Spring AOP

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

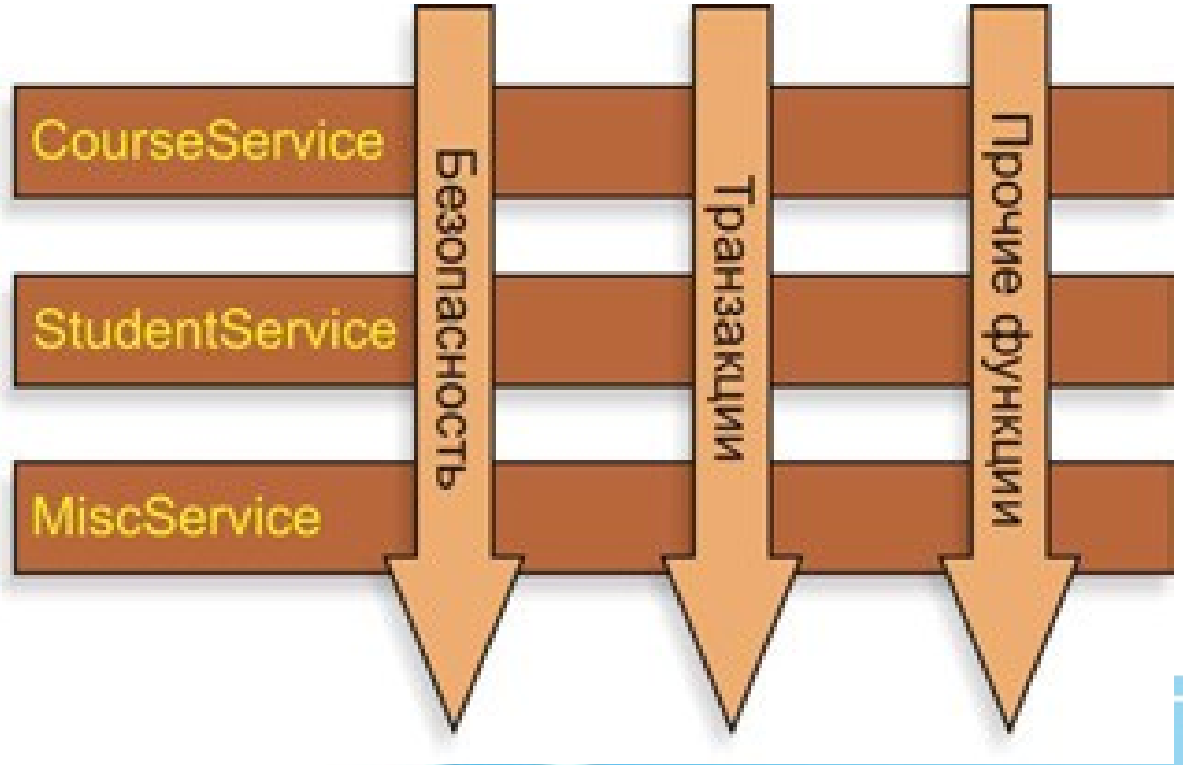
Аспектно-ориентированный Spring

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

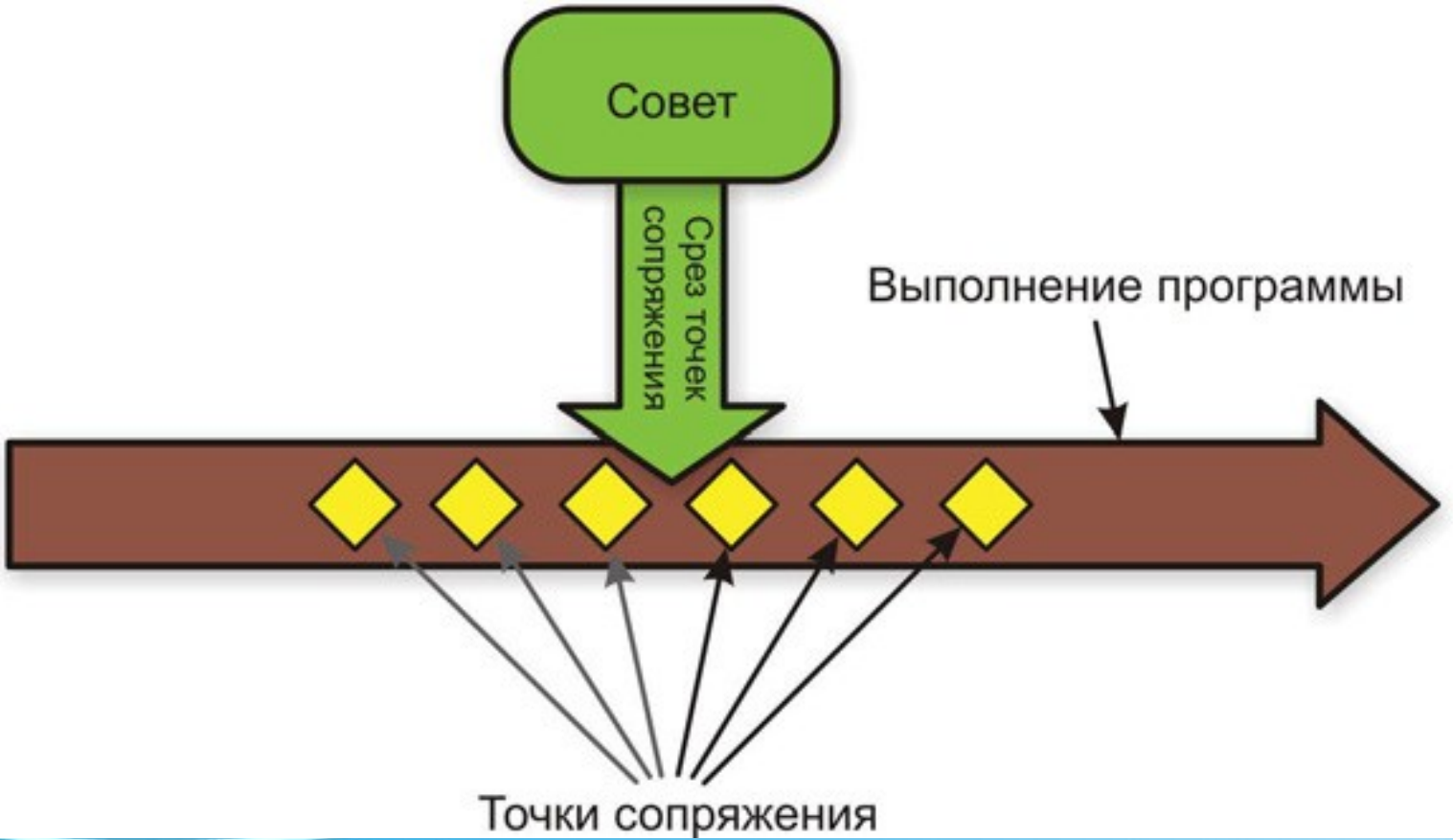
Знакомство с AOP

- Общая объектно-ориентированная методика повторного использования общей функциональности заключается в применении наследования или делегирования. Однако наследование может привести к созданию хрупкой иерархии объектов, если по всему приложению используется один и тот же базовый класс, а реализация делегирования может оказаться слишком громоздкой, поскольку может потребоваться выполнять сложные вызовы объекта-делегата.
- Аспекты предлагают альтернативу наследованию и делегированию, более простую во многих случаях. При использовании аспектно-ориентированного программирования общая функциональность так же определяется в одном месте, но порядок и место применения этой функциональности можно определять декларативно, без изменения класса, к которому применяются новые возможности.

Знакомство с AOP



Определение терминологии AOP



Советы

- В терминах AOP работа аспекта называется **совет**. **Совет** определяет, что и когда делает аспект. В дополнение к описанию работы, выполняемой аспектом, совет учитывает, когда следует ее выполнять.
- Аспекты Spring могут работать с пятью типами советов:
  - до – работа выполняется перед вызовом метода;
  - после – работа выполняется после вызова метода, независимо от результата;
  - после успешного вызова – работа выполняется после вызова метода, если его выполнение завершилось успешно;
  - после исключения – работа выполняется после того, как вызванный метод возбудит исключение;
  - вокруг – аспект обертывает метод, обеспечивая выполнение некоторых операций до и после вызова метода.

Точки сопряжения

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

Срезы множества точек сопряжения

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

Аспекты

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

Внедрение

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

Вплетение

- Вплетение – это процесс применения аспектов к целевому объекту для создания нового, проксированного объекта. Аспекты вплетаются в целевой объект в указанные точки сопряжения. Вплетение может происходить в разные моменты жизненного цикла целевого объекта.
    - Во время компиляции – аспекты вплетаются в целевой объект, когда тот компилируется. Это требует специального компилятора, такого как AspectJ, вплетающего аспекты на этапе компиляции.
    - Во время загрузки класса – вплетение аспектов выполняется в процессе загрузки целевого класса виртуальной машиной JVM. Это требует специального загрузчика, который дополняет байт-код целевого класса перед внедрением его в приложение.
    - Во время выполнения – вплетение аспектов производится во время выполнения приложения. В этом случае контейнер AOP обычно динамически генерирует объект с вплетенным аспектом, представляющий целевой объект.

Поддержка AOP в Spring

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

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

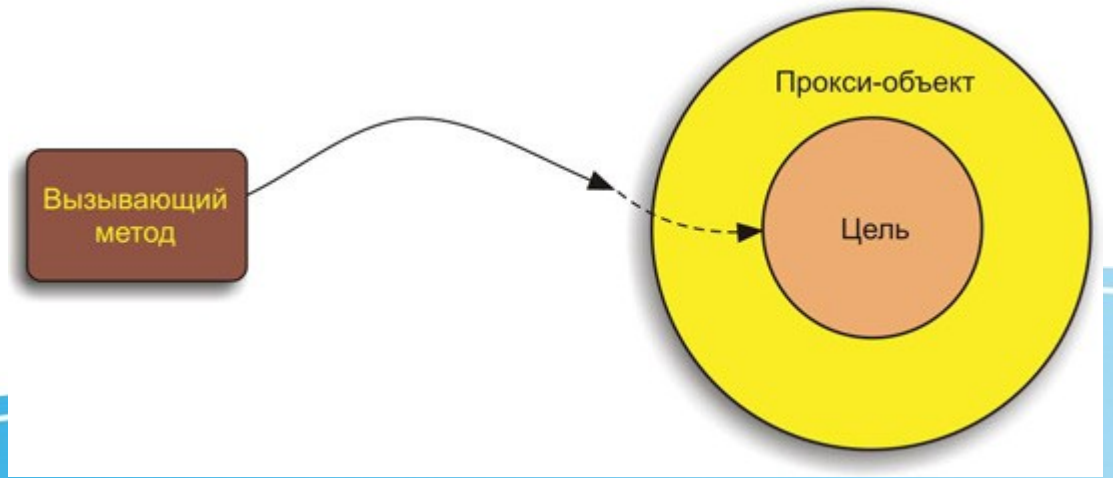
Советы в Spring реализуются на языке Java

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

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

Советы в Spring – это объекты времени выполнения

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



Spring поддерживает точки сопряжения только для методов

- Так как поддержка AOP в Spring основана на использовании динамических прокси-объектов, точками сопряжения могут служить только методы, в отличие от некоторых других фреймворков AOP, таких как AspectJ и JBoss, где помимо методов роль точек сопряжения могут играть поля и конструкторы. Отсутствие в Spring возможности применения аспектов к полям препятствует созданию высокоизбирательных аспектов, например для отслеживания изменений полей в объектах. Невозможность применить аспект к конструктору также препятствует реализации логики, которая должна выполняться в момент создания экземпляра компонента.
- Однако возможности перехватывать вызовы методов должно быть достаточно для удовлетворения если не всех, то большинства потребностей. Если потребуется нечто большее, чем возможность перехвата вызовов методов, всегда можно воспользоваться расширением AspectJ.

Выбор точек сопряжения в описаниях срезов

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

- Конструкции языка выражений описания срезов множества точек сопряжения в AspectJ, поддерживаемые аспектами в Spring:
- args() Ограничивает срез точек сопряжения вызовами методов, чьи аргументы являются экземплярами указанных типов
- @args() Ограничивает срез точек сопряжения вызовами методов, чьи аргументы аннотированы указанными типами аннотаций
- execution() Соответствует точкам сопряжения, которые являются вызовами методов
- this() Ограничивает срез точек сопряжений точками, где ссылка на компонент является ссылкой на прокси-объект указанного типа
- target() Ограничивает срез точек сопряжений точками, где целевой объект имеет указанный тип
- И тд. С некоторыми остальными вы можете ознакомиться в конспекте.

Определение срезов множества точек сопряжения

Определение срезов множества точек сопряжения

Использование указателя bean()

- execution(* com.springinaction.springidol.Instrument.play())

  and bean(eddie)

Объявление аспектов в XML

- Элементы настройки механизма AOP в Spring упрощают объявление аспектов, основанных на POJO:
- <aop:advisor> Определяет объект-советник
- <aop:after> Определяет AOP-совет, выполняемый после вызова метода (независимо от успешности его завершения)
- <aop:after-returning> Определяет AOP-совет, выполняемый после успешного выполнения метода
- <aop:after-throwing> Определяет AOP-совет, выполняемый после возбуждения исключения
- <aop:around> Определяет AOP-совет, выполняемый до и после выполнения метода
- <aop:aspect> Определяет аспект
- <aop:aspectj-autoproxy> Включает поддержку аспектов, управляемых аннотациями, созданными с применением аннотации @AspectJ
- С остальными аннотациями вы можете ознакомиться в конспекте.

Пример

```
package com.springinaction.spring;
public class Audience {
      public void takeSeats() { // Перед выступлением
            System.out.println("The audience is taking their
                        seats.");
      }
      public void turnOffCellPhones() { // Перед
                                                выступлением
            System.out.println("The audience is turning off their cellphones");
      }
      public void applaud() { // После выступления
            System.out.println("CLAP CLAP CLAP CLAP CLAP");
      }
      public void demandRefund() { // После неудачного выступления
            System.out.println("Boo! We want our money back!");
      }
}
```

Регистрация компонента

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

Пример

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

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

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

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

Пример

Бизнес-логика

```
performer.perform();
```

Аспект Audience

```
<aop:before
    method="takeSeats"
    pointcut-ref="performance"/>

<aop:before
    method="turnOffCellPhones"
    pointcut-ref="performance"/>

<aop:after-returning
    method="applaud"
    pointcut-ref="performance"/>

<aop:after-throwing
    method="demandRefund"
    pointcut-ref="performance"/>
```

Логика совета

```
try {
  audience.takeSeats();
  audience.turnOffCellPhones();
  audience.applaud();
} catch (Exception e) {
  audience.demandRefund();
}
```

Определение именованного среза множества точек сопряжения для устранения избыточных определений срезов

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

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

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

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

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

Объявление советов, выполняемых и до, и после

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

                joinpoint.proceed(); // Вызов целевого метода

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

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

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

Передача параметров советам

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

-
-

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

Передача параметров советам

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

- ```
  package com.springinaction.springidol;
  public class Volunteer implements Thinker {
      private String thoughts;
      public void thinkOfSomething(String thoughts) {
          this.thoughts = thoughts;
      }

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

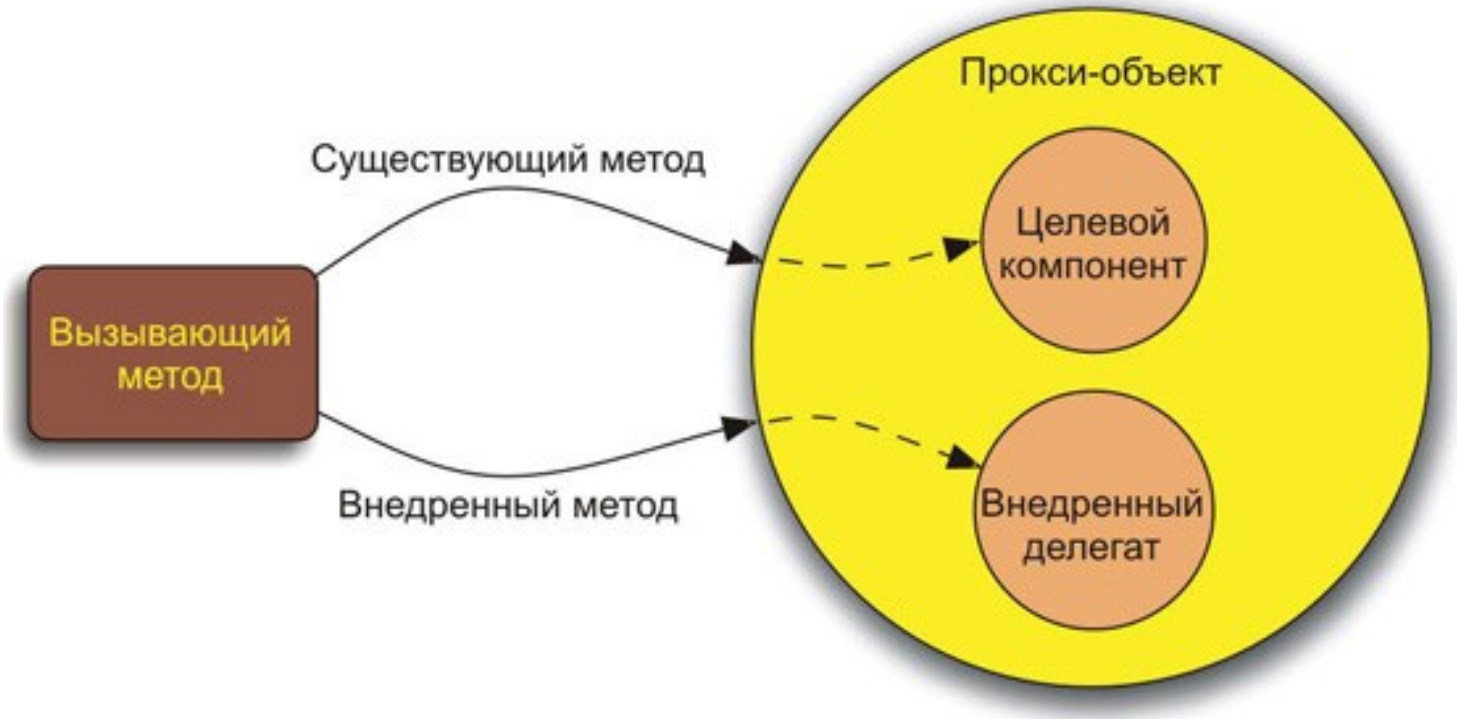
Передача параметров советам

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

Проверка

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

Внедрение новых возможностей с помощью аспектов



Пример

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

Пример

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

Пример

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

Аннотирование аспектов

```
@Aspect public class Audience {
   @Pointcut( // Определение среза
      "execution(* com.springinaction.springidol.Performer.perform(..))")
   public void performance() {
   }
   @Before("performance()")
   public void takeSeats() { // Перед выступлением
      System.out.println("The audience is taking their seats.");
   }
   @Before("performance()") // Перед выступлением
   public void turnOffCellPhones() {
      System.out.println("The audience is turning off their cellphones");
   }
   @AfterReturning("performance()") // После успешного выступления
   public void applaud() {
      System.out.println("CLAP CLAP CLAP CLAP CLAP");
   }
   @AfterThrowing("performance()")
   public void demandRefund() { // После неудачного выступления
      System.out.println("Boo! We want our money back!");
   }
}
```

Создание советов, выполняемых и до, и после

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

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

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

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

Создание советов, выполняемых и до, и после

```
@Aspect
	public class Magician implements MindReader {
		private String thoughts;
// Объявление параметризованного среза множества точек сопряжения
		@Pointcut("execution(* com.springinaction.springidol."
			+ "Thinker.thinkOfSomething(String)) && args(thoughts)")
	public void thinking(String thoughts) {
	}
	@Before("thinking(thoughts)") // Передача параметра в совет
	public void interceptThoughts(String thoughts) {
		System.out.println("Intercepting volunteer’s thoughts : "
			+ thoughts);
		this.thoughts = thoughts;
	}
	public String getThoughts() {
		return thoughts;
	}
}
```

Внедрение интерфейса Contestant с использованием аннотаций @AspectJ

```
@Aspect
public class ContestantIntroducer {

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

Спасибо за внимание!