Spring

Sesión 6: AOP (Programación Orientada a Aspectos)

© 2011 Depto. Ciencia de la Computación e IA

Indice

- Introducción a AOP. Terminología básica
- Cómo funciona la AOP en Spring.
- Sintaxis básica (AspectJ)
- Definir clases AOP

© 2011 Depto. Ciencia de la Computación e IA

Aspect Oriented Programming

- Intenta formalizar los aspectos transversales a todo el sistema
- La POO no es la respuesta a todos los problemas

```
public class MiObjetoDeNegocio {
   public void metodoDeNegocio1() throws SinPermisoException {    chequeaPermisos();
       //resto del código ... }
   public void metodoDeNegocio2() throws SinPermisoException {    chequeaPermisos();
       //resto del código ... }
  protected void chequeaPermisos() throws SinPermisoException {
       //chequear permisos de ejecucion ... }
 }
```

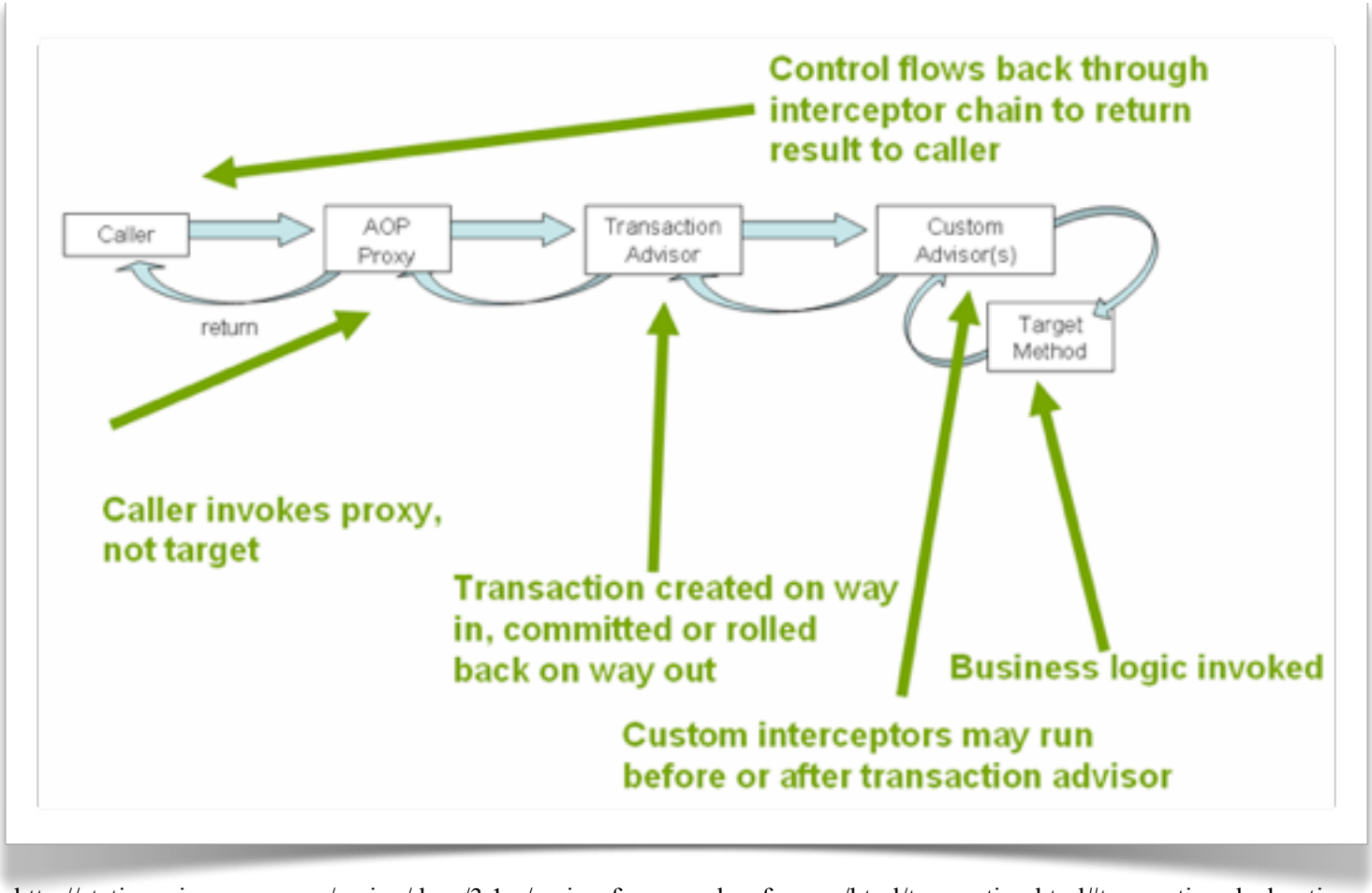

© 2011 Depto. Ciencia de la Computación e IA

Un poco de argot (*básico, hay más…)*

- **Aspecto:** transversal al sistema (aparece en muchas partes del código, disperso) y que queremos modularizar (Ej. *chequeo de permisos*)
- **Advice**: código a ejecutar para implementar un aspecto (Ej. chequeaPermisos())
- **Pointcut**: puntos del código donde se debe ejecutar algún *advice* (Ej: *cualquier método cuyo nombre comience por MetodoDeNegocio*)
- **Advisor**: *advice+pointcut* (*antes de llamar a cualquier método cuyo nombre comience por MetodoDeNegocio hay que ejecutar* chequeaPermisos())

© 2011 Depto. Ciencia de la Computación e IA

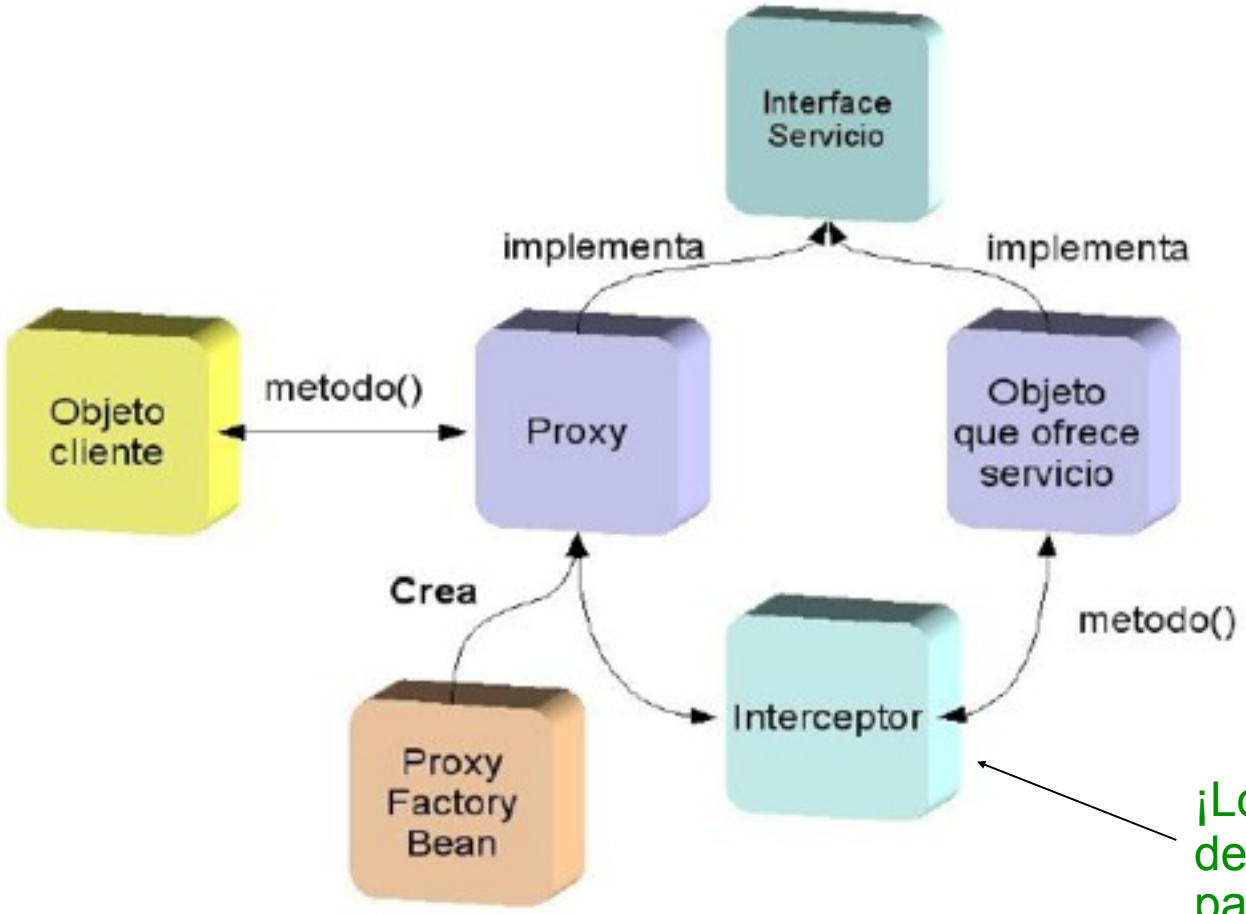
La transaccionalidad declarativa usa AOP



http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/transaction.html#transaction-declarative

© 2011 Depto. Ciencia de la Computación e IA

AOP genérica en Spring



¡Lo único que debemos implementar para tener AOP!

© 2011 Depto. Ciencia de la Computación e IA

Utilidad práctica de AOP en Spring

- Spring la usa “por debajo” para prácticamente todo
  - Transaccionalidad declarativa, como acabamos de ver
  - Seguridad con **Spring Security**
- A medida que se van incorporando cosas nuevas, cada vez se reduce más la necesidad de AOP “a medida”
  - Por ejemplo la versión 3.1 incorpora una *cache* basada en AOP

```
@Cacheable("books")
public Book findBook(ISBN isbn) {
     ...
}
```

  - No obstante, es útil saber cómo funciona por motivos didácticos

© 2011 Depto. Ciencia de la Computación e IA

Spring AOP y AspectJ

- **AspectJ** es un framework para incorporar características AOP al lenguaje Java. Lleva bastante tiempo y es maduro y estable
- Spring usa la sintaxis de **AspectJ**
- **CUIDADO**: **No es lo mismo** usar la sintaxis de AspectJ que usar AspectJ en sí. Nosotros no usaremos AspectJ, solo su sintaxis
  - Hay muchas cosas que pueden hacerse en AspectJ que no pueden hacerse en Spring AOP

© 2011 Depto. Ciencia de la Computación e IA

Configuración de AOP en Spring

- Librerías necesarias: aspectjrt.jar, aspectjweaver.jar
  - Si deseamos hacer AOP de clases “puras”, necesitaremos CGLIB
- Activar el soporte AOP en el fich. XML de beans:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd“>
  <aop:aspectj-autoproxy/>
<beans>
```


© 2011 Depto. Ciencia de la Computación e IA

Sintaxis de AspectJ *(con anotaciones)*

- Un aspecto es una clase Java con la anotación **`@Aspect`**
  - En Spring, además debe ser un **bean**
- Un *pointcut* es una "especie" de expresión regular (cuidado, "explicación para niños") pero con sintaxis AspectJ
- Un *Advice* es un método **`public void`** con una anotación que indica si hay que ejecutarlo "antes", "después" del *pointcut*,…

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Component
@Aspect public class Ejemplo {
	@Before("execution(public * get*())")
	public void controlaPermisos() {
		// ...
	}
}
```


© 2011 Depto. Ciencia de la Computación e IA

Pointcuts

- En Spring están limitados a **ejecuciones de métodos** (en AspectJ completo no)
- **execution():** ejecución de un método. Especificar
  - Modificador de acceso (public,…) [opcional]
  - Tipo de retorno
  - Nombre de clase (inc.paquetes) [opcional]
  - Nombre de método
  - Argumentos
  - Comodines:
    - * (1 token) .. (varios token)
    - En argumentos: . (1 argumento) .. (varios)

© 2011 Depto. Ciencia de la Computación e IA

Ejemplos de execution()

- execution(public * get*())")
  - Un getter (*público, cualquier tipo de retorno, sin parámetros, comienza por get*)
- execution(public * es.ua.jtech.aop.*.*(..))
  - Cualquier método público de cualquier clase dentro del paquete es.ua.jtech.aop con cualesquiera parámetros
- execution(public * es.ua.jtech..*.*(..))
  - Idem al anterior pero también con subpaquetes (fiajos en el ..)
- execution (void es.ua.jtech.*.*(String,..))
  - Cualquier método que devuelva void de cualquier clase dentro del paquete es.ua.jtech y cuyo primer parámetro sea un String

© 2011 Depto. Ciencia de la Computación e IA

Algo más avanzado

- Combinar con operadores lógicos al estilo C/Java
  - execution (public * get*()) || execution (public void set*(*))
- Darles nombre simbólico con **@Pointcut**
- Esto se hace dentro de la clase con el @Aspect

```
@Pointcut("execution(public * get*())")
public void unGetterCualquiera() {}

@Pointcut(“execution(* es.ua.jtech.ejemplo.negocio.*.*(..))")
public void enNegocio() {}

@Pointcut("unGetterCualquiera() && enNegocio()")
public void getterDeNegocio() {}
```


© 2011 Depto. Ciencia de la Computación e IA

Usar *pointcuts* con nombre

```
@Component
@Aspect
public class MiAspect {

//les damos nombre a los pointcuts
@Pointcut("execution(public * get*())")
public void unGetterCualquiera() {}

@Pointcut(“execution(* es.ua.jtech.ejemplo.negocio.*.*(..))")
public void enNegocio() {}

@Pointcut("unGetterCualquiera() && enNegocio()")
public void getterDeNegocio() {}

//usamos alguno de los nombres anteriores para definir un advice
@Before(“getterDeNegocio()”)
public void antesDeGetterDeNegocio() {
        System.out.println(“Estoy antes de ...”);
}
}
```

Otros predicados

- **within():** dentro de un paquete. Permite acortar la sintaxis de execution()
  - **within(es.ua.jtech..*)** (ejemplo 1 anterior)
- **args()** En la práctica se suele usar junto con **execution()** para poder acceder a los parámetros del método del pointcut en el código del *advice*

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect public class Ejemplo {
	@Before("execution(public void set*(*)) && args(nuevoValor)")
	public void log(int nuevoValor) {
		// ...
	}
}
```


© 2011 Depto. Ciencia de la Computación e IA

Advices+Pointcuts

- Se puede especificar si queremos ejecutar algo
  - Antes del pointcut (@Before)
  - Después del pointcut
    - Si se genera una excepción (@AfterThrowing)
    - Si se retorna normalmente (@AfterReturning)
    - En cualquier caso (tipo *finally*) (@After)
  - Antes y después (@Around)
- Veremos algunos ejemplos. Fijáos sobre todo en los parámetros de la anotación. Lo visto antes es aplicable a todas casi por igual

© 2011 Depto. Ciencia de la Computación e IA

@Before

- Normalmente para abortar la ejecución se lanza una excepción

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

@Aspect public class EjemploBefore {
   @Before(pointcut="execution(public * elimina*())")
   public void chequeaPermisos()
       if (!hayPermiso())
            throw new Exception(“¡Te pillamos!!”);
   }
}
```


© 2011 Depto. Ciencia de la Computación e IA

@AfterReturning

- Podemos acceder al valor de retorno
  - Aquí ponemos Object porque no importa el tipo
  - Si ponemos un tipo, solo encajará con los métodos que lo devuelvan.

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

@Aspect public class EjemploAfterReturning {
   @AfterReturning(pointcut="execution(public * get*())",
                           returning="valor") //acceso al valor de retorno
   public void postprocess(Object valor) {
      // ...
   }
}
```


© 2011 Depto. Ciencia de la Computación e IA

@AfterThrowing

- Podemos restringir el tipo de excepción que nos interesa

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

@Aspect public class EjemploAfterThrowing {
   @AfterThrowing(pointcut="execution(public * get*())",
                  throwing="daoe") //acceso a la excepción lanzada
   public void logException(DAOException daoe) {
      // ...
   }
}
```


© 2011 Depto. Ciencia de la Computación e IA

@Around

- Es obligatorio definir un parámetro de tipo **ProceedingJoinPoint** que representa el punto de corte. Es nuestra responsabilidad invocarlo o no

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect public class EjemploAround {
	@Around("execution(public * get*())")
	public Object profile(ProceedingJoinPoint pjp) throws Throwable {
		System.out.println(“Antes del joinpoint”);
		Object valorRetorno = pjp.proceed();
		System.out.println(“Después del joinpoint”);
		return valorRetorno;
	}
}
```


© 2011 Depto. Ciencia de la Computación e IA

¿Preguntas...?

© 2011 Depto. Ciencia de la Computación e IA