Аспектно - ориентированное программирование

Лекция №6

Антонов В.В.

Проектирование – определение зависимостей

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.2.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>3.2.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.7.3</version>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Интерфейс Performer и его реализацию

```java
public interface Performer {
    void doSmth();
    String print();
}
```

```java
public class PerformerImpl implements Performer {
    @Override
    public void doSmth() {
        System.out.println(print());
    }
    @Override
    public String print(){
        return "Performer is working...";
    }
}
```

Класс с «советами»

```
public class Aspect {
    private Logger logger;

    public void addAppender(){
        logger = Logger.getRootLogger();
        logger.setLevel(Level.INFO);
        PatternLayout layout = new PatternLayout("%d{ISO8601} [%t] %-5p %c %x - %m%n")
;
        logger.addAppender(new ConsoleAppender(layout));
    }
    public void before(){
        logger.info("Before method...");
    }
    public void after(){
        logger.info("After method...");
    }
    public void afterReturning(){
        logger.info("After returning...");
    }
    public void afterThrowing(){
        logger.info("After throwing...");
    }
}
```

Xml настройки нашего приложения

```
<?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:aop="http://www.spr
ingframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd">
    <bean id="performerBean" class="net.quizful.aop.PerformerImpl"/>
    <bean id="aspectBean" class="net.quizful.aop.Aspect" init-method="addAppender"/>
    <aop:config>
        <aop:aspect ref="aspectBean">
            <aop:pointcut id="performerPointcut" expression="execution (* net.quizful.
aop.PerformerImpl.doSmth(..))"/>
            <aop:before method="before" pointcut-ref="performerPointcut"/>
            <aop:after method="after" pointcut-ref="performerPointcut"/>
        </aop:aspect>
    </aop:config>
</beans>
```

параметре тега pointcut - expression.

```
expression="execution (* net.quizful.aop.PerformerImpl.doSmth(..))"
```

- execution означает, что аспект выполняется только при запуске соответствующего метода doSmth.
- Звездочка перед путем означает, что возвращаемое значение может быть любое
- две точки в скобочках, что аргументы могут быть любые.
- можно указывать путь к интерфейсу, а не класса. Тогда аспект будет работать для всех классов, которые имплементируют данный интерфейс

Главный класс

```
public class App {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationCo
ntext.xml");
        Performer performer = (Performer)context.getBean("performerBean");
        performer.doSmth();
    }
}
```

```
2013-08-31 13:57:53,179 [main] INFO  root  - Before method...
Performer is working...
2013-08-31 13:57:53,181 [main] INFO  root  - After method...
```

Добавим новые советы

```
public class Worker implements Performer {
    @Override
    public void doSmth() {
        System.out.println(print());
    }
    @Override
    public String print() {
        if(Math.random() < 0.5)
            throw new RuntimeException("Worker exception");
        return "Worker is working...";
    }
}
```

Добавим новые бины

```
<!--?xml version="1.0" encoding="UTF-8"?-->
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.or
g/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop" xsi:s
chemalocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd">
    <bean id="aspectBean" class="net.quizful.aop.Aspect" init-method="addAppender">
    <bean id="performerBean" class="net.quizful.aop.PerformerImpl">
    <bean id="workerBean" class="net.quizful.aop.Worker">
    <aop:config>
        <aop:aspect ref="aspectBean">
            <aop:pointcut id="performerPointcut" expression="execution (* net.quizful.
aop.PerformerImpl.doSmth(..))">
            <aop:before method="before" pointcut-ref="performerPointcut">
            <aop:after method="after" pointcut-ref="performerPointcut">
        </aop:after></aop:before></aop:pointcut></aop:aspect>
        <aop:aspect ref="aspectBean">
            <aop:pointcut id="workerPointcut" expression="execution(* net.quizful.aop.
Worker.doSmth(..))">
            <aop:after-throwing method="afterThrowing" pointcut-ref="workerPointcut">
            <aop:after-returning method="afterReturning" pointcut-ref="workerPointcut"
>
        </aop:after-returning></aop:after-throwing></aop:pointcut></aop:aspect>
    </aop:config>
</bean></bean></bean></beans>
```

```
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationCo
ntext.xml");
        Performer worker = (Performer)context.getBean("workerBean");
        worker.doSmth();
    }
```

```
Worker is working...
2013-08-31 14:15:06,322 [main] INFO  root  - After returning...
```

```
Exception in thread "main" java.lang.RuntimeException: Worker exception
2013-08-31 14:22:48,084 [main] INFO  root  - After throwing...
```

Фреймворк PostSharp

- это реализация аспектно-ориентированного подхода для .NET. PostSharp в отличие от многих своих аналогов работает как пост-компилятор, то есть он вносит изменения в MSIL (Microsoft Intermediate Language).
- PostSharp позволяет легко создавать атрибуты, которые меняют поведение методов, полей и типов. Для этого нужно унаследовать класс атрибута от одного из предоставляемых библиотекой базовых классов, реализовать его виртуальные методы и применить этот атрибут.

```
using System;
using PostSharp.Aspects;

namespace HelloAspects
{
    class Program
    {
        private static void Main()
        {
            hello();
        }

        [SayGoodbye]
        private static void hello()
        {
            Console.WriteLine("Hello!");
        }
    }

    [Serializable]
    class SayGoodbyeAttribute : OnMethodBoundaryAspect
    {
        public override void OnExit(MethodExecutionArgs args)
        {
            Console.WriteLine("Goodbye.");
        }
    }
}
```

- Метод OnExit называют советом (advice), он всегда выполняется (даже если выпадет исключение, так как OnExit вызывается из блока finally) после тела метода, к которому применяется атрибут. Помимо него класс OnMethodBoundaryAspect предоставляет ещё три совета:
- **OnEntry** — выполняется перед телом метода;
- **OnSuccess** — выполняется после успешного выполнения тела метода;
- **OnException** — выполнятся после тела метода в случае, если в методе выпало необработанное исключение.

```
using System;
using System.Diagnostics;
using PostSharp.Aspects;

namespace HelloAspects
{
    class Program
    {
        private static void Main()
        {
            Trace.Listeners.Add(new TextWriterTraceListener(Console.Out));
            hello();
        }

        [Trace]
        private static void hello()
        {
            Console.WriteLine("Hello!");
        }
    }

    [Serializable]
    public class TraceAttribute : OnMethodBoundaryAspect
    {
        public override void OnEntry(MethodExecutionArgs args)
        {
            Trace.WriteLine(string.Format("Entering {0}.{1}.", args.Method.DeclaringType.Name, args.Method.Name));
        }

        public override void OnExit(MethodExecutionArgs args)
        {
            Trace.WriteLine(string.Format("Leaving {0}.{1}.", args.Method.DeclaringType.Name, args.Method.Name));
        }
    }
}
```

- В чём же преимущество использования АОП в данном примере? Представим, что у нас есть несколько классов, в каждом из которых много методов и нам необходимо реализовать трассировку. Если не использовать АОП, то придётся в теле каждого метода прописывать **Trace.WriteLine**… Используя же АОП мы выделяем эту сквозную функциональность в отдельную сущность (аспект) и применяем её к методам при помощи атрибута.

PostSharp есть и другие аспекты

- **OnMethodBoundaryAspect:**
- **EventInterceptionAspect**
- **LocationInterceptionAspect**
- **OnExceptionAspect**
- **…**

PostSharp — это удобный инструмент для внедрения АОП в программы, написанные с использованием среды .NET. АОП дополняет ООП, выделяя сквозную функциональсть в отдельные аспекты, избавляется от дублирования кода (принцип DRY – Don't Repeat Yourself) и упрощает архитектуру приложения.

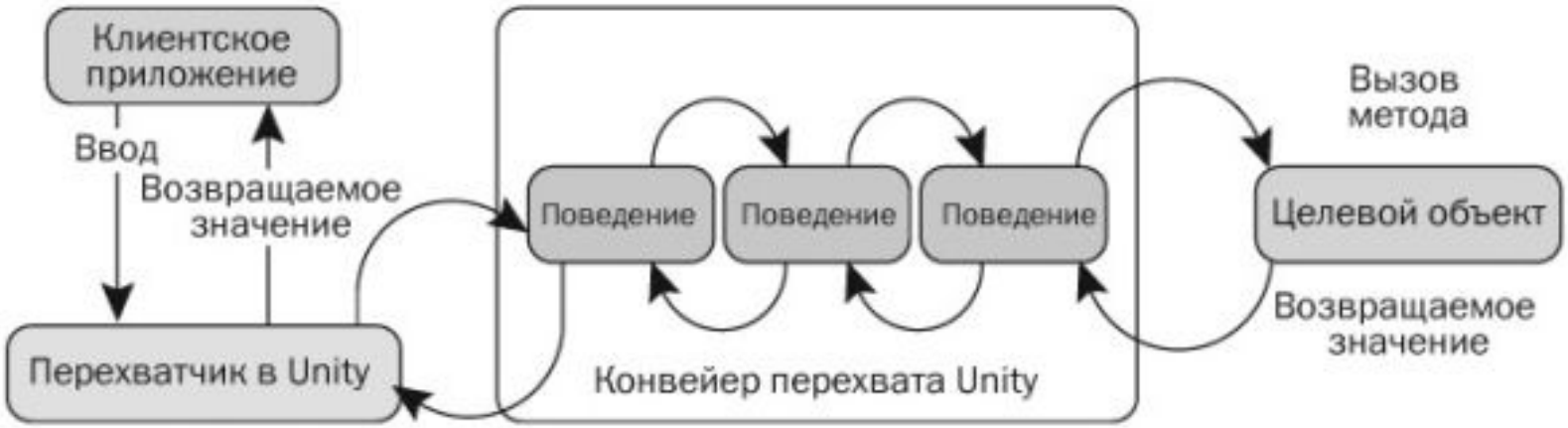
Краткий обзор Unity 2.0

Unity — это блок приложения, доступный как часть проекта Microsoft Enterprise Library, а также в отдельном виде. Microsoft Enterprise Library — это набор блоков приложения, которые снимают часть проблем, связанных с горизонтальным пересечением иерархии и характерных в разработке .NET-приложений (протоколирование, кеширование, шифрование, обработка исключений и др.).

Перехват в Unity 2.0

- Основная концепция перехвата в Unity позволяет разработчикам настраивать цепочку вызовов, необходимых для запуска какого-либо метода некоего объекта.
- Иначе говоря, механизм перехвата Unity захватывает вызовы, выдаваемые для настройки объектов, и изменяет поведение целевых объектов, добавляя дополнительный код до, после и вокруг обычного кода методов.
- Перехват — фактически очень гибкий подход к добавлению нового поведения для объекта в период выполнения, не затрагивающий его исходный код и не влияющий на поведение классов в той же цепочке наследования.
- Перехват в Unity — способ реализации популярного проектировочного шаблона Decorator, разработанного для расширения функциональности объекта в период выполнения и в момент его использования.
- Декоратор (decorator) — это объект-контейнер, который принимает (и поддерживает ссылку на) экземпляр целевого объекта и дополняет его возможности.

Перехват объекта в действии (в Unity 2.0)



Настройка перехвата

```
1. var container = new UnityContainer();
2. container.AddNewExtension<Interception>();
```

перехват реализуется простым добавлением нового расширения к контейнеру, чтобы описать, как будет разрешаться объект.

добавить в конфигурационный файл

- Цель этого сценарного кода — расширение схемы конфигурации новыми элементами и псевдонимами, специфичными для подсистемы перехвата

```
<sectionExtension type="Microsoft.Practices.Unity.InterceptionExtension.
  Configuration.InterceptionConfigurationExtension,
  Microsoft.Practices.Unity.Interception.Configuration"/>
```

Определение контейнера

- Перехватчик интерфейса (interface interceptor) — это перехватчик экземпляра, ограниченный в своих действиях до прокси только одного интерфейса объекта. Такой перехватчик создает класс прокси с помощью генерации динамического кода. Элемент поведения interception в конфигурации указывает внешний код, который должен выполняться вокруг перехватываемого экземпляра объекта.

```
<container>
 <extension type="Interception" />
 <register type="IBankAccount" mapTo="BankAccount">
   <interceptor type="InterfaceInterceptor" />
   <interceptionBehavior type="TraceBehavior" />
 </register>
</container>
```

- Класс TraceBehavior нужно конфигурировать декларативно, чтобы контейнер мог разрешать его и любые его зависимости. Чтобы сообщить контейнеру о классе TraceBehavior и его конструкторе вы используете элемент <register>:

```
<register type="TraceBehavior">
   <constructor>
     <param name="source" dependencyName="interception" />
   </constructor>
</register>
```

```
class TraceBehavior : IInterceptionBehavior, IDisposable    {
  private TraceSource source;
  public TraceBehavior(TraceSource source)      {
    if (source == null)
      throw new ArgumentNullException("source");
    this.source = source;
  }
  public IEnumerable<Type> GetRequiredInterfaces()      {
    return Type.EmptyTypes;
  }
  public IMethodReturn Invoke(IMethodInvocation input,
    GetNextInterceptionBehaviorDelegate getNext)      {
    // BEFORE the target method execution
    this.source.TraceInformation("Invoking {0}",
      input.MethodBase.ToString());
    // Yield to the next module in the pipeline
    var methodReturn = getNext().Invoke(input, getNext);
    // AFTER the target method execution
    if (methodReturn.Exception == null)         {
      this.source.TraceInformation("Successfully finished {0}",
        input.MethodBase.ToString());
    } else {
      this.source.TraceInformation(
        "Finished {0} with exception {1}: {2}",
        input.MethodBase.ToString(),
        methodReturn.Exception.GetType().Name,
        methodReturn.Exception.Message);
    }
    this.source.Flush();
    return methodReturn;
  }
  public bool WillExecute {
    get { return true; }
  }
  public void Dispose(){
    this.source.Close();
  }
}
```

- Класс поведения реализует IInterceptionBehavior, который в основном состоит из метода Invoke. Этот метод содержит всю логику, нужную для любого метода, который находится под контролем перехватчика. Если вы хотите сделать что-то до вызова целевого метода, то делаете это в начале метода. Когда вам требуется перейти к целевому объекту (или, точнее, к следующему поведению, зарегистрированному в конвейере), вы вызываете делегат getNext, предоставляемый инфраструктурой.

Гибкость конфигурирования

- Перехват и AOP в целом открывают целый ряд интересных возможностей. Например, перехват позволяет добавлять обязанности в индивидуальные объекты без модификации всего класса, благодаря чему решение получается гораздо более гибким, чем при использовании шаблона Decorator.