

Tema 2

Una breve introducción a la Programación Orientada a Aspectos



Bibliografía

- ▶ Gregor Kiczales; John Lamping; Anurag Mendhekar; Chris Maeda; Cristina Lopes; Jean-Marc Loingtier; John Irwin. *Aspect-oriented programming*. Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS. 1241. pp. 220–242, 1997.
- ▶ Ian Sommerville, Capítulo 21 de *Ingeniería del Software (9ª edición)*. Pearson (2009). Disponible en <http://www.ingebook.com> (acceso desde la UR). También (en inglés) en https://www.ou.nl/documents/40554/349790/T07351_01.pdf
- ▶ Robert E. Filman ... [et al.], *Aspect-oriented software development*. Addison-Wesley, cop. 2005. Referencia: 1E.14 05 FIL.
- ▶ Siobhán Clarke, Elisa Banniassad, *Aspect-oriented analysis and design: the theme approach*, Addison-Wesley, c2005. Referencia: Planta primera, 1E.14 05 CLA.
- ▶ Ramnivas Laddad, *AspectJ in action: enterprise AOP with Spring applications (2nd edition)* Manning, 2010. Referencia: 1E.18 JAV 10.
- ▶ Gradecki, Joseph D., Lesiecki, Nicholas, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Indianápolis (Indiana): Wiley, c2003. Referencia: 1E.7 ASJ 03
- ▶ *Eclipse AspectJ: Aspect-Oriented programming with AspectJ and the Eclipse AspectJ development tools*, de Adrian Colyer et al. Upper Saddle River (New Jersey), Addison Wesley (2005). Referencia: Depósito 1, X-89493.
- ▶ *Eclipse AspectJ*: <https://www.eclipse.org/aspectj/>
- ▶ *The AspectJTM Programming Guide*, the AspectJ Team, Xerox Parc Corporation. Disponible en: <https://www.eclipse.org/aspectj/doc/next/progguide/>

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

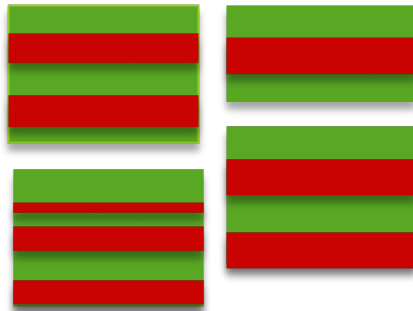
Motivación

- ▶ 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, gestión o manejo de errores, seguridad, etc.).
- ▶ ...y nos vemos **forzados a tomar decisiones** de diseño que **repercuten** de manera importante en el **desarrollo de la aplicación**.

Motivación

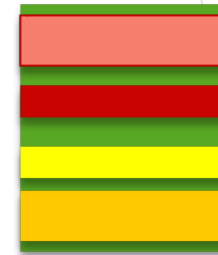
- Como resultado de lo anterior, nos encontramos con situaciones como:

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

► Por ejemplo:

```
class Cuenta{  
    ...  
    <todo lo de cuenta>  
    <gestión de errores>  
    <control de acceso>  
}
```

```
class Cliente{  
    ...  
    <todo lo de cliente>  
    <gestión de errores>  
    <control de acceso>  
}
```

```
class Banco{  
    ...  
    <todo lo de banco>  
    <gestión de errores>  
    <control de acceso>  
}
```

Funcionalidad o *intereses* base:

- Cuentas,
- Clientes,
- Bancos...

Intereses transversales o entrecruzados:

- **Gestión de errores**
- **Seguridad**

Motivación

Esto afecta de diversas maneras al desarrollo de software:

► **Código repetido y propenso a errores.**

- Mismos fragmentos de código en varios lugares.

► **Difícil 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

Motivación. Historia

- En **1991** el grupo **Demeter** propuso la *programación adaptativa*, la cual se puede considerar como una versión inicial de POA.
- En **1995** dos miembros de ese grupo, **Cristina Lopes** y **Walter Huersch**, publican un informe en el que se identifica el problema de la **separación de intereses** y se proponen algunas técnicas como los filtros composicionales y la programación adaptativa, para encapsular los intereses transversales.
- En ese mismo año se publica la **primera definición** de aspecto: *“un aspecto es una unidad que se define en términos de información parcial de otras unidades”*
- El **grupo Xerox PARC**, dirigido por **Gregor Kiczales**, con las colaboraciones de Cristina Lopes y Karl Lieberherr, han sido los contribuidores más importantes al desarrollo de este nuevo paradigma.
- En **1996** Gregor Kiczales acuñó el término **programación orientada a aspectos** (aspect-oriented programming), estableció el marco de la POA y proporcionó una definición más correcta y precisa de un aspecto: *“un aspecto es una unidad modular que se dispersa por la estructura de otras unidades funcionales...”*

Índice

Programación Orientada a Aspectos

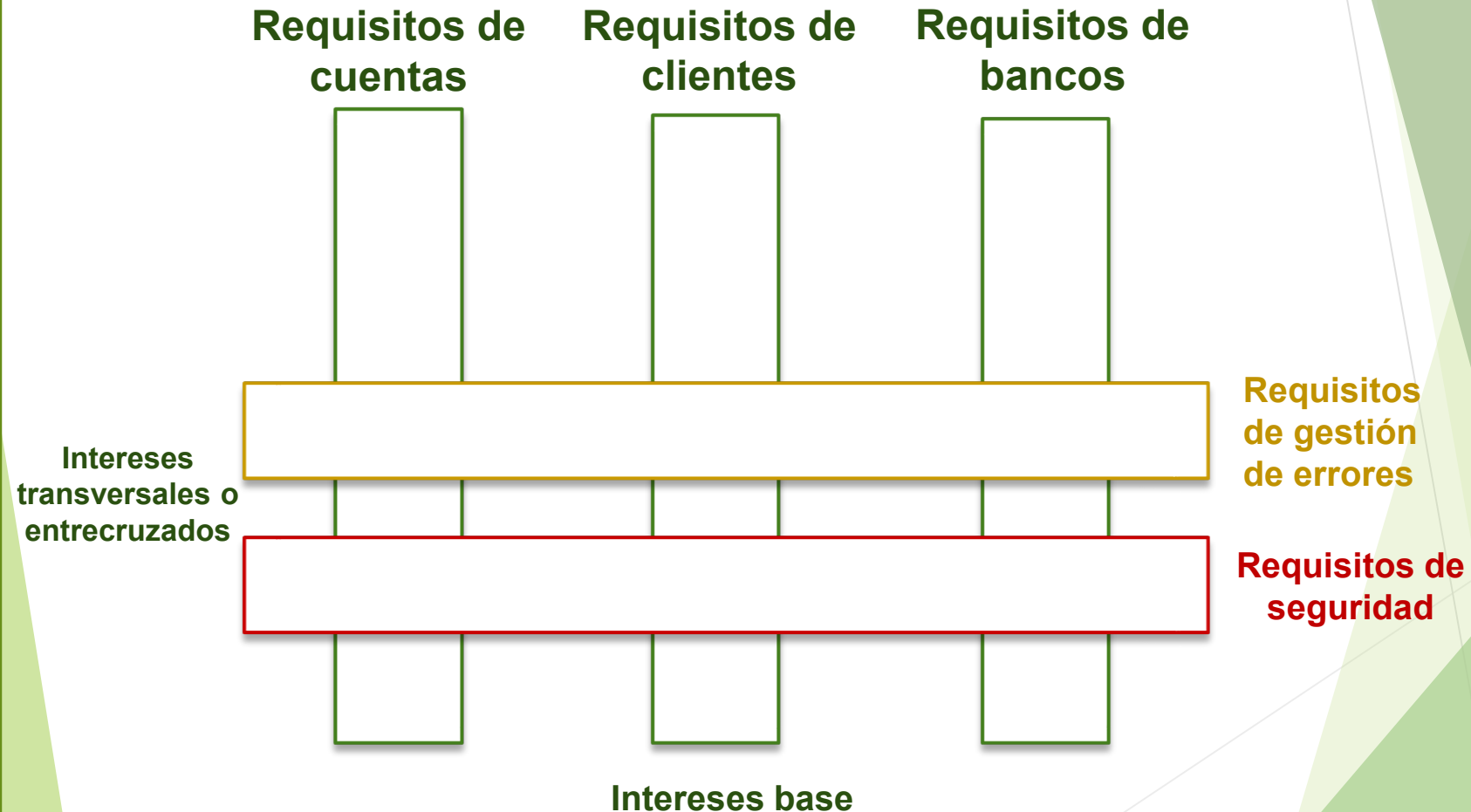
1. Motivación
2. **POA al rescate**
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

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 toda la aplicación.
- ▶ 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 (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: monitorización (logging, tracing), manejo de excepciones, seguridad, aspectos de rendimiento, persistencia, administración de transacciones, patrones de diseño, etc.

POA al rescate

Programación OA

⇒ Abstracción
de aspectos

Programación OO

⇒ Abstracción
de objetos

Programación procedural

⇒ Abstracción
de funciones

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. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

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).

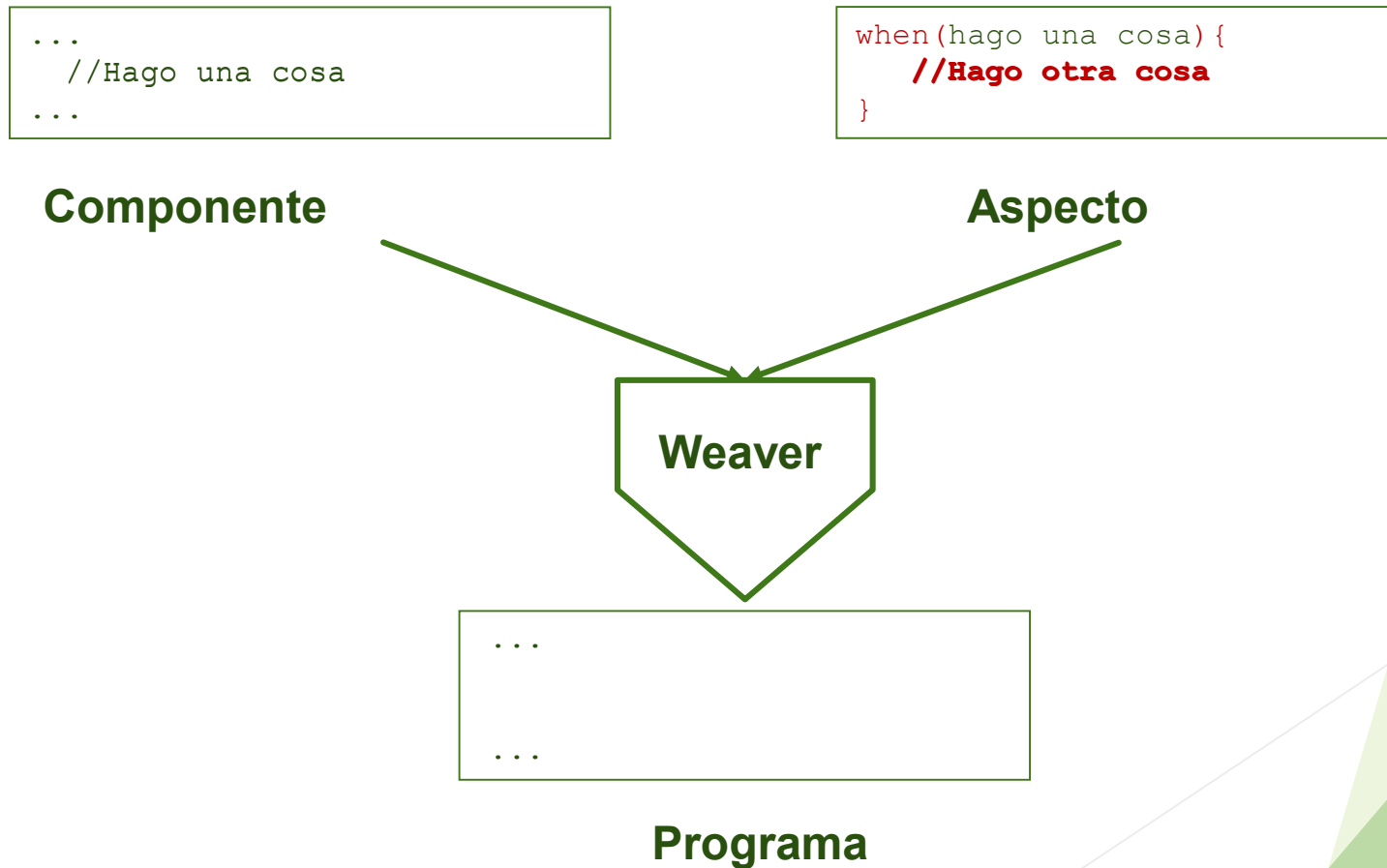
Se especifica cada interés entrecruzado por separado (**aspectos**), en el o los **lenguajes orientados a aspectos** (por ejemplo, en AspectJ)

???

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



Conceptos básicos de la POA

- ▶ La implementación de un aspecto está ligada a determinados puntos en la implementación de la aplicación. Dichos puntos corresponden a posiciones dentro de la ejecución dinámica del programa.
- ▶ 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

Punto de enlace

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

Componente

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

Aspecto

Weaver

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

Programa

Conceptos básicos de la POA

- ▶ Un **aviso** (**advice**) es un fragmento de código que se debe ejecutar “en el instante” (*antes*, *después* o, incluso, *en lugar de*) 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** (en función del momento en que se agrega el comportamiento adicional), como por ejemplo:
 - *before*: se ejecuta *antes* del punto de enlace.
 - *after*: se ejecuta *después* del punto de enlace.
 - *around*: se ejecuta *en lugar* (“en sustitución”) del punto de enlace.

Conceptos básicos de la POA

Punto de enlace

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

Aviso

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

Componente

Aspecto

Weaver

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

Programa

After

(en este ejemplo)

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.
 - *Por ejemplo*, 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).
- ▶ Durante la **ejecución** del sistema, un punto de enlace podría o no ser seleccionado por el punto de corte, en función de si en efecto el punto de enlace se ejecuta o no
 - *Por ejemplo*, si el punto de corte incluye en su definición varios constructores, y solo se invoca a uno en particular durante la ejecución

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** (con todos los puntos de enlace que capturan) junto con sus correspondientes **avisos** para garantizar que esa *propiedad* va a trabajar de manera coordinada con las componentes (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.
 - Los **puntos de corte** indican el *dónde* y los **avisos** *qué hacer*.

Conceptos básicos de la POA

Un ejemplo muy simplificado de POA

Clase Cuenta “básica”

```
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

Clase Cuenta “con traza” (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 POA, sería algo así como:

Clase Cuenta
“básica”



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

(*) Aquí Logging es un aspecto escrito en algún “posible” lenguaje de aspectos

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. **Tipos de tejedores. Frameworks y Lenguajes**
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

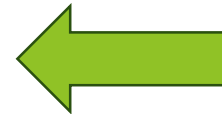
Tipos de entrelazado

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)

Tipos de tejedores

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

► Tejedores estáticos (o compile-time weavers)

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

► Tejedores dinámicos (o run-time weavers)

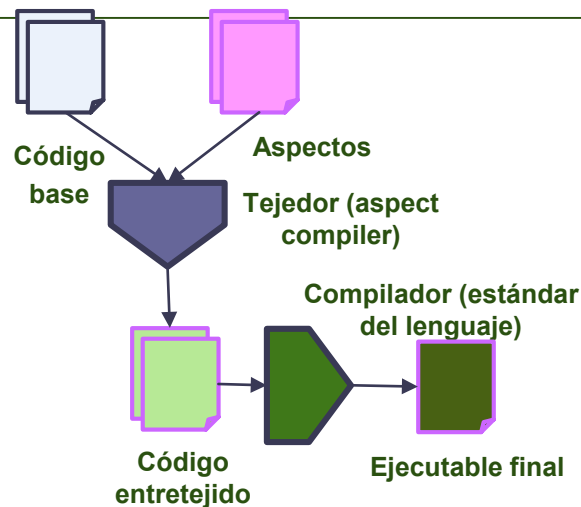
El proceso de entrelazado se realiza de forma dinámica en tiempo de ejecución.

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

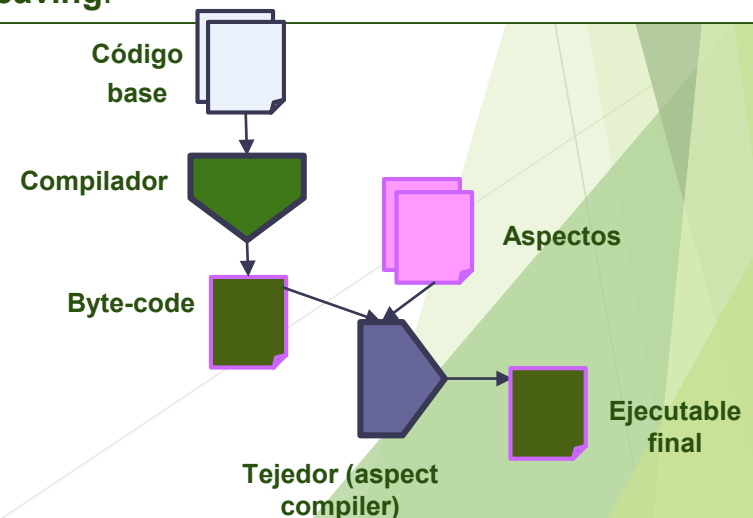
Tipos de tejedores

Tejedores 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**.



Tipos de tejedores

Tejedores estáticos (o compile-time weavers)

Ventajas:

- ▶ Al realizarse en entretelado en tiempo de compilación se evita que derive en un impacto negativo en el rendimiento de la aplicación.
- ▶ Proporcionan un mayor nivel de fiabilidad al realizarse todas las comprobaciones en tiempo de compilación, evitando así posibles errores durante la ejecución.
- ▶ Son más fáciles de implementar y consumen menos recursos que los dinámicos.

Desventajas:

- ▶ Cualquier modificación que deseemos realizar sobre los aspectos (aunque sea simplemente añadir un punto de unión) requiere recompilar toda la aplicación.
- ▶ Es difícil identificar los aspectos en el código ya entretelado.
- ▶ No se pueden añadir o modificar los aspectos dinámicamente.

Ejemplo: el tejedor de aspectos de AspectJ tiene las dos modalidades (AspectJ 1.0.x. genera código fuente, y en AspectJ 1.1.x. el entretelado se realiza a nivel de byte-code (más adelante veremos que tiene otra variante de weaver estático).

Tipos de tejedores

Tejedores 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.

Ventajas

- ▶ Puedes añadir, modificar o eliminar un aspecto dinámicamente (runtime).

Desventajas:

- ▶ Normalmente el rendimiento es peor que el que ofrecen las propuestas estáticas, ya que sobrecarga la ejecución del programa.
- ▶ Ponen en riesgo la seguridad/fiabilidad de la aplicación (se puede eliminar el comportamiento de un aspecto que se pueda necesitar más tarde o eliminar un aspecto en su totalidad y luego hacer mención a una aspecto que ya no existe).

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)

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. **Desarrollo Software Orientado a Aspectos**

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

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
 - Formal Method support
 - Aspect Oriented Modeling and Design

Desarrollo Software Orientado a Aspectos

➤ Ingeniería de requisitos orientada a aspectos

- *Aspect Oriented Requirement Engineering (AORE)*
- También conocida como “early aspects”.
- Se centra en la identificación, especificación y representación de propiedades transversales (crosscutting) a nivel de requisitos.
- Ejemplos de estas propiedades son seguridad, movilidad, disponibilidad y restricciones de tiempo-real.

➤ Gestión de procesos de negocio orientados a aspectos

- *Aspect Oriented Business Process Management (AOBPM)*
- Surge con el objetivo de aliviar la complejidad de la gestión de los procesos de negocio, definiendo de forma separada determinados aspectos (como seguridad y privacidad) que son transversales a los procesos que refieren a aspectos propios del negocio.
- AOBPM define un conjunto de requisitos y un modelo formal, diseñado utilizando Redes de Petri coloreadas. La propuesta se implementa como un Servicio en YAWL basándose en la Arquitectura Orientada a Servicios.

Desarrollo Software Orientado a Aspectos

➤ Arquitectura de sistemas orientada a aspectos

- *Aspect Oriented System Architecture*
- Los aspectos son definidos de forma independiente a los elementos arquitectónicos.
- Existen diferentes tipos de aspectos que varían dependiendo de las características del dominio del sistema software.

➤ Verificación formal

- Uso de métodos formales tanto para definir semánticamente los aspectos, como para analizar y verificar sistemas orientados a aspectos.
- Existen diferentes áreas de aplicación, como por ejemplo model checking para verificar sistemas orientados a aspectos.

Desarrollo Software Orientado a Aspectos

➤ Modelado y diseño orientado a aspectos

- Aspect Oriented Modeling and Design.
- El objetivo del diseño orientado a aspectos contribuye al diseño de un sistema software en lo que refiere a la identificación y modularización de los aspectos entrelazados según las propuestas tradicionales.
- Las propuestas suelen incluir un **proceso** (que toma los requisitos y genera un modelo de diseño que representa los aspectos y sus relaciones) y un **lenguaje** (que permite describir los elementos y relaciones identificadas en el diseño).
- Engloba tareas como el proceso de diseño OA, herramientas de soporte al diseño OA, adopción e integración de diseño OA, valoración/evaluación de diseño OA.
- Propuestas (todavía no adoptadas por OMG) de extensiones de UML para soportar el manejo de aspectos en la etapa de diseño.
 - La mayoría basadas en la definición de profiles (estereotipos), OCL (para JoinPoints)...
 - Otras proponen extensiones del metamodelo de UML (hay una propuesta del 2017 para extender UML 2.5)
 - La mayoría para elementos estructurales (diagramas de clases) y unos pocos para comportamiento (diagramas de interacción).

AspectJ

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. **Introducción a AspectJ**
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

Introducción a AspectJ. Historia

- Extensión del lenguaje Java para soportar POA.
- Desarrollado inicialmente por el grupo de Gregor 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/>

Introducción a AspectJ

Da soporte al **crosscutting** estático y dinámico añadiendo los siguientes constructores:

➤ **Static crosscutting o entrelazado estático:** los aspectos afectan a la estructura estática de la aplicación, como clases, interfaces o aspectos. También se podrían añadir advertencias (warnings) o errores en tiempo de compilación.

Sus **constructores** son: declaraciones inter-tipo, declaraciones de parentesco, declaraciones en tiempo de compilación, declaraciones de prioridad, excepciones suavizadas.

➤ **Dynamic crosscutting o entrelazado dinámico:** los aspectos afectan al comportamiento de la aplicación, añadiendo o modificando comportamiento nuevo.

Sus **constructores** son: puntos de enlace, puntos de corte, avisos.

AspectJ= Java + Aspectos (con los constructores anteriores)



Introducción a AspectJ.

Formas de trabajar

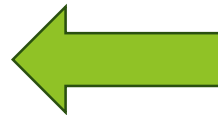
- Mediante anotaciones

- 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 entreteterse usando el weaver de AspectJ)

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

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

- Mediante sintaxis adicional



- 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 en código fuente base (**compile time weaving**), entreteter los aspectos binarios pre-compilados usando el AspectJ **binary weaver** y/o realizar el entretetido en el momento de cargar las clases (**load time weaving**).

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

Introducción a AspectJ.

Primera toma de contacto

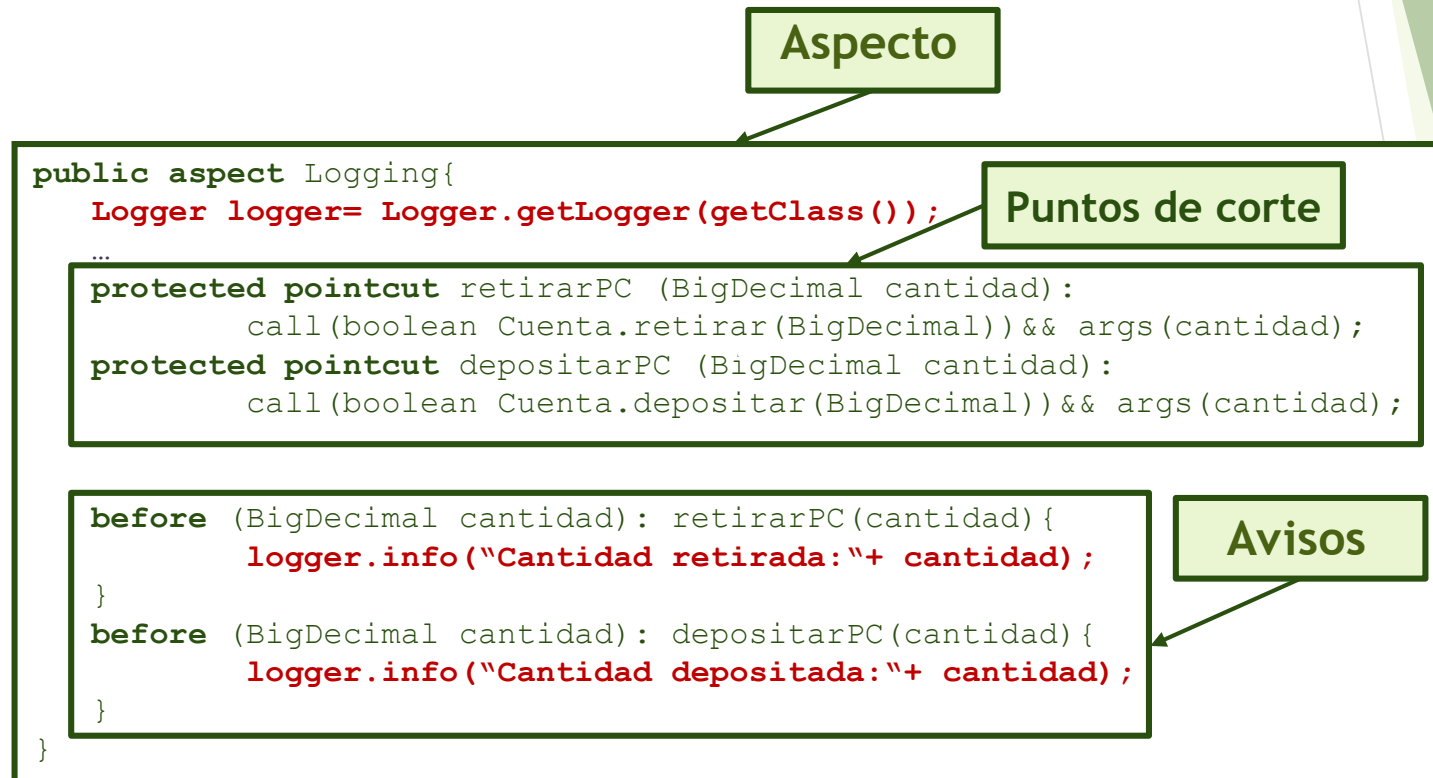
Pasos a seguir para crear un aspecto:

1. Identificar los **puntos de enlace (JoinPoint)** donde se desea añadir el comportamiento transversal.
2. Agrupar dichos puntos de enlace en uno o varios **puntos de corte (PointCut)**.
3. Implementar el comportamiento transversal con un **aviso (advice)**, 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. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

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

- AspectJ permite seleccionar puntos de enlace basados tanto en:
 - **Información estructural** (como por ejemplo tipos, nombres, argumentos, anotaciones)
 - **Condiciones** referentes a aspectos en **tiempo de ejecución** (como por ejemplo, en el flujo de control)

JoinPoints

Las categorías de puntos de enlace identificadas por AspectJ son (11):

Categoría	Punto de enlace	Código que representa	Descripción
Método	Llamada	Invocación del método	Cuando un método es invocado por un objeto (ej. <code>c.setX(10);</code>)
Método	Ejecución	Cuerpo del método	Engloba la ejecución de todo el código dentro del cuerpo de un método.
Constructor	Llamada	Invocación de la lógica de creación de un objeto	Cuando se invoca un constructor durante la creación de un nuevo objeto (ej. <code>Clase c = new Clase(5,5);</code>)
Constructor	Ejecución	Ejecución de la lógica de creación de un objeto	Engloba la ejecución del cuerpo de un constructor durante la creación de un nuevo objeto.
Acceso a atributo	Lectura	Acceso para leer un atributo de objeto o de clase	Cuando se lee un atributo de un objeto/clase dentro de una expresión (ej. <code>return c;</code>)
Acceso a atributo	Escritura	Acceso para escribir (modificar) un atributo de objeto o de clase	Cuando se asigna un valor a un atributo de un objeto (ej. <code>c=i;</code>)
Procesamiento de excepción	Handler	Bloque catch para manejar la excepción	Ocurre cuando se captura una excepción.
Inicialización	Inicialización de clase	Inicialización de una clase	Cuando se ejecuta el inicializador estático de una clase específica, en el momento en el que se carga dicha clase.
Inicialización	Inicialización de objeto	Inicialización de un objeto en un constructor	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: <code>super(); this.x = x; this.y = y;</code> , la llamada al constructor padre (<code>super</code>) no forma parte del punto de enlace.
Inicialización	Pre-inicialización de objeto	Pre-inicialización de un objeto en un constructor	Ocurre antes de que el código de inicialización de un objeto 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.
Aviso	Ejecución	Ejecución de un aviso.	Comprende la ejecución del cuerpo de un aviso .

JoinPoints

Hay más
diferencias
sutiles entre
los dos

Puntos de enlace de métodos.

Representan la *ejecución* o la *invocación* de un método.

Dos tipos: *ejecución* (execution) e *invocación* (call)

```
public class Cuenta{
    BigDecimal cantidadActual= getCantidadActual();
    ...
    public void retirar(BigDecimal cantidadARetirar) throws CantidadInsuficienteException{
        if(cantidad < cantidadRetirada)
            throw new CantidadInsuficienteException("Cantidad no suficiente");
        else
            cantidadActual = cantidadActual-cantidadARetirar;
    }
}
```

Punto de enlace de
ejecución del
método

```
Cuenta cuenta = new Cuenta(12323243,500);
Cuenta.retirar(100);
```

Punto de enlace de invocación
del método

La diferencia entre ambos se encuentra en el proceso de entretejido (en el weaver):

- Si se desea “avisar” la *llamada*, el tejedor añadirá el código del aviso en todas las llamadas o invocaciones al método.
- Si se desea “avisar” la *ejecución*, el tejedor añadirá el código del aviso dentro del cuerpo del método.

JoinPoints

Puntos de enlace de constructor.

Parecidos a los anteriores, con la diferencia de que representan la ejecución o la invocación de la construcción de un objeto.

Dos tipos: ejecución (*execution*) e invocación (*call*)

```
public class Cuenta{  
    ...  
    public Cuenta(int numCuenta, double cantidad){  
        this.numCuenta=numCuenta;  
        this.cantidadActual = cantidad;  
    }  
}
```

Punto de enlace de ejecución del constructor, comprende todo el cuerpo del constructor

```
Cuenta cuenta = new Cuenta(12323243,500);
```

Punto de enlace de invocación del constructor

La diferencia entre ambos es similar al caso anterior.

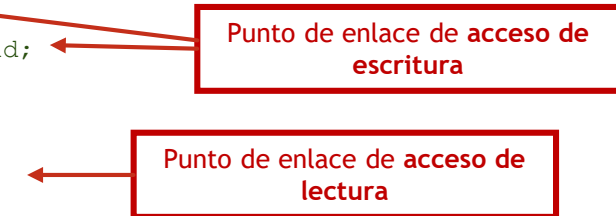
JoinPoints

Puntos de enlace de acceso a campos o atributos.

Se utilizan cuando se desea realizar alguna acción adicional cuando se accede (leer o escribir) a una instancia o miembro de una clase.

Dos tipos: lectura (*read*) y escritura (*write*)

```
public class Cuenta{
    ...
    public Cuenta(int numCuenta, double cantidad){
        this.numCuenta=numCuenta;
        this.cantidadActual = cantidad;
    }
    public String toString(){
        return "Cuenta: " + numCuenta;
    }
}
```



En el primer caso, el punto de enlace de acceso de escritura engloba la asignación a `cantidadActual` en el constructor.

En el segundo caso, el punto de enlace de lectura engloba la lectura del campo `numCuenta` como parte de la creación del `String` que devuelve el método `toString`.

Notar: estos puntos de enlace son similares a los puntos de enlace de ejecución de los métodos getters/setters, excepto por el hecho de que los primeros se dan a nivel de campo y no necesitan de la existencia de los métodos getters/setters.

JoinPoints

Puntos de enlace de manejador de excepciones.

Se utiliza cuando deseamos responder de alguna manera determinada a un manejador de excepciones.

Estos puntos de enlace representan (engloban) el bloque manejador (`catch`) de un tipo de excepción.

```
try{  
    cuenta.retirar(cantidad);  
}catch(throws CantidadInsuficienteException ex){  
    mensajePosterior(ex);  
    manejadorDeDescubierto.solicitarProtecciónDescubierto(cuenta, cantidad);  
}  
}
```

Punto de enlace de manejador
de excepciones

JoinPoints

Puntos de enlace de inicialización de clases.

Representan la carga de una clase, incluyendo la inicialización de la porción estática (incluyendo el bloque `static`).

```
public class Cuenta{  
    static{  
        try{  
            System.loadLibrary("cuenta");  
  
        }catch{UnsatisfiedLinkError er)  
            //tratamiento del error  
        }  
  
    }  
    ...  
}
```

Punto de enlace de
inicialización de clase

JoinPoints

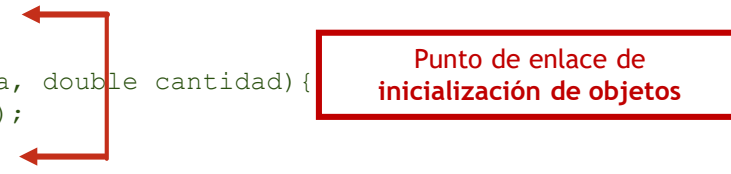
Puntos de enlace de inicialización de objetos.

Representan la inicialización de un objeto, **desde el retorno del constructor de la clase padre** hasta el final del primer constructor invocado.

```
public class CuentaAhorros extends Cuenta{

    public CuentaDeAhorros(int numCuenta, double cantidad, boolean descubierta){
        super(numCuenta, cantidad);
        this.descubierta?= descubierta;
    }
    public CuentaDeAhorros(int numCuenta, double cantidad){
        this(numCuenta, cantidad, false);
        this.saldoMedio= 58;
    }

}
```



Punto de enlace de inicialización de objetos

En el primer caso, el punto de enlace comprende la asignación del miembro **descubierta?** (y no **super**), mientras que en el segundo caso, el punto de enlace comprende las asignaciones del primer y del segundo constructor.

Notas:

- A diferencia del punto de enlace de *ejecución de constructor*, este punto de enlace ocurre solo en el primer constructor invocado para cada tipo de jerarquía.
- A diferencia del punto de enlace de *inicialización de clase* (que ocurre cuando se carga una clase) la inicialización de un objeto ocurre cuando el objeto es creado.

JoinPoints

Puntos de enlace de pre-inicialización de objetos.

Comprenden el fragmento desde el primer constructor invocado hasta el comienzo del constructor padre (prácticamente, las invocaciones realizadas mientras que se forman los argumentos de la invocación del método `super()` en el constructor).

No se utilizan mucho.

```
public class CuentaAhorros extends Cuenta{  
  
    public CuentaDeAhorros(int numCuenta, double cantidad){  
        super(numCuenta, cantidad,  
            manejadorDeCuentas.internalId(numCuenta));  
    }  
  
}
```

Punto de enlace de
pre-inicialización de objetos

En el ejemplo, el punto de enlace engloba únicamente la llamada al método `manejadorDeCuentas.internalId(numCuenta)` (y no la llamada entera a `super()`).

JoinPoints

Puntos de enlace de avisos.

Es un punto de enlace propio de AspectJ (no corresponde a ningún constructor Java).

Comprenden la ejecución de cualquier aviso en el sistema.

```
public aspect MonitorizacionDeActividadDeCuentaAspect{  
    ...  
    before(): actividadDeCuenta()  
        logActividadDeCuenta(thisJoinPoint);  
}  
  
}
```

Punto de enlace de
ejecución de aviso

En el ejemplo, el punto de enlace engloba el aviso `before` en el aspecto `MonitorizacionDeActividadDeCuentaAspect`.

PointCuts

- **Objetivo:** identificar un conjunto de puntos de enlace (uno o varios puntos de enlace) y permitir exponer el *contexto* de dichos puntos de enlace a los avisos.

- Elementos a tener en cuenta:

- **Contexto de un punto de enlace:** refiere a la información en tiempo de ejecución asociada a un punto de enlace.

Ejemplo: en una invocación a un método (*call*) la información de contexto sería el objeto que invoca al método, el objeto al cual pertenece el método invocado, los argumentos, las anotaciones del método, etc.

- **Signatura o patrones:** los puntos de corte usan patrones para seleccionar los puntos de enlace.

Ejemplo: si queremos hacer referencia, por ejemplo, a cualquier método público, independientemente del valor que devuelva y del número de parámetros, de la clase Cuenta bancaria, usaríamos la **signatura**:

```
public * Cuenta.* (..)
```

PointCuts

- **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

```
[!] designator [ && | || ]  
designator ::= designator_identifier(<signature>)
```

Un punto de corte anónimo consta de una o más expresiones o descriptores (**designator**) que identifican una serie de puntos de enlace. Cada **expresión** está formada por un descriptor del punto de corte (**designator_identifier**) y una **signatura** (**signature**).

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

Este punto de corte define una **expresión** que identifica, de **cualquier clase**, las llamadas a sus métodos **públicos** con **cualquier tipo devuelto** y **cualquier nombre**, con un argumento de tipo entero **int**.

PointCuts

Sintaxis de un punto de corte con nombre

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

Un punto de corte con nombre consta, además de las partes que tiene un punto anónimo, del modificador de acceso (**access_type**), que puede ser el mismo que el de datos o métodos Java (`public`, `private`, etc.), de la palabra reservada `pointcut`, del nombre del punto de corte (**pointcut_name**) y sus parámetros (**parameters**). A continuación, vienen los dos puntos `:` seguidos de las expresiones o descriptores que identifican a los puntos de enlace.

Los puntos de corte con nombre se pueden definir **dentro de clases** o de **aspectos**.

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

`publicIntCall` define dos expresiones combinadas mediante el operador lógico AND (`&&`). La primera expresión identifica las llamadas a los métodos públicos con cualquier tipo devuelto, nombre y un parámetro entero. La segunda expresión indica que el argumento del método anterior debe ser identificado por la variable `i` (más adelante hablaremos de los parámetros, tanto de los puntos de corte como de los avisos).

PointCuts. Operadores lógicos

Operadores lógicos para identificar los puntos de enlace:

`exp1 || exp2` La expresión compuesta se cumple cuando se satisface al menos una de las dos expresiones: *exp1* o *exp2*.

`exp1 && exp2` La expresión compuesta se cumple cuando se cumple *exp1* y *exp2*.

`! exp` La expresión compuesta se cumple cuando no se satisface *exp*.

PointCuts. Wildcards

Wildcards o Comodines para identificar los puntos de enlace:

- * En algunos contextos significa **cualquier número de caracteres excepto el punto "."** y en otros representa cualquier tipo (paquete, 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.

- .. 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 cualquier secuencia de caracteres que empieza y termina en un punto ".", por lo que se puede usar para representar 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 precedido de una clase (siempre se utiliza como sufijo).

`com.dtsi.Clase+`


Se pueden
usar caracteres
comodín

PointCuts. Patrones de Descriptores

La **signatura** de los **descriptores** sigue unos determinados patrones:

- **Patrones de tipo:** se utilizan para denotar un conjunto de tipos (clases, interfaces, anotaciones y tipos primitivos).

<code>int</code> →	tipo entero
<code>Point</code> →	solamente el tipo <code>Punto</code>
<code>*</code> →	cualquier tipo
<code>*Cuenta</code> →	cualquier tipo cuyo nombre termina en <code>Cuenta</code>
<code>Point[]</code> →	arrays de <code>Point</code>
<code>java.*.Date</code> →	tipo <code>Date</code> de cualquier subpaquete directo del paquete <code>java</code> (por ejemplo de <code>sql</code> o de <code>util</code>)
<code>java.io.OutputStream+</code> →	incluye subtipos de <code>OutputStream</code>
<code>Figura+ && !Figura</code> →	subtipos de <code>Figura</code> excluyendo a <code>Figura</code>
<code>java..*</code> →	cualquier tipo dentro del paquete <code>java</code> o cualquier de sus subpaquetes directos e indirectos (<code>java.awt</code> , <code>java.awt.event</code> , etc.).
<code>javax..*Model+</code> →	todos los tipos del paquete <code>javax</code> y sus subpaquetes directos o indirectos que tienen un nombre terminado en <code>Model</code> , y sus subtipos (<code>TableModel</code> , <code>TreeModel</code> , ..., <code>AbstractTableModel</code> , <code>DefaultTableModel</code> ...)



Se pueden
usar caracteres
comodín

PointCuts. Patrones de Descriptores

- **Patrones de método:** se utilizan para denotar un conjunto de métodos.

Teniendo en cuenta la siguiente declaración de método:

```
public final void write(int x) throws IOException
```

Un patrón del método anterior podría ser: `public final void Writer.write(int) throws IOException`

Otro patrón usando caracteres comodín para el ejemplo anterior:

```
* *.write(int) →
```

cualquier método llamado `write` publico, de cualquier clase, con un parámetro de tipo `int` (**notar** que en este caso no solamente cubrimos los métodos `public final void`. Además el método `write` podría estar en cualquier clase).

Otros ejemplos con uso de caracteres comodín:

`public void Figura+.set*(..)` → cualquier método `public` que comience por `set`, de la clase `Figura` o subclases, que devuelva `void` y tenga cualquier número de parámetros y de cualquier tipo.

`public * Cuenta.*(*)` → cualquier método `public` de la clase `Cuenta`, que devuelva cualquier tipo y tenga un único parámetro de cualquier tipo.

`* *.*(..)` → cualquier método del sistema (independientemente del modificador de acceso, del tipo devuelto, del tipo al que pertenece, del nombre y de sus argumentos (equivalentemente, `**(..)`).

`Cuenta ServicioCuenta.*(..)` → cualquier método de `ServicioCuenta` que devuelva un parámetro de tipo `Cuenta`.

Se pueden
usar caracteres
comodín

PointCuts. Patrones de Descriptores

- **Patrones de constructor:** seleccionan un conjunto de constructores (como los patrones de método pero con `new` y sin tipo devuelto).

`*.new(..) →`

constructor de acceso publico de cualquier clase y con cualquier tipo número y tipo de parámetros.

`*.Handler+.new(int,..) →`

cualquier constructor de acceso público de la clase `Handler` o subclases y que tenga al menos un parámetro entero.

- **Patrones de campo/atributo:** se utilizan para hacer referencia a un conjunto de campos/atributos.


`private int Line.x →`

campo privado (de instancia o estático) llamado `x`, de tipo `int`, de la clase `Line`.

`* !final * FigureElement+.* →`

campo no `final`, de la clase `FigureElement` o subclases, con independencia del modificador de acceso, tipo y nombre.

- **Patrón excepción** (como el de tipo)



Se pueden
usar caracteres
comodín

PointCuts. Patrones de Descriptores

➤ Patrones basados en genéricos

<code>Map<Long,Cuenta></code>	→	el tipo <code>Map</code> , con el primer parámetro genérico de tipo <code>Long</code> y el segundo de tipo <code>Cuenta</code> .
<code>*<Cuenta></code>	→	cualquier tipo con <code>Cuenta</code> como parámetro (<code>Collection<Cuenta></code>)

➤ Patrones que combinan otros patrones usando las operaciones unarias (!) y binarias (&& y ||)

<code>!Collection</code>	→	cualquier tipo que no sea <code>Collection</code>
<code>Collection Map</code>	→	los tipos <code>Collection</code> o <code>Map</code>

PointCuts. Patrones de Descriptores

➤ Ejercicios:

¿Qué seleccionan los siguientes puntos de enlace?

1. `!public * *.*(..) →`

2. `final * A.set*(String,String) → (ver nota 1)`

2. `* *.*(..,int) throws (Exception && !SecurityException) →`

3. `* java.io.PrintStream.printf(String, Object ...) → (ver nota 2)`

Nota:

- 1) Notar que la clase A debería estar en el mismo paquete que el aspecto. De no ser así, debería indicarse el nombre completo de la clase.
- 2) Los tres puntos refieren en Java a un número variable de argumentos (en este caso, de tipo `Object`)

PointCuts. Descriptores

AspectJ ofrece varios **descriptores** de puntos de corte. En particular, los puntos de corte pueden seleccionar los puntos de enlace de dos formas diferentes:

- ***Kinded pointcuts***: en este caso los puntos de corte seleccionan las categorías de **puntos de enlace indicadas anteriormente** (por ejemplo, *AspectJ proporciona un descriptor de punto de corte para “hacer matching” sobre la ejecución de un método*).

- `execution (<patrón-método>/<patrón-constructor>)`
- `call (<patrón-método>/<patrón-constructor>)`
- `get (<patrón-atributo>) / set (<patrón-atributo>)`
- `handler (<patrón-excepción>)`
- `staticinitialization (<patrón-tipo>)`
- `initialization (<patrón-constructor>)`
- `preinitialization (<patrón-constructor>)`
- `adviceexecution ()`

- ***Non-kinded pointcuts***: en este caso los puntos de corte seleccionan puntos de enlace en base a la información de la que disponen dichos puntos de enlace (por ejemplo tipos en tiempo de ejecución del contexto del punto de enlace, el flujo de control o el alcance léxico).

(Algunos) descriptores kinded

- **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.

```
get/set(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)
```

* Refiere a que hay más descriptores relacionados.
Los recuadros son meros ejemplos.

(Algunos) descriptores non-kinded

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

`withincode(<patrón-método/constructor>)`

Capturan los puntos de enlace que se localizan dentro del alcance léxico, es decir, en un fragmento de código fuente.

```
within(miAspecto)/within(MiClase+)
withincode(* com.dtsi.A.m(..))
```

```
class A {
    public void m() {
        B b = new B();
        b.n();
    }
}

class B {
    public void n() {
        ...
    }
}
```

El punto de corte

`withincode(* principal.A.m(..))` **capturará:**

- La llamada al constructor de B
- La llamada al método n

(Algunos) descriptores non-kinded

➤ **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 utilizado para ser asignado a un atributo.

(Algunos) descriptores non-kinded

➤ Explicación más detallada de `this` y `target`:

```
class A {  
    public void m() {  
        B b = new B();  
        b.n();  
    }  
}  
  
class B {  
    public void n() {  
        ...  
    }  
}
```

El punto de corte

`execution(* *.m(..))` **capturará**
`A.m()` **y tendrá:**

- `this()` **de tipo A**
- `target()` **de tipo A** (ambos serán la misma instancia de A)

En cambio, el punto de corte

`call(* *.n(..))` **capturará la invocación de**
`n()` **dentro de `A.m()` y en ese momento:**

- `this()` **será la instancia de A que hace la llamada**
- `target()` **será la instancia de B sobre la cual se está invocando el método n.**

PointCuts

Paso de información de contexto (I)

➤ 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 otros 3 objetos que nos permiten acceder a información de los punto de enlace de forma reflexiva. La información contenida en estos tres objetos es de dos tipos:

- **Información estática:** no cambia entre múltiples ejecuciones. Por ejemplo, el nombre y la localización origen de un método no cambia durante diferentes ejecuciones del método. **Objetos `thisJoinPointStaticPart` y `ThisEnclosingJoinPointStaticPart`.**
- **Información dinámica:** cambia con cada invocación del mismo punto de enlace. Por ejemplo, dos llamadas diferentes al método `Cuenta.retirar` probablemente tendrán objetos `cuenta` y cantidades a retirar diferentes. **Objeto `thisJoinPoint`.**

PointCuts

Paso de información de contexto (II)

➤ El objeto `thisJoinPoint` nos permite acceder a información dinámica sobre el contexto del punto de enlace actual. Entre sus métodos destacamos:

- `Object getThis()`: devuelve el objeto actual en ejecución.
- `Object getTarget()`: devuelve el objeto destino.
- `Object[] getArgs()`: devuelve los argumentos del punto de enlace.
- `Signature getSignature()`: devuelve un objeto de la clase `Signature` que representa la signatura del punto de enlace (a partir de dicho objeto, podemos acceder al identificador del punto de enlace (`getName()`), a sus modificadores de acceso (`getModifiers()`), etc.

➤ Equivalencias

```
pointcut pc(Cuenta c): this (c) → Cuenta c=(Cuenta) thisJoinPoint.getThis()
pointcut pc(Cuenta c): target (c) → Cuenta c=(Cuenta) thisJoinPoint.getTarget()
pointcut pc(Cuenta c, Cliente cl): args(c,cl) → Object[] argumentos= thisJoinPoint.getArgs();
                                           Cuenta c=(Cuenta) argumentos[0];
                                           Cliente cl=(Cliente) argumentos[1];
```

PointCuts

Paso de información de contexto (III)

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.



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

PointCuts. Combinaciones

Los **puntos de corte** pueden definirse como **combinaciones** de otros puntos de corte.

Por ejemplo:

```
pointcut pointCutA(): call ( * ClassA.*(..));  
pointcut pointCutB(): pointCutA() && within(ClassB);
```

En el ejemplo anterior, el punto de corte **pointCutB** capturaría el punto de enlace referido a la llamada a todos los métodos de la clase **ClassA** y todos los puntos de enlace dentro de la clase **ClassB**.

Ejemplos

Ejemplos

Ejemplos de puntos de corte y los puntos de enlace identificados:

Punto de corte	Puntos de enlace identificados
<code>execution(public void Clase.set*(int))</code>	Captura la ejecución de los métodos públicos de la clase Clase que empiezan por <code>set</code> , devuelven <code>void</code> y tienen un único parámetro de tipo <code>int</code> .
<code>execution(void C.m(..., int))</code>	Captura la ejecución de métodos llamados <code>m</code> , de la clase <code>C</code> , que devuelvan <code>void</code> y cuyo último parámetro sea de tipo entero <code>int</code> .
<code>call(public void Clase.*Modification(*))</code>	Captura la llamada de todos los métodos públicos de la clase Clase que terminan en <code>Modification</code> , devuelven <code>void</code> y tienen un único parámetro de cualquier tipo.
<code>call(public final void *.*() throws ArrayOutOfBoundsException)</code>	Captura las llamadas a los métodos, independientemente de su nombre o la clase a la que pertenecen, que no tengan argumentos, devuelvan <code>void</code> , sean <code>public</code> y <code>final</code> , y en su declaración indiquen que lanzan la excepción <code>ArrayOutOfBoundsException</code> .
<code>if(thisJoinPoint.getKind().equals("call"))</code>	Identifica si el punto de enlace refiere a una invocación.

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(s) de corte* y *cuerpo*.

declaración

```
[tipo de retorno] tipo aviso ([parámetros]): punto(s) de corte {
    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

Ejemplo:

Declaración

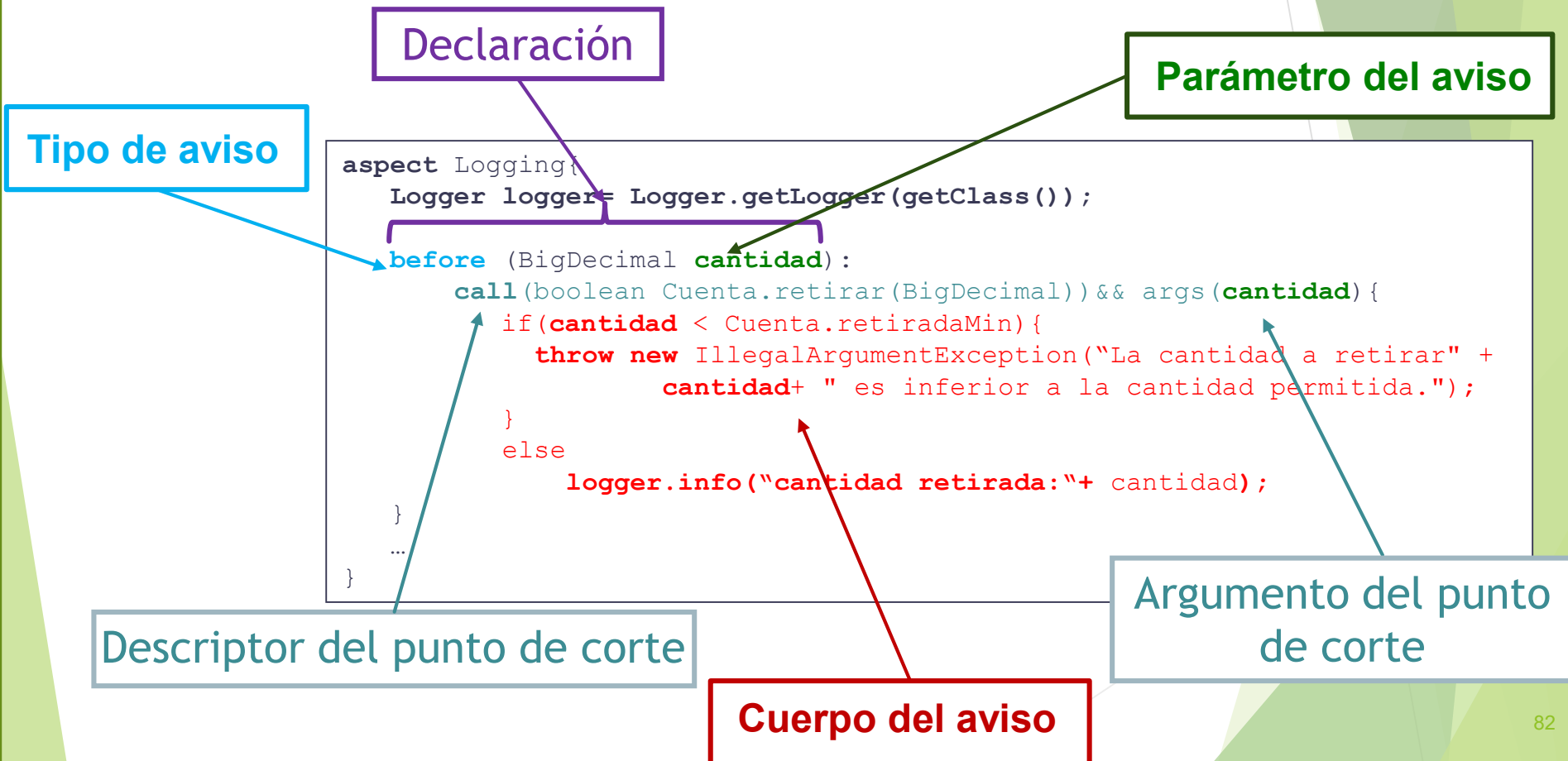
Punto de corte

```
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);  
}
```

Cuerpo

Avisos

Ejemplo (más completo):




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, usados para recuperar el valor retornado o la excepción lanzada por un punto de enlace, respectivamente).

Avisos

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



```
before(int i): call(public Clase.setX(int)) && args(i){  
    Logger.log(Level.INFO, "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){  
    Logger.log(Level.INFO, "Trazando: entrada a " + thisJoinPoint.getSignature() +  
        "con valor del parámetro"+ i);  
}
```

Avisos before

- Es el aviso más sencillo.
- Indica que el cuerpo del aviso se ejecuta cada vez **antes** de que el flujo de la ejecución del programa entre en un punto de enlace capturado por el punto de corte (o puntos de corte).
- **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.
- Esquema:

```
before ([parámetros]) : punto(s) de corte {  
    cuerpo  
}
```

Los **parámetros** reciben valores del **punto de corte**.

Avisos before

➤ Ejemplos:

```
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.log(Level.INFO, "cantidad retirada:" + cantidad);
}
```

De lanzarse la excepción, el método `retirar` no se ejecutará.

```
before(ClaseEjemplo ce) : call(* com.dtsi...*(..)) && this(ce) {
    logger.log(Level.INFO, "Llamada desde ClaseEjemplo" + ce);
}
```

```
before() : metodosATrazar() {
    logger.log(Level.INFO, "Antes: " + thisJoinPoint);
}
```

Avisos 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 (suelen llamarse *after (finally)* debido a su similitud con el bloque `finally`, que se ejecuta sin importar el resultado de la ejecución del bloque `try`).
 - `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. Si el punto de enlace termina con una excepción, el aviso no se ejecutará.
 - ¿Qué valor devuelve cada punto de enlace?
 - De llamada o ejecución de método → el mismo que el del método
 - Constructor → el objeto creado
 - Lectura de un atributo → el valor del atributo implicado.
 - Resto de categorías → null
 - `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. Si el punto de enlace finaliza normalmente, el cuerpo del aviso no se ejecutará. Si la excepción indicada es `Exception`, el aviso se ejecutará ante cualquier excepción.

Avisos after

➤ Esquemas de los avisos after:

After

```
after([parámetros]): punto(s) de corte {
    cuerpo
}
```

After returning

```
after([parámetros]) returning: punto(s) de corte {
    cuerpo
}
```

```
after([parámetros]) returning (valor devuelto): punto(s) de corte {
    cuerpo
}
```

After throwing

```
after([parámetros]) throwing: punto(s) de corte {
    cuerpo
}
```

```
after([parámetros]) throwing (excepción): punto(s) de corte {
    cuerpo
}
```


Avisos after

- Ejemplos de los tres tipos

After

```
after(): call(Cuenta+.new(..)) {
    logger.log(Level.INFO, "Se ha creado un cuenta");
}
```

After returning

```
after() returning(Cuenta c) : call(Cuenta+.new(..)) {
    logger.log(Level.INFO, "La cuenta creada es: " + c);
}
```

```
after (BigDecimal cantidad) returning: call(boolean Cuenta.depositar(BigDecimal))&&
    args(cantidad){
    logger.log(Level.INFO, "cantidad depositada:" + cantidad);
}
```

```
after() returning(Connection con) :
    call(Connection DriverManagerConnection.getConnection(..)) {
    logger.log(Level.INFO, "Obteniendo la conexión de la BD: " + con);
}
```

After throwing

```
after() throwing (Exception ex): call(public * *.*(..)){
    contadorExcepcion++;
    logger.log(Level.INFO, ex);
    logger.log(Level.INFO, "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.
- Esquema:

```
tipo de retorno around([parámetros]): punto(s) de corte {  
    ...  
    [return] [proceed(...);]  
    ...  
}
```

Avisos around

- Se puede usar el 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). Si no se invoca a `proceed`, entonces el punto de enlace es ignorado.
- 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.

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

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

Notas:

- `proceed` tiene el mismo tipo de retorno que el aviso → `Object`
- AspectJ realiza la conversión envolviendo el entero `int` devuelto en un `Integer`
- Aunque el aviso devuelva un objeto de la clase `Object`, AspectJ lo convierte en un entero `int` una vez terminada la ejecución.

Si el aviso `around` se aplica a puntos de enlace con diferentes tipos return la estrategia es indicar que devuelve `Object` (funciona incluso si un punto de enlace devuelve `void`)

Avisos around

➤ Ejemplos (I)

Uso para comprobar parámetros: modificación del ejemplo anterior.

```
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.log(Level.INFO,"cantidad retirada:"+ cantidad);
            return proceed(cantidad);
        }
    }
```

Cambiar el contexto del punto de enlace: se añade fecha/hora de impresión de todos los mensajes que imprimen.

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

Avisos around

➤ Ejemplos (II)

Modificar las salidas

```
public class Punto {
    int x,y;
    public void setX(int x1) {
        this.x = x1;
    }
    ...
    public static void main(String[] args) {
        Punto p = new Punto();
        p.setX(5);
        p.setY(4);
        System.out.println("(x,y): (" + p.x+ " ,"+p.y+"")");
    }
}
```

```
public aspect Aspect {
    pointcut puntoPC(int x):
        call(* Punto.setX(..)
            && args(x);

    void around(int x) : puntoPC(x) {
        x*= 2;
        proceed(x);
    }
}
```

Salida→ (x,y):(10,4)

Avisos around

➤ Ejemplos (III)

Registrar el tiempo requerido por la ejecución de operaciones

```
Object around (Connection connection)
: connectionOperation (connection){

    long startTime = System.nanoTime();
    Object ret= proceed (connection);
    logger.log(Level.INFO, "Operation "+ thisJoinPoint.getSignature().getName()
                + " on " + connection + " took"
                + (System.nanoTime()- startTime));

    return ret;
}
```

Avisos around

➤ Ejemplos (IV)

Lanzamiento de excepciones

```
void around (Cuenta cuenta, float cantidad)
    throws CantidadInsuficienteException
: call(void Cuenta.retirar(float) throws CantidadInsuficienteException)
  && target(cuenta)
  && args(cantidad){
    try{
      proceed(cuenta, cantidad);
    } catch (CantidadInsuficienteException ex)
      if (!procesoDescubierto(cuenta, cantidad))
        throw ex;
    }
}
```

El punto de corte selecciona cualquier llamada al método retirar de Cuenta que lance CantidadInsuficienteException.

Como el código del aviso lanza explícitamente una excepción, entonces el aviso necesita declararlo.

Si los puntos de enlace lanzan diferentes tipos de excepción se necesitaría usar otras técnicas (consultar el libro “AspectJ in action”)

```
Object around (Cuenta cuenta, float cantidad)
: call(void Cuenta.retirar(float) throws CantidadInsuficienteException){
  long start = System.nanoTime();
  proceed();
  long end= System.nanoTime();
  logger.log(Level.INFO, "El método retirar llevo "+ (end-start) + "nanosegundos");
}
```

Si el código del aviso no lanza explícitamente una excepción, entonces el aviso no necesita declararlo.

Aspectos

- Un **aspecto** puede declarar **puntos de corte**, **avisos** (y otros descriptores específicos para realizar el static crocutting), y además puede contener sus propios **atributos** y **métodos** como una clase Java normal.
- Los aspectos de **primer nivel** pueden tener solo acceso `public` o *package* (es el que se le asigna si no se indica nada). Los **aspectos anidados** (aspectos embebidos en clases o interfaces), al igual que las clases, pueden tener acceso `public`, `private`, `protected`, o *package*.
- **¿Dónde lo definimos?** Se pueden definir en varios sitios:
 - o Si el aspecto **afecta a distintas partes de la aplicación** → se suele crear en un **fichero independiente** con extensión `.aj`.
 - o Si el aspecto **afecta a una única clase de la aplicación** → se suele crear **como un miembro más de la clase (aspecto anidado)** → tendrá que ser `static`
 - De hecho, pueden definirse como miembros `static` de clases, interfaces u otros aspectos.
 - Suele recomendarse su definición como `private` para la encapsulación total, de forma que solo pueda ser accedido desde la clase que lo contenga.

Aspectos

- Ejemplo de aspecto anidado en una clase:

Archivo Cliente.aj

```
package modelo;

public class Cliente
{
    public static void main(String[] args) {
        Cliente c = new Cliente();
    }

    static private aspect miAspecto {
        pointcut myConstructor(): execution(new(..));

        before(): myConstructor() {
            logger.log(Level.INFO, "Entering Constructor");
        }
    }
}
```

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

AspectJ. Clases y aspectos

Diferencias y similitudes

Característica	Clases	Aspectos
Puede extender clases	Sí	Sí
Puede implementar interfaces	Sí	Sí
Puede extender aspectos	No	Sí (solo aspectos abstract)
Puede ser declarado internamente	Sí	Sí (se debe declarar static)
Puede ser instanciado directamente	Sí	No

AspectJ. Métodos y avisos

➤ Semejanzas. Comparados con los métodos, los avisos...

- Pueden tener un nombre (el cual se daría usando anotaciones)
- Toman los argumentos pasados como información de contexto, y luego pueden usar dicha información para su lógica (al igual que los métodos)
- Podrían declarar que lanzan excepciones. Todas las excepciones que puedan ser lanzadas por su implementación, deberían ser declaradas por el aviso.
- El código dentro de un aviso sigue las mismas reglas de control de acceso que los métodos para acceder a los miembros de otros tipos y aspectos.
- Pueden referir a la instancia del aspecto con `this`.
- `around` pueden devolver un valor (y, por tanto, dicho aviso declarará un tipo `return`).
- ...

➤ Diferencias. Comparados con los métodos, los avisos...

- Opcionalmente tienen un nombre
- No se pueden invocar directamente (el ejecutarlos es rol del sistema)
- No cuentan con especificador de acceso.
- `before` y `after` no incluyen un tipo devuelto en la signatura.
- Tienen acceso a pocas variables con información sobre los puntos de enlace, aparte de `this`, `target`, `args`, `thisJoinPoint`, `thisJoinPointStaticPart` y `thisEnclosingJoinPointStaticPart`.¹⁰⁰
- Podrían usar la palabra `proceed` en los avisos `around` para proceder con el punto de enlace del aviso.

AspectJ. Extensión de Aspectos

➤ Un aspecto puede **extenderse** de varias formas:

- **Construyendo un aspecto abstracto** `abstract` que pueda ser extendido por otros aspectos (heredando los avisos y puntos de corte, además de los atributos y métodos).

Un aspecto no puede heredar de aspectos concretos (provocará un error de compilación), sólo puede heredar de aspectos abstractos.

- **Heredando de clases o implementando interfaces.** Un aspecto abstracto o concreto podría extender una clase y podría implementar un conjunto de interfaces.

Una clase no puede extender o implementar un aspecto (provocará un error de compilación).

```
public abstract aspect A {  
    ...  
}
```

AspectJ. PointCuts abstractos

Puntos de corte abstractos

- Los puntos de corte **con nombre** pueden ser **abstractos** (y ser 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.

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

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

```
public 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 del mismo aspecto).

AspectJ. Prioridad entre Aspectos (precedence)

- ¿Qué hacemos cuando **dos avisos definidos en aspectos diferentes afectan al mismo punto de enlace**? ¿Y si nos interesa establecer una relación de prioridad entre los avisos, para que uno se ejecute antes que otro?

Declaraciones de prioridad: `declare precedence`

La instrucción se debe indicar dentro de un aspecto.

- Nos permiten indicar que todos los avisos de un aspecto deben tener prioridad a la hora de ejecutarse con respecto a los de otros aspectos, con respecto a un punto de enlace.
- Si no se indica la prioridad, el orden en el que se ejecutarían los avisos sería **impredecible**.

AspectJ. Prioridad entre Aspectos

- Si no se indica nada, si un aspecto A extiende a otro B, todos los avisos del subaspecto A tienen mayor prioridad que los avisos del superaspecto B.
- Si, por ejemplo, tenemos dos aspectos y queremos indicar la prioridad entre ellos, es recomendable **crear otro aspecto** que declare dicha prioridad.

```
public aspect AspectC{  
    declare precedence: AspectA, AspectB;  
}
```

Si la instrucción de prioridad se añade en uno de los dos aspectos, el resultado es el mismo, pero obligaría a que dicho aspecto supiera de la existencia del otro.

- En el listado de la instrucción `declare precedence` **no** se puede incluir un aspecto más de una vez.

AspectJ. Prioridad entre Aspectos

- Se pueden usar patrones de tipo para especificar la lista de prioridad.

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 prioridad)

```
public aspect AspectA {  
    declare precedence: AspectB, *; //AspectB domina sobre cualquier otro aspecto  
}
```

```
public aspect AspectA {  
    declare precedence: *, AspectB; //AspectB es el que tiene menos prioridad.  
}
```

AspectJ. Prioridad entre Aspectos

➤ Ejemplos:

- **Ejemplo 1:** `declare precedence: Security, Logging, * ;`

Para cada punto de enlace, los avisos de `Security` tienen prioridad respecto a los de `Logging`, cuyos avisos tienen prioridad sobre cualquier otro aviso.

- **Ejemplo 2:** `declare precedence: Security, Logging, L* ;`

Darí­a error de compilación porque el aspecto `Logging` aparecería dos veces.

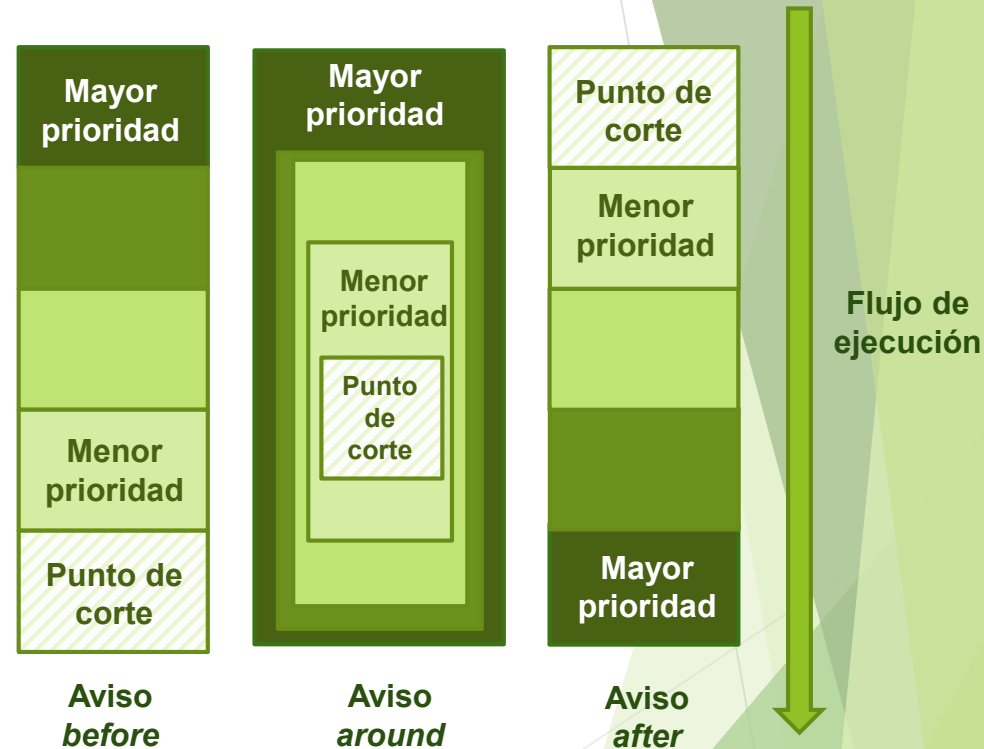
- **Ejemplo 3:** `declare precedence: Security+, * ;`

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

AspectJ. Prioridad entre Avisos

➤ También se establece un **orden entre los tipos de avisos**:

- o El aspecto con prioridad más alta ejecuta su aviso `before` en un punto de enlace, antes del aspecto con prioridad más baja.
- o El aspecto con prioridad más alta ejecuta su aviso `after` en un punto de enlace después del aspecto con prioridad más baja.
- o El aviso `around` del aspecto con prioridad más alta engloba al aviso `around` del aspecto con prioridad más baja. Esto permite al aspecto con mayor prioridad controlar si se ejecutará el aviso con prioridad más baja (mediante la ejecución o no a `proceed`).
 - Si el aspecto con mayor prioridad no invoca a `proceed` en el cuerpo del aviso, entonces no se ejecutará ni el aspecto con prioridad más baja ni el punto de enlace avisado (consultar el libro “AspectJ in action”).



AspectJ. Aspectos privilegiados

- Dentro de un aspecto, las reglas de acceso a los miembros de una clase o un aspecto son las mismas que en Java.
 - Desde el código de un aspecto no podemos acceder a los 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 `private` o `protected`, 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) throws RuntimeException : 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.

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. **Cosas en el tintero**
9. Mitos y realidades

AspectJ y AJDT

Cosas en el tintero

- **Instanciación de aspectos** (`aspectOf`)
- **AspectJ y anotaciones**
- **AspectJ y Spring**
- **Desarrollo Software Orientado a Aspectos**
- ...

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

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 POA permite solucionar determinados problemas de una manera más adecuada, con menos esfuerzo y mejorando el mantenimiento.
- ▶ La POA promueve la programación *sloppy (descuidada)*. **Falso**
 - La POA simplemente proporciona nuevas formas de solventar problemas en áreas que no satisfacen la PP y la POO.
- ▶ 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.

Índice

Programación Orientada a Aspectos

1. Motivación
2. POA al rescate
3. Conceptos básicos de la POA
4. Tipos de tejedores. Frameworks y Lenguajes
5. Desarrollo Software Orientado a Aspectos

AspectJ

5. Introducción a AspectJ
6. Elementos de AspectJ
7. Algunas particularidades de los aspectos
8. Cosas en el tintero
9. Mitos y realidades

AspectJ y AJDT

Desarrollo de aplicaciones AspectJ

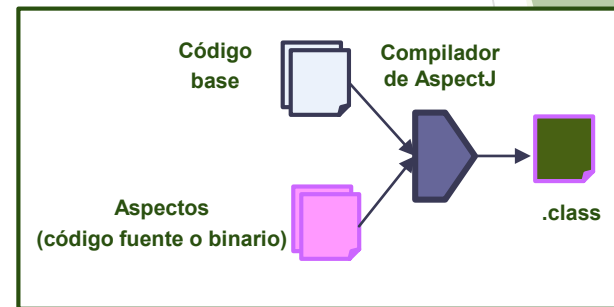
- AspectJ <http://www.eclipse.org/aspectj/>
- AspectJ Development Tools AJDT <http://www.eclipse.org/ajdt/>
 - Hay que integrar el plug-in en nuestra distribución de Eclipse
 - Última version: AJDT builds for Eclipse 4.10
 - AspectJ version: actualmente 1.9.7 (junio 2021)
 - Opciones de instalación:
 - **Desde Eclipse Help → Install new software → (Work with):**
<http://download.eclipse.org/tools/ajdt/410/dev/update>
 - Descargándose el .zip de la página web de downloads:
<http://www.eclipse.org/ajdt/downloads>

Desarrollo de aplicaciones AspectJ

➤ Los tipos de tejedores que soporta AspectJ son estáticos:

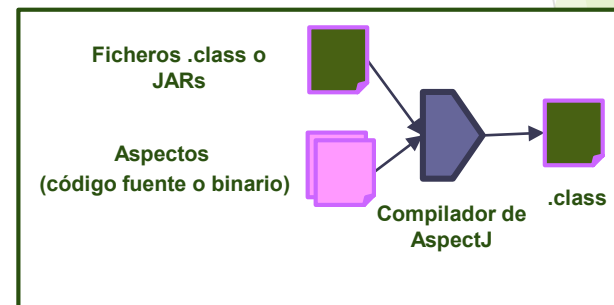
- **Compile-time weaving**

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.



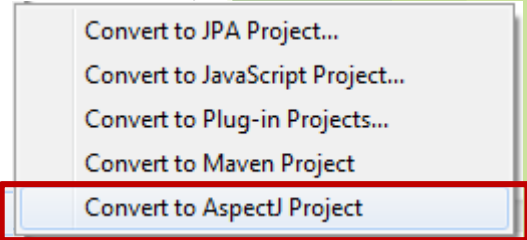
- **Load-time weaving**

El entretejido se aplaza hasta que las clases son cargadas.

Desarrollo de aplicaciones AspectJ. AJDT

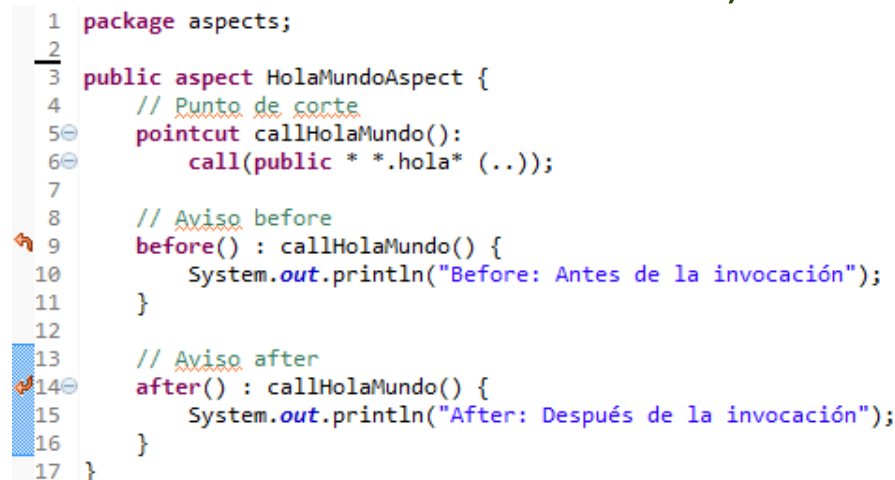
- Se puede crear directamente un **proyecto AspectJ** o crear un **proyecto Java y convertirlo** posteriormente en un proyecto AspectJ:

En el segundo caso debemos seleccionar, picando con el botón derecho sobre el nombre del proyecto → **Configure** → **Convert to AspectJ Project**



Convert to JPA Project...
Convert to JavaScript Project...
Convert to Plug-in Projects...
Convert to Maven Project
Convert to AspectJ Project

- El entorno marca el tipo de los avisos con iconos específicos en el margen (incluso se pueden cambiar los iconos de las marcas):



```

1 package aspects;
2
3 public aspect HolaMundoAspect {
4     // Punto de corte
5     pointcut callHolaMundo():
6         call(public * *.hola* (..));
7
8     // Aviso before
9     before() : callHolaMundo() {
10         System.out.println("Before: Antes de la invocación");
11     }
12
13     // Aviso after
14     after() : callHolaMundo() {
15         System.out.println("After: Después de la invocación");
16     }
17 }
  
```

- Dispone de asistente para crear aspectos.

Desarrollo de aplicaciones AspectJ. AJDT

- En un proyecto AspectJ (*WeaveMe*) puedes usar aspectos de otro proyecto AspectJ (*MyAspects*).
 - Ambos proyectos deben ser proyectos AspectJ
 - Pasos:
 - Desde el proyecto *WeaveMe*, seleccionar **Properties** e ir a la pestaña **AspectJ Aspect Path**.
 - Desde la pestaña *Libraries*, picar en **Add Class Folder** y seleccionar el directorio **bin** (o el nombre correspondiente al directorio de salida) del proyecto *MyAspects*.
- Puedes aplicar los aspectos de un proyecto de AspectJ (*MyAspects*) al código de otro proyecto (*MyJava*).
 - El código del proyecto Java puede ser código fuente o JAR.
 - Pasos:
 - Desde el proyecto *MyAspects*, seleccionar **Properties** e ir a la sección **AspectJ InPath**, para hacer referencia al proyecto Java *MyJava*.
 - Desde la pestaña **InPath**, picar en **Add Class Folder** o el botón **Add JARs** del proyecto *Java MyJava*.

Desarrollo de aplicaciones AspectJ. AJDT

- Los casos anteriores eran ejemplos de **compile-time weaving/ post-compile o binary weaving**, pero también se puede realizar un **load-time weaving**.
 - Se trata de un binary weaving aplazado hasta que se carga la clase.
 - Con un proyecto AspectJ con aspectos (*MyAspects*) y con otro Java o AspectJ (*WeaverMe*).
 - Pasos:
 - Desde el proyecto *WeaveMe* seleccionar **Run As → Run configurations...**, y picar en **AspectJ Load-Time Weaving Application** de la lista de configuración.
 - Presionar en el botón **New button**.
 1. Rellenar la pestaña **Main** con el nombre del proyecto *WeaveMe* y la clase que contiene el método principal de nuestra aplicación
 2. Pasar a la pestaña **LTW** (load-time weaving) **Aspectpath** y seleccionar **User Entries**. A continuación pulsar en el botón **Add Projects...** para seleccionar el proyecto *MyAspects*.
 3. Ahora, al seleccionar el botón **Run**, se ejecutará la aplicación normalmente, pero con el paso adicional de que se entretejerán los aspectos indicados en el LTW Aspectpath con las clases, a medida que éstas son cargadas.

Anexo.

Categorías de puntos de corte (descriptores)

PointCuts. Kindred pointcuts

Basados en métodos

❑ `call(<patrón-método>)`

Captura la llamada a un método, añadiendo el aviso antes/después/... de la llamada al método.

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

❑ `execution(<patrón-método>)`

Captura la ejecución de un método, añadiendo el aviso dentro del mismo.

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

PointCuts. Kindred pointcuts

Relacionados con constructores

- ❑ `call (<patrón-constructor>)`
- ❑ `execution (<patrón-constructor>)`

Siguen la misma idea que con métodos, pero con constructores.

Relacionados con la (pre) inicialización de objetos

- ❑ `initialization (<patrón-constructor>)`

Captura la creación (inicialización) de un objeto que utilice el constructor indicado, añadiendo el aviso antes/después/...de dicha creación.

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

- ❑ `preinitialization (<patrón-constructor>)`

Captura la creación de un objeto que utilice el constructor indicado, añadiendo el aviso antes/después/...de que el constructor `super` sea invocado.

```
preinitialization(paquete.clase.new())
```

PointCuts. Kindred pointcuts

Relacionados con la creación de clases

- ❑ `staticinitialization(<patrón-tipo>)`

Capturan la inicialización de un tipo después de ser cargado.

```
staticinitialization(paquete.clase)
```

Relacionados con atributos

- ❑ `get(<patrón-atributo>)`

- ❑ `set(<patrón-atributo>)`

Capturan la lectura o modificación de un atributo, respectivamente, añadiendo el aviso antes/después/...de que se lea/escriba el contenido del atributo sobre el que se aplica.

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

PointCuts. Kindred pointcuts

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)
```

Relacionados con la ejecución de avisos

❑ `adviceexecution()`

Representa los puntos de enlace donde se inicia la ejecución de un aviso. No requiere argumentos.

PointCuts. Non-kinded pointcuts

Relacionados con el control de flujo

- ❑ `cflow(Pointcut)`
- ❑ `cflowbelow(Pointcut)`

Capturan los puntos de enlace (de cualquier categoría) en el flujo de ejecución del punto de corte pasado por parámetro. En el caso de `cflowbelow(Pointcut)`, sin incluir los puntos de enlace identificados por el mismo punto de corte.

`cflow(execution(* Clase.set*...))` → Captura todos los puntos de enlace en el flujo de control de la ejecución de los métodos de la clase `Clase` que comiencen por `set` incluida la propia llamada a dichos métodos. El punto de corte que se pasa como parámetro es anónimo.

`cflow(call(* Clase.cambia...))` → Captura todos los puntos de enlace dentro del flujo de control de la llamada a cualquier método `cambia` dentro de la clase `Clase`, incluyendo la llamada a `cambia`. El ejemplo con `cflowbelow` sería idéntico pero sin incluir la llamada a `cambia`.

PointCuts. Non-kindred pointcuts

Relacionados con la localización de código (o en la estructura léxica)

- ❑ `within(<patrón-tipo>)`
- ❑ `withincode(<patrón-método/constructor>)`

Capturan los puntos de enlace que se localizan en determinados fragmentos de código fuente-alcance léxico (por ejemplo dentro de una clase o dentro del cuerpo de un método)

<code>within(miAspecto) / within(MiClase+)</code>	→ identifica a todos los puntos de enlace dentro del aspecto <code>miAspecto</code> / la clase <code>MiClase</code> y sus subclases.
<code>withincode(* Clase.set*(..))</code>	→ captura todos los puntos de enlace dentro del cuerpo de los métodos de la clase <code>Clase</code> cuyo nombre comienza por <code>set</code> .

PointCuts. Non-kinded pointcuts

Relacionados con el objeto en ejecución

❑ `this()`

❑ `target()`

```
this(<Tipo> o <IdentificadorObjeto>)  
target (<Tipo> o <IdentificadorObjeto>)
```

Se aplican sobre un objeto.

- `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.
 - Si se especifica el `<Tipo>` entonces se capturan los puntos de enlace donde la expresión “`this instanceof <Tipo>`” es cierta (capturando los subtipos también)
 - Si se especifica con `<IdentificadorObjeto>` captura el objeto “`this`” de forma que se puede usar el objeto en 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.

PointCuts. Non-kinded pointcuts

Relacionados con el objeto en ejecución

- `this(obj)`: refiere al objeto `this` en el punto de enlace correspondiente.
- `target(obj)`: refiere al objeto destino del punto de enlace correspondiente.
 - Para un punto de enlace de llamada a un método, el objeto destino es el objeto en el cual el método es invocado.
 - Para un punto de enlace de inicialización de objeto, el objeto destino es el objeto creado.
 - Para un punto de enlace de ejecución de un método, el objeto destino es el objeto `this` (el mismo que el recogido usando `this()`).
 - Para puntos de enlace de acceso a campos, el objeto destino es el objeto a cuyo campo se está accediendo.
 - (ver siguiente tabla)

PointCuts. Non-kinded pointcuts

Analizando `this` y `target`

Punto de enlace	Objeto <code>this</code>	Objeto <code>target</code>
Llamada a método	objeto que realiza la llamada*	objeto que recibe la llamada**
Ejecución de método	objeto que ejecuta el método*	objeto que ejecuta el método*
Llamada a constructor	objeto que realiza la llamada*	No existe (null)
Ejecución de constructor	El objeto creado	El objeto creado
Lectura de atributo	objeto que realiza la lectura*	objeto al que pertenece el atributo**
Asignación de atributo	objeto que realiza la asignación*	objeto al que pertenece el atributo**
Ejecución de manejador	objeto que ejecuta el manejador*	No existe
Inicialización de clase	No existe	No existe
Inicialización de objeto	El objeto creado	El objeto creado
Pre-inicialización de objeto	No existe	No existe
Ejecución de aviso	aspecto que ejecuta el aviso	aspecto que ejecuta el aviso

* No existe objeto `this` en contextos estáticos (como el cuerpo de un método estático o un inicializador estático)

** No existe objeto `target` para puntos de enlace asociados con métodos o campos estáticos

PointCuts. Non-kinded pointcuts

Relacionados con el argumento (argument pointcuts)

❑ `args()`

```
args(<Tipo> o <IdentificadorObjeto>, ...)
target (<Tipo> o <IdentificadorObjeto>)
```

Se utiliza para exponer los argumentos de un punto de enlace. Por ejemplo :

1. Para puntos de enlace de métodos o constructores, para referir a los argumentos pasados a un método o constructor,
2. Para puntos de enlace manejadores de excepciones, para referir a la excepción capturada por un punto de enlace handler,
3. Para puntos de enlace de acceso de escritura de campos, para referir al nuevo valor a asignar a un atributo.



PointCuts. Non-kinded pointcuts

Relacionados con el argumento (argument pointcuts)

❑ `args()`

```
args(<Tipo> o <IdentificadorObjeto>, ...)
target (<Tipo> o <IdentificadorObjeto>)
```

Ejemplos

`args(Cuenta, ..., int)` → cualquier punto de enlace de método o constructor donde el primer argumento sea de tipo `Cuenta`, y el último argumento sea de tipo `int`. Captura métodos como `print(Object, float, int)` siempre y cuando el primer argumento cumpla que es `instanceOf Cuenta`.

`args(RemoteException)` → cualquier punto de enlace con un único argumento de tipo `RemoteException`. Captura cualquier método o constructor que tenga un único argumento `RemoteException`, la modificación de un atributo asignando un valor de tipo `RemoteException`, o un manejador de excepción de tipo `RemoteException`.

PointCuts. Non-kinded pointcuts

Basados en condiciones

❑ `if(<expresión booleana>)`

Capturan puntos de enlace basándose en alguna condición.

`if(System.currentTimeMillis()>limite)` → Todos los puntos de enlace que ocurren tras sobrepasar cierta marca de tiempo.

`if(thisJoinPoint.getTarget() instanceof Clase)` → Todos los puntos de enlace cuyo objeto destino asociado sea una instancia de la clase Clase.

Una breve introducción a la Programación Orientada a Aspectos

