AspectJ en la Programación Orientada a Aspectos

Juan Manuel Nieto Moreno [*]
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla, España
nulain@yahoo.es

ABSTRACT

La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables.

AspectJ es una extensión al lenguaje Java orientada a aspectos y de propósito general, que extiende el popular lenguaje con una nueva clase de módulos llamados aspectos. Los aspectos cortan las clases, las interfaces y a otros aspectos mejorando la separación de competencias y haciendo posible localizar de forma limpia los conceptos de diseño del corte.

[*] Este documento es parte del proyecto final de carrera de Juan M. Nieto, tutelado por Antonia M. Reina del departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

1 INTRODUCCIÓN AL LENGUAJE

En AspectJ, un aspecto es una clase, exactamente igual que las clases Java, pero con una particularidad, que pueden contener unos constructores de corte, que no existen en Java. Los cortes de AspectJ capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, de constructores, lanzamiento de excepciones, acceso a atributos... Los veremos con más detalles en apartados siguientes. Los cortes no definen acciones, sino que describen eventos. De modo que en AspectJ los aspectos son constructores que trabajan cortando de forma transversal la modularidad de las clases de forma clara y específica.

Cualquier programa Java que sea válido, es también un programa valido en AspectJ. El compilador de AspectJ genera ficheros class conformes a la especificación Java byte-code, por lo que cualquier máquina virtual Java

(JVM) debe poder ejecutarlos. Al usar Java se obtienen todos los beneficios de este lenguaje, los cuales han quedado demostrados por su extensa aceptación entre la comunidad de programadores. Ello acelera además el proceso de aprendizaje en el lenguaje, puesto que sólo se ha de aprender a utilizar la parte nueva introducida por AspectJ.

Debemos diferenciar en AspectJ dos partes: la especificación del lenguaje y su implementación. La primera define el lenguaje en el que se escribe el código; en AspectJ la funcionalidad de los conceptos encapsulados se escribe en Java estándar, como el que ya todos conocemos y las extensiones del lenguaje se usan para definir dónde y cómo han de entrelazarse con el código de nuestra aplicación. La segunda parte es la implementación del lenguaje, provista de herramientas para compilar, depurar e integrar AspectJ con conocidos entornos de desarrollo integrados (*integrated development environment*, IDE) como JBuilder, Eclipse, Emacs...

Existen una serie de conceptos que son necesarios entender y conocer antes de poder utilizar AspectJ y que no tienen por qué entenderse de igual manera en los diferentes lenguajes que implementan los conceptos de la programación orientada a aspectos. Se hará referencia siempre al término inglés utilizado para designar estos conceptos en la documentación del lenguaje para facilitar así la lectura de otros documentos. No obstante al principio se da una explicación del significado de estos.

2 CROSSCUTTING EN ASPECTJ

En AspectJ, la implementación de las reglas de entrelazado se denomina *crosscutting* y a los conceptos encapsulados que originalmente estaban repartidos por todo o parte de la aplicación se les denomina *crosscutting concerns*. El proceso de combinación de éste código con el de la aplicación Java se conoce como *weaving process*, es decir, proceso de entrelazado. Las reglas de entrelazado definen de una manera sistemática las diferentes partes del código que deben ser afectadas por los conceptos encapsulados en los aspectos, respondiendo a la cuestión qué hacer cuando cierto punto de la ejecución de un programa se alcanza. A dichos puntos de ejecución se les llama *pointcuts*. En AspectJ se definen dos tipos de *crosscutting*: estático y dinámico.

2.1 CROSSCUTTING DINÁMICO

Se dice que un aspecto afecta a la aplicación de manera dinámica, o usando el término inglés, se habla de *dynamic crosscutting*, cuando dicho aspecto afecta al comportamiento global o una parte de la aplicación. *Crosscutting* dinámico es la forma más frecuente en AspectJ de entrelazado y es la manera de dotar de nuevo comportamiento a la ejecución de un programa. De esta forma se puede enriquecer o según sea el caso, reemplazar el flujo de ejecución de un programa, de forma que éste vaya por diferentes requisitos que en nuestro código se tengan encapsulados en diferentes módulos. Por ejemplo, algo muy frecuente es indicar que cierta acción ha de ser realizada antes de la ejecución de ciertos métodos o de la captura de determinadas excepciones dentro de un conjunto de clases, definiendo en un módulo aparte los puntos dónde el código se ha de entremezclar (*weaving points*) y la acción a realizar en el momento en el que se alcancen dichos puntos.

2.2 CROSSCUTTING ESTÁTICO

Se dice que un aspecto afecta a la aplicación de manera estática, o usando el término inglés, se habla de *static crosscutting*, cuando se añaden modificaciones en la estructura estática del programa, esto es, en las clases, interfaces y aspectos. Por tanto, este tipo de técnica no implica modificación del comportamiento del sistema. En la mayor parte de los casos su uso se requiere para dar soporte a la implementación de los conceptos de *crosscutting* dinámico. Por ejemplo, puede simplificarse la implementación de éstos modificando las relaciones de herencia entre clases e interfaces, para poder trabajar así con las clases abstractas en lugar de tener que hacerlo con las implementaciones específicas; o puede ser de utilidad modificar los atributos o métodos de una clase o interfaz para poder así definir estados específicos y comportamientos que pueden ser usados en las acciones que se lleven a cabo con entrelazado dinámico. También es posible con entrelazado estático definir advertencias (*warnings*) y errores (*errors*) en tiempo de compilación de una manera sencilla y que afecte a todo o una parte de la aplicación, permitiendo definir los puntos de corte de una manera general con predicados lógicos.

3 LAS REGLAS DE ENTRELAZADO

AspectJ es una extensión al lenguaje de programación Java para especificar las reglas de entrelazado para los aspectos que afectan a una aplicación ya sea de forma estática o dinámica. Las extensiones están diseñadas de forma que el proceso de aprendizaje de un programador Java para usarlas sea mínimo. A continuación se explican los constructores que utiliza AspectJ para especificar las reglas de entrelazado de los aspectos con la aplicación de manera precisa (al igual que en casos anteriores, especifico en primer lugar el término inglés):

- **Join points**, puntos de enlace. Un punto de enlace es un sitio identificable en la ejecución de un programa. Podría ser una llamada a un método o la asignación de un nuevo valor a un atributo de una clase. En AspectJ todo gira alrededor de los puntos de enlace, ya que son los lugares donde los valores añadidos de los aspectos son enlazados con el código.

- ***Pointcuts***, puntos de corte. Estos son constructores donde se especifican los puntos de enlace y se captura información referente al contexto. Por ejemplo, un punto de corte puede referirse a un punto de enlace que sea una llamada a un método, pudiendo, en tal caso, capturar el contexto del método, siendo éste, el objeto sobre el que se invoca el método, así como los argumentos de la llamada. Para entender la diferencia entre puntos de enlace y puntos de corte, se puede pensar en estos últimos como las reglas de especificación y en los primeros como las situaciones que satisfacen dichas reglas.

- ***Advice***, avisos. Este es el código que se debe ejecutar en los puntos de enlace que se especifiquen en un punto de corte. Los avisos se pueden ejecutar antes, después o alrededor. Alrededor quiere decir que se puede, con el aviso, modificar la ejecución del código a la altura del punto de enlace, reemplazarlo o ignorarlo, si se es eso lo que se desea. Por ejemplo, se podría especificar un aviso para registrar en un fichero histórico cierto mensaje, siempre antes de la ejecución de ciertas funciones que se encuentran dispersas por diferentes módulos. El cuerpo de un aviso se parece mucho al cuerpo de un método, ya que encapsula la lógica que debe ser ejecutada cuando se alcanza cierto punto de enlace en la ejecución. Los puntos de corte junto con los avisos, son las herramientas para implementar entrelazado dinámico. Los puntos de corte identifican dónde y los avisos lo completan indicando qué hacer.

- ***Introduction***, introducciones. Si lo anterior era para el entrelazado dinámico, las introducciones se usan para el estático. Con ella es posible introducir cambios a las clases, interfaces y aspectos del sistema para así permitir el entrelazado dinámico. Conlleva cambios estáticos en los módulos que no afectan directamente a su comportamiento. Por ejemplo, se pueden añadir métodos o atributos a clases y después usarlos tal como si hubieran sido definidos por la propia clase.

- ***Compile-Time declaration***, declaraciones en tiempo de ejecución. Esta es otra forma de implementar técnicas de entrelazado estático. Permite añadir advertencias y errores para en tiempo de compilación detectar ciertos patrones de uso que queramos advertir o prohibir (en el caso de que los identifiquemos

como errores). Por ejemplo, se puede declarar que es un error hacer uso de las llamadas del paquete `java.sql` fuera del paquete `dataAccess` de la aplicación.

– ***Aspect***, aspectos. Estos son la unidad central de AspectJ, al igual que lo son las clases en Java. Contienen el código que expresa las reglas de entrelazado tanto para los aspectos dinámicos como los estáticos. Puntos de corte, avisos, introducciones y declaraciones de compilación se combinan en los aspectos. Además de estos elementos propios de AspectJ, los aspectos pueden contener atributos, métodos y clases anidadas, tal como los tienen las clases normales en Java.

Veamos cómo funciona todo ello junto. Cuando se diseña un comportamiento que se encuentra disperso por la aplicación, lo primero que hay que hacer es identificar los puntos de enlace donde se quiere enriquecer el comportamiento o bien modificarlo. Una vez hecho, se diseña cual será el nuevo comportamiento. Para hacer esto, primero se escribe un aspecto que sirva como módulo para el comportamiento global. En él se escriben los puntos de corte para capturar los puntos de enlace que se deseen. Finalmente, se crean avisos para esos puntos de corte y se define dentro del cuerpo de los avisos las acciones que se deben realizar cuando se ejecute el código que definen los puntos de enlace. Es posible que para permitir ciertos avisos sea necesario hacer uso de entrelazado estático.

Un ejemplo sencillo sería el siguiente: considere una aplicación en la que queremos añadir un aspecto para registrar la ejecución de todos los métodos públicos de dete
rminados paquetes. Lo primero es crear el aspecto que encapsule dicho comportamiento, que como es fácil observar, está distribuido por diferentes clases. Después se escribe un punto de corte en el aspecto para capturar todos los puntos de enlace para las operaciones públicas en el conjunto de clases que se desee. Por último, se escribe un aviso para este punto de corte, donde, en su cuerpo, se escribirían las instrucciones para registrar la llamada en un fichero o de cualquier otra forma. Si se quisiera mantener algún tipo de estado sobre las clases sobre las que se han registrado las llamadas, como podría ser el número de métodos ejecutados en cada clase, debería usarse introducción estática en el aspecto para añadir un atributo entero a todas las mismas clases anteriores. En tal caso, el aviso puede actualizar y leer este entero y escribirlo también a la información que se registra. El siguiente apartado nos muestra un ejemplo muy parecido.

4 DETALLES DEL LENGUAJE

En esta sección se presentan los aspectos más relevantes del lenguaje AspectJ, tratando en primer lugar los puntos de corte, viendo su sintaxis y modo de uso; los avisos, sintaxis y tipos; el manejo de la información contextual; las reglas de entrelazado estático disponibles; extensión e instancias de los aspectos; y por último, una visión rápida de la API de AspectJ.

4.1 PUNTOS DE CORTE

Los puntos de corte (*pointcuts*) capturan o identifican puntos de enlace en el flujo del programa. Una vez definidos con los constructores que proporciona el lenguaje, puede especificarse mediante reglas de entrelazado cómo combinar los comportamientos definidos en los aspectos con el código original. Además los puntos de corte dan acceso al contexto del programa y las acciones se pueden beneficiar de la información contextual para llevar a cabo su cometido.

En AspectJ las *pointcuts* se pueden declarar anónimos o con un nombre. Al igual que las clases, un punto de corte anónimo se define en el lugar donde se usa, que bien podría ser en un *advice* o en la definición de otro *pointcut*. Los puntos de corte que se definen con un nombre pueden después ser referenciados desde múltiples partes del código, haciéndoles reusables. Su sintaxis es la siguiente:

```
pointcut nombre(argumentos) : definición
```

Por `definición` se entiende el conjunto de reglas que definen los puntos de enlace donde se quiere insertar cierto comportamiento. El nombre que se utilice para definir el *pointcut* es el que luego ha de utilizarse en los *advices* para hacer referencia a ellos. Véase el siguiente ejemplo:

```
pointcut operacionesPublicas(): call(public *.*(..));

before() : operacionesPublicas(){
    //acciones a realizar antes de las llamadas a las funciones públicas
}
```

Es posible también definir puntos de corte anónimos, es decir, sin un nombre que los referencie. Se definen en el lugar donde se usan y no pueden reusarse. No se aconseja, por tanto, usarlos si la expresión que los define es complicada o interesa utilizarlos en varios *advices*. Su sintaxis dentro de la definición de los *advices* es la siguiente:

```
[especificación-del-advice]: definición-del-pointcut
```

Si quisiéramos escribir el mismo ejemplo anterior pero con el *pointcut* anónimo, se haría como lo muestra el siguiente ejemplo –obsérvese que lo importante, la definición del punto de enlace, es igual al anterior:

```
before(): call(public *.*(..)) {
     //acciones a realizar antes de las llamadas a las funciones públicas
}
```

SINTAXIS Y SEMÁNTICA DE LOS PUNTOS DE ENLACE

Los puntos de corte harán referencia a una serie de puntos de interés (*joinpoints*) en el programa, por lo que se necesita un lenguaje eficiente para definirlos. AspectJ utiliza una sintaxis basada en elementos comodines que sirven para capturar puntos de enlace que comparten características comunes. Dichos elementos son los siguientes:

- * un asterisco denota cualquier número de caracteres excepto el punto.
- .. dos puntos seguidos denota cualquier número de caracteres incluyendo el punto.
- + el símbolo de la suma denota los descendientes de una clase.

Más adelante se verá que el significado de estos caracteres dependerá de los elementos a los que acompañen. Al igual que en Java y en muchos lenguajes las expresiones se pueden combinar para dar lugar a expresiones más complejas. Para ello se utilizan los siguientes operadores unarios y binarios:

- `!` la exclamación de cierre es un operador unario que denota todos los puntos de enlace, excepto aquellos que se especifiquen en la expresión que siga. Por ejemplo, `!within(aspects.*)` denota todos los puntos de enlace excepto aquellos del paquete `aspects`.
- `||` dos barras verticales es el operador binario que permite combinar puntos de enlace con nombre y anónimos, de tal forma que se cumpla alguno de los lados de la expresión.
- `&&` es el operador binario que permite combinar puntos de enlace con nombre y anónimos, de tal forma que se cumplan ambos lados de la expresión.

Además pueden usarse los paréntesis combinados con estos operadores con objeto de alterar la precedencia por defecto de las expresiones y facilitar también la legibilidad. Los elementos expuestos tendrán diferente significado según acompañen a paquetes, tipos (clases, interfaces o aspectos), métodos o atributos. Veamos algunos ejemplos del significado que adquieren según cómo y donde se usen:

Tabla 1 Patrones de puntos de enlace

Patrón	**Elementos referidos**
`Manager`	Tipos de nombre `Manager`.
`*Manager`	Tipos con un nombre que termine en `Manager`.
`java.*.Name`	Tipo `Name` en cualquiera de los subpaquetes del paquete `java`.
`java..*`	Cualquier tipo del paquete `java` o todos sus subpaquetes

`javax..*Set+`	Todos los tipos en el paquete `javax` o sus subpaquetes cuyo nombre termine en `Set` y sus subtipos.
`public void Stack.clean()`	El método `clean()` de la clase `Stack` que tenga acceso público, devuelva `void` y no tenga argumentos.
`public void Stack.push(Integer) throws RangeExceded`	El método `push()` de la clase `Stack` que tenga acceso público, devuelva `void`, tenga un único argumento de tipo `Integer` y lance la excepción `RangeExceded`.
`public void Stack.push*(*)`	Todos los métodos públicos de la clase `Stack` cuyo nombre empiece con `push` y tenga un único argumento de cualquier tipo.
`public void Stack.*()`	Todos los métodos en de la clase `Stack` que tengan acceso público devuelvan `void` y no tengan argumentos.
`public * Stack.*()`	Todos los métodos públicos de la clase `Stack` que no tengan argumentos y devuelvan cualquier tipo.
`public * Stack.*(..)`	Todos los métodos públicos de la clase `Stack` que tomen cualquier número de argumentos.
`* Stack.*(..)`	Todos los métodos de la clase `Stack`.
`!public * Stack.*(..)`	Todos los métodos que no tengan acceso público de la clase `Stack`.
`public static void MyClass.main (String[] args)`	El método estático `main()` de la clase `MyClass` con acceso público.
`* Stack+.*(..)`	Todos los métodos de la clase `Stack` o sus subclases.
`* java.io.Reader.read(char[],..)`	Métodos `read()` de la clase `Reader` con cualquier número de argumentos, con el primero del tipo `char[]`.
`* javax..*.add*List(List+)`	Métodos que empiecen con `add` y terminen en `List` en el paquete `javax` o sus subpaquetes y que tomen un argumento del tipo `List` o sus subtipos.
`* *.*(..) throws SQLException`	Cualquier método que lance `SQLException`.
`public MyClass.new()`	Constructores públicos de la clase `MyClass` sin argumentos.
`public MyClass.new(Integer)`	Constructores públicos de la clase `MyClass` que tomen un único argumento de tipo `Integer`.
`public MyClass.new(..)`	Todos los constructores de la clase `MyClass` que tomen cualquier número de argumentos.
`public *MyClass.new(..)`	Constructores públicos de clases cuyo nombre termine en `MyClass`.
`public MyClass+.new(..)`	Constructores públicos de la clase `MyClass` o sus subclases.
`public MyClass(..) throws InvalidFormatException`	Constructores públicos de la clase `MyClass` que lancen InvalidFormatException.
`private float MyClass.value`	Atributo privado `value` de la clase MyClass.
`* MyClass.*`	Todos los atributos de la clase MyClass.
`!public static * MyClass..*.*`	Todos los atributos no públicos estáticos de la clase MyClass y de sus subpaquetes.
`Public !final *.*`	Atributos no finales públicos de cualquier clase.

Como se ve en la tabla, el significado de los caracteres comodines depende si se aplican a tipos, métodos o atributos. Referido a los tipos, el asterisco especifica una parte de la clase, interfaz o paquete; los dos puntos seguidos se usan para denotar todos los subpaquetes; y el símbolo de suma denota los subtipos (subclases o subinterfaces). Mientras que referido a los métodos, los dos puntos seguidos denotan cualquier tipo y número de argumentos que tome el método.

IMPLEMENTANDO LOS PUNTOS DE CORTE

Una vez que tenemos una manera eficiente de definir los puntos de enlace en el código, se necesita una manera de utilizarlos. Para referirnos a ellos podemos hacerlo según el tipo de punto de enlace al que representan, como puede ser llamadas a métodos, ejecución de estos, lectura de atributos... Véase la siguiente tabla para una descripción de las diferentes posibilidades que ofrece AspectJ. En la terminología del lenguaje se hace referencia a ellos como *Kinded pointcuts*.

Tabla 2 Tipos de puntos de corte

Categoría	**Sintaxis**
Ejecución de métodos	`execution(MethodSignature)`
Llamada a métodos	`call(MethodSignature)`
Ejecución de constructores	`execution(ConstructorSignature)`
Llamada a constructores	`call(ConstructorSignature)`
Inicialización de clases	`staticinitialization(TypeSignature)`
Lectura de atributos	`get(FieldSignature)`
Escritura de atributos	`set(FieldSignature)`
Captura de excepciones	`handler(TypeSignature)`
Inicialización de objetos	`initialization(ConstructorSignature)`
Pre-inicialización de objetos	`preinitialization(ConstructorSignature)`
Ejecución de avisos	`adviceexecution()`

Se puede escribir un sencillo ejemplo para comprender la cantidad de puntos de enlace que tenemos en una pequeña aplicación e imaginarnos él número de éstos que podemos llegar a tener cuando se trate de una aplicación real. Definamos el siguiente aspecto:

```
public aspect ImprimeTodos{

  pointcut todos(): if(true);

   before(): todos() && !within(ImprimeTodos){
        System.out.println(thisJoinPoint);
   }
}
```

Para definir el *pointcut* `todos()` se usa un punto de corte condicional (véase Tabla 7) cuya condición lógica está siempre a cierto. Para evitar caer en un bucle infinito se utiliza los llamados puntos de corte basados en localización (véase Tabla 4), de modo que se excluyen los puntos de enlace de la propia clase mediante `!within`. El programa que utilizaremos como ejemplo es el siguiente:

```
public class MiClase{

   public static void main(String args[]){
       System.out.println("* Inicio *");
       new MiClase().operacion();
   }

   public void operacion(){
      System.out.println("* Hola *");
   }
}
```

Ejecutándolo la clase `MyClass` con el aspecto `ImprimeTodos` tenemos la siguiente salida donde se ve cómo se han alcanzado casi todos los `pointcuts` expuestos en la Tabla 2:

```
staticinitialization(MiClase.<clinit>)
execution(void MiClase.main(String[]))
get(PrintStream java.lang.System.out)
call(void java.io.PrintStream.println(String))
* Inicio *
call(MiClase())
initialization(MiClase())
execution(MiClase.<init>)
execution(MiClase())
call(void MiClase.operacion())
execution(void MiClase.operacion())
get(PrintStream java.lang.System.out)
call(void java.io.PrintStream.println(String))
* Hola *
```

Este ejemplo nos muestra la cantidad de posibilidades que tenemos a la hora de definir puntos de enlace. Para poder ser más concretos en la definición de los puntos de corte nos podemos servir de un buen número de *pointcuts* disponibles en AspectJ y que se muestran en las siguientes tablas.

El siguiente conjunto de *pointcuts* captura puntos de enlace siempre y cuando ocurran en el contexto de otro *pointcut*. Por eso, declaran siempre otro punto de corte como argumento. Por ejemplo, `cflow(doIt())`, identifica los puntos de enlace que ocurren entre recibir las llamadas al método `doIt` y volver de estas llamadas (incluyendo la propia llamada a `doIt`). Además de `cflow(Pointcut)` existe otra posibilidad, `cflowbelow(Pointcut)`, cuya utilidad veremos con un ejemplo a continuación de la siguiente tabla:

Tabla 3 Pointcuts basados en control-flow

Pointcut	**Descripción**
`cflow(call(* MyClass.operation(..))`	Puntos de enlace durante la ejecución del método `operation()` de la clase `MyClass`, incluyendo la llamada al método `operation()` de `MyClass`.
`cflowbelow(` `  call(* MyClass.operation(..))`	Puntos de enlace durante la ejecución del método `operation()` de la clase `MyClass`, excluyendo la llamada al método `operation()` de `MyClass`.
`cflow(Operations())`	Todos los puntos de enlace en el contexto de los puntos de enlace capturados por el punto de corte `Operations()`.
`cflow(staticinitializer(StClass))`	Puntos de enlace durante la inicialización de la clase `StClass`.
`cflowbelow(execution(MyClass.new(..))`	Puntos de enlace durante la ejecución de alguno de los constructores de la clase `MyClass`, excluyendo la ejecución misma del constructor.

El *pointcut* `cflowbelow()` puede ser de mucha utilidad en los algoritmos recursivos. La recursividad es una técnica muy útil, pero que genera muchas llamadas y, consecuentemente, muchos puntos de enlace repetitivos. Por lo general, los más interesantes de ellos serán los referentes a la primera y la última llamada. El siguiente ejemplo muestra como `cflowbelow()` puede ser de utilidad en este caso. Sea el siguiente programa recursivo que muestra la jerarquía de clases de un objeto:

```
public class Jerarquia{
   public static void main(String args[]){
        try{
               padres(Class.forName(args[0]));
        }catch(Exception e){}
   }

   static void padres(Class obj){
      if (null==obj)
         return;
       System.out.println(obj.getName());
       padres(obj.getSuperclass());
   }
}
```

Si probamos la aplicación `Jerarquia` con la clase `javax.servlet.http.HttpServlet` como entrada, obtendremos la siguiente salida:

```
javax.servlet.http.HttpServlet
javax.servlet.GenericServlet
java.lang.Object
```

En el caso de que quisiéramos enlazar avisos con las llamadas recursivas, podría interesarnos distinguir únicamente la primera llamada. El siguiente aspecto define dos *pointcuts*: `llamadas()`, que captura todas las llamadas al método `padres(Class)`; y `primera()` que captura sólo la primera de dichas llamadas.

```
public aspect LlamadasRecursivas{

  pointcut llamadas(): call(void Jerarquia.padres(Class));
  pointcut primera(): llamadas() && !cflowbelow(llamadas());

  before(): llamadas() {
       System.out.println("Otra: "+thisJoinPoint);
  }
```

```
  before(): primera() {
       System.out.println("Primera: "+thisJoinPoint);
  }

}
```

Cómo es deseado, el segundo de los avisos sólo se ejecutará una vez –antes de la primera llamada a `padres(Class)`. Combinando la clase `Jerarquia` con el aspecto y dándole la misma entrada tenemos la siguiente salida:

```
Otra: call(void Jerarquia.padres(Class))
Primera: call(void Jerarquia.padres(Class))
javax.servlet.http.HttpServlet
Otra: call(void Jerarquia.padres(Class))
javax.servlet.GenericServlet
Otra: call(void Jerarquia.padres(Class))
java.lang.Object
Otra: call(void Jerarquia.padres(Class))
```

El siguiente grupo de puntos de corte que se definen son los que se basan en la localización del código referenciado. Hay dos categorías: `within(TypePattern)` y `withincode(MethodSignature)`. El primero se usa para capturar todos los puntos de enlace en el cuerpo de las clases, aspectos o subclases especificados en `TypePattern`. El segundo se usa para capturar los puntos de enlace dentro de una estructura léxica de una clase o un método. La siguiente tabla muestra algunos ejemplos de su uso:

Tabla 4 Pointcuts basados en localización

Pointcut	**Descripción**
`within(MyClass)`	Puntos de enlace dentro de la clase `MyClass`.
`within(MyClass+)`	Puntos de enlace dentro de la clase `MyClass` y sus subclases.
`withincode(* MyClass.operation(..))`	Puntos de enlace dentro del método `operation()` de la clase `MyClass`.
`withincode(* *printer.flow(..))`	Puntos de enlace dentro del método `flow()` en clases cuyo nombre acabe en `printer`.

Un uso común de `within()` es para excluir los puntos de enlace del aspecto donde se define. Por ejemplo, el siguiente punto de corte excluye los puntos de enlace correspondientes a las llamadas a los métodos `print` de la clase `java.io.Stream` que ocurran dentro del aspecto `TraceAspect`:

```
call(* java.io.PrintStream.print*(..)) && !within(TraceAspect)
```

Los siguientes puntos de corte, `this` y `target`, están basados en los tipos de los objetos en tiempo de ejecución. Con `this` se referencia al objeto actual, mientras que con `target` es al objeto sobre el que se llama al método. Estos puntos de corte sirven además para recoger el contexto en el punto de enlace. `this` toma la forma `this(Type)` y así captura los puntos de enlace asociados con objetos del tipo especificado `Type`. Al igual, `target` es también de la forma `target(Type)`.

Tabla 5 Pointcuts de objetos

Pointcut	**Descripción**
`this(MyClass)`	Puntos de enlace donde `this` es una instancia de `MyClass` o sus subclases (`this` es el objeto que se esté ejecutando actualmente).
`target(MyClass)`	Puntos de enlace en los que el objeto donde se llama el método es instancia de `MyClass` o sus subclases.

Nótese como `this()` y `target()` toman como argumentos `Type` y no `TypePattern`. Eso quiere decir que no podremos usar los comodines asterisco o dos puntos seguidos (el comodín referente a los subtipos, `+`, no se necesita, pues éstos son capturados por las reglas de herencia de Java). Obsérvese también que, al no ejecutarse

los métodos estáticos sobre un objeto, éstos no tendrán un objeto `this` asociado, por lo que el punto de corte no incluirá este tipo de métodos (igualmente ocurre con `target)`. Esta última es la diferencia de uso que puede hacerse con `within`.

El siguiente *pointcut* que se basa en el tipo de los argumentos del punto de enlace. Por argumentos ha de entenderse lo siguiente: en los métodos y constructores los argumentos son los mismos que los argumentos de éstos; para los puntos de enlace que capturan excepciones, la excepción capturada es considerada el argumento; y para los accesos a atributos en escritura, se considera como argumento el nuevo valor a escribir sobre el atributo. Este tipo de punto de corte sirve también para capturar la información contextual en el punto de enlace (más detalles en el apartado 4.3).

Tabla 6 Pointcuts de argumentos.

Pointcut	Descripción
`args(String,..,int)`	Puntos de enlace en los que el método tenga como primer argumento un `String` y como último un `int`.
`args(SQLException)`	Puntos de enlace con un único argumento de tipo `SQLException.` Capturaría un método que tome un único argumento de tipo `SQLException`, la escritura sobre un atributo con un valor de tipo `SQLException`, o la captura de una excepción del tipo `SQLException`.

El último conjunto de *pointcuts* del lenguaje permite usar expresiones condicionales, lo cual tendrá la consecuencia de capturar sólo los puntos de enlace cuando dichas condiciones se evalúen a cierto. Toman la forma `if(`*`BooleanExpression`*`)`. La expresión lógica se evalúa en tiempo de ejecución, por los que debe ser válida en el contexto donde se sitúe. Véanse algunos ejemplos en la siguiente tabla.

Tabla 7 Pointcuts condicionales

Pointcut	Descripción
`if(Buffer.size() > MaxAllowed)`	Puntos de enlace que ocurran a partir de que el tamaño del buffer supere el valor de `MaxAllowed`.
`if(thisJoinPoint.getTarget()!=null)`	Puntos de enlace donde el objeto donde se estén ejecutando los métodos capturados no sea nulo.

Hemos examinado todas las posibilidades para definir puntos de corte en el lenguaje AspectJ. En la siguiente sección se explica el concepto de *advice*, que hará uso de los puntos de corte que hemos definido, para dotar de cierto comportamiento al programa en los puntos de enlace que se especifiquen.

4.2 AVISOS

Los puntos de corte son, junto con los avisos (*advices*), las herramientas para implementar conceptos de entrelazado dinámico en AspectJ (véase el apartado
Crosscutting dinámico, página 2). Los avisos son constructores parecidos a los métodos, donde se indica el código que se debe ejecutar en los puntos de enlace que se especifiquen en un punto de corte. Los avisos se pueden ejecutar antes, después o “alrededor” del punto de enlace. Alrededor quiere decir que puede modificarse con el aviso la ejecución del código a la altura del punto de enlace, reemplazarlo o ignorarlo, si se es eso lo que se desea. El cuerpo de un aviso se parece mucho al cuerpo de un método, ya que encapsula la lógica que debe ser ejecutada cuando se alcanza cierto punto de enlace en la ejecución. Los puntos de corte identifican *dónde* y los avisos lo completan indicando *qué* hacer.

La siguiente figura muestra una secuencia de ejecución y algunos de los posibles puntos de enlace donde se puede introducir nuevos comportamientos con avisos:

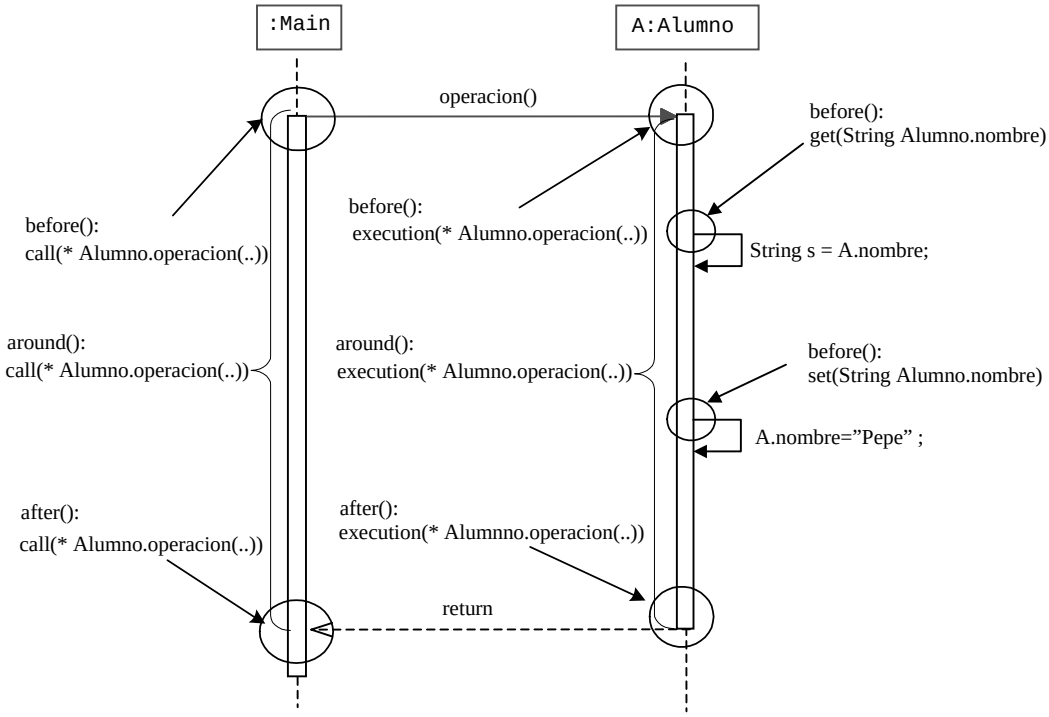


Figura 1 Diagrama de secuencia con puntos de enlace.

La figura representa un fragmento de un programa donde hay una clase `Main` y un objeto `A` de clase `Alumno`. Además se puede ver que la clase `Alumno` tiene un atributo `nombre` de tipo `String` y un método `operacion()`. En la figura se han dibujado con círculos los puntos de enlace donde los avisos `before` y `after` pueden alterar el comportamiento del programa. La parte de las líneas de vida entre círculos representa la zona de influencia de los avisos `around`. Se pueden observar los puntos de enlace de uso más habitual en AspectJ, estos son, llamadas y ejecución de funciones y acceso a atributos.

SINTAXIS DE LOS AVISOS

Los avisos pueden descomponerse en tres partes: su declaración, la definición del punto de corte (o el nombre del punto de corte que vaya a utilizar) y el cuerpo. Por claridad, los siguientes ejemplos harán uso del mismo punto de corte y que será el que a continuación se define:

```
pointcut connectionOperation(Connection connection):
  call(* Connection.*(..) throws SQLException) && target(connection);
```

El *pointcut* `connectionOperation` consta de dos puntos de corte anónimos: el primero captura todas las llamadas a métodos de la clase `Connection` que lancen la excepción `SQLException` –independientemente de los argumentos que tome o el valor que devuelva; y el segundo, `target(connection)`, captura el objeto sobre el que se hacen las llamadas. Al no ser anónimo y tener un nombre, podemos utilizarlo en la definición de los avisos cuantas veces queramos. Veamos dos ejemplos usando `before` y `around`:

```
before(Connection connection):  connectionOperation (connection){
   System.out.println("Performing operation on " + connection);
}


Object around(Connection connection) throws SQLException:
   connectionOperation(connection){
     System.out.println("Operación "+thisJoinPoint+" comenzada en "+ connection);
     proceed(connection);
     System.out.println("Operación "+thisJoinPoint+" terminada en "+ connection);
}
```

La parte anterior a los dos puntos es la declaración del aviso. En ella se especifica cuándo se ejecutará el aviso en relación con el punto de enlace, es decir, antes, después o alrededor. Además especifica la información contextual que estará disponible en el cuerpo del aviso, tal como el objeto sobre el que se ejecuta la llamada o los argumentos de esta. También habrá que indicar las excepciones que se puedan lanzar en el cuerpo del aviso, como se ha tenido que hacer en el segundo ejemplo con `SQLException`. A continuación, después de los dos puntos viene la definición del punto de corte. Pueden definirse los puntos de enlace o hacerse referencia a un punto de corte anteriormente definido, como se ha hecho en estos dos casos. Por último, el cuerpo del aviso contiene, al igual que el cuerpo de un método, las acciones a realizar y igualmente está encerrado entre llaves `{}`. La llamada al método `proceed()` en los avisos `around` ejecuta el método original capturado.

TIPOS DE AVISOS

Como se ha comentado, existen tres posibilidades para definir el momento de actuación de los avisos (`before`, `after` y `around`). Cada una de estas se explica a continuación.

Los avisos **before** se ejecutan antes de la ejecución del punto de enlace capturado. Al terminar, dan paso al método original. En el caso de que se lanzara una excepción en el cuerpo del aviso, el método original no se ejecutará. Este tipo de aviso se usa típicamente para llevar a cabo tareas de control o precondiciones, como el control de los parámetros, logging, autentificación... Véase el siguiente ejemplo:

```
before(Connection connection):  connectionOperation (connection){
     checkConnectionValidity(connection);
}
```

En este caso, el cuerpo del aviso podría comprobar mediante el método `checkConnectioValidity` y antes de llamar a las operaciones que captura `connectionOperation`, que el objeto `connection` que se va a utilizar no está corrupto.

Los avisos **after** se ejecutan después de la ejecución del punto de enlace capturado. En este caso hay que distinguir además el caso en el que un método termine con el lanzamiento de una excepción. Para ello AspectJ ofrece tres variantes para el aviso `after`: después de terminar normalmente, después de terminar lanzando una excepción y después de terminar, sea como sea (es decir, incluyendo los dos casos anteriores). La sintaxis para cada uno de esos casos es la siguiente:

```
after () : connectionOperation (connection) { ... }
```

Este aviso se ejecuta después de las llamadas capturadas por el punto de corte `connectionOperation`, ya terminen éstas normalmente o con el lanzamiento de una excepción.

```
after () returning (Object x) : connectionOperation(connection) { ... }
```

Este aviso se ejecutará después de las llamadas a los métodos capturados por el punto de corte `connectionOperation`, en el caso de que la ejecución de éstas se complete sin lanzar ninguna excepción. El valor devuelto recibe el nombre `x`, por lo que se podrá referenciar y usar en el cuerpo del aviso. Si el método termina con una excepción, el cuerpo del aviso no se ejecutará.

```
after () throwing (SQLException exc) : connectionOperation(connection){ ... }
```

Este aviso se ejecuta después de las llamadas a los métodos capturados por el punto de corte `connectionOperation`, en el caso de que éstas terminen de forma anómala devolviendo la excepción `SQLException`, según se ha indicado. A la excepción se le da el nombre `exc` en el cuerpo del aviso, por lo que así podrá utilizarse. Si el método termina normalmente, el cuerpo del aviso no se ejecutará.

Los avisos **around** envuelven el punto de enlace capturado, de tal forma que se puede anular la ejecución del punto de enlace o alterar el contexto antes de proceder. Permite también ejecutar varias veces la lógica capturada

en el punto de enlace, cada vez, por ejemplo, con diferentes argumentos. Es sin duda la más flexible de las posibilidades que ofrece AspectJ. Para permitir la ejecución por el punto de enlace capturado dentro del cuerpo del aviso, hay que utilizar una llamada a un método clave de nombre `proceed()`. Este debe ser invocado con el mismo número y tipo de argumentos que el original y devolver el mismo tipo de argumento que este. Véase el siguiente ejemplo:

```
Object around() memoryOperations(){
     try{
        proceed();
     }catch(Exception e){ /*Manejar la excepción e */ }
}
```

La intención del ejemplo es tratar de manera uniforme las excepciones que puedan lanzar las operaciones capturadas por el punto de corte `memoryOperations`. Lo que se hace es envolver la ejecución de `proceed()` entre `try/catch`, lo que permite eliminar este código de cada uno de los métodos y manejarlo aquí de manera igual para todos.

4.3 PASO DE INFORMACIÓN CONTEXTUAL

Una de las mayores utilidades en los avisos es el paso de la información contextual del punto de enlace. De esta forma se tiene acceso a información respecto a los métodos que se capturan, así como a los argumentos que estos reciben. Esto permite poder actuar de diferentes formas según el método o el tipo de los objetos capturados, así como alterar o hacer comprobaciones sobre los parámetros antes de pasar el control a los puntos de enlace. Dicha información es lo que se conoce como contexto. AspectJ dispone de una serie de herramientas para exponer dicho contexto en los avisos, como son `target()`, `this()` y `args()`. La sintaxis es muy similar a la utilizada en Java para el paso de parámetros en funciones: en la declaración de los avisos y los puntos de corte hay que especificar los parámetros con sus tipos. Dependiendo de si se utilizan puntos de corte anónimos o con nombre, la sintaxis varía ligeramente. Véanse los siguientes ejemplos, donde se ven las diferencias en los casos citados:

```
before ( Statement stmt, String query ) :
   call (* Statement.execute*( String )) && target ( stmt ) && args ( query ) {
      checkSQLQueryValidity(query);
   }
```

Este primer ejemplo muestra un aviso que hace uso de un punto de corte anónimo. En la definición de `before` se han de especificar todos los argumentos asociados con la ejecución del método capturado. En él, `target` recoge el objeto en el que se invoca el método capturado cuyo nombre ha de comenzar por `execute`, mientras que `args` captura los argumentos de dicho método. La parte del `advice` antes de los dos puntos especifica el tipo y nombre de cada uno de los argumentos que se capturan y que luego en el cuerpo pueden usarse referenciándolos con dichos nombres.

Cuando se usan puntos de corte con nombres, dichos puntos de corte deben recoger la información contextual y pasársela al aviso. El siguiente ejemplo es similar al anterior salvo que usa puntos de corte con nombre.

```
pointcut sqlQueryChecker( Statement stmt, String query ) :
   call (* Statement.execute*( String )) && target ( stmt ) && args ( query );


before (Statement stmt, String query ) :
   sqlQueryChecker (stmt, query ) {
        checkSQLQueryValidity(query);
   }
```

Funcionalmente el código es idéntico al anterior. El punto de corte `sqlQueryChecker`, además de definir los puntos de enlace, captura el contexto para que pueda ser usado en el aviso. Al igual que antes, se recoge el objeto sobre el que se invoca el método y sus argumentos. El mismo punto de corte declara el tipo y nombre de cada elemento que se recoge, tal como se declara en un método normal. La primera parte del aviso es igual que

en el caso anterior, no varía. Y a continuación se usa el punto de corte que se ha definido más arriba, haciendo coincidir los nombre de los argumentos, pero sin necesidad de indicar los tipos, pues ya se ha hecho. Otro ejemplo de paso de información contextual es el que se muestra a continuación:

```
after() returning(Connection conn ) :
   call(Connection DriverManager.getConnection(..)) {
      System.out.println("Obtenida la conexión: "+ conn );
   }
```

En él, el aviso `after returning` captura el valor devuelto por el método `getConnection` para poder ser utilizado en el cuerpo del aviso. Para ello se ha especificado el tipo y el nombre del valor devuelto en la parte `returning()` del aviso. De esta forma puede usarse el valor devuelto tal como cualquier otro objeto del contexto. Igualmente puede hacerse con el aviso `after throwing` para capturar la excepción lanzada, tal como se muestra en el siguiente ejemplo:

```
after() throwing (RemoteException ex ):
  call(* *.*(..) throws RemoteException) {
     System.out.println("Lanzada " + ex+ " al ejecutar "+ thisJointPoint );
  }
```

Haciendo uso de la API de AspectJ puede también accederse a la información contextual. Así se tienen las funciones `getThis()`, `getTarget()` y `getArgs()` que acceden a la misma información explicada antes. Sin embargo existe una diferencia: usar parámetros en la definición de los puntos de corte, como hemos visto primero, proporciona comprobación de tipos en tiempo de compilación, así como la seguridad de que tales parámetros existen. Por ejemplo, al usar el punto de corte `target(TipoX)` aseguramos que sólo se han llamado a funciones sobre un objeto de tipo `TipoX` y no de otro. Mientras que si usamos dentro del aviso la función `JoinPoint.getTarget()`, no podremos asegurar nada sobre el tipo del objeto que vamos a obtener y, ni siquiera, de si existe o no.

Otro aspecto a tener en cuenta es la devolución de valores desde los avisos `around`. Frecuentemente el valor que se devuelve se declara del mismo tipo del valor devuelto en los puntos de enlaces a los que afecta el aviso. La llamada a `proceed()` devuelve el valor retornado por el método en el punto de enlace. A no ser que se necesite manipular el valor devuelto, el aviso `around` simplemente devuelve el valor que devuelva `proceed()` –si no se invocara este método, habría que devolver un valor apropiado para la lógica del aviso. Se puede dar el caso de que el conjunto de puntos de enlaces al que afecta un `advice` tengan diferentes valores de retorno. En tal caso la solución es definir el valor devuelto por el `advice` como `Object`.

4.4 REGLAS DE ENTRELAZADO ESTÁTICO

En POA a menudo se necesita no sólo alterar el comportamiento dinámico de la aplicación, sino también modificar la estructura estática. Es lo que se conoce como entrelazado estático, frente a lo que hemos visto hasta ahora que se correspondía con el entrelazado dinámico. Hay cuatro maneras en las que AspectJ implementa lo que la terminología inglesa define como *static crosscutting*. Estas son: introducciones, modificaciones de la estructura jerárquica, errores y avisos en tiempo de compilación y suavizado de excepciones. Veamos a continuación cada una de ellas.

INTRODUCCIONES

Los aspectos a menudo necesitan introducir atributos y métodos en las clases a las que afectan. AspectJ dispone de un mecanismo llamado *introduction* para introducir dichos miembros en las clases e interfaces que se especifiquen, de forma que no sea necesario modificar las clases afectadas. Tales miembros pasan a ser parte de las clases entrelazadas y pueden tratarse como tal, con las condiciones de visibilidad que se especifiquen. La sintaxis para las introducciones es como se muestra a continuación:

```
Modifiers Type TypePattern.Id(Formals) { Body }
abstract Modifiers Type TypePattern.Id(Formals);
```

El efecto que tiene tal introducción es hacer que todos los tipos indicados en `TypePattern` dispongan de un nuevo método de nombre `Id` y cuerpo el especificado entre llaves. Igualmente se pueden introducir constructores de la siguiente forma:

```
Modifiers TypePattern.new(Formals) { Body }
```

A diferencia del ejemplo anterior, en este caso no está permitido introducir constructores en interfaces. Así que si `TypePattern` incluye una interfaz, se producirá un error de compilación. De forma similar, se pueden introducir atributos:

```
Modifiers Type TypePattern.Id = Expression;
Modifiers Type TypePattern.Id;
```

En una declaración se pueden introducir varios elementos en diferentes clases a la vez. En el siguiente ejemplo, se introducen tres métodos en tres clases diferentes, `Point`, `Line` y `Square`.

```
public String (Point || Line || Square).getName() { return name; }
```

Los tres métodos tienen el mismo nombre (`getName`), no tienen parámetros y devuelven un `String`. Los tres tendrán además el mismo cuerpo, lo cual es la gran ventaja de esta técnica, pues facilita la adaptación del código.

En cuanto a la visibilidad, AspectJ permite introducciones de miembros públicos, privados o con visibilidad de paquete (esto es, *protected*). El uso del modificador `private` implica una visibilidad privada con relación al aspecto, no necesariamente con el destino de la introducción. Así si un aspecto hace una introducción privada de un atributo en una clase, como podría ser, `private int A.x`, entonces el código en el aspecto puede hacer referencia al atributo `x`, pero ninguna otro clase puede hacerlo. Similarmente, si un aspecto hace una introducción con el modificador `protected`, tal como, `protected int A.x`, entonces cualquiera desde el paquete del aspecto (que no tiene porqué ser el mismo que el de `A`) podrá acceder al atributo `x`.

Por último comentar que si la palabra reservada `this` aparece en el cuerpo de los nuevos constructores o métodos introducidos, esta hará referencia al destino de `TypePattern` y no al aspecto donde se declara.

MODIFICACIÓN DE LA ESTRUCTURA ESTÁTICA

Muchas veces la implementación de los problemas que afectan de manera horizontal a las aplicaciones requieren actuar sobre un conjunto de clases o interfaces que comparten una base común, de tal forma que los avisos y aspectos trabajen únicamente con la API que ofrece tal tipo base. De tal forma, los avisos y aspectos serán dependientes del tipo base, en lugar de las clases e interfaces específicas de cada aplicación. Por ejemplo, un aspecto de manejo de la caché puede declarar que ciertas clases implementen la interfaz `Cacheable`. Así el aviso en el aspecto podrá trabajar solamente con la interfaz de `Cacheable`. El resultado de tal práctica es el desacoplamiento del aspecto de las clases específicas de la aplicación, haciendo así que los aspectos sean más reusables.

Con AspectJ, se puede modificar la estructura jerárquica de las clases haciendo que extiendan a una nueva clase o que implementen una lista de interfaces (siempre y cuando no se violen las reglas de herencia de Java). La sintaxis para tales declaraciones es la siguiente:

```
declare parents: [ChildTypePattern] implements [InterfaceList];
declare parents: [ChildTypePattern] extends [Class or InterfaceList];
```

Por ejemplo, el siguiente aspecto declara que todas las clases e interfaces del paquete `B` del paquete `A` tienen que implementar la interfaz `C`:

```
aspect MyAspect {
    declare parents : A..B.* implements C;
    //...
}
```

La declaración de superclases debe seguir las reglas jerárquicas que impone Java. Por ejemplo, no se puede declarar que una clase sea el padre de una interfaz. Y de igual forma, no se pueden declarar padres, de tal forma que resulte en herencia múltiple. Todo ello resultaría en errores en tiempo de compilación.

ERRORS Y WARNINGS EN TIEMPO DE COMPILACIÓN

AspectJ dispone de un mecanismo para declarar errores y advertencias en tiempo de compilación basado en los puntos de corte explicados anteriormente. De esta forma pueden implementarse comportamientos similares a los que se obtienen con las directivas de preprocesamiento `#error` y `#warning` que soportan algunos preprocesadores C y C++.

El constructor declare error permite declarar errores de compilación cuando el compilador detecta la presencia de cierto punto de enlace que se haya especificado con un punto de corte. En tal caso, el compilador informa del error mediante el mensaje proporcionado por el programador y aborta el proceso de compilación. La sintaxis del constructor es como se muestra a continuación:

```
declare error : <pointcut> : <message>;
```

De igual forma se pueden declarar advertencias mediante el siguiente constructor:

```
declare warning : <pointcut> : <message>;
```

Al afectar estas declaraciones al comportamiento en tiempo de compilación es evidente que no se pueden usar puntos de corte basados en el contexto como `this()`, `target()`, `args()`, `if()`, `cflow()`, y `cflowbelow()`.

Un uso típico de estos constructores es para definir reglas que eviten el uso de ciertos métodos no soportados o bien para limitar la localización de determinados usos del código. Un ejemplo sacado de la aplicación que sirve como ejemplo para este proyecto es el siguiente. A efectos de evitar la dispersión del código de base de datos, si limita el uso de éste al paquete `bd` y se permite, aunque avisando con un `warning`, en el paquete `aspects`.

```
pointcut dbCode(): call(Connection DriverManager.getConnection(..));
pointcut notRecommendedDbCode(): dbCode() && !within(db.*);
pointcut forbidedDbCode(): notRecommendedDbCode() && !within(aspects.*);

declare warning: notRecommendedDbCode(): "Codigo no recomendado!";
declare error: forbidedDbCode(): "Código no permitido!";
```

SUAVIZADO DE EXCEPCIONES

Es un dato estadístico el hecho de que en las aplicaciones Java aproximadamente el treinta por cierto del código corresponde al tratamiento de errores. El uso en las mismas de los mecanismos de lanzamiento de excepciones y su captura es una solución al problema, pero sin duda uno de los síntomas más evidentes de enredamiento del código. En AspectJ un aspecto puede especificar que cuando en los puntos de enlace indicados se lance cierta excepción, ésta debe saltarse el sistema usual de tratamiento de excepciones de Java y lanzarse como del tipo `org.aspectj.lang.SoftException`, que es subtipo de `RuntimeException` y que por lo tanto no necesita ser declarada. Se dice que la excepción ha sido suavizada, del inglés *softened*. La sintaxis para especificar tal comportamiento en AspectJ es la siguiente:

```
declare soft: TypePattern: Pointcut;
```

En el siguiente ejemplo, un aspecto `MyAspect` declara `Exception` en una cláusula `declare soft`, lo cual implica que cualquier excepción que se lance será encapsulada en una excepción del tipo `org.aspectj.lang.SoftException` y relanzada como tal.

```
aspect MyAspect {
  declare soft: Exception: execution(void main(String[] args));
}
```

El efecto conseguido es el mismo que se conseguiría con el siguiente aviso:

```
aspect MyAspect {
   void around() execution(void main(String[] args))
   {
      try {
        proceed();
      }catch (Exception e) {
        throw new org.aspectj.lang.SoftException(e);
      }
   }
}
```

4.5 EXTENSIÓN DE LOS ASPECTOS

Al igual que las clases, los aspectos se pueden extender, de forma que permitan la abstracción y composición de los problemas que implementan. Sin embargo, introducen algunas reglas a la hora de extenderse, tal como son:

- Un aspecto, abstracto o concreto, puede extender una clase e implementar un conjunto de interfaces.
- Las clases no pueden extender aspectos. Intentar que una clase extienda o implemente un aspecto resultará en un error de compilación.
- Aspectos que extienden aspectos. Los aspectos pueden extender otros aspectos, con lo cual no sólo se heredan los atributos y métodos, sino también los puntos de corte. Sin embargo, los aspectos sólo pueden extender aspectos abstractos. Intentar que un aspecto extienda un aspecto concreto resultará en un error de compilación.

Por ejemplo, algo usual es implementar los sistemas de log como aspectos. Así si queremos mantener un registro o historial de ciertas acciones, se puede definir un aspecto abstracto con un punto de corte abstracto, donde se indique cómo se quiere escribir la información, pero no cuales van a ser los puntos de enlace. Los aspectos que extiendan dicho aspecto, son los que concretan los puntos de corte, lo que da la posibilidad de definir clara y separadamente los elementos que se quieren controlar. El siguiente fragmento de código muestra el aspecto abstracto:

```
abstract public aspect Logger {
  abstract pointcut logPoint();

  before(): logPoint() && !within(Logger+){
    System.log(thisJoinPoint.getTarget()+"/"+thisJoinPoint.getThis());
  }
}
```

Del punto de corte se excluyen el propio aspecto y sus subclases para evitar la recursividad. El aviso recoge el objeto sobre el que se llaman las funciones y el objeto actual y los escribe en el log del sistema. El siguiente fragmento muestra un aspecto concreto que extiende a este aspecto:

```
public aspect LogDataBaseExecutions extends Logger{
  pointcut logPoint(): call(* DataBase.*.*(..));
}
```

Este aspecto proporciona una definición del punto de corte `logPoint`, que en este caso son las todas las llamadas a funciones de un supuesto paquete de base de datos `DataBase`.

4.6 INSTANCIAR ASPECTOS

A diferencia de las clases, los aspectos no se pueden instanciar con expresiones `new`. En lugar de eso, las instancias de los aspectos se crean automáticamente. Debido a que los `advices` sólo se ejecutan en el contexto de una instancia de un aspecto, el cómo se instancian los aspectos controla indirectamente cuando se ejecutan

los avisos. El criterio utilizado para determinar cómo se instancia un aspecto se hereda del aspecto al que extiende. Si éste no existiera, entonces por defecto, los aspectos siguen una política *singleton* (es decir, sólo existe una instancia de ellos). Los siguientes tres subapartados indican las diferentes posibilidades a la hora de instanciar aspectos y la sintaxis para cada una de ellas. Para ilustrarlo en cada apartado se ejecuta un ejemplo con las siguientes clases bases:

```
public class A {
  public static void main (String args[]){
      System.out.println("This     Target     Aspect");
      for (int i=0; i<3; i++)
         new B().test();
  }
}
```

La clase `A` contiene el método `main` y su cometido es crear tres instancias de la clase `B` y llamar al método `test` de éste. Primero crea una cabecera para los aspectos que se verán después. A continuación se muestra la clase `B`:

```
public class B {
  public static B b = new B();

  public void test(){
     b.metodo();
     new B().metodo();
  }

  public void metodo(){}

}
```

La clase `B` tampoco hace nada útil. Tiene un atributo estático `b` de la misma clase y dos métodos, `test` y `metodo`. El segundo de ellos será el que se capture en el punto de corte que definiremos a continuación, del que como se puede ver se producirán seis llamadas, tres de ellas con el objeto estático de la clase `B` y las otras tres con nuevos objetos creados en cada llamada a `test()`.

ASPECTOS SINGLETON

```
aspect Id { ... }
aspect Id issingleton { ... }
```

Por defecto, o usando el modificador `issingleton`, los aspectos tienen exactamente una instancia que se ejecuta para todo el programa. Dicha instancia está disponible en cualquier momento durante la ejecución del programa usando el método estático `aspectOf()` definido en los aspectos. Esta instancia se crea tal como se carga el fichero que contiene el *class* del aspecto y vive durante toda la ejecución del programa, por lo que sus avisos podrán ejecutarse en cualquier punto de enlace. Véase el siguiente ejemplo:

```
public aspect Singleton issingleton{

  pointcut m(): call (* *.metodo());

  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(Singleton.aspectOf());
  }
}
```

En este aspecto se capturan las llamadas a `metodo` y antes de ellas se imprime información del objeto actual, del objeto sobre el que se realiza la llamada y de la instancia del aspecto. Al ejecutarlo con las clases anteriores se obtiene la siguiente salida:

```
This       Target     Aspect
B@cf2c80   B@729854   Singleton@6eb38a
B@cf2c80   B@cd2e5f   Singleton@6eb38a
B@9f953d   B@729854   Singleton@6eb38a
B@9f953d   B@fee6fc   Singleton@6eb38a
B@eed786   B@729854   Singleton@6eb38a
B@eed786   B@87aeca   Singleton@6eb38a
```

Cada código diferente representa un objeto diferente. Como se ve en la primera columna, hay tres objetos, que son los tres creados por la clase `A` (de códigos `B@cf2c80`, `B@9f953d`, `B@eed786`). El método se llamó sobre cuatro objetos `B` diferentes, de los cuales el que se repite (de código `B@729854`) es el estático. Los otros corresponden a cada una de las nuevas instancias que se crean en el método `test` de la clase `B`. Y dado que se ha usado *singleton*, en todas las llamadas que capturó el punto de corte `m()` se ha usado la misma y, la única, instancia del aspecto que existía (de código `Singleton@6eb38a`). El mismo ejemplo se usará en los siguientes apartados para poder ver las diferencias.

ASPECTOS PER-OBJECT

```
aspect Id perthis(Pointcut) { ... }
aspect Id pertarget(Pointcut) { ... }
```

Si un aspecto A se define `perthis(Pointcut)` entonces existirá una instancia diferente del aspecto A cada vez que el objeto que se está ejecutando en los puntos de enlace definidos en `Pointcut` (esto es, el referenciado por `this`) sea diferente. Así, los avisos definidos en A pueden ejecutarse en cualquier punto de enlace donde el objeto actual en ejecución haya sido asociado con una instancia de A. Si no se ha creado aún una instancia de A para el objeto en cuestión, se crea una nueva. Tomando el ejemplo anterior, cambiando únicamente la política de instanciación del aspecto, tenemos el siguiente código:

```
public aspect PerThis perthis(m()){

  pointcut m(): call (* *.metodo());

  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(PerThis.aspectOf(thisJoinPoint.getThis()));
  }
}
```

En comparación con el código anterior, hay que observar la manera de obtener la instancia del aspecto. Al no haber una única instancia, no puede usarse como antes el método `aspectOf`, sino otro que tome como parámetro el objeto actual y devuelva, si es que existe, el aspecto que tiene asociado. Si no existiera tal objeto asociado se lanzaría la excepción `org.aspectj.lang.NoAspectBoundException`.

Al ejecutar el ejemplo se obtiene la salida que se muestra a continuación. En ella se ve como se tiene un aspecto diferente para cada objeto actual que está en ejecución. Hay tres instancias del objeto `test` (de códigos `B@6eb38a`, `B@eed786` y `B@2dacd1`) y para cada una de ellas se utiliza un aspecto diferente (de códigos `PerThis@9f953d` , `PerThis@87aeca` y `PerThis@ad086a` respectivamente).

```
This       Target     Aspect
B@6eb38a   B@cd2e5f   PerThis@9f953d
B@6eb38a   B@fee6fc   PerThis@9f953d
B@eed786   B@cd2e5f   PerThis@87aeca
B@eed786   B@e48e1b   PerThis@87aeca
B@2dacd1   B@cd2e5f   PerThis@ad086a
B@2dacd1   B@385c1    PerThis@ad086a
```

De manera similar, si un aspecto A es definido `pertarget(Pointcut)` entonces existirá un objeto de tipo A por cada objeto destino diferente (estos son, los referenciados por `target`) sobre el que se ejecuten los métodos capturados en los puntos de enlace definidos en el `Pointcut`. Así, los avisos definidos en A pueden ejecutarse en cualquier punto de enlace donde el objeto destino haya sido asociado con una instancia de A. O bien, si un objeto destino no tiene aún un aspecto asociado, se crea una nueva instancia de A, que será la que en adelante utilizará. Volviendo al ejemplo, tenemos esta vez el siguiente código:

```
public aspect PerTarget pertarget(m()){
  pointcut m(): call (* *.metodo());
  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
```

```
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(PerTarget.aspectOf(thisJoinPoint.getTarget()));
  }
}
```

Al igual que en el caso anterior, para obtener la instancia del aspecto es preciso indicar el objeto asociado, que, al ser *pertarget*, habrá de ser el objeto destino que se obtiene con `getTarget()`. En la salida que se obtiene al ejecutar el ejemplo se comprueba que se han creado tantos aspectos como objetos destino diferentes. A los objetos de códigos `B@cd2e5f` (este es el estático), `B@fee6fc`, `B@e48e1b` y `B@385c1` le corresponde los aspectos `PerTarget@9f953d`, `PerTarget@eed786`, `PerTarget@2dacd1` y `PerTarget@42719c`, respectivamente.

```
This       Target     Aspect
B@6eb38a   B@cd2e5f   PerTarget@9f953d
B@6eb38a   B@fee6fc   PerTarget@eed786
B@87aeca   B@cd2e5f   PerTarget@9f953d
B@87aeca   B@e48e1b   PerTarget@2dacd1
B@ad086a   B@cd2e5f   PerTarget@9f953d
B@ad086a   B@385c1    PerTarget@42719c
```

Al usar alguna de las dos técnicas anteriores (`perthis` o `pertarget`) hay que tener en cuenta la siguiente restricción a la hora de escribir el código: en los aspectos así definidos, en un punto de enlace donde el código fuente no esté disponible para el objeto actual, no se instanciará el aspecto definido `perthis` o `pertarget` para dicho punto de enlace. De tal modo que, los aspectos definidos `perthis(Object)/ pertarget(Object)` no crearán instancias del aspecto para cada objeto, sino sólo para aquellos para los que el compilador controla su código *class*.

ASPECTOS PER-CONTROL-FLOW

```
aspect Id percflow(Pointcut) { ... }
aspect Id percflowbelow(Pointcut) { ... }
```

Si un aspecto A se define `percflow(Pointcut)` o `percflowbelow(Pointcut)` entonces un objeto de tipo A se crea cada vez que el control pasa a uno de los puntos de enlace capturados por el `Pointcut`, bien tal como el flujo de control pasa al punto de enlace o a partir de éste, pero excluyéndolo, respectivamente. Véase el anterior ejemplo con una política de `PerCflow` y los resultados:

```
public aspect PerCflow percflow(m()){
  pointcut m(): call (* *.metodo());
  before(): m(){
     System.out.print(thisJoinPoint.getThis()+"   ");
     System.out.print(thisJoinPoint.getTarget()+"   ");
     System.out.println(PerCflow.aspectOf());
  }
}
```

En este caso el método `aspectOf()` no necesita ningún parámetro. AspectJ mantiene una pila de aspectos *percflow*, que se crean al entrar al llegar el flujo de control al punto de enlace y se destruyen al salir de él, por lo que se asegura que siempre se mantienen el número de instancias correctas. Durante la ejecución, los aspectos se toman de esta pila.

En el resultado que se muestra a continuación puede verse como se ha creado una instancia diferente del aspecto cada vez que se ha realizado una llamada y puesto que se realizaron seis, se tienen seis en total, indiferentes del objeto actual o del destino.

```
This       Target     Aspect
B@cd2e5f   B@9f953d   PerCflow@fee6fc
B@cd2e5f   B@eed786   PerCflow@87aeca
B@e48e1b   B@9f953d   PerCflow@2dacd1
B@e48e1b   B@ad086a   PerCflow@385c1
B@42719c   B@9f953d   PerCflow@30c221
B@42719c   B@19298d   PerCflow@f72617
```

Como en los ejemplos anteriores hay tres instancias del objeto actual `B` y cuatro como `target` (al contar con el objeto estático). Sin embargo, esta vez, las instancias del aspecto son diferentes para cada una de las llamadas.

ASPECT PRIVILEGE

```
privileged aspect Id { ... }
```

El código escrito en los aspectos está sujeto a las mismas reglas de control de acceso que el código Java en lo que respecta a los miembros de las clases o los aspectos. Así, por ejemplo, el código escrito en un aspecto no puede hacer referencia a miembros con visibilidad *protected*, a no ser que el aspecto esté definido en el mismo paquete. Mientras que estas restricciones son aceptables para muchos aspectos, puede haber casos en los que los aspectos necesiten en sus *advices* o *introductions* acceder a recursos que son privados o protegidos. Para permitir esto, los aspectos tienen que ser declarados `privileged`. En el código de los miembros `privileged` se tiene acceso a todos los miembros, incluidos los privados. En el siguiente ejemplo se muestra un ejemplo de un aspecto que gracias a ser declarado como *privileged* puede acceder a un atributo privado de la clase que utiliza. La clase `Divisor` contiene un atributo privado y un método que realiza la división dado un entero como parámetro. El aspecto chequea que el divisor no sea cero.

```
class Divisor {
   private int i;
   int divide(int x) { return x/i; }
}


Divisor(int value){i = value;}

privileged aspect C {
   before(Divisor d): call(int Divisor.divide(int)) && target(d) {
      if (d.i == 0) throw new RuntimeException();
   }
}
```

Si C no hubiese sido declarado `privileged`, el intento de referenciar a `d.i` hubiera implicado un error de compilación como el siguiente:

```
C.java:4:13: Divisor.i has private access
     if (d.i == 0) throw new RuntimeException();
```

4.7 ASPECTOS DOMINANTES

```
aspect Id dominates TypePattern { ... }
```

Un aspecto puede declarar que los *advices* que define dominan sobre los *advices* en otros aspectos, lo que implica que el aviso dominante tiene mayor precedencia que los especificados en los aspectos que coincidan con `TypePattern`. Por defecto en AspectJ no existe ningún orden definido, por lo que, si se precisa ejecutar los avisos en determinado orden, es necesario especificarlo con la cláusula `dominates`. La semántica es que si un aspecto `A` domina sobre un aspecto `B`, entonces los avisos de `A` tienen más prioridad y se ejecutan antes que los de `B`. En cuanto a las situaciones de conflicto como puede ser la siguiente, el compilador 1.0.6 no informa del error y simplemente actúa ignorando una de las dos.

```
aspect A dominates B{…}
aspect B dominates A{…}
```

Hay que advertir que en la versión 1.1 de AspectJ la sintaxis de `dominates` ha cambiado, desapareciendo `dominates` y pasando a usarse la siguiente declaración:

```
declare precedence ":"  TypePatternList ";"
```

Por poner un ejemplo, si quisiéramos definir una regla de precedencia, de modo que los avisos de los aspectos cuyo nombre contenga la cadena `Security` precedan a cualquier otro y que además, los avisos del aspecto `Logging` y sus subclases precedan a los restantes, entonces deberíamos escribir lo siguiente:

```
declare precedence: *..*Security*, Logging+, *;
```

Sería un error en una sentencia `precedence` que un aspecto pudiera ser referido por más de un patrón de tipos. Así por ejemplo, la siguiente declaración produciría un error:

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

Sin embargo no es un error tener en múltiples sentencias este tipo de problema de circularidad, como puede producirse con las siguientes dos declaraciones:

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

Y de hecho un sistema así definido puede ser correcto, siempre y cuando los avisos de `A` y `B` no compartan puntos de enlace. De esa forma queda claramente expresado que `A` y `B` son independientes.

Esta técnica se ha utilizado en la aplicación que sirve de ejemplo a este proyecto en los aspectos `Pooling` y `ConnectionChecking`, donde el primero de ellos es responsable de mantener un conjunto de conexiones a la base de datos. En caso de hacerse una llamada en la aplicación para crear una nueva conexión, ésta es capturada por el aspecto, el cual le proporciona una conexión de entre las disponibles (si es que las hay). Debido a que el estado de las conexiones almacenadas puede no ser correcto, es necesario, antes de servir estas conexiones, comprobar su validez. De eso se encarga el otro aspecto, `ConnectionChecking`, que ejecuta sus avisos, necesariamente, antes que los de `Pooling`. A continuación se presenta el código de estos dos aspectos (de forma abreviada, pues el código completo puede verse en los ficheros que acompañan a este proyecto):

Código ConnectionChecking.java

```
public aspect Pooling
{

  private static Stack pool = new Stack();

  pointcut poolGet(): call(static Connection DriverManager.getConnection(..));
  pointcut poolPut(): call(void Connection.close());


  Connection around() throws SQLException: poolGet()
  {
    synchronized(pool)
    {
      if( pool.empty() )
         return proceed();
      return (Connection)pool.pop();
    }
  }


  void around(): poolPut()
  {
    Connection conn = (Connection)thisJoinPoint.getTarget();
    pool.push(conn);
  }

}
```

Código ConnectionChecking.java

```
public aspect ConnectionChecking dominates Pooling
{

  Connection around() throws SQLException:
     call(static Connection DriverManager.getConnection(..))
  {
      Connection conn;
      do{
        conn = proceed();
      }while(badConnection(conn));

      return conn;
  }

  private boolean badConnection(Connection conn)
  {...}
}
```

4.8 LA API DE ASPECTJ

AspectJ viene con dos paquetes en su API: `org.aspectj.lang` y `org.aspectj.lang.reflect`. Se puede consultar la documentación completa en la distribución, por lo que aquí solo se comentarán de forma breve las partes que se usan más frecuentemente.

Interfaces JoinPoint y JointPointStaticPart
El compilador de AspectJ crea dos variables especiales para cada punto de enlace que se encuentra: `thisJoinPoint` y `thisJoinPointStaticPart`. Estas variables tienen los tipos que implementan `org.aspectj.lang.JoinPoint` y `org.aspectj.lang.JoinPoint.StaticPart`, respectivamente. La diferencia entre ambas, es que esta última contiene información que no depende del contexto en ejecución. Así podemos tener información sobre los puntos de enlace, como la signatura de los métodos, la localización del código, el tipo de punto de enlace (method-execution, call-execution, ...) Y respecto a la información contextual en tiempo de ejecución, podemos tener el objeto actual que se esté ejecutando, el objeto sobre el que se llaman los métodos o los argumentos disponibles en el punto de enlace.

Interface Signature y sus subinterfaces
La interfaz `org.aspectj.lang.Signature` declara métodos para describir los constructores del lenguaje en los puntos de enlace (métodos, atributos, constructores, excepciones...) Cada una de las subinterfaces añade métodos relevantes a cada tipo de signatura particular. Se tienen métodos para obtener los nombres de los métodos o los atributos, los nombres y los tipos de los parámetros de los métodos capturados, los tipos de las excepciones que pueden ser lanzadas, el tipo del valor devuelto por los métodos o los advices...

Interface SourceLocation
La interfaz `org.aspectj.lang.reflect.SourceLocation` ofrece métodos para localizar físicamente la posición de los puntos de enlace en el código fuente. Dispone de métodos para obtener el nombre del fichero que contiene el punto de enlace, así como la línea y la columna en la que está.

Exceptions
La API de AspectJ define tres expcepciones, de nombres: MultipleAspectsBoundException, NoAspectBoundException y SoftException.

5 HERRAMIENTAS DEL LENGUAJE

En los siguientes apartados se presentan las herramientas que acompañan la distribución de AspectJ, a saber, el compilador del lenguaje, la herramienta de documentación y el sistema de navegación por las clases y aspectos. Tan solo se describirán los parámetros más usados, pues para una información completa puede referirse a la documentación de AspectJ.

5.1 COMPILADOR DEL LENGUAJE

El compilador para el lenguaje AspectJ se llama `ajc`. Puede lanzarse desde la línea de comandos o ser invocado desde una aplicación java, ya que el mismo compilador está escrito en Java y puede ejecutarse usando la clase `org.aspectj.tools.ajc.Main` (de este modo se tienen los mismos parámetros que de la manera anterior). El formato general de las llamadas al compilador es el siguiente:

```
ajc [option] [file...| @file... | -arglist file...]
```

La herramienta `ajc` compilará los ficheros listados en la línea de comandos o los ficheros que se encuentre en el fichero indicado. Por comodidad puede usarse un fichero con la lista de opciones y los nombres de los ficheros a compilar, el cual se especifica con el símbolo "`@`" o con la opción `–arglist`. En dicho fichero, cada línea debe contener una sola opción o el nombre de un único fichero. Todos los ficheros que se quieran compilar tienen que ser indicados, debido a que `ajc` no busca en el *class path* las clases que se necesitan. Entre las opciones más usuales que acepta `ajc` están las siguientes:

- **-argfile <file>** Indica un fichero con los argumentos que ha de tomar el compilador.
- **-classpath <path>** Indica dónde encontrar las clases de los ficheros de los usuarios.
- **-d <dir>** Indica donde guardar los .class generados.
- **-encoding <encoding>** Indica la codificación de los caracteres usada en los ficheros.
- **-preprocess** No genera los `.class` si no sólo los `.java`.
- **-workingdir <dir>** Indica donde indicar los ficheros .java preprocesados.
- **-usejavac** Indica al compilador que use `javac` para general los ficheros `.class`.
- **-verbose** Muestra mensajes por pantalla de lo que esté haciendo el compilador.
- **-version** Muestra la versión del compilador.

5.2 GENERADOR DE DOCUMENTACIÓN

La herramienta `ajdoc` genera documentación en formato HTML acerca de la API. Está basada en la herramienta `javadoc` y acepta todos los mismos parámetros estándar que ésta acepta. Pero además de documentar las clases, `ajdoc` documenta los puntos de corte, los avisos y las declaraciones; en los *advices* e *introductions* aparecen enlaces a las clases afectadas; enlaza los miembros introducidos con las clases que los introducen; y los métodos afectados por avisos con los correspondientes avisos en los aspectos. Al igual que `ajc`, `ajdoc` necesita que, bien por la línea de comandos o mediante un fichero, se listen todos los ficheros fuentes que tiene que documentar. Sin embargo, a `ajdoc`, pueden indicársele nombres de paquetes. La forma general de usar `javadoc` es como sigue:

```
ajdoc [options] [packagenames] [sourcefiles] [@files]
```

-public Documenta solo las clases y los miembros públicos.
-protected Documenta las clases y miembros protegidos y públicos.
-package Documenta las clases y miembros protegidos, públicos y con visibilidad de paquete.
-private Documenta todas las clases y miembros.
-help Muestras las opciones.
-sourcepath <pathlist> Indica donde encontrar los ficheros que contienen el código fuente.
-classpath <pathlist> Indica dónde encontrar los ficheros *class* de usuario.
-d <dir> Indica el directorio destino de los ficheros de salida.
-verbose Muestra mensajes de lo que `javadoc` está haciendo mientras se ejecuta.
-version Imprime la versión de `javadoc`.

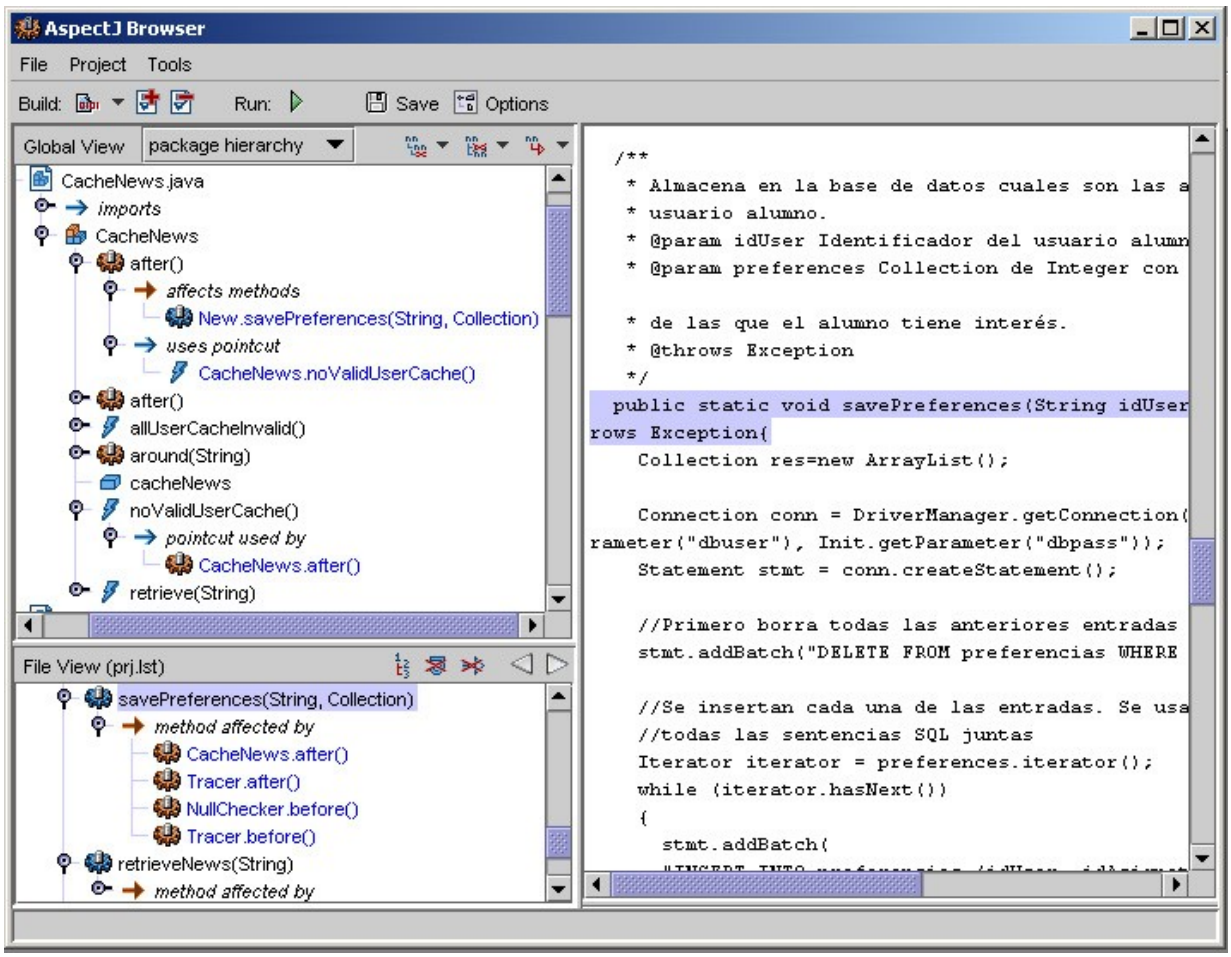
5.3 NAVEGADOR DE ASPECTJ

Otra de las herramientas que nos encontramos en el directorio `/bin` de AspectJ es `ajBrowser`. Esta ofrece un GUI (graphic user interface) para compilar programas con `ajc` y navegar por la estructura de la aplicación. Además permite editar el código de los ficheros y después de compilar, ejecutar el programa.

Para ejecutarlo, bien se escribe desde la línea de comandos `ajbrowser` (para invocar el *script* del subdirectorio `/bin`), o bien se llama usando `aspectjtools.jar`. No es preciso indicar ningún parámetro, pero pueden indicarse ficheros de configuración con las opciones y ficheros que se desean cargar (de extensión `.lst`). Haciendo uso de `aspectjtools.jar` un ejemplo sería:

```
java -jar <directorio de AspectJ>/lib/aspectjtools.jar <mi aplicación>/list.lst
```

De momento AjBrowser tiene soporte para Eclipse, JBuilder, SunONE/Forte y Emacs, a parte de poder usarse independientemente. En la Pantalla 1 se muestra una captura de `ajBrowser`, donde se han cargado los ficheros de la aplicación que sirve de ejemplo al proyecto.



Pantalla 1 AspectJ Browser

Para navegar por la estructura, se muestra a la izquierda un listado de las clases y aspectos cargados. En la parte superior, de cada aspecto se muestran los puntos de corte que define, así como los avisos que los utilizan y, éstos, a los métodos que afectan. En el cuadro superior izquierda, se ve como un aspecto de nombre `CacheNews`, define un *pointcut* `noValidUserCache()`, que es usado por un aviso `after` que afecta al método `savePreferences(String, Collection)` de la clase `New`. Abajo, puede verse desde la perspectiva contraria. Navegando por la clase `New` y el método `savePreferences`, podemos ver todos los aspectos que le afectan. Entre ellos, el primero que aparece es `CacheNews.after()`, que es el comentado antes, pero además se puede ver que hay otros tres.

Esto concluye el apartado referente a Aspect/J. El siguiente de los apartados establece una comparación entre los dos lenguajes de aspectos de propósito general que hemos visto, reflejando de manera clara sus diferencias conceptuales y sus objetivos.

6 ASPECTJ VS. HYPER/J

En los apartados anteriores se han analizado dos de los lenguajes que más éxito están teniendo en el paradigma de la POA y que, como se ha podido ver, no enfrentan el problema de la misma forma. El objetivo de la POA es identificar y separar conceptos no funcionales que afectan horizontalmente a la descomposición dominante de una aplicación.

Para conseguir este objetivo AspectJ extiende Java aumentando el modelo OO definiendo nuevos constructores que permiten la definición de aspectos capaces de encapsular en módulos separados conceptos dispersos por la estructura de clases, así como constructores para definir los puntos de enlace donde las clases o métodos han de ser modificados. Cada aspecto puede afectar a varias clases, lo que permite localizar de manera efectiva la implementación de los aspectos. El principal inconveniente hasta ahora es la dependencia de los aspectos con las clases a las que afectan, pues, por lo general, sin hacer referencia a éstas no se entienden fácilmente. La idea de Hyper/J es soportar la integración de módulos, recogidos en lo que en la terminología se denominan *hyperslices*, donde cada una de ellos encapsula una competencia específica. Los *hyperslices* pueden solaparse o existir de manera independiente unos de los otros. En lugar de definir puntos de enlace, existen reglas de composición para componer los *hyperslices*, de modo que, siguiendo una regla general de composición establecida en el lenguaje y las reglas específicas particulares para cada caso especial, combinan sus funcionalidades. Así que tenemos que, mientras que AspectJ es una extensión al lenguaje Java, Hyper/J es una herramienta que utiliza java como lenguaje.

Empezando con las diferencias, tenemos las reglas de composición. En AspectJ no hay como en Hyper/J una regla de composición general. En su lugar, cada aspecto contiene sus reglas, indicando particularmente la manera en la que dicho aspecto debe enlazarse con las clases bases. A diferencia, esto se hace en Hyper/J en ficheros separados –como hemos visto– lo cual permite de veras combinar aplicaciones que se escribieron sin tener en mente la POA. En el caso de AspectJ podemos decir que, habiendo conseguido encapsular aspectos, no encapsula las reglas de entrelazado, que son el nuevo *tangled code* (código disperso), problema que no tenemos usando Hyper/J.

La principal diferencia conceptual entre ambos parte de la existencia en AspectJ de un programa “base” en el cual los aspectos se enlazan. En Hyper/J no existe el concepto de un programa base, sino que cada componente de código es independiente y provee de forma completa el código para la unidad particular de descomposición. Esa es la principal ventaja de Hyper/J, y es el hecho de utilizar componentes verdaderamente independientes entre sí, lo cual beneficia la reusabilidad de los mismos (incluidas dos aplicaciones que se diseñaron por separado y que son capaces de funcionar independientemente).

Otra diferencia es que AspectJ trabaja con el código fuente, mientras que Hyper/J utiliza el bytecode de los ficheros class. Por tanto no necesitamos el código fuente, aunque sí conocer la interfaz (nombres de las clases, funciones...), lo cual se puede obtener fácilmente con compiladores inversos o con la documentación. Un problema de Hyper/J hoy en día es que es aún una herramienta en desarrollo, mientras que AspectJ tiene ya disponible su versión 1.1.

AspectJ es una herramienta ideal para incluir cierto comportamiento en muchos sitios diferentes de una aplicación. Sin embargo, su uso se complica cuando se trata de combinar grandes componentes con muchas clases. Habría que distinguir entre el código base y los muchos aspectos que identificáramos y entonces combinarlos. En el caso de tener muchos aspectos, si quisiéramos realizar distintas combinaciones y probarlas repetidas veces, entonces esto se convierte en un proceso muy laborioso. Y se debe a que en AspectJ cada aspecto es una entidad distinta y separada, y con muchos aspectos, se pierde cohesión. Por el contrario, en Hyper/J se tiene el concepto de hipersección. Un hyperslice contiene todas las clases y métodos asociados con una competencia específica. Por lo tanto añadir o quitar una de estas competencias implica un solo movimiento, respectivamente añadir o quitar la hipersección concreta.

De cualquier modo, el modelo propuesto por AspectJ ha tenido un mayor éxito entre la comunidad informática. Puede encontrarse mucha más información y empresas que lo están utilizando para sus proyectos. Aparecen

nuevos lenguajes cuyos modelos se basan en el de AspectJ, como es el caso de AspectC#, AspectC++ o AspectS, entre otros. Las nuevas versiones por venir y una mayor documentación pueden hacer ver un auge en Hyper/J.

7 APÉNDICE (EJEMPLO CON UN PAQUETE MATEMÁTICO)

Supongamos un pequeño paquete matemático que nos permite implementar sumatorios y potencias. Con objeto de que sea más legible, no se contemplan los casos particulares en las potencias, como ocurre cuando el cero o el uno intervienen. Cada tipo de operación está representada con una clase y cada una de ellas implementa un método `calcula` que es el encargado de realizar los cálculos.

Implementación del ejemplo

Código 1 Potencia.java

```
package math;

public class Potencia{

    int base;
    int exponente;

    public int getBase(){
        return base;
    }

    public int getExponente(){
        return exponente;
    }

    public void setBase(int dat){
        this.base=dat;
    }

    public void setExponente(int dat){
        this.exponente=dat;
    }

    public int calcula(){
        int result=0;
        for (int i=0; i<exponente; i++)
            result=result+base*base;

        return result;
    }
}
```

Código 2 Sumatorio.java

```
package math;

public class Sumatorio{

    int iteraciones;
    int sumando;

    public Sumatorio(int iteraciones, int
sumando){
        this.iteraciones = iteraciones;
        this.sumando = sumando;
    }

    private int getSumando(){
        return sumando;}

    private int getIteraciones(){
        return iteraciones;
    }

    public int calcula(){
        int result=getSumando();

        for (int i=0;
i<this.getIteraciones()-1; i++)

result=result+this.getSumando();

        return result;
    }
}
```

Para probar esta sencilla librería podemos utilizar el siguiente código de ejemplo contenido en un fichero `main.java`. En el se crean dos objetos, uno de cada una de las clases y después se llama a sus respectivos métodos `calcula()`:

Código 3 main.java

```
import math.*;

public class main{

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

   Potencia pot = new Potencia();
   pot.setBase(2);
   pot.setExponente(4);
```

```
   System.out.println(pot.getBase()+ " elevado a " + pot.getExponente()+" es");
   System.out.println(pot.calcula());

   Sumatorio sum = new Sumatorio(2,3);

   System.out.println("2 sumado 3 veces es"+ sum.calcula());
  }
}
```

Supongamos que estas operaciones para el cálculo del sumatorio y las potencias son altamente costosas y que decidimos implementar una política por la que los cálculos se lleven a cabo sólo si el procesador tiene el tiempo suficiente. En lugar de tener que modificar cada una de las clases, podemos crear un aspecto que implemente este nuevo requisito, de modo que no sea necesario cambiar, ni complicar el código ya escrito. Además, para el matemático que ha creado las clases anteriores, ¿qué tiene que ver con él ahora este nuevo comportamiento?. Sin duda, la solución que nos ofrece la POA es de utilidad en estos casos. Veamos cómo se haría:

Código 4 CheckTimeToProcess.java [1]

```
public aspect CheckTimeToProcess{

     pointcut processPoint(): execution(int math.*.calcula(..));

     int around(): processPoint()
     {
        while(!timeToProcess()){
           //sleep();
        }

        System.out.println("Comenzando calculos..");
        int result = proceed();
        System.out.println("Calculos terminados!");
        return result;
     }

     public boolean timeToProcess()
    {
        System.out.println("Comprobando tareas.. ");
        return true;
     }
}
```

Lo que se ha hecho es definir un punto de corte para indicar que queremos capturar los puntos donde se llaman a las funciones `calcula` del paquete `math` y un aviso, que utiliza dicho corte, para comprobar si se dan las condiciones para poder realizar los cálculos o no. Antes de ejecutar `calcula`, se chequeará la condición `timeToProcess` (que contendría la lógica para ver la ocupación del sistema) y en caso de ser afirmativa, se procede con el cálculo (eso se consigue con la llamada a `proceed`).

Se necesita establecer un acuerdo con el lenguaje base a fin de facilitar la definición de los puntos de enlace. En este caso se ha decidido que las funciones que realicen el cálculo pesado se llamen `calcula`. De este modo es fácil identificar dichos puntos en el código[2].

Además para fines de depuración queremos mostrar por pantalla todas las ejecuciones de las funciones públicas `get` y `set` de las clases del paquete `math` (para dar un poco de juego, observe que en `Sumatorio`, estos métodos no son públicos, así que no entran dentro del conjunto de puntos de enlace que queremos definir). Para ello escribiremos el siguiente aspecto en un fichero `Logger.java`:

[1] Por simplicidad, la funcionalidad no está totalmente implementada, pero se entiende qué es lo que tendría que hacerse en el cuerpo del bucle while y la operación timeToProcess.

[2] Se podría obligar a las clases a que tuvieran el método calcula, haciendo que las clases implementaran una clase abstracta Operaciones, que tuviera dicha función.

Código 5 Logger.java

```
public aspect Logger{
   pointcut logPoint(): execution(public * math.*.get*(..))
                     || execution(public * math.*.set*(..));

   before(): logPoint(){
       System.out.println(thisJoinPoint);
   }
}
```

Aquí se ha definido un corte `logPoint` que declara los puntos de enlace donde se llaman a las funciones públicas `get` y `set` de cualquiera de las clases del paquete `math`. Observe que para definir los patrones de los nombres de las funciones se puede utilizar el carácter especial asterisco `*` para especificar que el nombre puede terminar en cualquier cadena de caracteres. Además en un punto de corte se pueden encadenar la definición de varios puntos de enlace usando los símbolos `||`. La sintaxis es mucho más rica aún, pero se verá con detalle en los siguientes capítulos.

	Secuencia de llamadas	**Pointcut**	**Advice**	**Salida**
1	main(String args[])			Comienzo de main
2	void math.Potencia.setBase(int)	logPoint()	before	execution(void math.Potencia.setBase(int))
3	void math.Potencia.setExponente(int)	logPoint()	before	execution(void math.Potencia.setExponente(int))
4	int math.Potencia.getBase()	logPoint()	before	execution(int math.Potencia.getBase())
5	int math.Potencia.getExponente()	logPoint()	before	execution(int math.Potencia.getExponente())
6	timeToProcess()	processPoint()	around	Comprobando carga procesador...
7		processPoint()	around	Comenzando cálculos...
8	int math.Potencia.calcula()			
9		processPoint()	around	Calculos terminados!
10	main(String args[])			2 elevado a 4 es 16
11	.....	.....	.....	.....

Y bien esto es todo lo que se requiere escribir. Cuando se compile junto con las clases de la aplicación, el entrelazador de código se encargará de insertar estos nuevos comportamientos, sin que se haya requerido, por tanto, más intervención del programador. Si no se quisieran seguir utilizando, lo único que hay que hacer es volver a compilar la aplicación pero sin el aspecto que se quiera excluir (¡sin necesidad de retocar código!)

La variable `thisJoinPoint` la crea el compilador de AspectJ y contiene meta-información sobre el punto de enlace que se haya alcanzado. Es usual utilizar los métodos, `getThis()`, que devuelve el objeto sobre el que se estuviera ejecutando la función (si es que está disponible); `getTarget()`, que igualmente si está disponible representa al objeto capturado; `getSignature()`, que devuelve el nombre de la función a la que se refiere el punto de enlace; y `getArgs()` que devuelve los argumentos disponibles en el punto de enlace. Al ejecutarlo se obtiene el siguiente resultado por pantalla:

```
Comienzo de main
execution(void math.Potencia.setBase(int))
execution(void math.Potencia.setExponente(int))
execution(int math.Potencia.getBase())
execution(int math.Potencia.getExponente())
Comprobando carga del procesador...
Comenzando calculos...
Calculos terminados!
2 elevado a 4 es 16
Comprobando carga del procesador...
Comenzando calculos...
Calculos terminados!
2 sumado 3 veces es 6
```

Las cuatro primeras líneas muestran el momento previo a las llamadas a las funciones públicas de la clase Potencia, `setBase`, `setExponente`, `getBase` y `getExponente`. En el caso de `sumatorio`, las funciones `get` son privadas, por lo que no se muestran, ya que tales puntos de enlace no están definidos en el punto de corte `logPoint`. Nuestro otro aspecto nos ha avisado por pantalla del comienzo del proceso de cálculo, esto es de las llamadas a `calcula`, puesto que el procesador tenía tiempo libre. Para ver más claramente cual es la secuencia de llamadas podemos observar la tabla anterior donde se ha querido mostrar lo siguiente[3]:

- En la primera fila vemos el comienzo del método `main` y la salida que produce.
- Después de crearse el objeto `Potencia`, se llama a sus métodos `sets` y eso hace que se ejecute el aviso `before` ya que el punto de corte captura dichas llamadas. Eso se muestra en las filas tres y cuatro.
- Igualmente ocurre con los métodos `gets` en las filas cuatro y cinco.
- Cuando se llega al método `calcula()`, se ejecuta el aviso `around`, que hace que primero se llame a la función `timeToProcess()` (línea 6) y después se produce la salida mostrada en la línea siete que corresponde al cuerpo del aviso.
- A continuación se ejecuta el cuerpo de `calcula()`, que se muestra en la fila ocho y que no implica ninguna salida.
- Después de ejecutarse, se vuelve al aviso `around` y se produce la salida mostrada en la línea nueve.
- De vuelta en el método `main`, se produce la salida que muestra los resultados.

Proceso de entrelazado

El compilador de AspectJ nos permite ver los resultados intermedios que genera antes de producir el byte-code final. Esto nos permite entender mejor el funcionamiento del entrelazador de código y ver cómo funciona la implementación que el grupo de AspectJ ha hecho de los conceptos de la metodología de POA. Para ello basta con compilar los ficheros con la directiva `–preprocess`. Para compilar el ejemplo anterior obteniendo el código preprocesado, debemos escribir:

```
>ajc -preprocess main.java Logger.java checkTimeToProcess.java ./math/Sumatorio.java
./math/Potencia.java
```

Como resultado el compilador ajc produce en un directorio de trabajo (que también se podría haber especificado en la línea de comandos) un conjunto de ficheros Java algo más grandes y menos legibles que los originales y que incorporan los aspectos especificados entrelazados. En el caso del ejemplo, los ficheros que se han creado son los mismos, incluidos los que describen los aspectos. El siguiente listado muestra el nuevo fichero `logger.java` generado por el compilador:

Código 6 logger.java

```
public class Logger {

  public final void before0$ajc(org.aspectj.lang.JoinPoint thisJoinPoint){
    System.out.println(thisJoinPoint);
  }

  public Logger() {
    super();
  }

  public static Logger aspectInstance;

  public static Logger aspectOf() {
    return Logger.aspectInstance;
  }

  public static boolean hasAspect() {
    return Logger.aspectInstance != null;
  }
```

[3] Por brevedad no se ha mostrado la traza completa, pues no aporta ninguna idea nueva.

```
  static {
       Logger.aspectInstance = new Logger();
  }
}
```

La primera observación es que se ha convertido en una clase normal java y ha dejado de estar definido como un aspecto. Segundo, el cuerpo del aviso `before`, se ha convertido en una función normal java de nombre `before0$ajc` que recibe un parámetro del tipo `org.aspectj.lang.JoinPoint` con información contextual. El resto es código auxiliar necesario para la implementación.

El código generado para `Potencia.java` es bastante más largo que el original, ya que, además, se han añadido más métodos. Como ejemplo veremos un fragmento para ver que ha ocurrido con el método `getBase()`:

```
public int getBase() {
  final org.aspectj.lang.JoinPoint thisJoinPoint =org.aspectj.runtime.reflect.Factory.makeJP
       (Potencia.getBase$ajcjp1, this, this, new java.lang.Object[] {});

  Logger.aspectInstance.before0$ajc(thisJoinPoint);

  return this.base;
}
```

Este método ha conservado su nombre, pero ha variado su cuerpo. Se ve afectado por el aviso `before` del aspecto `Logger`, lo que se realiza en la segunda instrucción con la llamada a `Logger.aspectInstance.before0$ajc(thisJoinPoint)`. A continuación se ejecuta el cuerpo original que tuviera el método, en este caso únicamente la instrucción `return`. La primera instrucción facilita la información contextual, creando el objeto `thisJoinPoint` que recibe como primer parámetro un atributo estático que el compilador le ha añadido a la clase, `org.aspectj.lang.JoinPoint.StaticPart`, así como el propio objeto y la lista de parámetros de la llamada, en este caso, una lista de objetos vacíos.

Algo parecido ha ocurrido con el método calcula de las clases `Sumatorio` y `Potencia` que estaba afectado por un aviso `around`. El compilador ha generado principalmente tres métodos para ella en cada clase para entrelazar el código del aspecto `checkTimeToProcess`. A continuación se muestran estas funciones para la clase `Potencia`:

```
public int calcula() {
  return this.around2_calcula(((org.aspectj.runtime.internal.AroundClosure)(null)),
                                checkTimeToProcess.aspectInstance);
}

public final int around2_calcula(final org.aspectj.runtime.internal.AroundClosure
                                       ajc$closure, final checkTimeToProcess this_){
   while (!this_.timeToProcess()){
     //sleep();
   }

   System.out.println("Comenzando calculos..");
   int result = this.dispatch2_calcula();
   System.out.println("Calculos terminados!");

   return result;
}

final int dispatch2_calcula(){
   int result = 0;
   for (int i = 0; i < this.exponente; i++)
      result = result + this.base * this.base;
   return result;
}
```

El método calcula ha dado resultado en tres métodos. El original `calcula` llama a otro `around2_calcula`, donde se encuentra el código definido en el aviso y en el lugar donde se hace la llamada a `proceed` en el código original se ha sustituido por `this.dispatch2_calcula`. Esta función contiene el código original de la función

tal como está escrito en la clase Potencia. Examinado los ficheros nos encontramos bastante más código, que sirve fundamentalmente para la captura de información contextual.

Con objeto de poder comparar literalmente el código original de la clase Potencia y el código generado por el compilador `ajc` una vez que ha entrelazado el código de los dos aspectos `Logger` y `checkTimeToProcess` puede verse la tabla siguiente donde a la izquierda se muestra el código original y a la derecha el código compilado.

package math;	package math; import checkTimeToProcess; import Logger;
public class Potencia{	public class Potencia { static org.aspectj.runtime.reflect.Factory ajc$JPF; private static org.aspectj.lang.JoinPoint.StaticPart getBase$ajcjp1; private static org.aspectj.lang.JoinPoint.StaticPart getExponente$ajcjp2; private static org.aspectj.lang.JoinPoint.StaticPart setBase$ajcjp3; private static org.aspectj.lang.JoinPoint.StaticPart setExponente$ajcjp4;
int base; int exponente;	int base; int exponente;

public int getBase(){
 	
return base;
}
	public int getBase() {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP
	(Potencia.getBase$ajcjp1, this, this, new java.lang.Object[] {});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	return this.base;
	}

public int getExponente(){
 	
return exponente
}
	public int getExponente() {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP
	(Potencia.getExponente$ajcjp2, this, this, new java.lang.Object[] {});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	return this.exponente;
	}

public void setBase(int dat){
 	
this.base=dat;
}
	public void setBase(int dat) {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP(
	Potencia.setBase$ajcjp3, this, this, new java.lang.Object[] {new Integer(dat)});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	this.base = dat;
	}

public void setExponente(int dat){
 	
this.exponente=dat;
}
	public void setExponente(int dat) {
	final org.aspectj.lang.JoinPoint thisJoinPoint = org.aspectj.runtime.reflect.Factory.makeJP(
	Potencia.setExponente$ajcjp4, this, this, new java.lang.Object[] {new Integer(dat)});
	Logger.aspectInstance.before0$ajc(thisJoinPoint);
	this.exponente = dat;
	}

public int calcula(){
 	
int result=0;
for (int i=0; i<exponente; i++)
result=result+base*base;
return result;
 	
}
	public int calcula() {
	return this.around2_calcula(((org.aspectj.runtime.internal.AroundClosure)(null)),
	checkTimeToProcess.aspectInstance);
	}
	
	final int dispatch2_calcula() {
	int result = 0;
	for (int i = 0; i < this.exponente; i++)
	result = result + this.base * this.base;
	return result;
	}
	
	public final int around2_calcula(final org.aspectj.runtime.internal.AroundClosure ajc$closure,
	final checkTimeToProcess this_) {
	while (!this_.timeToProcess()){/*sleep();*/}
	System.out.println("Comenzando calculos...");
	int result = this.dispatch2_calcula();
	System.out.println("Calculos terminados!");
	return result;
	}

```
                                    public Potencia() {    super();  }

                                    static {
                                     Potencia.ajc$JPF = new org.aspectj.runtime.reflect.Factory("Potencia.java", Potencia.class);
                                     Potencia.getBase$ajcjp1 = Potencia.ajc$JPF.makeSJP("method-execution",
                                          Potencia.ajc$JPF.makeMethodSig("1-getBase-math.Potencia----int-"), 8, 2);
                                     Potencia.getExponente$ajcjp2 = Potencia.ajc$JPF.makeSJP("method-execution",
                                          Potencia.ajc$JPF.makeMethodSig("1-getExponente-math.Potencia----int-"), 9, 2);
                                     Potencia.setBase$ajcjp3 = Potencia.ajc$JPF.makeSJP("method-execution",
                                          Potencia.ajc$JPF.makeMethodSig("1-setBase-math.Potencia-int:-dat:--void-"), 11, 2);
                                     Potencia.setExponente$ajcjp4 = Potencia.ajc$JPF.makeSJP("method-execution",
                                       Potencia.ajc$JPF.makeMethodSig("1-setExponente-math.Potencia-int:-dat:--void-"), 12, 2);
}                                    }
                                    }
```

8 REFERENCIAS RECOMENDADAS

- Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John. Irwin, *Aspect-Oriented Programming*, Xerox Palo Alto Research Center, 1997.
- Ivan Kiselev, *Aspect-Oriented Programming with AspectJ*, Sams Publishing, 2003.
- *The AspectJ™ Programming Guide*, the AspectJ Team, Xerox Parc Corporation
- Gregor Kiczales, et al. *An Overview of AspectJ*. In Proceedings of the 5th European Conference on Object Oriented Programming (ECOOP), Springer. Budapest, Hungary.
- Erik Hilsdale, Jim Hugunin, Wes Isberg, Gregor Kiczales, Mik Kersten, *Aspect-Oriented Programming with AspectJ*, 2002.
- Ramnivas Laddad, *Introduction to AspectJ*, Manning Publications Co. 2003.
- S. Clarke y Robert J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In proceedings of ICSE 2001.
- P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. *N degrees of separation: Multi-dimensional separation of concerns*. In Proceedings of the 21st ICSE'99, May 1999.
- D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, December 1972.
- T. Elrad, R. E. Filman, and A. Bader. *Aspect-Oriented Programming*. Commun. ACM, 2001.
- Aspect Oriented Software Development. http://aosd.net
- Kiczales G., Hannemann J. Design pattern implementation in Java and AspectJ
- Distributed System Group, Trinity College Dublin. http://www.dsg.cs.tcd.ie/
- Howard Kim,. AspectC#: An AOSD implementation for C#. September, 2002.
- AspectC++ Homepage. *http://www.aspectc.org*