



Spring

Sesión 3: AOP (Programación Orientada a Aspectos)



Indice

- Introducción a AOP. Terminología básica
- Cómo funciona la AOP en Spring.
 - AOP 2.0 vs. 1.2
- Sintaxis básica (AspectJ)
- Definir clases AOP



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 ... }  
}
```

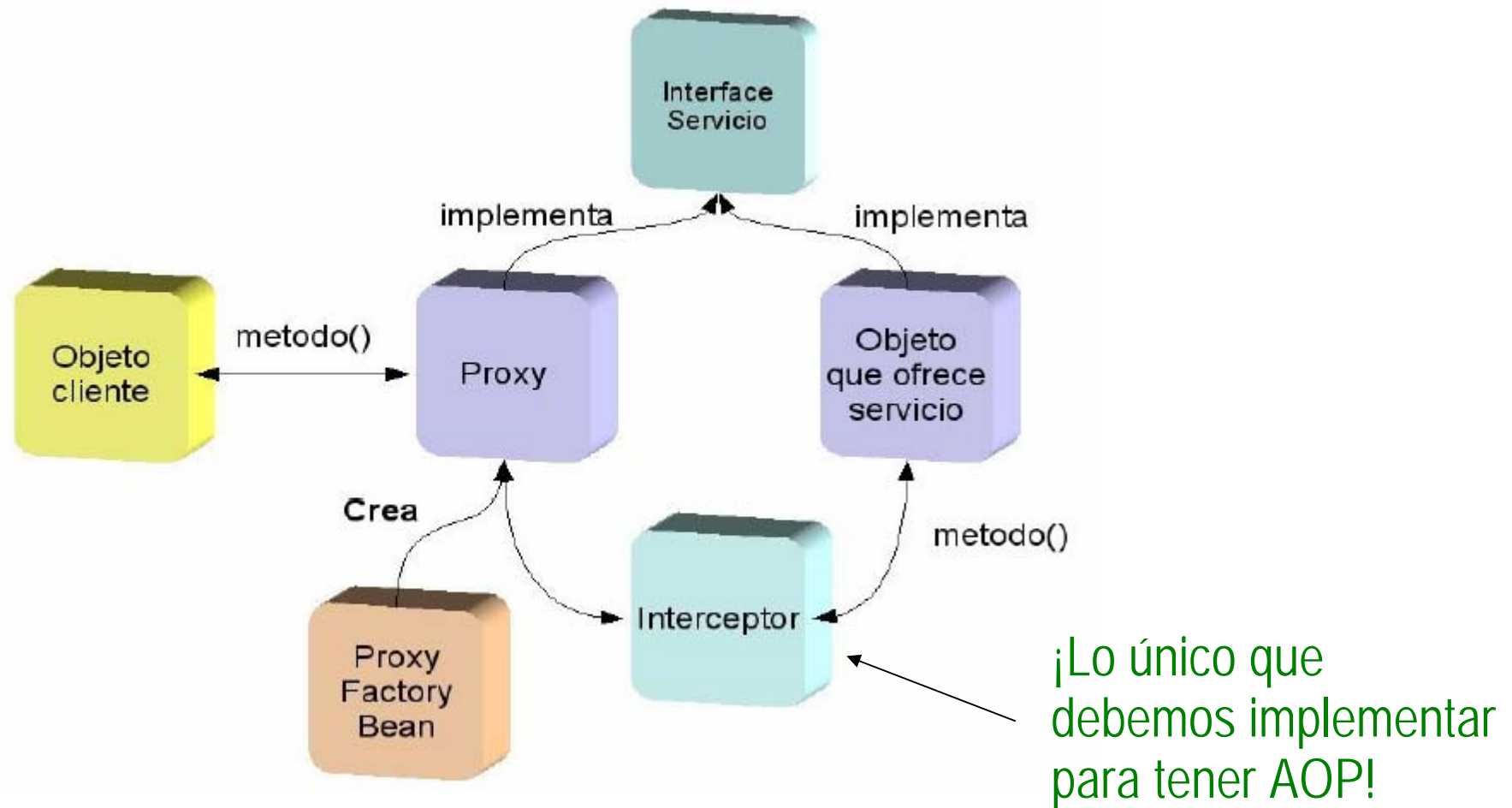


Un poco de argot (*básico, hay más...*)

- **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()`)



AOP en Spring





AOP en Spring 2.0 vs. 1.2

- **La forma de hacer AOP se ha simplificado y estandarizado:** Spring 2.0 usa la sintaxis de **AspectJ**, de “larga” tradición en el mundo “*AOPJavero*”
 - “Estándar”
 - Mucho más potente que la de Spring 1.2
- **No es lo mismo** usar la sintaxis de AspectJ que usar AspectJ en sí. Nosotros no usaremos AspectJ, solo su sintaxis
- No obstante **se puede seguir usando la sintaxis de 1.2**. Aviso: no se parece en casi nada a la que veremos, ¡cuidado!



Configuración de AOP en Spring

- Librerías necesarias: aspectjrt.jar, aspectweaver.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-2.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
    <aop:aspectj-autoproxy/>
    ..
</beans>
```



Sintaxis de AOP *(con anotaciones de Java 5)*

- Un aspecto es una clase Java con la anotación `@Aspect`
- 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;
@Aspect public class Ejemplo {
    @Before("execution(public * get*())")
    public void controlaPermisos() {
        // ...
    }
}
```




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)



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 (fijos 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`



Algo más avanzado

- Combinar con operadores lógicos al estilo C/Java
 - execution (public * get*()) || execution (public void set*(*))
- Darles nombre simbólico con @Pointcut

```
@Pointcut("execution(public * get*())")  
public void unGetterCualquiera() {} //CUIDADO, esto NO es un Advice, solo un  
                                   //nombre
```

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

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



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) {
        // ...
    }
}
```

Spring



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



@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="daoException")
    public void logException(DAOException daoException) {
        // ...
    }
}
```



@AfterReturning

- Ponemos Object para que encaje con cualquier cosa

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

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



@Around

- Es obligatorio definir un 1er parámetro 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 {
        long inst1 = System.currentTimeMillis();
        Object valorRetorno = pjp.proceed();
        long inst2 = System.currentTimeMillis();
        System.out.println(inst2-inst1);
        return valorRetorno;
    }
}
```




¿Preguntas...?