с параметром и создадим сеттер.

```
public void setVolume(double volume) {
	this.volume = volume;
}
```

Тогда для того чтобы внедрить значение в свойство необходимо описать следующую конфигурацию:

```
<bean id="engineBean"
	class="ru.firstapp.GasEngine">
	<property name="volume" value="3.0" />
</bean>
```

Как только будет создан экземпляр класса GasEngine, Spring воспользуется методами записи, чтобы присвоить указанные значения свойствам, описанным элементами <property>. Элемент <property> в этом фрагменте XML предписывает фреймворку Spring вызвать метод setVolume() для записи значения "3.0" в свойство volume. (Обратите на это внимание, что атрибуту name нужно передавать именно наименование свойства, а не метода. Spring сам догадается, что ему нужно внедрить значение в set-метод) В данном случае для внедрения числового значения в свойство используется атрибут value элемента <property>. Но элемент <property> позволяет внедрять не только числовые (int, float, double и другие) значения. В атрибуте value можно также указывать строковые и логические значения.

Подобным же образом в свойства можно внедрить ссылки используя атрибут ref, а не value. Пример для этого приводить не будем.

9.3 Подстановка переменных-заполнителей

Мы внедряли простые значения непосредственно в xml-конфигурации. Но что если мы хотим создать отдельный файл свойств, откуда хотим просто передавать значения в конфигурацию? Для этого в Spring можно воспользоваться механизмом подстановки переменных заполнителей.

В версиях Spring ниже 2.5 для настройки механизма подстановки переменных-заполнителей в определении контекста Spring необходимо было объявить компонент PropertyPlaceholderConfigurer. Хотя это было совсем несложно, тем не менее начиная с версии Spring 2.5 эта процедура была упрощена еще больше добавлением нового элемента <context:property-placeholder> в конфигурационное пространство имен context . Теперь механизм подстановки переменных-заполнителей можно настраивать так:

<context:property-placeholder location="classpath:gasEngine.properties"/>

Естественно, чтобы это заработало, необходимо создать в папке resources файл gasEngine.properties, в него внести пару ключ-значение gasEngine.volume=3.0.

Согласно этим настройкам, механизм подстановки переменных-заполнителей будет извлекать значения свойств из файла с именем gasEngine..properties, находящимся в корневом каталоге библиотеки классов (classpath).

Теперь можно заменить значения, объявленные в конфигурации Spring, переменными-заполнителями, опираясь на определения в файле gasEngine.properties:

<property name="volume" value="${gasEngine.volume}"/>

10 Жизненный цикл компонента

В традиционных Java-приложениях жизненный цикл компонента довольно прост. Сначала компонент создается с помощью ключевого слова new (при этом, возможно, выполняется его десериализация), после чего он готов к использованию. Когда компонент перестает использоваться, он утилизируется сборщиком мусора и в конечном счете попадает в большой «битоприемник» на небесах.

Напротив, жизненный цикл компонента внутри контейнера Spring намного сложнее. Иметь представление о жизненном цикле компонента в контейнере Spring очень важно, потому что в этом случае появляются новые возможности управления процессом создания компонента. Рисунок 3 иллюстрирует начальные этапы жизненного цикла типичного компонента, которые он минует, прежде чем будет готов к использованию.

Как показано на рисунке, фабрика компонентов выполняет несколько подготовительных операций, перед тем как компонент будет готов к использованию. Исследуем рисунок 3 более детально.

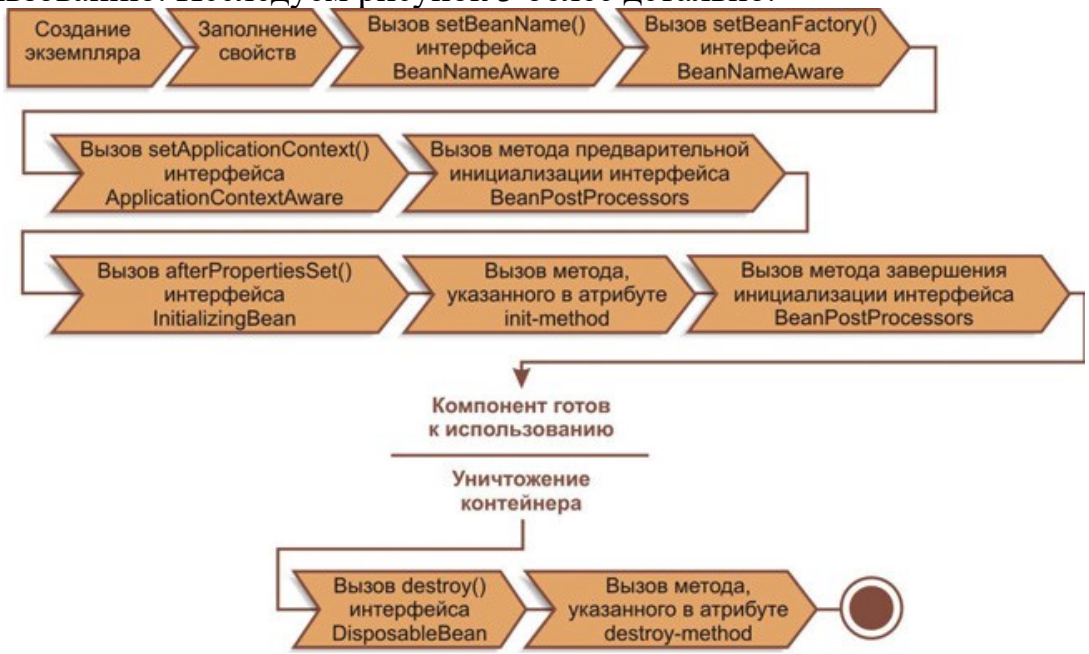

Рисунок 3 – От создания до уничтожения в контейнере Spring компонент преодолевает несколько этапов. Spring позволяет настроить выполнение каждого из этапов

1. Spring создает экземпляр компонента.
2. Spring внедряет значения и ссылки на компоненты в свойства данного компонента.
3. Если компонент реализует интерфейс BeanNameAware, Spring передает идентификатор компонента методу setBeanName() .
4. Если компонент реализует интерфейс BeanFactoryAware, Spring вызывает метод setBeanFactory() , передавая ему саму фабрику компонентов.
5. Если компонент реализует интерфейс ApplicationContextAware, Spring вызывает метод setApplicationContext(), передавая ему ссылку на вмещающий контекст приложения.
6. Если какие-либо из компонентов реализуют интерфейс Bean-Post Processor, Spring вызывает их методы postProcessBeforeInitialization().
7. Если какие-либо из компонентов реализуют интерфейс InitializingBean , Spring вызывает их методы afterPropertiesSet(). Аналогично, если компонент был объявлен с атрибутом init-method, вызывается указанный метод инициализации.
8. Если какие-либо из компонентов реализуют интерфейс BeanPostProcessor , Spring вызывает их методы postProcessAfterInitialization().
9. В этот момент компонент готов к использованию приложением и будет сохраняться в контексте приложения, пока он не будет уничтожен.
10. Если какие-либо из компонентов реализуют интерфейс DisposableBean, Spring вызывает их методы destroy(). Аналогично, если компонент был объявлен с атрибутом destroy-method, вызывается указанный метод.