

## Programación Orientada al Aspecto (AOP)

La Programación Orientada a Aspectos (POA u AOP) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden encapsular los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos.

La mejor forma de entender en qué consiste y que proporciona es viendo un pequeño ejemplo utilizando de paso el framework Spring.

Antes de nada definiremos algunos términos:

- **Aspect (Aspecto)** es la funcionalidad que se cruza a lo largo de la aplicación (cross-cutting) que se va a implementar de forma modular y separada del resto del sistema. El ejemplo más común y simple de un aspecto es el logging (registro de sucesos) dentro del sistema, ya que necesariamente afecta a todas las partes del sistema que generan un suceso.
- **Jointpoint (Punto de Cruce)** es un punto de ejecución dentro del sistema donde un aspecto puede ser conectado, como una llamada a un método, el lanzamiento de una excepción o la modificación de un campo. El código del aspecto será insertado en el flujo de ejecución de la aplicación para añadir su funcionalidad.
- **Advice (Consejo)** es la implementación del aspecto, es decir, contiene el código que implementa la nueva funcionalidad. Se insertan en la aplicación en los Puntos de Cruce.
- **Pointcut (Puntos de Corte)** define los Consejos que se aplicarán a cada Punto de Cruce. Se especifica mediante Expresiones Regulares o mediante patrones de nombres (de clases, métodos o campos), e incluso dinámicamente en tiempo de ejecución según el valor de ciertos parámetros.

## Paso a Paso.

1. Definir una interfaz, con los métodos que queremos trabajar y que estén accesibles.
2. Implementar dicha interfaz
3. Implementar un Consejo o Advice. Que es una clase que interceptará la ejecución de métodos hay 3 posibilidades dependiendo de las necesidades.
  - 3.1 Se implementa MethodInterceptor para realizar operación antes y después de la invocación del método.
  - 3.2 Se implementa MethodBeforeAdvice para realizar operaciones antes de la ejecución del método.
  - 3.3 Se implementa AfterReturningAdvice para realizar operaciones después de la ejecución del método.
4. Configurar el archivo spring-config.xml correctamente. Definiremos dentro de `<beans>` :
  - 4.1 Implementaciones de la interfaz `<bean id = "nombre" class = "Clase que lo implementa"/>`
  - 4.2 Implementaciones de la interceptación de metodos `<bean id = "nombre" class = "Clase que lo implementa"/>`
  - 4.3 Definición del Point Cut

El Point Cut o punto de corte, define un patrón para el que se aplicará un Advice (o Consejo). Tiene la siguiente estructura

```
<bean id = "nombre del PointCut" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
<!--Patron que utilizará para aplicar el Consejo-->
  <property name = "pattern">
    <value>.*.</value>
  </property>
<!--Referencia a un Bean que implementa un Advice (Punto 3)-->
  <property name = "advice">
    <ref bean = "metobdd"/>
  </property>
</bean>
```

Esto según el patrón establecer un patrón para todos los métodos que hereden de una clase, todos los métodos que comience por, que contengan , etc.

Supongamos que tenemos una clase que tiene métodos de acceso a base de datos, insertar, borrar, actualizar, consultar. Para cada operación hay que abrir y cerrar la conexión con la base de datos por lo que estamos repitiendo código.

```
Public class GestionUsuarioImpl implements GestionUsuario{
public Usuario consultaUsuario( String usuario ) {
//Abriendo Conexión con BBDD
//Usuario= consulta a la base de datos.
// "Cerrando Conexión con BBDD
return usuario;
}
```

```

    public void agregarUsuario( String usuario ) {
//Abriendo Conexión con BBDD
//Insertar Usuario
// "Cerrando Conexión con BBDD
    }

    public void actualizarUsuario( String usuario ) {
//Abriendo Conexión con BBDD
//Actualizar usuario
// "Cerrando Conexión con BBDD
    }

    public void borrarUsuario( String usuario ) {
//Abriendo Conexión con BBDD
//Borrar Usuario
// "Cerrando Conexión con BBDD
    }
}

```

En todos los casos se repite Abrir Conexión y Cerrar Conexión.

Mediante Spring AOP nos permite siguiendo un patrón la ejecutar de código antes, después o antes y después de la ejecución de un método. En este caso servirá para evitar duplicar código innecesario, realizando una abstracción para los procesos comunes.

También podría servir para poder implementar de una manera sumamente fácil la trazabilidad de un programa, calcular los tiempos de ejecución de los método, etc.

Veamos pues como seria con Spring.

## 1. Creamos la interfaz de métodos a manejar

```

public interface GestionUsuario {

    public String consultaUsuario( String usuario );
    public boolean agregarUsuario( String usuario );
    public int actualizarUsuario( String usuario );
    public boolean borrarUsuario( String usuario );

}

```

## 2. Implementamos dichos métodos, sin el código repetido.

```

public class GestionUsuarioImpl implements GestionUsuario{

    public String consultaUsuario( String usuario ) {
        System.out.println( "BD2 Consultando el Usuario : " + usuario );
        return usuario;
    }
    public String consulta2Usuario( String usuario ) {
        System.out.println( "BD2 Consultando el Usuario : " + usuario );
        return usuario;
    }

    public boolean agregarUsuario( String usuario ) {
        System.out.println( "BD2 Insertando Usuario : " + usuario );
        return true;
    }
}

```

```

public int actualizarUsuario( String usuario ) {
    System.out.println( "BD2 Actualizando Usuario : " + usuario );
    return 1;
}

public boolean borrarUsuario( String usuario ) {
    System.out.println( "BD2 Borrando Usuario : " + usuario );
    return true;
}

public void procesarInformacion() {
    System.out.println( "BD2 Info Procesada Usuario");
}
}

```

### 3. Creamos una clase que interceptara la ejecución de los métodos.

```

//La clase tiene que implementar MethodInterceptor para interceptar métodos, que consiste en implementar el método
invoke.
public class MethodInterceptorBBDD implements MethodInterceptor {
    public Object invoke( MethodInvocation methodInvocation ) throws Throwable {
        System.out.println( "Establecer la conexion" );
    }
    //La siguiente setencia ejecuta la ejecución del método interceptado
    Object objeto = methodInvocation.proceed();
    System.out.println( "Cerrar la conexion" );
    return objeto;
}
}

```

### 4. Configuramos el archivo spring-config.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <!--Clase que implementa la interfaz de acceso a bbdd-->
    <bean id = "usuarioServiceTarget" class = "Service.GestionUsuarioImpl"/>
    <!--Clase que implementa la intercepción de metodos para el Saber el tiempo de ejecucion-->
    <bean id = "metodolog" class = "Eventos.EventoMethodInterceptorAdvice"/>
    <!--Clase que implementa la intercepción de metodos para abrir y cerrar la conexion a bbdd-->
    <bean id = "metobbdd" class = "Eventos.MethodInterceptorBBDD"/>
    <!--Clase que implementa la intercepción de metodos antes de su ejecucion-->
    <bean id = "eventoBeforeAdvice" class = "Eventos.EventoBeforeAdvice"/>
    <!--Clase que implementa la intercepción de metodos despues de su ejecucion-->
    <bean id = "eventAfterAdvice" class = "Eventos.EventoAfterAdvice"/>

    <!--Bean que asocia la interfaz con la implementación del interceptor de métodos con el patrón de métodos que
    interceptará-->
    <!--Se trata de un Advice o Consejo que tiene definido un PointCut MethodPointCut Advisor-->
    <bean id = "consultaPointCut" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name = "pattern">
            <value>.*</value>
        </property>
        <property name = "advice">
            <ref bean= "metobbdd"/>
        </property>
    </bean>

```

```

<!--Bean que asocia la interfaz, con la implementación anteriormente descritas y el interceptor de metodos-->
<bean id = "usuarioService" class = "org.springframework.aop.framework.ProxyFactoryBean">
  <property name = "proxyInterfaces">
    <value>Service.GestionUsuario</value>
  </property>
  <property name = "interceptorNames">
    <list>
      <!--<value>eventAfterAdvice</value-->
      <!--<value>eventoBeforeAdvice</value-->
      <value>consultaPointCut</value>
    </list>
  </property>
  <property name = "target">
    <ref bean = "usuarioServiceTarget"/>
  </property>
</bean>

</beans>

```

Con esta configuración conseguimos que todos los métodos de la interfaz GestionUsuario sean capturados, interceptados por la clase MethodInterceptorBBDD de tal manera que podrá abrir y cerrar la conexión con la bbdd de manera que no se repita código y este centralizado en un solo punto.

```

public class Principal {

    private static ApplicationContext ctx;
    private String usuario;

    public Principal() {
        String[] paths = { "src/spring-config.xml" };
        ctx = new FileSystemXmlApplicationContext(paths);
        usuario = "Pepe";
        GestionUsuario service = (GestionUsuario)ctx.getBean( "usuarioService" );
        service.consultaUsuario( usuario );
        service.actualizarUsuario( usuario );
    }

    public static void main(String[] args) {
        Principal programa = new Principal();
    }

}

```

La salida es:

```

Establecer la conexion
BD2 Consultando el Usuario : DAO.Usuario@951a0
Cerrar la conexion
Establecer la conexion
BD2 Actualizando Usuario : DAO.Usuario@951a0
Cerrar la conexion
Establecer la conexion
BD2 Info Procesada Usuario
Cerrar la conexión

```

## Otros ejemplos de Advice o Consejos, para interceptar la ejecución de métodos.

Ejemplo 3.1 Antes y después de la ejecución del método.

```
public class EventoMethodInterceptorAdvice implements MethodInterceptor {
    public Object invoke( MethodInvocation methodInvocation ) throws Throwable {
        long tiempoInicial = System.currentTimeMillis();
        // Se procede a la invocacion del metodo.
        Object objeto = methodInvocation.proceed();
        System.out.println( "Tiempo de proceso del método ( milisegundos ) : " + ( System.currentTimeMillis() - tiempoInicial ) );
        return objeto;
    }
}
```

Ejemplo 3.2 Antes de la Ejecucion del Metodo

```
public class EventoBeforeAdvice implements MethodBeforeAdvice {
    public void before( Method method, Object[] parameters, Object target ) throws Throwable {
        System.out.println( "***** AOP BEFORE ADVICE *****" );
        //Del parámetro parameters[] podemos obtener el valor de los parámetros.
        System.out.println( "AOP BEFORE ADVICE : Metodo " + method.getName());
    }
}
```

Ejemplo 3.3 Despues de la ejecución del metodo

```
public class EventoAfterAdvice implements AfterReturningAdvice {
    public void afterReturning( Object valueReturned, Method method, Object[] parameters, Object target ) throws Throwable {
        System.out.println( "***** AOP AFTER ADVICE *****" );
        //Se puede accede a los parametros, el valor de retorno y método que se ha ejecutado.
        System.out.println("Valor de retorno: "+ valueReturned+ " del método "+ method.getName() );
    }
}
```

### Requisitos:

- Java
- Spring : spring.jar y commons-logging.jar

Se adjunta el proyecto en IntelliJ para poder ver, ejecutar y comprender mejor el funcionamiento. El proyecto incluye las librerías necesarias para la ejecución del mismo.

### Proyecto:

Abre el proyecto con IntelliJ o con Eclipse, configura la librería de Spring. Verifica que en la Clase Principal.java el path se corresponda con la ruta adecuada del archivo spring-config.xml. El código fuente está al final del documento.

### Conclusiones:

Aunque Spring AOP ofrece más opciones, aquí no comentadas, se puede percibir el potencial que tiene. A Spring AOP se puede combinar con los otros módulos de Spring DAO, Spring Hibernate, etc.

El concepto está más o menos claro y se entiende con facilidad, el mayor problema puede surgir a la hora de configurar correctamente el archivo spring-config.xml con este pequeño ejemplo, mini manual, se espera facilitar el aprendizaje de esta tecnología.

Proyecto Spring AOP: <http://kit.medianet.es/blogkit/wp-content/uploads/2008/03/SpringAOP.rar>

**Media Net Software**

[www.medianet.es](http://www.medianet.es)