1. Programación Orientada a Aspectos (AOP)

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, monitorización…

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);
 }

}
```

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.

AOP es una solución muy elegante para eliminar estos problemas que sigue en estos tres pasos:

1. Implementa la lógica de negocio de tu aplicación.
2. Implementa aspectos que resuelvan los problemas transversales a tu aplicación.
3. Enlaza estos aspectos en los puntos en los que sean necesarios.

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 Poincut.
- **Aspect:** módulo que encapsula pointcuts y advice.

Existen diferentes formas de trabajar con aspectos en Spring AOP (mediante XML, mediante anotaciones AspectJ, utilizando configuración por Java, utilizando configuración por XML, con CGLib, etc…) Nosotros nos centraremos en Spring AOP con anotaciones AspectJ, utilizando proxies que envuelven a las clases y ejecutan los aspectos. Hay que tener en cuenta que, por defecto, Spring AOP sólo permite definir aspectos sobre clases que implementen algún interfaz ya que para ello utiliza proxies dinámicos de JDK, para hacerlo sobre otras clases debemos utilizar CGLib y configurar la propiedad proxy-target-class a true.

1.1. Creación de un aspecto

Lo primero que haremos será permitir el uso de aspectos mediante la etiqueta: *<aop:aspectj-autoproxy/>.*

El siguiente paso será definir nuestro aspecto. Veamos una posible implementación:

```
@Aspect
public class MonitoringAspect {

  @Before("execution(* com.hospital.services.PatientService.findByRoom(*))")
  public void findingPatients() {
     System.out.println("Buscando pacientes en habitación...");
  }

}
```

En este caso, el poincut está expresando "método findByRoom del servicio PatientService" mientras que el advice sería "escribe Buscando pacientes en habitación…", ambos conceptos formarían nuestro aspecto. Más adelante veremos las diferentes formas de definir pointcuts.

Una vez implementado, lo añadiremos al contexto como cualquier otro bean y le diremos a Spring que lo registre como aspecto utilizando la etiqueta anterior:

```
...
<aop:aspectj-autoproxy>
```

```
    <aop:include name="monitoringAspect"/>
  </aop:aspectj-autoproxy>
  <bean id="monitoringAspect" class="com.hospital.aspects.MonitoringAspect"/>
  ...
```

A partir de ahora, cada vez que alguna clase de nuestra aplicación invoque al servicio de pacientes para buscar los pacientes asignados a una habitación obtendremos un mensaje.

El siguiente paso sería obtener información acerca del punto de ejecución de la aplicación (JoinPoint). Para ello, pasaremos un parámetro de este tipo a nuestro Advice, que, a partir de ese momento, tendrá acceso a información relativa al punto de ejecución de la aplicación, por ejemplo:

```
@Before("execution(* com.hospital.services.PatientService.findByRoom(*))")
public void findingPatientsWithJoinPoint(JoinPoint joinPoint) {
  String name = joinPoint.getSignature().toShortString();
  Object roomId = joinPoint.getArgs()[0];
  System.out.println(name + " buscando pacientes en habitación "+roomId);
}
```

1.2. Definición de Pointcuts

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]

Veámoslo con el pointcut anterior: * *(cualquier tipo de retorno)*com.hospital.services.PatientService *(interfaz PatientService del paquete com.hospital.services)* .findByRoom *(método findByRoom)* (*) *(aceptando un parámetro de cualquier tipo).*

Hay que recordar que los métodos que vayan a ser seleccionados por un pointcunt deben ser visibles (públicos). Algunos ejemplos más:

- ***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.
- ***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(* org.ejemplo.*.impl.*.*(..))***: cualquier método visible de cualquier clase de cualquier paquete impl situada dos escalones por debajo de org.ejemplo en la jerarquía de paquetes.
- ***execution(* org.ejemplo..impl.*.*(..))***: cualquier método visible de cualquier clase de cualquier paquete impl situada cualquier nivel por debajo por debajo de org.ejemplo en la jerarquía de paquetes. En este caso ".." indica que puede haber 0 o más directorios en la jerarquía de paquetes.

A la hora de definir pointcuts, también podremos hacer uso de los Named Pointcuts, donde la definición de estos se separa y se nombra para que posteriormente pueda ser reutilizada, combinada y externalizada (una buena práctica si trabajamos con mucho Pointcuts es definirlos todos en una única clase externa):

```
1
2     public class MyPointCuts {
3
4       @Pointcut("execution(* com.hospital..*Dao.*(..))")
5       public void daoMethods() {}
6
7       @Pointcut("execution(* com.hospital..*Service.*(..))")
8       public void daoMethods() {}
9
10    }
11
12    public class MyAspect {
13
14      @Before("org.example.MyPointCuts.daoMethods")
15      public void myAdviceForDaos() {
16        ...
17      }
18
19      @After("org.example.MyPointCuts.daoMethods
20          ||org.example.MyPointCuts.serviceMethods")
21      public void myAdviceForDaosAndServices() {
22        ...
23      }
24
```

1.3. Implementación de Advices

Veamos cómo se implementan los advices que ejecutan el código deseado para nuestro aspecto. Lo primero que tenemos que decidir es cuándo queremos que se ejecute este código, tenemos varias opciones:

- **@Before:** es la que utilizamos en el ejemplo anterior. El advice se ejecutará **antes** de la ejecución del método interceptado. **Si el advice lanza una excepción NO se ejecutará el método.**

- **@AfterReturning:** el advice se ejecutará **después de que el método haya terminado con éxito**. Opcionalmente podremos acceder al objeto devuelto por el método, utilizando la propiedad returning (implica que el método debe devolver un objeto de este tipo).

```
@AfterReturning(value="execution(* com..PatientService.find*(..))",
returning="patients")
public void returningPatientsWithJoinPoint(JoinPoint joinPoint, List patients) {
  String name = joinPoint.getSignature().toShortString();
  Object roomId = joinPoint.getArgs()[0];
  System.out.println(name + " devolviento "+patients.size()+
" pacientes en habitación "+roomId);
}
```

- **@AfterThrowing:** el advice se ejecutará **después de que el método lance una excepción del tipo indicado**. Esto no impedirá que la excepción se siga propagando aunque, opcionalmente, podremos lanzar una excepción de un tipo diferente.

```
@AfterThrowing(value="execution(* com..PatientService.find (..))",
throwing="incorrectResultSizeDataAccessException")
public void returningPatientsWithJoinPoint(JoinPoint joinPoint,
IncorrectResultSizeDataAccessException incorrectResultSizeDataAccessException) {
  String name = joinPoint.getSignature().toShortString();
  Object roomId = joinPoint.getArgs()[0];
  System.out.println("@AfterThrowing -> " + name
+ " ocurrió un error al buscar pacientes en habitación "+roomId);
}
```

- **@After:** el advice se ejecuta independientemente de si el método terminó con éxito o lanzó una excepción.

```
@After(value="execution(* com..PatientService.find*(..))")
public void returningWithJoinPoint(JoinPoint joinPoint) {
  String name = joinPoint.getSignature().toShortString();
  Object roomId = joinPoint.getArgs()[0];
  System.out.println("@After -> " + name
+ " terminó de buscar pacientes en habitación "+roomId);
}
```

- **@Around:** este advice es un poco diferente a los demás, puesto que lo que hace es "rodear" la ejecución del método y, por lo tanto, seremos nosotros los encargados de decidir cuando la ejecución debe continuar y devolver el objeto esperado o lanzar una excepción. Para ello, recibiremos un el JoinPoint como tipo ProceedingJoinPoint y haremos uso del método proceed(). En este caso, sí que podremos interceptar una excepción y hacer que no se propague.

```
@Around(value="execution(* com.hospital.*.PatientDao+.findByRoom(*))")
public List aroundWithJoinPoint(ProceedingJoinPoint joinPoint)
throws Throwable {
  String name = joinPoint.getSignature().toShortString();
  Object roomId = joinPoint.getArgs()[0];
  System.out.println("@Around -> " + name
+ " va a buscar pacientes en la habitación "+roomId);
  List patients = (List) joinPoint.proceed();
```

```
8       System.out.println("@Around -> " + name
9     + " terminó de buscar en la habitación "+roomId);
        return patients;
      }
```

Por último, también tenemos la posibilidad de acceder a la información de contexto utilizando la notación del Advice en lugar de recibir directamente el objeto de tipo JoinPoint, lo que nos evita tener que hacer casting a los tipos de objetos que esperamos (JointPoint trabaja con Object):

```
      @Before("execution(* com.hospital.*.PatientService.findByRoom(*))
1     && target(patientService) && args(roomId)")
2     public void findingPatientsWithJoinPoint(PatientService patientService,
      Integer roomId) {
3       String name = patientService.toString();
4       System.out.println("@Before (selección contexto) -> " + name +
5     " buscando pacientes en habitación "+roomId);
      }
```