Programación orientada a aspectos

La programación orientada a aspectos (AOP) permite modularidad en las aplicaciones.

Veamos un ejemplo que ilustra la idea del AOP: una clase Coche podría tener el método arrancar. Antes de arrancar es necesario entrar al coche, introducir la llave y comprobar los espejos. Estas acciones anteriores al arranque podrían ser implementadas separadamente como un Aspecto.

El objetivo de POA es separar las funcionalidades dentro del comportamiento de una clase.

- Funcionalidades específicas de los objetos de la clase.
- Funcionalidades asociadas al comportamiento del objeto, pero no que están directamente relacionadas con él.

Los conceptos principales de POA son:

- Aspect: funcionalidad que se va a implementar de forma modular y separada. En el ejemplo expuesto anteriormente entrar al coche, introducir la llave y comprobar los espejos.
- Pointcut: punto de ejecución dentro del sistema donde el aspecto está conectado. En nuestro ejemplo, el punto de ejecución será justo antes del método arrancar.

El concepto de AOP lo encontramos en Spring a partir de la versión 2.0.

Se configura por medio de anotaciones o a través de XML.

Veamos un ejemplo práctico:

Las dependencias Maven que vamos a necesitar son las siguientes:

```
<!--
https://mvnrepository.com/artifact/org.springframework/spri
ng-core -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>6.1.14</version>
</dependency>
<!--
https://mvnrepository.com/artifact/org.springframework/spri
ng-context -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.1.14</version>
</dependency>
<!--
https://mvnrepository.com/artifact/org.springframework/spri
ng-aop -->
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>6.1.14</version>
</dependency>
<!--
https://mvnrepository.com/artifact/org.springframework/spri
ng-aspects -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>6.1.14</version>
</dependency>
```

Creamos la clase Coche con el método arrancar que tendrá asociado un aspecto:

```
package org.example;

public class Coche {
    private String matricula;
    private String modelo;

    public Coche(String matricula, String modelo) {
        this.matricula = matricula;
        this.modelo = modelo;
    }

    public String getMatricula() {
        return matricula;
    }
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public void arrancar() {
        System.out.println ("El coche ha arrancado");
    }

    @Override
    public String toString() {
        return "Coche{" +
                "matricula='" + matricula + '\'' +
                ", modelo='" + modelo + '\'' +
                '}';
```

```
    }
}
```

Creamos la clase con la lógica del aspecto:

Antes de arrancar el coche, un conductor tendrá que colocar los espejos, regular el asiento y ponerse el cinturón de seguridad.

Después de arrancar el conductor tendrá que conducir con cuidado.

```
package org.example;

public class Conductor {
    public void antesArrancar() {
        System.out.println("Colocamos los espejos");
        System.out.println("Regulamos el asiento");
        System.out.println("Nos ponemos el cinturón de seguridad");
    }

    public void despuesArrancar() {
        System.out.println("Conducimos con cuidado");
    }
}
```

Ahora, en el archivo context.xml definimos los beans para coche y preparación, así como la configuración del aspecto y los puntos de ruptura.

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


    <bean id="coche" class="org.example.Coche">
        <constructor-arg index="0" value="5587ABC" type="java.lang.String"/>
        <constructor-arg index="1" value="Opel Corsa" type="java.lang.String"/>
    </bean>

    <bean id="prepacion" class="org.example.Conductor"/>

    <aop:config>
        <aop:aspect ref="prepacion">
```

```
        <aop:pointcut id="puntoArrancar"
            expression="execution(* org.example.Coche.arrancar(..))"/>
        <aop:before method="antesArrancar"
            pointcut-ref="puntoArrancar"/>
        <aop:after-returning method="despuesArrancar"
              pointcut-ref="puntoArrancar"/>
    </aop:aspect>
  </aop:config>

</beans>
```

Por último, obtenemos el bean coche e invocamos al método arrancar.

```
package org.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App
{
   public static void main( String[] args )
   {
      ApplicationContext context = new ClassPathXmlApplicationContext("context.xml");

      Coche c = (Coche) context.getBean("coche");

      c.arrancar();

      ((ClassPathXmlApplicationContext) context).close();

   }
}
```

Análisis del archivo context.xml

Lo primero que hemos hecho es declarar los beans que se van a utilizar y luego hemos incluido una etiqueta <aop:config> donde se concreta el trabajo de la AOP.

Para definir el aspecto utilizamos la etiqueta <aop:aspect ...> haciendo referencia al bean que representa dicho aspecto. También puede llevar un identificador.

```
<aop:aspect ref="prepacion" id="myAspect">
            ….
</aop:aspect>
```

Mediante la etiqueta <aop:pointcut…. /> definimos el punto del código donde vamos a poder intervenir dentro de nuestros beans.

```
<aop:pointcut id="puntoArrancar"
          expression="execution(* org.example.Coche.arrancar(..))"/>
```

```
<aop:before method="antesArrancar"
        pointcut-ref="puntoArrancar"/>
<aop:after-returning method="despuesArrancar"
            pointcut-ref="puntoArrancar"/>
```
Vamos a examinar un poco esta expresión:

```
            expression="execution(* org.example.Coche.arrancar(..))"/>
```

Estamos especificando que el punto de código donde se puede intervenir será cualquier método llamado arrancar dentro de la clase Coche, dentro del paquete org.example.

Con el asterisco del principio estamos indicando que para cualquier tipo de valor de retorno.

Con (..) estamos indicando que será válido para un método saludar con cualquier tipo de parámetros.

Así estaríamos interviniendo sobre cualquier método de la clase Coche.

```
            expression="execution(* org.example.Coche.*(..))"/>
```

Así estaríamos interviniendo sobre cualquier método de cualquier clase del paquete example.

```
            expression="execution(* org.example.*.*(..))"/>
```

¿Cómo puede la clase que implementa el aspecto comunicarse con el objeto?

```
<aop:config>
    <aop:aspect ref="prepacion">
      <aop:pointcut id="puntoArrancar"
        expression="execution(* org.example.Coche.arrancar(..)) and target (co)" />
      <aop:before method="antesArrancar"
      pointcut-ref="puntoArrancar" arg-names="co"/>
      <aop:after-returning method="despuesArrancar"
        pointcut-ref="puntoArrancar" arg-names="co"/>
    </aop:aspect>
  </aop:config>
```

Bajo el identificador “co” hacemos referencia al objeto de tipo Coche que se pasa al aspecto para que este tenga pleno acceso a las propiedades y métodos de este. Ahora podemos modificar la clase Conductor de la siguiente manera:

package org.example;

```
public class Conductor {
  public void antesArrancar(Coche co) {
    System.out.println("Colocamos los espejos al coche "+co.getMatricula());
    System.out.println("Regulamos el asiento del "+co.getMatricula());
```

```
        System.out.println("Nos ponemos el cinturón de seguridad en el coche
"+co.getModelo());
    }

    public void despuesArrancar(Coche co)
    {
        System.out.println("Conducimos con cuidado el coche "+co.getModelo());
    }
}
```

Programación orientada a aspectos con anotaciones

La clase que actúa como aspecto deberá estar anotada con @Aspect y los métodos que implementan la lógica del aspecto (Advice) deben ser marcados con @Before, @After, @AfterReturning, etc. El argumento de estás anotaciones será el pointcut o punto de intervención.

Archivo context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop.xsd">

       <context:component-scan base-package="org.example"/>

       <aop:aspectj-autoproxy />

</beans>
```

La línea <aop:aspectj-autoproxy /> activa el AOP por medio de anotaciones.

La clase Coche

package org.example;

import org.springframework.stereotype.Service;

```
@Service
public class Coche {
    private String matricula;
    private String modelo;
```

```
    public Coche() {
        this.matricula = "5577ABC";
        this.modelo = "Ford Fiesta";
    }

    public String getMatricula() {
        return matricula;
    }
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public void arrancar() {
        System.out.println ("El coche ha arrancado");
    }

    @Override
    public String toString() {
        return "Coche{" +
                "matricula='" + matricula + '\'' +
                ", modelo='" + modelo + '\'' +
                '}';
    }
}
```

La clase PreparарCoche

```
package org.example;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Service;

@Service
@Aspect
public class Conductor {
    @Before(value="execution(* org.example.Coche.arrancar(..)) && target (co)",
argNames="co")
    public void antesArrancar(Coche co) {
        System.out.println("Colocamos los espejos al coche "+co.getMatricula());
        System.out.println("Regulamos el asiento del "+co.getMatricula());
        System.out.println("Nos ponemos el cinturón de seguridad en el coche
"+co.getModelo());
```

```
    }

    @AfterReturning(value="execution(* org.example.Coche.arrancar(..)) && target
(co)", argNames="co")
    public void despuesArrancar(Coche co)
    {
        System.out.println("Conducimos con cuidado el coche "+co.getModelo());
    }
}
```