Práctica POA
Aspect J

Andres Esteban Romero Romero
Oscar Andres Mancera Garzón
Yerson Andres Valderrama Ceron

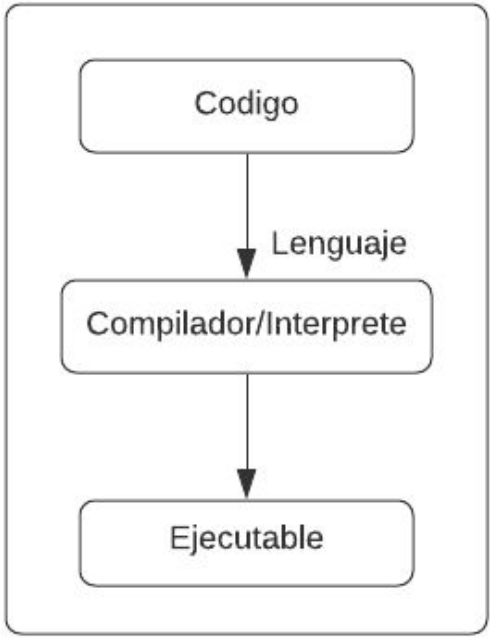
Fundamentos POA

Los principales requerimientos de la POA son:

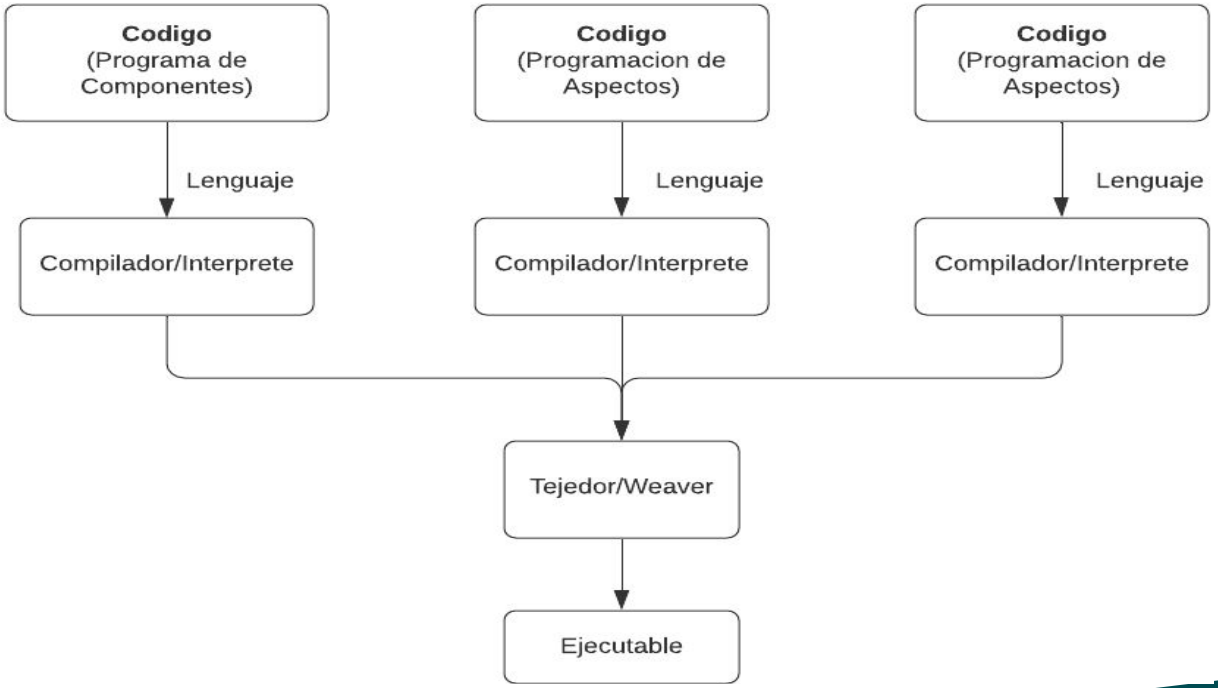
- Lenguaje para definir una funcionalidad básica.
- Lenguajes de aspectos para especificar el comportamiento de los aspectos.
- Un tejedor de aspectos (weaver) que se encarga de combinar los lenguajes.

Estructura general

La estructura de una implementación basada en aspectos es análoga a una implementación basada en los LPG.

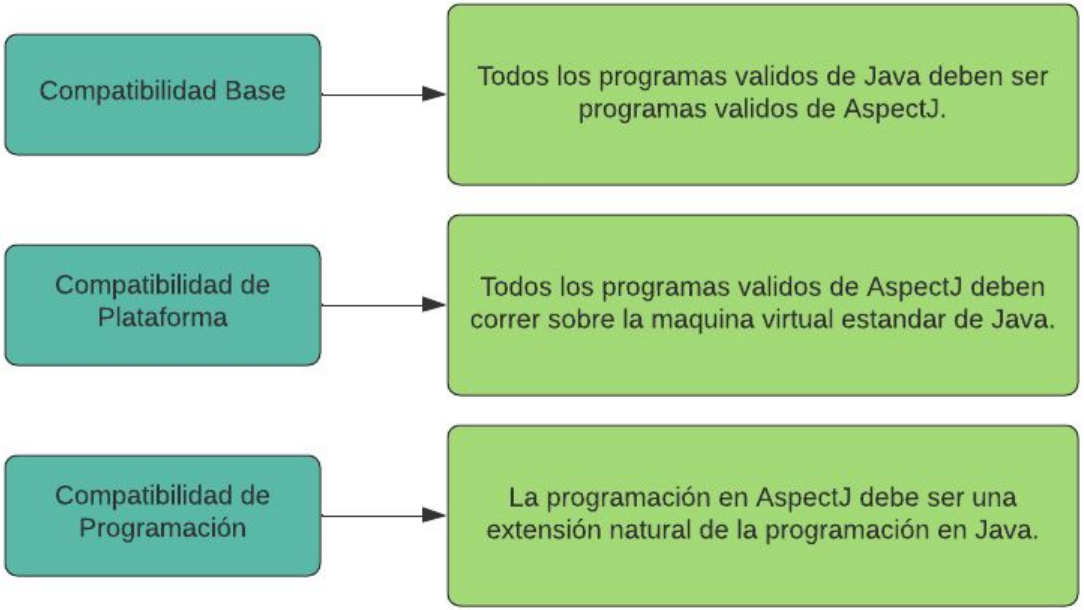


Estructura POA



Aspect J

Lenguaje orientado a aspectos, extension compatible de Java.



Crosscuting

La forma en que los aspectos “atraviesan” las clases de la aplicación principal. Existen dos tipos de crosscutting:

- Dinámico
- Estático

AspectJ = Java +

Dinámico

Los aspectos afectan al comportamiento de la aplicación, modificando o añadiendo nuevo comportamiento. Es el tipo de crosscutting más usado en AspectJ.

Estático

Los aspectos afectan a la estructura estática del programa (clases, interfaces y aspectos). La función principal del crosscutting estático es dar soporte a la implementación del crosscutting dinámico.

Aspects

Se declara de forma similar al de una clase, encapsula los puntos de cortes, avisos, instrucciones y declaraciones además de sus propios métodos y atributos.

Cómo definir un aspecto?

Si el aspecto afecta a distintas partes de la aplicación, lo normal es escribir el aspecto en un fichero independiente con extensión .aj.

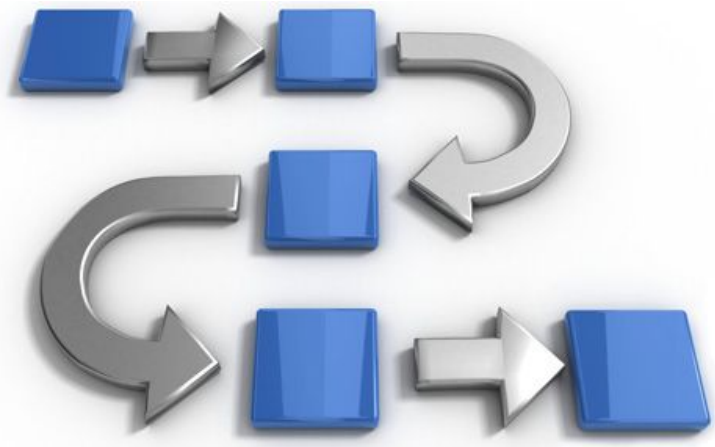
```
/**
 * Clase.java
 */
public class Clase {
	public static void main(String args[]) {
		System.out.println("Clase de ejemplo");
	}
}

/**
 * Aspecto.aj
 */
public aspect Aspecto {
	before():execution( void Clase.main(..) ) {
		System.out.println("Inicio metodo main");
	}
	after():execution( void Clase.main(..) ){
		System.out.println("Fin metodo main");
	}
}
```

Cuando el aspecto afecta a una única clase de la aplicación, optamos por escribir el aspecto como un miembro más de la clase

```
public class Clase {

	public static void main(String args[]) {
		System.out.println("Clase de ejemplo");
	}

	private static aspect Aspecto {

		before():execution( void Clase.main(..) ) {
			System.out.println("Inicio metodo main");
		}
		after():execution( void Clase.main(..) ){
			System.out.println("Fin metodo main");
		}
	}
}
```

Puntos de enlace (Jointpoints)

Son puntos bien definidos en la ejecución de un programa. No hay que confundirlos con posiciones en el código de un programa sino que son puntos sobre el flujo de ejecución de un programa.



llamada de un método:

```
punto.setX(10);
```

llamada de un constructor:

```
Punto p = new Punto(5,5);
```

ejecución de un método:

```
public void setX(int x) {

        this.x = x;

}
```

ejecución de un constructor:

```
public Punto(int x, int y){

        super();
        this.x = x;
        this.y = y;
}
```

lectura de un atributo:

```
public int getX() {
        return x;
}
```

escritura de un atributo:

```
public void setX(int i) {
        x = i;
}
```

ejecución de un manejador de excepciones:

inicialización de una clase:

```
try{

   ......................

}catch(IOException e){

        e.printStackTrace();

}
```

```
public class Main {

   ......................

   static {
        try {
          System.loadLibrary("lib");
        }
        catch(UnsatisfiedLinkErrore)
        {
        }
   }
   ......................
}
```

inicialización de un objeto:

```
public Punto(int x, int y)
{
        super();

        this.x = x;
        this.y = y;
}
```

pre-inicialización de un objeto:

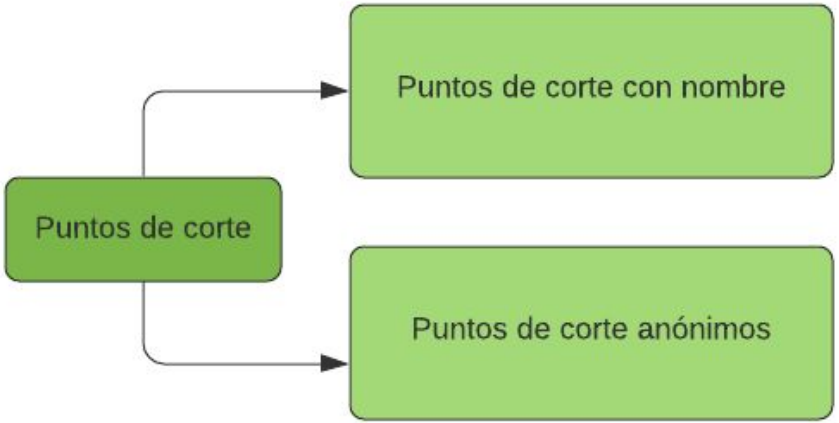
```
public CuentaAhorros(int numCuenta, String titular,
                     int saldoMinimo)
{
        super(titular,
              Cuenta.generarId()
              );
        this.saldoMinimo=saldoMinimo;
}
```

ejecución de un aviso:

```
after(Punto p): cambioPosicionPunto(p) {

        Logger.writeLog("...");

}
```

Puntos de corte (pointcuts)

Los puntos de corte identifican o capturan una serie de puntos de enlace y permiten exponer su contexto a los avisos (advice).



Puntos de corte anónimos

Un punto de corte anónimo consta de una o más expresiones que identifican una serie de puntos de enlace. Cada una de estas expresiones esta formada por un descriptor de punto de corte y una signatura.

El descriptor de punto de corte especifica el tipo de punto de corte y la signatura indica el lugar donde se aplica.

```
<pointcut> ::= { [!] designator [ && | || ] };

designator ::= designator_identifier(<signature>)
```

El descriptor de punto de corte especifica el tipo de punto de corte y la signatura indica el lugar donde se aplica

```
after(Punto p): call( * Punto.set*(int) ) && target(p)
{
      Logger.writeLog("Cambio posición Punto: "+p.toString());
}
```

Puntos de corte con nombre

La palabra reservada pointcut seguida del nombre del punto de corte y de sus parámetros, que permiten exponer el contexto de los puntos de enlace a los avisos (advice). Por último, vienen las expresiones que identifican los puntos de enlace precedidas del carácter dos puntos .

```
<pointcut> ::= <access_type> [abstract] pointcut
<pointcut_name>([<parameters>]) : {[!] designator [ && | || ]};

designator ::= designator_identifier(<signature>)
```

```
pointcut cambioPosicionPunto(Punto p):
            call( * Punto.set*(int) ) && target(p);
```

Signaturas

método	<access_type> <ret_val> <class_type>.<method_name>(<parameters>) [**throws** <exception>] public void Punto.setX(int) public void Linea.setP1(Punto)
constructor	<access_type> <class_type>.**new**(<parameters>)[**throws** <exception>] public Punto.new(int,int) public Linea.new(Punto, Punto)
atributo	<access_type> <field_type> <class_type>.<field_name> public int Punto.x public Punto Linea.punto1

Operadores

Caracteres

*

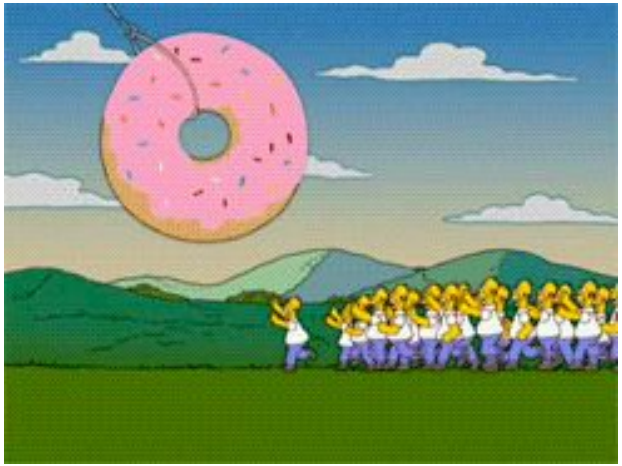
Parametros

...

Operadores

+

!



Operadores

&&

||

Algunos ejemplos

`public void Punto.set*(int)`	Todos los métodos públicos de la clase `Punto` que empiezan por `set`, devuelven `void` y tienen un único parámetro de tipo `int`.
`public void Punto.set*(*)`	Todos los métodos públicos de la clase `Punto` que empiezan por `set`, devuelven `void` y tienen un único parámetro de cualquier tipo.
`public void Punto.set*(..)`	Todos los métodos públicos de la clase `Punto` que empiezan por `set`, devuelven `void` y tienen cualquier número y tipo de parámetros.
`public * Punto.*()`	Todos los métodos públicos de la clase `Punto` que no tienen parámetros.
`public * Punto.*(..)`	Todos los métodos públicos de la clase `Punto` que devuelven cualquier tipo y tienen un número y tipo de parámetros arbitrario.
`* Punto.*(..)`	Todos los métodos de la clase *Punto*.

¡public * Punto.*(..)	Todos los métodos no públicos de la clase Punto.
* Elemento+.*(..)	Todos los métodos de clase *Elemento* y de sus subclases.
* java.io.Reader.read(char[],..)	Cualquier método `read` de la clase `java.io.Reader` que tenga un primer parámetro de tipo `char[]` y el resto de parámetros sea indiferente
* *.*(..) throws IOException	Todos los métodos que pueden lanzar la excepción `IOException`.
Punto.new()	Constructor por defecto de la clase *Punto*
public Elemento.new(..)	Todos los constructores públicos de la clase `Elemento`.
public Elemento+.new(..)	Todos los constructores públicos de la clase

Tipos de puntos de corte

basados en

- Categorias de puntos de enlace
- El flujo de control
  - Cflowbelow
  - Cflow
- Localizacion del codigo
  - Within
  - Withincode
- Los objetos en tiempo de ejecución
  - This
  - Target
- Los argumentos en los puntos de enlace
  - Args
- Condiciones
  - if

Punto de corte
call(void Punto.set*(..))
execute(public Elemento.new(..))
get(private * Linea.*)
set(* Punto.*)
handler(IOException)
initialization(Elemento.new(..))
staticinitialization(Logger)
adviceexecution() && within(MiAspecto)

Ejemplos basados en las categorías de puntos de enlace

`call(void Punto.set*(..))`	Llamadas a los métodos de la clase `Punto`, cuyo nombre empieza por `set`, devuelven `void` y tienen un número y de parámetros arbitrario.
`execute(public Elemento.new(..))`	Ejecución de todos los constructores de la clase `Elemento`
`get(private * Linea.*)`	Lecturas de los atributos privados de la clase `Linea`.
`set(* Punto.*)`	Escritura de cualquier atributo de la clase `Punto`.
`handler(IOException)`	Ejecución de los manejadores de la excepción `IOException`
`initialization(Elemento.new(..))`	Inicialización de los objetos de la clase `Elemento`
`staticinitialization(Logger)`	Inicialización estática de la clase `Logger`
`adviceexecution() && within(MiAspecto)`	Todas las ejecuciones de avisos pertenecientes al aspecto `MiAspecto`. No se puede capturar la ejecución de un único aviso.

basados en flujo de control

`cflow(call(* Cuenta.debito(..))`	Todos los puntos de enlace en el flujo de control del método `debito` incluida la propia llamada al método `debito`. El punto de corte que se pasa como parámetro es anónimo.
`cflowbelow(call(* Cuenta.debito(..))`	Todos los puntos de enlace en el flujo de control del método `debito`, sin incluir la llamada al método `debito`. El punto de corte que se pasa como parámetro es anónimo.
`cflow(operaciones())`	Todos los puntos de enlace en el flujo de control de los puntos de enlace capturados por el punto de corte con nombre `operaciones`
`cflowbelow(execution(Cuenta.new(..))`	Todos los puntos de enlace durante la ejecución de cualquier constructor de la clase `Cuenta`, sin incluir la propia ejecución del método.

basados en localización de código

`within(Elemento)`	Todos los puntos de enlace dentro de la clase `Elemento`.
`within(Elemento+)`	Todos los puntos de enlace dentro de la clase `Elemento` y sus subclases.
`withincode(* Punto.set*(..))`	Todos los puntos de enlace dentro del cuerpo de los métodos de la clase `Punto` cuyo nombre empieza por `set`.

Puntos de corte basados en objetos

`this(Punto)`	Todos los puntos de enlace donde el objeto actual (`this`) es una instancia de `Punto` o de alguno de sus descendientes
`target(Punto)`	Todos los puntos de enlace donde el objeto destino (`target`) es una instancia de `Punto` o de alguno de sus descendientes
`this(Elemento) && !within(Elemento)`	Todos los puntos de enlace donde el objeto actual es una instancia de alguna subclase de `Elemento`. Se excluye la propia clase `Elemento`.
`call(* *.*(..)) && target(Elemento)`	Todas las llamadas a los métodos de instancia de la clase `Elemento` y sus subclases. Con el siguiente punto de corte `call(*Elemento+.*(..))` también se capturarían las llamadas a los métodos estáticos.
`call(* Punto.*(..)) && this(Figura)`	Todas las llamadas a los métodos de la clase `Punto` realizadas desde un objeto de la clase Figura.
`set(* *.*) && target(Linea)`	Todas las asignaciones sobre cualquier atributo de la clase `Linea`.

Puntos de corte basados en argumentos

`call(* *.set*(..)) && args(String)`	Todos las llamadas a métodos, cuyo nombre empieza por `set` y tienen un único parámetro de tipo `String`. El siguiente punto de corte es equivalente: `call( * *.set*(String))`
`call( * *.set*(int))&& args(argInt)`	Todos las llamadas a los métodos cuyo nombre empieza por `set` y tienen un único parámetro de tipo `int`. El valor del parámetro se transfiere a la variable `argInt`

`call( * Punto.*(int,String)) && args(argInt, ..)`
 	
`call( * Punto.*(int,String)) && args(argInt,String)`
	Todas las llamadas a métodos de la clase `Punto` con dos argumentos, el primero de tipo int y el segundo de tipo String. El primero de los argumentos se transfiere a la variable `argInt`.

Puntos de corte basados en condiciones

Punto de corte	Puntos de enlace identificados
`if(true)`	Todos los puntos de enlace
`if(System.currentTimeMillis()>limite)`	Todos los puntos de enlace que ocurren tras sobrepasar cierta marca de tiempo.
`if(thisJoinPoint.getTarget() instanceof Linea)`	Todos los puntos de enlace cuyo objeto destino asociado sea una instancia de la clase `Linea`

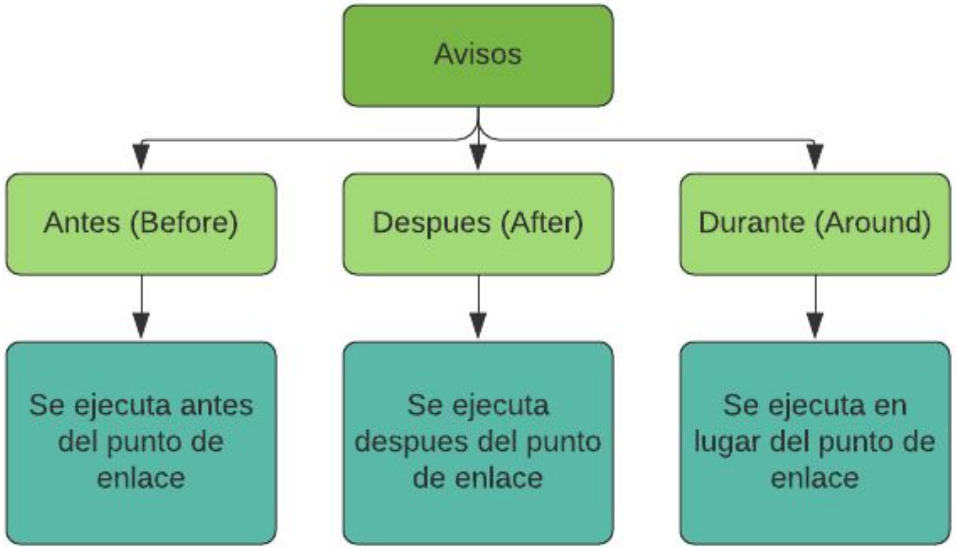
Advices (Avisos)

Establecen el código adicional que se ejecuta en los puntos de enlace capturados por los cortes.

Los avisos definen el comportamiento que entrecruza toda la funcionalidad, estan definidos en funcion de los cortes.

Advices (Avisos)

AspectJ soporta 3 tipos de avisos. El tipo de un aviso determina cómo este interactúa con el punto de enlace sobre el cual está definido.



Advices -Antes (Before)

Son el tipo de aviso más simple. El cuerpo del aviso se ejecuta antes del punto de enlace capturado no se ejecutaría.

Este tipo de avisos suele ser usado para realizar comprobacion de parametros, logging, autenticacion, entre otros.

Advices -Antes (Before)

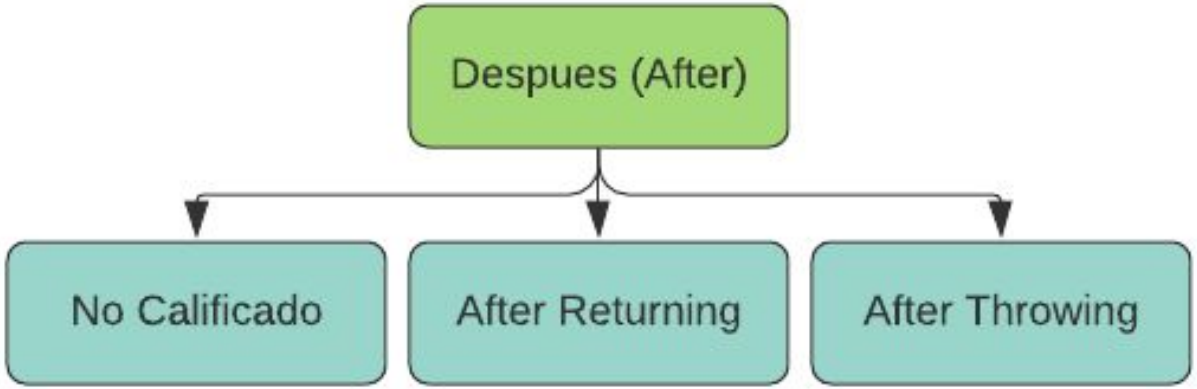
```
before(int cantidad) : call(void Cuenta.reintegro(int))
                       && args(cantidad)
{
      if(cantidad < Cuenta.MIN_REINTEGRO){
            throw new IllegalArgumentException("La cantidad a " +
            "reintegrar"+cantidad+ " es menor de lo permitido.");
      }
}
```

Advices -Antes (Before)

En el cuerpo del aviso anterior, se comprueba el parametro del metodo *reintegro*, y en el caso de que sea menor que el reintegro mínimo, se lanza una excepción (*IllegalArgumentException*) que provoca que no se ejecute el método.

Advices -Despues(After)

El cuerpo del aviso se ejecuta después del punto de enlace capturado.



Advice After (No calificado)

El aviso se calificara siempre, sin importar si el punto de enlace finalizó normalmente o con el lanzamiento de alguna excepcion.

Advice After (No calificado)

```
after(Punto p) cambioPosicionPunto(p)
{
        Logger.writeLog("Cambio posición Punto:"
                                    +p.toString());
}
```

Advice After Returning

El aviso solo se ejecutará si el punto de enlace termina normalmente, pudiendo acceder al valor de retorno devuelto por el punto de enlace.

Advice After Returning

El método *Logger.writeLog* se ejecutará tras la ejecución de todo punto de enlace.

```
after(Punto p) returning: cambioPosicionPunto(p){
        Logger.writeLog("Cambio posición Punto:"
                                +p.toString());
}
```

Advice After Returning

Un segundo ejemplo..

```
after() returning(int val_ret): call(* Punto.*(..)){
        System.out.println("Valor retornado "+val_ret);
}
```

Advice After Throwing

El aviso solo se ejecuta si el punto de enlace finaliza con el lanzamiento de una excepción, pudiendo acceder a la excepción lanzada.

Advice After Throwing

En el siguiente ejemplo el cuerpo del aviso se ejecutará tras las llamadas a los metodos publicos de cualquier clase que lancen una excepcion.

```
after() throwing (Exception ex): call(public * *.*(..)){
        contadorExcepciones++;
        System.out.println(ex);
        System.out.println("N° excepciones: " +
                                contadorExcepciones);

}
```

Advices -Durante(Around)

Se ejecutan en lugar del punto de enlace capturado. Mediante este tipo de avisos se puede;

- Reemplazar la ejecución original del punto de enlace por otra alternativa.
- Ignorar la ejecución del punto de enlace.
- Cambiar el contexto sobre el que se ejecuta el punto de enlace.

Advices -Durante(Around)

Dado que los advices around sustituyen al punto de enlace capturado, deberán devolver un valor de un tipo comparable al tipo compatible al tipo de retorno del punto de enlace reemplazado.

Advices -Durante(Around)

```
Object around(): call( int Punto.get*()){
        Integer i  = (Integer) proceed();
        System.out.println(i);
        return i;
}
```

Advices -Durante(Around)

Mediante un aviso around es posible realizar comprobación de parámetros de una forma más elegante, sin necesidad de utilizar excepciones como pasaba con los avisos before:

```
void around(int cantidad) : call(void Cuenta.reintegro(int)) &&
                            args(cantidad)
{
      if(cantidad >= MIN_REINTEGRO){
            proceed(cantidad);
      }
}
```

GRACIAS