Spring Framework

La Programación Orientada a Aspectos AOP (Aspect Oriented Programming) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de responsabilidades.

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. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables.

La programación orientada a objetos (OOP) supuso un gran paso en la ingeniería del software, ya que presentaba un modelo de objetos que parecía encajar de manera adecuada con los problemas reales. La cuestión era saber descomponer de la mejor manera el dominio del problema al que nos enfrentáramos, encapsulando cada concepto en lo que se dicen llamar objetos y haciéndoles interactuar entre ellos, habiéndoles dotado de una serie de propiedades. Surgieron así numerosas metodologías para ayudar en tal proceso de descomposición y aparecieron herramientas que incluso automatizaban parte del proceso. Esto no ha cambiado y se sigue haciendo en el proceso de desarrollo del software. Sin embargo, frecuentemente la relación entre la complejidad de la solución y el problema resuelto hace pensar en la necesidad de un nuevo cambio. Así pues, nos encontramos con muchos problemas donde la OOP no es suficiente para capturar de una manera clara todas las propiedades y comportamientos de lo que queremos dotar nuestra aplicación. Así mismo, la programación procedural tampoco nos soluciona el problema. Por lo que ante tales problemas, se utiliza la AOP.

Los conceptos y tecnologías reunidos bajo el nombre "programación orientada a aspectos" buscan resolver un problema identificado hace tiempo en el desarrollo de software. Se trata del problema de la separación de incumbencias (concerns).

La programación orientada a aspectos (POA) es una nueva metodología de programación que aspira a soportar la separación de competencias para los aspectos antes mencionados. Es decir, que intenta separar los componentes y los aspectos unos de otros, proporcionando mecanismos que hagan posible abstraerlos y componerlos para formar todo el sistema. En definitiva, lo que se persigue es implementar una aplicación de forma eficiente y fácil de entender.

AOP es un desarrollo que sigue al paradigma de la orientación a objetos, y como tal, soporta la descomposición orientada a objetos, además de la procedimental y la descomposición funcional. Pero, a pesar de esto, AOP no se puede considerar como una extensión de la OOP, ya que puede utilizarse con los diferentes estilos de programación ya mencionados.

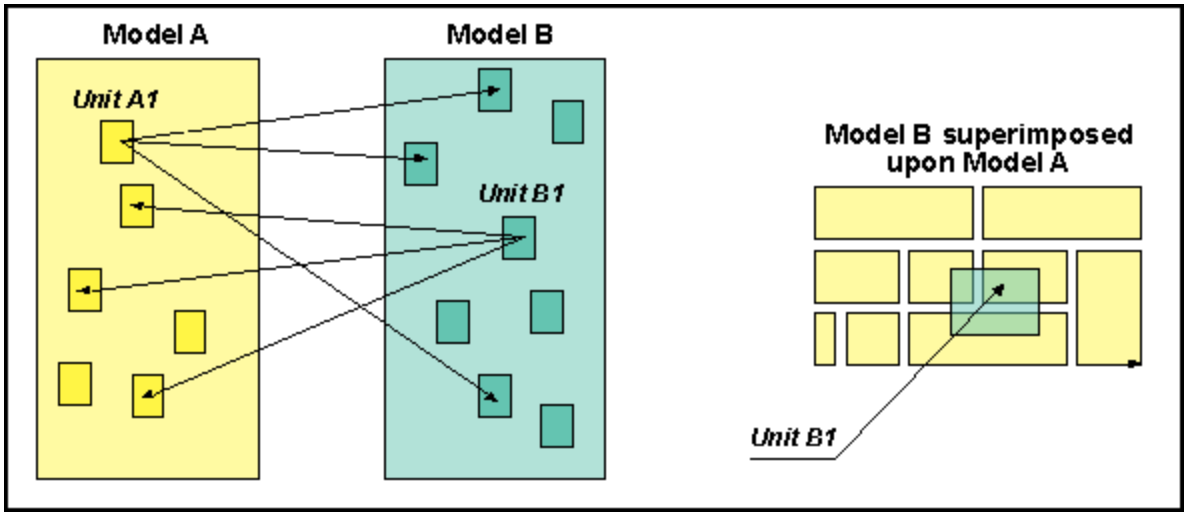
Principio Separación de Incumbencias/Asuntos.

El principio de separación de incumbencias fue identificado en los años 70, plantea que un problema dado involucra varias incumbencias que deben ser identificadas y separadas. Las incumbencias son los diferentes temas o asuntos de los que es necesario ocuparse para resolver el problema. Una de ellas es la función específica que debe realizar una aplicación, pero también surgen otras como por ejemplo distribución, persistencia, replicación, sincronización, etc. Separando las incumbencias, se disminuye la complejidad a la hora de tratarlas y se puede cumplir con requerimientos relacionados con la calidad como adaptabilidad, mantenibilidad, extensibilidad y reusabilidad.

El principio puede aplicarse de distintas maneras. Por ejemplo, separar las fases del proceso de desarrollo puede verse como una separación de actividades de ingeniería en el tiempo y por su objetivo. Definir subsistemas, objetos y componentes son otras formas de poner en práctica el principio de separación de incumbencias. Por eso podemos decir que se trata de un principio rector omnipresente en el proceso de desarrollo de software.

Las técnicas de modelado que se usan en la etapa de diseño de un sistema se basan en partirlo en varios subsistemas que resuelvan parte del problema o correspondan a una parte del dominio sobre el que trata. La desventaja de estas particiones es que muchas de las incumbencias a tener en cuenta para cumplir con los requerimientos no suelen adaptarse bien a esa descomposición.

El problema aparece cuando una incumbencia afecta a distintas partes del sistema que no aparecen relacionadas en la jerarquía. En ese caso, la única solución suele ser escribir código repetido que resuelva esa incumbencia para cada subsistema.



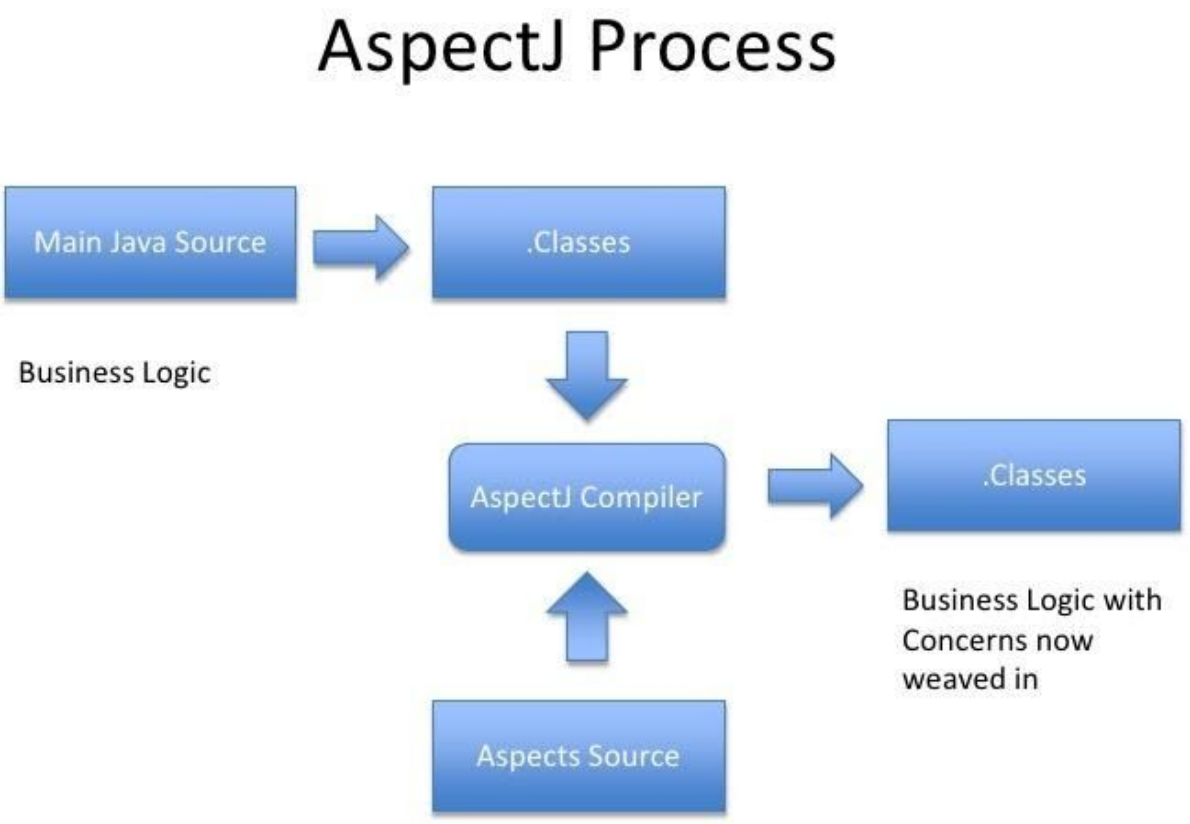
Fundamentos de la Programación Orientada a Aspectos

Para hacer un programa orientado a aspectos se necesita definir los siguientes elementos:

1. Un lenguaje para definir la funcionalidad básica. Este lenguaje se conoce como lenguaje base. Suele ser un lenguaje de propósito general, tal como C++ o Java.

2. El lenguaje de aspectos define la forma de los aspectos – por ejemplo, los aspectos de AspectJ se programan de forma muy parecida a las clases en Java.

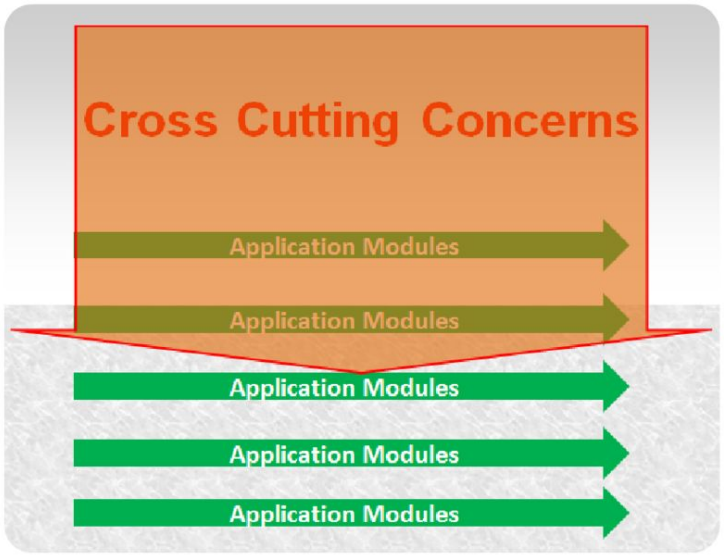
3. El compilador se encargará de combinar los lenguajes. El proceso de mezcla se puede retrasar para hacerse en tiempo de ejecución, o hacerse en tiempo de compilación.

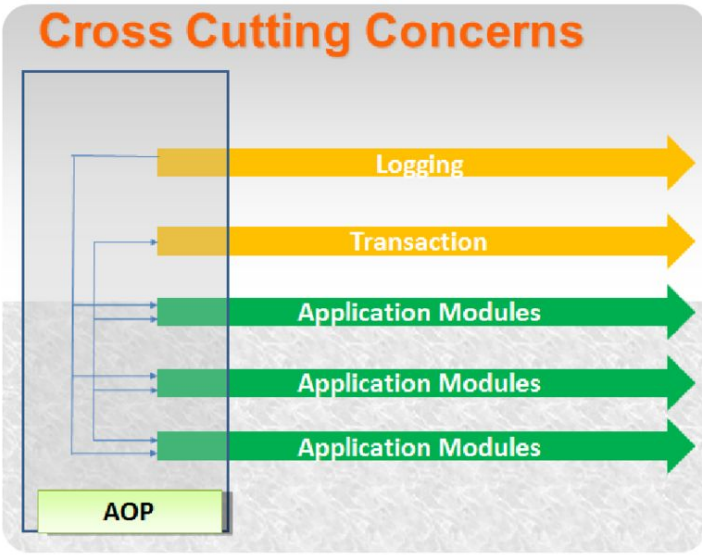


Características de AOP

De la consecución de estos objetivos se pueden obtener las siguientes ventajas:

1. Un código menos enmarañado, más natural y más reducido.
2. Una mayor facilidad para razonar sobre las materias, ya que están separadas y tienen una dependencia mínima.
3. Más facilidad para depurar y hacer modificaciones en el código.
4. Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.
5. Se tiene un código más reusable y que se puede acoplar y desacoplar cuando sea necesario.





¿Y qué puedo hacer con AOP?

La programación orientada a aspectos intenta formalizar y representar de forma concisa los elementos que son transversales a todo el sistema. En los lenguajes orientados a objetos, la estructura del sistema se basa en la idea de clases y jerarquías de clases. La herencia permite modularizar el sistema, eliminando la necesidad de duplicar código. No obstante, siempre hay aspectos que son transversales a esta estructura: el ejemplo más clásico es el de control de permisos de ejecución de ciertos métodos en una clase:

```
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
      ...
   }
}
```

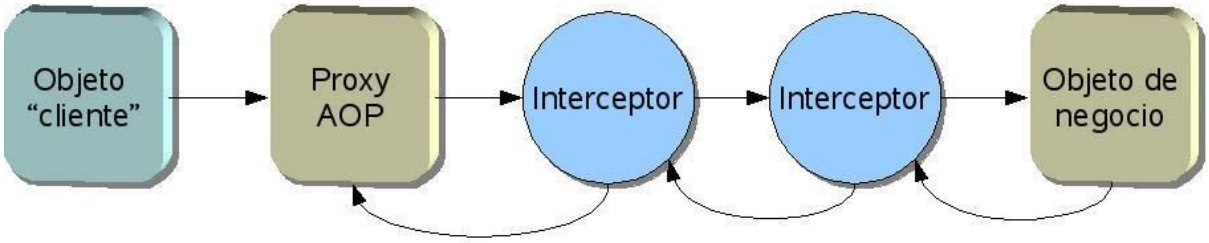
Estructurando adecuadamente el programa se puede minimizar la repetición de código, pero es prácticamente imposible eliminarla. La situación se agravaría si además tuviéramos que controlar permisos en objetos de varias clases. El problema es que en un lenguaje orientado a objetos los aspectos transversales a la jerarquía de clases no son modularizables ni se pueden formular de manera concisa con las construcciones del lenguaje. La programación orientada a aspectos intenta formular conceptos y diseñar construcciones del lenguaje que permitan modelar estos aspectos transversales sin duplicación de código. En nuestro ejemplo, se necesitaría poder especificar de alguna manera concisa

que antes de ejecutar ciertos métodos hay que llamar a cierto código.

En AOP, a los elementos que son transversales a la estructura del sistema y se pueden modularizar gracias a las construcciones que aporta el paradigma se les denomina aspectos (aspects). En el ejemplo anterior el control de permisos de ejecución, modularizado mediante AOP, sería un aspecto.

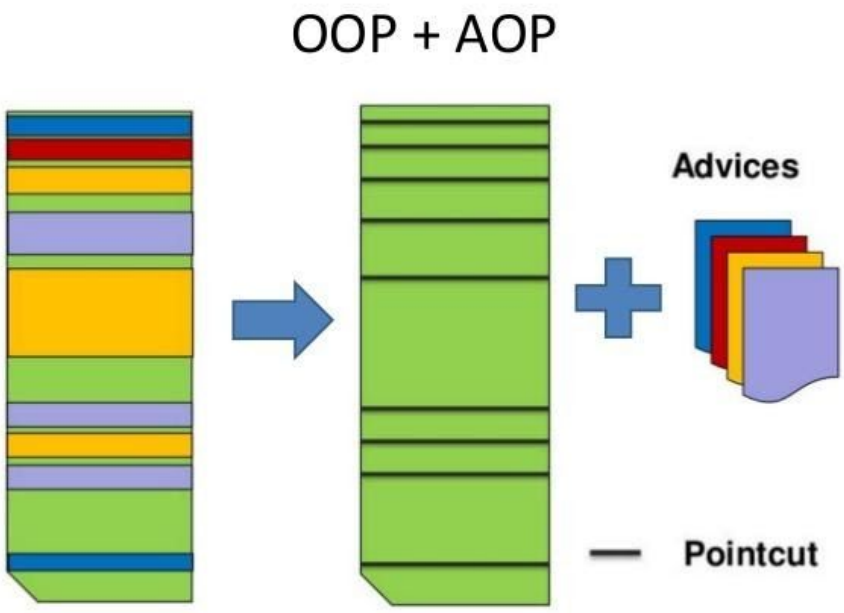
Un consejo (advice) es una acción que hay que ejecutar en determinado/s punto/s de un código, para conseguir implementar un aspecto. En el ejemplo, la acción a ejecutar sería la llamada a chequeaPermisos(). El conjunto de puntos del código donde se debe ejecutar un advice se conoce como punto de corte o pointcut. En nuestro caso serían los métodos metodoDeNegocio1() y metodoDeNegocio2().

En muchos frameworks de AOP (Spring incluido), el objeto que debe ejecutar esta acción se modela en la mayoría de casos como un interceptor: un objeto que recibe una llamada a un método propio antes de que se ejecute ese punto del código.



Cuando algún objeto llama a un método que forma parte del pointcut, el framework de AOP se las "arregla" para que en realidad se llame a un objeto proxy o intermediario, que tiene un método con el mismo nombre y signatura pero cuya ejecución lo que hace en realidad es redirigir la llamada por una cadena de interceptores hasta el método que se quería ejecutar.

En algunas ocasiones nos interesará usar un interceptor para interceptar las llamadas a todos los métodos de una clase. En otras solo nos interesará interceptar algunos métodos. En Spring, cuando deseamos interceptar las llamadas solo a algunos métodos debemos definir un advisor, que será una combinación de pointcut (dónde hay que aplicar AOP) más interceptor (qué hay que ejecutar).



AOP en Spring

Spring posee un muy buen módulo de AOP y siguiendo la filosofía de no reinventar la rueda, los desarrolladores de Spring han considerado que no era necesaria una sintaxis propia existiendo la de AspectJ ampliamente probado en la práctica. Hay que tener presente que no es lo mismo usar la sintaxis de AspectJ que usar AspectJ en sí.

Anotaciones vs. XML

Hay dos sintaxis alternativas para usar AOP en Spring. Una es mediante el uso de anotaciones en el propio código Java. La otra es con etiquetas en un archivo de configuración. Sin embargo, usando anotaciones podemos encapsular el AOP junto con el código Java en un único lugar. En teoría este es el sitio en que debería estar si el AOP es un requisito de negocio que debe cumplir la clase.

Para añadir soporte AOP a un proyecto Spring necesitaremos son necesarias 2 librerías: aspectjweaver.jar, y aspectjrt.jar. Además, si se quiere usar AOP con clases que no implementen ningún interface, se necesitaria la librería CGLIB.

Puntos de corte (pointcuts)

Un punto de corte o pointcut es un punto de interés en el código antes, después o "alrededor" del cual queremos ejecutar algo (un advice). Un pointcut no puede ser cualquier línea arbitraria de código. La versión actual de Spring solo soporta

puntos de corte en ejecuciones de métodos de beans. La implementación completa de AspectJ permite usar también el acceso a campos, la llamada a un constructor, etc, aunque esto en AOP de Spring no es posible.

Es importante destacar que al definir un pointcut realmente no estamos todavía diciendo que vayamos a ejecutar nada, simplemente marcamos un "punto de interés". La combinación de pointcut + advice es la que realmente hace algo útil.

Expresiones más comunes

La expresión más usada en pointcuts de Spring es execution(), que representa la llamada a un método que encaje con una determinada firma. Se puede especificar la firma completa del método incluyendo tipo de acceso (public, protected,...), tipo de retorno, nombre de clase (incluyendo paquetes), nombre de método y argumentos.

Teniendo en cuenta:

1. El tipo de acceso y el nombre de clase son opcionales, pero no así el resto de elementos
2. Se puede usar el comodín * para sustituir a cualquiera de ellos, y también el comodín .., que sustituye a varios tokens, por ejemplo varios argumentos de un método, o varios subpaquetes con el mismo prefijo.
3. En los parámetros, () indica un método sin parámetros, (..) indica cualquier número de parámetros de cualquier tipo, y podemos también especificar los tipos, por ejemplo (String, *, int) indicaría un método cuyo primer parámetro es String, el tercero int y el segundo puede ser cualquiera.

Por ejemplo, para especificar todos los métodos con acceso "public" de cualquier clase dentro del paquete ar.com.spring.aop pondríamos: execution(public * ar.com.spring.aop.*.*(..))

Expresiones más comunes

Se pueden combinar pointcuts usando los operadores lógicos &&, || y !, con el mismo significado que en el lenguaje C.

```
// todos los getters o setters de cualquier clase
execution (public * get*()) || execution (public void set*(*))
```

El operador && se suele usar en conjunción con args como una forma de "dar nombre" a los parámetros, por ejemplo: execution (public void set*(*)) && args(nuevoValor)

Pointcuts con nombre

Se puede asignar un nombre arbitrario a un pointcut (lo que se denomina una firma). Esto permite referenciarlo y reutilizarlo de manera más corta y sencilla que si tuviéramos que poner la expresión completa que lo define. La definición completa consta de la anotación @Pointcut seguida de la expresión que lo define y la signatura. Para definir la firma se usa la misma sintaxis que para definir la de un método Java en un interfaz. Eso sí, el valor de retorno debe ser void. Por

ejemplo:

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

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

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

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

Advices

Los advices son la pieza del puzzle que nos faltaba para que todo cobre sentido. Un advice es algo que hay que hacer en un cierto punto de corte, ya sea antes, después, o "alrededor" (antes y después) del punto.

Los advices se especifican con una anotación con el pointcut y la definición del método Java a ejecutar (firma y código del mismo). Como en Spring los puntos de corte deben ser ejecuciones de métodos los casos posibles son:

1. Antes de la ejecución de un método (anotación @Before)
2. Después de la ejecución normal, es decir, si no se genera una excepción (anotación @AfterReturning)
3. Después de la ejecución con excepción/es (anotación @AfterThrowing)
4. Después de la ejecución, se hayan producido o no excepciones (anotación @After)
5. Antes y después de la ejecución (anotación @Around)

Un aspecto (aspect) es un conjunto de advices. Siguiendo la sintaxis de AspectJ, los aspectos se representan como clases Java, marcadas con la anotación @Aspect. En Spring, además, un aspecto debe ser un bean, por lo que tendremos que anotarlo como tal

```
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class EjemploDeAspecto {

  //aquí vendrían los advices...
}
```

@Before

Esta anotación ejecuta un advice antes de la ejecución del punto de corte especificado. Por ejemplo:

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

@Aspect
public class EjemploBefore {

  @Before("execution(public * get*())")
  public void controlaPermisos() {
    // ...
  }

}
```

Ejecutaría controlaPermisos() antes de llamar a cualquier getter.

@AfterReturning

Esta anotación ejecuta un advice después de la ejecución del punto de corte especificado, siempre que el método del punto de corte retorne de forma normal (sin generar excepciones). Por ejemplo:

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

@Aspect
public class EjemploAfterReturning {

  @AfterReturning("execution(public * get*())")
  public void log() {
```

```
    // ...
  }

}
```

Para hacer log, nos puede interesar saber el valor retornado por el método del punto de corte. Este valor es accesible con la sintaxis alternativa:

```
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 log(Object valor) {
     // ...
  }
}
```

Al poner Object como tipo de la variable asociada al valor de retorno, estamos indicando que nos da igual el tipo que sea (incluso si es primitivo). Especificando un tipo distinto, podemos reducir el ámbito del advice para que solo se aplique a los puntos de corte que devuelvan un valor del tipo deseado.

@After

Esta anotación ejecuta un advice después de la ejecución del punto de corte especificado, genere o no una excepción, es decir, al estilo del finally de Java.

@Around

Esta anotación ejecuta parte del advice antes y parte después de la ejecución del punto de corte especificado. La filosofía consiste en que el usuario es el que debe especificar en el código del advice en qué momento se debe llamar al punto de corte. Por ello, el advice debe tener como mínimo un parámetro de la clase ProceedingJoinPoint, que representa el punto de corte. Llamando al método proceed() de esta clase, ejecutamos el punto de corte. Por ejemplo:

```
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 ejemploAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("ANTES");
        Object valorRetorno = pjp.proceed();
        System.out.println("DESPUES");
        return valorRetorno;
  }
}
```

Hibernate

Hibernate es una herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

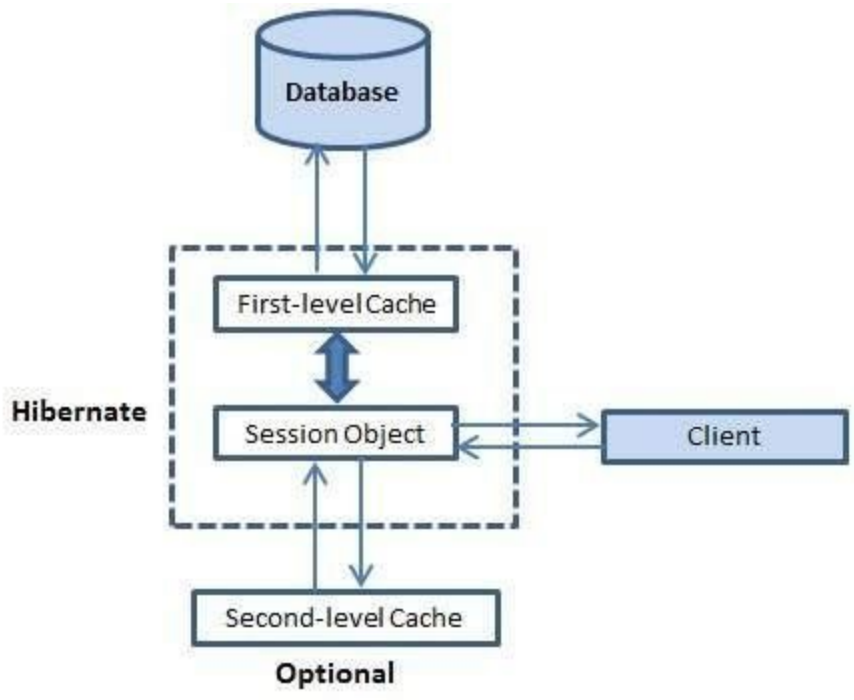
¿Por qué usar un framework ORM?

Cuando uno desarrolla una aplicación en la gran mayoría de los casos todo termina siendo un conjunto de ABMs (alta, baja, modificación de datos) para luego poder consultarlos. Para ello se utiliza una base de datos, donde existirán muchas tareas repetidas: por cada objeto que quiero persistir debo crear una clase que me permita insertalo, eliminarlo, modificarlo y consultarlo. Salvo aquellas consultas fuera de lo común, el resto es siempre lo mismo. Aquí es donde entra a jugar un rol importante un ORM: con solo configurarlo ya tiene todas esas tareas repetitivas realizadas y el

desarrollador solo tendrá que preocuparse por aquellas consultas fuera de lo normal.

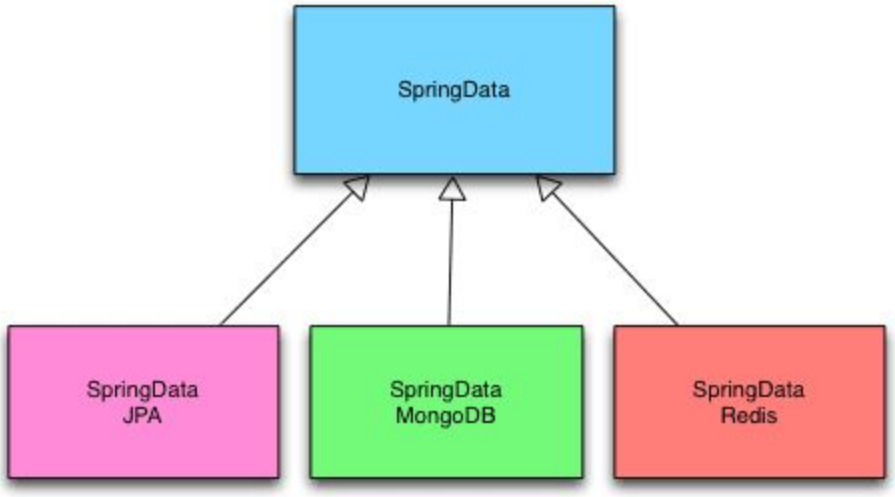
¿Cómo funciona Hibernate?

Básicamente el desarrollador deberá configurar en un archivo XML o mediante annotations donde corresponde un atributo de una clase, con una columna de una tabla. Es una tarea relativamente sencilla donde existen herramientas que lo hacen por nosotros.



Spring Data

Spring Data es uno de los frameworks que se encuentra dentro de la plataforma de Spring. Su objetivo es simplificar al desarrollador la persistencia de datos contra distintos repositorios de información.



El acceso a bases de datos es una de las tareas más comunes en el desarrollo de software, al principio esta tarea se realizaba simplemente haciendo uso de JDBC y poco a poco fue evolucionando utilizando patrones de diseño como el DAO y a través de frameworks y API's tales como Hibernate y JPA, estos ORM nos han ayudado a reducir muchas de las tareas que se hacían antes con JDBC.

Spring Data es un módulo de Spring que vamos a utilizar sobre JPA para hacer las tareas de acceso a base de datos aún más sencillas, las ventajas se notarán a simple vista ya que nosotros simplemente crearemos métodos en una interfaz y Spring Data se encargará de hacer las implementaciones por nosotros de tal modo que si nombramos un método como "findByName" Spring Data creará la implementación necesaria para buscar en la base de datos a través del nombre sin que nosotros creemos ni una sola conexión ni procesemos ningún resultado.