# Фреймворк Spring. Внедрение зависимостей.

Составил: Черниговский А.С.
Старший преподаватель кафедры “Информатика”
ИКИТ СФУ

# Предисловие

- 1996 г., JavaBeans
  - Набор приемов программирования, позволяющих повторно использовать простые Java-объекты и легко конструировать из них более сложные приложения
- 1998 г., Enterprise JavaBeans
  - 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.

# Spring. Введение

- Род Джонсон, Expert One-on-One: J2EE Design and Development
- Фундаментальная идея — упрощение разработки приложений на языке Java
- spring.io
- Огромное число проектов и примеров!

# POJO-модель

- Plain Old Java Object, «старый добрый объект Java»
- Простой Java-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели.

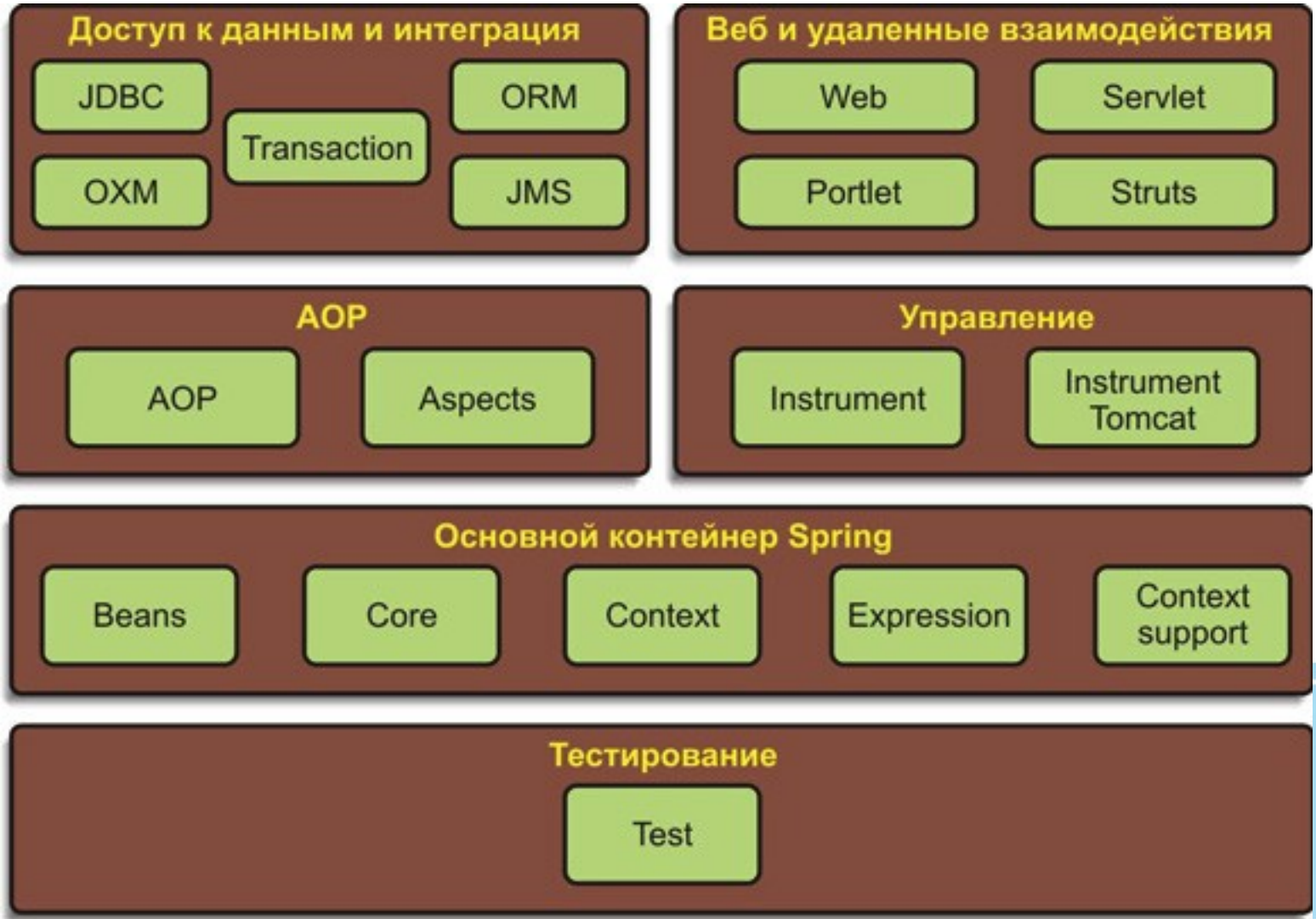
# IoC-контейнер Spring

- Inversion of Control
- Он позволяет объединять в единое целое разные части приложения, обеспечивая согласованность архитектуры.

# Преимущества

- Все классы в приложении являются простыми POJO-классами. 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.
- 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.
- И многие другие

# Модули


Доступ к данным и интеграция
JDBC
Transaction
ORM
OXM
JMS
Веб и удаленные взаимодействия
Web
Servlet
Portlet
Struts
AOP
AOP
Aspects
Управление
Instrument
Instrument Tomcat
Основной контейнер Spring
Beans
Core
Context
Expression
Context support
Тестирование
Test

# 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.

# Модуль AOP

- Пока не рассматривается, так как не знаю как его применить в практических работах :)

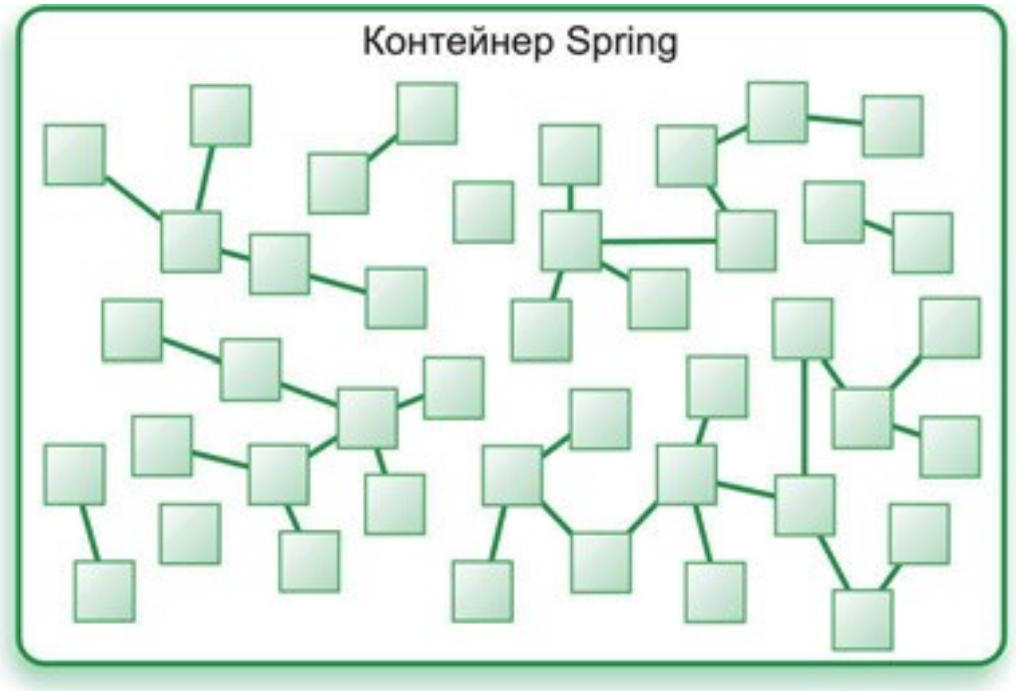
# Доступ к данным и интеграция

- JDBC (Java database connectivity)
- DAO (Data Access Objects)
- ORM (Object Relational Mapping)
- Hibernate
- JPA (Java Persistence API)

# Веб и удаленное взаимодействие

- MVC (Model — View — Controller)
- Слабо связные объекты в веб-слое

# Контейнер компонентов



# Контейнер Spring

- DI (Dipendency Injection)
- Фабрики компонентов (bean factories)
- Контекст приложений (application context)
- Чаще работа производится с контекстом приложений

# Контексты приложений

- Контекст создаёт и хранит экземпляры классов вашего приложения, определяет их зависимости друг с другом и автоматически их задаёт.
- 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.

# Внедрение зависимостей

- Внедрение зависимости (англ. Dependency injection, DI) — процесс предоставления внешней зависимости программному компоненту. Является специфичной формой «инверсии управления» (англ. Inversion of control, IoC).

# Сильная связность

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

public class Car {
    private Engine engine;

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

# Используем интерфейс

```
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
   }
}
```

# Inversion of Control

- 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

```
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
}
}
```

# Подготовка конфигурации Spring

- Необходимо сконфигурировать фреймворк Spring чтобы сообщить ему, какие компоненты должны находиться внутри.
- XML
- Java-код

# XML-конфигурация

```
<?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>
```

# Объявление простого компонента

```
public class GasEngine implements Engine {

    private double volume = 1.6; // объем двигателя

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

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

# Контекст приложения

- В приложении, созданном на основе Spring, контекст приложения загружает определения компонентов и связывает их вместе.

```
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();
    }
}
```

# Внедрение зависимостей 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.

# Внедрение простых значений через конструктор

```
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;
    }
}
```

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

# Получение объекта из контекста

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

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

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

# Внедрение ссылок через конструкторы

```
public interface Engine {

    public double getPower();

}
```

```
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;
    }
}
```

# Внедрение ссылок через конструкторы

```
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;
   }
}
```

# Внедрение через конструкторы

```
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());
    }
}
```

# Внедрение ссылок через конструкторы

```
<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>
```

# Внедрение через конструкторы

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

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

     car.drive();

    context.close();
```

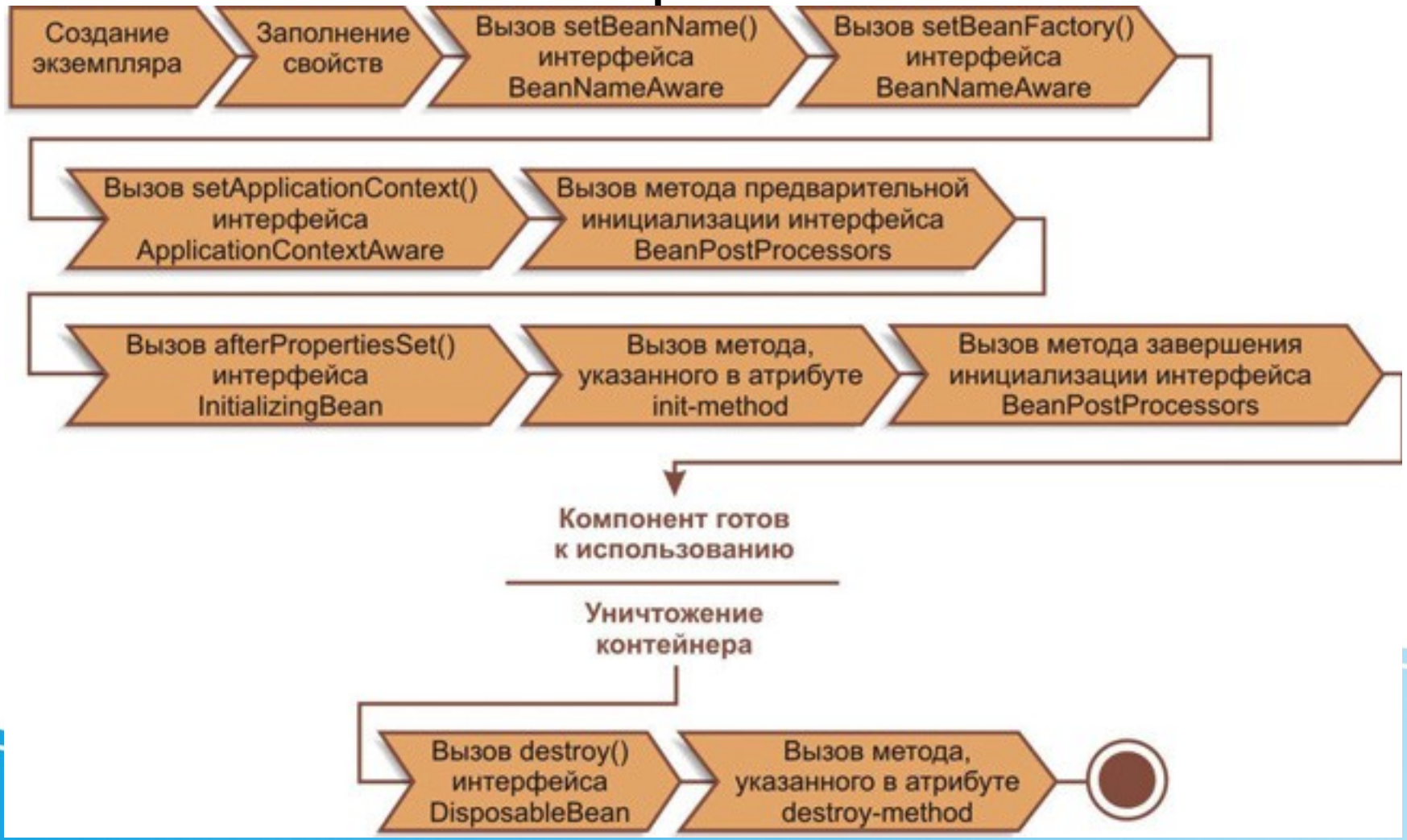
# Внедрение в свойства компонентов

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

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

...
ref
```

# Подстановка переменных-заполнителей

- Что делать если мы хотим использовать файл свойств и подставлять их в конфигурацию?
- <context:property-placeholder location="classpath:gasEngine.properties"/>
- <property name="volume" value="${gasEngine.volume}"/>

# Жизненный цикл компонента



# Жизненный цикл компонента

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.

Спасибо за внимание!