



Facultad de Informática de la
Universidad de Murcia

PROYECTO INFORMÁTICO

Programación Orientada a Aspectos
Una experiencia práctica con AspectJ

Alumno
Salvador Manzanares Guillén
smg3@alu.um.es

Director
Jesús J. García Molina
jmolina@um.es

Departamento de Informática y Sistemas
Junio 2005

Índice

1. Introducción.....	7
2. Programación orientada aspectos (POA).....	9
2.1. Introducción.....	9
2.2. Fundamentos de la POA.....	11
2.3. Lenguajes orientados a aspectos.....	14
2.4. Estado actual de la POA.....	18
3. AspectJ.....	20
3.1. Introducción.....	20
3.2. Especificación de AspectJ.....	20
3.2.1. Puntos de enlace.....	23
3.2.2. Puntos de corte.....	26
3.2.3. Avisos.....	36
3.2.4. Declaraciones inter-tipo.....	44
3.2.5. Declaraciones de parentesco.....	45
3.2.6. Declaraciones en tiempo de compilación.....	46
3.2.6. Suavizado de excepciones.....	47
3.2.7. Declaraciones de precedencia.....	48
3.2.5. Aspectos.....	48
3.3. Implementación de AspectJ.....	52
4. Aplicación práctica: Terminal punto de venta.....	53
4.1. Introducción.....	53
4.2. Patrones GoF con AspectJ.....	54
4.2.1. Acceso a servicios externos (Patrón Adapter).....	58
4.2.2. Elegir un servicio externo (Patrones Factoría y Singleton).....	61
4.2.3. Políticas de descuento (Patrones Estrategia y Composite).....	65
4.2.4. Soporte a reglas de negocio conectables (Patrón Fachada).....	75
4.2.5. Separación Modelo-Vista (Patrón Observer).....	76
4.2.6. Recuperación de fallos en servicios externos (Patrón Proxy).....	80
4.2.7. Manejar familias de dispositivos (Patrón Factoría Abstracta).....	84
4.2.8. Comportamiento dependiente del estado (Patrón Estado).....	86
4.3. Tracing con aspectos.....	89
4.4. Logging con aspectos.....	90
4.5. Comprobación de parámetros con aspectos.....	92
4.6. Manejo de excepciones con aspectos.....	95
4.7. Persistencia con aspectos.....	96
4.8. Interfaz gráfica del sistema TPV.....	101
4.9. Análisis de resultados.....	102
5. Conclusiones y Trabajos Futuros.....	104
Anexo I: Sintaxis de AspectJ.....	105
Anexo II: Instalación de AspectJ.....	109
Anexo III: Herramientas de AspectJ.....	122
Bibliografía.....	130

Índice de Figuras

Figura 1: Diagrama de clases del sistema editor de figuras	9
Figura 2: Comparación LPG & POA	12
Figura 3: Generar ejecutable: enfoque tradicional & POA	14
Figura 4: Entrelazado AspectJ 1.0.x / Entrelazado AspectJ 1.1.x.....	15
Figura 5: Especificación de AspectJ.....	21
Figura 6: <i>Crosscutting</i> en el sistema editor de figuras	22
Figura 7: Diagrama de clases del sistema TPV	54
Figura 8: <i>Crosscutting</i> en el patrón <i>Observer</i>	55
Figura 9: Adaptador Contabilidad	59
Figura 10: Adaptador Inventario	59
Figura 11: Diagrama de interacción de <i>relizarPago</i>	60
Figura 12: Clases adaptables y adaptadores	61
Figura 13: Factoría de servicios externos.....	62
Figura 14: Factoría <i>singleton</i> de servicios externos	62
Figura 15: Uso de la factoría de servicios	63
Figura 16: Aspecto <i>SingletonProtocol</i>	63
Figura 17: Uso de la factoría de servicios (AspectJ).....	65
Figura 18: Estructura del patrón <i>Estrategia</i> para el sistema TPV	66
Figura 19: Factoría de estrategias de descuento	67
Figura 20: Estructura del patrón <i>Composite</i> y <i>Estrategia</i> para el sistema TPV	67
Figura 21: Aspecto <i>StrategyProtocol</i>	68
Figura 22: Consecuencias de tejer el aspecto <i>EstrategiasDescuento</i>	70
Figura 23: Aspecto <i>CompositeProtocol</i>	70
Figura 24: Consecuencias de tejer el aspecto <i>EstrategiasCompuestas</i>	74
Figura 25: Fachada subsistema de reglas	75
Figura 26: Estructura del patrón <i>Observer</i> para el sistema TPV	77
Figura 27: Aspecto <i>ObserverProtocol</i>	77
Figura 28: Estructura del patrón <i>Proxy</i> para el sistema TPV	81
Figura 29: Uso del objeto proxy	81
Figura 30: Aspecto <i>ProxyProtocol</i>	81
Figura 31: Factoría abstracta de dispositivos JavaPOS	85
Figura 32: Factoría Abstracta de dispositivos JavaPOS (AspectJ)	86
Figura 33: Diagrama de estados del sistema TPV	86
Figura 34: Estructura del patrón <i>Estado</i> para el sistema TPV.....	87
Figura 35: Clases <i>Producto</i> y <i>CatalogoProductos</i>	96
Figura 36: Interfaz gráfica del sistema TPV	102
Figura 37: Ventana de presentación	109
Figura 38: Directorio de la distribución de Java.....	110
Figura 39: Directorio de instalación de AspectJ.....	110
Figura 40: Proceso de copia de ficheros.....	111
Figura 41: Ventana de fin de instalación	111
Figura 42: Plug-in instalados en Eclipse	114
Figura 43: Asociación de los archivos <i>.aj</i>	114
Figura 44: Asociación de los archivos <i>.java</i>	115
Figura 45: Personalización de la perspectiva actual.....	115
Figura 46: Menú <i>Archivo/Nuevo</i>	116
Figura 47: Opciones del compilador	116
Figura 48: El editor no reconoce las palabras clave de AspectJ.....	117

Figura 49: Opciones del editor Java	117
Figura 50: Opciones generales del compilador de AspectJ	118
Figura 51: Opciones del compilador AspectJ para un proyecto	119
Figura 52: Ventana <i>Nuevo proyecto</i>	119
Figura 53: Añadir aspectjrt.jar al nuevo proyecto	120
Figura 54: Ventana <i>importar ficheros</i> a un proyecto	120
Figura 55: Ventana de configuración de la ejecución	121
Figura 56: Mensajes impresos por la consola.....	121
Figura 57: Compilador AspectJ 1.0.x	122
Figura 58: Compilador AspectJ 1.1.x	122
Figura 59: Compilación incremental en <i>ajbrowse</i>	125
Figura 60: Classpath y Main class en <i>ajbrowse</i>	126
Figura 61: Interfaz de <i>ajbrowse</i>	126

Índice de Tablas

Tabla 1: Correspondencia Lenguajes base/Lenguajes orientados a aspectos.....	13
Tabla 2: Lenguajes orientados a aspectos y el tipo de entretejido que soportan	16
Tabla 3: Lenguajes de dominio específico	17
Tabla 4: Lenguajes de propósito general	18
Tabla 5: Ejemplos de signaturas	30
Tabla 6: Sintaxis de los puntos de corte según la categoría de los puntos de enlace	31
Tabla 7: Ejemplos de puntos de corte basados en la categoría de los puntos de enlace	31
Tabla 8: Sintaxis y semántica de los puntos de corte basados en flujo de control	32
Tabla 9: Ejemplos de puntos de corte basados en flujo de control.....	32
Tabla 10: Sintaxis y semántica de los puntos de corte basados en localización	33
Tabla 11: Ejemplos de puntos de corte basados en localización de código	33
Tabla 12: Sintaxis y semántica de los puntos de corte basados en objetos	34
Tabla 13: Ejemplos de puntos de corte basados en objetos.....	34
Tabla 14: Objetos <i>this</i> y <i>target</i> según la categoría del punto de enlace	35
Tabla 15: Argumentos según la categoría del punto de enlace	35
Tabla 16: Ejemplos de puntos de corte basados en argumentos.....	36
Tabla 17: Ejemplos de puntos de corte basados en condiciones	36
Tabla 18: Problemas de diseño & aspectos que los resuelven	54
Tabla 19: Beneficios de implementar patrones GoF con AspectJ	57
Tabla 20: Número de clases y aspectos en el sistema TPV	102
Tabla 21: Número de líneas de código del sistema TPV.....	102
Tabla 22: Tamaño del fichero <i>.jar</i>	103
Tabla 23: Opciones del comando <i>ajc</i>	124
Tabla 24: Opciones del comando <i>ajdoc</i>	128
Tabla 25: Opciones del comando <i>ajdb</i>	129

1. Introducción

Contexto

La separación de intereses (*separation of concerns*) consiste en descomponer un sistema en módulos independientes de forma que cada uno de ellos realice una función específica. Los diferentes paradigmas de programación: funcional, procedural y sobre todo el orientado a objetos, ofrecen potentes mecanismos para separar intereses, sobre todo los relacionados con la lógica del negocio de la aplicación, pero no ofrecen tan buenos resultados a la hora de tratar otros intereses que no pueden ser encapsulados en una única entidad puesto que “atraviesan” diferentes partes del sistema. A este tipo de intereses se les denomina intereses transversales (*crosscutting concern*).

Recientemente ha surgido un nuevo paradigma de programación que intenta resolver el problema de modularizar los intereses transversales: la **programación orientada a aspectos** (POA). Este nuevo paradigma nos permite capturar los intereses que atraviesan el sistema en entidades bien definidas llamadas **aspectos**, consiguiendo así una clara separación de intereses, con las ventajas que eso supone: eliminación de código disperso y enredado y las implementaciones resultan más comprensibles, adaptables y reusables.

En la última década la POA ha madurado y hay disponibles lenguajes y métodos que permiten su aplicación práctica en el desarrollo de software. En todos los análisis sobre el futuro del software, los expertos consideran la POA como un área que ejercerá una gran influencia y que será ampliamente utilizada.

Objetivos

El objetivo de este proyecto es realizar un estudio y análisis en profundidad del paradigma orientado a aspectos, desde la perspectiva del lenguaje AspectJ, el lenguaje orientado a aspectos más ampliamente utilizado. Para ello, se abordará como caso práctico el desarrollo de una aplicación de terminal de punto de venta.

Puesto que son pocos los libros sobre programación orientada a aspectos, y en concreto, sobre AspectJ existentes en la actualidad, y ninguno está traducido al español, otro de los objetivos de este proyecto será elaborar un documento que sirva para iniciarse en la POA mediante el uso del lenguaje AspectJ.

Metodología

En este apartado se va a comentar el método de trabajo seguido para la elaboración de este proyecto. En primer lugar se llevo a cabo una etapa de documentación acerca de la POA en la que se leyeron varios artículos [2, 6, 35, 36, 37] entre los que destaca el escrito por Gregor Kiczales [6], una de las personas que más ha contribuido al desarrollo de este nuevo paradigma, y que establece las bases de la programación orientada a aspectos.

Una vez comprendidos los fundamentos del nuevo paradigma se comenzó a profundizar en el lenguaje orientado a aspectos AspectJ, a través de los múltiples artículos publicados por el grupo Xerox PARC [4] y entre los que destacamos [18, 33, 34, 42]. Además se manejó el texto *Mastering AspectJ* [24] donde se describe con detalle todo lo relativo a este lenguaje.

Tras la adquisición de cierta destreza en el uso de este lenguaje, comenzó la etapa práctica de este proyecto, consistente en implementar el ejemplo de un sistema terminal punto de venta descrito en [28] aplicando el diseño orientado aspectos. La mayor parte del trabajo consistió en implementar en AspectJ los patrones GoF [21] utilizados durante el diseño del sistema, según se especifica en [22]. Una vez finalizada la aplicación se procedió a analizar los beneficios de la implementación con aspectos, frente la implementación clásica con Java. La última fase del proyecto consistió en la elaboración de esta memoria.

Organización del documento

En este primer capítulo se presenta el contexto y los objetivos del proyecto. En el Capítulo 2 se realiza un estudio del paradigma orientado a aspectos: fundamentos, lenguajes de programación, estado actual, etc. En el Capítulo 3 se profundiza en el lenguaje orientado aspectos AspectJ, describiendo en detalle su especificación. En el Capítulo 4 se recogen los detalles de diseño de la aplicación terminal punto de venta con aspectos. En el Capítulo 5 se recogen las conclusiones y los posibles trabajos futuros. Además, este documento va acompañado de tres anexos. En el Anexo I se encuentra la sintaxis de AspectJ, el Anexo II es un manual de instalación de AspectJ y el Anexo III es un tutorial de las herramientas de AspectJ: compilador, navegador, depurador y generador de documentación.

2. Programación orientada aspectos (POA)

2.1. Introducción

Los continuos avances en la ingeniería del software han ido incrementando la capacidad de los desarrolladores de software para descomponer un sistema en módulos independientes cada uno con una función bien definida, esto es, facilitar la separación de intereses [1] (*separation of concerns*¹, SOC)

Dentro de estos avances, quizás el más importante en estas dos últimas décadas ha sido la aparición de la *programación orientada a objetos* (POO). El paradigma orientado a objetos proporciona un potente mecanismo para separar intereses, sobre todo aquellos relacionados con la lógica del negocio de la aplicación, pero presenta dificultades a la hora de modelar otros intereses que no pueden ser encapsulados en una única entidad o clase, ya que afectan a distintas partes del sistema.

En [2] encontramos el siguiente ejemplo de un sistema editor de figuras cuyo diagrama de clases se muestra en la Figura 1.

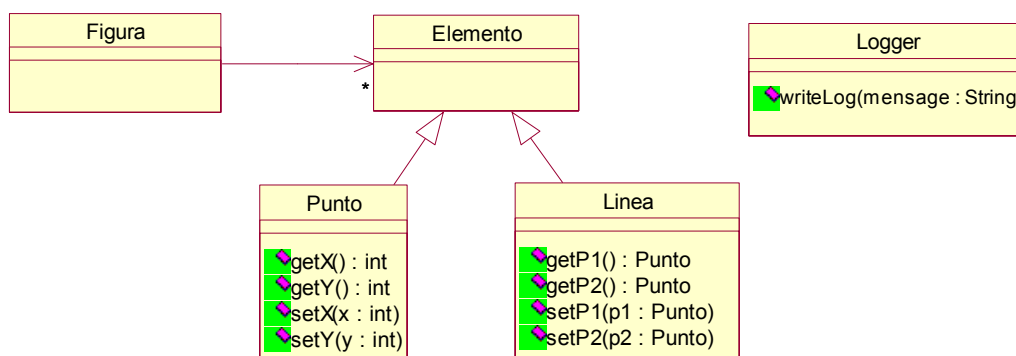


Figura 1: Diagrama de clases del sistema editor de figuras

Se desea disponer de un fichero histórico en el que se registren los cambios de posición de los elementos que componen una figura. Para ello, se tendrá que llamar al método `writeLog(String)` de la clase `Logger` para insertar una línea en el histórico, cada vez que se ejecute un método `set` de las clases `Punto` y `Linea`, como muestra el siguiente fragmento de código:

```

//Clase Punto
public void setX(int i) {
    x = i;
    Logger.writeLog("Cambio posición Punto: "+toString());
}

public void setY(int i) {
    y = i;
    Logger.writeLog("Cambio posición Punto: "+toString());
}

//Clase Linea
public void setP1(Punto punto) {

```

¹ Hablaremos de *concern* como interés o competencia indistintamente.

```

        p1 = punto;
        Logger.writeLog("Cambio posición Línea: "+toString());
    }

    /**
     * @param punto
     */
    public void setP2(Punto punto) {
        p2 = punto;
        Logger.writeLog("Cambio posición Línea: "+toString());
    }

```

Tras implementar este nuevo interés, las clases Punto y Línea además de realizar la función para la que fueron diseñadas, se encargan de realizar tareas de registro (*logging*). Cuando se dan este tipo de situaciones en las que un interés afecta a distintas partes de un sistema, se dice que el sistema ha sido “atravesado” (*crosscutting*) y a este tipo de intereses se les denomina **intereses o competencias transversales** (*crosscutting concern*). Las competencias transversales suelen estar relacionadas con la funcionalidad secundaria del sistema y ser de carácter no funcional. Pueden ir desde cuestiones de alto nivel como la seguridad o la calidad de los servicios ofrecidos, hasta cuestiones de más bajo nivel como pueden ser la sincronización, persistencia de datos, gestión de memoria, manejo de errores, logging, restricciones de tiempo, etc.

Las consecuencias directas de la existencia de competencias transversales son:

- **código disperso** (*scattered code*): el código que satisface una competencia transversal está esparcido por distintas partes del sistema. Podemos distinguir dos tipos de código esparcido:
 - bloques de código duplicados: cuando los mismos bloques de código aparecen en distintas partes del sistema.
 - bloques de código complementarios: cuando las distintas partes de una incumbencia son implementadas por módulos diferentes.
- **código enredado** (*tangled code*): una clase o módulo además de implementar su funcionalidad principal debe ocuparse de otras competencias.

El código disperso y enredado es un código difícil de reutilizar, mantener y evolucionar y de pobre trazabilidad. Estos efectos hacen que disminuya la calidad del software diseñado y se reduzca la productividad. De ahí la necesidad de nuevas técnicas que nos ayuden a conseguir una separación de intereses clara, para así reducir la complejidad del sistema a implementar y mejorar aspectos de calidad como la adaptabilidad, extensibilidad, mantenibilidad y reutilización.

En los últimos años han ido surgiendo distintas técnicas o paradigmas que intentan solucionar el problema de modularizar las competencias transversales como por ejemplo la *programación adaptativa*, la *programación subjetiva*, la *transformación de programas*, los *filtros composicionales*..., de las cuales la más popular y que con el tiempo ha ganado muchos adeptos es la **programación orientada a aspectos** (POA).

La POA es un nuevo paradigma de programación que permite capturar las competencias que atraviesan el sistema (competencias transversales) en entidades bien definidas llamadas **aspectos**, consiguiendo una clara separación de intereses, eliminando así el código disperso y enredado y los efectos negativos que este supone. La POA no sustituye al paradigma base con el que se desarrolla el sistema, sino que está un nivel de abstracción por encima. Además, no es exclusiva de la POO, sino que se

puede aplicar a otros paradigmas, por ejemplo a la programación procedural y funcional.

Teniendo en cuenta que los grandes avances producidos en la ingeniería del software siempre han tenido lugar con la aparición de nuevas formas de descomposición de los sistemas, es razonable pensar que el surgimiento de la POA puede ser un gran paso en la evolución de la ingeniería del software.

Un poco de historia

El grupo *Demeter* [3] fue el primero en usar ideas orientadas a aspectos. Su trabajo se centraba en la programación adaptativa, que en la actualidad se considera una versión inicial de la POA. En 1995 dos miembros de este 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 este informe se califica la separación de intereses como uno de los problemas más importantes a resolver en la ingeniería del software. Ese mismo año, el grupo Demeter publicó la primera definición de aspecto: “*un aspecto es una unidad que se define en términos de información parcial de otras unidades*”.

El grupo *Xerox Palo Reserach Center* [4], 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 [6]: “*un aspecto es una unidad modular que se dispersa por la estructura de otras unidades funcionales...*”. El grupo de investigación que dirige Gregor Kiczales, creó *AspectJ*, una de las primeras implementaciones de la POA de propósito general basada en Java. En diciembre del 2002, el grupo Xerox PARC transfirió el proyecto AspectJ a la comunidad open-source *Eclipse.org* [5].

En la actualidad la POA se encuentra en su fase de adolescencia y constantemente surgen nuevas herramientas orientadas a aspectos, nuevos ámbitos donde aplicar los aspectos, etc. Se asemeja a la situación en la que se encontraba la POO hace quince años.

2.2. Fundamentos de la POA

Para establecer las bases de la POA, Gregor Kiczales [6] consideró que los lenguajes OO, los lenguajes procedurales y los funcionales formaban parte de una misma familia de lenguajes, los **lenguajes de procedimiento generalizado** (*generalized-procedure languages*), ya que sus mecanismos de abstracción y composición tienen una raíz común en forma de un procedimiento generalizado. Para los lenguajes OO el procedimiento generalizado es una clase, para la programación funcional una función, para la procedural un procedimiento. Gregor Kiczales [6] clasifica las propiedades a implementar usando lenguajes de procedimiento generalizado en dos tipos:

- **un componente:** puede encapsularse claramente dentro de un procedimiento generalizado. Los componentes son unidades de descomposición funcional del sistema.

- **un aspecto:** no puede encapsularse claramente en un procedimiento generalizado. Suelen ser propiedades que afectan al rendimiento o a la semántica de los componentes.

Basándose en la clasificación anterior define el objetivo principal de la POA [6]:

“proporcionar un marco de trabajo que permita al programador separar claramente componentes y aspectos a través de mecanismos que hagan posible abstraerlos y componerlos para producir el sistema global.”.

y proporciona una definición formal del término aspecto [6]:

“un aspecto es una unidad modular que se dispersa por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de implementación es una unidad modular del programa que aparece en otras unidades modulares del programa.”

De manera más informal podemos hablar de los aspectos como aquellas propiedades o conceptos que atraviesan el sistema (*crosscutting concern*), no pudiendo ser encapsulados en una única entidad y dispersándose por todo el código. Los aspectos son la unidad básica de la POA. Ejemplos de aspectos son los relacionados con la gestión de memoria, sincronización, manejo de errores, etc.

En la Figura 2 podemos ver la estructura de una aplicación implementada mediante un lenguaje de procedimiento generalizado (LPG) y la estructura de la misma aplicación implementada mediante POA. Para la construcción de dicha aplicación se necesitan tener en cuenta cuestiones de sincronización y gestión de memoria. En la figura se distinguen mediante trazas de distinta forma y color, el código relativo a la funcionalidad básica de la aplicación, el código de sincronización y el código para gestionar la memoria.



Figura 2: Comparación LPG & POA

En la implementación tradicional, con un LPG, el código de sincronización y el de gestión de memoria se entremezclan con el código que implementa la funcionalidad básica de la aplicación, dando lugar a un código disperso y enredado. En cambio, en la implementación mediante POA, se consigue una separación de intereses completa, consiguiendo encapsular el código de sincronización y el código para gestionar la

memoria en entidades separadas, dando lugar a un programa más modular, donde cada módulo realiza una función bien definida.

Diseño orientado aspectos

El desarrollo de un sistema usando orientación a aspectos se suele dividir en tres etapas:

Etapla 1: Identificar competencias/intereses (*concern*)

Consiste en descomponer los requisitos del sistema en competencias y clasificarlas en:

- **competencias básicas** (*core-concern*): las que están relacionadas con la funcionalidad básica del sistema, de carácter funcional. Son las que Gregor Kiczales denomina *componentes* [6].
- **competencias transversales** (*crosscutting-concern*): las que afectan a varias partes del sistema, relacionadas con requerimientos no funcionales del sistema, normalmente de carácter no funcional. Son las que Gregor Kiczales denomina *aspectos* [6]

Etapla 2: Implementar competencias/intereses

Consiste en implementar cada interés independientemente:

Para implementar las competencias básicas usaremos el paradigma que mejor se ajuste a ellas (POO, programación procedural o funcional), y dentro del paradigma, el lenguaje que mejor satisfaga las necesidades del sistema y de los desarrolladores (Java, Lisp...). A este lenguaje lo denominaremos **lenguaje base**.

Para implementar las competencias transversales usaremos uno o varios **lenguajes orientados a aspectos**, de propósito específico o general, encapsulando cada competencia en unidades llamadas aspectos. Estos lenguajes orientados a aspectos deben ser compatibles con el lenguaje base para que los aspectos puedan ser combinados con el código que implementa la funcionalidad básica y así obtener el sistema final. Normalmente los lenguajes orientados a aspectos suelen ser extensiones del lenguaje base, como es el caso de AspectJ y AspectC, pero también puede ser lenguajes totalmente independientes.

Lenguaje base	Lenguaje de aspectos
Java	- AspectJ - AspectWerkz
C/C++	- AspectC - AspectC++
SmallTalk	- AspectS - Apostle
Python	- Pythius

Tabla 1: Correspondencia Lenguajes base/Lenguajes orientados a aspectos

Etapla 3: Componer el sistema final

Este proceso se conoce como **entretejido** (*weaving*) o **integración**. Consiste en combinar los aspectos con los módulos que implementan la funcionalidad básica del sistema dando lugar al sistema final. El módulo encargado de realizar este proceso recibe el nombre de **tejedor de aspectos** (*weaver*) y hace uso de unas reglas de entretejido para llevar a cabo el proceso.

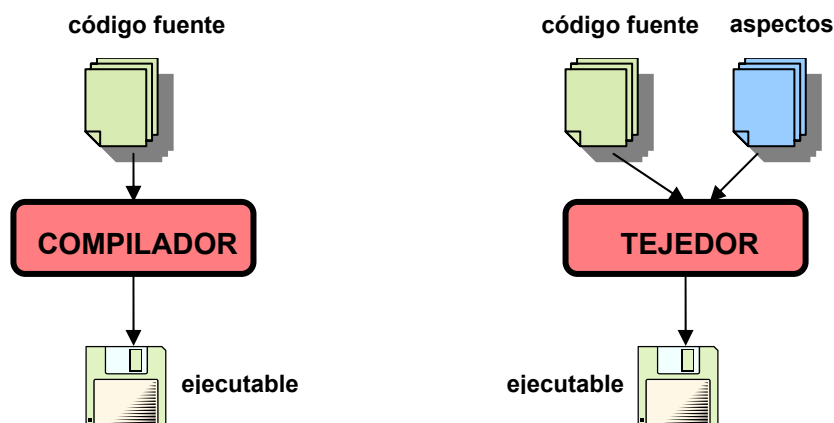


Figura 3: Generar ejecutable: enfoque tradicional & POA

En la Figura 3 podemos ver la diferencia existente en la compilación, entre el desarrollo tradicional y el desarrollo orientado aspectos.

2.3. Lenguajes orientados a aspectos

Como para el resto de los paradigmas de programación, un lenguaje que implemente el paradigma orientado a aspectos costa de dos partes:

- Una especificación del lenguaje que describe las construcciones y sintaxis del lenguaje.
- Una implementación del lenguaje que verifica que el código se ajusta a la especificación del lenguaje y convierte el código a un formato que la máquina pueda ejecutar.

Especificación del lenguaje

Un lenguaje orientado a aspectos debe proporcionar construcciones para especificar las reglas de entretejido que guiarán al tejedor de aspectos en su tarea de integrar los aspectos con el código principal de la aplicación. Los lenguajes orientados a aspectos suelen basar sus **reglas de entretejido** en el concepto de **punto de enlace** (*join point*). Un punto de enlace es un lugar bien definido en la ejecución de un programa. Mediante las reglas de entretejido indicaremos aquellos puntos de enlace que se ven afectados por cierto aspecto.

Las construcciones del lenguaje para definir las reglas de entretejido deben ser potentes y expresivas, permitiendo expresar desde reglas muy específicas que involucren a uno o pocos puntos de enlace, hasta reglas muy genéricas que engloben a un gran número de puntos. Estos mecanismos y construcciones varían de unos lenguajes a otros. En el capítulo siguiente profundizaremos en el caso particular de AspectJ, que

utiliza tres elementos para definir estas reglas: **puntos de enlace** (*join point*), **puntos de corte** (*point cut*) y **avisos** (*advice*).

Implementación del lenguaje

La implementación de un lenguaje orientado aspectos debe proporcionar una entidad, el *tejedor de aspectos*, que se encargue de integrar los aspectos con el código base, en un proceso que se denomina **entretejido** (*weaving*). Existen dos tipos diferentes de entretejido:

Entretejido estático

El tejedor genera un nuevo código fuente como resultado de integrar el código de los aspectos 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. Es el entretejido usado por AspectJ 1.0.x.

Otra posibilidad es que el entretejido tenga lugar a nivel de byte-code, es decir, el compilador de AspectJ entreteje los aspectos en los ficheros .class de nuestra aplicación, generando los .class de salida correspondientes. Es el entretejido usado por AspectJ 1.1.x.

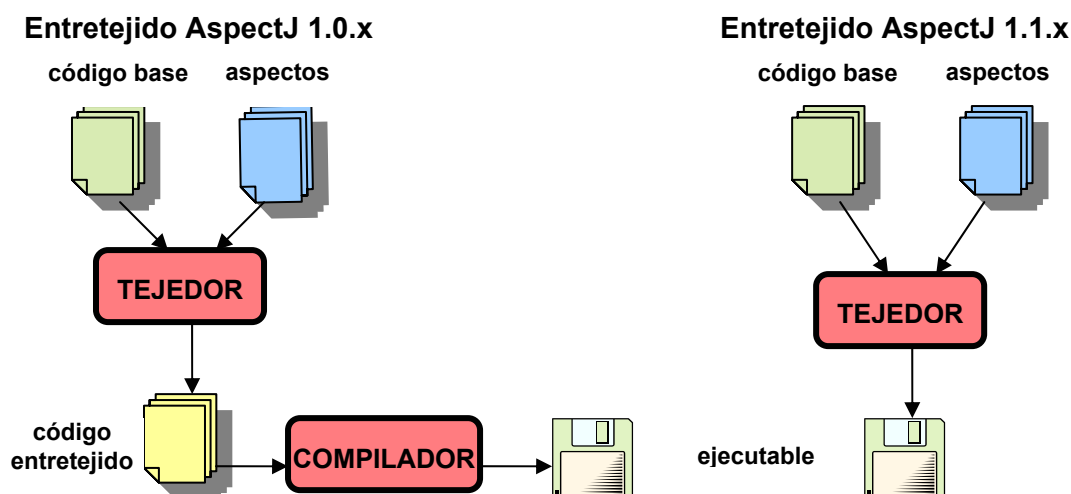


Figura 4: Entretejido AspectJ 1.0.x / Entretejido AspectJ 1.1.x

Las ventajas del entretejido estático son:

- el entretejido se realiza en tiempo de compilación, por lo que se evita un impacto negativo en la eficiencia de la aplicación al no existir sobrecarga en la ejecución
- 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

Las desventajas del entretejido estático son:

- al ser muy difícil identificar los aspectos en el código entretejido, es casi imposible poder modificarlos en tiempo de ejecución, permaneciendo fijos durante todo el ciclo de vida del programa. Tampoco es posible añadir o eliminar aspectos dinámicamente.

Entretejido dinámico

Para poder realizar entretejido dinámico es necesario que tanto los aspectos como sus reglas de entretejido sean modeladas como objetos y estén disponibles en tiempo de ejecución. La implementación del lenguaje orientado a aspectos proporcionará un entorno de ejecución controlado que cada vez que se llega a un punto de enlace ejecutará el código del aspecto o aspectos que lo afectan. Mediante este tipo de tejedores si es posible añadir, modificar y eliminar aspectos en tiempo de ejecución.

Un ejemplo de este tipo de tejedores es el de AOP/ST [7] que utiliza la herencia para añadir el código de los aspectos a las clases a las que afectan. Otro ejemplo de tejedor dinámico es el JPAL [8] (*Junction Point Aspect Language*) que utiliza una entidad llamada *Administrador de programas de aspectos* que se encarga de registrar nuevos aspectos, eliminarlos o modificar los existentes y de invocar a los métodos de los aspectos registrados. En AspectJ todavía no es posible realizar entretejido dinámico, pero esta previsto añadir esta posibilidad en futuras versiones del lenguaje.

La ventaja principal del entretejido dinámico frente al estático es que permite añadir, modificar o eliminar aspectos dinámicamente, en tiempo de ejecución, aumentando las posibilidades del programador. Por otra parte, las desventajas de usar entretejido dinámico son:

- Existe sobrecarga en la ejecución del programa, al realizar el entretejido en tiempo de ejecución, por lo que disminuye su rendimiento.
- Disminuye la fiabilidad de la aplicación, ya que se pueden producir errores como borrar el comportamiento de un aspecto que posteriormente será invocado.
- Son más difíciles de implementar que los tejedores estáticos y consumen muchos más recursos.

La tendencia es implementar un tejedor estático si el lenguaje base es estático e implementar un tejedor dinámico si el lenguaje base es dinámico. Aunque todas las combinaciones son posibles:

Lenguaje de aspectos	Lenguaje base	Entretejido
AspectJ	Java	Estático
AspectC	C	Estático
AspectWerkz	Java	Estático/Dinámico
AspectS	SmallTalk	Estático

Tabla 2: Lenguajes orientados a aspectos y el tipo de entretejido que soportan

La implementación del lenguaje además del tejedor, también puede proporcionar otro tipo de herramientas como un depurador, un generador de documentación...

Algunos lenguajes orientados a aspectos

Una vez vista la anatomía que debería tener un lenguaje orientado a aspectos comentamos brevemente algunos de los lenguajes de aspectos más conocidos existentes en la actualidad, que podemos clasificar en dos categorías:

Lenguajes de dominio específico

Esta clase de lenguajes son diseñados para trabajar sobre determinado tipo de aspectos, como por ejemplo sincronización, persistencia, manejo de errores, etc., pero no pueden tratar con aquellos aspectos para los que no fueron diseñados. Normalmente tienen un nivel de abstracción mayor que el del lenguaje base e imponen restricciones en la utilización del lenguaje base para evitar que los aspectos se programen en ambos lenguajes, lo que podría dar lugar a conflictos.

Los primeros lenguajes orientados aspectos fueron de dominio específico: AML [19] que es una extensión de *Matlab* para procesamiento de matrices esparcidas y RG [20] que es un lenguaje para el procesamiento de imágenes. El lenguaje COOL (*COOrdination Language*) es un lenguaje para sincronización en [9]. Su lenguaje base es Java, pero con algunas restricciones, ya que se eliminan los métodos *notify*, *notifyAll*, y *wait* y la palabra reservada *synchronized* para evitar que el lenguaje base implemente cuestiones de sincronización y entre en conflicto con la implementación realizada por COOL. El lenguaje RIDL (*Remote Interaction and Data transfers Language*) es un lenguaje para transferencia de datos remotos también usado en [9].

Lenguaje de aspectos	Dominio
COOL	Sincronización
RIDL	Distribución
AML	Calculo de matrices dispersas
RG	Procesamiento de imágenes

Tabla 3: Lenguajes de dominio específico

Lenguajes de propósito general.

Son diseñados para trabajar sobre cualquier tipo de aspecto. No ponen restricciones al lenguaje base y normalmente tienen el mismo nivel de abstracción que el lenguaje base. Suelen ser extensiones del lenguaje base, por lo que tienen su mismo juego de instrucciones, ampliado con las construcciones necesarias para la definición y manejo de los aspectos. Desde el punto de vista del desarrollador de software será menos costoso aprender un lenguaje de propósito general que varios de dominio específico, uno para cada aspecto a implementar en el sistema.

Uno de los primeros lenguajes de propósito general fue *AspectJ*. Surge como resultado de años de investigación del grupo Xerox PARC, dirigido por Gregor Kiczales. La semántica introducida por este lenguaje (puntos de enlace, puntos de corte, avisos) ha servido de base para el resto de lenguajes que han ido surgiendo con el paso de los años. Otros lenguajes de propósito general son *Java Aspect Component* [16], *AspectWerkz* [17], *AspectC* [10], *AspectC++* [11], *AspectS* [12], *Apostle* [13], *Pythius* [14] y *AspectR* [15]. La Tabla 4 muestra el lenguaje base para cada uno de estos lenguajes.

Lenguaje de aspectos	Lenguaje base
- AspectJ - Java Aspect Component. - AspectWerkz	Java
- AspectC	C
- AspectC++	C++
- AspectS - Apostle	SmallTalk
- Pythius	Python
- AspectR	Ruby

Tabla 4: Lenguajes de propósito general

2.4. Estado actual de la POA

Algunos desarrolladores de software son reticentes a usar este nuevo paradigma quizás por su carácter novedoso. Si bien es cierto que aprender algo nuevo supone cierto esfuerzo, la curva de aprendizaje de la POA una vez dominados los conceptos orientados a objetos es bastante menor. Esto se debe a que la POA es considerado el paso natural después de la POO, ya que permite aumentar en nivel de abstracción, pasando del espacio bidimensional ofrecido por la POO, a un espacio tridimensional que nos permite identificar los intereses transversales en esta nueva dimensión, consiguiendo implementarlos de una forma modular resolviendo así el problema de la separación de intereses.

El uso de la POA se extiende día a día, aunque a nivel empresarial todavía no se considera como una opción a tener en cuenta, quizás porque se desconocen las grandes posibilidades que ofrece y por su falta de estandarización, aunque a este respecto parece que comienzan a surgir movimientos de unificación como el promovido por los creadores de AspectJ y AspectWerkz (su competidor directo) que van a unir sus fuerzas para crear un único framework Java para POA [26]. Podríamos comparar el estado actual de la POA como el de la POO hace quince años.

Parece que esta década puede representar la madurez definitiva de la POA. Ya empiezan a surgir guías de diseño orientado a aspectos que incluyen malos usos de la POA, soluciones elegantes orientadas a aspectos para determinados problemas y son muchos los proyectos que están en marcha y que se basan en ideas orientadas a aspectos. Como ejemplo de este fenómeno en auge tenemos la implementación de los patrones de Gamma [21] con AspectJ, realizada por Gregor Kiczales [22] y que analizaremos más profundamente en apartados posteriores y la aparición del framework *Spring* que se basa en ideas orientadas aspectos, y que está realizando una importante contribución a la popularización de este joven paradigma de programación.

A continuación se listan los beneficios más importantes que ofrece su uso [23]:

- **Un diseño más modular.** Cada competencia se implementa de forma independiente con un acoplamiento mínimo, eliminando el código duplicado y dando lugar un código más ordenado, fácil de entender y mantener.
- **Mejora la trazabilidad** puesto que cada módulo tiene sus responsabilidades bien definidas.

- **Fácil evolución del sistema.** Si se quiere añadir un nuevo aspecto, no será necesario modificar el código que implementa la funcionalidad básica del sistema, reduciendo el tiempo de respuesta a nuevos requisitos.
- **Aumenta la reutilización.** Cada competencia transversal está encapsulada en un aspecto y los módulos principales del sistema no conocen su existencia, por lo que el acoplamiento es muy bajo, lo que facilita la reutilización.
- **Reduce el tiempo de mercado.** La clara separación de intereses permite a los desarrolladores trabajar con mayor destreza y rapidez, mejorando la productividad. La reutilización de código también reducirá el tiempo de desarrollo. La fácil evolución permite satisfacer rápidamente nuevos requisitos. Todo esto hace que el desarrollo y salida al mercado del producto se realice en un periodo de tiempo más reducido.
- **Reduce costes de futuras implementaciones.** Para añadir un nuevo aspecto ya no hay que modificar el código base con lo que se reduce el coste de implementar nuevas competencias transversales. Esto permite que los programadores estén más centrados en implementar la funcionalidad básica del sistema por lo que aumentará la productividad y la implementación de la funcionalidad básica también disminuirá.
- **Retrasar decisiones de diseño.** Puesto que con la POA es posible implementar las competencias transversales en módulos independientes, los desarrolladores del sistema pueden centrarse en los requisitos relacionados con la funcionalidad básica del sistema y posteriormente implementar las competencias transversales sin tener que modificar las clases principales del sistema. Por lo tanto, con la POA podemos llevar a la práctica uno de los principios principales de la programación extrema “*you aren’t gonna need it*” (YAGNI), traducido al español sería “*no lo vas a necesitar*” y viene a decir que no deberías añadir hoy código que sólo será usado por alguna característica que será necesaria mañana. Primero implementa la funcionalidad básica y en un futuro añade la funcionalidad secundaria del sistema.

3. AspectJ

3.1. Introducción

AspectJ es un lenguaje orientado a aspectos de propósito general, creado en 1998 por *Gregor Kiczales* y el grupo de investigación que dirige, el *Xerox PARC* [4]. Constituye una de las primeras implementaciones del paradigma orientado a aspectos. En diciembre del 2002, el proyecto AspectJ fue cedido por sus creadores a la comunidad open-source *Eclipse.org* [5].

Consiste en una extensión de Java para soportar la definición y manejo de aspectos, por lo que cualquier programa válido para Java, lo será también para AspectJ. Además, el compilador de AspectJ genera byte-codes compatibles con cualquier máquina virtual de Java. Implementar AspectJ como una extensión de Java hace que el lenguaje sea fácil de aprender, puesto que tan sólo añade unos pocos términos nuevos al lenguaje base (Java), y que conserve las características ventajosas de Java, como la independencia de la plataforma, que han hecho de este lenguaje OO, el más usado en la actualidad.

AspectJ es un lenguaje relativamente nuevo y por lo tanto en continua evolución. En el momento de realizar este proyecto, la última versión estable era AspectJ 1.2.1, que es la que se ha utilizado para el desarrollo de este trabajo. El uso de AspectJ nos permitirá llevar a la práctica el paradigma orientado a aspectos, cuya principal meta es conseguir una separación de intereses clara, encapsulando las competencias transversales (*crosscutting concern*) en módulos independientes llamados aspectos.

3.2. Especificación de AspectJ

La especificación de un lenguaje describe la sintaxis y la semántica de sus construcciones. En AspectJ, para expresar la funcionalidad de los aspectos se utiliza Java estándar, mientras que para especificar las reglas de entretejido (*weaving rules*) el lenguaje proporciona una serie de constructores. La implementación de las reglas de entretejido se suele llamar *crosscutting*, puesto que especifican la forma en que los aspectos “atraviesan” las clases de la aplicación principal. Existen dos tipos de *crosscutting*:

- **Crosscutting dinámico:** los aspectos afectan al comportamiento de la aplicación, modificando o añadiendo nuevo comportamiento. Es el tipo de *crosscutting* más usado en AspectJ. Los constructores de *crosscutting* dinámico son:
 - **Puntos de enlace** (*join point*): son puntos bien definidos en la ejecución de un programa. Por ejemplo, la llamada a un método, la lectura de un atributo, etc. Representan los lugares donde los aspectos añaden su comportamiento.
 - **Puntos de corte** (*cut point*): agrupan puntos de enlace y permiten exponer su contexto a los avisos. Por ejemplo, mediante un punto de corte podemos agrupar las llamadas a todos los métodos de cierta clase y exponer su contexto (argumentos, objeto invocador, objeto receptor).

- **Avisos** (*advice*): especifican el código que se ejecutará en los puntos de enlace que satisfacen cierto punto de corte, pudiendo acceder al contexto de dichos puntos de enlace.

Por lo tanto los puntos de corte indican *dónde* y los avisos *qué hacer*. Un ejemplo de crosscutting dinámico podría ser añadir cierto comportamiento tras la ejecución de ciertos métodos o sustituir la ejecución normal de un método por una ejecución alternativa.

- **Crosscutting estático**: los aspectos afectan a la estructura estática del programa (clases, interfaces y aspectos). Otra posibilidad es añadir advertencias (*warning*) o errores en tiempo de compilación. La función principal del crosscutting estático es dar soporte a la implementación del crosscutting dinámico. Los constructores de crosscutting estático son:
 - **Declaraciones inter-tipo** (*inter-type declarations*): permiten añadir miembros (campos, métodos o constructores) a clases, interfaces o aspectos de una aplicación.
 - **Declaraciones de parentesco** (*declare parents*): permiten especificar que ciertas clases implementan una interfaz o extienden nuevas clases.
 - **Declaraciones en tiempo de compilación** (*compile-time declarations*): permiten añadir advertencias o errores en tiempo de compilación para notificar ciertas situaciones que deseamos advertir o evitar, por ejemplo la utilización de cierto método en desuso.
 - **Declaraciones de precedencia** (*declare precedence*): permiten especificar relaciones de precedencia entre aspectos.
 - **Excepciones suavizadas** (*softening exceptions*): permiten ignorar el sistema de chequeo de excepciones de Java, silenciando las excepciones que tienen lugar en determinados puntos de enlace, encapsulándolas en excepciones no comprobadas y relanzándolas.

Todos los constructores citados anteriormente (puntos de corte, avisos, y declaraciones) se encapsulan en un entidad llamada **aspecto** (*aspect*). Un aspecto es la unidad básica de AspectJ, al igual que una clase lo es de un lenguaje OO. Un aspecto también puede contener atributos, métodos y clases anidadas como una clase Java normal.

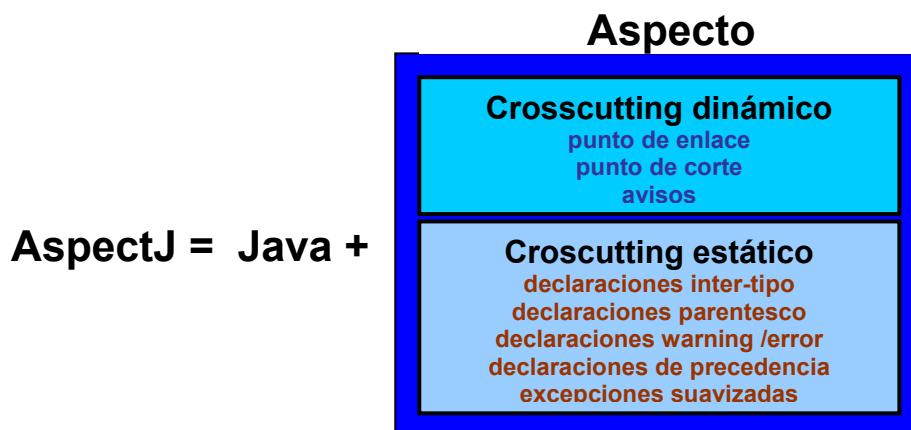


Figura 5: Especificación de AspectJ

La forma de proceder a la hora de implementar con AspectJ un comportamiento que afecta a distintas partes del sistema (competencia transversal) es la siguiente:

- Identificar aquellos puntos de enlace donde queremos añadir el comportamiento cruzado, agrupándolos con un punto de corte.
- Implementar el comportamiento transversal con un aviso, cuyo cuerpo será ejecutado cuando se alcance alguno de los puntos de enlace que satisfacen el punto de corte definido en el paso anterior.

Todo ello encapsulado en un aspecto, de forma que la implementación de la competencia transversal queda totalmente localizada en una sola entidad.

Como primera toma de contacto con AspectJ, se va a implementar el ejemplo comentado en la introducción del Capítulo 1: un sistema editor de figuras [2] en el que se desea disponer de un fichero histórico en el que se registren los cambios de posición de los elementos que componen una figura. Para satisfacer este interés, en la implementación tradicional con Java se tenía que insertar una llamada al método `writeLog(String)` en cada método `set` de las clases `Punto` y `Línea`, de forma que el código para satisfacer el interés se encontraba esparcido en varias clases. Podríamos decir que este interés atraviesa (*crosscutting*) las clases `Punto` y `Línea` lo que dificulta la separación de intereses, como se muestra en la Figura 6.

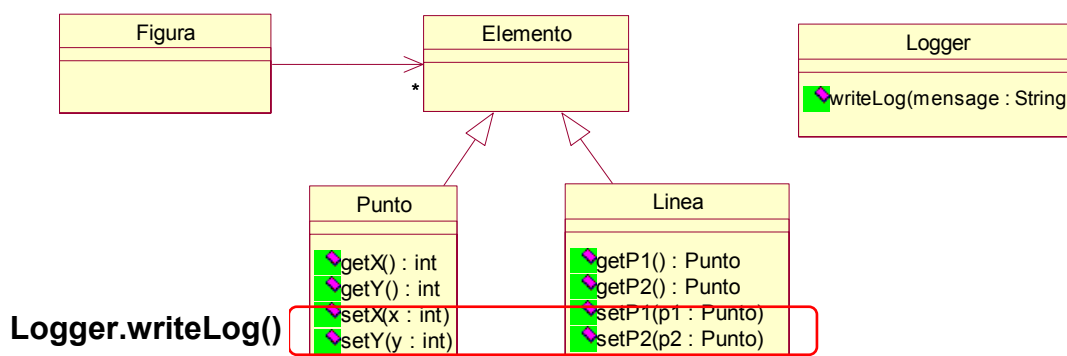


Figura 6: *Crosscutting* en el sistema editor de figuras

Vamos a aplicar los pasos descritos anteriormente para implementar una competencia transversal con AspectJ, para el caso del sistema editor de figuras:

1. Definir uno a varios puntos de corte (`pointcut`) que capture las llamadas a los métodos `set` de las clases `Punto` y `Línea`. Para ello usamos el descriptor de punto de corte `call` que captura la llamada a un método.

```

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

pointcut cambioPosicionLinea(Linea l):
    call( * Linea.set*(Punto) ) && target(l);
  
```

2. Crear dos avisos (`advice`) para cada uno de los puntos de corte anteriores que ejecuten el método `Logger.writeLog()`. Utilizamos un aviso de tipo **after** que se ejecuta después de la ejecución del punto de enlace capturado, es decir, que después de la ejecución de un método `set` se ejecutará el método `Logger.writeLog()`.

```

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

after(Linea l): cambioPosicionLinea(l)
{
    Logger.writeLog("Cambio posición Linea: "+l.toString());
}

```

Y todo ello encapsulado en un aspecto:

```

public aspect LogCambiosPosicion {

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

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

    pointcut cambioPosicionLinea(Linea l):
        call( * Linea.set*(Punto) ) && target(l);

    after(Linea l): cambioPosicionLinea(l)
    {
        Logger.writeLog("Cambio posición Linea: "+l.toString());
    }

}

```

Implementando el mecanismo de *log* (registro) como un aspecto, conseguimos que este no atravesase las clases `Punto` y `Linea`, quedando totalmente encapsulado en una única entidad, el aspecto `LogCambiosPosicion`, a diferencia de la solución OO, que implicaba añadir una llamada al método `Logger.writeLog` al final de cada método `set` de las clases `Punto` y `Linea`, provocando que el código encargado de satisfacer el interés quedara esparcido en varias clases.

Por lo tanto, usando la POA logramos encapsular el comportamiento cruzado del sistema editor de figuras, logrando una separación de intereses completa. En los siguientes apartados se analizarán cada uno de estos constructores, tanto estáticos como dinámicos, que forman parte de la especificación del lenguaje AspectJ [23, 24, 25, 43]. En el Anexo I de este documento se recoge la sintaxis formal de cada uno de las construcciones del lenguaje AspectJ.

3.2.1. Puntos de enlace

Los puntos de enlace son puntos bien definidos en la ejecución de un programa. No hay que confundirlos con posiciones en el código de un programa sino que son puntos sobre el flujo de ejecución de un programa. AspectJ soporta los siguientes tipos de puntos de enlace:

- **Llamada a un método:** un punto de enlace *llamada a método* ocurre cuando un método es invocado por un objeto. En el caso de los métodos estáticos no existe objeto invocador. Ejemplo:

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

Punto de enlace **llamada a método**

- **Llamada a un constructor:** un punto de enlace *llamada a constructor* ocurre cuando un constructor es invocado durante la creación de un nuevo objeto. Ejemplo:

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

Punto de enlace **llamada a constructor**

- **Ejecución de un método:** un punto de enlace *ejecución de método* engloba la ejecución del cuerpo de un método. Ejemplo:

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

Punto de enlace **ejecución de método**

- **Ejecución de un constructor:** un punto de enlace *ejecución de constructor* abarca la ejecución del cuerpo de un constructor durante la creación de un nuevo objeto. Ejemplo:

```
public Punto(int x, int y) {
    super();
    this.x = x;
    this.y = y;
}
```

Punto de enlace **ejecución de constructor**

- **Lectura de un atributo:** un punto de enlace *lectura de atributo* ocurre cuando se lee un atributo de un objeto dentro de una expresión. Ejemplo:

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

Punto de enlace **lectura de atributo**

- **Escritura de atributo:** un punto de enlace *escritura de atributo* ocurre cuando se asigna un valor a un atributo de un objeto. Ejemplo:

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

Punto de enlace **escritura de atributo**

- **Ejecución de un manejador de excepciones:** un punto de enlace *ejecución de manejador* ocurre cuando un manejador es ejecutado. Ejemplo:

```
try{
    .....
} catch(IOException e){
    e.printStackTrace();
}
```

Punto de enlace **ejecución de manejador**

- **Inicialización de clase:** un punto de enlace *inicialización de clase* ocurre cuando se ejecuta el inicializador estático de una clase específica, en el momento en que el cargador de clases carga dicha clase. Ejemplo:


```

public class Main {
    .....

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

```

Punto de enlace
inicialización de clase

- **Inicialización de objeto:** un punto de enlace *inicialización de objeto* ocurre cuando un objeto es creado. Comprende desde el retorno del constructor padre hasta el final del primer constructor llamado. En el siguiente ejemplo

```

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

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

```

Punto de enlace inicialización
de objeto

Si el constructor anterior es llamado, la llamada al constructor padre (`super`) no forma parte del punto de enlace.

- **Pre-inicialización de un objeto:** un punto de enlace *pre-inicialización de un objeto* ocurre antes de que el código de inicialización para una clase particular se ejecute. Comprende desde el primer constructor llamado hasta el comienzo del constructor padre. Es raramente usado y suele abarcar las instrucciones que representan los argumentos del constructor padre. Ejemplo:

```

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

```

Punto de enlace
pre-inicialización de
objeto

Si el constructor anterior es llamado, el punto de enlace sólo abarca la llamada al método `Cuenta.generarId()`.

- **Ejecución de un aviso:** un punto de enlace *ejecución de aviso*, comprende la ejecución del cuerpo de un aviso. Ejemplo:

```

after (Punto p): cambioPosicionPunto(p) {
    Logger.writeLog("...");
}

```

Punto de enlace
ejecución de aviso

Para comprobar los puntos de enlace de distinto tipo que tienen lugar en cierta aplicación se puede utilizar el siguiente aspecto que capturará todos los puntos de enlace y los imprimirá por la salida estándar.

```
public aspect TodosPuntosEnlace {
    before(): !within(TodosPuntosEnlace) {
        System.out.println(thisJoinPoint);
    }
}
```

Por ejemplo, para el siguiente fragmento de código:

```
public class Prueba {
    public void holaMundo() {
        System.out.println("Hola Mundo");
    }
    public static void main(String[] args) {
        Prueba p = new Prueba();
        p.holaMundo();
    }
}
```

los puntos de enlace capturados son:

```
staticinitialization(Prueba.<clinit>)
execution(void Prueba.main(String[]))
call(Prueba())
preinitialization(Prueba())
initialization(Prueba())
execution(Prueba())
call(void Prueba.holaMundo())
execution(void Prueba.holaMundo())
get(PrintStream java.lang.System.out)
call(void java.io.PrintStream.println(String))
Hola Mundo
```

Una vez conocidos los tipos de puntos de enlace y sus particularidades, pasamos a ver como identificarlos mediante los puntos de corte.

3.2.2. Puntos de corte

Los puntos de corte identifican o capturan una serie de puntos de enlace y permiten exponer su contexto a los avisos (*advice*). Forman parte de la definición de un aspecto, permitiéndonos especificar los puntos de enlace donde se aplicará dicho aspecto. Usando las construcciones ofrecidas por el lenguaje, podremos crear puntos de corte desde muy generales, que identifican a un gran número de puntos de enlace, a muy específicos, que identifican a un único punto de enlace. Existen puntos de corte anónimos y con nombre. Un punto de corte anónimo se define en el lugar donde se usa (en un aviso o en otro punto de corte) y no puede ser referenciado en otras partes del código, a diferencia de los puntos de corte con nombre que sí se pueden reutilizar.

Puntos de corte anónimos

La sintaxis de un punto de corte anónimo es la siguiente:

```
<pointcut> ::= { [!] designator [ && | || ] };
designator ::= designator_identifier(<signature>)
```

Un punto de corte anónimo consta de una o más expresiones (*designator*) que identifican una serie de puntos de enlace. Cada una de estas expresiones esta formada por un descriptor de punto de corte y una signatura. El descriptor de punto de corte especifica el tipo de punto de corte y la signatura indica el lugar donde se aplica. El siguiente ejemplo muestra un punto de corte anónimo asociado a un aviso de tipo *after* que consta de dos expresiones combinadas mediante el operador lógico AND. La primera expresión utiliza el descriptor de punto de corte *call* con su correspondiente signatura de método, para identificar las llamadas a los métodos *set* de la clase *Punto* y la segunda usa el descriptor *target* para exponer el contexto del punto de enlace.

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

Puntos de corte con nombre

La sintaxis de un punto de corte con nombre es la siguiente:

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

Un punto de corte con nombre consta del modificador de acceso (*access_type*), por defecto *package*, la palabra reservada *pointcut* seguida del nombre del punto de corte y de sus parámetros, que permiten exponer el contexto de los puntos de enlace a los avisos (*advice*). Por último, vienen las expresiones que identifican los puntos de enlace precedidas del carácter dos puntos “:”. Estas expresiones son idénticas a las utilizadas en los puntos de corte anónimos. Se pueden definir puntos de corte abstractos (*abstract*) dentro de un aspecto, de forma que sean los subaspectos de dicho aspecto los encargados de implementarlos. El siguiente ejemplo muestra un punto de corte con nombre equivalente al punto de corte anónimo anterior.

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

Tanto en los puntos de corte anónimos como en los puntos de corte con nombre, se pueden combinar las expresiones que identifican los puntos de enlace mediante los operadores lógicos ! (not), || (or) y && (and). Para más información sobre la sintaxis de los puntos de corte ver el Anexo I.

Signaturas: método, constructor, atributo y tipo

La sintaxis de los distintos tipos de signatura que pueden aparecer en un punto de corte son:

Signatura de un método

```
<access_type> <ret_val> <class_type>.<method_name>(<parameters>)  
[throws <exception>]
```

Algunos ejemplos de signatura de método son:

```
public void Punto.setX(int)  
public void Linea.setP1(Punto)
```

Signatura de un constructor

```
<access_type> <class_type>.new(<parameters>)[throws <exception>]
```

Las siguientes signaturas pertenecen a constructores:

```
public Punto.new(int,int)  
public Linea.new(Punto, Punto)
```

Signatura atributo

```
<access_type> <field_type> <class_type>.<field_name>
```

Algunos ejemplos de signatura de atributo son:

```
public int Punto.x  
public Punto Linea.punto1
```

Signatura tipo

```
<type>
```

Cuando en AspectJ hablamos de tipo nos referimos a una clase, interfaz, tipo primitivo o incluso un aspecto. Por lo tanto ejemplos de signatura de tipo podrían ser:

```
Punto  
IOException
```

Comodines y operadores lógicos

A la hora de expresar los puntos de enlace se pueden usar una serie de **comodines** para identificar puntos de enlace que tienen características comunes. El significado de estos comodines dependerá del contexto en el que aparezcan:

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

.. : el carácter dos puntos 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.

+ : el operador suma representa una clase y todos sus descendientes, tanto directos como indirectos.

Las expresiones que identifican los puntos de enlace se pueden combinar mediante el uso de los siguientes operadores lógicos:

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

exp1* && *exp2: operador lógico AND. La expresión compuesta se cumple cuando se cumple *exp1* y *exp2*.

! *exp* : operador de negación. La expresión compuesta se cumple cuando no se satisface *exp*.

La precedencia de estos operadores lógicos es la misma que en Java. Es recomendable el uso de paréntesis para hacer más legible las expresiones o para modificar la precedencia por defecto de los operadores. La Tabla 5 recoge algunos ejemplos de firmas de métodos, constructores, atributos y tipos:

Signatura	Elementos que identifica
<code>public void Punto.set*(int)</code>	Todos los métodos públicos de la clase <i>Punto</i> que empiezan por <code>set</code> , devuelven <code>void</code> y tienen un único parámetro de tipo <code>int</code> .
<code>public void Punto.set*(*)</code>	Todos los métodos públicos de la clase <i>Punto</i> que empiezan por <code>set</code> , devuelven <code>void</code> y tienen un único parámetro de cualquier tipo.
<code>public void Punto.set*(..)</code>	Todos los métodos públicos de la clase <i>Punto</i> que empiezan por <code>set</code> , devuelven <code>void</code> y tienen cualquier número y tipo de parámetros.
<code>public * Punto.*()</code>	Todos los métodos públicos de la clase <i>Punto</i> que no tienen parámetros.
<code>public * Punto.*(..)</code>	Todos los métodos públicos de la clase <i>Punto</i> que devuelven cualquier tipo y tienen un número y tipo de parámetros arbitrario.
<code>* Punto.*(..)</code>	Todos los métodos de la clase <i>Punto</i> .
<code>!public * Punto.*(..)</code>	Todos los métodos no públicos de la clase <i>Punto</i> .
<code>* Elemento+.*(..)</code>	Todos los métodos de clase <i>Elemento</i> y de sus subclases.
<code>* java.io.Reader.read(char[],..)</code>	Cualquier método <code>read</code> de la clase <code>java.io.Reader</code> que tenga un primer parámetro de tipo <code>char[]</code> y el resto de parámetros sea indiferente
<code>* *.*(..) throws IOException</code>	Todos los métodos que pueden lanzar la excepción <code>IOException</code> .
<code>Punto.new()</code>	Constructor por defecto de la clase <i>Punto</i>
<code>public Elemento.new(..)</code>	Todos los constructores públicos de la clase <i>Elemento</i> .
<code>public Elemento+.new(..)</code>	Todos los constructores públicos de la clase

	Elemento y de sus subclases.
<code>(Elemento+ && ;Elemento).new(..)</code>	Todos los constructores de las subclases de Elemento, sin incluir los de la propia clase Elemento.
<code>Punto Elemento</code>	Tipo Punto o tipo Elemento.
<code>Elemento+</code>	El tipo Elemento y todos sus descendientes
<code>*Punto</code>	Cualquier tipo cuyo nombre empiece por la palabra Punto.
<code>!Punto</code>	Todos los tipos excepto el tipo Punto.
<code>java.*.Punto</code>	Todos los tipos de nombre Punto dentro de cualquier subpaquete directo del paquete java.
<code>Java..*Model</code>	Todos los tipos cuyo nombre termina en Model, pertenecientes al paquete java o a alguno de sus subpaquetes, directos o indirectos.
<code>com.company..*</code>	Todos los tipos del paquete com.company o de alguno de sus subpaquetes, es decir, cualquier tipo que empiece con la ruta com.company
<code>private Punto Linea.*</code>	Todos los atributos privados de tipo Punto de la clase Linea
<code>* Linea.punto*</code>	Todos los atributos de la clase Linea cuyo nombre empieza por la palabra punto
<code>* Linea.*</code>	Todos los atributos de la clase Linea
<code>public !final *.*</code>	Todos los atributos no finales públicos de cualquier clase
<code>!public * java...*</code>	Todos los atributos no públicos de las clases del paquete Java y de todos sus subpaquetes directos o indirectos.

Tabla 5: Ejemplos de firmas

Tipos de puntos de corte

AspectJ proporciona una serie de descriptores de puntos de corte que nos permiten identificar grupos de puntos de enlace que cumplen diferentes criterios. Estos descriptores se clasifican en diferentes grupos:

- **Basados en las categorías de puntos de enlace:** capturan los puntos de enlace según la categoría a la que pertenecen: llamada a método (`call`), ejecución de método (`execute`), lectura de atributo (`get`), etc.
- **Basados en el flujo de control:** capturan puntos de enlace de cualquier categoría siempre y cuando ocurran en el contexto de otro punto de corte. Estos descriptores son `cflow` y `cflowbelow`.
- **Basados en la localización de código:** capturan puntos de enlace de cualquier categoría que se localizan en ciertos fragmentos de código, por ejemplo, dentro de una clase o dentro del cuerpo de un método. Estos descriptores son `within` y `withincode`.
- **Basados en los objetos en tiempo de ejecución:** capturan los puntos de enlace cuyo objeto actual (`this`) u objeto destino (`target`) son de un cierto tipo. Además de capturar los puntos de enlace asociados con los objetos referenciados, permite exponer el contexto de los puntos de enlace.

- **Basados en los argumentos del punto de enlace:** capturan los puntos de enlace cuyos argumentos son de un cierto tipo mediante el descriptor `args`. También puede ser usados para exponer el contexto.
- **Basados en condiciones:** capturan puntos de enlace basándose en alguna condición usando el descriptor `if(expresionBooleana)`.

Basados en las categorías de puntos de enlace

Este tipo de puntos de corte identifica puntos de enlace que pertenecen a cierta categoría. Existe un descriptor para cada categoría de puntos de enlace. La Tabla 6 recoge la sintaxis de cada uno de ellos:

Descriptor	Categoría punto de enlace
<code>call(methodSignature)</code>	Llamada a método
<code>execution(methodSignature)</code>	Ejecución de método
<code>call(ctorSignature)</code>	Llamada a constructor
<code>execute(ctorSignature)</code>	Ejecución de constructor
<code>get(fieldSignature)</code>	Lectura de atributo
<code>set(fieldSignature)</code>	Asignación de atributo
<code>handler(typeSignature)</code>	Ejecución de manejador
<code>staticinitialization(typeSignature)</code>	Inicialización de clase
<code>initialization(ctorSignature)</code>	Inicialización de objeto
<code>preinitialization(ctorSignature)</code>	Pre-inicialización de objeto
<code>adviceexecution()</code>	Ejecución de aviso

Tabla 6: Sintaxis de los puntos de corte según la categoría de los puntos de enlace

Las sintaxis de las firmas de método, constructor, atributo y tipo fueron vistas en páginas anteriores y se describen con más detalle en el Anexo I. La Tabla 7 muestra algunos ejemplos de uso de este tipo de puntos de corte:

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

Tabla 7: Ejemplos de puntos de corte basados en la categoría de los puntos de enlace

Puntos de corte basados en flujo de control

Este tipo de puntos de corte identifica puntos de enlace de cualquier categoría siempre y cuando ocurran en el flujo de control de los puntos de enlace capturados por otros puntos de corte. La Tabla 8 recoge la sintaxis y la semántica de los descriptores basados en flujo de control.

Sintaxis	Semántica
<code>cflow(pointcut)</code>	Captura todos los puntos de enlace en el flujo de control del punto de corte que se le pasa como parámetro, incluidos los puntos de enlace identificados por el punto de corte.
<code>cflowbelow(pointcut)</code>	Captura todos los puntos de enlace en el flujo de control del punto de corte que se le pasa como parámetro, excluidos los puntos de enlace identificados por el punto de corte.

Tabla 8: Sintaxis y semántica de los puntos de corte basados en flujo de control

Para una mejor comprensión de este tipo de puntos de corte se muestran a continuación una serie de ejemplos ilustrativos:

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

Tabla 9: Ejemplos de puntos de corte basados en flujo de control

Un uso típico de estos descriptores es, cuando trabajamos con métodos recursivos. Por ejemplo, supongamos que tenemos un método recursivo llamado `factorial(int)` dentro de la clase `Math`. El siguiente punto de corte identificará todas las llamadas al método recursivo.

```
pointcut llamadas(): call(int Math.factorial(int));
```

Mientras que éste otro punto de corte sólo capturará la primera llamada al método recursivo:

```
pointcut primera(): llamadas() && !cflowbelow(llamadas());
```


Puntos de corte basados en localización de código

Este tipo de puntos de corte identifica puntos de enlace de cualquier categoría situados dentro de cierto fragmento de código fuente. La Tabla 10 recoge la sintaxis y la semántica de los descriptores de puntos de corte basados en localización de código:

Sintaxis	Semántica
<code>within(typeSignature)</code>	Captura todos los puntos de enlace dentro del cuerpo de las clases o aspectos especificados mediante <code>typeSignature</code>
<code>withincode(methodSignature)</code> <code>withincode(constructorSignature)</code>	Captura todos los puntos de enlace dentro del cuerpo de los métodos o constructores especificados por <code>methodSignature</code> y <code>constructorSignature</code> respectivamente.

Tabla 10: Sintaxis y semántica de los puntos de corte basados en localización

En la siguiente tabla se muestran ejemplos de este tipo de puntos de corte:

Punto de corte	Puntos de enlace identificados
<code>within(Elemento)</code>	Todos los puntos de enlace dentro de la clase <code>Elemento</code> .
<code>within(Elemento+)</code>	Todos los puntos de enlace dentro de la clase <code>Elemento</code> y sus subclases.
<code>withincode(* Punto.set*(..))</code>	Todos los puntos de enlace dentro del cuerpo de los métodos de la clase <code>Punto</code> cuyo nombre empieza por <code>set</code> .

Tabla 11: Ejemplos de puntos de corte basados en localización de código

Un uso común del descriptor `within` es excluir los puntos de enlace generados en un aspecto. Por ejemplo, el siguiente aspecto implementa un mecanismo de *tracing*², y para ello creamos un punto de corte que captura todos los puntos de enlace excepto los generados por código perteneciente al aspecto `TracingAspect`.

```
public aspect TracingAspect {

    pointcut pointsToBeTraced(): !within(TracingAspect);

    before():pointsToBeTraced()
    {
        System.out.println("Tracing: Enter " +
                           thisJoinPoint.getSignature());
    }

    after():pointsToBeTraced()
    {
        System.out.println("Tracing: Exit " +
                           thisJoinPoint.getSignature());
    }
}
```

² Seguimiento del flujo de ejecución de un programa

Puntos de corte basados en objetos

Este tipo de puntos de corte identifican puntos de enlace atendiendo al tipo del objeto actual (`this`) y al tipo del objeto destino (`target`) en tiempo de ejecución. Entendemos por objeto actual el objeto que se está ejecutando actualmente y por objeto destino, el objeto sobre el que se invoca un método. Además de capturar los puntos de enlace asociados con los objetos referenciados, permite exponer el contexto de los puntos de enlace. Los descriptores de puntos de corte basados en objetos son:

Sintaxis	Semántica
<code>this(typeSignature)</code>	Captura todos los puntos de enlace cuyo objeto actual es una instancia del tipo especificado por <code>typeSignature</code> o cualquiera de sus subclases.
<code>this(<identifier>)</code>	Expone el objeto actual del punto de enlace y lo guarda en la variable de nombre <code><identifier></code>
<code>target(typeSignature)</code>	Captura todos los puntos de enlace cuyo objeto destino es una instancia del tipo especificado por <code>typeSignature</code> o cualquiera de sus subclases.
<code>target(<identifier>)</code>	Expone el objeto destino del punto de enlace y lo guarda en la variable de nombre <code><identifier></code>

Tabla 12: Sintaxis y semántica de los puntos de corte basados en objetos

Los descriptores (`this`, `target`) no capturarán llamadas a métodos estáticos, ya que estos métodos carecen de un objeto actual y destino asociados. Si queremos capturar también los puntos de enlace asociados a elementos estáticos podríamos usar el descriptor `within`, con el operador `+` para capturar los métodos tanto estáticos como de instancia de la clase en cuestión o cualquiera de sus subclases. La expresión `typeSignature` en los descriptores `this` y `target` no puede contener los comodines asterisco y dos puntos. El comodín `+`, no se necesita porque las subclases ya son tenidas en cuenta por las reglas de herencia de Java. La Tabla 13 muestra algunos ejemplos de uso de este tipo de puntos de corte.

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

Tabla 13: Ejemplos de puntos de corte basados en objetos

Para ciertos puntos de enlace no existen los objetos `this` o `target` o bien representan al mismo objeto. La Tabla 14 define los objetos `this` y `target` para cada categoría de puntos de enlace, si es que existen.

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*	
Llamada a constructor	objeto que realiza la llamada*	No existe
Ejecución de constructor	objeto que ejecuta el constructor	
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*	
Inicialización de clase	No existe	No existe
Inicialización de objeto	objeto que realiza la inicialización	
Pre-inicialización de objeto	No existe	No existe
Ejecución de aviso	aspecto que ejecuta el aviso	

Tabla 14: Objetos `this` y `target` según la categoría del punto de enlace

Puntos de corte basados en argumentos

Estos puntos de corte identifican los puntos de enlace cuyos argumentos son de un cierto tipo. También son usados para exponer el contexto de los puntos de enlace, en concreto, para exponer los argumentos de un punto de enlace. Existe un único descriptor de puntos de corte basados en argumentos cuya sintaxis es la siguiente:

args(typeSignature o <identifier>, typeSignature o <identifier>)

La Tabla 15 define los argumentos para cada categoría de puntos de enlace.

Punto de enlace	Argumentos
Llamada a método	argumentos del método
Ejecución de método	argumentos del método
Llamada a constructor	argumentos del constructor
Ejecución de constructor	argumentos del constructor
Lectura de atributo	no existen
Asignación de atributo	nuevo valor del atributo
Ejecución de manejador	excepción manejada
Inicialización de clase	no existen
Inicialización de objeto	argumentos del constructor
Pre-inicialización de objeto	argumentos del constructor
Ejecución de aviso	argumentos del aviso

Tabla 15: Argumentos según la categoría del punto de enlace

La siguiente tabla muestra algunos ejemplos de uso del descriptor de punto de corte `args`:

Punto de corte	Puntos de enlace identificados
<code>args(int,...,String)</code>	Todos los puntos de enlace cuyo primer argumento sea de tipo <code>int</code> y su último argumento sea de tipo <code>String</code> .

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

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

<code>call(* *.set*(..)) && args(String)</code>	Todos las llamadas a métodos, cuyo nombre empieza por <code>set</code> y tienen un único parámetro de tipo <code>String</code> . El siguiente punto de corte es equivalente: <code>call(* *.set*(String))</code>
<code>call(* *.set*(int)) && args(argInt)</code>	Todos las llamadas a los métodos cuyo nombre empieza por <code>set</code> y tienen un único parámetro de tipo <code>int</code> . El valor del parámetro se transfiere a la variable <code>argInt</code>
<code>call(* Punto.*(int,String)) && args(argInt, ..)</code> <code>call(* Punto.*(int,String)) && args(argInt,String)</code>	Todas las llamadas a métodos de la clase <code>Punto</code> con dos argumentos, el primero de tipo <code>int</code> y el segundo de tipo <code>String</code> . El primero de los argumentos se transfiere a la variable <code>argInt</code> .
<code>call(Punto.new(int,int)) && args(x,y)</code>	Todas las llamadas al constructor de la clase <code>Punto</code> con dos argumentos de tipo <code>int</code> . Los argumentos se transfieren a las variables <code>x</code> e <code>y</code> respectivamente.

Tabla 16: Ejemplos de puntos de corte basados en argumentos

Puntos de corte basados en condiciones

Este tipo de puntos de corte identifica puntos de enlace cuando se cumplen ciertas expresiones condicionales. Existe un único descriptor de puntos de corte condicionales cuya sintaxis es la siguiente:

`if (expresionBooleana)`

La Tabla 17 muestra algunos ejemplos de puntos de corte condicionales:

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

Tabla 17: Ejemplos de puntos de corte basados en condiciones

Una vez vistos los puntos de corte soportados por AspectJ, en el siguiente apartado profundizaremos en el último mecanismo de *crosscutting dinámico* que nos falta por ver: los **avisos** (*advice*).

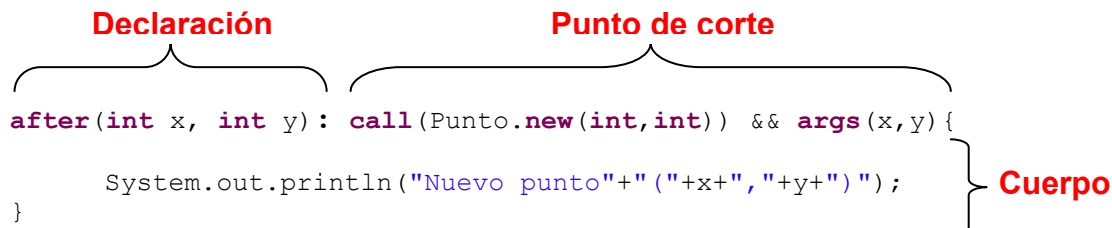
3.2.3. Avisos

Los avisos son construcciones, similares a los métodos de una clase, que especifican las acciones a realizar en los puntos de enlace que satisfacen cierto punto de corte. Mientras los puntos de corte indican *donde*, los avisos especifican *que hacer*. Los avisos aparecen en el cuerpo de un aspecto y no tienen un nombre o identificador como es el caso de los puntos de corte o los métodos y atributos de una clase, ya que no será necesario referirnos a ellos en el código. Esto tiene algunos inconvenientes como por ejemplo que no se podrá capturar la ejecución de un determinado aviso ya que no

podemos identificarlo, tan sólo podremos capturar la ejecución de los avisos definidos en el cuerpo de cierto aspecto mediante el siguiente punto de corte:

```
adviceexecution() && within(MiAspecto)
```

Un aviso lo podemos dividir en tres partes: declaración, especificación del punto de corte y cuerpo del aviso.



```
after(int x, int y) : call(Punto.new(int,int)) && args(x,y) {
    System.out.println("Nuevo punto"+"("+x+", "+y+")");
}
```

La declaración de un aviso está delimitada por el carácter ":" y consta del tipo de retorno (sólo para avisos de tipo `around`), el tipo de aviso (`before`, `after` o `around`) y su lista de parámetros, que expone información de contexto para que pueda ser usada en el cuerpo del aviso. Después de la lista de parámetros se especifican las excepciones que puede lanzar el cuerpo del aviso mediante la cláusula `throws`. La especificación del punto de corte puede ser o un punto de corte anónimo o el identificador de un punto de corte predefinido con sus correspondientes parámetros si los tiene. Por último, el cuerpo de un aviso encierra las acciones a realizar cuando se alcanza algún punto de enlace que satisface la especificación del punto de corte.

Acceso a información de contexto: parámetros & reflexión

Mediante los **parámetros** de un aviso se puede exponer información de contexto para que sea accesible desde el cuerpo de un aviso. Para ligar estos parámetros con la información de contexto se utiliza:

- el descriptor de punto de corte `args()` para exponer los argumentos de un punto de enlace.
- el descriptor de punto de corte `this()` para obtener el objeto actual asociado a la ejecución de un punto de enlace.
- el descriptor de punto de corte `target()` para exponer el objeto destino asociado a la ejecución de un punto de corte.
- las cláusulas `returning` y `throwing` para recuperar el valor retornado o la excepción lanzada por un punto de enlace. Sólo aplicables a los avisos de tipo `around`.

Los parámetros de un aviso se comportan como variables locales en el cuerpo del aviso, de forma que cualquier modificación de los parámetros en el cuerpo del aviso, no tendrá efecto fuera de él. Esta situación varía si utilizamos avisos de tipo `around`, en cuyo caso los cambios si que tendrán efecto fuera del aviso. Un ejemplo de acceso a información de contexto mediante parámetros sería:

```
pointcut metodosSet(Punto p, int i) : call( * *.set*(*) ) &&
                                     target(p) && args(i);

before(Punto p, int i) : metodosSet(p,i) {
```

```

        System.out.println(p);
    }

    after(Punto p, int i): metodosSet(p,i) {
        System.out.println(p);
    }

```

En el punto de corte `metodosSet` capturamos todas las llamadas a los métodos `set` de la clase `Punto` que tienen un único parámetro de tipo entero. Además, exponemos el objeto destino de la llamada (`target(p)`) y el parámetro (`args(i)`) para que puedan ser usados en el cuerpo del aviso.

A la hora de utilizar parámetros tanto en los puntos de corte como en los avisos se tiene que cumplir la siguiente regla: *todos los parámetros que aparecen a la izquierda del caracter delimitador dos puntos “:” deben estar ligados a alguna información de contexto a la derecha de los dos puntos*. En el ejemplo anterior podemos ver como se cumplía dicha regla:

```

pointcut metodosSet(Punto p, int i): target(p) && args(i) &&
    call( * *.set*(*) );

before(Punto p, int i): metodosSet(p,i) {
    System.out.println(p);
}

after(Punto p, int i): metodosSet(p, i) {
    System.out.println(p);
}

```

Los siguientes ejemplos darán lugar a un error por parte del compilador indicando que el parámetro “i” no esta ligado a ninguna información de contexto.

```

pointcut metodosSet(Punto p,int i): call( * *.set*(*) )
    && target(p) ;

before(Punto p,int i): metodosSet(p) {
    System.out.println(p);
}

```

La regla anterior no tiene porque cumplirse en sentido contrario, es decir, de derecha a izquierda. Por ejemplo, el siguiente aviso sólo usa un parámetro, mientras que el punto de corte tiene dos. El segundo parámetro del punto de corte contiene el tipo del parámetro en lugar de un identificador.

```

before(Punto p): metodosSet(p,int) {
    System.out.println(p);
}

```

También podemos acceder al contexto de un punto de enlace de **forma reflexiva**, mediante una serie de variables especiales proporcionadas por AspectJ:

- **thisJoinPoint**: variable de tipo `org.aspectj.lang.JoinPoint` que nos permite acceder a información, tanto estática como dinámica, sobre el contexto del punto de enlace actual mediante una serie de métodos:
 - `Object getTarget()`: retorna el objeto destino.
 - `Object getThis()`: retorna el objeto actual.
 - `Object[] getArgs()`: devuelve los argumentos del punto de enlace.
 - `String getKind()`: devuelve una cadena que indica la categoría del punto de enlace.
 - `Signature getSignature()`: devuelve un objeto de la clase `Signature` que representa la signatura del punto de enlace. A través de este objeto podemos acceder al identificador del punto de enlace (`getName()`), a sus modificadores de acceso (`getModifiers()`), etc.
 - `SourceLocation getSourceLocation()`: devuelve un objeto de la clase `SourceLocation` que representa al objeto invocador del punto de enlace. A través de este objeto podemos por ejemplo obtener la clase del objeto invocador con el método `getWithinType()`.
 - `String toString()`: retorna una cadena que representa el punto de enlace.
 - `String toShortString()`: retorna una cadena corta que representa el punto de enlace.
 - `String toLongString()`: retorna una cadena larga que representa el punto de enlace.
 - `JoinPoint.StaticPart getStaticPart()`: devuelve el objeto que encapsula la parte estática del punto de enlace.
- **thisJoinPointStaticPart**: variable perteneciente a la clase `org.aspectj.lang.JoinPoint.StaticPart` con la que sólo podemos acceder a las partes estáticas del contexto de un punto de enlace mediante los métodos `getKind()`, `getSignature()`, `getSourceLocation()`, `toString()`, `toShortString()` y `toLongString()`. La funcionalidad de estos métodos es idéntica a la de sus homólogos de la clase `org.aspectj.lang.JoinPoint`. Si sólo estamos interesados en acceder a información estática de un punto de enlace es mejor usar `thisJoinPointStaticPart` que `thisJoinPoint` por cuestiones de rendimiento

El siguiente fragmento de código muestra un ejemplo de acceso reflexivo al contexto de un punto de enlace:

```
before(): call( * Punto.set*(int) ) {
    System.out.println(thisJoinPoint.getTarget());
}

before(): call( * Punto.set*(int) ) {
    System.out.println(
        thisJoinPointStaticPart.getSourceLocation().getFileName()
        + "\t" +
        thisJoinPointStaticPart.getSourceLocation().getLine()
    );
}
```

Vistas las dos opciones que tenemos para acceder al contexto de un punto de enlace se recomienda usar un acceso reflexivo sólo en el caso de que la información que buscamos no pueda ser accedida a través de parámetros, como por ejemplo para obtener el número de línea donde ocurrió un cierto punto de corte (`thisJoinPoint.getSourceLocation().getLine()`). Esto es debido a que el acceso reflexivo es más lento y consume más recursos que su equivalente mediante parámetros.

Otra ventaja del acceso mediante parámetros, es que además de exponer el contexto del punto de enlace realizan comprobación de tipos. Por ejemplo, si tengo la siguiente declaración de aviso `after(Punto p)` y en la especificación del punto de corte asociado aparece `target(p)`, estoy forzando a que el objeto destino del punto de enlace sea una instancia de la clase `Punto`.

Tipos de avisos: *before*, *after* y *around*

El tipo de un aviso indica cuando se ejecutará el aviso en relación con el punto de enlace capturado. AspectJ soporta tres tipos de avisos:

Avisos *before*

Son el tipo de aviso más simple. El cuerpo del aviso se ejecuta antes del punto de enlace capturado. Si se produjera una excepción durante la ejecución del aviso, el punto de enlace capturado no se ejecutaría. Este tipo de avisos suele ser usado para realizar comprobación de parámetros, logging, autenticación etc. El siguiente código muestra un ejemplo de comprobación de parámetros mediante un aviso *before*:

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

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

Avisos *after*

El cuerpo del aviso se ejecuta después del punto de enlace capturado. Se distinguen tres tipos de avisos *after*:

- Aviso *after* (no calificado): el aviso se ejecutará siempre, sin importar si el punto de enlace finalizó normalmente o con el lanzamiento de alguna excepción. Se comporta como un bloque `finally` en Java. En el siguiente ejemplo el método `Logger.writeLog` se ejecutará tras la ejecución de todo punto de enlace capturado por el punto de corte `cambioPosicionPunto`.

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



```
}
```

Al igual que ocurría con los avisos *after returning*, un aviso *after throwing* sólo se ejecutará si el tipo de la excepción lanzada por el punto de enlace capturado coincide con el indicado en la cláusula *throwing*. Si el tipo indicado en la cláusula *throwing* es *Exception*, el aviso se ejecutará para cualquier excepción.

Avisos *around*

Los avisos *around* son el tipo de aviso más complejo pero a la vez más potente. Se ejecutan en lugar del punto de enlace capturado, ni antes ni después. Mediante este tipo de avisos podemos:

- reemplazar la ejecución original del punto de enlace por otra alternativa
- ignorar la ejecución del punto de enlace, por ejemplo cuando los parámetros de un método son ilegales.
- 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.

Para ejecutar el punto de enlace original dentro del cuerpo del aviso usamos un método especial: **proceed()**. Este método toma como parámetros 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, deberán devolver un valor de un tipo compatible al tipo de retorno del punto de enlace reemplazado. Dependiendo de si el punto de corte abarca uno o muchos puntos de enlace con diferentes tipos de retorno, tendremos que ser más o menos específicos a la hora de indicar el tipo de retorno del aviso, para que sea compatible con todos los tipos de retorno de los puntos de enlace capturados. Si optamos por un tipo de retorno amplio (como *Object*), AspectJ se encarga automáticamente de realizar el *casting* al tipo de retorno del punto de enlace después de la ejecución del cuerpo del aviso. Con el método `proceed()` ocurre lo mismo, AspectJ se encarga de realizar el casting necesario para que el tipo devuelto por `proceed()` coincida con el tipo de retorno del aviso, incluso para el caso de los tipos primitivos que los envuelve en su correspondiente tipo envolvente. Veámoslo con un ejemplo:

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

El punto de corte asociado al aviso anterior captura todas las llamadas a métodos `get` de la clase *Punto* que devuelven un entero. Si nosotros ejecutamos dentro del cuerpo del aviso el método original mediante `proceed()`, lo normal sería que nos devolviera un entero, pero no es el caso puesto que `proceed()` tiene el mismo tipo de retorno que el aviso, por lo que retornará un objeto de tipo *Object*. AspectJ realiza esta conversión de tipos de forma transparente al usuario, envolviendo el entero en un objeto de la clase *Integer*. Puesto que el cuerpo del aviso sustituye a la ejecución del método `get` original, debería retornar un entero para cumplir con el contrato de los métodos capturados por el punto de corte. Aunque el aviso devuelva un objeto de la clase

Object, AspectJ lo convierte en un entero una vez finalizada la ejecución del cuerpo del aviso de forma totalmente transparente al usuario.

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

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

En el cuerpo del aviso anterior se comprueba que el parámetro que se le pasa al método `reintegro` de la clase `Cuenta`. Si es mayor que el reintegro mínimo se ejecuta el método, en caso contrario se ignora la ejecución del método. Veamos con un ejemplo como es posible **cambiar el contexto de un punto de enlace**, en concreto los parámetros que se le pasan a un método:

```
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;
}
```

En este aviso se añade la fecha y hora de impresión al principio de todos los mensajes que se imprimen por pantalla. Para ello, se identifica mediante un punto de corte anónimo todas las llamadas a los métodos de impresión (`PrintStream.println`) y se expone el mensaje a imprimir para que pueda ser modificado en el cuerpo del aviso, donde se concatena con la fecha y hora actuales y se pasa como parámetro al método `proceed` que se encarga de ejecutar el método original. Otro ejemplo de aplicación de avisos *around* es el **manejo de excepciones**. Imaginar que tenemos una serie de métodos que pueden originar excepciones que deseamos manejar de la misma forma. Con el siguiente aviso nos ahorraríamos escribir para cada operación su correspondiente método `try/catch`:

```
Object around(): operaciones(){
    try{
        return proceed();
    }catch(Exception e){
        //Manejo de la excepcion
    }
}
```

El punto de corte `operaciones()` identifica los métodos que pueden lanzar las excepciones a tratar y dentro del cuerpo del aviso encapsulamos la llamada al método dentro de un bloque `try/catch`.

3.2.4. Declaraciones inter-tipo

Las declaraciones inter-tipo (*inter-type declarations*), también llamadas introducciones, nos permiten añadir nuevos miembros a un tipo (clase, interfaz o aspecto) sin modificar su código. Reciben este nombre porque son declaraciones sobre la estructura de un programa que ocurren entre tipos, por ejemplo desde un aspecto se puede añadir un nuevo campo a una clase sin modificar el código de la clase afectada. Podemos declarar los siguientes miembros usando declaraciones inter-tipo:

Campos

Se puede añadir un campo a cualquier tipo: clases, interfaces o aspectos. Por ejemplo, la declaración para añadir un campo `color` a la clase `Punto` será la siguiente:

```
public java.awt.Color Punto.color = java.awt.Color.black;
```

Sólo puede haber una declaración de campo inter-tipo por línea. La siguiente declaración provocará un error en tiempo de compilación:

```
public java.awt.Color Punto.color, Linea.color;
```

Métodos concretos

Se puede añadir un método concreto a cualquier tipo, incluido interfaces. Por ejemplo, para añadir los métodos `setColor` y `getColor` a la clase `Punto` utilizaremos las siguientes declaraciones:

```
public void Punto.setColor(java.awt.Color c){
    color = c;
}

public java.awt.Color Punto.getColor(){
    return color;
}
```

Métodos abstractos

Podemos añadir un método abstracto a tipos abstractos (clases o aspectos) y a interfaces. La palabra clave `abstract` puede ir antes o después de los modificadores de acceso. Por ejemplo, la declaración para añadir el método abstracto `toXML` a la clase abstracta `XMLSupport` será la siguiente.

```
public abstract void XMLSupport.toXML(PrintStream out);
```

Constructores

Se pueden añadir tanto a clases abstractas como concretas, pero no a interfaces o aspectos. Con la siguiente declaración se añade un constructor a la clase `Punto`:

```
public Punto.new(int x, int y, java.awt.Color color){
    this(x,y);
}
```

```
        this.color = color;  
    }
```

Cualquier miembro inter-tipo (campo, método o constructor) añadido a cierta clase, aspecto o interfaz será heredado por las subclases de la clase o aspecto destino o en el caso de que el tipo destino sea una interfaz, por las clases que la implementen. Los modificadores de acceso usados en las declaraciones inter-tipo definen el ámbito respecto al aspecto que las declara y no respecto al tipo de destino:

- **private**: el miembro inter-tipo sólo podrá ser visto por el aspecto que lo define y en ningún caso podrá ser accesible desde el tipo destino o cualquier otra clase.
- **package**: el miembro inter-tipo sólo podrá ser visto por el aspecto que lo define y por las clases de su mismo paquete
- **protected**: este modificador no está soportado por las declaraciones inter-tipo.
- **public**: el miembro inter-tipo será accesible desde cualquier clase.

En AspectJ 1.0 una declaración inter-tipo podía afectar a más de un tipo destino mediante el uso de comodines u operadores lógicos a la hora de especificar el tipo destino, por ejemplo con la siguiente declaración se conseguía añadir el atributo `color` a las clases `Punto` y `Línea`:

```
public java.awt.Color (Punto || Línea).color;
```

Esta posibilidad se elimina a partir de la versión 1.1. porque originaba problemas. Aún existe una posibilidad para conseguir que una declaración inter-tipo afecte a varios tipos destino: declarar los miembros en una interfaz y que las clases destino implementen dicha interfaz. Para más información sobre la sintaxis de las declaraciones inter-tipo consultar el Anexo I.

Herencia Múltiple

Hemos visto que AspectJ nos permite definir métodos concretos y campos no públicos en interfaces mediante declaraciones inter-tipo. Esto supone una gran diferencia respecto a Java, donde no es posible que se den tales situaciones. Aún así, las interfaces en AspectJ siguen manteniendo la naturaleza de las interfaces Java, ya que estas no pueden ser instanciadas directamente y por consiguiente no poseen constructores. Por lo tanto, AspectJ permite que las interfaces soporten tanto comportamiento abstracto como concreto posibilitando un tipo especial de herencia múltiple, lo que puede llevar a la aparición de conflictos como resultado de heredar el mismo comportamiento por más de una vía.

3.2.5. Declaraciones de parentesco

Este mecanismo de *crosscutting estático* nos permite declarar que ciertas clases implementan nuevas interfaces o extienden nuevas clases.

Implementar nuevas interfaces

Puesto que mediante este tipo de declaraciones indicamos que una clase implementa una nueva interfaz, tendremos que implementar los métodos definidos en el

cuerpo de la interfaz. Tenemos dos opciones: que sea la propia clase destino la que implemente los métodos o bien usar declaraciones inter-tipo para añadir los métodos concretos a la clase destino. En el siguiente ejemplo se usa una declaración de parentesco (`declare parents`) para indicar que la clase `Punto` implementa la interfaz `Cloneable` y mediante una declaración inter-tipo se añade a la clase `Punto` la implementación del método `clone()`.

```
declare parents: Punto implements Cloneable;

public Object Punto.clone() throws CloneNotSupportedException {
    Punto p = (Punto) super.clone();
    return p;
}
```

Extender nuevas clases

Con la construcción `declare parents` también podemos especificar que ciertas clases extienden nuevas clases. Por ejemplo, mediante la siguiente declaración indicamos que la clase `Poligono` hereda de la clase `Elemento`.

```
declare parents: Poligono extends Elemento.
```

En el caso de que especifiquemos más de una clase en la lista de clases a extender, estas clases deben pertenecer a la misma jerarquía de herencia y deben ser compatibles con el supertipo original de la clase destino de la declaración. Además se deben seguir las reglas jerárquicas de Java como que una clase no puede ser padre de una interfaz y no se pueden hacer declaraciones que den lugar a herencia múltiple. A diferencia de las declaraciones inter-tipo, en las declaraciones de parentesco si que podemos usar patrones de tipo para que estas afecten a más de una clase destino. Por ejemplo, la siguiente declaración indica que todas las clases del paquete `com.company` implementan la interfaz `Serializable`.

```
declare parents: com.company.* implements Serializable
```

Para profundizar en la sintaxis de las declaraciones de parentesco consultar el Anexo I.

3.2.6. Declaraciones en tiempo de compilación

AspectJ ofrece un mecanismo para añadir advertencias o errores en tiempo de compilación y así notificar ciertas situaciones (puntos de enlace) que deseamos advertir o evitar, por ejemplo la utilización de cierto método en desuso. Cuando el compilador detecta algún error o advertencia imprime el mensaje especificado en la declaración del error o la advertencia. La única diferencia entre las advertencias y los errores, es que estos últimos abortan el proceso de compilación.

Estas declaraciones se basan en el uso de un punto de corte que identifica los puntos de enlace para los que deseamos que el compilador genere una advertencia o un error. Puesto que son declaraciones en tiempo de compilación sólo podremos usar aquellos descriptores de puntos de corte que operan en tiempo de compilación (no manejan información de contexto, en tiempo de ejecución) que son: `call()`, `execution()`, `get()`, `set()`, `adviceexecution()`, `intialization()`,

`staticInitialization()`, `handler()`, `within()`, and `withinCode()`. Los puntos de corte que sólo contienen descriptores de la lista anterior, se denominan puntos de corte determinables estáticamente (*statically determinable*). La sintaxis detallada de este tipo de declaraciones se recoge en el Anexo I. Veamos a continuación un ejemplo de declaración de una advertencia en tiempo de compilación:

```
declare warning: llamadasIlegales() && !fachada():
    "Uso ilegal del sistema Motor Reglas - usar el objeto
    FachadaMotorReglas";
```

Recientemente, ha arrancado el proyecto *PatternTesting* [27] que estudia las posibilidades de este potente mecanismo ofrecido por AspectJ para la creación de patrones de prueba reutilizables.

3.2.6. Suavizado de excepciones

El lenguaje Java obliga a que todas excepciones que son subclases de `Exception`, excepto las subclases de `RuntimeException`, tengan que ser declaradas en la signatura de un método mediante la cláusula `throws` o bien deban ser capturadas y manejadas mediante un bloque `try/catch`. AspectJ ofrece un mecanismo que nos permite ignorar el sistema de comprobación de excepciones de Java, silenciando las excepciones que tienen lugar en determinados puntos de enlace, encapsulándolas en excepciones no comprobadas del tipo `org.aspectj.lang.SoftException` (subclase de la clase `RuntimeException`) y relanzándolas. Se dice entonces que la excepción ha sido suavizada.

En este tipo de declaraciones (`declare soft`), mediante un punto de corte indicamos los puntos de enlace donde se lanzan excepciones del tipo que deseamos suavizar. Sólo podemos usar puntos de corte determinables estáticamente (ver apartado anterior). Una situación en la que puede ser útil el suavizado de excepciones es cuando debemos manejar determinadas excepciones que tenemos certeza que no ocurrirán, por lo que podríamos suavizarlas y así ahorrarnos escribir los bloques `try/catch` que enmarañan el código de la aplicación. Además, muchos manejadores de excepciones se limitan a imprimir la pila de llamadas por lo que una opción viable y que evitaría la captura y manejo de esas excepciones sería suavizarlas. Las siguientes dos construcciones suavizan todas las excepciones que ocurren en una aplicación y l. En la primera construcción utilizamos una declaración de excepciones suavizadas mientras que en la segunda utilizamos un aviso *around*.

```
declare soft: Exception : execution(* *.*(..));

Object around() : execution(* *.*(..))
{
    try{
        return proceed();
    } catch (Exception e){
        e.printStackTrace();
    }
}
```

Para más información sobre la sintaxis de este tipo de declaraciones consultar el Anexo I.

3.2.7. Declaraciones de precedencia

En ocasiones, dos avisos definidos en aspectos diferentes afectan a los mismos puntos de enlace y nos interesa establecer una relación de precedencia entre los avisos, para que uno se ejecute antes que otro. Mediante las declaraciones de precedencia (`declare precedence`) podemos especificar que todos los avisos de un aspecto tengan preferencia a la hora de ejecutarse con respecto a los de otros aspectos. Es posible el uso de comodines a la hora de especificar la lista de precedencia, teniendo en cuenta que un mismo tipo no puede aparecer más de una vez en la lista de precedencia. Por ejemplo, la siguiente declaración provocaría un error de compilación indicando que el aspecto `LogAspect` aparece dos veces en la lista de precedencia:

```
declare precedence: Aspecto, LogAspect, L*;
```

Si el carácter “*” aparece en solitario en la lista de tipos, representa a cualquier aspecto no listado y se puede usar para situar cualquier aspecto no listado antes, entre o después de los aspectos que aparecen en la lista, en términos de precedencia. El comodín “*” no podrá aparecer más de una vez en solitario, en la lista de precedencia. Por ejemplo, en la siguiente declaración el aspecto `Aspecto` es el de mayor precedencia, `LogAspect` el de menor y con una precedencia intermedia se encuentran el resto de aspectos no listados.

```
declare precedence: Aspecto, *, LogAspect;
```

Si queremos dar mayor precedencia a los aspectos que extienden cierto aspecto. Podemos usar el comodín “+”, por ejemplo:

```
declare precedence: TraceAspect+, *;
```

A la hora de crear el orden de precedencia, el compilador ignora los aspectos abstractos que aparecen en la lista de precedencia así como aquellos tipos que aparecen en la lista y no corresponden con ningún aspecto. Además, el compilador de `AspectJ` permite determinadas relaciones de precedencia circulares, como por ejemplo:

```
declare precedence: Aspecto, LogAspect;  
declare precedence: LogAspect, Aspecto;
```

siempre y cuando los aspectos implicados (`Aspecto` y `LogAspect`) no declaren avisos que afecten a los mismos puntos de enlace. Para más información sobre la sintaxis de las declaraciones de precedencia consultar el Anexo I.

3.2.5. Aspectos

Un aspecto es una entidad que encapsula puntos de enlace, puntos de corte, avisos y declaraciones y además puede contener sus propios atributos y métodos como una clase Java normal (consultar sintaxis en Anexo I). El código de un aspecto puede situarse en distintas partes. Si el aspecto afecta a distintas partes de la aplicación, lo normal es escribir el aspecto en un fichero independiente con extensión `.aj`, como se muestra en el siguiente fragmento de código:


```

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

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

```

Cuando el aspecto afecta a una única clase de la aplicación, optamos por escribir el aspecto como un miembro más de la clase. Para ello definimos el aspecto como estático (`static`) y si además es privado (`private`) la encapsulación será total y sólo podrá ser accedido por la clase que lo contiene.

```

public class Clase {

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

    private static aspect Aspecto {

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

```

Extensión de Aspectos

Un aspecto se puede extender de diferentes formas:

- construyendo un aspecto abstracto (modificador `abstract`) y usándolo como base para otros aspectos. Un aspecto no puede heredar aspectos concretos, provocará un error de compilación, sólo puede heredar aspectos abstractos. Cuando un aspecto hereda un aspecto abstracto, no sólo hereda los atributos y métodos del aspecto sino también los puntos de corte y los avisos.
- heredando de clases o implementando una serie de interfaces. Sin embargo, AspectJ no permite a una clase heredar un aspecto, provocará un error de compilación.

Instanciación de aspectos

Un aspecto no posee constructores por lo que no se podrá instanciar como una clase Java normal. Las instancias de un aspecto se crean automáticamente según al tipo de instanciación del aspecto que puede ser: *singleton*, *per-object* y *per-control-flow*. El tipo de instanciación de un aspecto se puede declarar explícitamente o bien se hereda de un aspecto padre. Si un aspecto no hereda de ningún aspecto y no se especifica su tipo de instanciación, se aplicará la instanciación por defecto, esto es *singleton*. Las características de cada tipo de instanciación son:

- **Singleton:** una única instancia del aspecto será creada para toda la aplicación principal. Dicha instancia se crea cuando se carga la clase que define el tipo del aspecto y está disponible durante toda la ejecución de la aplicación principal a través del método estático del aspecto `aspectOf()`. La sintaxis para especificar que un aspecto es *singleton* es:

```
aspect nombreAspecto issingleton() {
    .....
}
```

- **Per-object:** dentro de este tipo de instanciación existen dos variantes:
 - **per-this:** una instancia del aspecto es creada por cada objeto actual (*this*) diferente asociado a los puntos de enlace identificados por un determinado punto de corte. Para obtener la instancia del aspecto, y puesto que ahora existe una instancia distinta por cada objeto *this* distinto, le pasamos al método `aspectOf()` el objeto actual: `aspectOf(thisJointPoint.getThis())`. La sintaxis para usar una instanciación *per-this* es:

```
aspect nombreAspecto perthis(pointcut) {
    .....
}
```

- **per-target:** una instancia del aspecto es creada por cada objeto destino (*target*) diferente asociado a los puntos de enlace identificados por un determinado punto de corte. Para obtener la instancia del aspecto, y puesto que ahora existe una instancia distinta por cada objeto *target* distinto, le pasamos al método estático `aspectOf()` el objeto destino: `aspectOf(thisJointPoint.getTarget())`. La sintaxis para usar una instanciación *per-target* es:

```
aspect nombreAspecto pertarget(pointcut) {
    .....
}
```

- **Per-control-flow:** dentro de este tipo de instanciación existen dos variantes:
 - **per-cflow:** una instancia del aspecto es creada cada vez que se ejecuta un punto de enlace bajo el flujo de control de los puntos de enlace identificados por un punto de corte, incluidos los propios puntos de

enlace identificados por el punto de corte. La sintaxis para usar una instancia *per-cflow* es:

```
aspect nombreAspecto percflow(pointcut) {
    .....
    .....
}
```

- **per-cflowbelow**: una instancia del aspecto es creada cada vez que se ejecuta un punto de enlace bajo el flujo de control de los puntos de enlace identificados por el punto de corte, excluidos los propios puntos de enlace identificados por el punto de corte. La sintaxis para usar una instancia *per-cflowbelow* es:

```
aspect nombreAspecto percflowbelow(pointcut) {
    .....
    .....
}
```

En ambos casos (*per-cflow* y *per-cflowbelow*), para obtener la instancia del aspecto en tiempo de ejecución usaremos el método estático del aspecto `aspectOf()`, sin pasarle ningún parámetro. Esto se debe a que cada vez que se crea una nueva instancia del aspecto al llegar a cierto punto de enlace, esta se mete en una pila y se destruye al terminar la ejecución del punto de enlace. Cuando llamamos al método `aspectOf()` este devolverá la instancia del aspecto que hay en el tope de la pila.

Precedencia de Aspectos

Cuando los puntos de corte asociados a dos avisos tienen puntos de enlace en común, el tejedor de aspectos debe establecer una precedencia entre los avisos para determinar que aviso debe ejecutarse primero. Esta precedencia dependerá de si los avisos residen en el mismo aspecto o en aspectos diferentes.

Si residen en diferentes aspectos:

- Usando declaraciones de precedencia podemos especificar que unos aspectos tienen precedencia sobre otros. Ejemplo: A tiene precedencia sobre B

```
declare precedence: A, B;
```

- Si un aspecto extiende a otro, todos los avisos del subaspecto tienen mayor prioridad que los avisos del superaspecto.
- Si no se dan ninguna de las situaciones anteriores, la relación de precedencia queda indefinida.

Si residen en el mismo aspecto:

- Si uno de los avisos es de tipo `after`, el aviso definido más tarde en el código del aspecto, tendrá mayor prioridad.
- Si ninguno de los avisos es de tipo `after`, el aviso definido primero en el código del aspecto, tendrá mayor prioridad.

Aspectos privilegiados

En AspectJ 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 se podrá acceder a miembros privados 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. Los intentos de acceder a miembros privados o protegidos darán lugar a errores de compilación. Puede que en algunas situaciones sea imprescindible acceder a algún miembro privado o protegido en el cuerpo de un aviso o en una declaración de métodos inter-tipo. Si declaramos el aspecto como `privileged` tendremos acceso a cualquier miembro de cualquier clase o aspecto.

```
privileged aspect nombreAspecto {  
    .....  
    .....  
}
```

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

3.3. Implementación de AspectJ

Los creadores de AspectJ no pensaron en este lenguaje únicamente como una extensión de Java con un compilador para convertir ese conjunto de palabras clave en código ejecutable (byte-code) por la máquina virtual de Java, sino que veían AspectJ como una nueva plataforma de programación, que estaba integrada por otras herramientas además del compilador:

- un depurador para las clases generadas por el compilador (**ajdb**).
- un generador de documentación javadoc (**ajdoc**).
- un navegador de aspectos (**ajbrowser**).

Estas herramientas que forman parte de la plataforma y que se encuentran ubicadas en el directorio `/bin` de la distribución de AspectJ se describen con detalle en el Anexo III.

4. Aplicación práctica: Terminal punto de venta

4.1. Introducción

Para mostrar de una forma práctica los beneficios de la programación orientada aspectos y en concreto del lenguaje AspectJ, se ha optado por aplicar un diseño orientado a aspectos sobre el caso práctico de un aplicación de terminal punto de venta (TPV), tal y como se presenta en [28].

Un TPV es una aplicación informática, utilizada en comercios, para registrar ventas y realizar pagos. Esta aplicación interactúa con varios servicios externos para realizar varias tareas, como el cálculo de impuestos o el control de inventario. Además, debería soportar múltiples terminales e interfaces (terminal en un navegador Web, interfaz de usuario implementada con las clases swing de Java, terminal táctil...). Un sistema TPV no es una aplicación exclusiva para un cliente concreto, sino que podrá ser adquirida por diferentes clientes cada uno con sus propias necesidades, por lo que necesitaremos mecanismos flexibles a la hora de especificar la lógica de negocio aplicable en determinados puntos del sistema, como al añadir una nueva línea de venta o calcular un descuento.

Se ha elegido este caso de estudio por ser un problema familiar, fácil de asimilar y que posee interesantes problemas de diseño. En concreto, nos hemos centrado en el caso de uso *Procesar Venta* cuya especificación se encuentra en el capítulo 6 de [28]. Partiendo de la implementación de las clases del dominio del problema extraída del capítulo 20 de [28], y cuyo diagrama de clases podemos ver en la Figura 7, vamos a ir abordando los problemas de diseño que en [28] se resuelven aplicando patrones *Gang of Four* (GoF) [21], y que ahora implementaremos mediante aspectos expresados en AspectJ. Además, se han implementado otra serie de funcionalidades secundarias que demuestran los beneficios de la programación orientada aspectos como logging (fichero log), tracing, persistencia y manejo de excepciones.

En la siguiente tabla se muestran los problemas de diseño abordados y los aspectos que los resuelven:

Problema de diseño	Aspecto (*.aj)
Acceso a servicio externo de contabilidad	AdaptadorContabilidad
Acceso a servicio externo de inventario	AdaptadorInventario
Acceso a servicio externo de autorización de pago con tarjeta	AdaptadorAutorizacionPagoTarjeta
Elegir un servicio externo	FactoriaServiciosSingleton
Política de descuentos	FactoriaEstrategiasSingleton EstrategiasDescuento EstrategiasCompuestas
Soporte a reglas de negocio conectables	FachadaMotorReglasSingleton ForzarUsoFachada
Separación Modelo-Vista	ObservadorVenta
Recuperación de fallos en acceso a servicios externos	ProxyContabilidad
Manejar familias de dispositivos	FactoriaDispositivosJavaPOSDefecto
Comportamiento dependiente del estado	TransicionesEstadoTPV
Tracing	TracingAspect
Logging	LoggingAspect
Comprobación de parámetros	ParameterCheckAspect

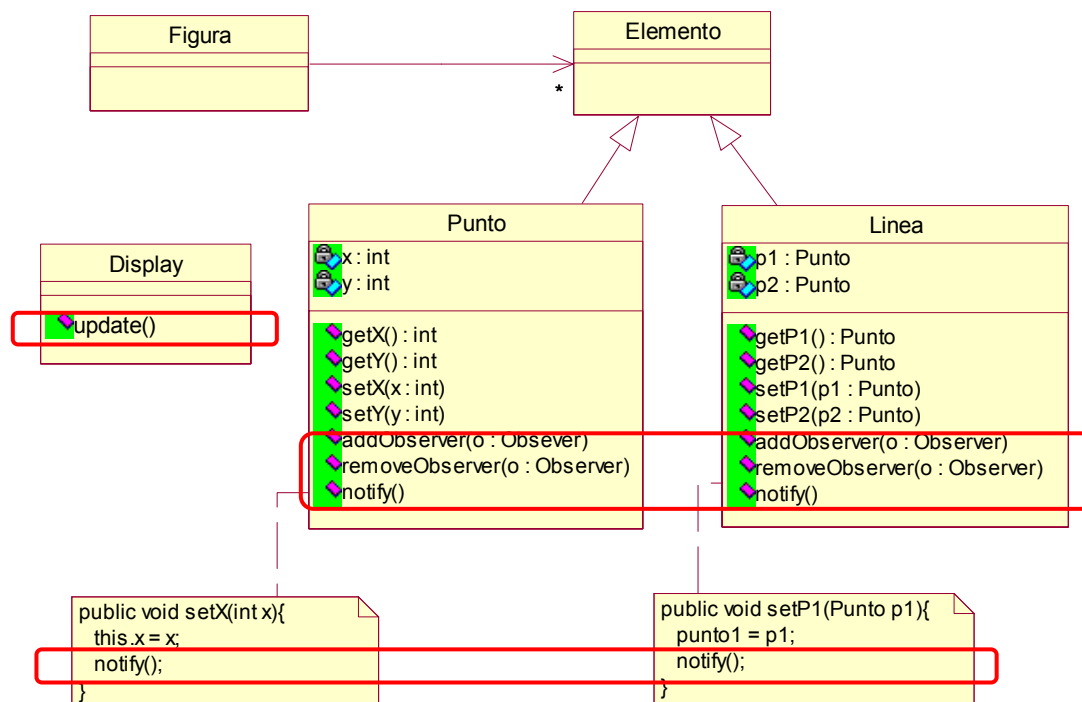


Figura 8: Crosscutting en el patrón *Observer*

El comportamiento cruzado introducido por algunos patrones hace que, el código relativo al patrón se mezcle con el código de la lógica de la aplicación, dando lugar a un código disperso y enredado. Las consecuencias directas de esta situación son:

- se hace difícil distinguir entre la instancia concreta del patrón y el código perteneciente a la lógica de la aplicación.
- añadir o eliminar un patrón del código de la aplicación resultará una tarea difícil (patrones no (des)conectables).
- surgen problemas a la hora de generar la documentación de la aplicación.
- la composición de patrones enmaraña aún más el código de la aplicación siendo difícil razonar sobre el código y seguir el rastro de una determinada instancia de un patrón, sobre todo si los patrones afectan a las mismas clases.

Puesto que la programación orientada aspectos resuelve el problema de las competencias transversales, parece lógico estudiar las ventajas o inconvenientes de implementar los patrones GoF mediante un lenguaje orientado a aspectos. Jan Hannemann y Gregor Kiczales han abordado dicha tarea implementando los patrones GoF con AspectJ [22] y comparando los resultados con la implementación tradicional de los patrones GoF en Java. El propósito, motivación y aplicabilidad de los patrones sigue siendo el mismo, lo único que variará será su estructura (en algunos patrones) y la forma de implementarlos. El código fuente de los patrones GoF implementados en AspectJ, junto con una serie de ejemplos ilustrativos del uso de cada patrón implementado tanto en Java como en AspectJ está disponible en la página Web de Jan Hannemann [31].

Como sabemos, los patrones asignan roles a las clases participantes que definen la funcionalidad de los participantes en el contexto del patrón. Por ejemplo, al aplicar el patrón *Observer* en el editor de figuras, las clases *Punto* y *Linea* juegan el rol de *Observables* (Sujeto) y *Display* de *Observador*.

Para estudiar la estructura cruzada que imponen algunos patrones, Jan Hannemann y Gregor Kiczales identifican dos tipos de roles [22]:

- **roles definidos** (*defining*): roles asignados a participantes que no tienen funcionalidad fuera del contexto del patrón, por tanto los roles definen completamente la funcionalidad del participante. Por ejemplo, en el patrón *Fachada* la clase *fachada* proporciona una interfaz para acceder a un subsistema y no suele tener un comportamiento propio.
- **roles superpuestos** (*superimposed*): roles asignados a participantes que tienen su propia funcionalidad fuera del contexto del patrón. En este caso, el rol no define por completo al participante, este tiene su propio significado. Por ejemplo, en el patrón *Obsever* los roles *Observable* y *Observador*.

Atendiendo a la clasificación anterior se tiene que, en los patrones que sólo presentan **roles definidos** cada clase representa un único interés. Por lo tanto, no existe comportamiento cruzado y la implementación con AspectJ no mejorará la implementación tradicional en Java.

En los patrones que sólo tienen **roles superpuestos** los participantes tienen su propia funcionalidad fuera del contexto del patrón de forma que, cuando una clase participa en una instancia de un patrón jugando un determinado rol superpuesto, la clase implementará dos intereses: su funcionalidad original y el comportamiento específico del rol dentro del patrón, dando lugar a un código enredado que repercute en la modularidad de la clase (clases *Punto*, *Línea* y *Display* en la Figura 8). Estos patrones que superponen comportamiento a los participantes son los que presentan beneficios a la hora de implementarlos con AspectJ, ya que se consigue una separación clara entre la funcionalidad del patrón y la funcionalidad original de la clase, modularizando el comportamiento general de un patrón en un aspecto abstracto y creando un subaspecto por cada instancia del patrón que asigna los roles e implementa el comportamiento específico de cada rol.

En los patrones que tienen **ambos tipos de roles** las ventajas de implementarlos con AspectJ dependerán del número de participantes implementando un tipo de rol concreto. Por ejemplo en el patrón *Composite*, donde los roles superpuestos *Composite* y *Leaf* se asocian a muchos participantes, los beneficios son notables.

La Tabla 19 [22] muestra los beneficios de implementar los patrones GoF con AspectJ, en términos de una serie de propiedades relacionadas con la modularidad:

- **localización**: el código de un patrón está claramente localizado en un aspecto.
- **reutilización**: se puede abstraer el comportamiento general de un patrón en un aspecto abstracto reutilizable.
- **composición transparente**: la composición de patrones no repercute negativamente en la estructura y el código de la aplicación.
- **(des)conectabilidad**: las clases participantes pueden añadirse o eliminarse de una instancia concreta de un patrón, sin modificar el código de las clases participantes. Es una consecuencia directa de la localización de la instancia de un patrón en un aspecto.

Pattern Name	Propiedades de modularidad				Tipos de Roles	
	Localización**	Reutilización	Composición Transparente	(Des) Conectable	Definido*	Superpuesto
Facade	Igual implementación para Java y AspectJ				Facade	-
Abstract Factory	no	no	no	no	Factory, Product	-
Bridge	no	no	no	no	Abstraction, Implementor	-
Builder	no	no	no	no	Builder, (Director)	-
Factory Method	no	no	no	no	Product, Creator	-
Interpreter	no	no	n/a	no	Context, Expression	-
Template Method	(si)	no	no	(si)	(AbstractClass) (ConcreteClass)	(AbstractClass) (ConcreteClass)
Adapter	si	no	si	si	Target, Adapter	Adaptee
State	(si)	no	n/a	(si)	State	Context
Decorador	si	no	si	si	Component, Decorator	Concrete Component
Proxy	(si)	no	(si)	(si)	(Proxy)	(Proxy)
Visitor	(si)	si	si	(si)	Visitor	Element
Command	(si)	si	si	si	Command	Commanding, Receiver
Composite	si	si	si	(si)	(Component)	(Composite, Leaf)
Iterator	si	si	si	si	(Iterator)	Aggregate
Flyweight	si	si	si	si	FlyweightFactory	Flyweight
Memento	si	si	si	si	Memento	Originator
Strategy	si	si	si	si	Strategy	Context
Mediator	si	si	si	si	-	(Mediator), Colleague
Chain of Responsibility	si	si	si	si	-	Handler
Prototype	si	si	(si)	si	-	Prototype
Singleton	si	si	n/a	si	-	Singleton
Observer	si	si	si	si	-	Subject, Observer

Tabla 19: Beneficios de implementar patrones GoF con AspectJ

En la tabla podemos ver como mientras para los patrones con roles definidos no existen beneficios a la hora de implementar los patrones con AspectJ, para los patrones con roles superimpuestos se obtienen beneficios en todas las categorías. Para los que tienen ambos tipos de roles las mejoras dependen del patrón en cuestión. Podemos resumir los datos de la tabla con las siguientes cifras:

- 17 de los 23 (74%) patrones se modularizan, permitiendo que el código de la instancia del patrón de alguno o todos los participantes se localice en un aspecto, lo que conlleva los siguientes beneficios:
 - una instancia de un patrón está perfectamente localizada y no se pierde o se dispersa en el código de la aplicación.

** Cuando aparece (si) en alguna propiedad, significa que la propiedad se cumple con alguna restricción.

* Cuando el rol aparece entre paréntesis, significa que no está claro si es definido o superpuesto.

- se invierten las dependencias, ahora el patrón depende de los participantes y no al revés, por lo que las clases participantes no conocen el rol que juegan dentro del patrón.
 - las clases participantes no contienen código del patrón por lo que podrán ser usadas en otros contextos sin necesidad de modificarlas, mejorando así la reutilización de estas clases.
 - las clases participantes podrán fácilmente abandonar o participar en una instancia de un patrón sin modificar su código, todos los cambios se realizan sobre el código del patrón haciendo las clases participantes conectables/desconectables.
 - que todo el código de la instancia de un patrón este localizado en un aspecto permite que la propia implementación del patrón sea (des)conectable.
 - mejoras en la documentación. La mera existencia de clases que contienen exclusivamente código relativo a un patrón nos informa de que patrones están siendo usados.
 - los casos problemáticos de composición de patrones se vuelven mejor estructurados cuando las instancias de los patrones son definidas en unidades modulares separadas.
- En 12 de esos 17 (52%) patrones se consigue abstraer el comportamiento general del patrón en un módulo reutilizable. Esto ocurre al abstraer el comportamiento general de un patrón (roles, operaciones conceptuales y protocolos de comunicación) en un aspecto abstracto reutilizable. Para crear una instancia concreta de un patrón se definen los participantes y se implementa el código específico de la instancia del patrón en un aspecto. Cambios en el comportamiento general del patrón no provocarán una cadena de cambios en los participantes
 - 14 (61%) patrones de esos 17 permiten una composición de patrones transparente de forma que múltiples patrones puedan compartir participantes, los mismos participantes pueden asumir diferentes roles en diferentes instancias del mismo patrón y múltiples instancias del mismo patrón en una misma aplicación no se confunden.

En los siguientes subapartados veremos las implementaciones en AspectJ de los patrones GoF [21] empleados para resolver los diferentes problemas de diseño que van surgiendo durante el desarrollo del sistema TPV y que se detallan en los capítulos 23 y 32 de [28].

4.2.1. Acceso a servicios externos (Patrón Adapter)

En el sistema TPV se quiere dar soporte a servicios externos cuya interfaz puede ser diferente. Estos servicios externos podrían ser un sistema de contabilidad, un sistema de inventario, un sistema de autorización de pagos, etc. Una posible solución a este problema es crear un adaptador que adapte la interfaz del servicio externo a la interfaz usada por la aplicación. Estamos ante un ejemplo típico del patrón *Adapter* (*Adaptador*) [21].

El propósito del patrón Adapter es convertir la interfaz de una clase en otra interfaz, que es la que esperan las clases clientes o implementar clases que puedan

interaccionar con clases de las que sólo se conoce su funcionalidad pero no su interfaz concreta. Esto es, permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles.

Implementación tradicional en Java.

La estructura del patrón *Adapter* aplicada al sistema TPV para dar soporte a un servicio externo de contabilidad y a un servicio externo de control de inventario es:

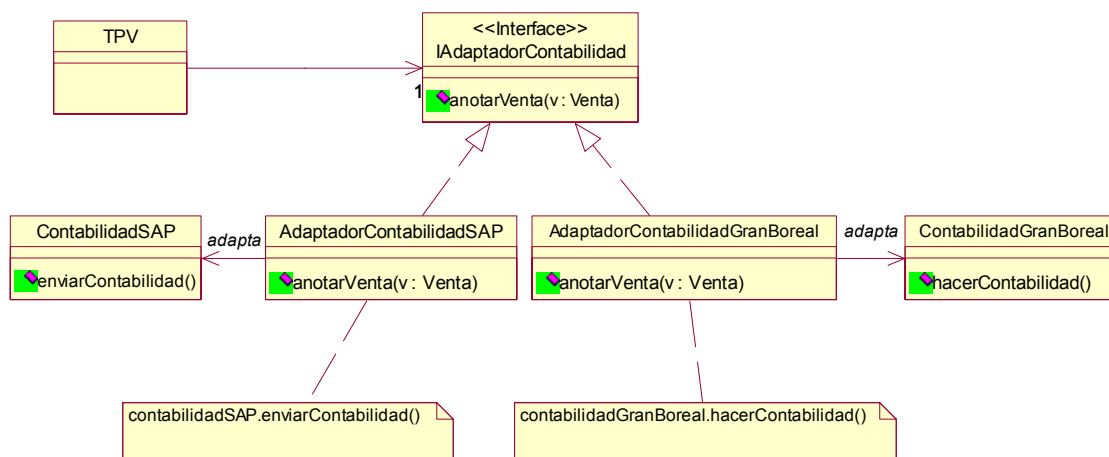


Figura 9: Adaptador Contabilidad

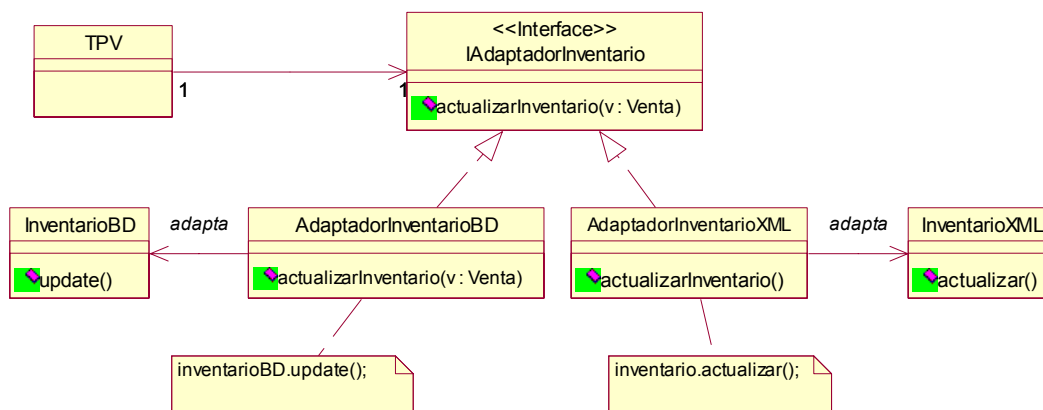


Figura 10: Adaptador Inventario

Las interfaces `IAdaptadorContabilidad` y `IAdaptadorInventario` son las interfaces que usa la aplicación para acceder a los servicios externos, pero las clases que nos ofrecen estos servicios (`ContabilidadSAP`, `ContabilidadGranBoreal`, `InventarioBD`, `InventarioXML`) no son compatibles con estas interfaces por lo que tendremos que crear adaptadores que adapten la interfaz de cada servicio externo a la esperada por la aplicación. La clase `TPV`, que representa al dispositivo terminal punto de venta y que juega el papel de controlador que recibe los eventos de la capa de presentación y los traslada a las clases de la capa del dominio, mantendrá referencias a cada una de las interfaces de acceso a servicios externos, de forma que el diagrama de

interacción para generar la contabilidad de una venta y actualizar el inventario podría ser el siguiente:

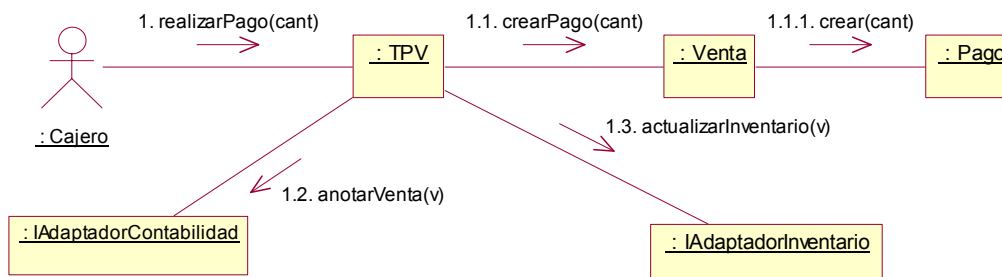


Figura 11: Diagrama de interacción de *realizarPago*

Implementación orientada aspectos con AspectJ

Veamos cual sería el resultado de implementar con AspectJ el patrón Adapter y resolver así el problema de acceder a servicios externos con diferentes interfaces. Usaremos un aspecto para cada instancia del patrón Adapter que se encargará de, mediante declaraciones de parentesco (`declare parents`), indicar que las clases que ofrecen el servicio externo implementan la interfaz esperada por la clase TPV y mediante declaraciones inter-tipo añadir a dichas clases los métodos adaptadores. De esta forma la clase adaptable y adaptador son la misma.

El aspecto que implementa el adaptador para el servicio externo de contabilidad es el siguiente:

```

public aspect AdaptadorContabilidad {

    declare parents:
        ContabilidadSAP implements IAadaptadorContabilidad;
    declare parents:
        ContabilidadGranBoreal implements IAadaptadorContabilidad;
    declare parents:
        ContabilidadLocal implements IAadaptadorContabilidad;

    public boolean ContabilidadSAP.anotarVenta(Venta v)
    {
        enviarContabilidad(v);
        return false;
    }
    public boolean ContabilidadGranBoreal.anotarVenta(Venta v)
    {
        hacerContabilidad(v);
        return true;
    }
}
  
```

El aspecto que implementa el adaptador para el servicio externo de control del inventario es:

```

public aspect AdaptadorInventario {

    declare parents: InventarioBD implements IAadaptadorInventario;
    declare parents: InventarioXML implements IAadaptadorInventario;

    public void InventarioBD.actualizarInventario(Venta v)
    {
  
```

```

        actualizar(v);
    }
    public void InventarioXML.actualizarInventario(Venta v)
    {
        update(v);
    }
}

```

Con los dos aspectos anteriores conseguimos que las clases que proporcionan los servicios externos implementen la interfaz esperada por la clase TPV sin efectuar modificaciones en dichas clases. Ahora las clases adaptables desempeñan también el rol de adaptador:

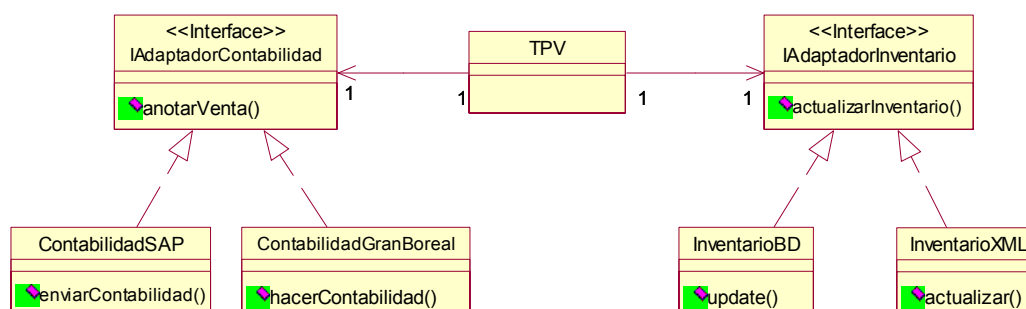


Figura 12: Clases adaptables y adaptadores

Las ventajas de la implementación del patrón *Adapter* con AspectJ frente a la implementación tradicional con Java son:

- el código resultante de implementar cada instancia del patrón *Adapter* está localizado en una única entidad (aspectos `AdaptadorContabilidad` y `AdaptadorInventario`). En cambio, en la implementación Java los métodos adaptadores están dispersos en varias clases.
- implementamos el patrón sin necesidad de crear nuevas clases ni modificar las existentes. En la implementación con Java teníamos que crear una clase adaptador por cada clase que ofrecía un servicio externo, en cambio en la solución orientada a aspectos, con un único aspecto adaptamos todas las clases que ofrecen cierto servicio.

4.2.2. Elegir un servicio externo (Patrones Factoría y Singleton)

Una vez resuelto el problema de acceder a servicios externos mediante el patrón *Adapter*, se plantea el problema de elegir un servicio externo concreto para cierta funcionalidad, esto es, elegir que adaptador concreto hay que crear. Una solución típica es usar el patrón **Factoría** [21], que define un objeto encargado de encapsular la lógica de creación de los adaptadores. Para determinar el tipo de adaptador existen muchas alternativas, de las cuales la más frecuente es leer el tipo de una fuente externa, como puede ser una base de datos, una propiedad del sistema, un fichero de configuración..., y cargar la clase adaptador dinámicamente, mediante reflexión. De esta forma

conseguimos crear diferentes tipos de adaptador sin modificar el código de la factoría, simplemente cambiando la fuente externa de donde leemos el tipo de adaptador a crear.

Implementación tradición en Java

La implementación tradicional de este patrón sería:

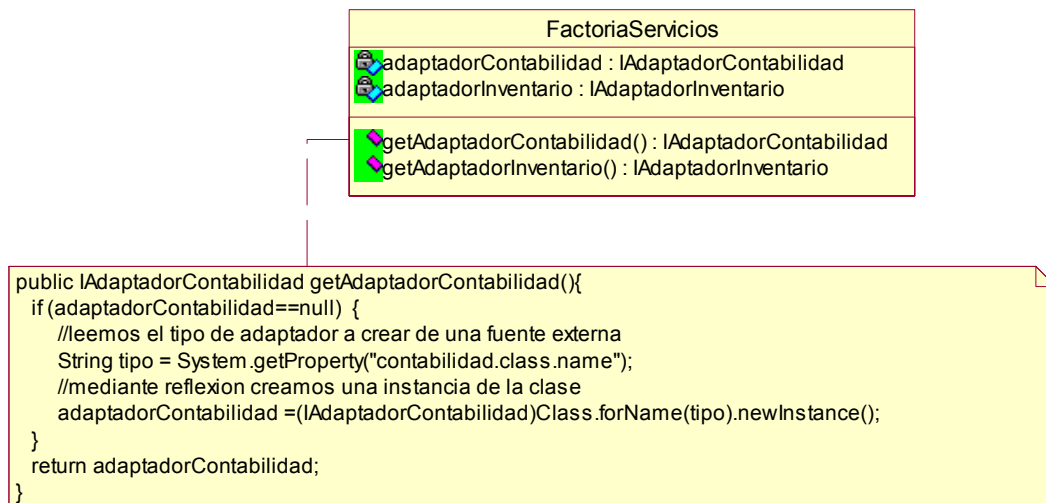


Figura 13: Factoría de servicios externos

Una vez creada la factoría de servicios, debemos planteamos quién creará el objeto factoría y como. Es evidente que sólo necesitaremos una instancia de esta clase para toda la aplicación y que son varios los lugares en la aplicación donde se invoca a los servicios externos, por lo que necesitaremos un acceso global a dicha instancia. Esta situación es resuelta por el patrón *Singleton* [21], definiendo un método estático en la clase afectada que devuelva la única instancia de la clase. La estructura de la clase singleton `FactoriaServicios` quedaría como:

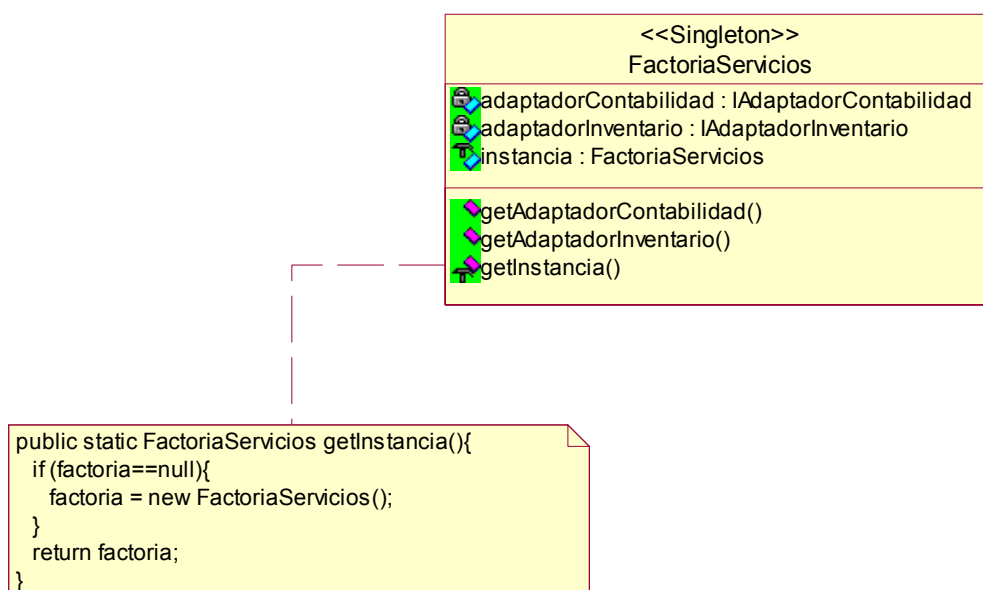


Figura 14: Factoría *singleton* de servicios externos

El controlador TPV es el encargado de inicializar los adaptadores llamando a los métodos correspondientes de la factoría. Para obtener la única instancia de la factoría se utiliza el método estático `getInstancia()`, como se puede ver en el siguiente diagrama de interacción:

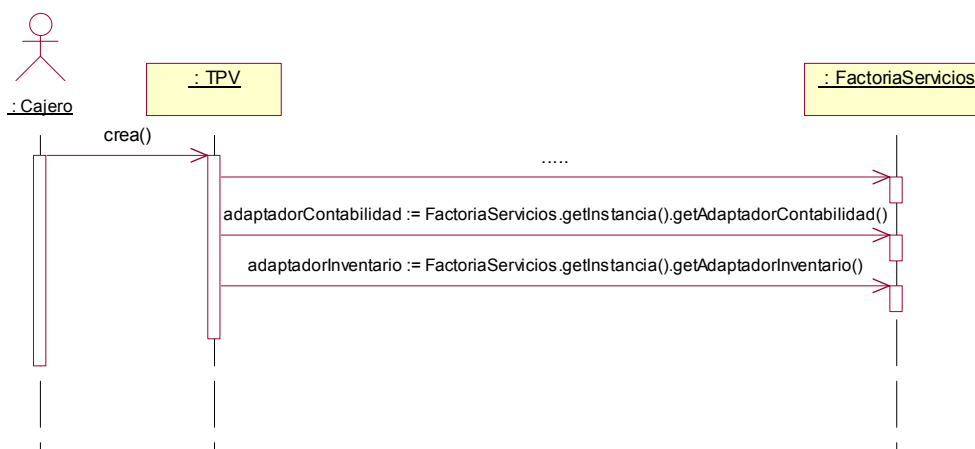


Figura 15: Uso de la factoría de servicios

Implementación orientada aspectos con AspectJ

Implementar el patrón *Factoría* con AspectJ no supone ningún beneficio, puesto que este patrón no introduce comportamiento cruzado sobre las clases participantes.

Veamos ahora la implementación del patrón *Singleton* con AspectJ. Vamos a hacer uso de la librería de patrones reutilizables implementados con AspectJ de [22], que nos proporciona un aspecto abstracto que implementa el comportamiento general del patrón *Singleton* de forma que puede ser reutilizado en las distintas instancias del patrón. En la siguiente figura se muestra la estructura de dicho aspecto:

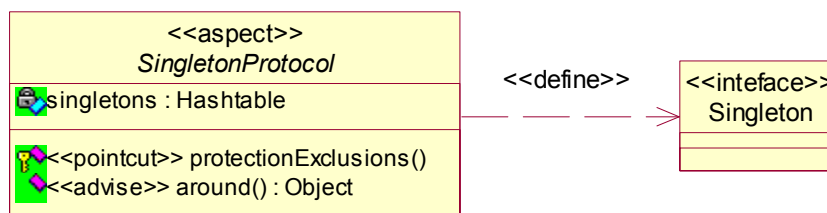


Figura 16: Aspecto *SingletonProtocol*

El perfil UML usado en la Figura 16 ha sido extraído de [25] y será utilizado en posteriores diagramas para representar aspectos. Algunos detalles del perfil no han podido ser reflejados debido a limitaciones de la herramienta utilizada para dibujar los diagramas. A continuación se muestra el código del aspecto *SingletonProtocol*:

```

public abstract aspect SingletonProtocol {

    /**
     * Almacena las instancias Singleton.
     */
    private Hashtable singletons = new Hashtable();

    /**

```

```

    * Esta interfaz define el rol Singleton.
    * Cualquier clase que quiera ser Singleton implementara esta
    * interfaz.
    */
    public interface Singleton {}

    /**
     * Define excepciones a la protección del constructor, permitiendo
     * que determinadas clases puedan acceder al constructor original
     */
    protected pointcut protectionExclusions();

    /**
     * Reemplaza las llamadas al constructor de una clase Singleton
     * para crear una única instancia bajo demanda, en lugar de crear
     * una nueva cada vez que se llama al constructor.
     */
    Object around():
        call((Singleton+).new(..)&& !protectionExclusions())
    {
        Class singleton=thisJoinPoint.getSignature().getDeclaringType();
        if (singletons.get(singleton) == null)
        {
            singletons.put(singleton, proceed());
        }
        return singletons.get(singleton);
    }
}

```

Para crear una determinada instancia del patrón Singleton tendremos que crear un subaspecto de SingletonProtocol que realice las siguientes tareas:

- especificar la clase *singleton*, esto es, la clase que implementa la interfaz Singleton mediante una declaración de parentesco (declare parents).
- sobrescribir el punto de corte protectionExclusions en el caso de que se desee que determinadas subclases de la clase *singleton* puedan acceder al constructor original, es decir, que subclases de una clase *singleton* no hereden esta propiedad.

La instancia del patrón *Singleton* para la clase FactoriaServicios será implementada por el siguiente aspecto:

```

public aspect FactoriaServiciosSingleton extends SingletonProtocol{

    declare parents: FactoriaServicios implements Singleton;

}

```

En el subaspecto anterior, no se sobrescribe el punto de corte protectionExclusions porque no se tiene ninguna subclase *no-singleton* de FactoriaServicios.

Con esta nueva implementación del patrón *Singleton*, para acceder a la única instancia de la clase, no tendremos que usar un método estático, sino que obtendremos una instancia mediante el constructor de la clase, como si se tratara de una clase normal. En el siguiente diagrama de interacción vemos como el TPV instancia al objeto de FactoriaServicios para crear los adaptadores:

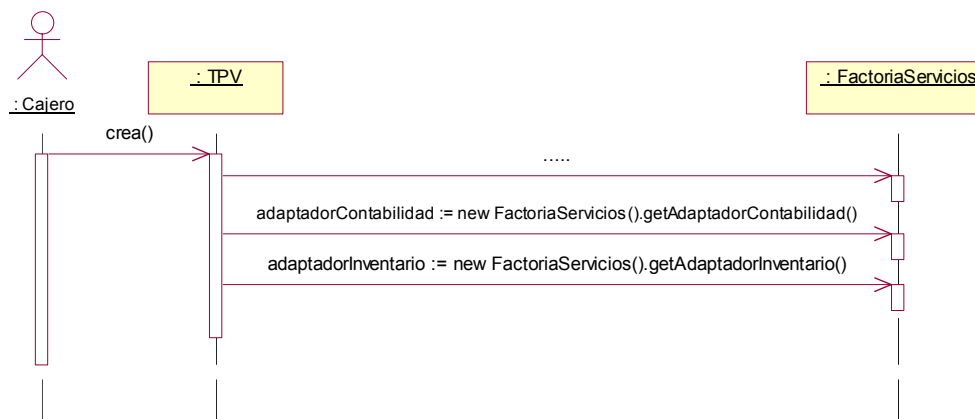


Figura 17: Uso de la factoría de servicios (AspectJ)

Las ventajas a la hora de implementar el patrón *Singleton* con AspectJ frente a la implementación tradicional en Java son:

- El código relativo al patrón queda totalmente encapsulado en los aspectos `SingletonProtocol` y `SingletonInstance`, lo que significa que no hemos tenido que modificar el código fuente de la clase `FactoriaServicios` para convertirla en una clase *Singleton*. En la implementación tradicional si que teníamos que modificar la clase `FactoriaServicios` para ocultar su constructor y crear el método estático que devolvía la única instancia de la clase.
- El funcionamiento del patrón *Singleton* es totalmente transparente al cliente de la clase `FactoriaServicios`, ya que no tenemos que utilizar un método especial (`getInstancia()`) para poder acceder a la única instancia de la clase, si no que seguimos usando la construcción `new()` como si se tratara de una clase normal.
- Para que la clase `FactoriaServicios` deje de ser *Singleton* basta con eliminar el subaspecto `SingletonInstance` del proceso de entretejido, no tendríamos que modificar la clase como ocurriría con la implementación tradicional. Se puede decir que ahora el patrón es (des)conectable.
- El código a escribir para implementar el patrón es mucho menor (subaspecto `SingletonInstance`). Recordar que el aspecto `SingletonProtocol` pertenece a una librería de patrones implementados en AspectJ [22], y no es escrito por nosotros.

4.2.3. Políticas de descuento (Patrones Estrategia y Composite)

Vamos a atacar otro problema de diseño descrito en [28]: las políticas de descuento. El sistema TPV debería disponer de una serie de políticas para fijar los precios que podrán ir cambiando de unas ventas a otras dependiendo de diversos factores. Esta situación motiva la utilización del patrón **Estrategia** (*Strategy*) [21] cuyo propósito es definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Si además es posible la aplicación simultánea de varias políticas de

fijación de precio entonces se tendrá que definir una política de resolución de conflictos para saber que descuento aplicar sobre una venta. Por ejemplo, se podría aplicar el descuento más favorable para el bolsillo del cliente o por el contrario se podría mirar por los intereses del establecimiento y aplicar el menor descuento posible. Además una venta no tiene porque ser consciente de si tiene asociadas una o varias políticas de descuento. Esta situación puede ser resuelta de forma elegante con el patrón *Composite* [21] que combina objetos en estructuras jerárquicas para representar jerarquías de parte-todo, permitiendo que los clientes traten de manera uniforme a los objetos hoja y a los compuestos. A continuación resolvemos el problema de las políticas de descuento implementando los patrones *Estrategia* y *Composite* sin aplicar aspectos.

Implementación tradicional en Java

Para el ejemplo del TPV podemos definir dos estrategias de descuento: un descuento según un porcentaje o un descuento absoluto si el total de una venta supera cierto umbral. La estructura del patrón *Estrategia* aplicada a este ejemplo sería:

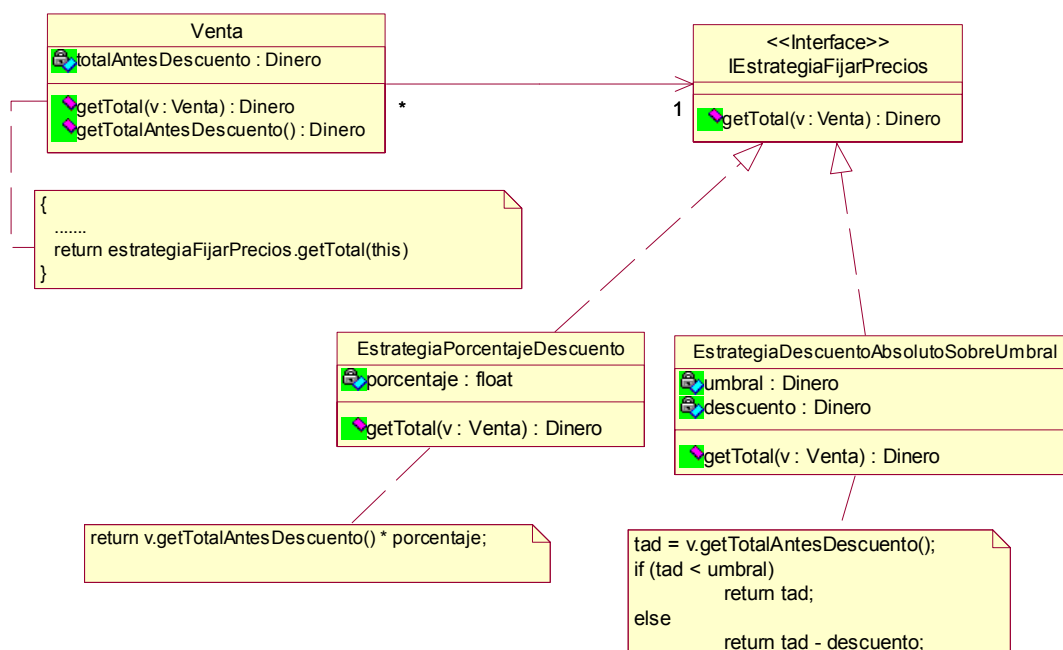


Figura 18: Estructura del patrón *Estrategia* para el sistema TPV

En el diagrama de clases anterior podemos ver una interfaz común a todas las estrategias (*IEstrategiaFijarPrecios*) y las estrategias concretas que implementan dicha interfaz. La clase *Venta* es el objeto de contexto que necesita una referencia a un objeto estrategia de forma que, cuando llamemos al método *getTotal* de *Venta*, delegue parte de su trabajo al objeto estrategia.

Una vez resuelto el problema de fijar los precios, nos volvemos a preguntar quien debería crear las estrategias de fijación de precios. Como en el ejemplo de los servicios externos, sería interesante crear una *factoría singleton* que encapsule la lógica de creación de estas estrategias. Para obtener el tipo de estrategia a crear, podemos usar un diseño dirigido por los datos que lea el tipo de estrategia y sus parámetros de alguna fuente externa, por ejemplo mediante propiedades del sistema. En la siguiente figura podemos ver la clase *FactoriaEstrategias* con métodos para obtener la estrategia

de fijación de precios a aplicar para cada situación (ventas tradicionales, ventas a estudiantes...)

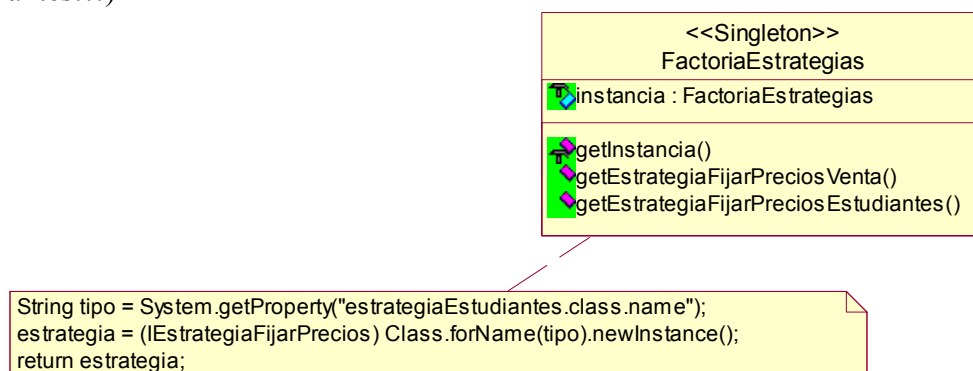


Figura 19: Factoría de estrategias de descuento

A continuación, para lograr que una venta no sea consciente de si tiene asociadas una o más estrategias de descuento y solucionar los conflictos entre estrategias, aplicamos el patrón *Composite* que hace que tanto los objetos hoja como los compuestos implementen la misma interfaz, añadiendo los métodos necesarios a los objetos compuestos para que gestionen sus objetos internos. La estructura de este patrón para el ejemplo del sistema TPV es la siguiente:

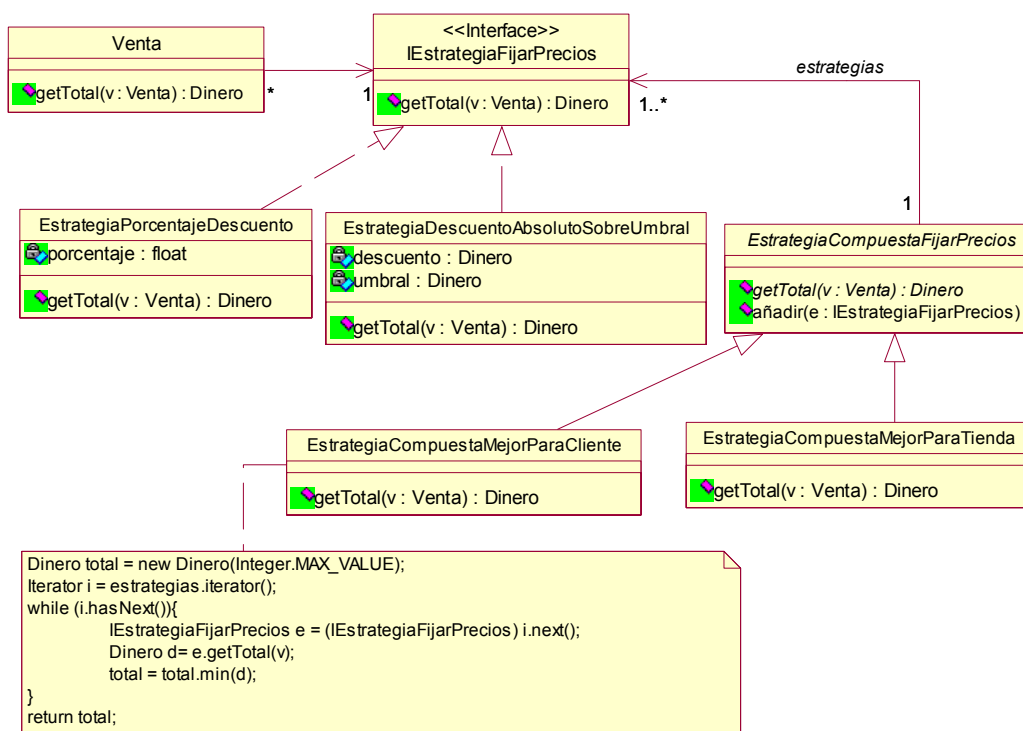


Figura 20: Estructura del patrón *Composite* y *Estrategia* para el sistema TPV

Implementación orientada aspectos con AspectJ

Vamos a implementar el mismo problema de diseño mediante aspectos. Para implementar el patrón *Estrategia* usaremos el aspecto `StrategyProtocol`,

perteneciente a la librería de patrones AspectJ reutilizables [22], que implementa el comportamiento general del patrón y cuya estructura es la siguiente:

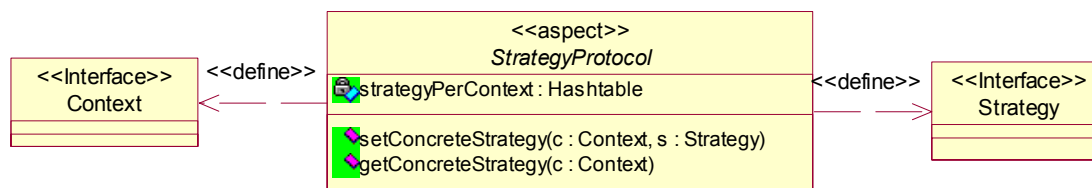


Figura 21: Aspecto *StrategyProtocol*

El aspecto anterior define dos interfaces (Context y Strategy) que representan los roles del patrón *Estrategia*: contexto y estrategia. Además incluye una tabla *hash* (strategyPerContext) para mantener la correspondencia entre los objetos de contexto y su estrategia asociada y los métodos para manipularla. El código del aspecto abstracto StrategyProtocol es el siguiente:

```

public abstract aspect StrategyProtocol {

    Hashtable strategyPerContext = new Hashtable();

    /**
     * Define el rol Estrategia
     */
    protected interface Strategy { }

    /**
     * Define el rol Contexto
     */
    protected interface Context { }

    /**
     * Asocia una estrategia con un contexto dado
     */
    public void setConcreteStrategy(Context c, Strategy s) {
        strategyPerContext.put(c, s);
    }

    /**
     * Devuelve la estrategia asociada a un contexto dado
     */
    public Strategy getConcreteStrategy(Context c) {
        return (Strategy) strategyPerContext.get(c);
    }
}
  
```

Para implementar una instancia del patrón con el fin de establecer la política de fijación de precios del sistema TPV, tendremos que crear un subaspecto de StrategyProtocol que realice las siguientes tareas:

- especificar las clases que participan en la instancia del patrón y los roles que desempeñan mediante declaraciones de parentesco (declare parents). La clase Venta es el contexto y las clases EstrategiaPorcentajeDescuento y EstrategiaDescuentoAbsolutoSobreUmbral.
- añadir funcionalidad a la clase Venta para que a la hora de calcular el total tenga en cuenta la estrategia de fijación de precios asociada. Para ello reemplazamos el método calcularTotal por una ejecución alternativa que aplique la estrategia

correspondiente y añadimos un atributo (`totalAntesDescuento`) para almacenar el precio de la venta antes de aplicar la estrategia de descuento.

- establecer la correspondencia entre una venta y su correspondiente estrategia de descuento en el momento de crear la instancia de la clase `Venta`. Para ello se usa un aviso *after* que captura las llamadas al constructor de la clase `Venta` y a la nueva venta le asocia una estrategia de descuento.

El código del subaspecto será el siguiente:

```
public aspect EstrategiasDescuento extends StrategyProtocol {

    //Establecemos los roles que juegan las clases en el patrón.
    declare parents: Venta implements Context;
    declare parents: EstrategiaPorcentajeDescuento implements Strategy;
    declare parents:
        EstrategiaDescuentoAbsolutoSobreUmbral implements Strategy;

    //Declaraciones inter-tipo sobre la clase Venta
    /**
     * Contiene el total antes de aplicar el descuento
     */
    private Dinero Venta.totalAntesDescuento;

    /**
     * Devuelve el total antes de aplicar el descuento
     */
    public Dinero Venta.getTotalAntesDescuento() {
        return totalAntesDescuento;
    }

    /**
     * Añadimos funcionalidad al método calcularTotal de la clase Venta
     * para que una vez calculado el total de la venta
     * se aplique el descuento correspondiente.
     */
    Dinero around(Venta v): call(Dinero Venta.calcularTotal()) && target(v) {

        //Calcula el total de la venta
        Dinero total = proceed(v);
        //Guardamos el total antes de proceder con el descuento
        v.totalAntesDescuento = total;
        //Obtenemos la estrategia asociada a la venta actual
        Strategy s = getConcreteStrategy(v);
        //Aplicamos la estrategia correspondiente
        total = ((IEstrategiaFijarPrecios) s).getTotal(v);
        v.setTotal(total);
        return total;
    }

    /**
     * Asociamos cada venta con una estrategia para fijar precios
     * en el momento de crear la instancia de Venta.
     */
    after(Venta v): execution( Venta.new(..) ) && target(v) {

        //Obtenemos la estrategia a través de la factoria
        Strategy s = (Strategy) new
            FactoriaEstrategias().getEstrategiaFijarPreciosVenta();
        //Asociamos la nueva venta con la estrategia
        setConcreteStrategy(v, s);
    }
}
```

Como consecuencia de tejer el aspecto `EstrategiasDescuento` con las clases de la aplicación, la estructura estática de algunas clases se ve alterada:

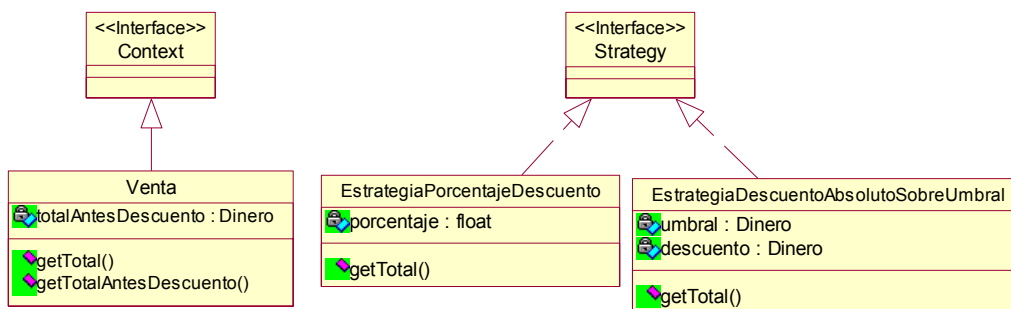


Figura 22: Consecuencias de tejer el aspecto *EstrategiasDescuento*

Para crear las estrategias de descuento hacemos uso de una *factoría singleton* (FactoriaEstrategias) cuya implementación es similar a la factoría de servicios vista en el apartado 4.2.2. por lo que omitimos los detalles de la implementación.

Las ventajas de implementar una instancia del patrón Estrategia con AspectJ frente a la implementación tradicional con Java son:

- Se consigue introducir la política de fijación de precios en el sistema TPV sin modificar el código fuente de la clase *Venta*, que es la que hace el papel de *Contexto* dentro del patrón *Estrategia*. Por lo tanto, esta clase podrá ser reutilizada en otros sistemas donde no se tengan en cuenta distintos criterios a la hora de establecer precios.
- Si deseamos eliminar las estrategias de descuento del sistema TPV basta con no incluir el aspecto *EstrategiasDescuento* en el proceso de entretejido. Por lo tanto el patrón es (des)conectable.

De nuevo utilizaremos un aspecto perteneciente a la librería de patrones AspectJ [22] para implementar un patrón GoF, en concreto el patrón *Composite*. El aspecto *CompositeProtocol* define el comportamiento general del patrón *Composite* y su estructura es la siguiente:

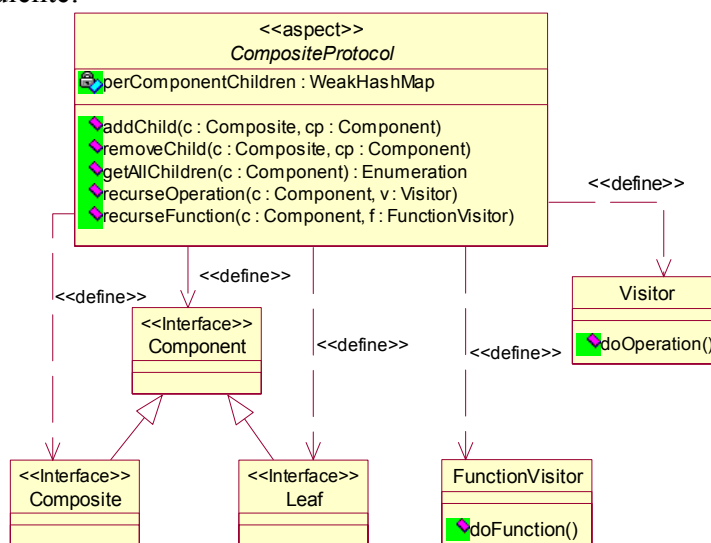


Figura 23: Aspecto *CompositeProtocol*

El aspecto define los roles (interfaces) del patrón: Component, Composite y Leaf. También mantiene la correspondencia entre un objeto compuesto y sus objetos internos (campo `perComponentChildren`) ofreciendo operaciones para manipular dicha correspondencia: `addChild`, `removeChild`, `getAllChildren`. Por último define dos interfaces para aquellos objetos visitantes que deseen recorrer la estructura jerárquica para aplicar cierta operación a cada objeto: `Visitor` y `FunctionVisitor`. El código del aspecto `CompositeProtocol` es el siguiente:

```
public abstract aspect CompositeProtocol {

    /**
     * Define el rol Component (Componente). El rol es público para permitir
     * a los clientes manejar objetos de este tipo.
     */
    public interface Component {}

    /**
     * Define el rol Composite (Compuesto). Este rol sólo es
     * usado dentro del contexto del patrón, por eso se define
     * como protected.
     */
    protected interface Composite extends Component {}

    /**
     * Define el rol Leaf (Hoja). Este rol sólo es
     * usado dentro del contexto del patrón, por eso se define
     * como protected.
     */
    protected interface Leaf extends Component {}

    /**
     * Almacena la correspondencia entre componentes y sus hijos
     */
    private WeakHashMap perComponentChildren = new WeakHashMap();

    /**
     * Retorna un vector con los hijos de un componente.
     * Método privador usado en addChild, removeChild y
     * getAllChildren
     */
    private Vector getChildren(Component s) {
        Vector children = (Vector)perComponentChildren.get(s);
        if ( children == null ) {
            children = new Vector();
            perComponentChildren.put(s, children);
        }
        return children;
    }

    /**
     * Añade un nuevo hijo a un objeto compuesto
     * @param composite - objeto compuesto
     * @param component - nuevo hijo
     */
    public void addChild(Composite composite, Component component) {
        getChildren(composite).add(component);
    }

    /**
     * Elimina un hijo de un objeto compuesto
     * @param composite - objeto compuesto
     * @param component - hijo a eliminar
     */
}
```

```

public void removeChild(Composite composite, Component component) {
    getChildren(composite).remove(component);
}

/**
 * Retorna un Enumerado con todos los hijos de un componente
 * @param composite - componente
 */
public Enumeration getAllChildren(Component c) {
    return getChildren(c).elements();
}

/**
 * Define una interfaz para objetos que deseen recorrer la estructura
 * jerárquica para aplicar cierta operación.
 * Estos visitantes son implementados por subaspectos
 * concretos y usados en el método recurseOperation(Component, Visitor).
 */
protected interface Visitor {

    /**
     * Método genérico que realiza cierta acción sobre un componente
     * @param c - componente sobre el que se realiza la acción
     */
    public void doOperation(Component c);
}

/**
 * Implementa la lógica de reenvío de métodos: si un método se aplica
 * sobre un objeto compuesto este lo remite a sus hijos.
 *
 * @param c - componente actual
 * @param v - el visitante que efectua la operación
 */
public void recurseOperation(Component c, Visitor v) {
    for (Enumeration enum = getAllChildren(c); enum.hasMoreElements(); ) {
        Component child = (Component) enum.nextElement();
        v.doOperation(child);
    }
}

/**
 * Define una interfaz para objetos que deseen recorrer la estructura
 * jerárquica para aplicar cierta función.
 * Estos visitantes son implementados por subaspectos concretos
 * y usados en el método recurseFunction(Component, Visitor).
 * La diferencia con respecto a la interfaz Visitor, es que FuncionVisitor
 * aplica una función sobre la estructura jerárquica, no un procedimiento.
 */
public interface FunctionVisitor {

    /**
     * Función genérica que realiza cierta operación sobre un componente.
     * @param c - componente sobre el que se aplica la función
     */

    public Object doFunction(Component c);
}

/**
 * Implementa la lógica de reenvío de métodos: si una función se aplica
 * sobre un objeto compuesto este la remite a sus hijos.
 * @param c - componente actual
 * @param fv - visitante que aplica la función
 * @return Enumerado con los resultados de aplicar la función sobre
 * la estructura jerárquica de un componente.
 */
public Enumeration recurseFunction(Component c, FunctionVisitor fv) {

```



```

        Vector results = new Vector();
        for (Enumeration enum = getAllChildren(c); enum.hasMoreElements(); ) {
            Component child = (Component) enum.nextElement();
            results.add(fv.doFunction(child));
        }
        return results.elements();
    }
}

```

Para implementar la instancia del patrón *Composite* creamos un subaspecto de *CompositeProtocol* que realice las siguientes tareas:

- especificar que clases participan en la instancia del patrón y los roles que juegan mediante declaraciones de parentesco (`declare parents`).
- Definir las políticas de fijación de precios de las estrategias compuestas mediante declaraciones inter-tipo. Para ello haremos uso de un objeto que implementa la interfaz *FunctionVisitor* y se encarga de aplicar el método `getTotal(Venta)` sobre cada una de las estrategias que forman parte de una estrategia compuesta.

El código del subaspecto será el siguiente:

```

public aspect EstrategiasCompuestas extends CompositeProtocol {

    //nodos hoja
    declare parents: EstrategiaPorcentajeDescuento implements Leaf;
    declare parents: EstrategiaDescuentoAbsolutoSobreUmbral implements Leaf;
    //nodos compuestos
    declare parents:
        EstrategiaCompuestaMejorParaCliente implements Composite;
    declare parents:
        EstrategiaCompuestaMejorParaTienda implements Composite;

    /**
     * Aplica la estrategia mejor para el cliente
     * @param v - venta
     * @return total de la venta
     */
    public Dinero EstrategiaCompuestaMejorParaCliente.getTotal(final Venta v)
    {
        /**
         * Mediante un objeto FunctionVisitor aplicamos la funcion
         * getTotal(Venta) sobre cada una de la estrategias que componen
         * la estrategia compuesta
         */
        Enumeration valores =
            EstrategiasCompuestas.aspectOf().recurseFunction(this,
                new FunctionVisitor() {
                    public Object doFunction(Component c) {
                        return ((IEstrategiaFijarPrecios) c).getTotal(v);
                    }
                });
        /**
         * De los valores devueltos pos recurseFunction
         * cogemos el de menor valor
         */
        Dinero minValor = new Dinero(Integer.MAX_VALUE);
        while (valores.hasMoreElements()) {
            Dinero d = (Dinero) valores.nextElement();
            minValor = minValor.min(d);
        }
        return minValor;
    }
}

```

```

/**
 * Aplica la estrategia mejor para el establecimiento
 * @param v - venta
 * @return total de la venta
 */
public Dinero EstrategiaCompuestaMejorParaTienda.getTotal(final Venta v)
{
    /**
     * Mediante un objeto FunctionVisitor aplicamos la funcion
     * getTotal(Venta) sobre cada una de la estrategias que
     * componen la estrategia compuesta
     */
    Enumeration valores =
        EstrategiasCompuestas.aspectOf().recurseFunction(this,
            new FunctionVisitor() {
                public Object doFunction(Component c) {
                    return ((IEstrategiaFijarPrecios) c).getTotal(v);
                }
            });

    /**
     * De los valores devueltos pos recurseFunction
     * cogemos el de mayor valor
     */
    Dinero maxValor = new Dinero(Integer.MIN_VALUE);
    while (valores.hasMoreElements()) {
        Dinero d = (Dinero) valores.nextElement();
        maxValor = maxValor.max(d);
    }
    return maxValor;
}
}

```

Tras aplicar el aspecto `EstrategiasCompuestas` la estructura de las clases implicadas será:

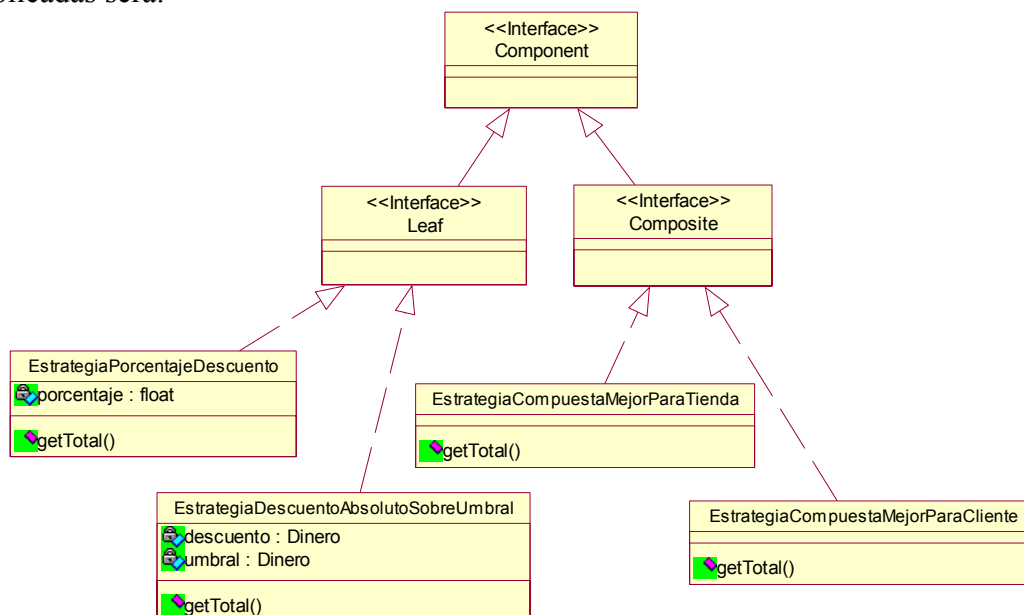


Figura 24: Consecuencias de tejer el aspecto *EstrategiasCompuestas*

Las ventajas de usar esta nueva implementación del patrón *Composite* frente a la implementación tradicional en Java son:

- La correspondencia entre un objeto compuesto y los objetos que lo forman y las operaciones para manipular dicha correspondencia son proporcionadas por el aspecto abstracto `CompositeProtocol`. En la implementación tradicional esta correspondencia la teníamos que implementar nosotros.
- El aspecto *CompositeProtocol* también nos facilita la tarea de recorrer la estructura jerárquica para aplicar cierta operación sobre cada nodo de la jerarquía, mediante los métodos `recurseOperation` y `recurseFunction` que toman como parámetro objeto `Visitor` o `FunctionVisitor` respectivamente. En la implementación con Java el código para recorrer la estructura y aplicar una operación a cada nodo es escrito por nosotros.

4.2.4. Soporte a reglas de negocio conectables (Patrón Fachada)

El siguiente problema de diseño que se nos presenta en [28] es dar soporte a reglas de negocio conectables en determinados puntos predecibles del escenario del sistema, como por ejemplo a la hora de introducir un artículo o cuando se va a realizar un pago. En concreto, estas reglas se encargarán de invalidar la acción si se cumplen ciertas condiciones. Por ejemplo, una regla de negocio podría ser que existe un límite a la hora de adquirir artículos en promoción, si se sobrepasa ese límite se debe anular la línea de venta.

Se desea ocultar el subsistema de reglas conectables detrás de un objeto que ofrezca un único punto de entrada al subsistema. A este objeto se le denomina *fachada* y suele ser un objeto *singleton*. El propósito del patrón **Fachada** [21] es proporcionar una interfaz unificada para un conjunto de interfaces de un subsistema, definiendo una interfaz de más alto nivel que hace que el subsistema sea más fácil de usar.

Implementación tradicional en Java

El siguiente diagrama ilustra el uso del patrón *Fachada* para ocultar el subsistema de reglas en el sistema TPV:

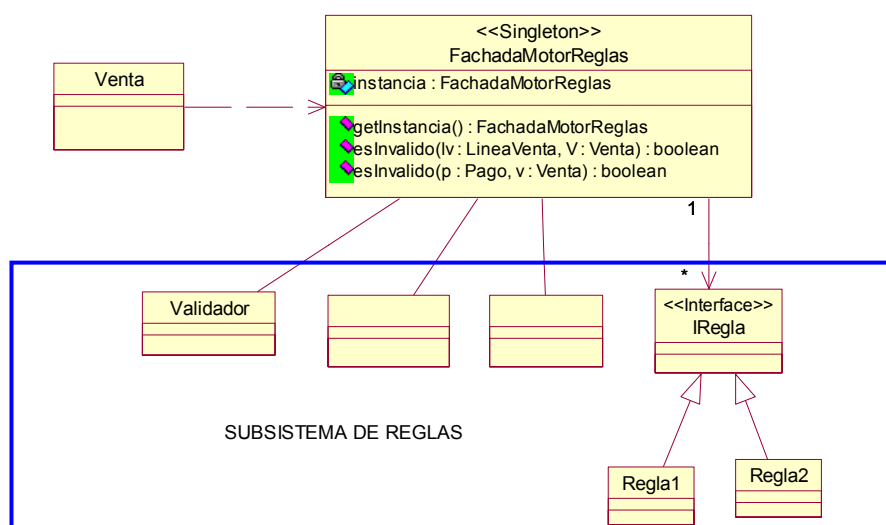


Figura 25: Fachada subsistema de reglas

La implementación del subsistema de reglas de negocio conectables se oculta detrás del objeto fachada, que es el punto de acceso al subsistema. El cliente del subsistema (clase `Venta`) únicamente verá el acceso público que ofrece el objeto fachada.

Implementación orientada aspectos con AspectJ

La implementación de la fachada con AspectJ es idéntica a la realizada en Java, ya que el patrón no introduce ningún comportamiento cruzado en las clases participantes, que pueda ser capturado mediante aspectos. Lo que sí podríamos conseguir mediante AspectJ, es crear una advertencia o error en tiempo de compilación cuando una clase cliente haga uso del subsistema de reglas a través de un acceso ilegal, no ofrecido por el objeto fachada. El código del aspecto que define la advertencia en tiempo de compilación es el siguiente:

```
public aspect ForzarUsoFachada {

    /**
     * Capturamos cualquier llamada a métodos del subsistema
     */
    pointcut llamadasIlegales():
        call(* (Validador || IRegla+ ).*(..));

    /**
     * Acceso legal mediante la fachada
     */
    pointcut fachada(): within(FachadaMotorReglas);

    /**
     * Declaramos un aviso en tiempo de compilación cuando se accede al
     * subsistema "Motor Reglas" mediante un acceso ilegal
     */
    declare warning: llamadasIlegales() && !fachada():
        "Uso ilegal del sistema Motor Reglas - usar un objeto
        FachadaMotorReglas";
}
```

4.2.5. Separación Modelo-Vista (Patrón Observer)

Otro requisito de diseño que aparece en [28] es, que cuando cambia el total de una venta se actualice la información de la ventana que muestra el total de la venta. Una solución es hacer que el objeto `Venta` envíe un mensaje a la ventana para que se actualice cuando el total de la venta cambia. Esta solución no es aceptable ya que violamos el principio de separación *Modelo-Vista*: las clases del modelo no deben conocer la existencia de los objetos que implementan la vista, de esta forma podemos cambiar la vista de nuestra aplicación sin modificar las clases del modelo.

Una opción más apropiada sería utilizar el patrón **Observer** (*Observador*) [21]. El propósito de este patrón es definir una dependencia uno a muchos entre objetos, de forma que cuando un objeto observable cambie de estado, se notifique dicho cambio a sus observadores para que realicen las acciones pertinentes.

Implementación tradicional en Java

Para el ejemplo del sistema TPV, el objeto `Venta` es el objeto *observable*, mientras que el objeto `VentanaTotal` es el *observador*. Cada vez que cambie el

precio total de la venta se notificará a la ventana y esta se actualizará automáticamente. El siguiente diagrama muestra la aplicación del patrón *Observer* al sistema TPV:

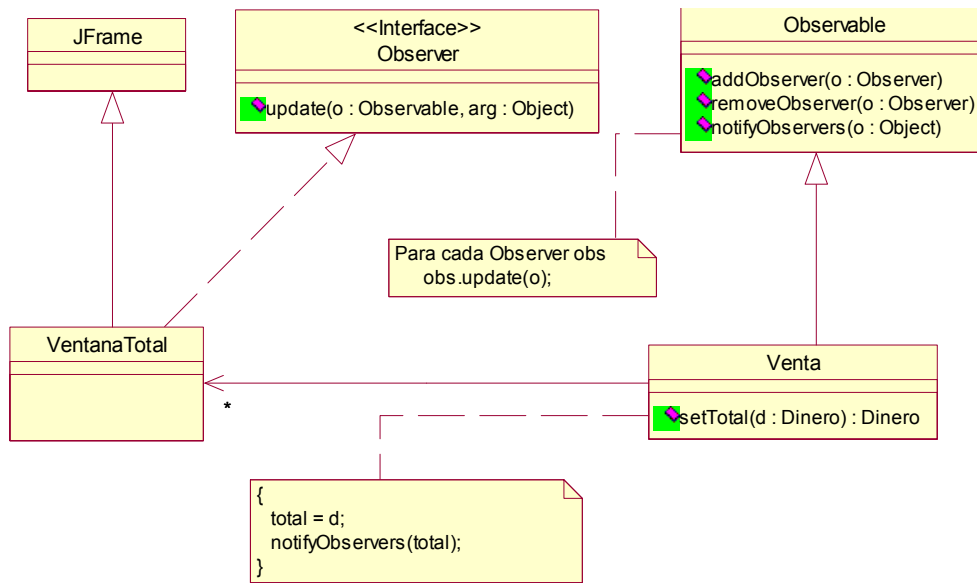


Figura 26: Estructura del patrón *Observer* para el sistema TPV

Se ha usado la interfaz *Observer* y la clase *Observable* proporcionadas por Java para implementar el patrón *Observer*. La clase *Venta* hereda de *Observable* e incluye notificaciones a los observadores (llamada al método *notifyObservers*) en los puntos donde cambia su precio total. La clase *VentanaTotal* implementa la interfaz *Observer*, especificando las acciones a realizar cuando recibe una notificación de cambio de precio total (método *update*).

Implementación orientada aspectos con AspectJ

Veamos como resultaría la implementación del patrón *Observer* con AspectJ. De nuevo usaremos la librería de patrones AspectJ de [22], que incluye un aspecto abstracto *ObserverProtocol* que implementa el comportamiento general del patrón *Observer*. El siguiente diagrama muestra la estructura del aspecto *ObserverProtocol*:

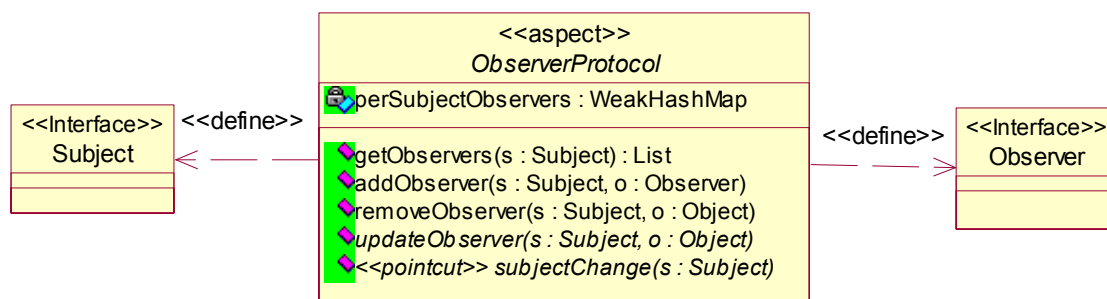


Figura 27: Aspecto *ObserverProtocol*

En el aspecto *ObserverProtocol*, la correspondencia entre objetos observables (*Subject*) y sus observadores (*Observer*) se implementa mediante el atributo *perSubjectObservers* y una serie de operaciones para manipular dicha

correspondencia: `getObservers`, `addObserver` y `removeObserver`. Además, el aspecto define un punto de corte abstracto, `subjectChange`, que será implementado por los subaspectos especificando los cambios de estado de un sujeto que deben ser notificados a sus observadores y un método abstracto, `updateObserver`, que también tendrá que ser implementado por los subaspectos indicando las acciones a realizar por los observadores cuando reciben una notificación de cambio de estado. Nos podríamos plantear porque el método `updateObserver` se define en el aspecto y no dentro de la interfaz `Observer`. La razón es, porque de esta forma un objeto `Observer` puede participar en distintas relaciones sujeto-observador con diferentes métodos de actualización. Si el método estuviera en la interfaz `Observer`, aunque el objeto observara distintos sujetos, la actualización siempre sería la misma. El código del aspecto `ObserverProtocol` es el siguiente:

```
public abstract aspect ObserverProtocol {

    /**
     * Define el rol Subject (Sujeto).
     */
    protected interface Subject { }

    /**
     * Define el rol Observer (Observador).
     */
    protected interface Observer { }

    /**
     * Implementa la correspondencia entre Sujeto y Observadores.
     * Para cada sujeto se guarda una LinkedList con sus observadores.
     */
    private WeakHashMap perSubjectObservers;

    /**
     * Retorna los observadores de un sujeto particular.
     * Este método es usado internamente, no es público.
     * @param subject - el sujeto a consultar
     * @return lista de observadores
     */
    protected List getObservers(Subject subject) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers = (List)perSubjectObservers.get(subject);
        if (observers == null) {
            observers = new LinkedList();
            perSubjectObservers.put(subject, observers);
        }
        return observers;
    }

    /**
     * Añade un observador a un Sujeto.
     * @param s - sujeto al que añadir un nuevo observador
     * @param o - nuevo observador
     */
    public void addObserver(Subject subject, Observer observer) {
        getObservers(subject).add(observer);
    }

    /**
     * Elimina un observador de un sujeto.
     * @param s - sujeto al que eliminar un observador
     * @param o - observador a eliminar
     */
    public void removeObserver(Subject subject, Observer observer) {
```

```

        getObservers(subject).remove(observer);
    }

    /**
     * Captura los cambios de estado del sujeto que
     * deben ser notificados a los observadores.
     * Debe ser implementado por los subaspectos.
     */
    protected abstract pointcut subjectChange(Subject s);

    /**
     * Este aviso notifica a los observadores los cambios
     * de interés del sujeto, para que se actualicen.
     * @param subject - sujeto que cambia
     */
    after(Subject subject): subjectChange(subject) {
        Iterator iter = getObservers(subject).iterator();
        while ( iter.hasNext() ) {
            updateObserver(subject, ((Observer)iter.next()));
        }
    }

    /**
     * Define como se actualizan los observadores cuando se produce
     * un cambio de interés en el sujeto.
     * Debe ser implementado por los subaspectos.
     * @param subject - sujeto que cambia
     * @param observer - observador a ser notificado
     */
    protected abstract void updateObserver(Subject subject, Observer observer);
}

```

Para implementar una instancia del patrón *Observer* que actualice la ventana que muestra el total de una venta cuando este cambia, debemos crear un subaspecto que realice las siguientes tareas:

- Definir los roles de las clases que intervienen en el patrón mediante declaraciones de parentesco (`declare parents`). La clase *Venta* juega el rol *Subject* y la clase *VentanaTotal* el rol *Observer*.
- Implementar el punto de corte `subjectChange` para que capture los puntos de enlace donde cambia el total de una venta.
- Implementar el método `updateObserver` para que la ventana muestre el nuevo precio total de la venta.

El código del subaspecto que implementa este comportamiento es el siguiente:

```

public aspect ObservadorVenta extends ObserverProtocol{

    //Definimos los roles que juegan las clases en el patrón.
    declare parents: Venta implements Subject;
    declare parents: VentanaTotal implements Observer;

    /**
     * Captura las llamadas al método setTotal de la
     * clase Venta y expone el sujeto.
     */
    protected pointcut subjectChange(Subject subject):
        call(void Venta.setTotal(Dinero)) && target(subject);

    /**
     * Actualiza la ventana con el nuevo precio de la venta
     */
}

```

```

protected void updateObserver(Subject subject, Observer observer) {
    java.text.DecimalFormat df =
        new java.text.DecimalFormat("#,##0.00");
    Venta v = (Venta) subject;
    VentanaTotal vt = (VentanaTotal) observer;
    vt.actualizarTotal(df.format(v.getTotal().getCantidad()));
}
}

```

Las ventajas de implementar el patrón *Observer* con aspectos con respecto a la implementación clásica con Java son:

- Se consigue encapsular el comportamiento cruzado que introduce el patrón *Observer* en las clases participantes en una única entidad, el subaspecto *ObservadorVenta*. Por lo tanto, se implementa el patrón sin necesidad de modificar el código fuente de las clases *Venta* y *VentanaTotal*, al contrario de lo que sucedía en la implementación con Java.
- Para añadir o eliminar la instancia del patrón *Observer* del sistema TPV basta con incluir o no el subaspecto en el proceso de entretejido con las clases de la aplicación. La instancia del patrón es (des)conectable.

4.2.6. Recuperación de fallos en servicios externos (Patrón Proxy)

Se desea gestionar el mantenimiento de los servicios externos usados por el sistema TPV ante posibles fallos, en concreto el sistema de contabilidad, que se encarga de registrar las ventas efectuadas en el terminal a través de un servicio externo. Para ello, se dispondrá de un servicio local que será invocado cuando se produzcan fallos en el servicio externo. Este servicio local, se encargará de almacenar las ventas de forma local (en un fichero, en una BD...) para poder reenviarlas al servicio de contabilidad externo cuando vuelva a estar disponible. Una forma elegante de implementar el mantenimiento de servicios externos es mediante el uso del patrón **Proxy** [21]. El propósito de este patrón es proporcionar un representante o sustituto de un objeto para controlar el acceso a este. Para ello, se introduce un nivel de indirección mediante el objeto *proxy*, que implementa la misma interfaz que el objeto real que representa, consiguiendo así controlar el acceso a este objeto.

Implementación tradicional en Java

Para el ejemplo del sistema TPV, la interfaz que debe implementar el proxy es *IAdaptadorContabilidad*, y su función será llamar al servicio externo (objeto real) y en caso de error, invocar al servicio local de contabilidad que almacenará la venta en un fichero mediante serialización. En los siguientes diagramas podemos ver la estructura y el comportamiento del patrón *Proxy* para el sistema de contabilidad del TPV:

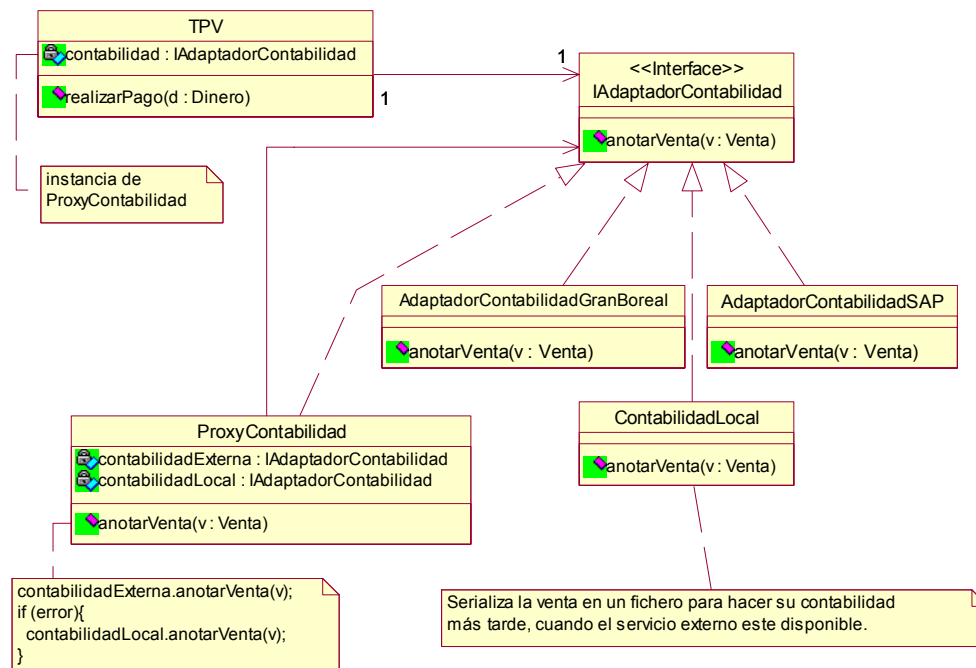
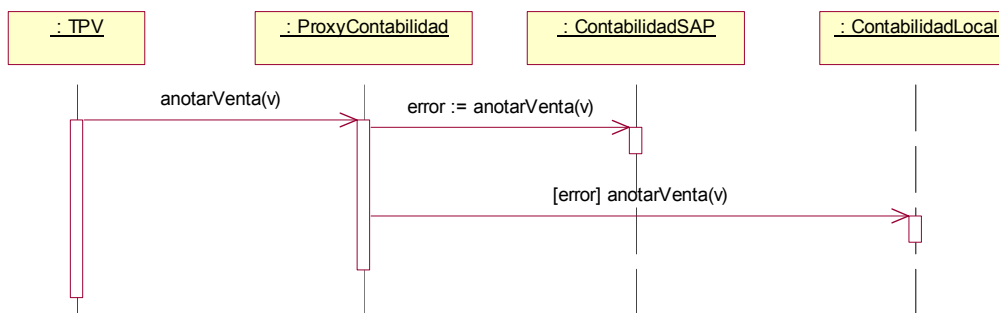
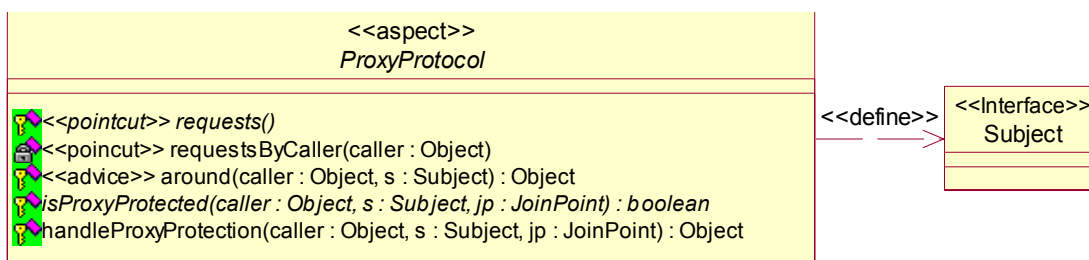
Figura 28: Estructura del patrón *Proxy* para el sistema TPV

Figura 29: Uso del objeto proxy

La clase TPV no conoce la existencia del proxy, ya que envía el mensaje `anotarVenta(v)` como si se tratara de un servicio de contabilidad externo.

Implementación orientada aspectos con AspectJ

A continuación se implementa el mantenimiento del sistema de contabilidad del TPV utilizando AspectJ. Nos basaremos en el aspecto reusable `ProxyProtocol` perteneciente a la librería de [22] y que encapsula el comportamiento general del patrón *Proxy*. En la Figura 30 se puede ver la estructura de este aspecto:

Figura 30: Aspecto *ProxyProtocol*

En el aspecto `ProxyProtocol` se define el rol `Subject` que representa al objeto real y comportamiento abstracto que debe ser implementado por los subaspectos que implementen una instancia del patrón. En concreto, el punto de corte `requests` que debe especificar los accesos al objeto real que deben ser controlados por el objeto proxy, el método `isProxyProtected` que indica si una petición al objeto real debe ser manejada o no por el proxy y el método `handleProxyProtection` que debe implementar las acciones a realizar por el proxy para manejar cierta petición al objeto real. El código del aspecto `ProxyProtocol` es el siguiente:

```
public abstract aspect ProxyProtocol {

    /**
     * Define el rol Sujeto.
     */
    protected interface Subject {}

    /**
     * Captura los accesos (peticiones al Sujeto) que deberían ser
     * controlados por el objeto proxy.
     */
    protected abstract pointcut requests();

    /**
     * Extiende el punto de corte requests() para exponer el objeto invocador
     * Es privado, por lo tanto para uso interno del aspecto.
     */
    private pointcut requestsByCaller(Object caller):
        requests() && this(caller);

    /**
     * Implementa la lógica del proxy. Si la petición enviada al Sujeto
     * debe ser controlada por el Proxy (isProxyProtected()), se invoca
     * al método handleProxyProtection()
     *
     * @param caller el objeto que realiza la petición
     * @param subject el objeto receptor de la petición
     */
    Object around(Object caller, Subject subject):
        requestsByCaller(caller) && target(subject) {

        if (! isProxyProtected(caller, subject, thisJoinPoint) )
            return proceed(caller, subject);
        return handleProxyProtection(caller, subject, thisJoinPoint);
    }

    /**
     * Indica si la petición deber ser manejada por el proxy o no.
     *
     * @param caller objeto que realiza la petición
     * @param subject objeto receptor de la petición
     * @param joinPoint punto de enlace asociado a la petición
     * @return true si la petición deber ser controlada por el proxy y false
     * en caso contrario
     */
    protected abstract boolean isProxyProtected(Object caller,
        Subject subject,
        JoinPoint joinPoint);

    /**
     * Acciones a realizar por el proxy sobre una determinada petición. La
     * opción mas usada suele ser delegar en otro objeto. Este metodo no es
     * abstracto, se da una implementación por defecto, para que así
     * los subaspectos no estén obligados a implementar el metodo.
     *
     * @param caller objeto que realiza la petición
     * @param subject objeto receptor de la petición

```

```

* @param joinPoint punto de enlace asociado a la petición
* @return un valor alternativo al normal
*/
protected Object handleProxyProtection(Object caller,
                                       Subject subject,
                                       JoinPoint joinPoint) {

    return null;
}
}

```

Para crear un proxy que gestione los fallos producidos por el servicio externo de contabilidad en el sistema TPV se tendrá que crear un subaspecto de ProxyProtocol que realice las siguientes tareas:

- Indicar la clase o clases que juegan el rol de Sujeto, que serán las clases que implementen la interfaz IAdaptadorContabilidad.
- Especificar que peticiones al Sujeto deben ser controladas por el proxy (punto de corte requests), en concreto las llamadas al método anotarVenta.
- Especificar las acciones a realizar por el proxy para manejar cierta petición (método handleProxyProtection). En concreto, el proxy enviará una petición al servicio externo de contabilidad para registrar una venta, y en el caso de que el servicio falle, se almacenará dicha venta en un fichero local mediante serialización y se volverá a intentar más tarde cuando el servicio externo vuelva a estar disponible.

El código del subaspecto anterior podría ser el siguiente:

```

public aspect ProxyContabilidad extends ProxyProtocol {

    //Define el rol Sujeto
    declare parents: IAdaptadorContabilidad implements Subject;

    /**
     * Especifica las peticiones que deberían ser protegidos por el proxy,
     * en este caso todas las llamadas al metodo
     * IAdaptadorContabilidad.anotarVenta(..)
     */
    protected pointcut requests():
        call(* IAdaptadorContabilidad.anotarVenta(..)) &&
        !within(ProxyContabilidad);

    /**
     * Chequea si la petición deber ser manejada por el proxy o no,
     * Para nuestro caso siempre.
     */
    protected boolean isProxyProtected(Object caller,
                                       Subject subject,
                                       JoinPoint joinPoint) {

        return true;
    }

    /**
     * Implementa la labor del proxy de delegar en el servicio local si
     * el servicio externo de contabilidad falla.
     */
    protected Object handleProxyProtection(Object caller,
                                       Subject subject,
                                       JoinPoint joinPoint) {

        Object[] args = joinPoint.getArgs();
        Venta v = (Venta) args[0];
        IAdaptadorContabilidad cont = (IAdaptadorContabilidad) subject;
    }
}

```

```

/*
 * Si se produce un error al llamar al servicio externo
 * de contabilidad, entonces se registra la venta en un fichero
 * para intentarlo mas tarde.
 */
if (!cont.anotarVenta(v)) {

    try{
        String fichero = System.getProperty("fichContabilidad");
        FileOutputStream fos = new FileOutputStream(fichero);
        ObjectOutputStream out = new ObjectOutputStream(fos);
        // Escribimos la venta en el fichero
        out.writeObject(v);

    }catch(Exception e){
        e.printStackTrace();
    }

    return null;
}

//Para poder serializar la clase Venta
declare parents: Venta implements Serializable;
declare parents: Dinero implements Serializable;
declare parents: LineaVenta implements Serializable;
declare parents: Producto implements Serializable;
declare parents: itemID implements Serializable;
declare parents: Pago implements Serializable;
}

```

Las ventajas de implementar el patrón Proxy con AspectJ frente a la implementación clásica con Java son:

- Todo el código relacionado con el patrón *Proxy* se encuentra en los aspectos *ProxyProtocol* y *ProxyContabilidad*. Por lo tanto se ha logrado implementar el patrón sin tener que modificar ninguna otra clase de la aplicación. En la implementación clásica mediante Java si que era necesario modificar ciertas clases como por ejemplo la clase *FactoriaServicios* para que devolviera una instancia del proxy, las clases *Producto*, *Venta*, *Pago...* para que implementaran la interfaz *Serializable* y así poder serializar un objeto de la clase *Venta*. En la solución con AspectJ todas estas acciones son llevadas a cabo por el subaspecto *ProxyContabilidad*.
- Para conectar/desconectar el patrón de la aplicación bastará con incluir o no el subaspecto *ProxyContabilidad* en el proceso de entretejido. En cambio en la implementación clásica con Java, para eliminar la instancia del patrón del código de la aplicación, habrá que deshacer los cambios mencionados en el punto anterior. Con AspectJ, el patrón es (des)conectable.

4.2.7. Manejar familias de dispositivos (Patrón Factoría Abstracta)

Como se describe en [28], un sistema TPV interactúa con una serie de dispositivos físicos que forman parte de un terminal de punto de venta, como pueden ser el cajón de la caja, el dispensador de moneda, etc. Existe un estándar industrial, *UnifiedPOS* que define las interfaces orientadas a objetos para estos dispositivos, siendo *JavaPOS* la especificación del estándar para Java. Los fabricantes de terminales de punto de venta (IBM, NCR...) proporcionan las clases que implementan las interfaces de *JavaPOS*, y que ayudaran a controlar los dispositivos físicos del terminal adquirido.

Si se opta por comprar terminales de distintos fabricantes tendremos distintas implementaciones de JavaPOS y dependiendo del terminal en cuestión, se usará una implementación u otra. Esta situación motiva la utilización del patrón **Factoría Abstracta** [21], cuyo objetivo es proporcionar una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. Para cada familia de objetos se creará una factoría concreta.

Implementación tradicional en Java

La interfaz `IFactoriaDispositivosJavaPOS` es la factoría abstracta, para cada fabricante se implementará una factoría concreta que implementa dicha interfaz. Si además se quieren tener implementaciones por defecto de los métodos de la factoría abstracta para que sean heredadas por las factorías concretas, tendremos que crear una nueva clase (`FactoriaDispositivosJavaPOS`) que implemente la interfaz `IFactoriaDispositivosJavaPOS` con las implementaciones por defecto de los métodos de la factoría abstracta. Esta clase también contendrá un método estático (`getInstancia`) que leerá de una fuente externa la familia de dispositivos a usar. El siguiente diagrama muestra la estructura del patrón para el ejemplo mencionado:

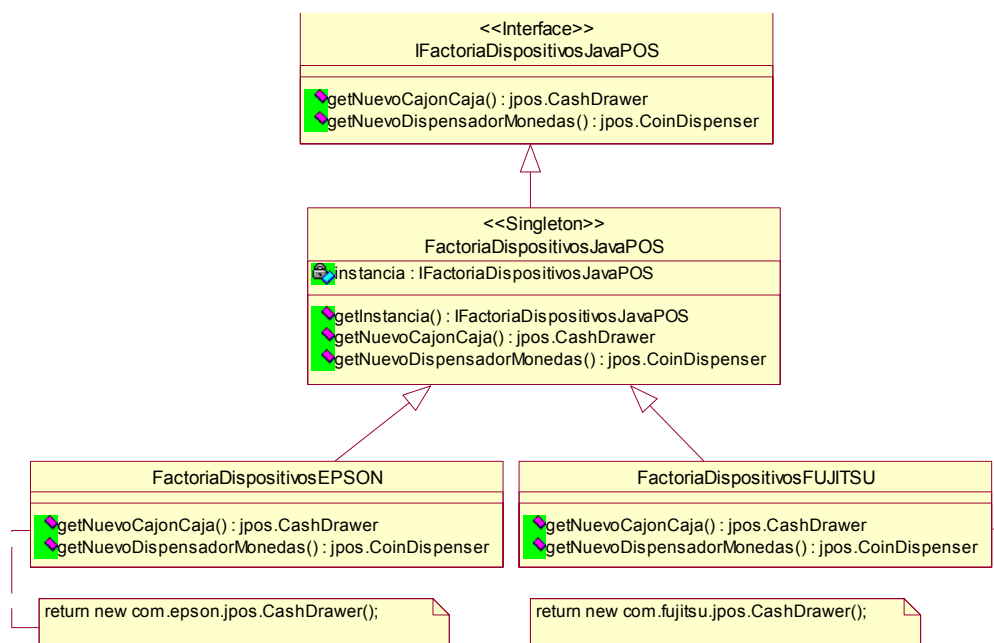


Figura 31: Factoría abstracta de dispositivos JavaPOS

Implementación orientada aspectos con AspectJ

La implementación clásica del patrón *Factoría Abstracta* realiza una separación de intereses clara, encapsulando cada familia de dispositivos en una clase distinta, no existiendo ningún comportamiento cruzado que puede ser capturado con aspectos. Aún así, veamos como con AspectJ podemos añadir implementación a las interfaces mediante declaraciones inter-tipo. Para establecer implementaciones por defecto para los métodos de la factoría abstracta `IFactoriaDispositivosJavaPOS`, basta con crear un aspecto que mediante métodos inter-tipo defina las implementaciones por defecto de los métodos de la interfaz. Con respecto a la implementación clásica con Java, se evita crear la clase `FactoriaDispositivosJavaPOS` por lo que las factorías

concretas implementan directamente la interfaz `IFactoriaDispositivosJavaPOS` en lugar de heredar de `FactoriaDispositivosJavaPOS`. De esta forma las factorías concretas podrán participar en otras relaciones de herencia, aunque no es un caso usual.

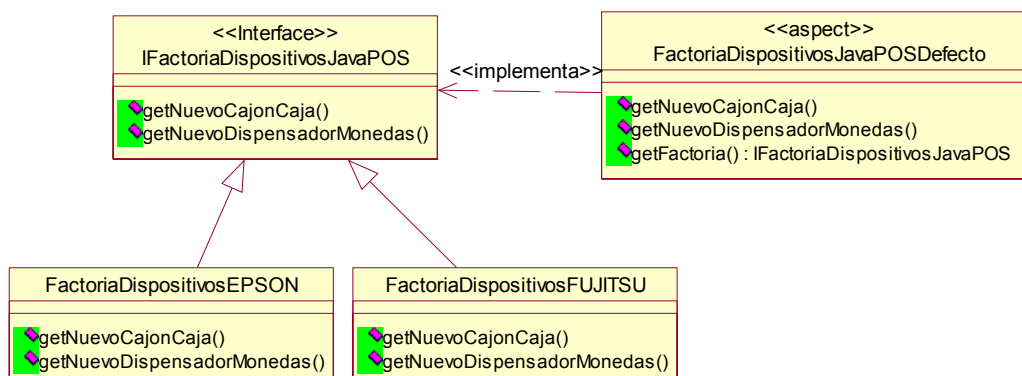


Figura 32: Factoría Abstracta de dispositivos JavaPOS (AspectJ)

4.2.8. Comportamiento dependiente del estado (Patrón Estado)

El comportamiento del sistema TPV variaría dependiendo del estado en el que se encuentre en cada instante. Por ejemplo, si el terminal punto de venta está apagado, la única acción posible es conectarlo. Una vez conectado, el cajero introducirá su número de identificación y es entonces cuando el sistema está listo para comenzar a registrar los productos de una nueva venta. La Figura 33 representa el diagrama de estados para nuestro terminal punto de venta.

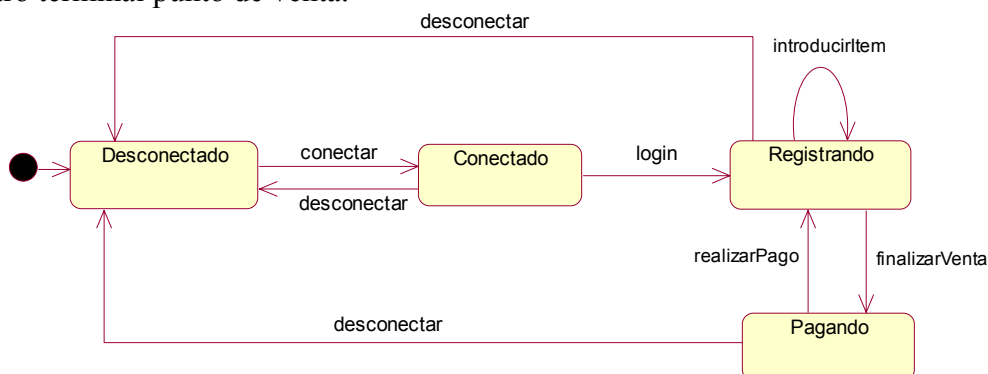
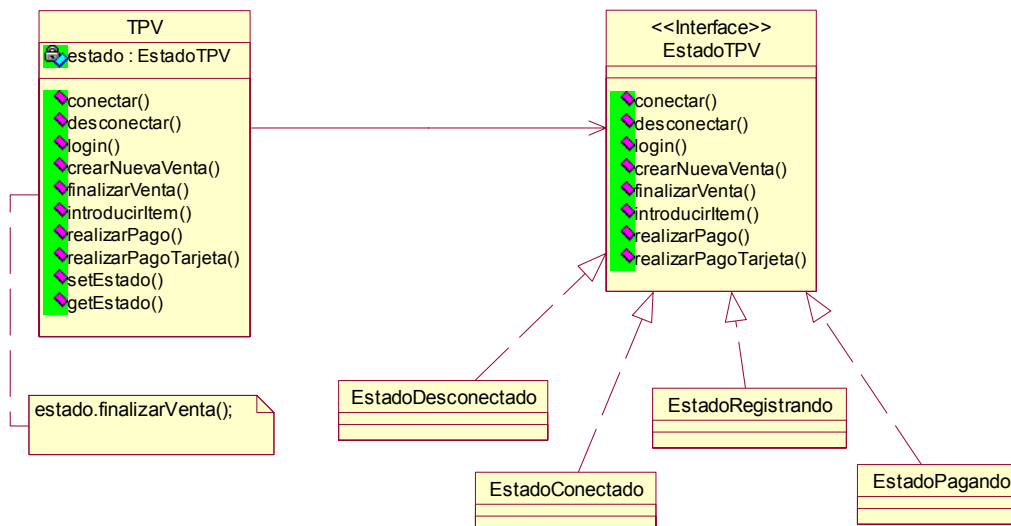


Figura 33: Diagrama de estados del sistema TPV

Uno de los patrones GoF nos permite modelar este tipo de situaciones: el patrón *State* (Estado) [21]. Su propósito es permitir que un objeto modifique su comportamiento cada vez que cambia su estado interno.

Implementación tradicional en Java

Se define una interfaz (`EstadoTPV`) que deben implementar todas aquellas clases que representen estados del terminal punto de venta. Estas clases implementarán el comportamiento específico de cada estado además de las transiciones a otros estados. El controlador TPV tendrá un atributo del tipo `EstadoTPV` al que delegará todas sus acciones. En el siguiente diagrama de clases podemos ver la estructura de patrón para el sistema TPV:

Figura 34: Estructura del patrón *Estado* para el sistema TPV

Implementación orientada aspectos con AspectJ

La implementación clásica del patrón *Estado* realiza una separación de intereses clara, localizando el comportamiento dependiente de cada estado en una clase. En la página 282 de [21] se plantea la siguiente cuestión: ¿dónde se deben definir las transiciones entre estados, en el cliente o en las subclases estado? Si las reglas de transición son fijas, estas pueden llevarse a cabo en el cliente. Sin embargo, resulta más flexible que sean las propias subclases estado las que definan sus transiciones a otros estados. Esto tiene el inconveniente de que las subclases estado deberán conocer los estados a los que transitan lo que introduce dependencias entre subclases.

Usando AspectJ, se consigue encapsular las transiciones entre estados en un aspecto, de forma que los estados sólo tendrán que preocuparse de implementar su comportamiento específico, sin tener en cuenta cuando tienen lugar las transiciones ni a que estados llevan. Ahora, un estado no conoce la existencia de los otros estados por lo que se elimina la dependencia entre subclases que existía en la implementación clásica del patrón. Además, el hecho de tener localizadas las transiciones en un mismo lugar mejora la comprensibilidad de las reglas de transición y aumenta la flexibilidad a la hora de modificar estas reglas o añadir nuevas ya que no se modifican las clases estado involucradas. En las siguientes líneas se muestra el código del aspecto `TransicionesEstadoTPV` que define las transiciones entre los estados del terminal punto de venta:

```

public aspect TransicionesEstadoTPV {

    //ESTADOS
    protected EstadoDesconectado eDesconectado = new EstadoDesconectado();
    protected EstadoConectado eConectado = new EstadoConectado();
    protected EstadoPagando ePagando = new EstadoPagando();
    protected EstadoRegistrar eRegistrar = new EstadoRegistrar();

    //REGLAS DE TRANSICIÓN

    //Estado inicial: Desconectado
    after(TPV tpv): initialization(new(..)) && target(tpv) {
        tpv.setEstado(eDesconectado);
    }
}
  
```

```

//Desconectado --> Conectado
after(TPV tpv, EstadoTPV estado): call(public void conectar(TPV)) &&
    target(estado) && this(tpv)
{
    if (estado==eDesconectado) {
        tpv.setEstado(eConectado);
    }
}

//Estado X --> Desconectado
after(TPV tpv, EstadoTPV estado): call(public void desconectar(TPV)) &&
    target(estado) && this(tpv)
{
    if (!(estado==eDesconectado)){
        tpv.setEstado(eDesconectado);
    }
}

//Conectado --> Registrar
after(TPV tpv, EstadoTPV estado) returning (Object o):
    call(public boolean login(TPV, String)) &&
    target(estado) && this(tpv)
{
    Boolean resultado = (Boolean) o;
    if ( estado==eConectado && resultado.booleanValue()){
        tpv.setEstado(eRegistrar);
    }
}

//Registrar --> Pagando
after(TPV tpv, EstadoTPV estado): call(public void finalizarVenta(TPV))
    && target(estado) && this(tpv)
{
    if (estado==eRegistrar){
        tpv.setEstado(ePagando);
    }
}

//Pagando (Efectivo) --->Registrar
after(TPV tpv, EstadoTPV estado):
    call(public void realizarPago(TPV,Dinero))
    && target(estado) && this(tpv)
{
    if (estado==ePagando){
        tpv.setEstado(eRegistrar);
    }
}

//Pagando (Con tarjeta) --->Registrar
after(TPV tpv, EstadoTPV estado):
    call(public void realizarPagoTarjeta(TPV,String,Date))
    && target(estado) && this(tpv)
{
    if (estado==ePagando){
        Respuesta respuesta =
            ((PagoTarjeta)tpv.getVenta().getPago()).getRespuesta();
        if (respuesta.isOK()){
            tpv.setEstado(eRegistrar);
        }
    }
}
}

```


4.3. Tracing con aspectos

Durante el desarrollo de una aplicación suele ser útil disponer de algún mecanismo de *tracing* que nos permita llevar un seguimiento del flujo de ejecución de nuestra aplicación. Este mecanismo es muy útil a la hora de depurar los programas, ya que nos permite detectar comportamientos no esperados de nuestra aplicación y localizar la causa del error para corregirlos.

A continuación vamos a ver la forma tradicional de implementar este mecanismo de monitorización del sistema, frente a la solución orientada aspectos proporcionada por AspectJ.

Implementación tradicional en Java

La opción más usada a la hora de implementar un mecanismo de *tracing* es añadir, al principio y al final de cada método de todas las clases, una instrucción de salida (por pantalla o a un fichero) que indique el inicio y el final de la ejecución del método respectivamente.

```
public void metodo () {
    System.out.println("Entrando Clase.metodo()");
    .....
    .....
    .....
    System.out.println("Saliendo Clase.metodo()");
}
```

El código de *tracing* se mezcla con el código que implementa la funcionalidad básica de la aplicación dando lugar a un código enredado y esparcido. Además, el mecanismo de tracing no es (des)conectable. Si una vez alcanzada una versión estable de la aplicación se desea eliminar el mecanismo de tracing tendremos que modificar el código de las clases de la aplicación.

Los mecanismos de monitorización de un sistema como puede ser el *tracing*, *logging*, *profiling*... son fácilmente encapsulables en un aspecto, como se verá en la siguiente sección.

Implementación orientada aspectos con AspectJ

AspectJ nos ofrece una forma sencilla y flexible de implementar un mecanismo de *tracing*. Mediante un aspecto conseguimos encapsular todo el código que implementa esta funcionalidad secundaria:

```
public aspect TracingAspect {

    pointcut pointsToBeTraced(): execution (* *.*(..));

    before():pointsToBeTraced()
    {
        System.out.println("Tracing: Enter to "+thisJoinPoint.getSignature());
    }

    after():pointsToBeTraced()
    {
        System.out.println("Tracing: Exit from "+thisJoinPoint.getSignature());
    }
}
```

Se observan los siguientes beneficios respecto a la solución tradicional:

- Se logra una separación de intereses clara ya que el mecanismo de *tracing* está totalmente encapsulado en un aspecto, evitando modificar las clases principales del sistema para implementarlo.
- Si se desea cambiar el mecanismo de *tracing* sólo tendremos que hacer cambios en el aspecto y no en un gran número de clases como ocurría en la anterior solución.
- El aspecto puede ser reutilizado para aplicar *tracing* a otras aplicaciones.
- El mecanismo de *tracing* es (des)conectable, una vez alcanzada una versión estable de la aplicación se elimina no incluyendo el aspecto en el proceso de entretendido.

4.4. Logging con aspectos

Se desea que el sistema terminal punto de venta tenga un fichero histórico (fichero de log) en el que se registren todas las ventas que tienen lugar durante el uso del sistema. Para ello, se hará uso de la clase `Logger`, cuyo método `writeLog()` permite escribir una línea en el fichero de log. En primer lugar se verá la implementación tradicional del mecanismo de *logging* en Java y luego la compararemos con la solución obtenida usando AspectJ.

Implementación tradicional en Java

Si no utilizamos aspectos, el código para implementar el mecanismo de log se entremezcla con las clases que implementan la lógica del negocio de la aplicación, como se puede ver en el siguiente fragmento de código que muestra algunos de los métodos de la clase `TPV`:

```
public void crearNuevaVenta () {
    estado.crearNuevaVenta(this);
    //Log
    Logger.writeLog("\nNueva Venta\t"+new Date());
}

public LineaVenta introducirItem (itemID id, int uni) {

    LineaVenta lv = estado.introducirItem(this,id,uni);

    //Log
    if (lv!=null)
        Logger.writeLog("Nuevo Item:"+lv.getProducto().getDescripcion()
            +"\tUnidades:"+lv.getUnidades() +"\tSubtotal:" +
            formatoDecimal.format(lv.getSubtotal().getCantidad())+" €");
    else Logger.writeLog("Linea de Venta CANCELADA");

    return lv;
}
```

Implementación orientada aspectos con AspectJ

Veamos a continuación cómo podemos encapsular la implementación del mecanismo de registro de ventas en un aspecto, evitando así el código enmarañado de la solución anterior.

```

public aspect LoggingAspect {

    private java.text.DecimalFormat formatoDecimal =
        new java.text.DecimalFormat("#,##0.00");

    pointcut conectar(): execution(void TPV.conectar());

    pointcut desconectar(): execution(void TPV.desconectar());

    pointcut login(): execution(boolean TPV.login(String));

    pointcut nuevaVenta(): execution(void TPV.crearNuevaVenta());

    pointcut finalizarVenta(): execution(void TPV.finalizarVenta());

    pointcut nuevoItem(): execution(LineaVenta TPV.introducirItem(..));

    pointcut pagoEfectivo(Dinero cantidad):
        execution(void TPV.realizarPago(Dinero)) && args(cantidad);

    pointcut pagoTarjeta(String numCC):
        execution(void TPV.realizarPagoTarjeta(String, Date)) &&
        args(numCC, Date);

    pointcut controlInventario():
        execution(void IAdaptadorInventario.actualizarInventario(Venta));

    pointcut contabilidad():
        execution(void IAdaptadorContabilidad.anotarVenta(Venta));

    pointcut autorizacionPago():
        execution(Respuesta
            IAdaptadorAutorizacionPagoTarjeta.autorizarPagoTarjeta(PagoTarjeta, String));

    after():conectar(){
        Logger.writeLog("\nSISTEMA TPV CONECTADO\n");
    }

    after():desconectar(){
        Logger.writeLog("\nSISTEMA TPV DESCONECTADO\n");
    }

    after(TPV tpv) returning(boolean resultado): login() && this(tpv){
        if (resultado)
            Logger.writeLog("Cajero: "+tpv.getCajero().getNombre());
    }

    after():nuevaVenta(){
        Logger.writeLog("\nNueva Venta\t"+new Date());
    }

    after(TPV tpv):finalizarVenta() && this(tpv){
        Logger.writeLog("TOTAL:"+formatoDecimal.format(tpv.getVenta().calcularTotal().getCantidad())+" €");
    }

    after() returning(LineaVenta lv):nuevoItem(){
        if (lv!=null)
            Logger.writeLog("Nueva Item:"
                + lv.getProducto().getDescripcion()
                + "\tUnidades:"+lv.getUnidades() + "\tSubtotal:" +
                formatoDecimal.format(lv.getSubtotal().getCantidad())+"€");
        else Logger.writeLog("Linea de Venta CANCELADA");
    }
}

```

```

    before(TPV tpv, Dinero cantidad): pagoEfectivo(cantidad) && this(tpv) {
        Logger.writeLog("Pago en efectivo. Cantidad entregada:
        "+formatoDecimal.format(cantidad.getCantidad())+" €"
        +"\tDevolución: "+formatoDecimal.format(cantidad.getCantidad()-
        tpv.getVenta().getTotal().getCantidad())+" €");
        Venta v = null;
        v.getTotal();
    }

    before(String numCC): pagoTarjeta(numCC) {
        Logger.writeLog("Pago con tarjeta. Num Tarjeta: "+numCC);
    }

    after(TPV tpv): pagoTarjeta(String) && this(tpv) {
        PagoTarjeta pt = (PagoTarjeta)tpv.getVenta().getPago();
        if (!pt.getRespuesta().isOk())
            Logger.writeLog("TARJETA INVALIDA");
    }

    after(): controlInventario() {
        Logger.writeLog("Acceso al servicio externo de control de
        inventario");
    }

    after(): controlInventario() {
        Logger.writeLog("Acceso al servicio externo de contabilidad");
    }

    after(): autorizacionPago() {
        Logger.writeLog("Acceso al servicio externo de autorización de
        pago con tarjeta");
    }
}

```

4.5. Comprobación de parámetros con aspectos

En muchas ocasiones debemos añadir líneas de código a un método para comprobar que los valores que se le pasan como parámetro, están dentro del rango de valores permitidos, es decir, añadimos código para comprobar que se cumple la precondition del método.

Por ejemplo, siguiendo con el ejemplo del TPV vamos a añadir código para asegurarnos que al constructor de la clase *Dinero* le pasamos un cantidad positiva. El nuevo constructor de la clase *Dinero* quedará de la siguiente forma:

```

public Dinero(double cantidad)
{
    if (cantidad<0) //Cantidad negativa
        cantidad=0;
    else
        this.cantidad = cantidad;
}

```

En el código anterior, la comprobación de parámetros se mezcla con la funcionalidad básica del método. Sería interesante poder capturar los valores que se pasan como parámetro a un método, para poder chequearlos y cambiarlos por valores apropiados si procediera, sin añadir código alguno al método implicado.

Esto es posible de realizar mediante un aspecto, en el que un punto de corte se encargue de capturar las llamadas al método (en este caso al constructor de la clase *Dinero*) y exponga los parámetros al consejo asociado para que este los valide, pudiendo sustituir los valores no permitidos por otros más apropiados.

```

public aspect ParameterCheck {

    /**
     * Capturamos las llamadas al constructor Dinero.new(double) y chequeamos el
     * valor del parámetro <i>cantidad</i>. Si es negativo, lo cambiamos por
     * el valor 0 y ejecutamos el constructor.
     * @param cantidad
     */
    Dinero around (double cantidad): call(Dinero.new(double)) && args(cantidad)
    {
        if (cantidad<0)
        {
            return proceed(0);
        }
        else
        {
            return proceed(cantidad);
        }
    }
}

```

Obsérvese que la restricción anterior, se tiene en cuenta sólo como ejemplo, ya que en nuestro terminal punto de venta podemos tener cantidades negativas, por ejemplo cuando tiene lugar la devolución de un producto.

Otro caso típico de comprobación de parámetros tiene que ver con el método `main` de una aplicación Java. El sistema TPV usa un fichero de configuración para obtener cierta información necesaria para su correcto funcionamiento: como la fuente externa donde se almacena el catalogo de productos (cadena de conexión), las estrategias de descuentos a aplicar, los tipos de servicios externos a solicitar, etc.

La ruta de este fichero de configuración puede pasarse como parámetro por la línea de comandos a la aplicación, o de lo contrario se usará el fichero de configuración por defecto (*config.properties*). Por lo tanto, tendremos que añadir código al método `main` de la clase principal de la aplicación, para validar el parámetro y en caso de error imprimir un mensaje informativo.

El siguiente fragmento de código muestra como el método `main`, no sólo realiza las tareas correspondientes a un TPV, sino que también se encarga de determinar el fichero de configuración a usar y de establecer la información de dicho fichero como propiedades del sistema.

```

public class Main {

    private static String ficheroConfiguracion = "config.properties";

    public static void main(String[] args)
    {

        String fichero;
        if (args.length == 1){
            //Se pasa como parametro el fichero de configuracion
            fichero = args[0];
        }
        else{
            //Fichero de configuracion por defecto
            fichero = ficheroConfiguracion;
        }

        File f = new File(fichero);
        FileInputStream fis;
        try{

```

```

        //Obtenemos las opciones de configuracion del fichero
        fis = new FileInputStream(f);
        Properties propiedades = new Properties();
        propiedades.load(fis);
        Enumeration enum = propiedades.propertyNames();
        //Insertamos las opciones como propiedades del sistema
        while (enum.hasMoreElements())
        {
            String propiedad = (String) enum.nextElement();
            String valor = (String)
                propiedades.getProperty(propiedad);
            System.setProperty(propiedad,valor);
        }
    } catch (Exception e)
    {
        System.out.println("Error: no se pudo cargar el fichero de
                           configuracion '"+fichero+"'");
        return;
    }
}

/*
 * A partir de aquí comienza la funcionalidad básica de este
 * metodo. Se ha omitido por simplicidad, para mas detalles ver
 * el código fuente.
 */
.....
.....
.....
}
}

```

Como vimos en líneas anteriores, utilizando un aspecto podemos encapsular la lógica de validación de los parámetros que se le pasan al método `main`, consiguiendo que dicho método se dedique única y exclusivamente a las funciones que tiene asignadas, dando lugar a una separación de intereses clara. Para el ejemplo que nos ocupa escribiremos las siguientes líneas en el aspecto `ParameterCheckAspect`:

```

/**
 * Fichero de configuracion por defecto
 */
private static String Main.ficheroConfiguracion = "config.properties";

void around(String[] argumentos) :
    execution(void Main.main(String[])) && args(argumentos)
{
    /**
     * Validamos los parámetros que se le pasan a la aplicación.
     * La sintaxis de la aplicación es: Main [fichero_configuracion].
     * En caso de éxito, se leen las opciones del fichero de
     * configuracion y se cargan como propiedades del sistema.
     * Ver el código en la pagina anterior.
     */

    /**
     * Ejecutamos el metodo main una vez cargada la información
     * del fichero de configuracion
     */
    proceed(argumentos);
}

```

El uso de aspectos para realizar comprobación de parámetros es una de las áreas de aplicación de la programación orientada aspectos que esta continua discusión. Para profundizar más sobre las posibilidades de AspectJ en este ámbito consultar [42].

4.6. Manejo de excepciones con aspectos

En el sistema TPV se desea tener un fichero de error en el que se registren las excepciones que tengan lugar durante el uso del sistema. Si no usáramos aspectos, tendríamos que escribir el código para registrar la excepción en cada uno de los bloques try/catch que aparecen dispersos por el código de la aplicación, como muestra el siguiente fragmento de código:

```
try{
    String tipo = System.getProperty("contabilidad.class.name");
    externo = (IAdaptadorContabilidad)
        Class.forName(tipo).newInstance();
    adaptadorContabilidad =
        new ProxyContabilidad(externo, new ContabilidadLocal());
} catch (Exception e) {
    //Registramos la excepcion en el fichero
    e.printStackTrace(LoggerError.getSalida());
    LoggerError.getSalida().flush();
}
```

Si se opta por usar el suavizado de excepciones de AspectJ, se consigue evitar que las excepciones tengan que ser capturadas en el código de la aplicación, por lo que ya no se tendrán que escribir los bloques try/catch que tanto enmarañan el código. Además, con un aviso de tipo *around* se consiguen capturar y registrar todas aquellas excepciones que tiene lugar durante la ejecución de la aplicación, como se muestra en el código del siguiente aspecto:

```
public aspect HandlerExceptionAspect {

    /*
     *Suavizamos las excepciones que tienen lugar en el código
     *de la aplicación
     */
    declare soft: Exception : execution(* *.*(..)) || execution(*.new(..));

    /*
     * Capturamos las excepciones suavizadas y registramos la pila
     * de llamadas en el fichero de error.
     */
    Object around():execution(* *.*(..))
    {
        try{
            return proceed();
        } catch (Exception e)
        {
            e.printStackTrace(LoggerError.getSalida());
            LoggerError.getSalida().flush();
            return null;
        }
    }
}
```

Para profundizar más sobre las posibilidades de AspectJ en el manejo de excepciones consultar [42].

4.7. Persistencia con aspectos

El sistema TPV, como es lógico, necesitará de un servicio de persistencia para almacenar y recuperar ciertos objetos como es el caso del catálogo de productos. Existen diferentes formas de implementar persistencia en Java como JDBC (*Java DataBase Connectivity*), EJB (*Enterprise JavaBeans*), JDO (*Java Data Objects*), etc. La estrategia de persistencia a utilizar no es lo importante en este ejemplo, por lo que se opta por usar la más sencilla de ellas, JDBC, el API estándar que ofrece Java para acceso a datos.

En la Figura 35 se muestra la estructura de las clases `Producto` y `CatalogoProductos` del sistema TPV. Cada vez que se arranque el sistema, se tendrá que cargar desde una base de datos el conjunto de productos que conforman el catálogo. Si durante el uso del sistema se realizan modificaciones sobre el catálogo, como por ejemplo añadir un nuevo producto, se tendrá que actualizar la base de datos donde reside el catálogo.



Figura 35: Clases *Producto* y *CatalogoProductos*

Implementación tradicional con Java

Para hacer persistente la clase `CatalogoProductos` se deben incluir sendos métodos: `store` y `restore` encargados de almacenar y restaurar el catálogo de productos respectivamente. El método `restore` es invocado dentro del constructor de la clase para cargar los productos en el catálogo, mientras que el método `store` es invocado cada vez que modificamos el catálogo con el método `putProducto`. El código de la clase `CatalogoProductos` es el siguiente:

```

public class CatalogoProductos {

    private Map productos = new HashMap ();

    public CatalogoProductos () {
        restore();    //Cargamos catalogo
    }

    /**
     * Devuelve un producto del catalogo
     * @param id - identificador del producto
     * @return producto
     */
    public Producto getProducto (itemID id) {
        return (Producto) productos.get(id);
    }

    /**
     * Introduce un nuevo producto en el catalogo
     * @param p - nuevo producto
     */
    public void putProducto(Producto p) {
  
```



```

        productos.put(p.getId(),p);
        store(); //Actualizamos catalogo
    }

    /**
     * Devuelve todos los productos del catalogo
     * @return colección de productos
     */
    public Collection getProductos() {
        return productos.values();
    }

    /**
     * Carga el catalogo de productos almacenado en una fuente externa
     */
    public void restore(){
        try{
            //El driver es una propiedad del sistema
            Class.forName(System.getProperty("driverJDBC.class.name"));
            //La cadena de conexión es una propiedad del sistema
            Connection con =
            DriverManager.getConnection(System.getProperty("cadenaConexion"));
            Statement st = con.createStatement();
            ResultSet rs= st.executeQuery("SELECT * FROM CatalogoProductos");
            while (rs.next()){
                String id = rs.getString("id");
                double precio = rs.getDouble("precio");
                String descripcion = rs.getString("descripcion");
                itemID ID = new itemID(id);
                Producto p =
                    new Producto(ID,new Dinero(precio),descripcion);
                putProducto(p);
            }
            st.close();
            con.close();
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    /**
     * Almacena el catalogo de productos en una fuente externa
     */
    public void store()
    {
        try{
            //El driver es una propiedad del sistema
            Class.forName(System.getProperty("driverJDBC.class.name"));
            //La cadena de conexión es una propiedad del sistema
            Connection con =
            DriverManager.getConnection(System.getProperty("cadenaConexion"));
            Statement st = con.createStatement();

            Iterator itr = getProductos().iterator();
            while (itr.hasNext()){
                Producto p = (Producto) itr.next();
                String id = p.getId().getId();
                String precio = p.getPrecio().toString();
                String descripcion = p.getDescripcion();
                ResultSet rs= st.executeQuery(
                    "SELECT * FROM CatalogoProductos WHERE id="+id);
                if (rs.next()){ //Existe, actualizamos producto
                    st.executeUpdate(
                        "UPDATE CatalogoProductos SET "
                        + "precio= '" + precio + "' , "
                        + "descripcion= '" + descripcion+ "'"
                        + "WHERE id = " + id+ "");
                }
            }
        }
    }

```

```

        else{//no existe, insertamos producto
            st.executeUpdate(
                "INSERT into " +
                "CatalogoProductos(id,precio,descripcion)" +
                " values ("
                + "'" + id + "',"
                + "'" + precio + "',"
                + "'" + descripcion + "'"
                + ")");
        }
    }
    st.close();
    con.close();
} catch (Exception e){
    e.printStackTrace();
}
}

```

Podemos afirmar entonces que el mecanismo de persistencia atraviesa la clase `CatalogoProductos`, ya que en el constructor de la clase se llama al método `restore` para cargar los productos y en el método `putProducto` se invoca al método `store` para actualizar la base de datos. El mecanismo de persistencia también podría haber atravesado la clase `Producto`, si se hubiera usado serialización para hacer el catálogo persistente, ya que entonces la clase `Producto` debería de implementar la interfaz `Serializable`. Por lo tanto, no existe una separación de intereses clara ya que la clase `CatalogoProductos` no sólo se ocupa de la gestión del catálogo sino que también se ocupa de cuestiones de persistencia. Además, el código JDBC aparece en la propia clase `CatalogoProductos` por lo que cambios en el modo de almacenamiento afectarán a la clase. Este último inconveniente se puede solucionar mediante el patrón DAO que encapsula el acceso a los datos en objetos DAO (*Data Access Objects*) haciendo el mecanismo de persistencia independiente de la plataforma.

Implementación con AspectJ

A continuación se intenta modelar el comportamiento cruzado del mecanismo de persistencia de la clase `CatalogoProductos` mediante un aspecto. Como se vio en líneas anteriores, el catálogo de productos se carga de una fuente externa cuando se crea una nueva instancia y se almacena en dicha fuente externa cuando se modifica el catálogo mediante la operación `putProducto`. El aspecto a implementar tendrá que capturar estos puntos de enlace y cargar o almacenar el catálogo según corresponda. Además, añadirá los métodos `restore` y `store` a la clase `CatalogoProductos` mediante declaraciones inter-tipo. El código del aspecto es el siguiente:

```

public aspect PersistenciaCatalogoProductos {

    /**
     * Identifica los puntos donde se carga el catalogo
     */
    private pointcut cargar(CatalogoProductos catalogo) :
        execution(CatalogoProductos.new(..)) && target(catalogo);

    /**
     * Carga el catalogo
     */
    after(CatalogoProductos catalogo) : cargar(catalogo)
    {
        catalogo.restore(); //Cargamos catalogo
    }
}

```

```

/**
 * Identifica los puntos donde se almacena el catalogo
 */
private pointcut actualizar(CatalogoProductos catalogo) :
    execution(void CatalogoProductos.putProducto(..)) &&
    target(catalogo) && !cflow(adviceexecution());

/**
 * Almacena el catalogo
 */
after(CatalogoProductos catalogo) : actualizar(catalogo)
{
    catalogo.store(); //Actualizamos catalogo
}

/**
 * Carga el catalogo de productos almacenado en una base de datos
 */
public void CatalogoProductos.restore(){
    /*
     * Cargamos el catalogo desde una BD con JDBC
     * Para más información ver código de la clase.
     */
}

/**
 * Almacena el catalogo de productos en una base de datos
 */
public void CatalogoProductos.store(){
    /*
     * Guardamos el catalogo en una BD mediante JDBC.
     * Para más información ver código de la clase.
     */
}
}

```

Con este aspecto se consigue una separación de intereses clara: la clase `CatalogoProductos` se encarga de gestionar el catálogo mientras que el aspecto `PersistenciaCatalogoProductos` se encarga de implementar el mecanismo de persistencia para esta clase. Se puede mejorar la solución anterior, si conseguimos que parte del código que implementa el mecanismo de persistencia pueda ser reutilizado para proporcionar persistencia a otras clases. Para ello creamos el siguiente aspecto abstracto que define un protocolo de persistencia reutilizable:

```

public abstract aspect ProtocoloPersistencia {

/**
 * Define el rol objeto Persistente
 */
public interface ObjetoPersistente {

    /**
     * Almacena el objeto en una fuente externa
     */
    public void cargar();

    /**
     * Recupera el objeto de una fuente externa
     */
    public void almacenar();
}

/**
 * Identifica los puntos donde es necesario recuperar el objeto
 * @param op - objeto a recuperar

```

```

    */
    protected abstract pointcut cargarObjeto(ObjetoPersistente op);

    after(ObjetoPersistente op) : cargarObjeto(op) {
        op.cargar();
    }

    /**
     * Identifica los puntos donde es necesario almacenar el objeto
     * @param op - objeto a guardar
     */
    protected abstract pointcut almacenarObjeto(ObjetoPersistente op);

    after(ObjetoPersistente op) : almacenarObjeto(op) {
        op.almacenar();
    }
}

```

Para hacer persistente una clase tendremos que crear un subaspecto de `ProtocoloPersistencia` que realice las siguientes tareas:

- Especificar la clase que juega el rol `ObjetoPersistente` mediante declaraciones de parentesco (`declare parents`).
- Dicha clase tendrá que implementar los métodos de la interfaz `ObjetoPersistente`: `cargar` y `almacenar`.
- Especificar los puntos donde es necesario cargar el objeto (punto de corte `cargarObjeto`) y los puntos donde es necesario almacenar el objeto (punto de corte `almacenarObjeto`)

La instancia concreta del protocolo de persistencia anterior para la clase `CatalogoProductos` podría ser la siguiente:

```

public aspect PersistenciaCatalogoProductos
    extends ProtocoloPersistencia{

    /**
     * Declaramos que el CatalogoProductos es un objeto persistente
     */
    declare parents: CatalogoProductos implements ObjetoPersistente;

    /**
     * Carga el catalogo de productos almacenado en una fuente externa (BD,
     * fichero...)
     */
    public void CatalogoProductos.cargar() {
        /*
         * Cargamos el catalogo desde una BD con JDBC
         * Para más información ver código de la clase.
         */
    }

    /**
     * Almacena el catalogo de productos en una fuente externa (BD,
     * fichero...)
     */
    public void CatalogoProductos.almacenar() {
        /*
         * Guardamos el catalogo en una BD mediante JDBC.
         * Para más información ver código de la clase.
         */
    }
}

```

```

/**
 * Identifica los puntos donde se carga el catalogo
 */
protected pointcut cargarObjeto(ObjetoPersistente op):
    execution(CatalogoProductos.new(..) && target(op);

/**
 * Identifica los puntos donde se almacena el catalogo
 */
protected pointcut almacenarObjeto(ObjetoPersistente op):
    execution(void CatalogoProductos.putProducto(..) &&
    target(op) && !cflow(adviceexecution( ));
}

```

Ahora, no sólo nos beneficiamos de la ventajas de una separación de intereses clara sino que además disponemos de un protocolo de persistencia reutilizable (aspecto `ProtocoloPersistencia`).

4.8. Interfaz gráfica del sistema TPV

La Figura 36 muestra la interfaz gráfica del sistema TPV. Esta interfaz ha sido implementada usando las clases swing de Java. Para comenzar a utilizar el terminal hay que conectarlo pulsando el botón *CONECTAR/DESCONECTAR*. Una vez conectado, el terminal queda a la espera de que el cajero introduzca su identificador para poder comenzar a registrar ventas. En una venta se pueden distinguir dos etapas: registrar artículos con o sin código y realizar el pago en efectivo o con tarjeta.

Para registrar un producto se teclea su código y se pulsa el botón *ACEPTAR*. Si el producto no tiene código se pulsa el botón *SIN CODIGO*, se teclea su precio y se pulsa el botón *ACEPTAR*.

Para especificar varias unidades de un producto la secuencia de teclas a marcar es la siguiente: *teclear unidades + botón "X" + teclear identificador* si el producto tiene un identificador o *teclear unidades + botón "X" + botón SIN CODIGO + precio* para artículos sin código.

En caso de devolución de algún artículo, el cajero especificará un número de unidades negativas, por ejemplo si se quieren devolver dos artículos con código 0001, la secuencia de teclas a marcar será: *teclear 2 + botón "x/-" + botón "X" + teclear 0001*. Para introducir cantidades decimales utilizar el botón *"."*.

Una vez registrados todos los productos de una venta se pulsa el botón *TOTAL* y a continuación se pulsa el botón *EFFECTIVO* o *TARJETA* según la modalidad de pago. Para efectuar un pago en efectivo se introduce la cantidad entregada por el cliente y se pulsa el botón *ACEPTAR*. Para efectuar un pago con tarjeta se teclea el número de tarjeta en el cuadro de texto que simula el lector de tarjetas, se pulsa el botón *OK* y para confirmar la transacción se pulsa el botón *ACEPTAR*. En cualquiera de los casos para pasar a atender a un nuevo cliente tendremos que cerrar el cajón de la caja.

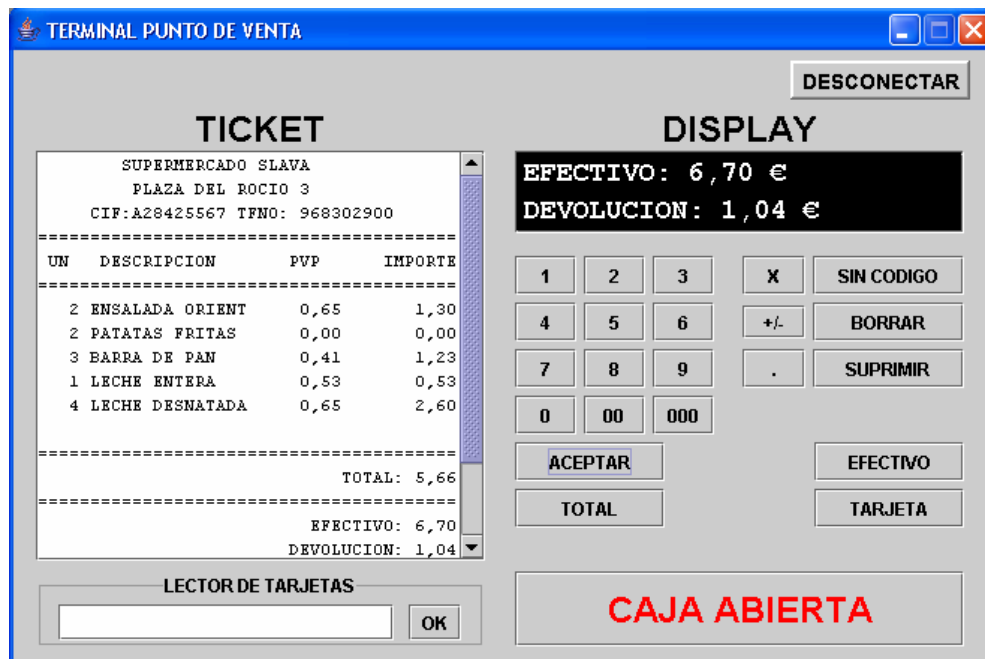


Figura 36: Interfaz gráfica del sistema TPV

4.9. Análisis de resultados

A lo largo de este proyecto se han implementado dos versiones del sistema terminal punto de venta. La primera implementación utilizando únicamente Java y la segunda usando además de Java, aspectos mediante AspectJ. La siguiente tabla muestra algunos datos de interés sobre cada una de las implementaciones:

	Nº clases	Nº de aspectos	Total
TPV (Java)	57	0	57
TPV(Java+AspectJ)	48	20	68

Tabla 20: Número de clases y aspectos en el sistema TPV

De la información recogida por la Tabla 20 se puede concluir que la versión en la que usamos aspectos es más modular, puesto que conseguimos encapsular el comportamiento cruzado que en la primera versión está disperso por las distintas clases de la aplicación, en aspectos.

	Nº de líneas de código (.java)	Nº de líneas de código (.aj)	Total
TPV (Java)	4605	0	4605
TPV(Java+AspectJ)	3847	1072	4919

Tabla 21: Número de líneas de código del sistema TPV

En los datos de la Tabla 21 también se puede ver como los intereses transversales que antes aparecían esparcidos por las distintas clases ahora se localizan en los aspectos.

Tras empaquetar cada una de las implementaciones en un fichero *.jar* para su distribución, obtenemos la siguiente información:

	Tamaño jar
TPV (Java)	76,4 KB (78.294 bytes)
TPV(Java+AspectJ)	196 KB (200.895 bytes)

Tabla 22: Tamaño del fichero *.jar*

Como se puede ver en la Tabla 22, el compilador *ajc* de AspectJ introduce una sobrecarga en el tamaño del fichero *.jar*, ya que durante el proceso de entretejido el compilador genera múltiples *.class* para conseguir tejer el comportamiento cruzado de los aspectos en las clases del sistema. En futuras versiones del compilador, se espera que esta sobrecarga vaya siendo cada vez menor.

5. Conclusiones y Trabajos Futuros

La POA es un paradigma muy reciente y por lo tanto poco consolidado, pero las indiscutibles posibilidades que ofrece a los desarrolladores de software están haciendo que su popularidad crezca a gran velocidad. La POA ofrece una solución elegante al problema de modelar intereses que afectan a diferentes partes del sistema y que con la POO se encontraban dispersos y enredados en múltiples clases. La POA encapsula estos intereses en entidades llamadas *aspectos* eliminando el código disperso y enredado y dando lugar a implementaciones más comprensibles, adaptables y reusables.

La POA no se tiene que ver como un sustituto de la POO, sino como una extensión de la misma, dando lugar a un paradigma de programación en el que coexisten clases y aspectos. Las clases modelan la funcionalidad básica del sistema mientras que los aspectos se encargan de modelar comportamiento cruzado que no puede ser encapsulado en una única clase, dando lugar a una separación de intereses completa. Las ventajas de aplicar POA son numerosas: obtenemos diseños más modulares, se mejora la trazabilidad, se consigue una mejor evolución del sistema, aumenta la reutilización, se reduce el tiempo de desarrollo y el coste de futuras implementaciones y se consigue retrasar decisiones de diseño haciendo más fácil razonar sobre los intereses principales de la aplicación. También existen inconvenientes, la mayoría relacionados con el carácter novedoso del paradigma, que hace que no existen estándares que faciliten su consolidación. Los próximos años pueden representar la madurez definitiva de la POA, las futuras investigaciones en el campo de la orientación a aspectos ayudarán a consolidar el paradigma en el marco de la ingeniería del software actual.

Entre los futuros trabajos que se podrían realizar a partir de este proyecto estarían:

- Estudiar las diferentes propuestas que están surgiendo relativas a cómo aplicar los aspectos en la fase de diseño del software [39, 40, 41].
- Análisis de otros lenguajes de programación orientados a aspectos como AspectC++, Aspecto o AspectWerkz, y compararlos con AspectJ.
- Estudio del framework orientado a aspectos *Spring* [38] que utiliza la POA.
- Investigar acerca de la formalización de los aspectos y la búsqueda de estándares que fomenten el crecimiento del paradigma orientado aspectos.
- Analizar nuevas versiones de AspectJ y su integración con las distintas herramientas de desarrollo.
- Analizar la relación entre la POA y el paradigma de desarrollo de software dirigido por modelos, MDA.

Anexo I: Sintaxis de AspectJ

En este anexo se muestra la sintaxis de AspectJ extraída de [24].

Puntos de corte

Anónimos

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

designator ::= designator_identifier(signature)
designator_identifier ::= call | execution | get | set |
                        handler | staticinitialization |
                        initialization | preinitialization |
                        adviceexecution | this | target |
                        cflow | cflowbelow | within |
                        withincode | args | if

signature ::= signatureMethod | signatureConstructor |
              signatureField | signatureType
```

Con nombre

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

access_type ::= public | private | package | protected
pointcut_name ::= identifier
parameters ::= {identifier type}
designator ::= designator_identifier(signature)
designator_identifier ::= call | execution | get | set |
                        handler | staticinitialization |
                        initialization | preinitialization |
                        adviceexecution | this | target |
                        cflow | cflowbelow | within |
                        withincode | args | if

signature ::= signatureMethod | signatureConstructor |
              signatureField | signatureType
identifier ::= letter { letter | digit }
type ::= tipos Java
```

Signaturas

```
signatureMethod ::=
    access_type ret_val class_type.method_name(parameters)
    [throws exception]

signatureConstructor ::=
    access_type class_type.new(parameters)[throws exception]

signatureField ::= access_type field_type class_type.field_name

signatureType ::= type
                 ;clase, interfaz, aspecto o tipo primitivo
```

```

access_type ::=      ;modificadore de acceso de Java.
ret_val ::=          ;tipo de retorno, cualquier tipo Java.
class_type ::=       ;clase Java
field_type ::=       ;tipo Java
method_name ::=      ;identificador Java
field_name ::=       ;identificador Java
parameters ::=       ;lista de parametros en Java
excpetion ::=        ; excepcion Java

```

Avisos

```

Advice ::= [ReturnType] TypeOfAdvice "["[Formals]"
           [AfterQualifier] [throws TypeList] ":"
           Pointcut "{"[AdviceBody]"

TypeOfAdvice ::= before | after | around
ReturnType ::= tipo java          ;sólo para avisos de tipo around
AfterQualifier ::= throwing "["[Formal]" |
                  returning "["[Formal]"
                  ; solo para avisos de tipo after
Formals ::=       ; como una lista de parámetros en Java
Pointcut ::=      ; ver sintaxis de puntos de corte
AdviceBody ::=    ; como el cuerpo de un método Java

```

Declaraciones inter-tipo

Campos

```

InterTypeField ::= Modifiers TypeOfField TargetType.Id
                  [Initialization];

Modifiers ::=      ;modificadores de acceso de Java
TypeOfField ::=   ;clase, interfaz o aspecto
Initialization ::= "=" expression

```

Métodos concretos

```

ConcreteInterTypeMethod ::=
    Modifiers ReturnType TargetType"." Id "(" Formals ")"
    [ThrowsClause] "{" MethodBody "}"

Modifiers ::= ;modificadores de acceso de Java
ReturnType ::= ;tipo retorno de un metodo Java
TargetType ::= ;clase, interfaz o aspecto.
Id ::= ;identificador del método
Formals ::= ; como una lista de parámetros en Java
ThrowsClause ::= ;igual que la clausula throws de Java
MethodBody ::= ;como el cuerpo de un método Java

```

Métodos abstractos

```

AbstractInterTypeMethod ::=
abstract Modifiers ReturnType
    AbstractTargetType"."Id
    "(" Formals ")" [ThrowsClause] ";

Modifiers ::= ;modificadores de acceso de Java

```

```

ReturnType ::= ;tipo retorno de un metodo Java
AbstractTargetType ::= ;clase o aspecto abstracto o una interfaz
Id ::= ;identificador del método
Formals ::= ; como una lista de parámetros en Java
ThrowsClause ::= ;igual que la clausula throws de Java

```

Constructores

```

InterTypeConstructor ::=
    Modifiers ConstructorTargetType "."new(" Formals ")
    [ThrowsClause] ";"

Modifiers ::= ;modificadores de acceso de Java
ConstructorTargetType ::= ;clase o aspecto concreto
Formals ::= ; como una lista de parámetros en Java
ThrowsClause ::= ;igual que la clausula throws de Java

```

Declaraciones de parentesco

Implementar interfaces

```

InterfaceDeclaration ::=
declare parents ":" TypePattern implements TypeList ";"

TypePattern ::= ;patrón de tipo
TypeList ::= Type {"," Type}
Type ::= ;interfaz

```

Extender clases

```

SuperclassDeclaration ::=
    declare parents ":" TypePattern extends TypeList ";"

TypePattern ::= ;patrón de tipo
TypeList ::= Type {"," Type}
Type ::= clase, aspecto o interfaz

```

Declaraciones en tiempo de compilación

```

MessageDeclaration ::=
    declare [ error | warning ] ":"
    StaticallyDeterminablePointcut ":" Message ";"

StaticallyDeterminablePointcut ::= ; punto de corte estático
Message ::= ;cadena a mostrar por el compilador

```

Declaraciones de excepciones suavizadas

```

SoftenerDeclaration ::= declare soft ":" Type ":"
    StaticallyDeterminablePointcut ";"

StaticallyDeterminablePointcut ::= ; punto de corte estático
Type ::= tipo de excepción

```

Declaraciones precedencia

```
PrecedenceDeclaration ::= declare precedence ":" TypePatternList ";"  
TypePatternList ::= TypePattern { "," TypePattern }  
TypePattern ::= ;patrón de tipo
```

Aspectos

```
aspect ::= access [privileged] [static] aspect identifier  
[class_identifier] [instantiation]  
{  
    //puntos de corte  
    //avisos  
    //declaraciones  
    //metodos y atributos  
}  
access ::= public | private [abstract] (protected por defecto)  
identifier ::= letter { letter | digit }  
class_identifier ::= [dominates] [extends]  
instantiation ::= issingleton | perthis | pertarget |  
                 percflow | perflowbelow
```

Anexo II: Instalación de AspectJ

Instalación básica

Para obtener una distribución de **AspectJ** debemos dirigirnos a [5]. En la sección de descargas podemos adquirir las distintas distribuciones existentes de AspectJ. En el momento de escribir este documento, la última versión estable era la 1.2.1 que es con la que a partir de ahora trabajaremos.

La distribución 1.2.1 de AspectJ viene empaquetada en un archivo .jar (`aspectj-1.2.1.jar`), que no debemos descomprimir, ya que es un fichero de instalación auto-extraíble en Java, por lo que para realizar la instalación necesitamos que nuestro sistema disponga de Java 1.2 o posteriores.

La distribución que se va instalar contiene:

- el compilador de AspectJ (**ajc**), que requiere para su ejecución Java 1.3 o posteriores.
- un generador de documentación javadoc (**ajdoc**).
- un navegador que nos permitirá navegar por la estructura cruzada de los programas, mostrando las partes del programa que se ven afectadas por determinados aspectos (**ajbrowser**).
- documentación y ejemplos.

El proceso de instalación es el mismo independientemente de la plataforma. Para comenzar la instalación tenemos dos opciones: hacer doble clic sobre el icono del archivo `aspect-1.2.1.jar` en una interfaz gráfica del sistema o bien ejecutar sobre la línea de comandos `java -jar aspectj-1.2.1.jar`.

La instalación comienza con la siguiente ventana de presentación:

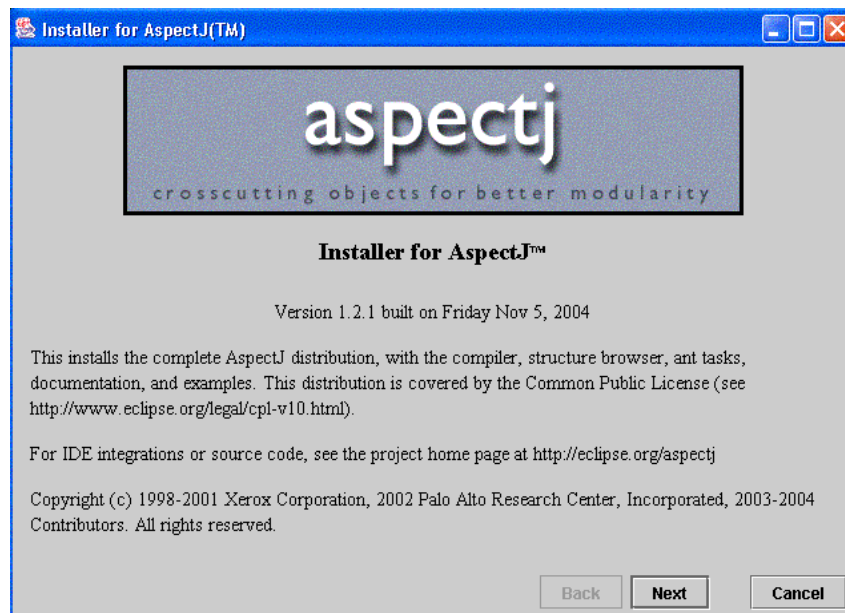


Figura 37: Ventana de presentación

Tras leer la información de la ventana, en la que se nos indican los contenidos de la distribución, pinchamos en *Next* para continuar con la instalación y nos aparecerá la siguiente pantalla:

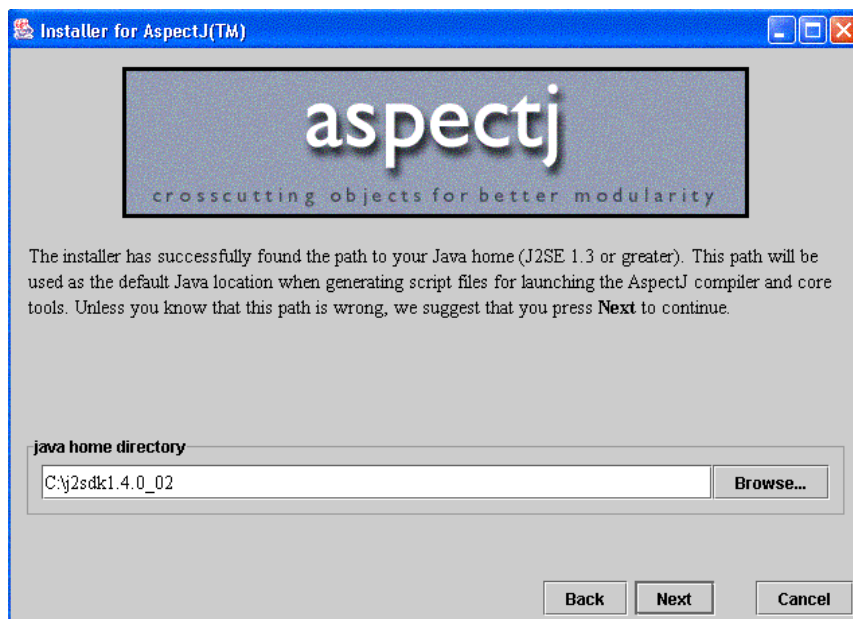


Figura 38: Directorio de la distribución de Java

En esta ventana se nos informa de la necesidad de tener instalado Java 1.3 o posterior para poder ejecutar con éxito las herramientas de AspectJ. Además, muestra la ruta de la distribución de Java instalada en nuestro sistema (variable de entorno `JAVA_HOME`). En el caso de que el instalador no encuentre la ruta correcta, podemos especificarla manualmente pinchando en el botón *Browse*.

Una vez introducida la ruta de la distribución de Java, pinchamos en *Next* y nos aparecerá una pantalla donde escribiremos el directorio de instalación de AspectJ.

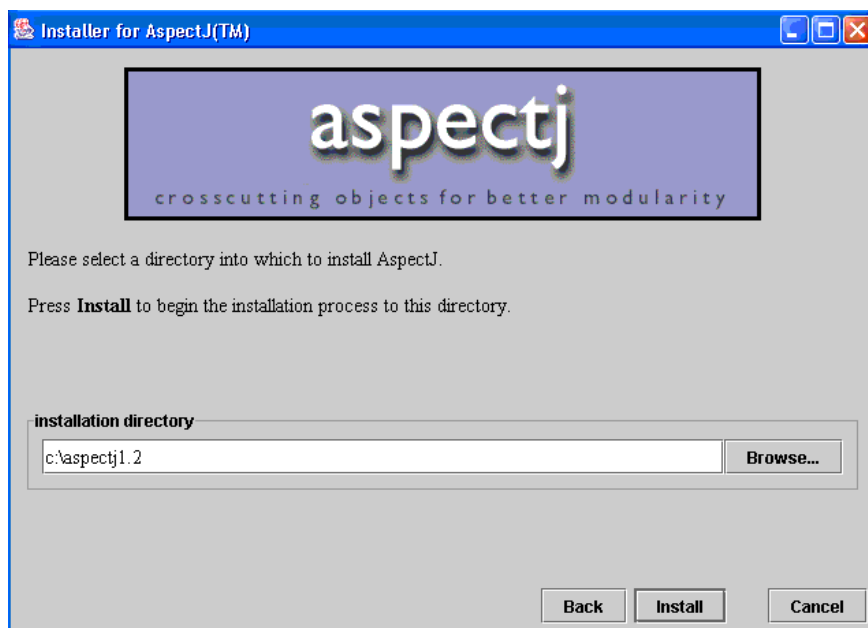


Figura 39: Directorio de instalación de AspectJ

A continuación, hacemos clic en el botón *Install*, y se lleva a cabo la copia de los ficheros al directorio indicado en nuestro sistema. Si queremos modificar alguno de los pasos anteriores, basta con pinchar en el botón *Back* que aparece en cada una de las pantallas de la instalación.

Una vez finalizada la copia de ficheros nos aparecerá la siguiente pantalla:

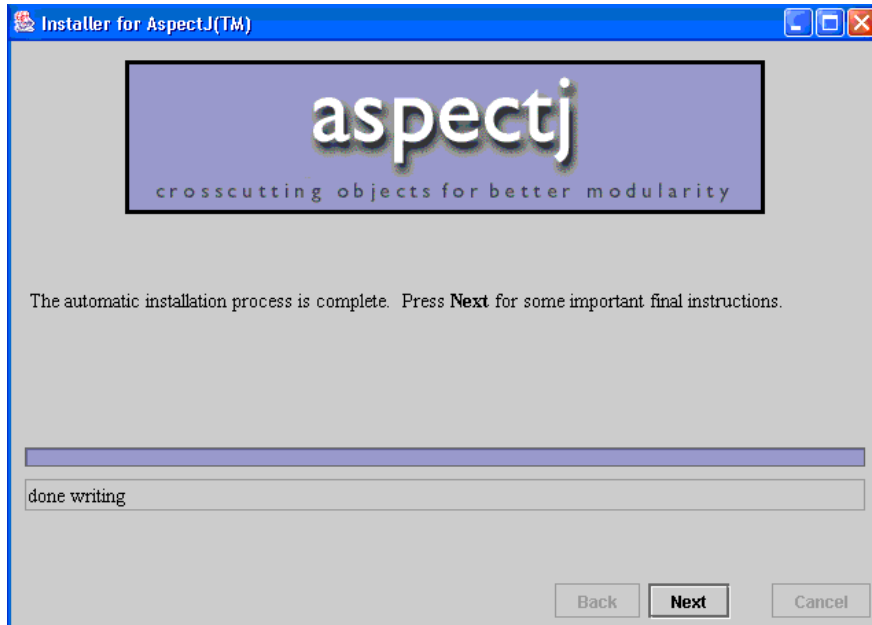


Figura 40: Proceso de copia de ficheros

La instalación termina mostrando la siguiente ventana:

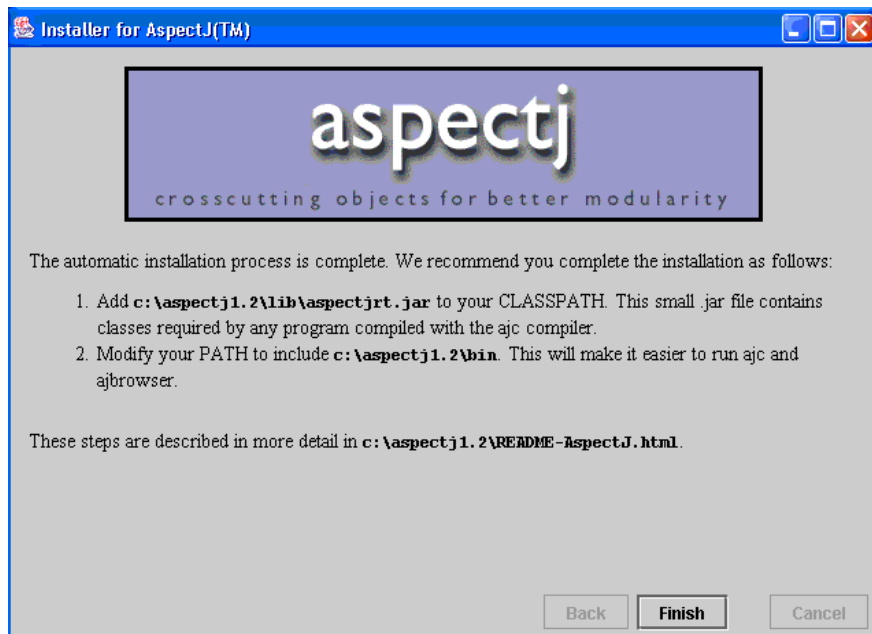


Figura 41: Ventana de fin de instalación

En ella se nos indican los últimos pasos que se deben realizar para un correcto funcionamiento del entorno AspectJ:

- Añadir la ruta `c:\aspectj1.2\lib\aspectjrt.jar` al CLASSPATH de tu sistema. También podemos optar por usar la opción `-classpath` cada vez que compilamos con `ajc`. Recomendamos la primera opción para no tener que indicar el classpath cada vez que compilamos.
- Añadir la ruta `c:\aspectj1.2\bin` al PATH del sistema. Para ello, en entornos Windows tendremos que modificar la variable de entorno PATH. En entornos UNIX/Linux tendremos que modificar algún fichero de configuración, que variará dependiendo de la distribución y que normalmente está ubicado en `/home` o `/etc`.

Para cerciorarnos del correcto funcionamiento de la instalación, realizaremos los siguientes pasos:

1. Escribir en la línea de comandos `javac` y nos tendrán que salir por pantalla todas las opciones del compilador de Java. Si no es el caso, debemos añadir al PATH del sistema la ruta del directorio `/bin` de la distribución de Java instalada.
2. Escribir en la línea de comandos `ajc`. Al igual que antes, nos deberían salir las opciones para el compilador de AspectJ. Si no es así, es que el comando no se ha encontrado, por lo que habrá que añadir al PATH del sistema la ruta del directorio `/bin` de la distribución de AspectJ.
3. Escribir la siguiente clase java, que imprime un mensaje por pantalla, compilarla (`javac HolaMundo.java`) y ejecutarla (`java HolaMundo`):

```
class HolaMundo {

    public void holaMundo()
    {
        System.out.println("Hola Mundo");
    }

    public static void main(String args[]) {
        HolaMundo h = new HolaMundo();
        h.holaMundo();
    }
}
```

Debería aparecer por pantalla el mensaje *“Hola mundo”*.

4. Escribir el siguiente aspecto y guardarlo como `HolaMundoAspect.aj`. Normalmente se usa la extensión **.aj** para aquellos archivos que contienen código AspectJ, pero también se podría usar la extensión **.java**.

```
public aspect HolaMundoAspect {
    pointcut callHolaMundo():
        call(public void HolaMundo.holaMundo*());

    before() : callHolaMundo() {
        System.out.println("Before Call");
    }

    after() : callHolaMundo() {
        System.out.println("After Call");
    }
}
```


Compilar `HolaMundo.java` y `HolaMundoAspect.java` escribiendo la siguiente línea:

```
ajc -classpath c:\aspectj1.2\lib\aspectjrt.jar  
HolaMundo.java HolaMundoAspect.aj
```

El compilador crea `HolaMundo.class` y `HolaMundoAspect.class`.

5. Ejecutar el programa (`java HolaMundo`). Se deberá imprimir por pantalla la siguiente secuencia de mensajes:

```
Before Call  
Hola Mundo  
After Call
```

6. Repetir el paso 4, pero esta vez sin la opción `classpath`. Si el compilador da algún error es porque no encuentra el fichero `aspectjrt.jar`, por lo que deberemos revisar el `CLASSPATH` y añadir la ruta de este fichero jar.

Como hemos podido observar, el proceso de entretejido se realiza durante la fase de compilación. El compilador **ajc** de la distribución **AspectJ 1.2**, entreteje los aspectos con las clases a nivel de byte-code, sin modificar los ficheros fuentes de estas clases, generando los ficheros **.class** correspondientes que podrán ser ejecutados sobre cualquier máquina virtual. En un futuro esta previsto añadir entretejido dinámico, donde el proceso tendrá lugar cuando las clases se carguen en la máquina virtual por lo que se necesitara un cargador de clases especial, que realice dicha función.

Integración en entornos de desarrollo

En la sección de descargas de [5] existen una serie de plug-in para integrar AspectJ con los entornos de desarrollo más populares, permitiéndonos editar, compilar y depurar proyectos que contienen aspectos de una forma sencilla y amigable. En el momento de escribir este documento existía soporte para los entornos Eclipse, JBuilder, Forte, NeatBeans y Emacs JDEE.

Para el desarrollo de este proyecto se está usando el entorno de desarrollo de Eclipse, por lo que veremos como se integra AspectJ en este entorno. El primer paso es descargar el plug-in de integración “**AspectJ Development Tools**” (AJDT) de [32]. Existen varias versiones, dependiendo de la versión de Eclipse en la que queramos integrar AspectJ. En nuestro caso, trabajamos con Eclipse 2.1.2 por lo que tendremos que descargarnos la versión 1.1.4 de AJDT. Para instalar el plug-in, basta con descomprimir el fichero descargado en la carpeta de instalación del entorno de desarrollo Eclipse.

En el directorio `plugins\org.eclipse.ajdt.ui_0.6.4\releaseNotes` de la instalación de Eclipse, podemos encontrar un fichero HTML (`readme.html`) que contiene información sobre como verificar la correcta instalación del plug-in y algunos detalles de configuración del entorno, para la correcta integración del plug-in. Veamos un resumen de la información más importante que contiene este documento:

Para verificar la correcta instalación del plug-in, abrimos Eclipse y pinchamos en el menú *Help*, opción *About* (Ayuda/Acerca de..). En el cuadro de dialogo resultante (ver Figura 42), pinchamos en el botón *Plug-in Details* (Detalles de conectores) y comprobamos que los siguientes plug-in están instalados:

- AspectJ Development Tools, versión 1.1.4.
- AspectJ Development Tools – UI, versión 0.6.4.

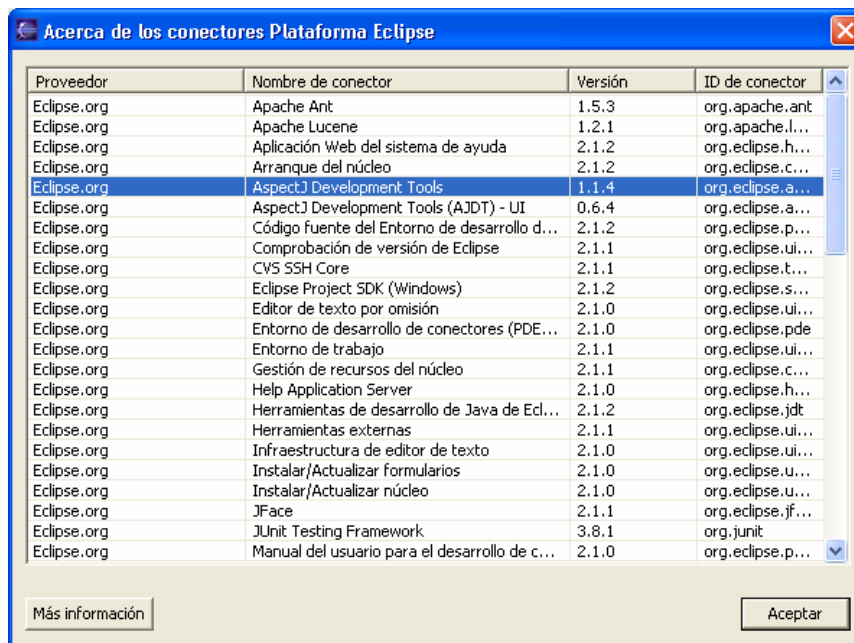
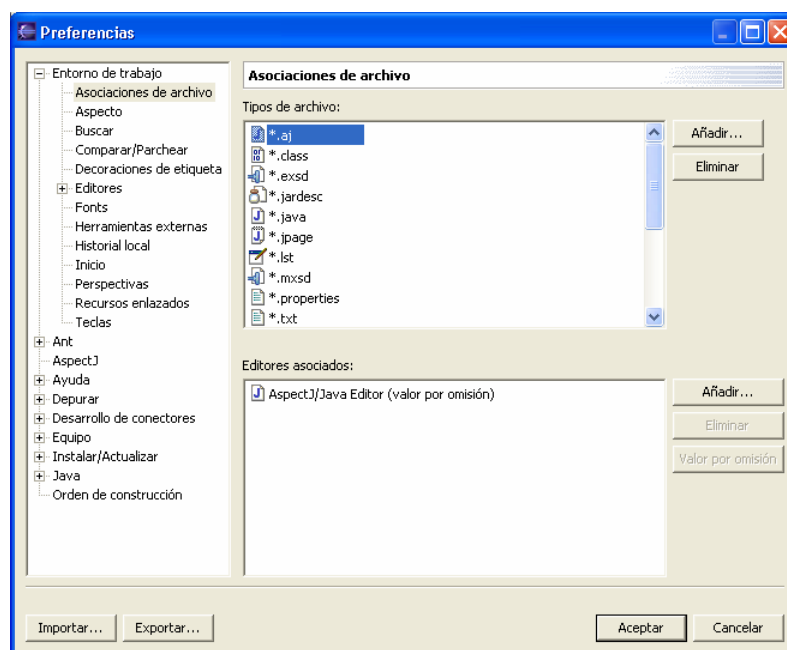


Figura 42: Plug-in instalados en Eclipse

En cuanto a la configuración del entorno, realizaremos las siguientes tareas:

- **Configurar el editor:** pinchar en *Window/Preferences* (Ventana/Preferencias). En el apartado *Workbench* (entorno de trabajo), elegimos *File associations* (asociaciones de archivo) y comprobamos que los archivos con extensión ***.aj** están asociados al editor *AspectJ/Java*:

Figura 43: Asociación de los archivos *.aj*

y los archivos con extensión ***.java** están asociados a *AspectJ/Java Editor* (por defecto) y al editor Java estándar:

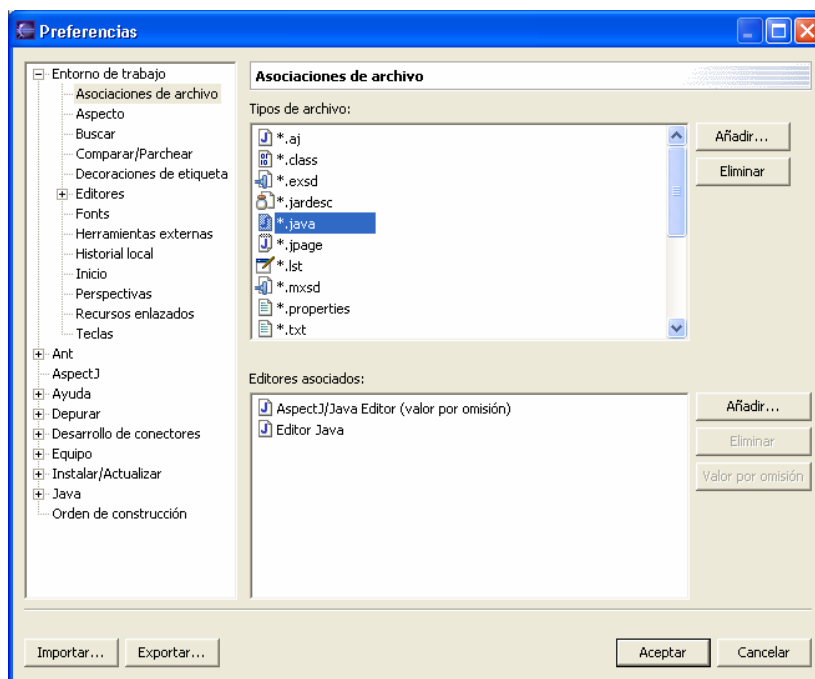


Figura 44: Asociación de los archivos .java

- Para facilitar la creación de nuevos aspectos y proyectos AspectJ, pinchamos en el menú *Window/Customize Perspective* (Ventana/Personalizar perspectiva), desplegamos el nodo *File>New* (Archivo>Nuevo) y activamos las casillas **Aspect** y **AspectJ Project**, como se muestra en la siguiente figura:

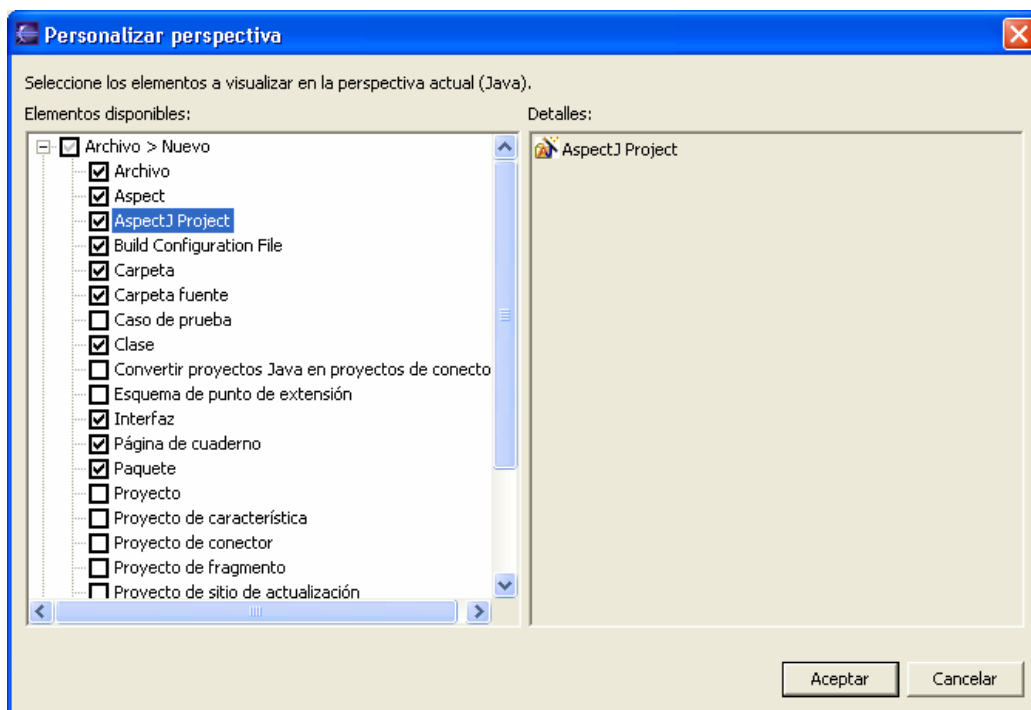


Figura 45: Personalización de la perspectiva actual

De esta forma, cuando pinchemos en el menú File/New (Archivo/Nuevo) nos saldrán dos nuevas opciones: crear un nuevo aspecto o crear un nuevo proyecto AspectJ, como se puede ver en la siguiente imagen:

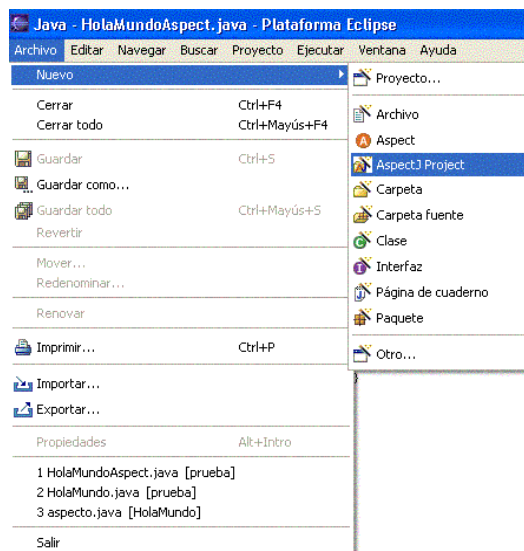


Figura 46: Menú *Archivo/Nuevo*

- En el documento también se nos aconseja que el compilador no genere avisos (warning) para las sentencias *import* no utilizadas, ya que esto puede llevar a confusiones a la hora de trabajar con aspectos. Para ello, hacemos clic en *Window/Preferences* (Ventana/Preferencias). Expandimos el nodo *Java* y seleccionamos la rama etiquetada con *Compiler* (Compilador). A continuación en la pestaña *Problems* (Problemas), establecemos la opción *Unused private types, methods or fields* (tipos, métodos o campos privados no utilizados) a *Ignore* (Ignorar), como se muestra en la siguiente figura:

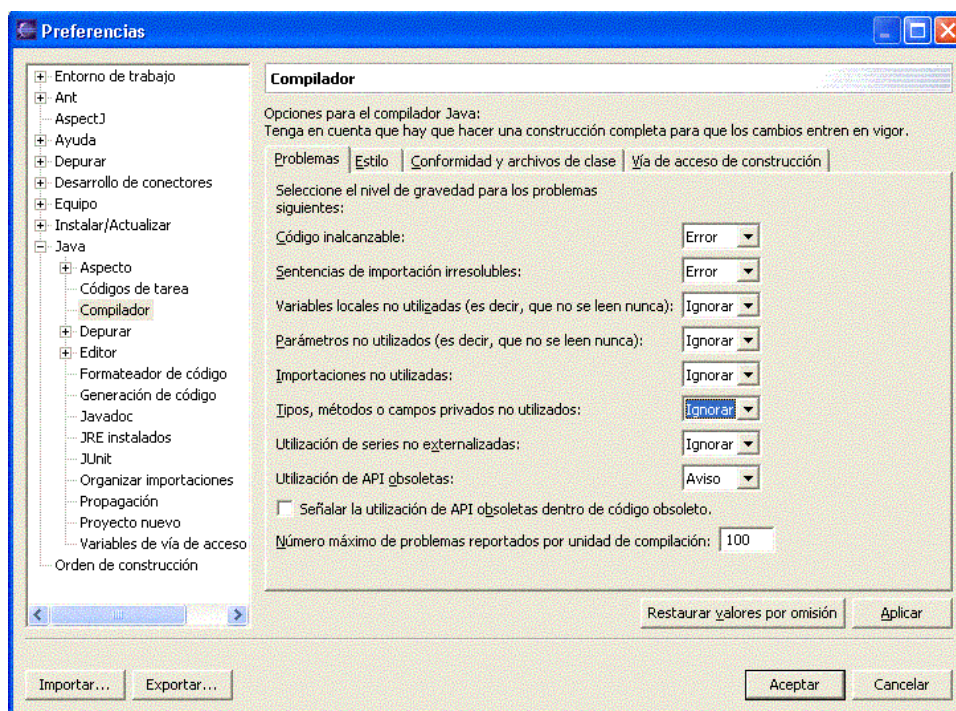
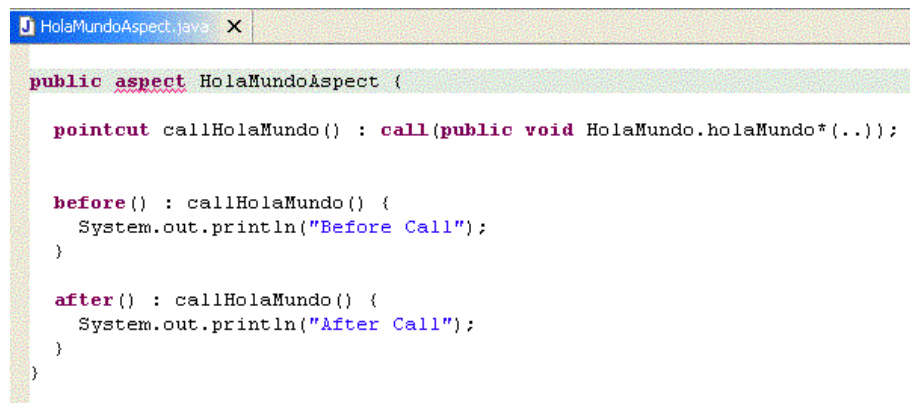


Figura 47: Opciones del compilador

- Cuando nosotros trabajamos con el entorno Eclipse, el editor subraya errores mientras tecleamos. Esta característica es muy útil en un editor, la pena es que no reconoce las palabras clave que forman parte de la extensión de AspectJ, por lo que estas palabras serán marcadas por el editor como errores, como se puede ver en la siguiente imagen:



```
public aspect HolaMundoAspect {  
  
    pointcut callHolaMundo() : call(public void HolaMundo.holaMundo*(..));  
  
    before() : callHolaMundo() {  
        System.out.println("Before Call");  
    }  
  
    after() : callHolaMundo() {  
        System.out.println("After Call");  
    }  
}
```

Figura 48: El editor no reconoce las palabras clave de AspectJ

Para que no ocurra esto tendremos que deshabilitar esta opción del editor. Para ello, hacemos clic en *Window/Preferences* (Ventana/Preferencias), expandimos el nodo *Java*, seleccionamos la opción *Editor*, pinchamos en la pestaña *Annotations* (Anotaciones) y desactivamos la casilla *Analyse annotations while typing* (analizar anotaciones al teclear), como se muestra en la siguiente pantalla:

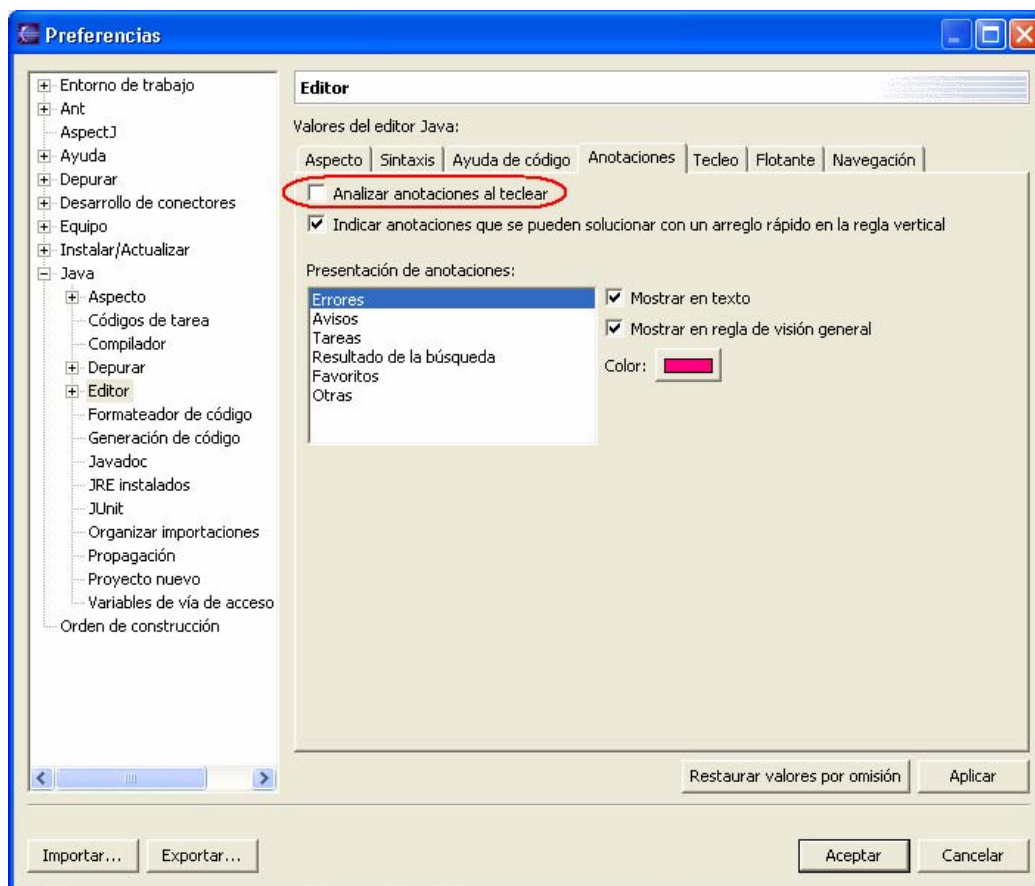


Figura 49: Opciones del editor Java

El inconveniente de deshabilitar esta característica del editor, es que ahora los errores no se nos mostraran mientras tecleamos (antes de compilar), sino que aparecerán tras la compilación.

- Si creamos un proyecto AspectJ y comenzamos a trabajar sobre él, veremos que cuando modificamos el código fuente y volvemos a ejecutar el proyecto, no se ven reflejados los cambios en la ejecución del programa, es decir, que no se lleva a cabo una compilación automática antes de ejecutar, por lo que tendremos que recompilar el proyecto antes de ejecutarlo. Esto ocurre porque en un principio AspectJ no soportaba la **compilación incremental**. A partir de la versión 1.1, se añade esta característica al compilador de AspectJ. Como nosotros estamos usando la versión 1.2, la compilación incremental si que está disponible, pero no por defecto, por lo que tendremos que activarla. Para ello, hacemos clic en *Window/Preferences* (Ventana/Preferencias), seleccionamos el nodo *AspectJ* y desactivamos la casilla *Supress automatic builds for AspectJ projects*.

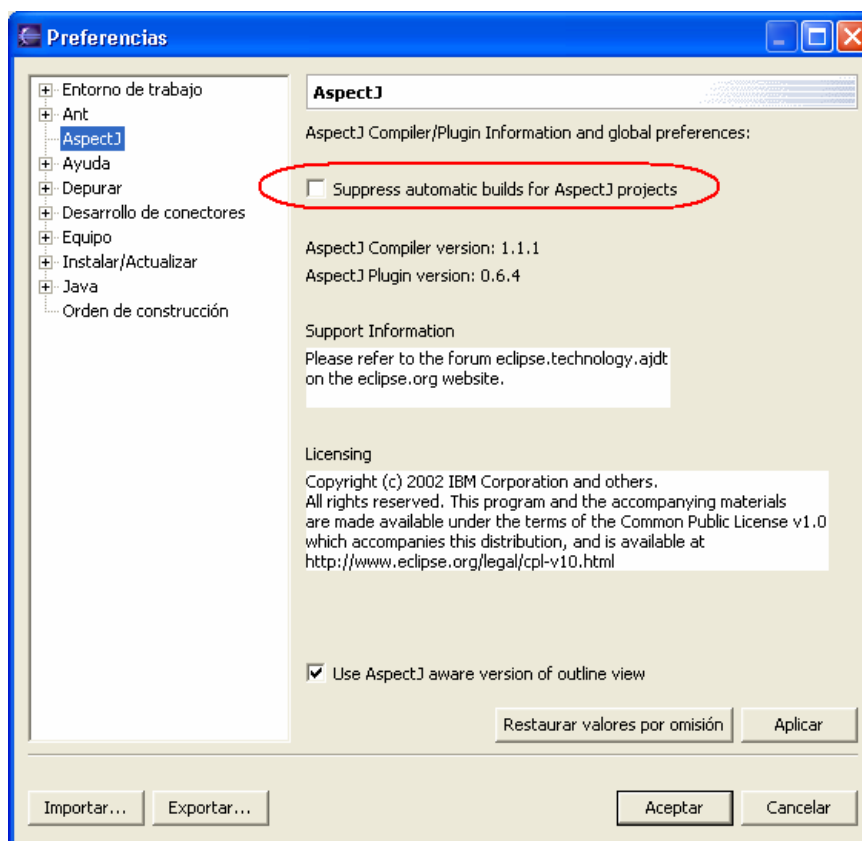


Figura 50: Opciones generales del compilador de AspectJ

También tendremos que indicar en las propiedades del proyecto en cuestión, que queremos usar este tipo de compilación, por lo que pinchamos en *Project/Properties* (Proyecto/Propiedades), seleccionamos el nodo *AspectJ Compiler* y activamos la casilla *Use incremental compilation*.

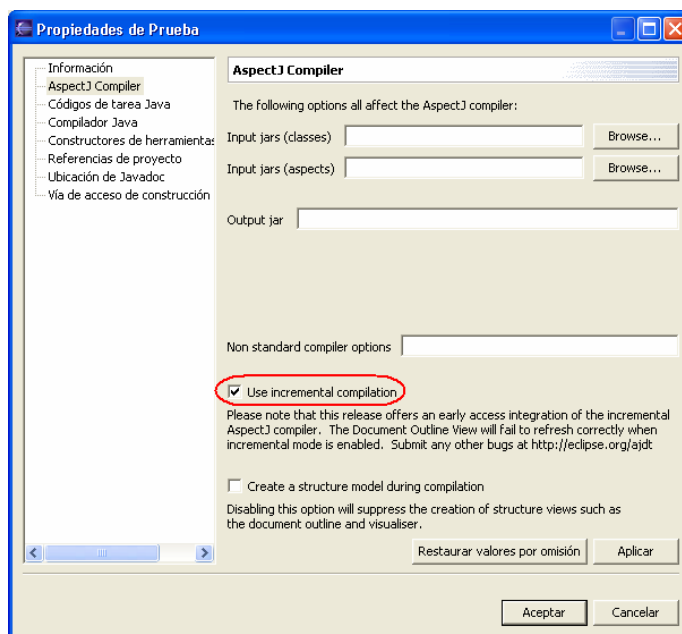


Figura 51: Opciones del compilador AspectJ para un proyecto

- Otro detalle a tener en cuenta es, que en un proyecto AspectJ, solo podemos establecer **puntos de interrupción** en las clases java, pero no en los aspectos. Para depurar los aspectos, tendremos que hacer uso del *debugger* (**ajdb**), pero éste sólo está disponible en las distribuciones de AspectJ, hasta la 1.0.

A continuación vamos a intentar compilar y ejecutar las clases ejemplo del apartado anterior, pero esta vez usando el entorno Eclipse, para probar así la correcta integración de **AspectJ**.

1. En primer lugar creamos un proyecto AspectJ, para ello pinchamos en *File/New* (Archivo/Nuevo) y elegimos *AspectJ Project*. Nos aparecerá un asistente, cuya primera pantalla nos permite especificar el nombre del proyecto y el directorio donde se almacenará.

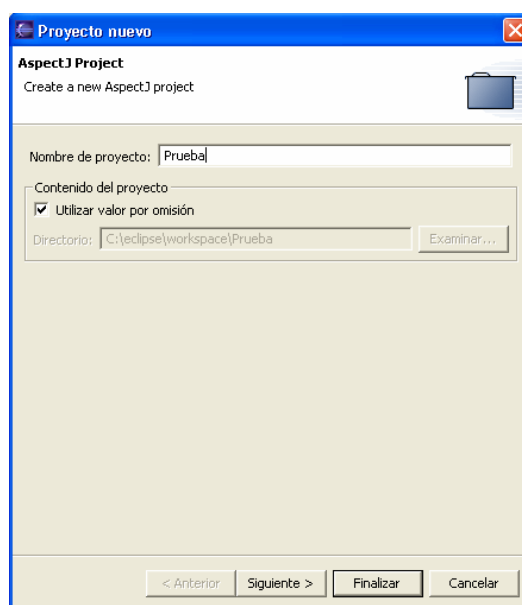


Figura 52: Ventana *Nuevo proyecto*

2. A continuación pinchamos en el botón *Next* (Siguiente), y nos aparece una ventana donde podemos establecer las propiedades del nuevo proyecto. En la pestaña *Libraries*, hacemos clic en el botón *Add external Jars* (añadir JAR externos) y buscamos el fichero **aspectjrt.jar** que se encuentra en el directorio `C:\eclipse\plugins\org.aspectj.ajde_1.1.4` y que es necesario para un correcto funcionamiento de las herramientas de AspectJ. Esta operación no hace falta, si anteriormente ya hemos añadido el fichero al CLASSPATH del sistema. Por último, pinchamos en el botón *Finish* (Finalizar).

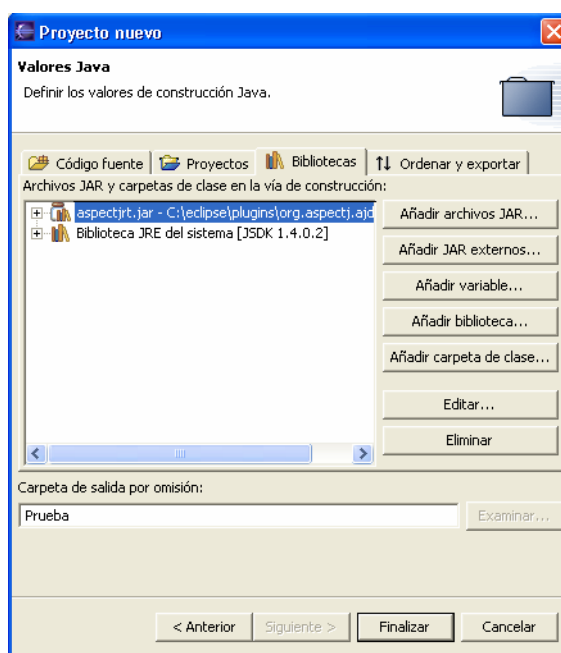


Figura 53: Añadir aspectjrt.jar al nuevo proyecto

3. Una vez creado el proyecto, le añadimos los ficheros fuentes `HolaMundo.java` y `HolaMundoAspect.aj`. Para ello, pinchamos en *File/Import* (Archivo/Importar), seleccionamos la opción *FileSystem* (Sistema de ficheros) y nos saldrá el siguiente cuadro de dialogo en el que podremos seleccionar los archivos y el proyecto donde queremos insertarlos:

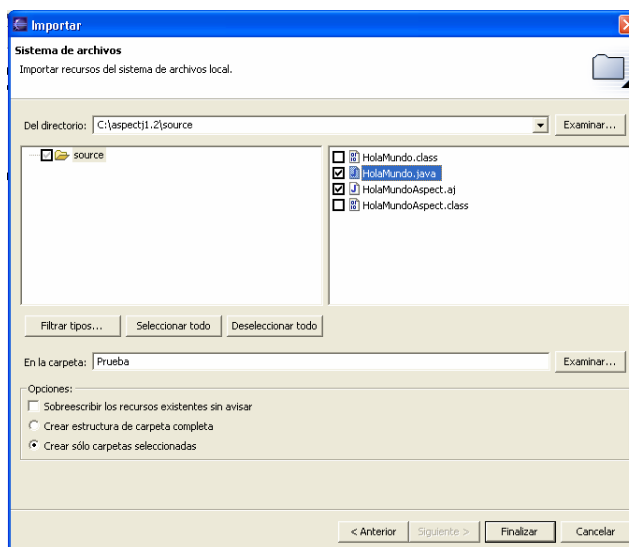


Figura 54: Ventana importar ficheros a un proyecto

4. Tras añadir los ficheros fuente, vamos a compilar el proyecto. Para ello pinchamos en *Project/Rebuild Project* (Proyecto/Reconstruir proyecto).
5. Por último nos queda ejecutar el proyecto, por lo que hacemos clic en el menú *Run/Run...* (Ejecutar/Ejecutar...) y nos aparecerá un asistente para crear una nueva configuración de ejecución, seleccionamos la opción *Java Application* (Aplicación Java) y especificaremos el proyecto a ejecutar y la clase Main.

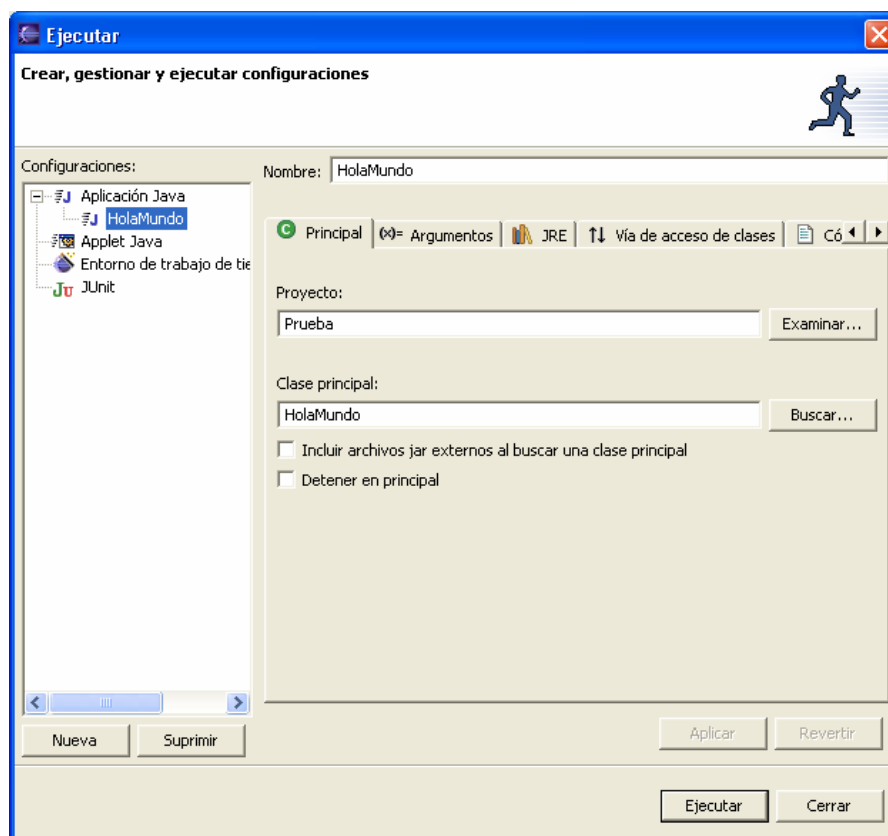


Figura 55: Ventana de configuración de la ejecución

Una vez insertados los datos, pinchamos en el botón Run (Ejecutar) y se produce la ejecución de la clase `HolaMundo.java`. Como resultado de la ejecución se imprime por pantalla la secuencia de mensajes esperada:

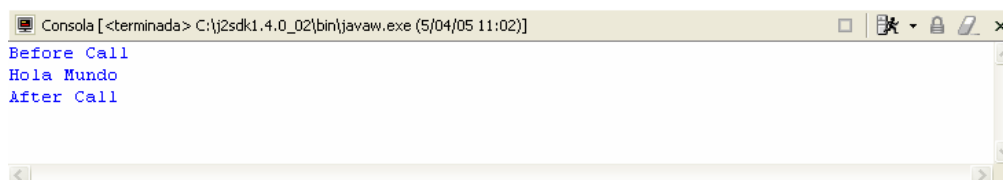



Figura 56: Mensajes impresos por la consola

Para posteriores ejecuciones, bastará con pinchar el botón de ejecución de la barra de herramientas , y se ejecutará el proyecto usando la configuración anteriormente creada.

Anexo III: Herramientas de AspectJ

Compilador: ajc

El compilador de AspectJ (comando `ajc`) es el encargado de tejer los aspectos en el código base de nuestra aplicación, por lo que también se le suele llamar tejedor. A este proceso se le denomina entretejido y puede realizarse en distintas fases de la vida de un programa. El compilador de la **versión 1.0.x** de AspectJ, genera un nuevo código fuente como resultado de integrar los aspectos en el código fuente de la aplicación. El nuevo código fuente se pasa al compilador de java (`javac`) para generar el ejecutable final. Por lo tanto el proceso de entretejido tiene lugar en el código fuente.

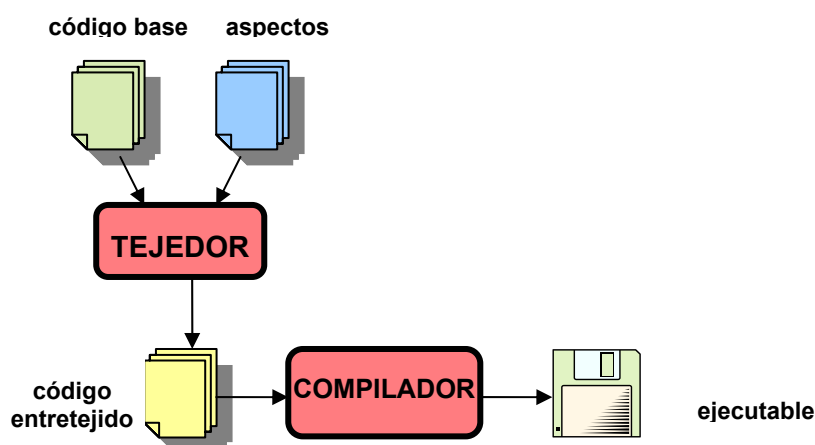


Figura 57: Compilador AspectJ 1.0.x

En cambio, en el compilador de la **versión 1.1** y posteriores de AspectJ, el entretejido tiene lugar a nivel de byte-code, es decir, el compilador de AspectJ entreteje los aspectos en los ficheros `.class` de nuestra aplicación, generando los `.class` de salida correspondientes. Esto nos permitirá añadir aspectos a aplicaciones de las que sólo tenemos el código compilado y no las fuentes. Aún así, el compilador de AspectJ continúa aceptando como entrada ficheros fuentes (`.java`) e internamente los compila antes de realizar el entretejido.

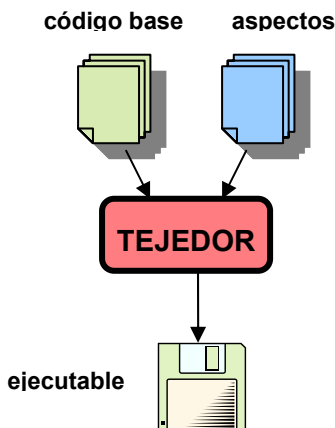


Figura 58: Compilador AspectJ 1.1.x

El compilador `ajc` está implementado en Java, por lo tanto podrá ser invocado desde una aplicación Java, a través de la clase `org.aspectj.tools.ajc.Main`, que se encuentra empaquetada en el fichero `aspectjtools.jar` de la distribución de AspectJ. Pero la forma usual de invocar al compilador de AspectJ es a través de la línea de comandos. La sintaxis de una llamada al compilador desde la línea de comandos es la siguiente:

```
ajc [opciones] [ ficheros_fuente | @fichero_lst ]
```

A diferencia del compilador de Java, `ajc` no busca en el *sourcepath* las clases a compilar de forma automática, por lo que habrá que indicarle al compilador los directorios donde se encuentran los fuentes (opción `-sourcepath`) o bien escribir todos los ficheros en la línea de comandos.

En algunos proyectos puede que el número de ficheros a compilar sea bastante elevado y que además nos interese hacer uso de varias opciones del compilador. En este tipo de situaciones, escribir en la línea de comandos todos los ficheros que intervienen en la compilación y las opciones del compilador a usar, puede ser una tarea tediosa. Existe la posibilidad de especificar en un fichero, tanto la lista de archivos que se le pasarán al compilador, como las opciones. En dicho fichero, cada línea debe contener una opción o un nombre de fichero. Estos ficheros se suelen nombrar con la extensión `.lst`. Para indicarle al compilador la existencia de un fichero que contiene los argumentos que debe tomar (opciones y ficheros fuentes), se usa el símbolo “@”, de forma que la llamada al compilador quedaría:

```
ajc @argumentos.lst
```

o bien la opción **argfile**:

```
ajc -argfile argumentos.lst
```

Si escribimos en la línea de comandos `ajc`, sin argumentos, nos aparecerá por pantalla la sintaxis del comando, así como todas las posibles opciones que admite. En la siguiente tabla se listan algunas de las opciones más útiles del compilador de AspectJ 1.2. [29]:

Opciones del comando ajc	
-argfile fich_lst	Especifica la ruta del fichero de argumentos.
-injars jars ¹	Entreteje los aspectos con los <code>.class</code> empaquetados en los ficheros <code>.jar</code> especificados. En desuso, usar la opción -inpath , que además de <code>.jar</code> permite especificar directorios.
-inpath path ³	Entreteje los aspectos con los <code>.class</code> contenidos en los directorios o <code>.jar</code> indicados
-aspectpath jars	Entreteje los aspectos empaquetados en los ficheros <code>jar</code> con el código base. Útil, cuando trabajamos con librerías de aspectos.
-outjar fich.jar	Empaqueta las clases generadas por el compilador en un fichero <code>.jar</code> . El nuevo fichero <code>jar</code> no podrá ser usado en un nuevo proceso de entretejido, a no ser que se haya compilado con la opción -Xreweable .

¹ *jars*: lista de rutas a ficheros `zip/jar`, delimitadas por un símbolo que depende de la plataforma: “:” en sistemas UNIX y “;” en sistemas Windows

² *path*: lista de rutas a directorios o ficheros `zip/jar`, delimitadas por un símbolo que depende de la plataforma: “:” en sistemas UNIX y “;” en sistemas Windows



-sourceroots dirs ⁴	Compila todos los ficheros con extensión .java o .aj situados en los directorios especificados.
-incremental	Realiza compilación incremental. Disponible a partir de AspectJ 1.1. Se debe pasar también la opción -sourceroots para indicar el directorio sobre el que realizar la compilación incremental.
-emacs sym	Genera un fichero de símbolos (.ajesym) para dar soporte a Emacs
-encoding enc	Especifica la codificación de los ficheros fuentes a compilar
-noExit	No llama a <code>System.exit</code> cuando termina la compilación.
-repeat N	Repite el proceso de compilación N veces.
-classpath path -cp path	Especifica donde encontrar las clases de nuestra aplicación.
-bootclasspath path	Modifica el directorio del que se toman las clases básicas de Java en la compilación
-extdirs path	Modifica la localización de los directorios de extensión
-d directorio	Especifica el directorio donde se almacenaran los .class generados. El directorio por defecto es el actual.
-target [1.1 1.4]	Establece la versión de la MV a la 1.1 o la 1.4
-source version	Establece la versión del compilador a la 1.3 o la 1.4
-help -?	Imprime información sobre las opciones del compilador y su uso.
-version o -v	Imprime por pantalla la versión del compilador
-showversion	Muestra la versión del compilador y continua
-verbose	Muestra información sobre las actividades del compilador.
-showWeaveInfo	Muestra información sobre el proceso de entretejido
-log fich.log	Guarda los mensajes del compilador en un fichero de log.
-time	Muestra información de tiempo.
-progress	Muestra información de progreso.
-nowarn	No emite mensajes de aviso (warning).
-warn: items	Emite mensajes de aviso (warning) para determinadas situaciones.
-X	Muestra por pantalla las opciones no estándar.
-XnoWeave	El compilador no realiza el proceso de entretejido, solo realiza el proceso de compilación tradicional.
-Xreweavable	Genera .class que pueden ser entretejidos sucesivas veces.
-Xlint:level	Establece el nivel de abstracción para los mensajes de error. <i>level</i> puede ser <i>ignore</i> , <i>warning</i> o <i>error</i>

Tabla 23: Opciones del comando *ajc*



Navegador: ajbrowser


La herramienta **ajbrowser** nos proporciona una interfaz gráfica que nos permite compilar un proyecto AspectJ, ejecutarlo, editar sus ficheros fuentes y navegar por su estructura. Para abrir esta herramienta basta con escribir en la línea de comandos:

```
ajbrowser [fichero1.lst fichero2.lst fichero3.lst ...]
```

Al comando **ajbrowser** le podemos pasar como parámetro un conjunto de ficheros con extensión *lst*. Cada uno de estos ficheros contiene los archivos que componen un proyecto y las opciones que se le pasarán al compilador **ajc** a la hora de compilar el proyecto. Una vez arrancado el navegador, pinchando en el botón **Build**  se desplegará una lista con las configuraciones disponibles, pudiendo cargar la que deseemos. Para añadir una nueva configuración pinchamos en el botón  (*select*

³ *dirs*: lista de rutas a directorios delimitadas por un símbolo que depende de la plataforma: “:” en sistemas UNIX y “;” en sistemas Windows

build configuration) y seleccionamos el nuevo fichero `.lst`. Esta nueva configuración pasará a estar disponible en la lista desplegable del botón . Para eliminar una configuración de la lista, usar el botón  (*close build configuration*).

Para compilar un proyecto, hacer clic en el botón  de la barra de herramientas o bien pinchar en el menú *Project* y seleccionar la opción *Build*. Si deseamos que la compilación sea incremental, debemos ir al menú *Tools*, pinchar en la opción *Options*, y en el cuadro de dialogo que aparece, seleccionar la pestaña *AspectJ Build Options* y activar la casilla *Incremental compile*, como se muestra en la Figura 59.

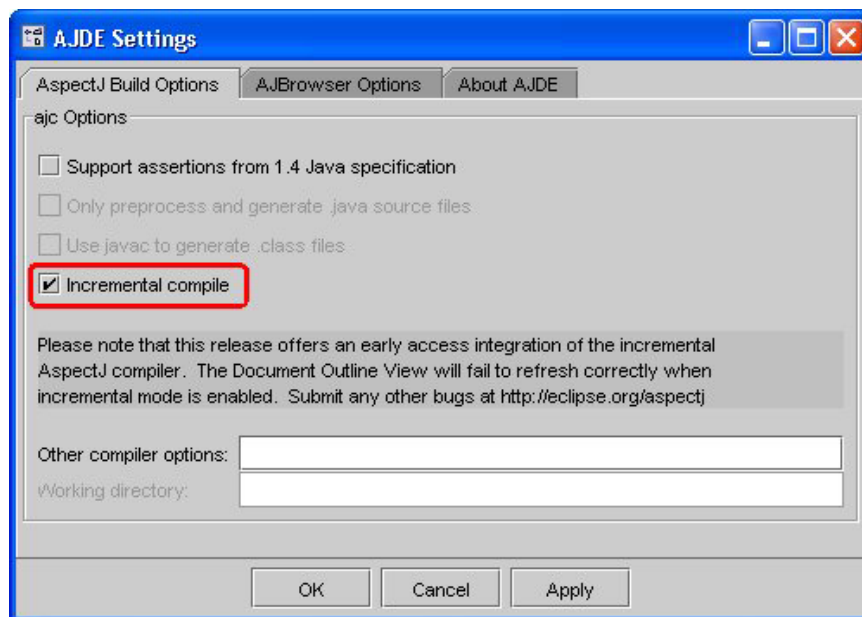





Figura 59: Compilación incremental en *ajbrowse*

Una vez activada la compilación incremental, si deseamos forzar una compilación completa del proyecto basta con mantener presionada la tecla *Shift* mientras pinchamos el botón de compilación  o la opción *Project/Build*. Si a la hora de compilar el navegador muestra un mensaje de error indicando que no se encontró el fichero *aspectjrt.jar*, debemos ir al Menú *Tools*, opción *Options*, seleccionar la pestaña *AJBrowse Options* del cuadro de diálogo emergente y escribir la ruta del archivo *aspectjrt.jar* en el cuadro de texto *Classpath*.

Para ejecutar un proyecto, hacer clic en el botón  de la barra de herramientas o bien a través del menú *Project*, donde podemos seleccionar entre ejecutar el proyecto en la misma máquina virtual en la que se ejecuta el navegador o bien en un nuevo proceso. Para que la ejecución pueda tener lugar con éxito, hay que especificar la clase principal (*class Main*) del proyecto, pinchando en el botón  *Options* y escribiendo el nombre de la clase en el cuadro de texto *Run Options* que aparece en la pestaña *AJBrowse Options*, como muestra la Figura 60.

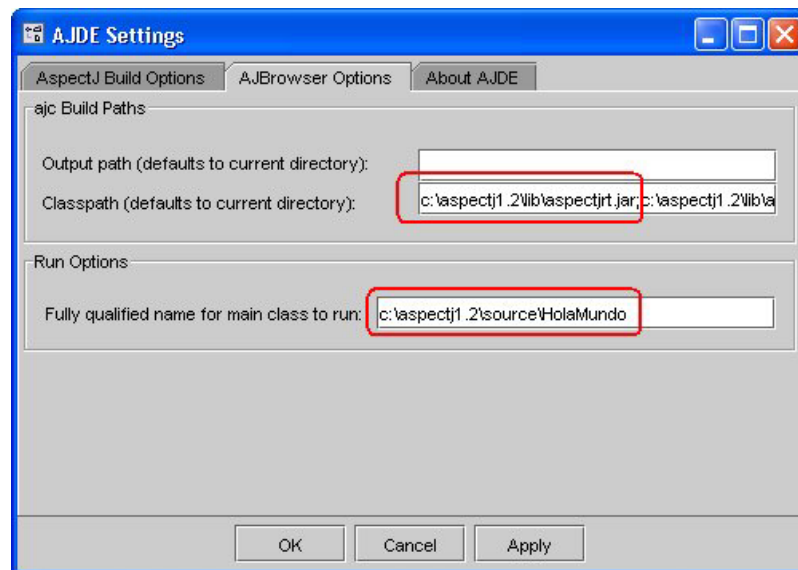


Figura 60: Classpath y Main class en *ajbrowse*

Una vez cargado un proyecto y compilado con éxito, el aspecto del navegador será el mostrado por la Figura 61.

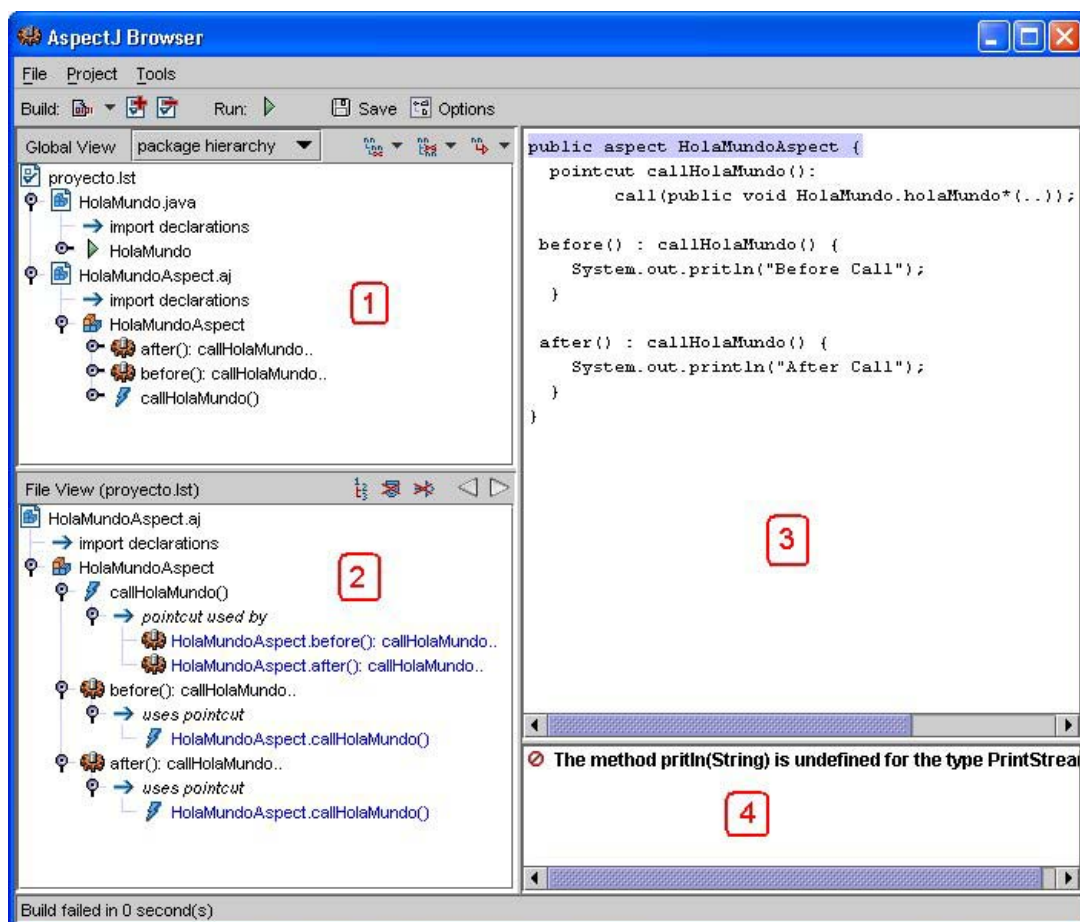







Figura 61: Interfaz de *ajbrowse*

Como podemos ver en la imagen anterior el navegador se divide en cuatro partes bien diferenciadas:

1. **Vista global:** en esta vista podemos ver el proyecto desde diferentes perspectivas: jerarquía de paquetes (vista por defecto), jerarquía de tipos y estructura cruzada. Además existen una serie de botones  para cambiar la granularidad y el filtrado de los elementos que forman la vista.
2. **Vista de fichero:** en esta parte del navegador se encuentra la vista de un fichero concreto. Si seleccionamos un fichero en la vista global, éste aparecerá expandido en esta vista. Podremos navegar por la estructura cruzada del proyecto, observando los avisos y los puntos de corte que usan, los miembros a los que afectan, etc. Además, existen una serie de botones para modificar el aspecto de la vista: el botón  para ordenar los miembros de la vista,  para ocultar los elementos que no son propios de AspectJ y el botón  para ocultar las asociaciones entre puntos de corte/avisos y los miembros a los que afectan.
3. **Código fuente:** en la parte derecha del navegador aparece el código fuente del archivo seleccionado en la vista global. Podemos editarlo y guardar los cambios mediante el botón  Save o bien usando el menú *File/Save*.
4. **Mensajes de error:** en la ventana situada en la parte inferior derecha del navegador se muestran los errores que tuvieron lugar durante el proceso de compilación.

Las características ofrecidas por la herramienta `ajbrowser` son limitadas, no deja de ser una simple interfaz de programación con las características más básicas. Por lo tanto, recomendamos el uso de entornos de desarrollo más potentes que incluyan soporte para AspectJ, como es el caso de Eclipse, cuyo plug-in de integración se aborda en el Anexo II de este documento.

Generador de documentación: `ajdoc`

Otra de las herramientas es el comando `ajdoc`, que se encarga de generar documentación HTML para proyectos AspectJ de forma similar a como lo hace el generador de documentación de java, `javadoc`. De hecho, `ajdoc`, esta basado en `javadoc`, y acepta las mismas opciones estándar que este. Se diferencian en que `ajdoc` no sólo documenta las clases java, sino que también documenta las construcciones características del lenguaje AspectJ: aspectos, puntos de corte, avisos, declaraciones intertipos..., enlazando cada uno de ellos con los elementos a los que afecta, de forma que leyendo la documentación, podemos navegar por la estructura cruzada de la aplicación en cuestión. La sintaxis del comando es:

```
ajdoc [opciones] [ficheros_fuente| paquetes | @fichero_lst ]
```

Al igual que para el compilador `ajc`, al comando `ajdoc` hay que pasarle por línea de comandos los ficheros fuentes o paquetes para los cuales queremos generar la documentación o bien usar un fichero donde se listen dichos ficheros fuentes o paquetes. El formato del fichero es el mismo que para el usado en el compilador,

introduciendo un argumento por línea, con la única diferencia que ahora el fichero no puede contener opciones, sólo nombres de ficheros o paquetes.

Para indicarle al generador de documentación el uso de un fichero con los argumentos que debe tomar, se usa el símbolo “@”, de forma que la llamada sería:

```
ajdoc @argumentos.lst
```

o bien la opción **argfile**:

```
ajdoc -argfile argumentos.lst
```

Además, el comando `ajdoc`, acepta las siguientes opciones [29]:

Opciones del comando <code>ajdoc</code>	
<code>-argfile fich_lst</code>	Especifica la ruta del fichero de argumentos.
<code>-classpath path⁵</code>	Especifica donde encontrar las <code>.class</code> de usuario.
<code>-bootclasspath path</code>	Modifica el directorio del que se toman las clases básicas de la distribución de Java.
<code>-d directorio</code>	Especifica el directorio donde se almacenara la documentación generada. El directorio por defecto es el actual.
<code>-help</code> <code>-?</code>	Imprime información sobre las opciones del compilador y su uso.
<code>-package</code>	Documenta las clases y miembros protegidos, públicos y con visibilidad de paquete.
<code>-protected</code>	Documenta las clases y miembros protegidos y públicos.
<code>-private</code>	Documenta todas las clases y miembros.
<code>-public</code>	Documenta solo las clases y miembros públicos.
<code>-sourcepath path</code>	Documenta todos los ficheros con extensión <code>.java</code> o <code>.aj</code> situados en los directorios especificados
<code>-version</code> o <code>-v</code>	Imprime por pantalla la versión de <code>ajdoc</code> .
<code>-verbose</code>	Muestra información sobre las actividades de <code>ajdoc</code>

Tabla 24: Opciones del comando *ajdoc*

Depurador: `ajdb`

La versión 1.0.6 de AspectJ o anteriores incluye la herramienta `ajdb` para depurar las clases generadas por el compilador. Las distribuciones posteriores a las 1.0.6 de AspectJ no incluyen este depurador, proponiendo una alternativa mejor, realizar la tarea de depuración a través de nuestro entorno de desarrollo habitual con soporte para AspectJ, como por ejemplo el entorno Eclipse.

No obstante, y aunque esté en desuso, vamos a profundizar en el uso de este depurador. Se basa en la herramienta de depuración de java `jdb` y por lo tanto acepta muchas de sus opciones. La sintaxis de una llamada es la siguiente:

```
ajdb [<opciones>] [<clase>] [<argumentos>]
```

⁵ argumento que contiene una lista de rutas a directorios o ficheros zip/jar, delimitadas por un símbolo que depende de la plataforma: “:” en sistemas UNIX y “;” en sistemas Windows

Donde `<opciones>` son las opciones que se le pasan al depurador, `<clase>` es el nombre de la clase a depurar y `<argumentos>` los argumentos que se le pasan al método `main` de la clase a depurar. La Tabla 25 muestra las opciones más importantes del depurador `ajdb`.

Opciones del comando <code>ajdb</code>	
<code>-read fichero</code>	Especifica la ruta de un fichero que contiene una lista de opciones, una por línea, que se deben pasar al depurador.
<code>-classpath path⁶</code>	Especifica donde encontrar las <code>.class</code> de usuario.
<code>-Dnombre=valor</code>	Establece el valor para una propiedad del sistema.
<code>-gui</code>	Ejecuta el depurador en modo gráfico.
<code>-sourcepath path</code>	Especifica donde encontrar los ficheros fuentes
<code>-workingdir dir</code>	Establece el directorio de trabajo del depurador.
<code>-Xoption</code>	Especifica una opción no estándar que se pasan a la máquina virtual
<code>-help</code>	Imprime información sobre las opciones del compilador y su uso.
<code>-verbose</code>	Muestra información sobre las actividades de <code>ajdoc</code>

Tabla 25: Opciones del comando `ajdb`

Algunas cuestiones a tener en cuenta a la hora de depurar una clase con `ajdb` son [29]:

- Compilar el proyecto que se desea depurar con el compilador `ajc` y la opción `-preprocess`. Esta opción le indica al compilador que no borre los ficheros intermedios creados por este para el proceso de entretelado y que se encuentran en el directorio de trabajo (opción `-workingdir`), ya que estos ficheros serán utilizados por el depurador.
- Debemos indicarle al depurador el directorio donde se encuentran los ficheros fuentes mediante la opción `-sourcepath path` al lanzar el depurador o bien una vez ejecutado escribiendo el comando `use path` en el shell del depurador.
- Algunos comandos útiles a la hora de realizar la depuración son:
 - `list <clase>`: lista el código fuente de una clase a depurar.
 - `stop on <clase>.<metodo>[(tipo argumento1,...)]`: establece un punto de corte (*breakpoint*) en el método especificado.
 - `stop on <clase>:<num_linea>`: establece un punto de corte (*breakpoint*) en una línea de la clase especificada.
 - `run <clase>`: ejecuta el código de la clase especificada.
 - `step`: ejecuta la instrucción actual.
 - `stepi`: ejecuta *i* instrucciones a partir de la actual.
 - `locals`: muestra los valores de las variables locales.
 - `exit`: para salir del depurador.

⁶ argumento que contiene una lista de rutas a directorios o ficheros zip/jar, delimitadas por un símbolo que depende de la plataforma: “.” en sistemas UNIX y “;” en sistemas Windows

Bibliografía

- [1] D. L. Parnas, “*On the criteria to be used in decomposing systems into modules*”. Communications of the ACM, 1972.
- [2] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, “*Discussing Aspects of AOP*”, Communications of the ACM 2001.
- [3] Página Web del grupo Demeter: <http://www.ccs.neu.edu/research/demeter/>
- [4] Página Web del grupo Xerox PARC: <http://www2.parc.com/csl/groups/sda/>
- [5] Página Web de AspectJ: <http://eclipse.org/aspectj/>
- [6] Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. “*Aspect-Oriented Programming*”, European Conference on Object Oriented Programming (ECOOP), 1997.
- [7] AOP/ST Home Page: <http://www.germany.net/teilnehmer/101,199268/>
- [8] L.Berger, “*Junction Point Aspect: A Solution to Simplify Implementation of Aspect Languages and Dynamic Management of Aspect Programs*”, ECOOP’00.
- [9] Cristina Lopes, “*D: A Language Framework for Distributed Programming*”, Ph.D. thesis, Northeastern University, Noviembre de 1997.
- [10] Página Web de AspectC: www.cs.ub.ca/labs/spl/projects/aspectc.html
- [11] Página Web de AspectC++: www.aspectc.org
- [12] Página Web AspectS www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS
- [13] Página Web de Apostle: www.cs.ub.ca/labs/spl/projects/apostle/
- [14] Página Web de Pythius: <http://sourceforge.net/projects/pythius/>
- [15] Página Web de AspectR: <http://aspectr.sourceforge.net>
- [16] Página Web de Java Aspect Component (JAC): <http://jac.aopsys.com>
- [17] Página Web de AspectWerkz: <http://aspectwerkz.codehaus.org>
- [18] Cristina Videira Lopes, Gregor Kiczales, “*Recent Developments in AspectJ™*”, ECOOP’98.
- [19] J. Irwin, J.-M.Loingtier, J. R. Gilbert, and G. Kiczales. “*Aspect-oriented programming of sparse matrix code*”. Lecture Notes in Computer Science, 1997.

- [20] Mendhekar A., Kiczales G. and Lamping J. “*RG: A Case-Study for Aspect-Oriented Programming*”. Xerox PARC, 1997.
- [21] Gamma E., Helm R., Johnson R., Vlissides J., “*Design Patterns: Elements of Reusable Object-Oriented Software*”. Addison-Wesley, 1994.
- [22] Hannemann J., Kiczales G., “*Design Pattern Implementation in Java and AspectJ*”. OOPSLA’2002.
- [23] Ramnivas Laddad, “*Aspectj in action. Practical aspect-oriented programming*”. Ed. Manning, 2003.
- [24] Joseph D. Gradecki, Nicholas Lesiecki, “*Mastering AspectJ. Aspect-Oriented Programming in Java*”. Ed. Wiley, 2003
- [25] Russell Miles, “*AspectJ Cookbook*”. Ed. O’Reilly, 2004.
- [26] AspectJ and AspectWerkz to Join Forces:
<http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/aj5announce.html>
- [27] Proyecto PatternTesting: <http://patterntesting.sourceforge.net>
- [28] Larman C. “*UML y Patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*”. 2ª Edición. Prentice-Hall, 2003.
- [29] Xerox Corporation, “*The AspectJ[™] Development Environment Guide*”, 2002.
<http://www.eclipse.org/aspectj/doc/devguide/progguide/index.html>
- [30] Coplien, J. O. “*Software Design Patterns: Common Questions and Answers*”, Rising L., (Ed.), *The Patterns Handbook: Techniques, Strategies, and applications*. Cambridge University Press, NY, 1998,
- [31] Página web Aspect-Oriented Design Pattern Implementations
<http://www.cs.ubc.ca/~jan/AODPs/>
- [32] Página Web del plug-in de integración de AspectJ con Eclipse:
www.eclipse.org/ajdt
- [33] Gregor Kiczales, et al. “*An Overview of AspectJ*”. ECOOP’01.
- [34] Erik Hilsdale, Jim Hugunin, Wes Isberg, Gregor Kiczales, Mik Kersten, “*Aspect-Oriented Programming with AspectJ*”, 2002.
- [35] Antonia Mª Reina Quintero, “*Visión General de la Programación Orientada a Aspectos*”, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, 2000.
- [36] Ken Wing Kuen Lee, “*An Introduction to Aspect-Oriented Programming*”, 2002.

- [37] Ramnivas Laddad, *"I want my AOP", Part 1,2 and 3*, JavaWorld, 2002.
- [38] Página web del framework Spring <http://www.springframework.org/>
- [39] J. Suzuki y Y. Yamamoto. *"Extending UML with Aspects: Aspect Support in the Design Phase"*. ECOOP'99.
- [40] J.L. Herrero, M. Sánchez y F. Sánchez. *"Changing UML metamodel in order to represent separation of concerns"*. ECOOP'00.
- [41] S. Clarke. *"Composition of Object-Oriented Software Design Models"*. Doctor of Philosophy in Computer Applications thesis. January, 2001, Dublin City University.
- [42] Martin Lippert and Cristina Videira Lopes, *"A Study on Exception Detection and Handling Using Aspect-Oriented Programming"*, Xerox PARC 1999
- [43] Xerox Corporation, *"The AspectJtm Programming Guide"*, 2002.
<http://www.eclipse.org/aspectj/doc/released/progguide/index.html>

