Tema 20. Introducción a la programación orientada a aspectos

1. Introducción

Los diferentes paradigmas de programación han ido aportando diferentes formas de modularización. Es decir diferentes formas de encapsular software en diferentes entidades con el objetivo de reutilizarlo. La Programación Orienta a Objetos aportó elementos para modularizar: la interfaz, la clase y el paquete fundamentalmente. Pero aún con estos elementos muchas aplicaciones construidas actualmente contienen código que para implementar determinados requisitos está disperso en distintas partes o enmarañado con código que cumple otros objetivos. Esto ocurre con la implementación de los mecanismos de seguridad, de acceso a recursos, transacciones, etc. es decir el software que implementa esos requisitos no está aislado, usualmente, en módulos sino que está disperso por las diferentes clases si estamos usando programación orientada a objetos.

La Programación Orientada a Aspectos es un avance en las técnicas de modularización. En Programación Orientada a Objetos, como hemos comentados, la unidad de modularización es la clase que se a su vez se puede agrupar en paquetes. La Programación Orientada a Aspectos aporta nuevos conceptos y una nueva unidad de modularización: el aspecto. Pretende agrupar en las nuevas unidades de modularización (los aspectos) el software que implementa requisitos, como los vistos anteriormente, que hasta ahora estaba disperso por diferentes clases.

Aquí veremos una versión concreta de la Programación Orientad a Aspectos: *AspectJ* que es una implementación de la Programación Orientada a Aspectos a Java. *AspectJ* aporta nuevos conceptos a Java. Mantendremos su denominación en inglés por ser la más extendida.

- *Join point*: Es un punto bien definido en el flujo de un programa. Ejemplos pueden ser: llamada a un método, acceso a un atributo de una clase, llamada a un constructor, etc. Podemos identificarlos, por tanto, por puntos en un programa y específicamente en el flujo de control de ese programa aunque también veremos *Joint Points* descritos en términos estáticos. Es decir con referencia al código del programa. Aunque usaremos la palabra en inglés una forma de denominarlos en castellano es *punto de unión* o *punto de enlace*.
- *Pointcut*: Es un conjunto de *Joint Points* y valores que describen su contexto. Este contexto puede ser, como veremos, el valor de los parámetros reales de un método, el objeto sobre el que se invoca un método o desde el que se ha invocado, etc. El conjunto de *Joint Point*s se describe por expresiones que recuerdan las expresiones regulares. Se usan asteriscos que pueden ser instanciados por secuencias de caracteres, etc. Si *Joint Point* concreto pertenece al conjunto descrito por un *Pointcut* decimos que concuerda con él o que se adecua a él. En castellano podemos designarlo por *conjunto de puntos de unión*.
- *Advice*: Es el código que queremos que se ejecute en cada *Joint Point* descrito por un *Pointcut*. A los *Joint Points* podemos describirlos como puntos de unión o puntos de enlace porque en ellos se une, se enlaza, en definitiva se ejecuta un código según se indica en el *Advice*. En castellano podemos designarlo como *código del punto de unión*.
- *Declaraciones Inter Tipo*: Manera de modificar la estructura estática de un programa. Es decir los miembros de una clase o la relación entre ellas
- *Aspect*: Es la unidad de modularidad que puede incluir *Pointcut*, *Advise*, declaraciones inter tipos y también todos los elementos de una clase Java: atributos, métodos, etc. En castellano los designaremos como *aspectos*.

Un programa *AspectJ* puede ser considerado como una extensión de un programa Java. Añade aspectos a un programa *Java* y con ello añade dos elementos. Uno dinámico: ejecuta el código indicado en los puntos señalados (el código incluido en los *advices*). Al incluir código, es posible cambiar el flujo de control del programa original *Java,* simplemente añadiendo código adicional o dejando de ejecutar código existente. El segundo elemento es estático: es posible, con los aspectos añadidos, cambiar la estructura estática del programa *Java*. Se pueden añadir nuevos atributos a las clases del programa Java sin modificarlo, hacer que una clase implemente una interfaz o herede de una clase. Al poder agruparse todos los elementos en un aspecto tenemos un mecanismo de modularidad nuevo.

1.1 Joint Points

Como hemos explicado anteriormente un *Joint Point* es un punto concreto en un programa. En *AspectJ* los *Joint Points* más importantes son los siguientes:

- Llamada a un método.
- Ejecución del cuerpo de un método.

- Llamada a un constructor.
- Ejecución del cuerpo de un constructor
- Acceso a atributo
- Asignación de valor a atributo
- Ejecución de un gestor de excepciones

Otros *Joint Points* son las ejecuciones de diversos inicializadores: inicialización o pre inicialización de un objeto o de un inicializador estático.

Cada *Joint Point* tiene un contexto: el objeto que está ejecutándose, el objeto sobre el que se va a invocar un método y los parámetros reales de la llamada.

En *AspectJ* se usa una sintaxis concreta para describir conjuntos de *Pointcut* (conjuntos de *Joint Point*). Veamos la forma de describir los diferentes tipos de puntos de un programa.

Un método se describe por su signatura que incluye los modificadores del método. Para describir conjuntos de métodos se usan los comodines .., **, !,* . El símbolo * en el nombre de un método representa cualquier número de caracteres que no incluyan el "." y en la zona parámetros un parámetros de cualquier tipo. El símbolo .. en la zona de parámetros representa cero o más parámetros de cualquier tipo. El símbolo *!* delante de un modificador del método o de una excepción disparada representa el resto de modificadores o excepciones. Es decir todos los métodos que no tienen ese modificador o no disparan esa excepción. Llamaremos *Patrón de Método* a un conjunto de métodos descritos de esta forma.

Los constructores se describen como un método más pero su nombre se forma con el nombre del tipo (o un *Patrón de Tipo*) seguido de punto, la palabra new y los posibles parámetros con sus tipos. Además, como todos los métodos, puede tener modificadores y excepciones disparadas. Llamaremos *Patrón de Constructor* a un conjunto de constructores de esta forma.

Los atributos se describen mediante su tipo y su nombre. Se puede usar el comodín *. Así se forma un *Patrón de Atributo*.

Los tipos se describen por su nombre (posiblemente junto con el paquete donde están). Para describir conjuntos de tipos se usan los comodines *,..,+. El símbolo * describe cualquier secuencia de caracteres que no incluya ".". El símbolo .. describe cualquier secuencia de caracteres que empiece y acabe con ".". El símbolo *+* al final de un tipo representa todos los subtipos del mismo. Llamaremos *Patrón de Tipo* a un conjunto de tipos descritos de esta forma.

Un *Pointcut* se puede escribir a partir de un conjunto de *Pointcut* primitivos. Con los elementos anteriores podemos describir la sintaxis concreta de estos *Poincut* primitivos*:*

- *call(Patrón de Método):* Llamada a cualquier método de los descritos en el patrón
- *execution(Patrón de Método):* Ejecución de cualquier método de los descritos en el patrón
- *get(Patrón de Atributo):* Lectura de cualquier atributo de los descritos
- *set(Patrón de Atributo):* Asignación de valor a cualquier atributo descrito.
- *call(Patrón de Constructor):* Llamada a un constructor descrito por el patrón

- *execution(Patrón de Constructor):* Ejecución de un constructor descrito por el patrón
- *initialization(Patrón de Constructor):* Inicialización de un objeto que se construye con uno de los constructores descritos.
- *preinitialization(Patrón de Constructor):* Pre-inicialización de un objeto que se construye con uno de los constructores descritos.
- *staticinitialization(Patrón de Tipo):* Inicialización estática de cualquier tipo descrito por el patrón.
- *handler(Patrón de Tipo):* Punto de un programa donde se ha capturado una excepción del tipo descrito y se comienza a gestionar. Usualmente es el comienzo de un bloque *catch*.
- *adviceexecution():* Ejecución de un *advice* tal como lo veremos más adelante.
- *within(Patrón de Tipo):* Cualquier punto interno del código que define uno de los tipos descritos (el código de las clases correspondientes)
- *withincode(Patrón de Método):* Cualquier punto interno del código del cuerpo del método descrito.
- *withincode(Patrón de Constructor):* Cualquier punto interno del código del cuerpo del constructor descrito.
- *cflow(Pointcut):* Cualquier punto dentro del flujo de control que comienza en el *Pointcut* descrito incluido el mismo.
- *cflowbelow(Pointcut):* Cualquier punto dentro del flujo de control que comienza en el *Pointcut* descrito pero sin incluir a el mismo.
- *this(Tipo o Id):* Describe cualquier punto en la ejecución de un programa donde el objeto que se está ejecutando es una instancia de *Tipo* o del tipo del identificador *Id* que debe estar fijado por el contexto.
- *target(Tipo o Id):* Describe cualquier punto en la ejecución de un programa donde el objeto que sobre el que se va llamar un método o acceder a un atributo es una instancia de *Tipo* o del tipo del identificador *Id* que debe estar fijado por el contexto.
- *args(Tipo or Id, ...):* Describe cualquier punto en la ejecución de un programa donde los parámetros son del tipo indicado o del tipo de *Id*.
- *PointcutId(Patrón de Tipo o Id, ...):* Describe cualquier punto descrito por un Pointcut designado por el identificador *Id*.
- *if(Expresión Lógica):* Todos los puntos donde la expresión lógica es verdadera.
- *! Pointcut:* Cada punto no incluido en los descritos por le *Pointcut*.
- *Pointcut0 && Pointcut1:* Intersección de los conjuntos de puntos descritos por ambos *Pointcut*
- *Pointcut0 || Pointcut1:* Unión de los conjuntos de puntos descritos por ambos *Pointcut*
- *(Pointcut):* Conjuntos de puntos descritos por el *Pointcut* .

Una expresión formada por *Pointcuts* primitivos combinados con los operadores *!*, (), *&&* y *||* define un nuevo *conjunto de puntos de unión*.

Junto a la visión anterior es conveniente añadir algunas precisiones más detalladas entre los puntos de unión *call* y *execution*. El primero designa el punto de la llamada al método el segundo el punto de la ejecución del mismo. Son casi iguales pero tienen algunas diferencias.

El punto unión que denominamos *call* comienza cuando con la llamada inicial y acaba cuando el control retorna al llamador. El punto unión que denominamos *execution* comienza al principio del cuerpo del método (o constructor) y acaba cuando termina el cuerpo (ya sea en el final o disparando una excepción). Por lo tanto el punto de unión *execution* siempre ocurre dentro de las cotas de *call*. En un punto de unión *call* el contexto (para determinar *this, target*, etc.) es el de la llamada. En un punto de unión *execution* el contexto es el posterior a la llamada es decir el del llamado. El objeto *this,* por lo tanto, es el llamador en el punto de unión *call* y el llamado en el punto de unión *execution*.

La recomendación es usar el punto de unión (*joint point*) *execution* cuando un determinado código se ejecute y usar el punto de unión *call* cuando una signatura concreta es llamada.

Una declaración de *Pointcut* sirve para dar un identificador a una expresión que describe un *Pointcut*. Se compone de una cabecera y un cuerpo. Su sintaxis se compone de la cabecera y el cuerpo separados por dos puntos y terminada en punto y coma. La cabecera (equivalente a la cabecera de un método) comienza por la palabra clave *pointcut*, el identificador, los parámetros formales. El cuerpo es una expresión de *Pointcut*. Un cuerpo de forma aislada define un *Pointcut* anónimo.

Una declaración *Pointcut* se puede hacer dentro de una clase o dentro de un aspecto (como veremos más adelante) y se trata como un miembro de la clase. Es decir de forma similar a un método. Como estos puede tener los siguientes modificadores: *public*, *private*, *abstract*, *final*. Una declaración con el modificador *abstract* define un identificador para un *Pointcut* pero no le asigna una expresión. Es decir una declaración *abstract* define un *Pointcut* sin un cuerpo como en el caso de una declaración de método. Los significados de estos modificadores son los mismos que en el caso de los métodos pero a diferencia de ellos los *Pointcuts* no pueden sobrecargarse.

Una declaración de *Pointcut,* como en el caso de una declaración de método, puede tener parámetros formales. Pero a diferencia de los métodos ahora los parámetros toman su valor en el cuerpo del *Pointcut* y ofrecen este valor en el sitio donde el *Pointcut* está siendo utilizado. Es decir los parámetros funcionan desde dentro del cuerpo hacia fuera a diferencia de los métodos donde los parámetros reales son usados en el cuerpo.

1.2 Ejemplos de Conjuntos de Puntos de Unión (Pointcuts)

- *call(void Punto.setX(int)):*

  Llamada al método del tipo Punto con esa signatura

- *call(* Punto.create*(..)):*

  Todas las llamadas a métodos del tipo Punto cuyo nombre comienza con *create* e independientemente del número o tipo de sus parámetros.

- *call(public * Punto.* (..)):*

Llamadas a métodos públicos del tipo Punto independientemente de nombre, del tipo que devuelven y los parámetros que tienen.

- *pointcut move(): call(void Punto.setXY(int,int)) ||*
  call(void Punto.setX(int)) ||
  call(void Punto.setY(int)) ;

  Definición un conjunto de puntos de unión (*Pointcut*) sin parámetros , de nombre *move*. Describe el conjunto de puntos de unión formado por las llamadas a los métodos indicados del tipo *Punto*.

- *cflow(move())*:

  Cada punto de unión que ocurre entre la llamada a un método descrito por *move()* y su retorno. Es decir en el flujo de control de los métodos descritos por *move().*

- *pointcut setXY(Punto p, double x, double y):*
  call(void Punto.setXY(double, double)) &&
  target(p) &&
  args(x, y);

  Llamadas al método *setXY* del tipo punto con dos parámetros de tipo *int*. El *Pointcut* define los parámetros *p, x, y* que se instancian al punto sobre el que se invoca el método, y los parámetros reales.

- *execution(public !static * *(..)):*

  Ejecución de cualquier método público pero no static

- *call(int get*()):*

  Llamadas a métodos que comiencen por *get*, sin parámetros y que devuelvan *int.*

- *execution(private C.new() throws ArithmeticException):*

  Ejecución del constructor privado y sin parámetros de la clase *C* y que dispara la excepción indicada.

- *within(com.xerox..*):*

  Métodos definidos en cualquier sub-paquete de *com.xerox*

- *call(*Punto+.new()):*

  Llamadas a constructores sin parámetros de objetos de un tipo cuyo nombre acaba en

Punto o uno de sus subtipos.

- *call((Punto+ && ! Punto).new(..)):*

  Llamadas a constructores de objetos que son instancias de subtipos de *Punto* pero no el mismo tipo *Punto*.

1.3 Código en los puntos de unión (advices)

Un *advice*, código en un punto de unión, es la especificación de que código se ejecutará en el punto de unión correspondiente. La sintaxis concreta de un *advice* es :

- *[strictfp] TipoDeAdvice [throws Lista de Tipos] : Pointcut { Código }*

Donde *TipoDeAdvice* es una de las alternativas.

- *before(Parámetros Formales)*
- *after(Parámetros Formales) returning [(Tipo)]*
- *after(Parámetros Formales) throwing [(TipoExcepcion)]*
- *after(Parámetros Formales)*
- *Tipo around(Parámetros Formales)*

Parámetros Formales es una lista de parámetros formales, con sus tipos respectivos, de forma similar a los parámetros formales de un método. La diferencia está en que su valores serán encontrados a partir de las restricciones impuestas en el *Pointcut.* En la llamada a un método, sin embargo, hay que proporcionar valores concretos, los parámetros reales, para cada parámetro formal. Por *Tipo* y *TipoExcepcion* representamos el tipo del valor que se devuelve o de la excepción disparada.

Aspectj dispone de tres tipos de tres formas distinta de insertar del código en cada punto de unión. E s decir de tres tipos de *advices*: *before, after, around*. El primero, *before*, inserta el código antes de cada uno de los puntos de unión descrito en el *Pointcut*. El segundo, *after*, inserta el código después de cada punto de unión. El tercero, *around*, inserta el código antes y después de cada punto de unión.

El *advice* tipo *after* tiene tres modalidades: *after-returnig* cuando la ejecución tras el punto de unión se hace en modo normal devolviendo, posiblemente, un valor del tipo indicado, *after-throwing* cuando la ejecución es en modo excepcional disparando una excepción del tipo indicado o *after* que no hace distinción si la ejecución es en modo normal o excepcional.

El *advice* de tipo *around* es más complejo, veremos ejemplos más adelante, y permite insertar código antes, después o en lugar del código que comienza en el punto de unión. En el cuerpo de un *advice* de tipo *around* está disponible la llamada al método *proceed(…)* con los mismos parámetros formales y tipo devuelto que el *advice*, de tipo *around*, donde se está usando. La llamada a *proceed(…)* continúa con la ejecución del código que seguía al punto de unión descrito en el *advice* pero con los parámetros reales proporcionados. La llamada devuelve el valor devuelto por el código correspondiente tras su ejecución.

En el código de un *advice* hay disponibles tres variables que guardan información sobre el contexto. Estas variables son:

- *thisJoinPoint* de tipo *JointPoint*.
- *thisJoinPointStaticPart* de tipo *JoinPointStaticPart* que contiene la parte estática del contexto y es un subtipo de *JoinPointStaticPar*
- *thisEnclosingJoinPointStaticPart* de tipo *EnclosingJoinPointStaticPart.*

Pueden existir a la hora de ejecutar el código definido por distintos advices que comparten puntos de unión concretos. Para resolver estos conflictos se aplican las siguientes reglas cuando ambos *advice* están definidos en la misma unidad (aspecto):

- Entre dos *advices* de tipo *after* tiene prioridad el que se define posteriormente. En otro caso el que se define en primer lugar.

Es posible indicar explícitamente la prioridad entre dos *advices* declarados en aspectos diferentes. Esto se hace con la declaración de *precedence*:

- *declare precedence: a1,a2,a3,…;*

Donde el código declarado en cada aspecto tiene prioridad sobre el siguiente en la declaración.

1.4 Los tipos JointPoint, JointPoint.StaticPart

El tipo *JointPoint* tiene los siguientes métodos y propiedades:

- *Object[] getArgs():* Devuelve los parámetros reales en este punto de unión.
- *String getKind():* Devuelve el tipo de punto de unión
- *Signature getSignature():* Devuelve la signatura en el punto de unión
- *SourceLocation getSourceLocation():* Devuelve la ubicación en el código fuente correspondiente al punto de unión
- *JoinPoint.StaticPart getStaticPart():* Devuelve la parte estática del contexto del punto de unión
- *Object getTarget():* Devuelve el objeto sobre el que se invoca el código correspondiente
- *Object getThis():* Devuelve el objeto desde el que se invoca el código
- *toString(), toLongSring(), toShortString():* Diversas representaciones en hileras de caracteres de la información del contexto.

El tipo *JoinPoint.getStaticPart()* contiene solo la información estática del contexto es decir un subconjunto de los métodos anteriores. En concreto los métodos: *getKind(), getSignature(), getSourceLocation() y* las correspondientes versiones de *toString().*

El tipo *Signature* contiene la información sobre la signatura en el punto de unión. Sus propiedades y métodos, además de la versiones correspondientes de *toString(),* son:

- *Class getDeclaringType():* Devuelve la clase, interfaz o aspecto que representa el tipo al que pertenece la signatura
- *String getDeclaringTypeName():* El nombre cualificado del tipo al que pertenece la signatura
- *int getModifiers():* Devuelve un entero que codifica los diversos modificadores presentes
- *String getName() :* Devuelve el identificador presente en la signatura

El tipo ofrece otros subtipos que proporcionan información más detallada sobre atributos, métodos, constructores, cláusulas *catch*, etc.

El tipo *SourceLocation* contiene la información sobre la ubicación en el código del punto de unión. Sus propiedades y métodos, además de la versiones correspondientes de *toString(),* son:

- *String getFileName():* Nombre del fichero fuente
- *int getLine() :* Número de línea en el fichero
- *Class getWithinType() :* Tipo dentro del cual está declarado el punto de unión

1.5 Ejemplos de joint points y advices

- *pointcut move(): call(void Punto.setXY(int,int)) ||*
  call(void Punto.setX(int)) ||
  call(void Punto.setY(int)) ;
  Conjunto de puntos de unión formados por las llamadas a los métodos *setX, setY, setXY* del tipo Punto.
- *before(): move() { System.out.println("a punto de trasladarse"); }*
  Muestra el mensaje correspondiente antes de cada uno de los puntos de unión en *move*.
- *after() returning: move() { System.out.println("el punto se ha trasladado con éxito"); }*
  Muestra el mensaje correspondiente después de cada uno de los puntos de unión en *move*.
- *pointcut setXY(Punto p, double x, double y):*
  call(void Punto.setXY(double, double)) && target(p) && args(x, y);
  Conjunto de puntos de unión formado por las llamadas al método *setXY* del tipo *Punto* con dos parámetros de tipo *double*. Se captura el contexto en las variables *p,x,y*: el objeto sobre el que se invoca el método, el primer argumento y el segundo.
- *after(Punto p, double x, double y) returning: setXY(fe, x, y) {*
  System.out.println(p + " se ha trasladado a (" + x + ", " + y + ").");
  }
  Despúes de las llamadas a los métodos incluidas en el *pointcut setXY* se muestra un mensaje con el objeto sobre el que se invoca y las nuevas coordenadas.
- *void around(Punto p, double x): target(p) && args(x) && call(void setX(double)) {*
  if (p.assertX(x)) proceed(x);

}

En las llamadas al método *setX* se comprueba si los parámetros reales cumplen la condición especificada en el método *assertX* del tipo *Punto.* Si la cumplen se ejecuta el método en otro caso no se ejecuta el método.

1.6 Otros puntos de unión disponibles en AspectJ

Cualquier declaración en Java puede ser marcada con una anotación. Es decir se puede usar sobre los siguientes elementos del lenguaje: paquete, clase, interface, atributo, método, constructor, parámetro y enumerado. Una anotación puede ser anotada con otra anotación (llamadas meta anotaciones)

Las anotaciones pueden tomar parámetros y algunos parámetros arrays de valores. Los array de valores se rodean de {…} y los parámetros de (…).

Algunos ejemplos son:

```
@AnotacionDeClase(arg1="val1", arg2={"arg2.val1","arg2.val2"})
public class AnnotationExample {
   @AnotacionDeAtributo()
   public String field;
   @AnotacionDeConstructor()
   public AnnotationsTest() {        // code    }
   @AnotacionADeMetodo("val")
   @AnotacionBDeMetodo(arg1="val1",arg2="val2")
   public void someMethod(String string) {        // code    }
}
```

Se pueden declarar tipos de anotaciones. Se declaran como las interfaces. Con la palabra *@interface*. Por ejemplo:

```
public @interface NuevaAnotacion {
    int getCodigo();
    String getNombre();
    String getReferencia();
}
```

Basadas en las anotaciones están disponibles nuevos puntos de unión que son:

- *@this(Anotacion):* El tipo del objeto que se está ejecutando está anotado con la anotación correspondiente.
- *@target(Anotacion):* El tipo del objeto sobre el que se invoca el método está anotado con la anotación correspondiente.
- *@args(Anotacion), @args(*,Anotacion), @args(Anotacion1,..,Anotacion2):* El parámetro correspondiente está anotado con la anotación indicada.
- *@within(Anotacion), @withincode(Anotacion):* Conjunto de puntos de unión internos al código que define un tipo (o un método constructor) anotado con la aanotación indicada.

1.7 Declaraciones Inter Tipo

Las declaraciones inter tipo son declaraciones que pueden modificar el código de una clase de forma externa a ella. Sirve para incluir externamente nuevos atributos, métodos y constructores en una clase pero poniendo el código fuera de la clase. Estas declaraciones se introducen en tiempo de compilación. Es decir estáticamente. Sin embargo las declaraciones de *advice* tienen efecto normalmente en tiempo de ejecución

Algunos ejemplos son:

- *private int Punto.x = 0;* se incluye el atributo privado *x* en el tipo punto.
- *public int Punto.getX() { return this.x; } : S*e incluye un nuevo método llamado *getX* en el tipo *Punto*.
- *public Punto.new(int x, int y) { this.x = x; this.y = y; } :* Se incluye un nuevo constructor en el tipo Punto.

1.8 Declaraciones que modifican la estructura estática de un programa y la respuestas del compilador

Con ellas cambiar las relaciones de herencia y de implementación, cambiar el tipo de errores generados por el compilador, proporcionar implementaciones compartidas por varias clases, etc.

Las posibilidades son:

- *declare parents: Punto implements Comparable<Punto>;* La clase *Punto* implementa *Comparable<Punto>*
- *declare parents: Punto extends ObjetoGeometrico;* La clase punto extiende *ObjetoGeometrico.* Solamente está permitido si *ObjetoGeometrico* extiende la superclase original de *Punto*.
- *declare warning : set(* Punto.*) && !within(Punto) : " set no adecuado" ;* El compilador genera un mensaje indicando que no es adecuado modificar un atributo del tipo Punto fuera del código que define la clase.
- *declare error : call(Singleton.new(..)) : "constructor no adecuado" ;* El compilador genera un error si encuentra la llamada a un constructor de un *singleton*.
- *declare soft : IOException : execution(C.new(..));* Cualquier excepción disparada desde un constructor del tipo C queda convertida en otra de tipo *org.aspectj.SoftException.* Esta extiende *java.util.RunTimeException* con las implicaciones que esto tiene.
- *declare precedence : Security, Logging, * ;* Los puntos de unión y *advices* declarados en Security tienen precedencia sobre los declarados en *Logging* y estos sobre los demás.

Un elemento interesante, que veremos cuando hablemos de aspectos, es la posibilidad de proporcionar interfaces junto con una implementación y declarar que varias clase implementan esa interfaz. Esta técnica nos permite una forma de herencia múltiple que no está disponible en *Java*.

1.9 Aspectos

Un Aspecto es la unidad de modularidad que ofrece AspectJ. Tiene una correspondencia directa con el uso que se hace de una clase en Java. Una clase en Java incluye atributos y métodos y puede heredar de otra clase e implementar varias interfaces. De forma similar un aspecto puede heredar de otro o de otra clase, implementar interfaces, puede incluir atributos y métodos y además declaraciones de *pointcuts*, *advice*s y declaraciones inter tipo. Un aspecto, por lo tanto, tiene las posibilidades de una clase extendidas con los detalles específicos de la programación orientada a aspectos.

Un ejemplo es:

```
aspect Logging {
  OutputStream logStream = System.err;
  pointcut move():
    call(void Figura.setXY(int,int)) ||
    call(void Punto.setX(int)) ||
    call(void Punto.setY(int)) ||
    call(void Linea.setP1(Punto)) ||
    call(void Linea.setP2(Punto));
  before(): move() {
    logStream.println("a punto de cambiar");
  }
}
```

A diferencia de las clases los aspectos no se crean con un operador *new().* Las instancias de un aspecto son creadas automáticamente por el compilador y se puede tener acceso a ellas mediante el método estático *aspectOf(..).* Por cada tipo de aspecto hay diferentes políticas para crear instancias: una sola instancia del aspecto (*singleton*) o una instancia por objeto afectado o incluso por cada flujo de control. La opción por defecto es *singleton*.

La forma de declarar la política de creación se hace en la cabecera del aspecto:

- *aspect Id { ... }*
- *aspect Id issingleton() { ... }*
- *aspect Id perthis(Pointcut) { ... }*
- *aspect Id pertarget(Pointcut) { ... }*
- *aspect Id percflow(Pointcut) { ... }*
- *aspect Id percflowbelow(Pointcut) { ... }*

La opción por defecto es issingleton(). En este caso se crea una sólo instancia del aspecto (un singleton). La instancia creada puede ser obtenida en cualquier momento mediante el método estático *aspectOf().* Así para el aspecto *A* (declarado como un singleton) la instancia concreta del mismo que se crea puede ser obtenida mediante *A.aspectOf()*.

Otra posibilidad es que se creen varias instancias del aspecto. Estas se pueden crear asociadas a objetos concretos o a flujos de control. Las opciones *perthis* y *pertarget* crean una instancia

del aspecto por cada objeto que ocupe el papel de *this* o *target* en pointcut explicitado en la cabecera del aspecto. Si un aspecto A es definido *perthis(Pointcut)* entonces se crea una instancia de *A* por cada objeto que se esté ejecutando (*this*) en uno de los puntos de unión descritos por el *Pointcut*. Igualmente ocurre con un aspecto *A* definido *pertarget(Pointcut)* donde se crea una instancia de *A* por cada objeto sobre que ocupa el papel de *target* en uno de los puntos de unión descritos en el *Pointcut*. En los dos casos anteriores la instancia del aspecto creada puede ser obtenida mediante el método estático *A.aspectOf(Object).*

Una segunda posibilidad es crear una instancia del aspecto por cada flujo de control. Esto se consigue con las declaraciones en la cabera del aspecto: *percflow(Pointcut)* y *percflowbelow(Pointcut)*. En ambos caso se crea una instancia del aspecto por cada flujo de control descrito por el *Pointcut* correspondiente (ya al entrar en el flujo de control o tras entrar en él). En cada flujo de control la llamada a *A.aspectOf()* nos devuelve las instancia del aspecto creada.

2. Ejemplos

Para apoyar los ejemplos posteriores partimos de la interfaz *Punto*, la clase *PuntoImpl* y la clase *TestPunto.*

```
public interface Punto {
	Double getX();
	Double getY();
	void setX(Double x);
	void setY(Double y);
}

public class PuntoImpl implements Punto {
	private Double x;
	private Double y;

	public PuntoImpl(){
		x = 0.;
		y = 0.;
	}
	public PuntoImpl(Double x1, Double y1){
		x = x1;
		y = y1;
	}
	public Double getX() {
		return x;
	}
	public Double getY() {
		return y;
	}
	public void setX(Double x1) {
		x = x1;
	}
	public void setY(Double y1) {
		y = y1;
	}
```

```
	public boolean equals(Object p){
		boolean r = false;
		if(p instanceof Punto){
			Punto p1 = (Punto)p;
			r = getX().equals(p1.getX()) && getY().equals(p1.getY());
		}
		return r;
	}
	public int hashCode(){
		return getX().hashCode()+31*getY().hashCode();
	}
	public String toString(){
		String s;
		s = "("+getX()+","+getY()+")";
		return s;
	}
}

public class TestPunto extends Test {

	public static void numeros(){}

	public static void main(String[] args) {
		Punto p = new PuntoImpl();
		Punto p1 = new PuntoImpl(2.0,3.0);
		p1.setX(-3.4);
		p1.setY(27.);
		mostrar("La suma de las coordenadas de "+ p1+
		  " es  "+p1.getX()+p1.getY());
		TestPunto.numeros();
	}

}
```

Ejemplo 1: traza simple

```
aspect SimpleTracing {
	pointcut tracedGetCall():
		call(* Punto.get*());
	pointcut tracedSetCall(Double c):
		call(* Punto.set*(*)) &&
		args(c);
	before(): tracedGetCall() {
		Test.mostrar("Obteniendo el valor de: " +
				thisJoinPointStaticPart);
	}
	before(Double c): tracedSetCall(c) {
		Test.mostrar("Cambiando Coordenada a   "+ c +"  en  "+
				thisJoinPointStaticPart) ;
	}
}
```

Ejemplo 2: contador de llamadas a métodos y constructores

```
public aspect CountAspect {
    static int getCount = 0;
```

```
    static int getCountInToString = 0;
    static int setCount = 0;
    static int numObject = 0;

    public static String numeroDellamadas(){
       String s = "getCount = "+getCount+ ", "+
       "getCountInToString = "+getCountInToString+ ", "+
       "setCount = "+ setCount+ ", "+
       "numObject = "+numObject;
       return s;
    }
    before(): call(* Punto.get*()) {
       getCount++;
    }
    before(): call(* Punto.set*(*)) {
       setCount++;
    }
    before(): call(* Punto.get*()) && withincode(* toString()) {
       getCountInToString++;
    }
    after(): call(* TestPunto.numeros()){
       Test.mostrar(numeroDellamadas());
    }
    after(): call(Punto+.new(..)){
       numObject++;
    }
}
```

Ejemplo 3: Restricción de interacciones entre módulos

```
public aspect PermittedInteractions {
       pointcut sysoutAccess() : get(* System.out);
       pointcut inTestModule() : within (Test+);
       declare warning : sysoutAccess() && !inTestModule() :
    "No es conveniente acceder a System.out fuera de la clase Test";
}
```

Ejemplo 4: introducción de precondiciones

```
public aspect PuntoPrecondition {

       pointcut modifyXPunto(Double x): call(* Punto.setX(Double)) && args(x);

       pointcut modifyYPunto(Double y): call(* Punto.setY(Double)) && args(y);

       pointcut createPunto(Double x, Double y):
            call(Punto+.new(Double,Double)) && args(x, y);

       before(Double x): modifyXPunto(x) {
            if(x<0) throw
               new RuntimeException("La coordenada X no puede ser negativa");
    }
```

```
        before(Double y): modifyYPunto(y) {
            if(y<0) throw
              new RuntimeException("La coordenada X no puede ser negativa");
        }

        before(Double x, Double y): createPunto(x,y) {
            if(x<0 || y<0)
          throw new RuntimeException("Las coordenadas no pueden ser negativas");

        }
}
```

Ejemplo 5: un **módulo general de traza**

```
import java.io.PrintStream;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.reflect.*;

public abstract aspect AbstractTrace {

            abstract pointcut classes();
            abstract pointcut constructors();
            abstract pointcut methods();
            abstract pointcut exceptions();

            before(): classes() && constructors() {
               doTraceEntry(thisJoinPoint, true);
            }
            after(): classes() && constructors() {
               doTraceExit(thisJoinPoint,  true);
            }

            before(): classes() && methods() {
               doTraceEntry(thisJoinPoint, false);
            }
            after(): classes() && methods() {
               doTraceExit(thisJoinPoint,  false);
            }

            before(): classes() && exceptions() {
               printIndent();
               getStream().print("--> ");
               getStream().print(thisJoinPoint);
               getStream().println();
            }

            private PrintStream stream = System.out;

            private Integer callDepth = 0;

            public void initStream(PrintStream _stream) {
               setStream(_stream);
            }

            private PrintStream getStream() {
```

```
        return stream;
    }
    private void setStream(PrintStream s) {
        stream = s;
    }
    private int  getCallDepth() {
        return callDepth;
    }
    private void setCallDepth(int n) {
        callDepth = n;
    }

    private void doTraceEntry (JoinPoint jp, boolean isConstructor) {
        setCallDepth(getCallDepth() + 1);
        printEntering(jp, isConstructor);
    }

    private void doTraceExit (JoinPoint jp,  boolean isConstructor) {
        printExiting(jp, isConstructor);
        setCallDepth(getCallDepth() - 1);
    }

    private void printEntering (JoinPoint jp, boolean isConstructor) {
        printIndent();
        getStream().print("--> ");
        getStream().print(jp);
        printParameters(jp);
        getStream().println();
    }

    private void printExiting (JoinPoint jp, boolean isConstructor) {
        printIndent();
        getStream().print("<--  ");
        getStream().print(jp);
        printParameters(jp);
        getStream().println();
    }

    private void printParameterTypes(JoinPoint jp) {
            Class<?>[]  ptypes =
                ((CodeSignature)jp.getSignature()).getParameterTypes();

            getStream().print("(");
            for (int i = 0; i < ptypes.length; i++) {
                  getStream().print(ptypes[i].getName());
                  if (i < ptypes.length - 1) getStream().print(", ");
            }
            getStream().print(")");
    }

    private void printIndent() {
        for (int i = 0; i < getCallDepth(); i++)
            getStream().print(" ");
    }

    private void printParameters(JoinPoint jp) {
        Class<?>[]  ptypes =
                ((CodeSignature)jp.getSignature()).getParameterTypes();
        String[] pnames =
```

```
                    ((CodeSignature)jp.getSignature()).getParameterNames();
            Object[] params = jp.getArgs();

            getStream().print("(");
            for (int i = 0; i < ptypes.length; i++) {
                  getStream().print(ptypes[i].getName() + " " +
                                    pnames[i]           + "=" +
                                    params[i]);
                  if (i < ptypes.length - 1) getStream().print(", ");
            }
            getStream().print(")");
          }
}
```

```
aspect TraceMyClasses extends AbstractTrace {

    pointcut classes(): within(PuntoImpl) || within(Test+);

    pointcut constructors(): execution(new(..));

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

    pointcut exceptions(): handler(Exception+);
}
```

3. Cuestiones

1. Explique el efecto de incluir el aspecto AspectoCuenta en un programa que tiene la clase CuentaBancaria.

```
public class CuentaBancaria {
  private int balance = 0;
  public void depositar (int cantindad ) { balance += cantidad ; }
}
public aspect AspectoCuenta {
   private int contador = 0;
   pointcut pc (): execution (void CuentaBancaria.depositar(int ));
   after (): pc () { contador++; }
}
```

2. Considere el siguiente código Java

```
public class CuentaBancaria {
       private Integer numeroCuenta;
       private Integer balance;

      public CuentaBancaria(Integer numero, Integer balance) {
         numeroCuenta = numero;
         this.balance = balance;
      }

      public void depositar(Integer cantidad) {
           balance += cantidad;
```

```
      }

      public void sacar(Integer cantidad) {
    balance -= cantidad;
      }

      public Integer getBalance() {
    return balance;
      }

      public String toString() {
           return "BankAcount(" + number + ", " + balance + ")";
      }
}
```

¿Cuál de los siguientes poincuts se refiere a la ejecución de los métodos que cambian el balance?.

- pointcut balanceCambiado() : execution (public * *.* (..))
- pointcut balanceCambiado() : execution (public void CuentaBancaria.* (Integer))
- pointcut balanceCambiado() : execution (public * CuentaBancaria.sacar (Integer))
- pointcut balanceCambiado() : execution (public void CuentaBancaria.sacar (Integer))

3. Dado el código responder a las preguntas

```
class FactoriaFigura {

      public static Punto create(int x, int y) { return new Punto(x,y); }

      public static Linea create(Punto p1, Punto p2) { return new
Linea(p1,p2); }

}

interface FiguraElemento {

      void traslada(int dx, int dy);

}

class Linea implements FiguraElemento {

      private Punto p1, p2;

      public Linea(Punto p1, Punto p2){this.p1 = p1; this.p2 = p2;}

      Punto getP1() { return p1; }

      Punto getP2() { return p2; }

      void setP1(Punto p1) { this.p1 = p1; }

      void setP2(Punto p2) { this.p2 = p2; }
```

```
        void traslada(int dx, int dy) { p1.traslada(dx,dy); p2.traslada(dx,dy);
}

}

class Punto implements FiguraElemento {

        private int x = 0, y = 0;

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

        int getX() { return x; }

        int getY() { return y; }

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

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

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

}
```

- Escriba un *pointcut* para describir llamadas a los métodos públicos de la clase Punto.
- Escriba un *pointcut* para describir llamadas a los métodos públicos de la clase Línea que empiecen por set.
- Escriba un aspecto para generar un error cuando se acceda a los constructores de *Punto* o *Línea* fuera de la clase *FactoriaFigura.*
- Escriba un aspecto para generar un error cuando se acceda al atributo *out* de *System* fuera de clases que extiendan la clase *Test*.
- Describa el objetivo del aspecto `TraceAllCalls` siguiente.

```
public aspect TraceAllCalls {

        pointcut pointsToTrace() : call(* *.*(..)) && !within(TraceAllCalls);

        before() : pointsToTrace() {

                System.err.println("Enter " + thisJoinPointStaticPart);

        }

        after() : pointsToTrace() {

                System.err.println("Exit " + thisJoinPointStaticPart);

        }

}
```

- Porqué es necesario `&& !within(TraceAllCalls);`

4. Ejercicios

Ejercicio 1.

- Escribir un aspecto que escriba en un fichero una línea de información cada vez que un método es llamado (con el tiempo en que es llamado y sus parámetros reales) y otras cada vez que la llamada se ha completado. Asegúrese de que el mensaje dice si el método se completa con normalidad o a través de una excepción.
- Escribir un aspecto que escriba en un fichero una línea de información cada vez que se accede o se modifica un atributo dado.

Ejercicio 2.

- Escribir un aspecto que asegure en tiempo de ejecución que los objetos de la clase A sólo se pueden crear en métodos de la clase B.

Ejercicio 3.

Un coeficiente binomial $bc(n,m) = \binom{n}{m}$ viene definido por las propiedades

$$bc(n,m) = \begin{cases} 1, & m = 0 \\ 1, & m = n \\ bc(n-1,m-1) + bc(n-1,m), & m \neq n, m \neq 0 \end{cases}$$

- Escribir un método recursivo que lo calcule
- Escribir un aspecto que cuente el número de llamadas recursivas
- Escribir un aspecto que utilice los valores previamente calculados, en las llamadas recursivas anteriores, en vez de volver a evaluarlos
- Escriba un aspecto que calcule el tiempo que tarda en ejecutarse un método. Usar ese aspecto para medir la diferencia de tiempo de ejecución si usamos los valores precalculados o no.
- Escribir un aspecto que realice la traza de las llamadas al método