

Spring: AOP

Antonio Espín Herranz

Spring AOP

- Introducción.
- Conceptos y terminología.
- Definir puntos de corte.
- Configuración por XML y anotaciones.
- Añadir nuevas funcionalidades (introducción).

Introducción

- La *Programación Orientada a Aspectos (AOP)* es un paradigma de programación que busca poder tratar de extraer como módulos aquel **código que resuelve problemas transversales a los componentes de una aplicación**.
- Para ver dónde podemos aplicar AOP, debemos identificar esas funcionalidades genéricas que se utilizan en muchos puntos diferentes.
- Algunas de las más comunes son: logs, transacciones, seguridad, cachés, manejo de errores

Ejemplo

- Un ejemplo sencillo, imaginemos que ***para todos los métodos del servicio PatientServiceImpl se requiere que el usuario tenga un determinado rol:***

```
public class PatientServiceImpl implements PatientService{  
    public Patient findById(Integer patientId) {  
        if(!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        return patientDao.findById(patientId);  
    }  
    public List findByRoom(Integer roomId) {  
        if(!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        return patientDao.findByRoom(roomId);  
    }  
}
```

Comentarios al ejemplo

- Este código tiene **dos problemas**:
 - **Mezcla y acopla conceptos que son diferentes: los métodos findById y findByRoom deben preocuparse de encontrar los datos, no de gestionar la seguridad. Esto es lo que conocemos como “code tangling” (enredo de código).**
 - La solución a un mismo problema aparece **repetida varias veces en diferentes partes de la aplicación: el código que comprueba el rol de usuario está repetido en diferentes puntos. Esto es lo que conocemos como “code scattering” (dispersión de código). En una aplicación pequeña puede que esto no suponga un gran problema pero a medida que nuestra aplicación crece, es muy costoso mantener código disperso cuya funcionalidad, además, está entremezclada con otras.**

Solución

- Utilizar AOP:
 - AOP es una solución muy elegante para eliminar estos problemas que sigue en estos tres pasos:
 - Implementa la lógica de negocio de tu aplicación.
 - Implementa aspectos que resuelvan los problemas transversales a tu aplicación.
 - Enlaza estos aspectos en los puntos en los que sean necesarios.

Introducción

- Dentro de las aplicaciones hay funcionalidades que son comunes, como:
 - Transacciones, seguridad, inicio de sesiones, etc.
 - **Funcionalidades Trasversales.**
- ¿deberían de añadirse estas a los objetos que implementan la lógica de negocio?
- Los objetos de la aplicación deberían dedicarse a lo que realmente realiza la aplicación y esas funcionalidades comunes deberían estar fuera de estos.

Introducción

- La **inyección de dependencias** nos permite **desacoplar** los **objetos** de una aplicación entre sí.
- En el caso de la **AOP** nos permite **desacoplar** los **objetos** de la aplicación de las **funcionalidades transversales**.
- Para estas tareas se utiliza la AOP.

Introducción

- Una funcionalidad transversal puede ser cualquier funcionalidad que afecte a varios puntos de la aplicación.
- **La AOP permite definir una funcionalidad común en una ubicación** y definir de forma declarativa (XML / anotaciones) como y donde se va aplicar sin tener que modificar la clase donde se aplica la nueva característica.

Introducción

- **AOP** permite escribir los objetos de la aplicación de una forma sencilla y por otro lado centralizar todas las funcionalidades transversales indicado mediante aspectos y puntos de corte donde habrá que aplicar dichas funcionalidades y a quién hay que aplicárselas.
- Los **Aspectos** se representan con **clases Java**.

Introducción

- **La AOP en Spring se aplica a nivel de método.**
Cuando se define un Aspecto se indica a qué método de un objeto se aplica.
- **Spring construye de forma dinámica un proxy que intercepta las peticiones a este método y lanza lo que se llama un consejo** (es decir, ejecuta un método de la clase que representa el Aspecto).
- Mediante un lenguaje se especificará en que momento se desencadena el consejo.

Conceptos

- Existen ciertos conceptos de AOP que utilizaremos mucho y que conviene conocer:
 - **Join Point:** un punto en la ejecución de un programa (llamada a un método, asignación...)
 - **Pointcut:** expresión que selecciona uno o más Join Points.
 - **Advice:** código que queremos que se ejecute cuando un Join Point es seleccionado por un Pointcut.
 - **Aspect:** módulo que encapsula pointcuts y advice.

Terminología

- **Advise / Consejo:** representa el propósito de un aspecto, es decir, la tarea que tiene que realizar (un método). También llamado **interceptor**.
- Los interceptores se pueden ejecutar en diversas situaciones:
 - **Antes** de que se ejecute el método.
 - **Después** de que finalice el método, sea cual sea el resultado.
 - **Después de la devolución** (cuando el método se completa con éxito).
 - **Después de que se produzca un error.**
 - **Alrededor:** El consejo encapsula al método.

Terminología

- **Puntos de cruce / Joint Point**
- Define en que punto de la ejecución de la aplicación se va a **conectar** con un Aspecto.
 - Puede ser un **método que se ejecuta**.
 - Una **propiedad** que se modifica.
 - Un **error** que se produce.
- En el caso de **Spring sólo se permite ligarlo a la ejecución de un método**.
- **AspectJ, Jboss**, permiten más funcionalidades.

Terminología

- **Puntos de Corte / Pointcut**
 - Los puntos de corte se utilizan para **indicar donde** (en nuestro caso en que métodos) se van a lanzar los consejos.
 - Los **puntos de corte** se especifican mediante patrones o **expresiones regulares** que definen las coincidencias que se tienen que dar para ejecutar el consejo.

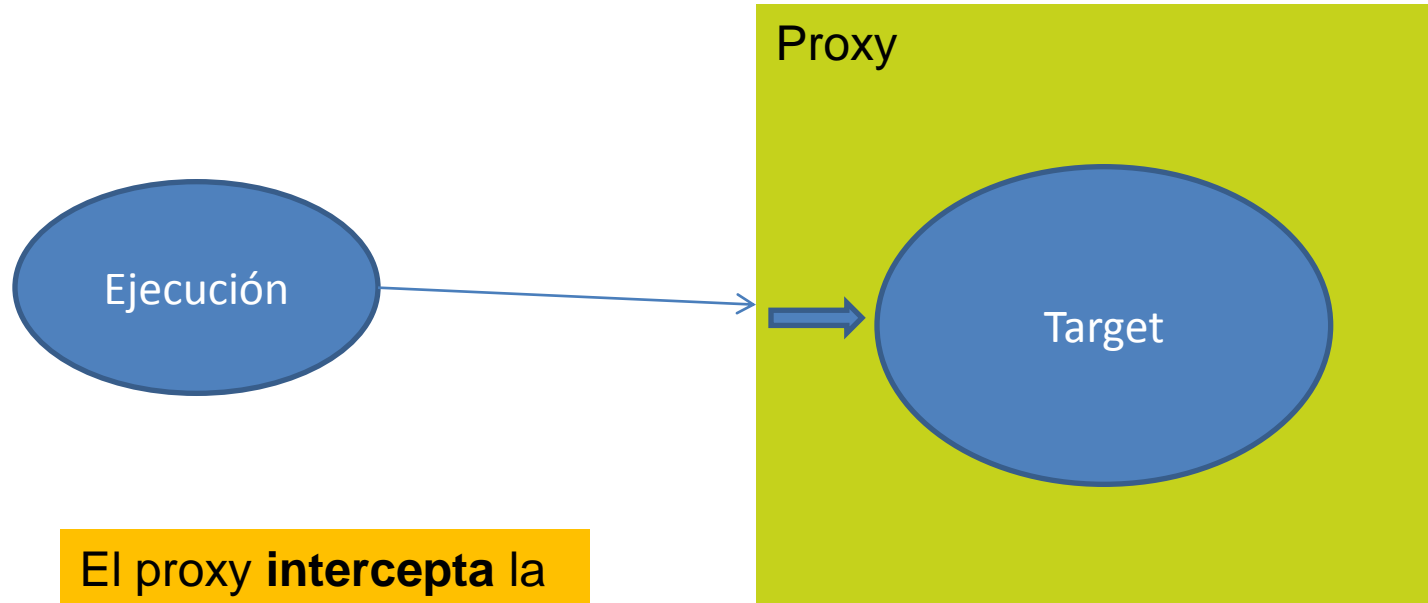
Terminología

- **Aspectos:**
 - Un aspecto combina los consejos y los puntos de corte. De forma conjunta definen todo lo que debe hacerse sobre un aspecto. Donde, que hacer y cuando.
- **Introducción:**
 - Permite **añadir nuevas funcionalidades** a una clase **en tiempo de ejecución**, sin modificar las clases ya definidas.
- **Entrelazado:**
 - El entrelazo es el proceso de aplicar aspectos a un objeto destino para crear un nuevo objeto proxy. En el caso de Spring se realiza de forma dinámica mediante un objeto proxy que intercepta las peticiones al método donde se ha asociado el aspecto.

Aspectos

- Los **Aspectos** se escriben en **clases Java**.
- Los puntos de corte / cruce se especifican mediante **programación declarativa**, puede ser mediante **XML** en el fichero de configuración de Spring y también mediante anotaciones de **AspectJ**.

Funcionamiento



El proxy **intercepta** la petición de Ejecución del método aconsejado
Y reenvía la ejecución al bean Target.

Seleccionar puntos de corte

- Los **puntos de corte** se utilizan para indicar **dónde debe aplicarse el consejo de un aspecto.**
- Son los elementos más importantes de un aspecto.
- Se definen en un **lenguaje de expresiones** llamado **AspectJ.**

Puntos de Corte

- Para definir pointcuts en Spring AOP utilizaremos la notación de AspectJ y seguirán el siguiente patrón:
 - `execution(<patrón>)` Para que un método sea “interceptado” por un aspecto, deberá cumplir el patrón que indiquemos.
 - Además, podemos componer pointcuts utilizando `&&` (and), `||` (or) y `!` (not).
 - Los patrones los definiremos siguiendo esta estructura:
 - **[Modificadores] TipoRetorno [Clase] NombreMétodo ([Parámetros]) [throws TipoExcepción]**
 - **Los métodos que vayan a ser seleccionados por un pointcut deben ser visibles (públicos)**

Ejemplos

- ***execution(void send*(String))***: cualquier método visible que comience por send, tome un String como único parámetro y cuyo tipo de retorno sea void.
- ***execution(* send(*))***: cualquier método visible llamado send que tome como parámetro un parámetro de cualquier tipo.
- ***execution(* send(int, ..))***: cualquier método visible llamado send que tome al menos un parámetro de tipo int. En este caso “..” indica 0 o más.
- ***execution(void org.ejemplo.MessageServiceImpl.*(..))***: cualquier método visible de la clase org.ejemplo.MessageServiceImpl que tenga como tipo de retorno void.

Ejemplos

- ***execution(void org.ejemplo.MessageService+.send(*))***: cualquier método visible con nombre send de las clases del tipo org.ejemplo.MesssageService, incluyendo hijos e implementaciones, que reciban un único parámetro de cualquier tipo y tengan void como tipo de retorno.
- ***execution(@javax.annotation.security.PermitAll void send*(..))***: cualquier método visible que comience por send y que esté anotado con la anotación @PermitAll.
- ***execution(* send(int, ..))***: cualquier método visible llamado send que tome al menos un parámetro de tipo int. En este caso “..” indica 0 o más.
- ***execution(void org.ejemplo.MessageServiceImpl.*(..))***: cualquier método visible de la clase org.ejemplo.MessageServiceImpl que tenga como tipo de retorno void.

Ejemplos

- ***execution(void org.ejemplo.MessageService+.send(*))***: ***cualquier*** método visible con nombre send de las clases del tipo org.ejemplo.MesssageService, incluyendo hijos e implementaciones, que reciban un único parámetro de cualquier tipo y tengan void como tipo de retorno.
- ***execution(@javax.annotation.security.PermitAll void send*(..))***: cualquier método visible que comience por send y que esté anotado con la anotación @PermitAll.

Designadores de puntos de corte

<code>args()</code>	Ejecuta los métodos cuyos argumentos son instancias de tipos datos.
<code>@args()</code>	Ejecuta los métodos cuyos argumentos son anotados con los tipos de anotación indicados.
<code>execution()</code>	Hace coincidir los puntos de cruce que son ejecuciones del método.
<code>this()</code>	Limita las coincidencias a aquellas que el bean proxy es de un determinado tipo.
<code>target()</code>	Limita las coincidencias a aquellas que el bean objetivo es de un determinado tipo.
<code>@target()</code>	Cuando la clase de un objeto en ejecución cuenta con una anotación de tipo dado.
<code>within()</code>	Limita la coincidencia dentro de ciertos tipos.
<code>@within()</code>	Se limita a ciertos tipos que cuenten con la anotación dada.
<code>@annotation</code>	Si el método tiene la anotación dada se seleccionará.

Ejemplos

- execution (* com.ejemplos.MiClase.metodo(..))
- Se activa con la **ejecución del método**.
- * indica que devuelve cualquier tipo.
- Se indica el **paquete, la clase y método**.
- .. Indica que el método acepta cualquier argumento.

Ejemplos

- execution (* com.ejemplos.MiClase.metodo(..))
&& within(com.ejemplos.*)
- && → and
- **within**: cuando el método se ejecuta desde cualquier clase en el paquete com.ejemplos.

Ejemplo con el designador bean()

- **bean()** permite especificar un bean por su id dentro de la expresión de punto de cruce.
- **execution (*com.ejemplos.MiClase.miMetodo(..)) and bean(miObjeto)**
- Indicamos que el consejo lo queremos aplicar a la ejecución de un método: `miMetodo` aunque se limita al bean: `miObjeto`.

Ejemplo con el designador bean()

- Se puede utilizar la negación para indicar a todos los bean menos en que se indica.
- **execution (*
com.ejemplos.MiClase.miMetodo(..)) and
!bean(miObjeto)**

Declarar aspectos en XML

- Se incluye un nuevo espacio de nombres dentro del fichero de configuración de Spring.
- El espacio de nombre de configuración **aop**.
- Elementos XML disponibles:
 - `<aop:config>` Elemento de mayor jerarquía.
 - `<aop:aspect>` Declarar un aspecto.
 - `<aop:pointcut>` Definir un punto de corte.
 - `<aop:declare-parents>` Introducir interfaces adicionales.
 - `<aop:after>`, `<aop:after-returning>`, `<aop:after-around>`, `<aop:before>`, `<aop:throwing>` en que momento se lanza...

Declarar aspectos en XML

- Para las declaraciones en XML necesitamos agregar los siguientes jar al proyecto (no vienen dentro de Spring).



aopalliance-1.0.jar



aspectjrt-1.5.4.jar



aspectjweaver-1.6.10.jar


Ejemplo

- Suponemos una actuación musical en la que hay personajes que tocan instrumentos, todos ellos englobados en el interface “Instrument” con el método play.
- Cada personaje (clase Instrumentalist) sabe tocar un instrumento, se indica mediante la propiedad instrument. Y aparte cada personaje implementa el interface Performer con el método perform que lo que hace es ponerse a tocar el instrumento.
- Si en este escenario se quisiera simular el comportamiento del público se podría representar mediante la clase Audience, de tal forma que estuviera ligado al acto de tocar el instrumento. Esto se puede representar con un aspecto.

Ejemplo

```
package com.springinaction.springidol;

public class Guitar implements Instrument {
    public void play() {
        System.out.println("Strum strum strum");
    }
}
```




```
package com.springinaction.springidol;

public class Instrumentalist implements Performer {
    public void perform() throws PerformanceException {
        instrument.play();
    }

    private Instrument instrument;

    public void setInstrument(Instrument instrument) {
        this.instrument = instrument;
    }

    public Instrument getInstrument() {
        return instrument;
    }
}
```



```
public class Audience {
    public void takeSeats() {
        System.out.println("The audience is taking their seats.");
    }
    public void turnOffCellPhones() {
        System.out.println("The audience is turning off their cellphones");
    }
    public void applaud() {
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }
    public void demandRefund() {
        System.out.println("Boo! We want our money back!");
    }
}
```

```
package com.springinaction.springidol;

public interface Performer {
    void perform() throws PerformanceException;
}
```

```
Performer eddie = (Performer) context.getBean("eddie");
eddie.perform();
```


Ejemplo Declaración

```
- <bean class="com.springinaction.springidol.Instrumentalist" id="eddie">
  - <property name="instrument">
    <bean class="com.springinaction.springidol.Guitar"/>
  </property>
</bean>
<!--<start id="audience_bean" />-->
<bean class="com.springinaction.springidol.Audience" id="audience"/>
<!--<end id="audience_bean" />-->
<!--<start id="audience_aspect" />-->
- <aop:config>
  - <aop:aspect ref="audience">
    <!--<co id="co_refAudienceBean"/>-->
    <aop:before method="takeSeats" pointcut="execution(* com.springinaction.springidol.Performer.perform(..))"/>
    <!--<co id="co_beforePointcut"/>-->
    <aop:before method="turnOffCellPhones" pointcut="execution(* com.springinaction.springidol.Performer.perform(..))"/>
    <!--<co id="co_beforePointcut2"/>-->
    <aop:after-returning method="applaud" pointcut="execution(* com.springinaction.springidol.Performer.perform(..))"/>
    <!--<co id="co_afterPointcut"/>-->
    <aop:after-throwing method="demandRefund" pointcut="execution(* com.springinaction.springidol.Performer.perform(..))"/>
    <!--<co id="co_afterThrowingPointcut"/>-->
  </aop:aspect>
</aop:config>
<!--<end id="audience_aspect" />-->
```

El aspecto está representado por la clase `Audience` y el bean `audience`. Los puntos de corte se asocian a la ejecución del método **perform**. Los métodos **takeSeats** y **turnOffCellPhone** se lanzan antes de la ejecución del método `perform`. **applaud** después de la correcta ejecución y **demandRefund** en caso de que se produzca una excepción.

Agrupar pointcut repetidos

```
<aop:config>
- <aop:aspect ref="audience">
  <aop:pointcut id="performance" expression="execution(* com.springinaction.springidol.Performer.perform(..))"/>
  <!--<co id="co_defPointcut"/>-->
  <aop:before method="takeSeats" pointcut-ref="performance"/>
  <!--<co id="co_refPointcut"/>-->
  <aop:before method="turnOffCellPhones" pointcut-ref="performance"/>
  <!--<co id="co_refPointcut"/>-->
  <aop:after-returning method="applaud" pointcut-ref="performance"/>
  <!--<co id="co_refPointcut"/>-->
  <aop:after-throwing method="demandRefund" pointcut-ref="performance"/>
  <!--<co id="co_refPointcut"/>-->
</aop:aspect>
</aop:config>
```

- En este caso se declara primero el punto de corte y después los consejos.
- Para ello se utiliza el atributo **pointcut-ref**. Para referenciarlo.
- Por supuesto se pueden definir varios pointcut y varios aspectos dentro de la declaración de aop:config.

Declaración <aop:around>

- También se puede definir consejos de tipo around, en este caso se puede determinar la ejecución o no del método que ha sido ejecutado.
- Disponemos de más control, además se podría ejecutar varias veces, por ejemplo, operaciones de reintento.
- Por ejemplo, nos podría interesar controlar cuanto tiempo tarda en ejecutarse el método.
- El método del consejo tiene que recibir un argumento de tipo: ProceedingJoinPoint.

Ejemplo

- Este método englobaría a todos los métodos de la clase Audience en uno sólo.

```
public void watchPerformance(ProceedingJoinPoint joinpoint) {
    try {
        // Tareas que queremos hacer antes del método:
        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(); // OJO, en este caso tenemos que lanzar la ejecución del método.
                          // en este caso es el método perform().

        long end = System.currentTimeMillis();

        // Tareas que queremos hacer después de la ejecución del método.
        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!"); // Si hay error
    }
}
```

Ejemplo: Declaración del Aspecto

```
- <aop:config>
  - <aop:aspect ref="audience">
    <aop:pointcut id="performance2" expression="execution(* com.springinaction.springidol.Performer.perform(..))"/>
    <aop:around method="watchPerformance()" pointcut-ref="performance2"/>
    <!--<co id="co_around"/>-->
  </aop:aspect>
</aop:config>
```

- Se define otro punto de corte y cuando se lance la ejecución del método perform se ejecuta el consejo around.

Añadir parámetros al consejo

- Es posible pasar parámetros al método aconsejado.
- Incluso tenemos la opción de inspeccionar los argumentos que estamos pasando al método.
- Cuando los **parámetros** que queremos pasar **no sean tipos primitivos** tenemos que cualificar la clase con el nombre del paquete.

Ejemplo

- Suponemos el siguiente escenario: Hay una persona que se encarga de leer el pensamiento a los demás, y por otro lado tenemos un concursante que va a pensar en algo.
- Disponemos de las clases:
 - Magician que implementa la interface MindReader.
 - Volunteer que implementa la interface Thinker.

Ejemplo

```
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;
    }
}
```

```
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;
    }
}
```


Ejemplo Declaración

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

- Primero se declaran los beans, el mago y el pensador.
- Después se define el aspecto al mago y el punto de corte.
- Cuando se ejecuta el método thinkOfSomething del pensador se interceptan los pensamientos por el mago.

```
public void magicianShouldReadVolunteersMind() {
    volunteer.thinkOfSomething("Queen of Hearts");

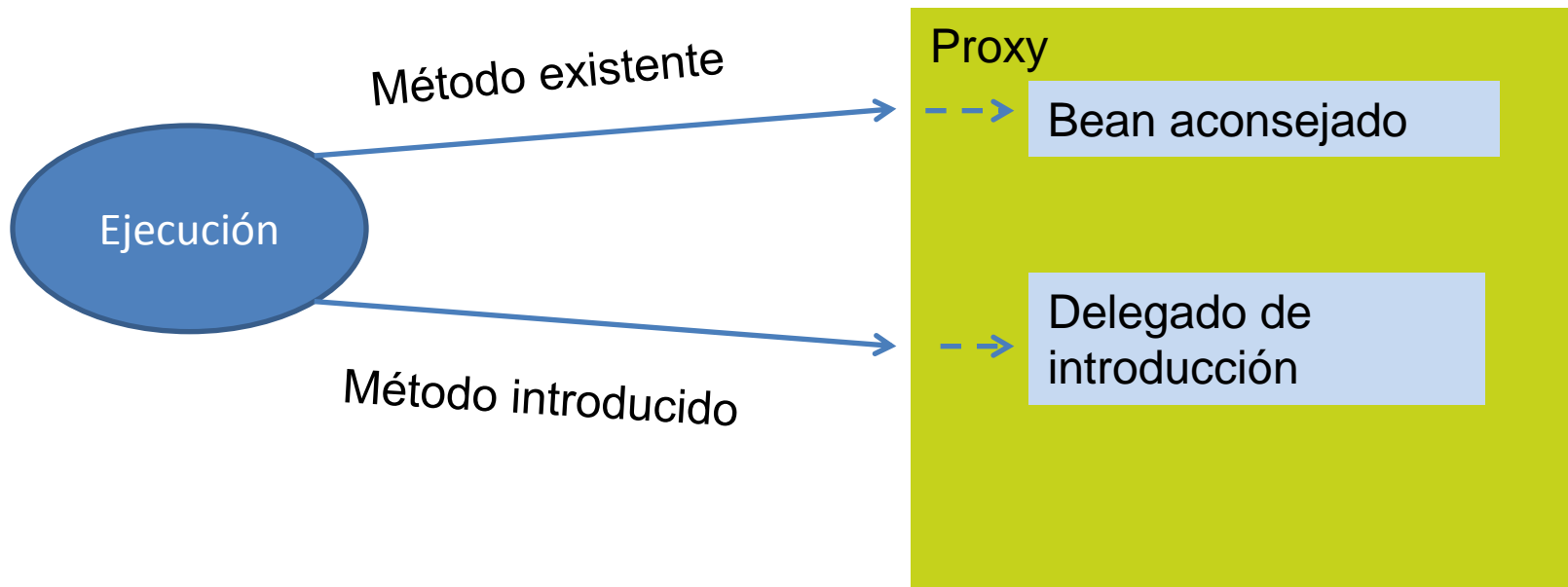
    assertEquals("Queen of Hearts", magician.getThoughts());
}
```

Incluir nuevas funcionalidades mediante aspectos

- A diferencia de otros lenguajes como Ruby o Groovy, **Java no permite añadir nuevas funcionalidades a la clase una vez que esta fue compilada.**
- Esto se puede simular mediante la programación orientada a aspectos.
- **Los aspectos son sólo proxy que implementan la misma interface que los bean que contienen.**
- Los proxy podrían implementar nuevas interfaces y cuando se ejecuta un método de la interface introducida, el proxy delega la ejecución a otro objeto que proporciona la implementación de la nueva interface.

<aop:declare-parents>

- Esta es la declaración mediante la que se pueden añadir nuevas funcionalidades a los bean.
- En vez de cambiar las implementaciones de las clases, se pueden añadir interfaces (nuevas funcionalidades).



<aop:declare-parents>

- Declara que el bean al que se aconseja va a contar con nuevos elementos principales en su jerarquía de objetos.

```
<aop:aspect>
```

```
  <aop:declare-parents
```

```
    types-matching="com.spring.ejemplo.Performer+"
```

```
    implement-interface="com.spring.ejemplo.OtroInterface"
```

```
    default-impl="com.spring.ejemplo.OtroInterfaceImpl"
```

```
  />
```

```
</aop:aspect>
```

<aop:declare-parents>

- **type-matching:**
 - Se indica el interface que se quiere modificar.
 - Al que se quieren añadir nuevas funcionalidades.
- **implement-interface:**
 - El nuevo interface que tienen que implementar.
- **default-impl:**
 - La clase que implementa el nuevo interface.
 - Este último se puede sustituir por **delegate-ref**.
 - Para este habría que indicar el id de un bean.

Ejemplo

- Se podría utilizar de la siguiente forma:

En este caso a los bean que cumplen el interface Performer se le añade un nuevo interface (Contestant) que dispone del método receiveAward().

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

Anotaciones

- A parte de declarar los aspectos de forma declarativa en el fichero de configuración de Spring también **se pueden convertir clases en Aspectos simplemente añadiendo una serie de anotaciones.**
- Para convertir una clase en un Aspecto hay que anotarla con **@Aspect**.
- Dentro de la clase se van definiendo los puntos de corte de una forma parecida al XML.

Ejemplo

```
package com.springinaction.springidol;
```

```
import org.aspectj.lang.annotation.AfterReturning;  
import org.aspectj.lang.annotation.AfterThrowing;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;
```

```
@Aspect
```

```
public class Audience {
```

```
    @Pointcut(
```

```
        "execution(*
```

```
            com.springinaction.springidol.Performer.perform(..))"
```

```
    public void performance() {
```

```
    }
```

```
    @Before("performance()")
```

```
    public void takeSeats() {
```

```
        System.out.println("The audience is taking their seats.");
```

```
    }
```

```
    @Before("performance()")
```

```
    public void turnOffCellPhones() {
```

```
        System.out.println("The audience is turning off their cellphones");
```

```
    }
```

```
    @AfterReturning("performance()")
```

```
    public void applaud() {
```

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

```
    }
```

```
    @AfterThrowing("performance()")
```

```
    public void demandRefund() {
```

```
        System.out.println("Boo! We want our money back!");
```

```
    }
```

```
}
```

No es necesario incluir información de estos elementos en el XML.

Solo es necesario declarar un bean **autoproxy** para que Spring convierta los bean anotados en consejos de proxy.

<aop:aspectj-autoproxy>

Este es el único elemento que tenemos que declarar dentro del fichero de XML de Spring.

Aspectos XML vs Anotaciones

- El elemento `<aop:aspect>` y las anotaciones `@AspectJ` son las dos formas que tenemos de convertir un **POJO** (Plain Old Java Object) en un aspecto.
- **`<aop:aspect>` tiene la ventaja que no necesita el código fuente de la clase** para contar con la funcionalidad del aspecto, en cambio con `@AspectJ` es necesario anotar las clases y los métodos con lo cual necesita el código fuente.

Argumentos al Aspecto anotado

```
package com.springinaction.springidol;
```

```
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Before;  
import org.aspectj.lang.annotation.Pointcut;
```

@Aspect

```
public class Magician implements MindReader {  
    private String thoughts;
```

```
@Pointcut("execution(* com.springinaction.springidol." + "Thinker.thinkOfSomething(String)) && args(thoughts)")  
public void thinking(String thoughts) {  
}
```

@Before("thinking(thoughts)")

```
public void interceptThoughts(String thoughts) {  
    System.out.println("Intercepting volunteer's thoughts : "  
        + thoughts);  
    this.thoughts = thoughts;  
}  
  
public String getThoughts() {  
    return thoughts;  
}  
}
```

Anotación de introducciones

```
package com.springinaction.springidol;
```

```
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.DeclareParents;
```

@Aspect

```
public class ContestantIntroducer {
```

@DeclareParents(

```
    value = "com.springinaction.springidol.Performer+",  
    defaultImpl = GraciousContestant.class)
```

```
public static Contestant contestant;
```

```
}
```

- Contestant es el interface que agrega las nuevas operaciones que serán implementadas por la clase GraciousContestant.
- Performer en el interface al que se le añaden las operaciones.
- Y aparte en el contexto se necesita declarar el bean:
- <bean class="com.ejemplo.ContestantIntroducer"

El atributo **value** es equivalente a types-matching de <aop:declare-parents>

Identifica los tipos de bean que deben ser introducidos con la interface.

default-Impl: equivale a default-impl de <aop:declare-parents> que identifica la clase que va a proporcionar la implementación.

La propiedad estática anotada con **@DeclareParents** especifica la interface que se va a introducir.