Programación Orientada a Aspectos

Por Juan Nicolás Nobza, Felipe Pieschacon, Tom Erick Perez, David León

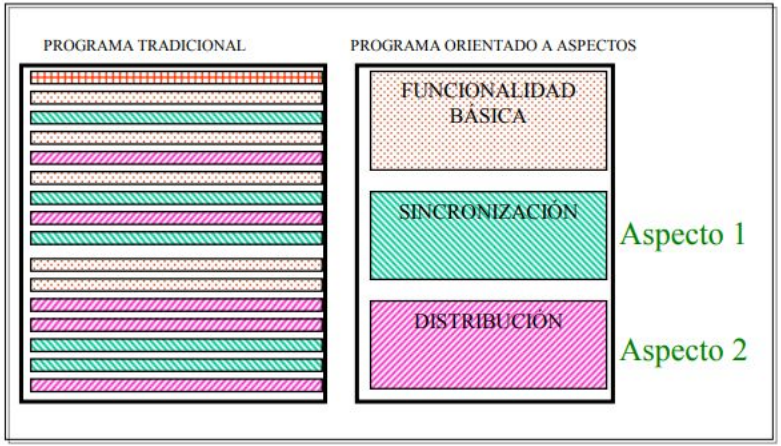
Contenidos

- Historia
- Conceptos Básicos
- Modelos Join Point
- Implementación
- Comparación con otros paradigmas
- Problemas al adoptar el paradigma
- Lenguajes de Aspectos de Propósito General vs. Dominio Específico
- Lenguajes de programación con implementación de POA
- Ejemplos AspectJ
- Referencias

Historia

¿Qué es POA?

- La Programación Orientada a Aspectos es un paradigma de programación cuya intención es permitir una adecuada modularización y posibilitar una mejor separación de conceptos.
- La programación Orientada a Aspectos basa su filosofía en tratar las obligaciones transversales de nuestros programas como módulos separados (aspectos).



Ahora sí, un poco de historia sobre la POA

¿Dónde empezar? ¡Por el principio!

- Datos entremezclados
- Funciones entremezcladas
- Estructura de flujo compleja e incomprensible.
- Un montón de hilos intrincados y anudados.
- Se relaciona con el estilo de programación de lenguajes básicos y antiguos, donde el flujo se controla por sentencias muy primitivas como GOTO.

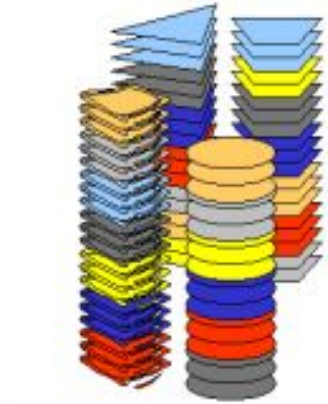
Sofware = Datos (formas)
+ Funciones (colores)

Segunda y Tercera generación al rescate

- En conjunto conviven distintos datos, pero la descomposición del sistema se hace en base a las funciones.
- Los datos quedan esparcidos por todos los conjuntos.
- Existe una agrupación que facilita la comprensión del código.

(¿)La solución(?)
Descomposición en Objetos

- Ahora existe el enfoque en los datos, ya que la agrupación está basada en ellos.
- Las funcionalidades, pese a estar agrupadas, se encuentran dispersas por los diferentes conjuntos.
- Los conjuntos representan objetos que relacionamos con la vida real, facilitando su comprensión para el programador.
- Se solucionan la mayoría de problemas de las generaciones pasadas, pero surgen problemas de seguridad, optimización, performance, entre otros.

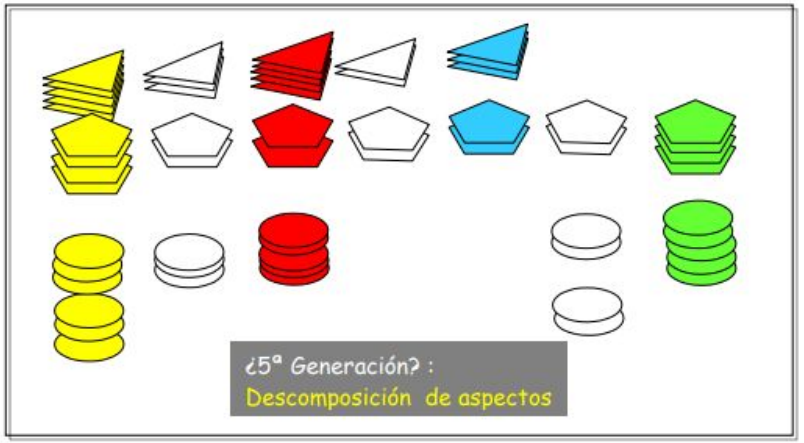


4ª Generación:
Descomposición en objetos

¿Y entonces? ¿Cuál es la solución?

¡La Descomposición de Aspectos!

- Los conjuntos se construyen teniendo en cuenta, tanto las funciones como los datos.
- Las distintas funcionalidades están relacionadas.
- ¿Complicar más la Programación Orientada a Objetos? No en realidad.

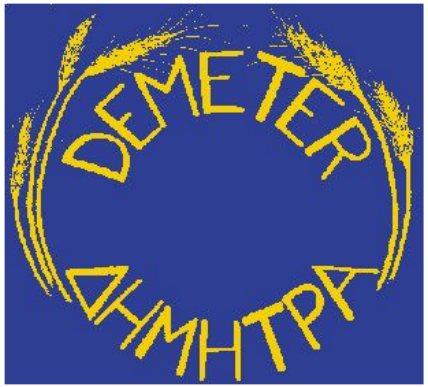


¿Quién se inventó todo esto?

- El concepto fue introducido por Gregor Kiczales. (El hombre de la imagen)
- Junto a unos colegas en Xerox PARC desarrollaron AspectJ, una extensión para Java.
- La definición actual de aspecto, y la más adecuada fue la dicha por el profesor Gregor Kiczales.

Pero hubo alguien antes, varios...
El grupo Demeter

- El grupo Demeter sin saberlo estaba utilizando una instancia temprana de lo que más adelante se conocería como Programación Orientada a Aspectos, conocida como la programación adaptativa.
- Los programas se dividían en varios bloques de cortes. Se separaban la representación de los objetos del sistema de cortes. Luego se añadían comportamientos de estructuras y estructuras de clases como bloques constructores de corte.



Aspect Coupling Table

Aspects	Coupling to other aspects		
		with AP	without AP
grammar	visitors	low	moderate
traversals	grammar	low	high
	visitors	low	high
object descriptions	grammar	low	high

Ahora, Cristina Lopes

- Propuso la sincronización y la invocación remota como nuevos bloques.
- Es parte del grupo Demeter.
- Antes de ser profesora, trabajó como científica investigadora en Xerox PARC.
- Mientras estuvo en PARC, era conocida como la fundadora del grupo que desarrolló la Programación Orientada en Aspectos.

Karl Lieberherr

- Científico de la computación.
- A mediados de los 1980s empieza su investigación sobre la Programación Orientada a Objetos, enfocado en los problemas del diseño de software y la modularidad.
- Crea el grupo Demeter, junto a la Ley de Demeter (“Habla solo con tus amigos”, una forma explícita de hablar sobre el control en parejas).
- Crea los sistemas Demeter/Flavors, Demeter/C++, DemeterJ, entre otros

Mira Mezini

- Trabajó junto a Gregor Kiczales en la creación del documento “Aspect-Oriented Programming and Modular Reasoning” donde se habla a cabalidad sobre la Programación Orientada a Aspectos, y la implementación de módulos en el trabajo.
- Actualmente trabaja en la implementación de Programación Orientada a Aspectos en Bases de Datos.

La colaboración entre Cristina Lopes, Karl J. Lieberherr, Gregor Kiczales y su grupo logró que se introdujera el término Programación Orientada a Aspectos.

Conceptos Básicos

Aspecto (Aspect)

"Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa"(G. Kiczales)

De manera más informal podemos decir que los aspectos son la unidad básica de la POA. También pueden verse como los elementos que se diseminan por todo el código y que son difíciles de describir localmente con respecto a otros componentes.


MODELO DE OBJETOS
ASPECTO DE GESTIÓN DE MEMORIA
ASPECTO DE SINCRONIZACIÓN
PROGRAMA
ASPECTO DE DISTRIBUCIÓN
ASPECTO DE GESTIÓN DE ERRORES
...

Punto de Enlace(Join Point)

Una clase especial de interfaz entre los aspectos y los módulos del lenguaje de componentes. Son los lugares del código en los que éste se puede aumentar con comportamientos adicionales. Estos comportamientos se especifican en los aspectos.

Avisos (Advice)

Definen partes de la implementación del aspecto que se ejecutan en puntos bien definidos. Estos puntos pueden venir dados bien por cortes con nombre, bien por cortes anónimos. En la siguiente figura se puede ver un advice declarado de manera anónima.

```
advice (Point p1, int newval): p1 & (void setX (newval)  | void setY(newval) {
   before {System.out.println ("P1:" + p1);
}
```

El cuerpo de un aviso se puede añadir en distintos puntos del código, cada uno de los cuales se define mediante una palabra clave:

- Aviso before.- Se ejecuta justo antes de que lo hagan las acciones asociadas con los eventos del corte.
- Aviso after.- Se ejecuta justo después de que lo hayan hecho las acciones asociadas con los eventos del corte.
- Aviso catch.- Se ejecuta cuando durante la ejecución de las acciones asociadas con los eventos definidos en el corte se ha elevado una excepción del tipo definido en la propia cláusula catch.
- Aviso finally.- Se ejecuta justo después de la ejecución de las acciones asociadas con los eventos del corte, incluso aunque se haya producido una excepción durante la misma.
- Aviso around.- Atrapan la ejecución de los métodos designados por el evento. La acción original asociada con el mismo se puede invocar utilizando thisJoinPoint.runNext().

Cortes (PointCut)

- Capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y señalización y gestión de excepciones. Los cortes no definen acciones, simplemente describen eventos.
- Un corte está formado por una parte izquierda y una parte derecha, separadas ambas por dos puntos. En la parte izquierda se define el nombre del corte y el contexto del corte. La parte derecha define los eventos del corte.

```
pointcut emisores (Point p1, int newval): p1 & (void setX (newval)|void setY(newval);
advice (Point p1, int newval): emisores (p1, newval){
   before {System.out.println ("P1:" + p1);
}
```

Introducciones (Introduction)

Se utilizan para introducir elementos completamente nuevos en las clases dadas. Entre estos elementos podemos añadir:

- Un nuevo método a la clase.
- Un nuevo constructor.
- Un atributo.
- Varios de los elementos anteriores a la vez.
- Varios de los elementos anteriores en varias clases.

Tejedor (Weaving)

El tejedor se encarga de mezclar los diferentes mecanismos de abstracción y composición que aparecen en los lenguajes de aspectos y componentes ayudándose de los puntos de enlace.

Las clases y los aspectos se pueden entrelazar de dos formas distintas: de manera estática o bien de manera dinámica.

Entrelazado estático

El entrelazado estático implica modificar el código fuente de una clase insertando sentencias en estos puntos de enlace. Es decir, que el código del aspecto se introduce en el de la clase. Un ejemplo de este tipo de tejedor es el Tejedor de Aspectos de AspectJ.

La principal ventaja de esta forma de entrelazado es que se evita que el nivel de abstracción que se introduce con la programación orientada a aspectos se derive en un impacto negativo en el rendimiento de la aplicación. Pero, por el contrario, es bastante difícil identificar los aspectos en el código una vez que éste ya se ha tejido.

Entrelazado dinámico

Una pre condición o requisito para que se pueda realizar un entrelazado dinámico es que los aspectos existan de forma explícita tanto en tiempo de compilación como en tiempo de ejecución.

Para conseguir esto, tanto los aspectos como las estructuras entrelazadas se deben modelar como objetos y deben mantenerse en el ejecutable. Dado un interfaz de reflexión, el tejedor es capaz de añadir, adaptar y borrar aspectos de forma dinámica, si así se desea, durante la ejecución.

Este tejedor también tiene en cuenta el orden en el que se entremezclan los aspectos. Esto lo resuelve asignando una prioridad al aspecto. El aspecto que tenga asignado un número menor es el que se teje primero, y por lo tanto, aparecerá antes en la jerarquía de herencia.

El principal inconveniente subyacente bajo este enfoque es el rendimiento y que se utiliza más memoria con la generación de todas estas subclases.

Una de las primeras clasificaciones de las formas de combinar el comportamiento de los componentes y los aspectos fue dada por John Lamping:

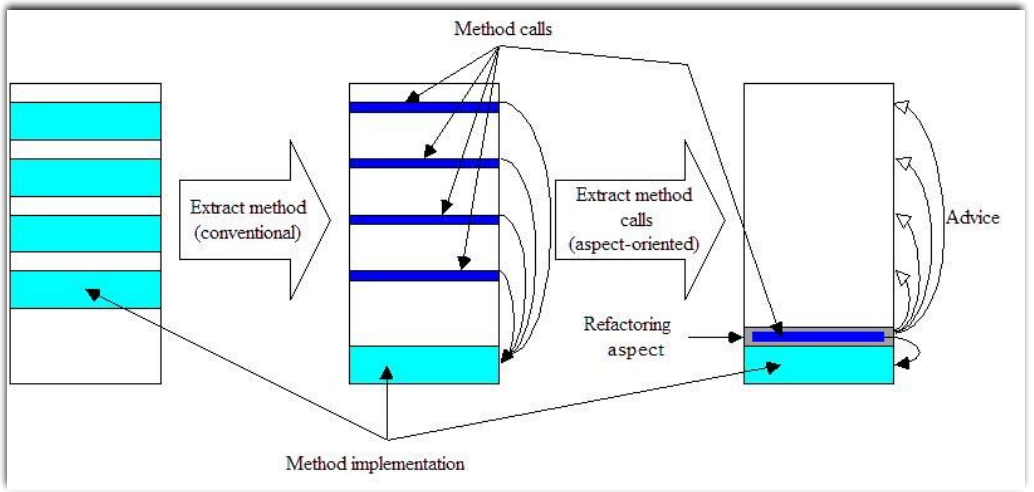
1. Yuxtaposición. Consiste en la intercalación del código de los aspectos en el de los componentes. La estructura del código mezclado quedaría como el código base con el código de los aspectos añadidos en los puntos de enlace. En este caso, el tejedor sería bastante simple.
2. Mezcla. Es lo opuesto a la yuxtaposición, todo el código queda mezclado con una combinación de descripciones de componentes y aspectos.
3. Fusión. En este caso, los puntos de enlace no se tratan de manera independiente, se fusionan varios niveles de componentes y de descripciones de aspectos en una acción simple.

Joint Points y su implementación

Join Point

Para poder hablar de Join Points debemos hablar antes de:

- Pointcuts
- Weaving
- Advice
- Anatomía de un aspecto



Anatomía de un aspecto

```
public aspect AspectoModales
{
    pointcut entregaMensaje():
        call(* ComunicadorMensaje.muestraMensaje(..));

    before(): entregaMensaje()
    {
        S.o.p("Hola! ");
    }
}
```

Anatomía de un aspecto - Advice

```
public aspect AspectoModales
{
    pointcut entregaMensaje():
        call(* ComunicadorMensaje.muestraMensaje(..));

    before(): entregaMensaje()
    {
        S.o.p("Hola! ");
    }
}
```

Anatomía de un aspecto - Pointcut

```
public aspect AspectoModales
{
    pointcut entregaMensaje():
        call(* ComunicadorMensaje.muestraMensaje(..));

    before(): entregaMensaje()
    {
        S.o.p("Hola! ");
    }
}
```

Anatomía de un aspecto - Pointcut

```
pointcut entregaMensaje():
    call(* ComunicadorMensaje.muestraMensaje(..));


public static void main(String[] args)
{
        ComunicadorMensajes cm = new ComunicadorMensajes();
        cm.muestraMensaje("Alan, ¿quieres aprender AspectJ?");
}
```

Anatomía de un aspecto - Pointcut

```
pointcut entregaMensaje():
    execution(* ComunicadorMensaje.muestraMensaje(..));

public class ComunicadorMensajes
{
    public void muestraMensaje(String mensaje)
    {
        S.o.p(mensaje);
    }
}
```

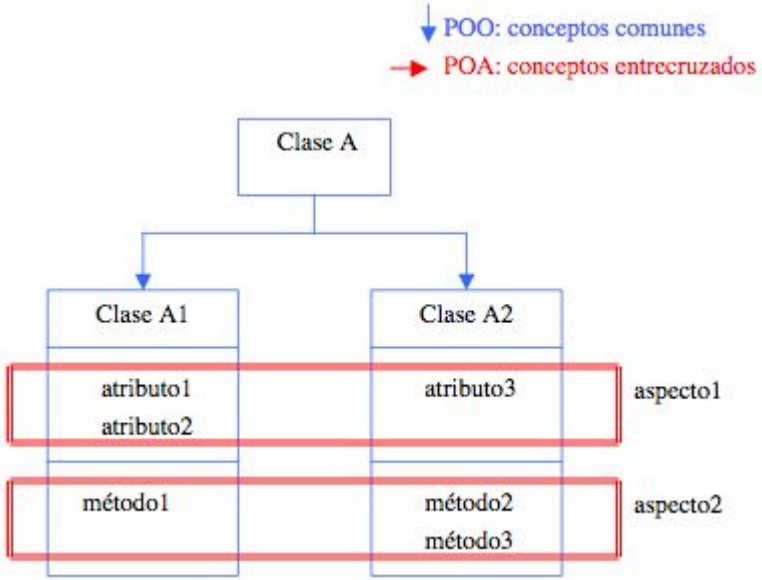
Join Points por tipo

Categoría de Join Point	Sintaxis
Ejecución de Método	execution(FimaMetodo)
Llamada a Método	call(FirmaMetodo)
Ejecución de Constructor	execution(FirmaConstructor)
Llamada a Constructor	call(FirmaConstructor)
Inicialización de Clase	staticinitialization(FirmaTipo)
Lectura de un atributo	get(FirmaAtributo)
Escritura de un atributo	set(FirmaCampo)
Ejecución de manejo de Excepción	handler(FirmaTipo
Inicialización de Objeto	initialization(FirmaConstructor)
Pre-Inicialización de Objetos	preinitialization(FirmaConstructor)
Ejecución del Advice	adviceexecution()

Comparación con otros paradigmas

POA vs POO

- POO sirve para modelar conjuntos de conceptos comunes, POA sirve para modelar conceptos que se entrecruzan.
- Tanto POO como POA crean una estructura por módulos de bajo acoplamiento.
- En POA se modela desde los conceptos que se entrecruzan hacia un concepto principal, en POO es directamente al revés.



Ventajas

- Facilidad para modularizar un programa independientemente de lo extenso o complejo que llegue a ser.
- Facilita el clean code.
- Facilita el trabajo en equipo.
- Por su versatilidad, se puede mezclar con cualquier otro paradigma de programación
- Permite la comunicación entre diversos lenguajes de programación que comparten aspectos.
- Facilita reutilización de código.
- Los sistemas modelados son más flexibles, la separación de conceptos permite agregar, eliminar o modificar aspectos fácilmente.

Problemas al adoptar el paradigma

Problemas de adoptar POA

- Presenta “acciones a distancia”, un antipatrón de diseño de software.
- Dificulta la comprensión del código puesto que algunas funciones llaman tareas que no deberían ser llamadas.
- Aún no es muy claro en qué casos usar POA eficientemente.
- Posibles choques entre el código funcional y el código de aspectos.
- Algunos aspectos pueden llegar a violar el principio de encapsulamiento, entre otros principios de diseño.
- Posibles choques entre aspectos diferentes.
- Posibles choques entre los aspectos y mecanismos de los lenguajes, como la anomalía de la herencia.

Lenguajes de Aspectos de Propósito General vs. Dominio Específico

Lenguajes de aspectos de dominio específico

- No pueden soportar otros aspectos distintos de aquellos para los que fueron diseñados (distribución, coordinación, manejo de errores, ...).
- Normalmente tienen un nivel de abstracción mayor que el lenguaje base.
- Imponen restricciones en la utilización del lenguaje base.
- Fuerzan la separación de funcionalidades.
- Ejemplos: COOL, que trata el aspecto de sincronización, y RIDL, para el aspecto de distribución.

Lenguajes de aspectos de propósito general

- Utilizados con cualquier clase de aspecto, no solamente con aspectos específicos.
- Soportan la definición separada de los aspectos proporcionando unidades de aspectos
- Normalmente tienen el mismo nivel de abstracción que el lenguaje base.
- Permiten la separación del código, pero no garantizan la separación de funcionalidades.
- Ejemplo: AspectJ, que utiliza Java como base, y las instrucciones de los aspectos también se escriben en Java.

Lenguajes de programación con implementación de POA

Algunos lenguajes con implementación interna o de librería externa

.NET Framework languages (C# / VB.NET)	The Cocoa Objective-C frameworks	Groovy	Lua	Python
ActionScript	COBOL	Haskell	Matlab	Racket
Ada	ColdFusion	Java	Perl	Ruby
AutoHotkey	Common Lisp	JavaScript	PHP	UML 2.0
C / C++	Delphi	Logtalk	Prolog	XML

Ejemplos AspectJ

Sincronización de una cola circular con AspectJ

```
aspect ColaCirSincro{
 private int eltosRellenos = 0;

 pointcut insertar(ColaCircular c):
    instanceof (c) && receptions(void Insertar(Object));
 pointcut extraer(ColaCircular c):
    instanceof (c) && receptions (Object Extraer());

 before(ColaCircular c):insertar(c) {
          antesInsertar(c);}

 protected synchronized void antesInsertar
                                    (ColaCircular c){
   while (eltosRellenos == c.getCapacidad()) {
     try { wait(); } catch (InterruptedException ex) {};
   }
}

after(ColaCircular c):insertar(c) { despuesInsertar();}
protected synchronized void despuesInsertar (){
     eltosRellenos++;
     notifyAll();
}

before(ColaCircular c):extraer(c) {antesExtraer();}
protected synchronized void antesExtraer (){
   while (eltosRellenos == 0) {
     try { wait(); } catch (InterruptedException ex) {};
   }

after(ColaCircular c):extraer(c) {
                 despuesExtraer();}

protected synchronized void despuesExtraer (){
     eltosRellenos--;
     notifyAll();
}
```

```
public class ColaCircular
 {
 private Object[] array;
 private int ptrCola = 0, ptrCabeza = 0;


    public ColaCircular (int capacidad)
     {
          array = new Object [capacidad];
     }

    public void Insertar (Object o)
     {
          array[ptrCola] = o;
          ptrCola = (ptrCola + 1) % array.length;

     }

    public Object Extraer ()
     {
      Object obj = array[ptrCabeza];

          array[ptrCabeza] = null;
          ptrCabeza = (ptrCabeza + 1) % array.length;


      return obj;
     }

   public int getCapacidad(){
      return capacidad;
     }

 }
```

Otro ejemplo Java vs AspectJ

```
interface Shape {
  public moveBy(int dx, int dy);
}

class Point implements Shape {
  int x, y; //intentionally package public

  public int getX() { return x; }
  public int getY() { return y; }

  public void setX(int x) {
    this.x = x;
    Display.update();
  }
  public void setY(int y) {
    this.y = y;
    Display.update();
  }

  public void moveBy(int dx, int dy) {
    x += dx; y += dy;
    Display.udpate();
  }
}

class Line implements Shape {
  private Point p1, p2;

  public Point getP1() { return p1; }
  public Point getP2() { return p2; }

  public void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dx; p2.y += dy;
    Display.update
  }
}
```

```
interface Shape {
  public moveBy(int dx, int dy);
}

class Point implements Shape {
  int x, y; //intentionally package public

  public int getX() { return x; }
  public int getY() { return y; }

  public void setX(int x) {
    this.x = x;

  }
  public void setY(int y) {
    this.y = y;

  }

  public void moveBy(int dx, int dy) {
    x += dx; y += dy;  }

  }
}

class Line implements Shape {
  private Point p1, p2;

  public Point getP1() { return p1; }
  public Point getP2() { return p2; }

  public void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dx; p2.y += dy;

  }
}

aspect UpdateSignaling {
  pointcut change():
    execution(void Point.setX(int))
    || execution(void Point.setY(int))
    || execution(void Shape+.moveBy(int, int));

  after() returning: change() {
    Display.update();
  }
}
```

Referencias

- https://codingornot.com/que-es-la-programacion-orientada-a-aspectos-aop
- http://www.angelfire.com/ri2/aspectos/Tesis/tesis.pdf
- https://medium.com/@PabloLeonPsi/una-aproximaci%C3%B3n-a-la-programaci%C3%B3n-orientada-a-aspectos-a62d377ebe79
- https://es.slideshare.net/wfranck/programacin-orientada-a-aspectos-poa
- https://www.researchgate.net/profile/Antonia_Reina_Quintero/publication/253410957_Vision_General_de_la_Programacion_Orientada_a_Aspectos/links/0f3175340507d3f5ac000000/Vision-General-de-la-Programacion-Orientada-a-Aspectos.pdf
- https://www.researchgate.net/publication/4200498_Aspect-oriented_programming_and_modular_reasoning