Una breve introducción a la

Programación Orientada a Aspectos

Seminario Mirian Andrés
17 de enero de 2018

Índice

Programación Orientada a Aspectos

1. **Motivación**
2. POA al rescate
3. Conceptos básicos de la POA
4. Frameworks/Lenguajes

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Cosas en el tintero
8. Mitos y realidades

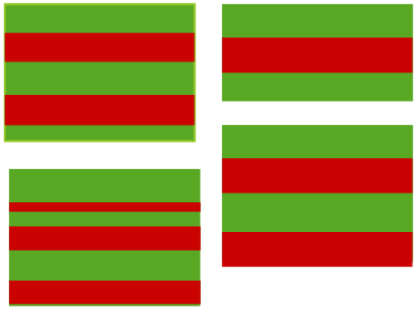
Motivación

- Durante el desarrollo de aplicaciones nos podemos encontrar con **problemas** que **no podemos resolver** de una manera adecuada con las técnicas habituales usadas en PP o POO…
  - Existen determinados aspectos que refieren a **requisitos transversales** compartidos por todos o por parte de las componentes base de la aplicación…
  - …..y que **no pueden encapsularse** dentro de una **única unidad** funcional (por ejemplo: monitorización, manejo de errores, seguridad, etc.).
- …nos vemos **forzados** a **tomar decisiones** de diseño que **repercuten** de manera importante en el **desarrollo de la aplicación**.

Motivación

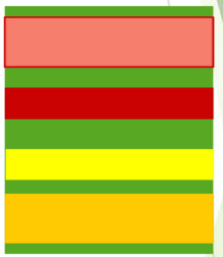
- Como resultado, nos encontramos con situaciones como:

Código **disperso** y **repetido** a través de diferentes componentes



Code scattering (disperso/diseminado)

Superposición de funcionalidades de más de un requisito dentro de un componente



Code tangling (enmarañado/mezclado)

Motivación

```
clase Cuenta{
  …
  <todo lo de cuenta>
  <manejo de errores>
  <control de acceso>
}
```

```
clase Cliente{
  …
  <todo lo de cliente>
  <manejo de errores>
}
```

```
clase Banco{
  …
  <todo lo de banco>
  <manejo de errores>
}
```

Funcionalidad o *intereses* base:

- Cuentas,
- Clientes,
- Bancos…

***Intereses* entrecruzados:**

- **Manejo de errores**
- **Seguridad**

Motivación

Esto afecta al desarrollo de software de diversas maneras:

- **Código repetido.**
  - Mismos fragmentos de código en varios lugares.
- **Díficil razonar sobre dicho código.**
  - No tiene una estructura bien definida.
  - La "figura general" del código enmarañado no es clara.
- **Código difícil de modificar y mantener.**
  - Se tiene que encontrar todo el código involucrado….
  - …y estar seguro para cambiarlo de forma consistente.

Motivación

Consecuencias de esas limitaciones

- Menor productividad
- Reusabilidad disminuida
- Código de calidad empobrecida
- Evolución difícil

Índice

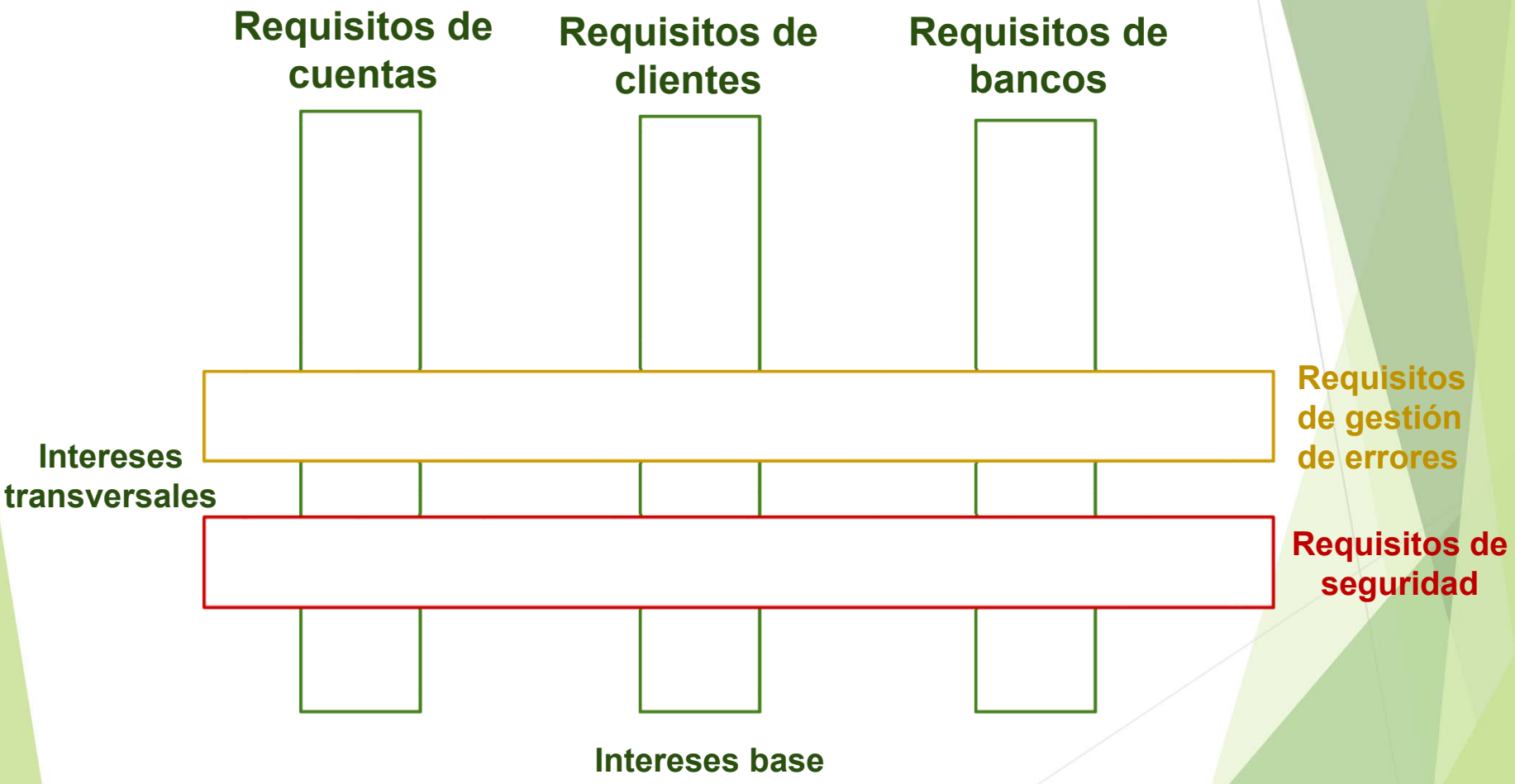
Programación Orientada a Aspectos

1. Motivación
2. **POA al rescate**
3. Conceptos básicos de la POA
4. Frameworks/Lenguajes

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Cosas en el tintero
8. Mitos y realidades

POA al rescate



POA al rescate

- El objetivo de la **Programación Orientada a Aspectos POA** (Aspect-oriented programming o AOP) es proporcionar mecanismos que hacen posible **separar** los **elementos que son transversales** a todo el sistema.
- Gracias a la POA se pueden capturar los diferentes intereses entrecruzados que componen una aplicación en **entidades bien definidas**, **eliminando las dependencias** inherentes entre cada uno de los módulos que la componen.
- Cada interés entrecruzado será encapsulado en una unidad separada → **aspecto**

POA al rescate

Distinción entre

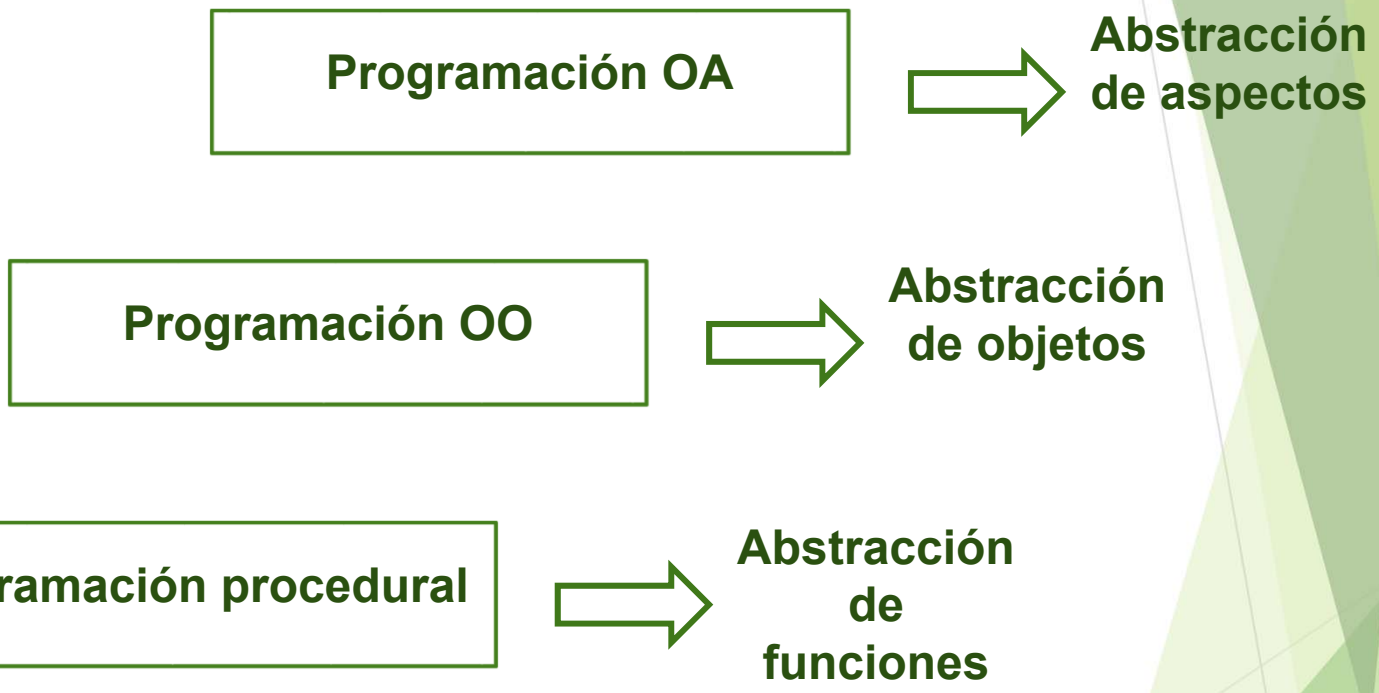
- **Core concern (interés base o componente)**: refiere principalmente a una unidad de la descomposición funcional del sistema y que puede ser encapsulada *limpiamente* (bien localizada, y fácil de acceder y componer)

  Ejemplos: gestión de clientes y de cuentas, transacciones entre bancos.

- **Cross-cutting concern (interés entrecruzado o aspecto)**: refiere a una propiedad o requisito *secundario* que *atraviesa diferentes módulos (no puede ser encapsulada* limpiamente*)*.

  Ejemplos: manejo de errores, seguridad, aspectos de rendimiento, persistencia, administración de transacciones, patrones de diseño, etc.

POA al rescate



La POA NO SE TRATA DE UNA EXTENSIÓN de POO, ni de ningún otro estilo de programación existente anteriormente, sino que se construye sobre las metodologías existentes (PP, POO) **aumentándolas/complementándolas** con conceptos y constructores, con objeto de modularizar los intereses transversales.

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. **Conceptos básicos de la POA**
4. Frameworks/Lenguajes

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Cosas en el tintero
8. Mitos y realidades

Conceptos básicos de la POA

¿Qué necesitamos?

- **Lenguaje base**: un lenguaje para definir la funcionalidad básica (componentes).

- **Uno o varios lenguajes orientados a aspectos**: el lenguaje de aspectos define la forma de los aspectos.

- **Un tejedor de aspectos (o weaver)**: es el encargado de combinar o entretejer dichos lenguajes.

Conceptos básicos de la POA

¿Qué pasos hay que seguir?

1

Se escribe la funcionalidad o interés base (componentes) en el lenguaje base (por ejemplo, en Java).

Se especifica cada interés entrecruzado por separado (aspectos), en el o los lenguajes orientados a aspectos (por ejemplo, en AspectJ)

???

2

Se realiza un proceso de combinación, utilizando el **tejedor de aspectos** (o weaver), que combinará o entretejerá dichos lenguajes, componiendo el **programa final**.

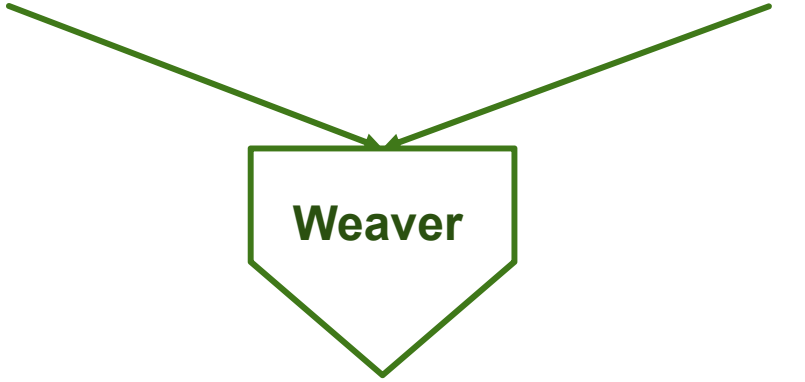
Conceptos básicos de la POA

```
...
  //Hago una cosa
...
```

Componente

```
when(hago una cosa){
   //Hago otra cosa
}
```

Aspecto



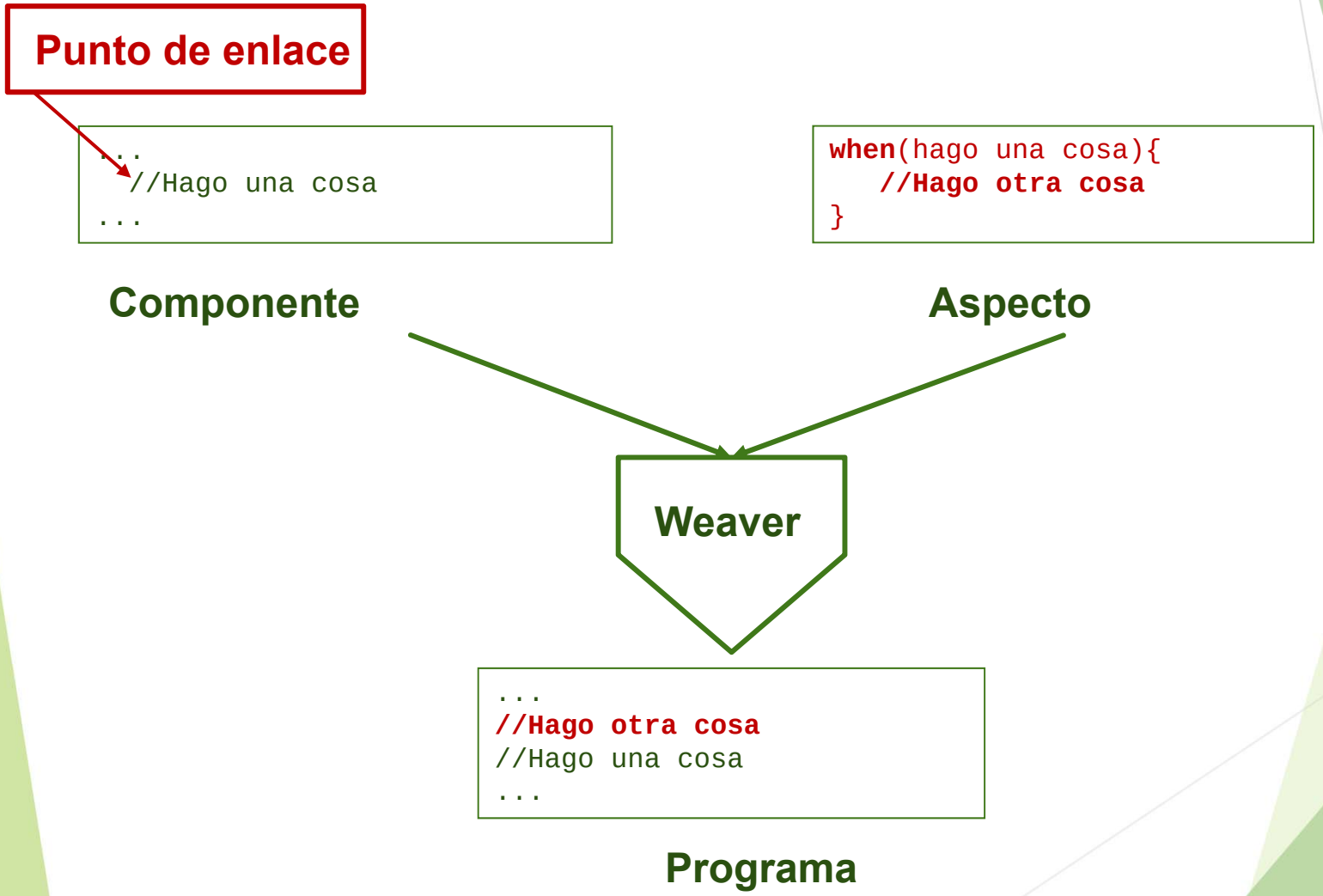

```
...


...
```

Programa

Conceptos básicos de la POA

- Un punto de unión o de enlace **(joinpoint)** es un **punto identificable** durante la ejecución de un programa. Se trata de un lugar dentro del código donde es posible agregar un comportamiento adicional.
  - **Ejemplos**: la invocación a un método, la ejecución de un constructor, la lectura de un valor de una variable, el lanzamiento de una excepción, etc.

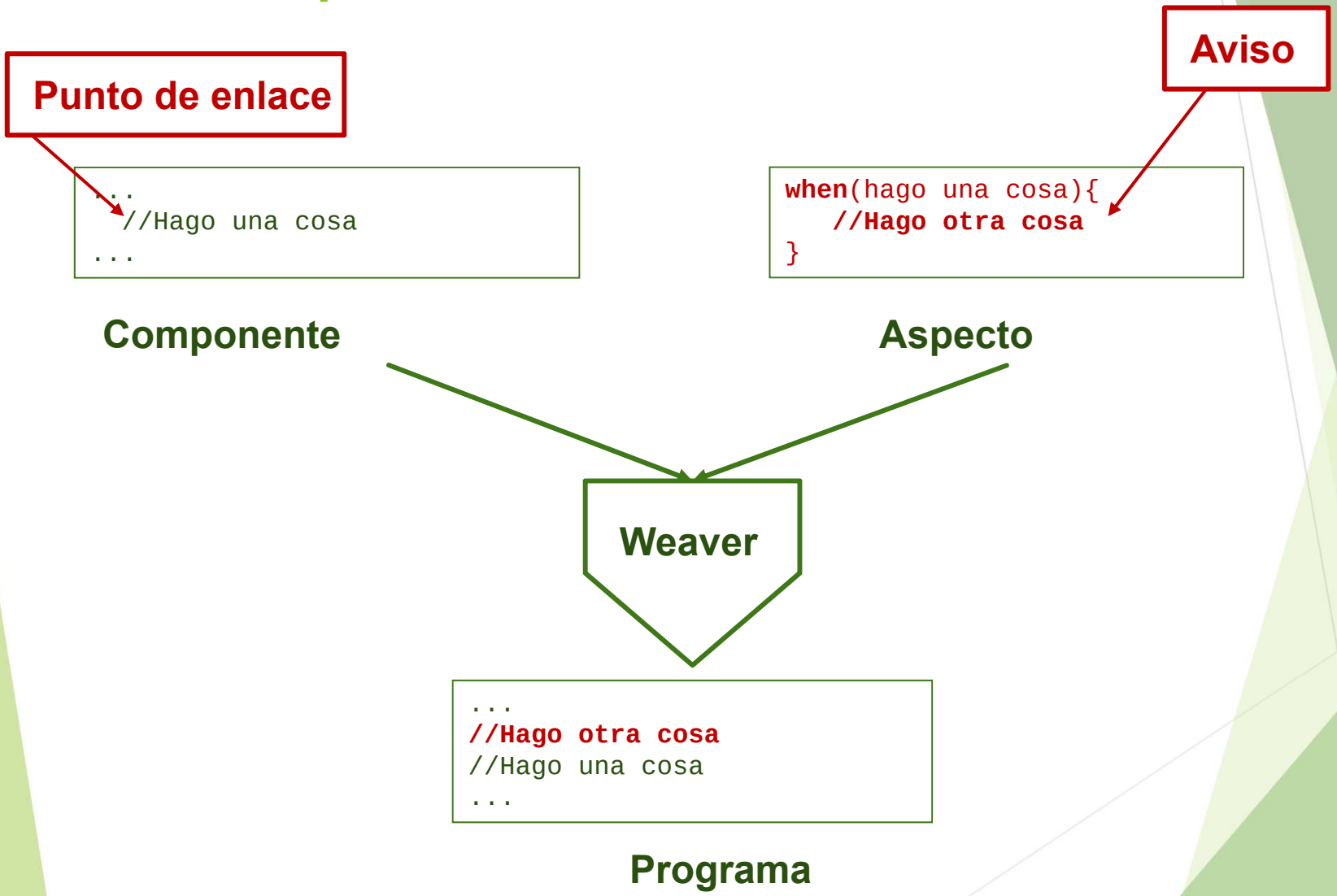
Conceptos básicos de la POA



Conceptos básicos de la POA

- Un aviso (**advice**) es un *fragmento de código que se ejecuta* "en el instante" en el que se detecta un **punto de enlace** (se trata pues del comportamiento adicional a agregar cuando se detecta un punto de enlace).
- Hay **diferentes tipos de avisos** (el comportamiento adicional puede agregarse en diferentes momentos), como por ejemplo:
  - *before* (antes): se ejecuta antes del punto de enlace.
  - *after* (después): se ejecuta después del punto de enlace.
  - *around* (en lugar de): se ejecuta "en lugar del" punto de enlace.

Conceptos básicos de la POA



Conceptos básicos de la POA

- Un corte o punto de corte (**pointcut**) es una colección de puntos de enlace que se utilizan para definir cuándo debe ejecutarse un aviso.
  - En el caso de monitorización o registro de operaciones, el corte serían todos los puntos de enlace del sistema que son de interés para realizar dicho registro (por ejemplo, todas las ejecuciones de los métodos de administración de usuarios).

Conceptos básicos de la POA

- Un aspecto (**aspect**) es la unidad de programación que nos va a permitir ubicar la especificación de los puntos de **corte** (todos los puntos de enlace) junto con sus correspondientes **avisos** para garantizar que esa propiedad va a trabajar de manera coordinada con las clases y objetos del sistema.
- El encargado de la composición final entre componentes y aspectos se llama tejedor de aspectos o **weaver**.
  - Guiado por los puntos de enlace **teje** el **código base** con el **código de los avisos**.
  - **Incorpora** el código de los avisos "en" los puntos de enlace especificados.

Conceptos básicos de la POA

Un ejemplo muy simplificado de POA

Clase Cuenta

```
class Cuenta{
   BigDecimal cantidadActual= getCantidadActual();
   …
   public void retirar(BigDecimal cantidadRetirada){
      cantidadActual = cantidadActual-cantidadRetirada;
   }
   public void depositar(BigDecimal cantidadDepositada){
      cantidadActual = cantidadActual+cantidadDepositada;
   }
}
```

Conceptos básicos de la POA

Un ejemplo muy simplificado de POA

Forma tradicional de registrar la traza

```
class Cuenta{
   BigDecimal cantidadActual= getCantidadActual();
   Logger logger= Logger.getLogger(Cuenta.class);
   …
   public void retirar(BigDecimal cantidadRetirada){
      logger.info("Cantidad retirada:"+ cantidadRetirada);
      cantidadActual = cantidadActual-cantidadRetirada;

   }
   public void depositar(BigDecimal cantidadDepositada){
      logger.info("Cantidad depositada:"+ cantidadDepositada);
      cantidadActual = cantidadActual+cantidadDepositada;
   }
}
```

Conceptos básicos de la POA

Un ejemplo muy simplificado de POA

Registrar la traza siguiendo la POA, sería algo así como:

Clase Cuenta

```
aspect Logging{
   Logger logger= Logger.getLogger(getClass());
   …
   when retirar(cantidad){
      logger.info("Cantidad retirada:"+ cantidad);
   }
   when depositar(cantidad){
      logger.info("Cantidad depositada:"+ cantidad);
   }
}
```

Conceptos básicos de la POA. Tipos de entrelazado

En la POA un programa puede modificarse o extenderse de diversas formas. En función de cómo se modifique un programa, se distinguen dos **tipos de crosscutting o entrelazado**:

- **Static crosscutting o entrelazado estático**

  Los aspectos afectan a la **estructura estática** de la aplicación, permitiendo insertar nuevos atributos, métodos o constructores a clases, interfaces o aspectos (ej. añadir un nuevo atributo a una clase, modificar jerarquías, etc.)

- **Dynamic crosscutting o entrelazado dinámico**

  Se trata del tipo de entrelazado **más utilizado**, en el que los aspectos afectan al **comportamiento** de la aplicación, modificando o añadiendo comportamiento nuevo (ej. añadir cierto comportamiento tras la ejecución de ciertos métodos)

  Los puntos de corte indican el *dónde* y los avisos *qué hacer*.

Conceptos básicos de la POA. Tipos de tejedores

La diferencia entre los tipos de tejedores o weavers radica en el **momento** en el que tiene lugar el proceso de **tejido** (o entrelazado) y **cómo** se lleva a cabo dicho proceso. Hay **dos tipos de weavers**:

- **AOP Estáticos (o compile-time weavers)**

  El proceso de entrelazado constituye otro paso en el proceso de construcción de la aplicación.

- **AOP Dinámicos (o run-time weavers)**

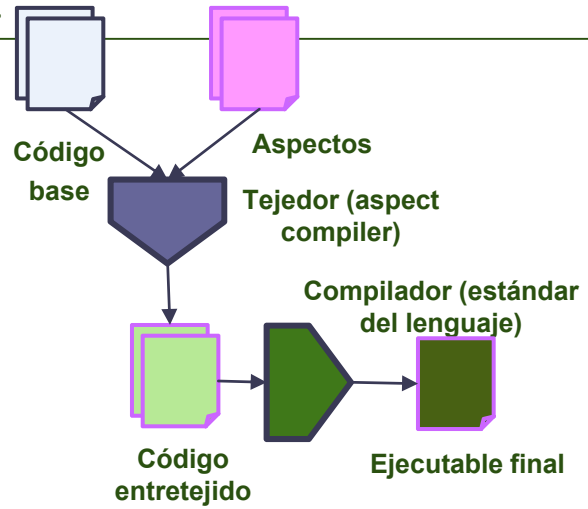
  El proceso de entrelazado se realiza de forma dinámica en tiempo de ejecución.

También existen tejedores que soportan tanto entrelazado estático como dinámico, pero para ello deben seguir una serie de pautas [KLMM97].

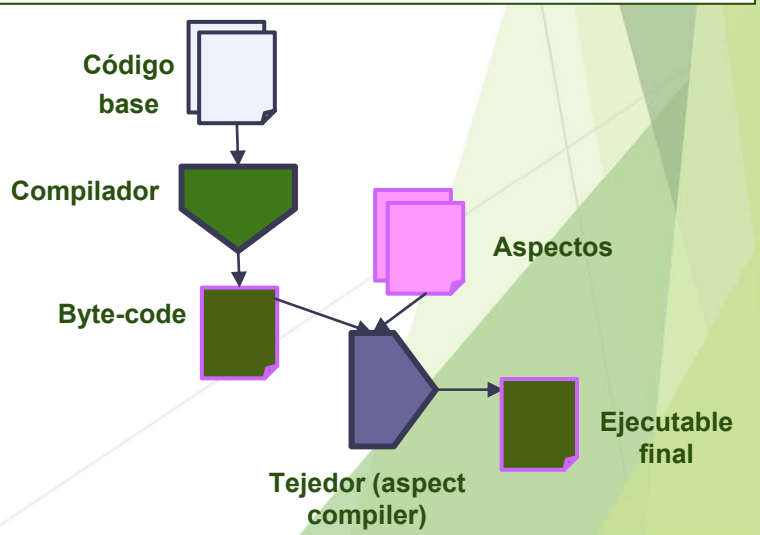
Conceptos básicos de la POA. Tipos de weavers

AOP Estáticos (o compile-time weavers): el proceso de entrelazado constituye otro paso en el proceso de construcción de la aplicación. Hay dos estrategias de esta modalidad.

Se modifica el código fuente escrito en el lenguaje de componentes o código base, integrando el código del aspecto en los puntos de enlace correspondientes del código base. En algunos casos el tejedor tendrá que convertir el código de aspectos al lenguaje base, antes de integrarlo. El nuevo código fuente se pasa al compilador del lenguaje base para generar el ejecutable final.



Otra posibilidad es que el entretejido tenga lugar a nivel de byte-code, es decir, el compilador entreteje los aspectos en los ficheros byte-code de nuestra aplicación, generando los ficheros de salida correspondientes. Esta modalidad es más habitual en lenguajes OO como Java y C#. También llamado **post-compile** o **binary weaving**.



Conceptos básicos de la POA. Tipos de weavers

AOP Dinámicos (o run-time weavers): el proceso de entrelazado se realiza de forma dinámica en tiempo de ejecución.

La **forma** en la que se realiza dicho proceso depende de la **implementación** (implementation-dependent).

- En **Spring AOP**, por ejemplo, se crean proxies para todos los objetos avisados, permitiendo que los *avisos* sean invocados a medida que se vayan requiriendo.
- El weaver **Tejedor AOP/ST** utiliza la herencia para añadir el código específico del aspecto a sus clases.

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. **Frameworks/Lenguajes**

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Cosas en el tintero
8. Mitos y realidades

Frameworks/lenguajes

- **Java:**
  - **AspectJ, Spring Framework (Spring AOP).**
- **C#/.Net:**
  - **AOP.NET, Proyecto LOOM.NET, aspectC#, Proyecto CAMEO, Eos, Aspect.Net, Spring.NET AOP** (extensión para .NET de **Spring AOP**)
- **C/C++:**
  - **AspectC, AspectC++, FeatureC++, Xweaver.**
- **Otros**
  - **PHP** (PHPAspect, Aspect-Oriented PHP , ...)
  - **Python** (Lightweight Python AOP)
  - **JavaScript** (Ajaxpect, jQuery AOP, Aspects, AspectJS,...)
  - **Cocoa** (AspectCocoa)
  - **CommonLisp** (AspectL)

AspectJ

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Frameworks/Lenguajes

AspectJ

5. **Introducción a AspectJ**
6. Elementos de AspectJ
7. Cosas en el tintero
8. Mitos y realidades

Introducción a AspectJ. Historia

- Desarrollado inicialmente por el grupo de Kiczales (Xerox PARC).
- Se lanzó por primera vez en 1998.
- En 2002, el Proyecto AspectJ pasa a la comunidad open-source de Eclipse.
  - AspectJ: https://www.eclipse.org/aspectj/
  - AspectJ Development Tools: https://www.eclipse.org/ajdt/
- **AspectJ= Java + Aspectos**

  (incluyendo los constructores que proporciona su **crosscutting estático** y **dinámico)**.

Introducción a AspectJ. Formas de trabajar

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

@Aspect
public class EjemploAspecto
{ ... }
```

- Mediante anotaciones
  - Disponible a partir de AspectJ 5
  - Para Java 5 o superior
  - Utiliza únicamente el lenguaje Java (los ficheros fuente se compilan con javac guardándose las anotaciones en los ficheros .class. No obstante, dichos .class tendrán que entretejerse usando el weaver de AspectJ)
- Mediante sintaxis adicional 

```
public aspect EjemploAspecto
{ ... }
```

  - Disponible en todas las versiones de AspectJ
  - Para cualquier versión de Java
  - Añade nueva sintaxis (se considera una pequeña extensión de Java) → es necesario compilar los aspectos con el compilador de AspectJ y/o entretejer los aspectos binarios pre-compilados usando el AspectJ binary weaver.

Introducción a AspectJ.
Primera toma de contacto

Pasos a seguir para crear un aspecto:

1. Identificar los **puntos de enlace (JoinPoint)** donde se desea añadir el comportamiento transversal.

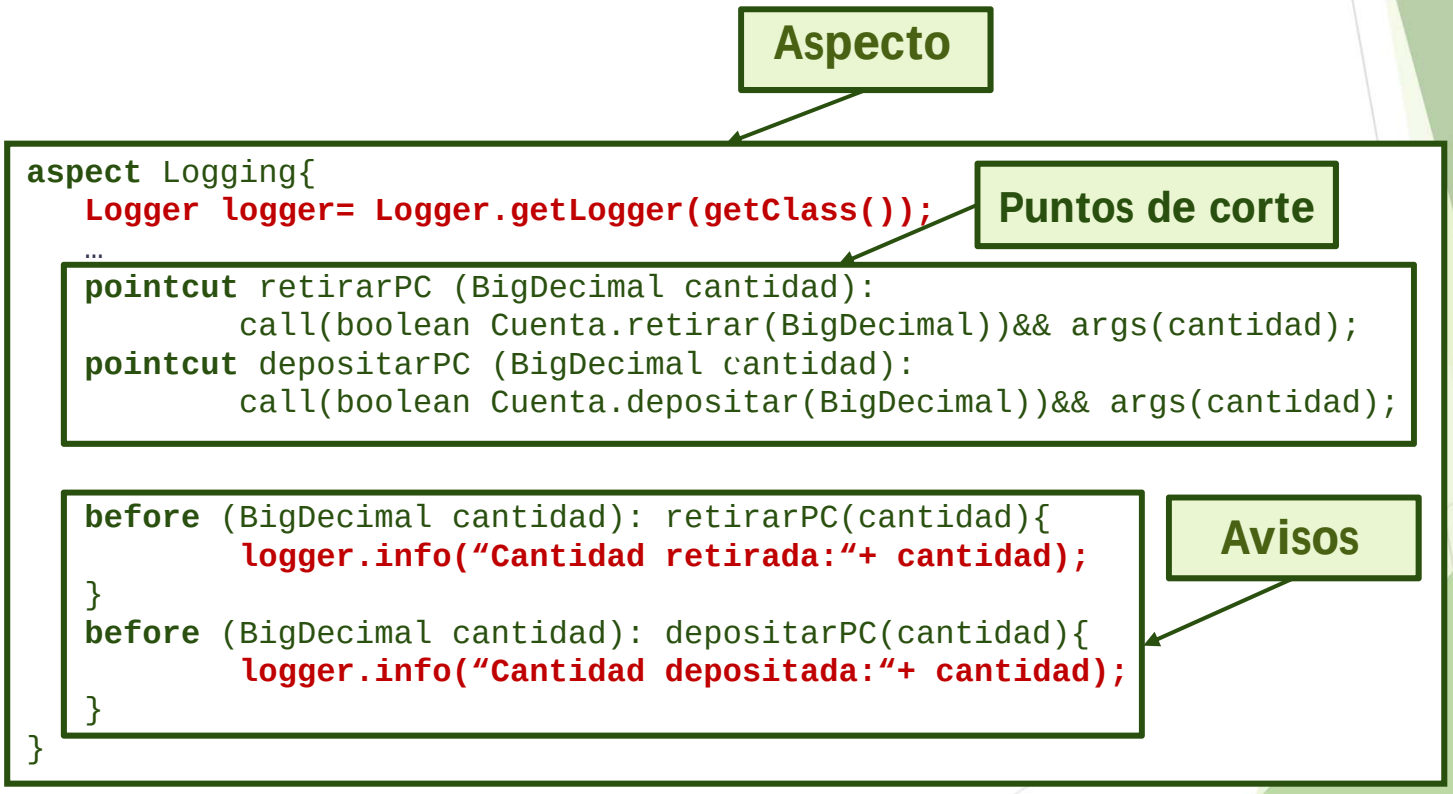
2. Agrupar dichos puntos de enlace con uno o varios **puntos de corte (PointCut)**.

3. Implementar el comportamiento transversal con un **aviso** (**advise**), cuyo cuerpo será ejecutado cuando se alcance alguno de los puntos de enlace que satisfacen el punto de corte definido en el paso anterior.

4. Encapsular lo anterior en un **aspecto** (**aspect**), de forma que la implementación de la competencia transversal quede totalmente localizada en una sola entidad.

Introducción a AspectJ.
Una vista rápida

Registrar la traza siguiendo la POA (AspectJ):



Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Frameworks/Lenguajes

AspectJ

5. Introducción a AspectJ
6. **Elementos de AspectJ**
7. Cosas en el tintero
8. Mitos y realidades

Elementos de AspectJ

Los elementos principales de AspectJ (dinámico) son:

- Puntos de enlace o **JoinPoints**
- Puntos de corte o **PointCuts**
- Avisos o **Advices**
- Aspectos o **Aspects**

JoinPoints

Los puntos de enlace soportados por AspectJ son los siguientes (11):

JoinPoint o punto de enlace	Descripción
Llamada a un método	Cuando un **método es invocado** por un objeto (ej. c.setX(10);)
Ejecución de un método	Engloba la **ejecución del cuerpo** de un método.
Llamada a un constructor	Cuando un **constructor es invocado** durante la creación de un nuevo objeto (ej. Clase c = new Clase(5,5);)
Ejecución de un constructor	Engloba la **ejecución del cuerpo de un constructor** durante la creación de un nuevo objeto.
Lectura o acceso a un atributo	Cuando se **lee un atributo** de un objeto dentro de una expresión (ej. return c;)
Escritura de un atributo	Cuando se **asigna un valor** a un atributo de un objeto (ej. c=i;)
captura de una excepción	Ocurre se captura una excepción.
Inicialización de clase	Cuando se **ejecuta el inicializador** estático de una clase específica, en el momento en el que cargador de clases carga dicha clase.
Inicialización de objeto	Ocurre cuando se **crea un objeto**. Comprende desde el retorno del constructor padre hasta el final del primer constructor llamado. En un ejemplo en el que el constructor tiene: super(); this.x = x; this.y = y;, la llamada al constructor padre (super) no forma parte del punto de enlace.
Pre-inicialización de objeto	Ocurre **antes de que el código de inicialización de una clase particular se ejecute**. Comprende desde el primer constructor llamado hasta el comienzo del constructor padre. Se utiliza raramente y suele abarcar las instrucciones que representan los argumentos del constructor padre.
Ejecución de un aviso	Comprende la **ejecución del cuerpo de un aviso**.

PointCuts

- **Objetivo**: identificar un conjunto de puntos de enlace (uno o varios puntos de enlace) y permitir exponer su contexto a los avisos.
- **Tipos de puntos de corte**:
  - De **muy específicos** (identifican un único punto de enlace) a **muy generales** (identifican un gran número de puntos de enlace).
  - **Anónimos** o con **nombre**.
    - Los **anónimos** se definen en el lugar donde se usan (en un aviso o en otro punto de corte) y no pueden ser referenciados en otras partes del código.
    - Los puntos de corte con **nombre** sí se pueden reutilizar.

PointCuts

- **Sintaxis de un punto de corte anónimo**

```
[!] descriptor [ && | || ]
descriptor ::= identificador_descriptor(<signatura>)
```

Ejemplo:

```
call(public * *.* (int))
```

- **Sintaxis de un punto de corte con nombre**

```
<tipo_acceso> [abstract] pointcut <nombre_pointcut>([<parametros>]):
                                                    [!] descriptor [ && | || ] ;
descriptor ::= identificador_descriptor(<signatura>)
```

Ejemplo:

```
pointcut publicIntCall(int i): call(public * *.* (int)) && args(i)
```

PointCuts. (Algunos) tipos de descriptores

- **Basados en métodos/constructores*:** `call(<patrón-método>/<patrón-constructor>)`

  Captura la llamada a un método/constructor.

```
call (public void paquete.clase.metodo(String))
```

- **Relacionados con atributos:** `get(<patrón-atributo>)/ set(<patrón-atributo>)`

  Capturan la lectura o modificación de un atributo.

```
set/get(tipoAtributo paquete.clase.atributo)
```

- **Relacionados con la creación de objetos*:** `initialization (<patrón-constructor>)`

  Captura la creación (inicialización) de un objeto que utilice el constructor indicado.

```
initialization(paquete.clase.new())
```

- **Relacionados con la ejecución de un manejador (solo con avisos `before`):** `handler(<patrón-excepción>)`

  Captura la excepción del tipo indicado por la expresión entre paréntesis. La captura está siempre especificada por una cláusula `catch`.

```
handler(paquete.excepcion)
```

PointCuts. (Algunos) tipos de descriptores

- **Relacionados con la localización de código:** `within(<patrón-tipo>)/ withincode(<patrón-método/constructor>)`

  Capturan puntos de enlace que se localizan en determinados fragmentos de código.

```
within(miAspecto)/within(MiClase+)
```

- **Basados en condiciones:** `if(<expresión booleana>)`

  Capturan puntos de enlace basándose en alguna condición.

```
if(thisJoinPoint.getTarget()
                    instanceof Clase)
```

- **Relacionados con el objeto en ejecución:** `this()/target()`

  Se aplican sobre un objeto.

```
this(paquete.Clase)
target(paquete.Clase)
```

  - `this()`: para referir al **objeto actual** asociado a la ejecución del punto de enlace. Se comprueba si el objeto this de Java es de la clase indicada, y de ser así, se aplica el aviso.
  - `target()`: para referir al **objeto destino** asociado a la ejecución de un punto de enlace. Para comprobar el objeto sobre el que se va a aplicar el aviso.

`args()` se utiliza para exponer los argumentos de un punto de enlace. Por ejemplo para referir (1) a los argumentos pasados a un método o constructor, (2) a la excepción capturada por un punto de enlace `handler`, `(3)` al nuevo valor a asignar a un atributo.

PointCuts

Comodines para identificar los puntos de enlace:

***** En algunos contextos significa **cualquier número de caracteres excepto** el **punto** "." y en otros representa cualquier tipo (clase, interfaz, tipo primitivo o aspecto).

```
com.*.Clase→ todas las clases llamadas "Clase", dentro de los subpaquetes
             del paquete "com"
com.dtsi.Clase.get*(*)→ todos los métodos que comiencen por "get", de la
             clase "com.dtsi.Clase", que tengan un único parámetro.
```

.. Representa **cualquier número de caracteres**, **incluido** el **punto "."**. Cuando se usa para indicar los **parámetros** de un método, significa que el método puede tener un número y tipo de parámetros arbitrario (cero o varios). Referido a **paquetes**, representa el paquete actual y todos sus subpaquetes (directos o indirectos).

```
javax..* → todas las clases de "javax", "javax.swing", "javax.swing.tree", etc.
com.dtsi.Clase.*(int,..)→ todos los métodos de la clase "com.dtsi.Clase", con
             un primer parámetro de tipo int.
```

+ Representa a una **clase** o **interfaz** y **todos** sus **descendientes**, tanto directos como indirectos (clases hijas o que implementan la interfaz). Debe ir siempre seguido de un tipo.

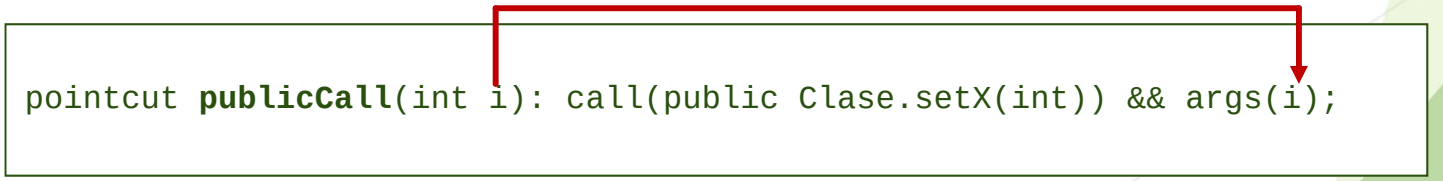
```
com.dtsi.Clase+
```

PointCuts

Paso de información de contexto

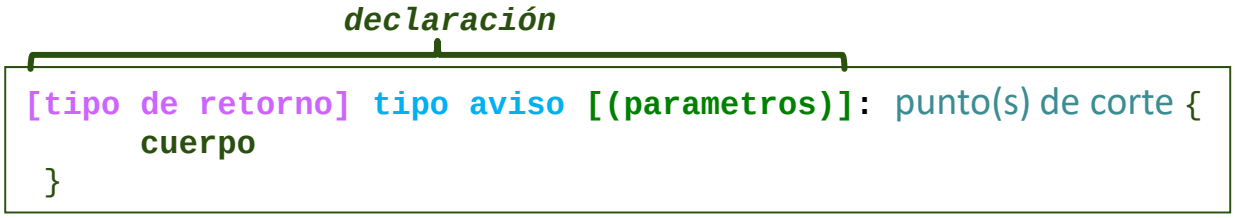
- Todos los puntos de corte tienen un **contexto**, que se puede pasar **posteriormente al aviso.**
- Para capturar la información de contexto, se usan los descriptores ya vistos: `this(), target(), args().`
- También se puede usar la variable `thisJoinPoint` para acceder a información (estática y dinámica) sobre el contexto del punto de enlace actual. Algunos métodos: `getThis(), getTarget(), getArgs(), getSignature().`
- **Regla a seguir a la hora de utilizar parámetros (también en los avisos):**

Los parámetros que aparecen a la izquierda del carácter delimitador dos puntos ":" deben estar ligados a alguna información de contexto a la derecha de los dos puntos.



Avisos

- Indican la acción a llevar a cabo “en el instante” en el que se detecta un punto de enlace.
- Un aviso se podría dividir en tres partes: *declaración*, *punto de corte* y *cuerpo*.



Tipo de aviso: hay tres tipos `before`, `after` (`after`, `after returning`, `after throwing`) y `around`.

El tipo de retorno: solamente se utiliza para los avisos de tipo `around`.

Parámetros de un aviso: permiten exponer información de contexto para que sea accesible desde el cuerpo del aviso.

Punto de corte: cuyos parámetros, si los tiene, deberán tener relación con los del aviso.

Avisos

- Parámetros de un aviso. Los parámetros de un aviso permiten **exponer información de contexto** para que sea accesible desde el cuerpo de un aviso. Para ligar estos parámetros con la **información de contexto** se utilizan los **descriptores** de puntos de corte (`args`, `this`, `target`) y **cláusulas** específicas de los avisos `after` (`returning` y `throwing`), que se explican más adelante.

Recordatorio: regla de los parámetros a izquierda y derecha.

```
before(int i): call(public Clase.setX(int)) && args(i){
     System.out.println("Trazando: Entrada a " + thisJoinPoint.getSignature()+
                                          "con valor del parámetro"+ i);
}
```

```
pointcut publicCall(int i): call(public Clase.setX(int)) && args(i);

before(int i):publicCall(i){
     System.out.println("Trazando: entrada a " + thisJoinPoint.getSignature() +
                                          "con valor del parámetro"+ i);
}
```

- Puntos de corte. Los avisos pueden utilizar puntos de corte anónimos o con nombre.

Avisos

- **Before**
  - Es el aviso más sencillo e indica que el cuerpo del aviso se ejecuta **antes** del punto de enlace capturado.
  - **Si se produce una excepción durante la ejecución del aviso, el punto de enlace capturado no se ejecutaría**.
  - Suelen utilizarse para comprobar valores de parámetros o para auditorías.
- **After**
  - El cuerpo del aviso se ejecuta **después** del punto de enlace capturado.
  - Hay **tres tipos** de avisos `after`:
    - `After`: el aviso se ejecutará **siempre**, sin importar si el punto de enlace finalizó normalmente o con el lanzamiento de alguna excepción.
    - `After returning`: el aviso **sólo se ejecutará** si el punto de enlace termina normalmente, pudiendo acceder al valor de retorno devuelto por el punto de enlace.
    - `After throwing`: el aviso **sólo se ejecutará** si el punto de enlace finaliza con el lanzamiento de una excepción, pudiendo acceder a la excepción lanzada.

Avisos. Ejemplos

Before

```
before (BigDecimal cantidad): call(boolean Cuenta.retirar(BigDecimal))
                                                  && args(cantidad){
   if(cantidad < Cuenta.retiradaMin){
      throw new IllegalArgumentException("La cantidad a retirar" +
            cantidad+ " es inferior a la cantidad permitida."); }
   else
      logger.info("cantidad retirada:"+ cantidad);
}
```

```
before (ClaseEjemplo ce) : call(* com.clases..*(..)) && this(ce) {
     System.out.println("Llamada desde ClaseEjemplo"+ ce);
}
```

After

```
after(): call(Cuenta+.new(..)) {
     System.out.println("Se ha creado un cuenta");
}
```

After returning

```
after() returning (Cuenta c) : call(Cuenta+.new(..)) {
     System.out.println("La cuenta creada es: " + c);
}
```

After throwing

```
after() throwing (Exception ex): call(public * *.*(..)){
     contadorExcepcion++;
     System.out.println(ex);
     System.out.println("Nº de excepciones: " + contadorExcepcion);
}
```

Avisos

- **Around**
  - Son el tipo de avisos **más complejos** y **potentes**.
  - Se ejecutan en **lugar del punto de enlace** capturado (ni antes ni después).
  - Nos permiten realizar tareas como: (1) **continuar la ejecución original** del punto de enlace, **ignorar** la ejecución del punto de enlace (por ejemplo cuando los parámetros de un método son ilegales), o **reemplazar** la ejecución original del punto de enlace por otra alternativa, (2) causar la **ejecución** del punto de enlace **múltiples veces**, (3) **cambiar el contexto sobre el que se ejecuta el punto de enlace**: los argumentos, el objeto actual (`this`) o el objeto destino (`target`) de una llamada.
  - Uso del método `proceed` para ejecutar el punto de enlace original dentro del cuerpo del aviso. Este método debe tener la misma **signatura** que el **aviso** (toma como parámetros, parámetros del mismo tipo que los definidos en la lista de parámetros del aviso, y devuelve un valor del tipo especificado en el aviso).
    - Puesto que los avisos `around` sustituyen al punto de enlace capturado, tendrán que **devolver un valor de un tipo compatible** al tipo de retorno del punto de enlace remplazado.

Avisos. Ejemplos

Around

```
boolean around (BigDecimal cantidad):
            call(boolean Cuenta.retirar(BigDecimal))&& args(cantidad){
           if(cantidad < Cuenta.retiradaMin){
                    throw new IllegalArgumentException(“La cantidad a retirar" +
                        cantidad+ " es inferior a la cantidad permitida."); }
           else{
              logger.info(“cantidad retirada:“+ cantidad);
              proceed(cantidad);
            }
 }
```

```
void around(Object msg):
       call(public void print*(String)) && args(msg) {
            SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyyy HH:mm:ss");
            String fecha = sdf.format(new Date());
            proceed("["+fecha+"] - "+msg);
}
```

Avisos

- Un **aspecto** puede declarar **puntos de corte**, **avisos** (y otros descriptores específicos para realizar el static croscutting), y además puede contener sus propios **atributos** y **métodos** como una clase Java normal.

- ¿**Dónde lo definimos**? Se puede definir en varios sitios:

  - Si el aspecto **afecta a distintas partes de la aplicación**→ se suele crear en un **fichero independiente** con extensión **.aj**.

  - Si el aspecto **afecta a una única clase de la aplicación** → se suele crear **como un miembro más de la clase**→ tendrá que ser `static` (se recomienda que también `private`)

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Frameworks/Lenguajes

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. **Cosas en el tintero**
8. Mitos y realidades

Cosas en el tintero

- **Aspectos privilegiados** para acceder a miembros `private` de otra clase o aspecto, o a miembros `protected`/`package` de clases o aspectos de paquetes en los que no se encuentre el aspecto.
- **Extensión de aspectos** (extensión de aspectos abstractos, extensión de clases o implementación de interfaces).
- **Precedencia entre aspectos**
- **Instanciación de aspectos** (`aspectOf`)
- …

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Frameworks/Lenguajes

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Cosas en el tintero
8. **Mitos y realidades**

POA. Mitos y realidades

- El flujo del programa en un sistema basado en POA es difícil de seguir. Verdad
- La POA no proporciona soluciones a problemas no resueltos hasta el momento. Verdad
  - La AOP permite solucionar determinados problemas de una manera más adecuada, con menos esfuerzo y mejorando el mantenimiento.
- La POA promueve la progamación *sloppy* (*descuidada*). Falso
  - La AOP simplemente proporiona nuevas formas de solventar problemas en áreas que no satisfacen la programación procedural y la OO.
- La POA rompe la encapsulación. Verdad, pero solamente de una forma sistemática y controlada.
  - En POO una clase encapsula todo el comportamiento. El entretejido añadido por POA, elimina este tipo de control de la clase.

Una breve introducción a la

Programación Orientada a Aspectos

¿Preguntas?

Diapositivas adicionales

Desarrollo Software Orientado a Aspectos

- **Inicialmente**, la separación entre componentes base y aspectos se realizaba en las **fases de implementación**.
- A medida que el paradigma de programación se ha hecho **más conocido y aceptado** han ido surgiendo propuestas que han trasladado los conceptos AOP a todo el proceso de desarrollo de Software.
  - Aspect Oriented Requirement Engineering (AORE)
  - Aspect Oriented Business Process Management (AOBPM)
  - Aspect Oriented System Architecture
  - Aspect Oriented Modeling and Design
  - Formal Method support

En [Somm11], el capítulo 21 está dedicado explícitamente a la ingeniería de software orientada a aspectos. También en https://www.ou.nl/documents/40554/349790/T07351_01.pdf

Elementos de AspectJ.
Puntos de corte abstractos

Puntos de corte abstractos

Los puntos de corte **con nombre** pueden ser **abstractos**, y definidos sin un cuerpo. **Un punto de corte abstracto solamente puede definirse en un aspecto abstracto**, y serán los subaspectos de dicho aspecto los encargados de implementarlo.

```
abstract aspect A {
    abstract pointcut publicCall(int i);
}
```

En ese caso, el aspecto que extienda al punto de corte podría sobrescribirlo.

```
aspect B extends A {
      pointcut publicCall(int i): call(public Punto.setPunto(int)) &&
                                                              args(i);
}
```

Un punto de corte **no puede sobrecargarse** (no puede haber dos puntos de corte con el mismo nombre dentro de la declaración de la misma clase o aspecto).

AspectJ. Aspectos privilegiados

Dentro de un aspecto, las reglas de acceso a miembros de una clase o un aspecto son las mismas que en Java.

- Desde el código de un aspecto no podemos acceder a miembros `private` de otra clase o aspecto, ni a miembros `protected`/`package` a no ser que el aspecto pertenezca al mismo paquete que la clase o aspecto del miembro al que intentamos acceder. Si intentamos acceder→ **error de compilación**.

Si en el cuerpo de un aviso (o en una declaración de métodos inter-tipo) deseamos acceder a algún miembro privado o protegido, tendremos que declarar el aspecto como `privileged`.

- De esa forma tendremos acceso a cualquier miembro de cualquier clase o aspecto.

No debemos abusar de los aspectos privilegiados, ya que violan el principio de encapsulación.

AspectJ. Aspectos privilegiados

Ejemplo:

```
class C {
   private int i = 0;
   void incI(int x) {
         i = i+x;
   }
}

privileged aspect A {
    static final int MAX = 1000;
    before(int x, C c): call(void C.incI(int)) && target(c) && args(x) {
        if (c.i+x > MAX)
               throw new RuntimeException();
        }
}
```

Si el aspecto `A` no se hubiera declarado como `privileged`, la referencia `c.i` hubiera dado error de compilación.

AspectJ. Precedencia de aspectos (Static)

- ¿Qué hacemos cuando dos avisos definidos en aspectos diferentes afectan al mismo punto de enlace?¿Y si nos interesa establecer una relación de precedencia entre los avisos, para que uno se ejecute antes que otro?
  - Declaraciones de dependencia: `declare precedence`
- Nos permiten indicar que todos los avisos de un aspecto deben tener preferencia a la hora de ejecutarse con respecto a los de otros aspectos, con respecto a un punto de enlace.
- En el listado **no** se puede incluir un aspecto más de una vez.
- Se pueden usar patrones de tipo para especificar la lista de precedencia.
  - En la lista, se puede utilizar el comodín "*" en solitario, representando a cualquier aspecto no listado (se puede utilizar para situar cualquier aspecto no listado **antes**, **entre** o **después** de los aspectos que aparecen en la lista, en términos de precedencia)

AspectJ. Precedencia de aspectos (Static)

- Si un aspecto extiende a otro, todos los avisos del subaspecto tienen mayor prioridad que los avisos del superaspecto.
- Si no se indica la precedencia con el `declare precedence`, ni se da la situación anterior, la relación de precedencia queda indefinida.
  - ***Ejemplo 1***: `declare precedence : Security, Logging, * ;`

    Para cada punto de enlace, los avisos de `Security` tienen precedencia respecto a los de `Logging`, el cual tiene precedencia sobre cualquier otro aviso.
  - ***Ejemplo 2***: `declare precedence : Security, Logging, L* ;`

    Daría error de compilación porque el aspecto `Logging` aparecería dos veces.
  - ***Ejemplo*** 3: `declare precedence : Security+, * ;`

    Les damos mayor precedencia a los aspectos que extienden el aspecto `Security`.

Desarrollo de aplicaciones AspectJ

- Los tipos de weavers que soporta AspectJ son estáticos.
  - Compile-time weaving

    El programa base está en código fuente. El entretejido tiene lugar cuando se compila el código fuente.

  - Post-compile or binary weaving

    El programa base se encuentra en ficheros .class o JARs.

  - Load-time weaving

    El entretejido se aplaza hasta que las clases son cargadas.