UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CONTROL DE REENTRANCIA DE ASPECTOS EN ASPECTJ

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

CARLOS SEBASTIÁN CABRERA HORMAZÁBAL

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
JOHAN FABRY
JOSÉ PIQUER GARDNER

SANTIAGO DE CHILE
MAYO 2010

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN
POR: CARLOS CABRERA H.
FECHA: 24/05/2010
PROF. GUÍA: Sr. ÉRIC TANTER

"CONTROL DE REENTRANCIA DE ASPECTOS PARA ASPECTJ"

La programación orientada a aspectos (POA) es un paradigma de programación. Permite encapsular funcionalidad que se encuentra dispersa en un sistema. Para ello utiliza *pointcuts*, predicados que definen eventos del programa, y *advices*, el código que es ejecutado en los eventos definidos por un pointcut. Un aspecto es una entidad que agrupa pointcuts y advices.

AspectJ es un lenguaje de programación para POA. Está diseñado como una extensión de Java, de forma que cualquier programa Java es también un programa AspectJ válido. Además del compilador oficial del proyecto AspectJ existen otros, de los cuales AspectBench Compiler (abc) es el más avanzado.

La reentrancia de aspectos ocurre cuando la ejecución de un aspecto desencadena nuevamente su propia ejecución; produciéndose bucles infinitos. Actualmente la reentrancia se soluciona utilizando chequeos y patrones adhoc.

La introducción de niveles de ejecución evita la reentrancia de aspectos. La ejecución del programa se separa en distintos niveles. Por defecto, la computación base ocurre en el nivel 0, mientras que los aspectos que observan esta ejecución se ubican en el nivel 1. La ejecución en el nivel 1 sólo puede ser observada desde el nivel 2, y así sucesivamente. Esta estructura para la ejecución de los programas soluciona casi todos los casos de reentrancia. Para el caso faltante, se utiliza un mecanismo adicional de control de reentrancia.

Para esta memoria se extendió el compilador abc para incorporar una adaptación de niveles de ejecución. El lenguaje soportado por el compilador extendido incorpora nueva sintaxis para ello. Y los programas compilados contienen rutinas adicionales que agregan la estructura de niveles de ejecución y el control de reentrancia. Además, es posible controlar el nivel de ejecución en que se ejecutará una expresión, si fuese necesario.

Se hicieron distintas pruebas para validar el trabajo realizado. Se confeccionaron tests para las distintas funcionalidades que, en conjunto, implementan niveles de ejecución. También se verificó la correcta compilación y ejecución de AJHotDraw, un framework para interfaces gráficas de programas de dibujo. Adicionalmente se probó el compilador con RacerAJ, una herramienta para la detección de *data races* implementada en AspectJ. RacerAJ es de interés porque incorpora pointcuts para evitar la ocurrencia de reentrancia de aspectos; removidos estos pointcuts, el programa funciona correctamente al ser compilado con esta versión extendida de abc.

Además se realizó un ligero análisis de performance para medir el impacto en los programas compilados. Para ello se utilizó una *suite* de *benchmarks* para AspectJ. Se compararon los tiempos de ejecución logrados al utilizar el compilador desarrollado y la versión original.

Índice General

Índice de figuras

Capítulo 1

Introducción

En esta memoria se presentará el trabajo realizado para implementar control de reentrancia al lenguaje AspectJ [12]. El capítulo 2 introduce conceptos que serán utilizados durante el resto de la memoria. El capítulo 3 muestra los problemas que se resuelven con el trabajo y expone razones que respaldan la decisión de escoger AspectJ como el lenguaje de implementación. El capítulo 3.3 explica la semántica presentada en [1], que es la que se ha implementado en este trabajo. En el capítulo 4 se detalla cómo se realizó la implementación en AspectJ: las clases implementadas, herramientas involucradas, una breve descripción del compilador utilizado y la sintaxis nueva para el lenguaje. Las pruebas realizadas están disponibles en el capítulo 5. Finalmente, las conclusiones y direcciones posibles para nuevo trabajo están en el capítulo 6.

Capítulo 2

Trasfondo: AOP/AspectJ

2.1. Programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación utilizado ampliamente, y gran parte de los lenguajes modernos lo siguen. Una de las ideas principales es la agrupación de los datos y las operaciones que se realizan sobre ellos en entidades únicas. Estas entidades son llamadas *objetos*, y a menudo son modelados a partir de entidades existentes en el mundo real.

Una característica importante de los objetos es que permiten establecer relaciones de especialización entre ellos. Con esto es posible establecer jerarquías, y así representar las relaciones entre entidades observadas en la realidad, como por ejemplo, la taxonomía de los animales o los distintos tipos de usuarios de un sistema y sus roles.

Para definir las características de los objetos a menudo se utilizan clases[1]. En ellas se definen las características en común de un grupo de objetos, como las variables que definirán su estado o los métodos que poseerán. Las clases cumplen el rol de un molde, con el cual se pueden generar nuevos objetos.

2.2. Java y la máquina virtual de Java

Java es un lenguaje de programación con una sintaxis similar a la del lenguaje C, creado en *Sun microsystems*. Java fue diseñado siguiendo el paradigma de programación orientada a objetos y puede ser utilizado prácticamente en todas las plataformas de la actualidad.

Los programas escritos en Java se compilan a un *código de máquina* intermedio llamado *bytecode*. Las instrucciones de este código corresponden a las definidas para una *máquina*

[1]Existen lenguajes que no utilizan este concepto, como los lenguajes orientados a objetos con prototipos.

virtual, llamada la *máquina virtual de java*.

El que exista este paso intermedio entre el código Java y la ejecución en un computador ha posibilitado la aparición de múltiples lenguajes experimentales. Estos permiten aprovechar la infraestructura existente en los dispositivos capaces de ejecutar el código de la máquina virtual de Java, así como también aprovechar las librerías que acompañan al lenguaje. Junto con lo anterior, en muchos casos es posible acceder desde estos lenguajes a entidades, como clases y métodos, definidas en código fuente Java una vez que estas ya están compiladas en bytecode.

2.3. Programación orientada a aspectos

A menudo aparecen problemas durante la construcción de sistemas de software al tratar de incorporar funcionalidades que involucran varias componentes de este al mismo tiempo. Un ejemplo de esto es cuando se quieren ofrecer facilidades para la autenticación y el control de acceso a ciertos recursos ofrecidos por el software, de modo de detener accesos no deseados a módulos del sistema. Para esto es necesario que los controles de seguridad sean codificados en el mismo lugar que el código que provee la funcionalidad, de modo de poder negar el acceso cuando sea necesario. Entonces, al implementar todos estos chequeos, el código queda repartido en distintos módulos del software. Por lo anterior, es imposible encapsular esta funcionalidad en particular dentro de un sólo módulo.

La Programación Orientada a Aspectos (POA) pretende ser una solución para problemas como el del ejemplo anterior, proveyendo una manera para encapsular la funcionalidad dispersa. De esta forma, el código que implementa la funcionalidad queda reunido en módulos separados del resto de la aplicación, lo que facilita que sea reutilizado y/o modificado.

La POA introduce el concepto de *aspecto*, un ente que agrupa: (a) un predicado que identifica ciertos momentos en la ejecución del programa y (b) código a ejecutar en estos momentos. En este modelo, la especificación de un conjunto de momentos relevantes en la ejecución de un programa es llamado un *pointcut*, y el código que se quiere ejecutar en estos momentos es llamado *advice*. Cada uno de los instantes en la ejecución del programa que puede ser seleccionado por un pointcut es llamado un *join point*.

En el ejemplo, utilizando pointcuts es posible entonces expresar los momentos en que es necesario el control de acceso. Luego, el código que chequea la autorización puede ser escrito

en un advice que es ejecutado en los momentos que el pointcut especifica. Es decir, toda la funcionalidad necesaria puede ser encapsulada dentro de un aspecto.

2.4. AspectJ

AspectJ es una extensión a Java para POA. Actualmente, AspectJ es el lenguaje para POA más utilizado.

Cada programa Java es un programa AspectJ válido, pero dado que AspectJ define nueva sintaxis y artefactos (de los cuales el más relevante es la declaración de aspectos), los programas AspectJ no necesariamente son programas Java válidos. El compilador de AspectJ compila a *bytecode*, y este requiere para su ejecución la inclusión de una pequeña librería en tiempo de ejecución.

En un programa AspectJ se puede definir una construcción llamada *aspect*, dentro de la cual se pueden definir *pointcuts* y *advices*.

En el siguiente código de ejemplo se puede apreciar la definición de un pointcut que selecciona los momentos en que se abra un archivo:

```
1  pointcut fileOpen() : call( FileInputStream.new(..) );
```

El siguiente código de ejemplo muestra la definición de un advice que se ejecutará antes de los join points especificados por el pointcut `fileOpen()` . Este advice ejecutará un chequeo de privilegios implementado en la clase `SecurityManager`:

```
1  FileInputStream around() : fileOpen() {
2
3    if (SecurityManager.checkFileOpenPermission()){
4      return proceed();
5    }else{
6      System.err.println("Permiso negado para abrir archivo.");
7      return null;
8    }
9  }
```

Como el advice debe ser ejecutado alrededor de los join points de creación de un objeto de tipo FileInputStream, debe ser declarado su tipo de retorno con un tipo adecuado (en el ejemplo, FileInputStream).

El advice realiza el chequeo de permisos, y en caso de que estos existan permite la creación del objeto. En caso contrario, imprime un mensaje y devuelve `null` en reemplazo del objeto.

Finalmente, la definición de un aspecto simple compuesto por el pointcut y advice definidos arriba podría ser como sigue:

```
1   public aspect FileOpenPermissionEnforcer{
2
3     pointcut fileOpen() : call( FileInputStream.new(..) );
4
5     FileInputStream around() : fileOpen() {
6
7       if (SecurityManager.checkFileOpenPermission()){
8         return proceed();
9       }else{
10        System.out.println("Permiso negado para abrir archivo.");
11        return null;
12      }
13    }
14  }
```

Con el compilador de AspectJ, estos aspectos pueden ser incorporados a un sistema ya compilado en bytecode, lo que permite agregar funcionalidad a este sin tener acceso al código fuente original ni recompilar todo el sistema.

Junto con el compilador, el proyecto AspectJ provee un depurador y plugins para algunos IDE (en español, Entorno Integrado de Desarrollo).

2.4.1. Join points en AspectJ

El modelo de join point de AspectJ considera varios puntos en la ejecución de un programa Java.

En AspectJ, se puede acceder a un objeto que representa el join point mientras se esté ejecutando un advice. Este objeto, llamado `JoinPoint`, puede ser mediante la referencia especial `thisjoinpoint`. Ésta contiene información sobre el tipo de join point sobre el cual actúa el advice como el objeto que está actualmente en ejecución o la posición dentro del código fuente a la cual corresponde [2].

Los objetos de tipo `JoinPoint` también exponen información de contexto del join point sobre el cual el advice actual esté actuando. Los tipos de join point que se pueden encontrar están descritos en las primeras dos columnas de la tabla en el cuadro 2.1.

[2]En otros lenguajes de POA, como en AspectScheme, los join points pueden ser manipulados directamente. En AspectScheme, por ejemplo, dentro del código de un pointcut están disponibles como un stack de join points.

Tipo de join point	descripción	pointcuts asociados
INITIALIZATION	Inicialización de un objeto	`initialization()`
PREINITIALIZATION	Justo antes de la inicialización de un objeto	`preinitialization()`
STATICINITIALIZATION	Ejecución de los inicializadores estáticos[a] , de existir alguno	`staticinitialization()`
ADVICE_EXECUTION	La ejecución de un advice	`adviceexecution()`
CONSTRUCTOR_CALL	Invocación de un constructor	`call()`
CONSTRUCTOR_EXECUTION	Ejecución del cuerpo correspondiente a un constructor declarado en una clase	`execution()`
EXCEPTION_HANDLER	Ejecución del handler del código correspondiente al handler para una excepción	`handler()`
FIELD_GET	Acceso al atributo de un objeto	`get()`
FIELD_SET	Cambio de valor de un atributo de un objeto	`set()`
METHOD_CALL	Invocación de un método de un objeto	`call`
METHOD_EXECUTION	Ejecución del código de algún método de un objeto. Notar que es distinto al join point `METHOD_CALL`, pues este join point ocurre en el contexto del objeto al cual pertenece el método, mientras que el otro en el objeto desde el cual se ha generado la llamada	`execution()`

Cuadro 2.1: Join points y pointcuts en AspectJ

[a]Son bloques de código que se pueden agregar a las clases. Ellos son ejecutados en el momento en que una clase es cargada. Estos bloques no son parte de ningún método del objeto.

2.4.2. Pointcuts en AspectJ

Los pointcuts básicos de AspectJ están estrechamente ligados a los join points disponibles. En la tercera columna de la tabla en el cuadro 2.1 están descritos los pointcuts correspondientes a los join points existentes.

Los pointcuts pueden ser declarados dentro de un aspecto y asignárseles un nombre. Alternativamente, pueden aparecer como parte de la declaración de un advice. Un pointcut nuevo puede ser formado como una combinación de pointcuts básicos y de otros pointcuts ya declarados en un aspecto. Para combinar los pointcuts se pueden utilizar las operaciones lógicas: AND, OR y NOT.

Algunos de los pointcuts de AspectJ sirven para exponer información presente el ambiente durante su evaluación. Utilizando estos pointcuts es posible obtener referencias al objeto actualmente en ejecución (*this()*), al objeto del cual se está invocando un método(`target()`) y a los argumentos con los que se ha llamado a un método(`args()`). Estos *bindings* pueden ser utilizados dentro del advice.

Los pointcuts más utilizados son los que sirven para seleccionar los momentos en que se ejecuta (`execution()`) o se realiza la invocación de un método (`call()`). Otro pointcut muy utilizado es el que permite seleccionar todos los join points que se encuentren dentro del flujo de control de otro join point (`cflow()` y `cflowbelow()`). Para especificar los join points a los cuales se les debe examinar su flujo de control se puede utilizar cualquier pointcut válido en AspectJ. Este otro pointcut es pasado como parámetro.

Existe también un pointcut especial que permite utilizar el lenguaje Java mismo para definir un pointcut (`if()`). Como argumento para este pointcut es posible utilizar cualquier expresión Java válida que pueda ser evaluada como un boolean.

Por último, está disponible un pointcut especial que permite seleccionar join points basados en la posición del código fuente que los genera (`within()`).

2.4.3. Advices en AspectJ

Los advices en AspectJ pueden ser declarados dento de un aspecto. Se indican los momentos en los que se ejecutará asociandolos a un pointcut, ya sea alguno definido anteriormente o uno declarado en linea junto con el advice.

En aspectJ[3], un *advice* puede ser definido para ser ejecutado antes, después o alrededor del join point seleccionado. Alrededor quiere decir que el advice reemplazará la ejecución del join point, con la opción de resumirla mediante la invocación del método especial `proceed()`. Un advice que se ejecuta alrededor de un join point tiene además acceso a el valor de retorno de un join point (de existir este valor) una vez que ha terminado su ejecución.

Adicionalmente, en un advice definido para ejecutarse después (de un join point) es posible obtener el valor de retorno y la excepción arrojada por un método. Para lograr lo primero el advice se declara como `after returning` y para lo segundo como `after throwing`.

[3]Como también en la mayoría de los lenguajes POA.

Capítulo 3

Motivación

Un posible problema al emplear POA es la aparición de *reentrancia de los aspectos.* En términos informales, la reentrancia es que un aspecto se active más veces de lo esperado. Esto podría generar un programa que no cumple con lo diseñado, en caso de que la reentrancia no haya sido planeada, o incluso hacer que este entre en un bucle infinito.

Reentrancia

La reentrancia ocurre cuando la ejecución del código de un aspecto genera, directa o indirectamente, un join point que luego es calzado por un pointcut del mismo aspecto[1].

3.1. Reentrancia en Aspectos y bucles infinitos

La reentrancia en aspectos puede producir búcles debido a la ejecución de un advice o la evaluación de un pointcut.

Para explicar cada caso, se utilizará el siguiente ejemplo: objetos que representan puntos que pueden ser movidos en el espacio. Luego se definirá un aspecto que monitoree la actividad de estos puntos.

A continuación está el código para el objeto *Point* a utilizar:

```
1  public class Point{
2
3     int x,y;
4
5     public Point(int x, int y) { this.x = x; this.y = y; }
6
7     public int getX() { return x;}
8
```

[1]Por simplicidad, consideraremos que existe reentrancia aunque el calze de este pointcut no tenga ningún efecto; como puede ocurrir en AspectJ cuando el pointcut no está asociado a ningún advice.

```
9       public int getY() { return y;}
10
11      public void setX(int x) { this.x = x; }
12
13      public void setY(int y) { this.y = y; }
14
15      public void moveTo(int x, int y) { setX(x); setY(y); }
16
17      public String toString(){ return getX() + "," + getY(); }
18
19      public boolean isInside( Area a ){
20       /*codigo para chequear si el punto esta dentro del area*/
21      }
22
23    }
```

Un punto está definido por dos coordenadas con *getters* y *setters* para cada una de ellas. Además posee métodos para moverlo (`moveTo`) , recuperar una representación del punto como cadena de caracteres (`toString`) y determinar si éste se encuentra dentro de un área específica (`isInside`).

3.1.1. Bucles infinitos causados por la ejecución de un advice

Es posible que la ejecución de un advice cause la aparición de un join point tal que, este es seleccionado por el pointcut asociado al advice. Esto comienza una nueva ejecución del advice, derivando en un búcle infinito.

Para ilustrar esta situación, utilizaremos un aspecto que imprime un mensaje cuando un punto tiene actividad:

```
1  public aspect Activity{
2
3     // Pointcut que selecciona los metodos de Point
4     pointcut active(Point p): execution(* Point.*(..)) && this(p);
5
6     before(Point p): active(p) {
7       System.out.println(
8        "Se ha producido actividad en el punto:"+p);
9     }
10 }
```

El aspecto contiene un pointcut (`active()`) que selecciona los join points correspondientes a la ejecución de cualquier método de un punto. Y un advice que se ejecuta antes de cada uno de estos join points.

Cuando se trata de imprimir el punto `p` en el advice, se invoca implícitamente para ello el método `toString`. Pero este método es también uno de los métodos de un objeto de tipo

`Point` por lo que el pointcut `active()` selecciona su ejecución nuevamente, con lo que se vuelve a ejecutar el advice. Más aún, volverá a ocurrir lo mismo en cada activación posterior del aspecto.

3.1.2. Bucles infinitos causados por la evaluación de un pointcut

Es posible que en la declaración de un pointcut `if` se incluya código arbitrario, cuya ejecución cause la generación de un join point que sea evaluado por el mismo pointcut. Cuando la segunda evaluación del pointcut vuelve a llegar al punto en que se produjo este join point, se generará otro idéntico, lo que deriva en un bucle infinito.

Supongamos ahora que sólo nos interesa que se muestre la actividad de los puntos que se encuentren ubicados dentro de una cierta área. El aspecto encargado de monitorear estos puntos podría ser implementado como sigue:

```
1  public aspect Activity {
2
3    Area area = /* referencia a un area */
4
5    //Pointcut que selecciona los metodos de Point de interes
6    pointcut activeAndInArea(Point p):
7      execution(* Point.*(..)) && this(p) && if(p.isInside(area));
8
9    before(Point p): activeAndInArea(p)
10     {
11       System.out.println(
12         "Se ha producido actividad en un punto dentro del area");
13     }
14 }
```

Para trabajar sólo con los puntos que están en el área de interés se agregó al pointcut original una condición adicional. Esta está expresada utilizando un pointcut `if()`: el método `isInside()` debe devolver verdadero para esta área.

La reentrancia ocurre durante la evaluación del pointcut `activeAndInArea()`. La ejecución del método `isInside()` produce un join point que es nuevamente evaluado por el pointcut `activeAndInArea()`. Esta nueva evaluación luego produce un joinpoint al ejecutar el método `isInside()` que trata de ser evaluado por el pointcut `activeAndInArea()`, y así sucesivamente en cada intento de evaluación posterior.

3.2. Soluciones actuales a los problemas de reentrancia en AspectJ

Situaciones como las anteriores pueden ocurrir al programar utilizando AspectJ. El evitar la ocurrencia de estos errores al programar aspectos le impone una carga adicional al programador que no tiene que ver con el quehacer de escribir adecuadamente el programa, sino más bien con una semántica poco apropiada del mismo AspectJ.

Actualmente, para controlar la reentrancia de aspectos se utilizan patrones para definir pointcuts, los cuales se valen de `cflow()` y `adviceexecution()`. Para evitar la reentrancia los patrones deben ser introducidos en los pointcuts adecuados a conciencia, lo que sólo es posible luego de que quien esté implementando se haya percatado de que existe un problema cuya causa es la reentrancia.

Para remediar la situación del ejemplo utilizado en la subsección 3.1.1, el programador debe conseguir que el pointcut asociado al advice no calce los join points generados por la evaluación del advice mismo. Esto se suele implementar en la práctica utilizando la construcción `!cflow(within(A))`, en donde `A` es el aspecto reentrante. Este pointcut excluye a todos los join points en el flujo de control de cualquier join point que esté léxicamente en el aspecto `A`. El ejemplo reescrito para incorporar el ajuste al pointcut quedaría así:

```
1  public aspect Activity{
2
3     // Pointcut que selecciona los metodos de Point
4     pointcut active(Point p): execution(* Point.*(..)) && this(p);
5
6     before(Point p): active(p) && !cflow(within(Activity)){
7       System.out.println(
8        "Se ha producido actividad en el punto:"+p);
9     }
10 }
```

A diferencia de como ocurre con el ejemplo de la subsección 3.1.1, es imposible evitar el problema del ejemplo de la subsección 3.1.2 manteniendo toda la lógica correspondiente a la selección de join points dentro del pointcut. Esto es debido a que en las implementaciones actuales de AspectJ los join points que aparecen léxicamente en la definición de un pointcut quedan escondidos. Por ello, no es posible evitar que se produzca el error escribiendo un pointcut que indique que se deben ignorar las llamadas producidas en el flujo de control de éste join point, pues no es posible referirse al join point.

Es posible obtener el comportamiento correcto implementando la lógica de "detección del

punto en el área" como una condición al principio del cuerpo del advice; pero esto no es correcto, debido a que se estaría mezclando dentro del advice tanto lógica que es propia de él como lógica perteneciente al pointcut.

3.3. Implementación en AspectJ

Se ha escogido implementar niveles de ejecución en AspectJ por ser este lenguaje para AOP más utilizado.

Se espera que esta implementación sirva para probar los conceptos desarrollados en una mayor cantidad de aplicaciónes. Además, dada la comunidad de desarrollo en torno a AspectJ y la cantidad y tamaño de las aplicaciones, es de esperar también que se podrá apreciar la utilidad y correctitud de las ideas generadas aplicándolas a proyectos que sean interesantes. Se espera encontrar proyectos que cuenten con código que implemente algoritmos lo suficientemente complejos como para estresar las ideas expuestas, revelando posibles oportunidades de mejora. Y que el contar con una implementación en AspectJ posibilite que, de ser suficientemente atractivo el modelo, este se masifique.

3.4. Niveles de ejecución para POA

La semántica que se expondrá en este capítulo es un subconjunto de [1].

3.4.1. Descripción

El código base ejecuta por defecto en nivel 0. Los join points que se generan como consecuencia de la ejecución que ocurre en el nivel 0 aparecen en el nivel 1. Los join points en el nivel 1 pueden ser calzados por pointcuts, y desencadenar la ejecución de advices. Estos advices ejecutarán en el nivel 1 también, y generarán join points en el nivel 2.

En la sección 3.1 se utilizó un ejemplo con puntos. Considerar la clase `Point` de aquel ejemplo junto con este nuevo aspecto que registra el tiempo en que cada punto se ha movido.

```
1  public aspect Activity{
2
3     pointcut moving(Point p): call(* Point.moveTo(..)) && this(p);
4
5     before(Point p): moving(p) {
6       logMovement( p );
7     }
8
9     protected static void logMovement(Point p){
10      /*  codigo para registrar la actividad del punto  */
11    }
12 }
```

Antes de cada invocación del método para mover un punto (`moveTo`), el advice llama al método `logMovement`; y este registra la actividad del punto. Una vez registrado el cambio de posición, se continúa con la ejecución del método `moveTo`. En la figura 3.1 se muestran algunos join points y los niveles en los que se generan.

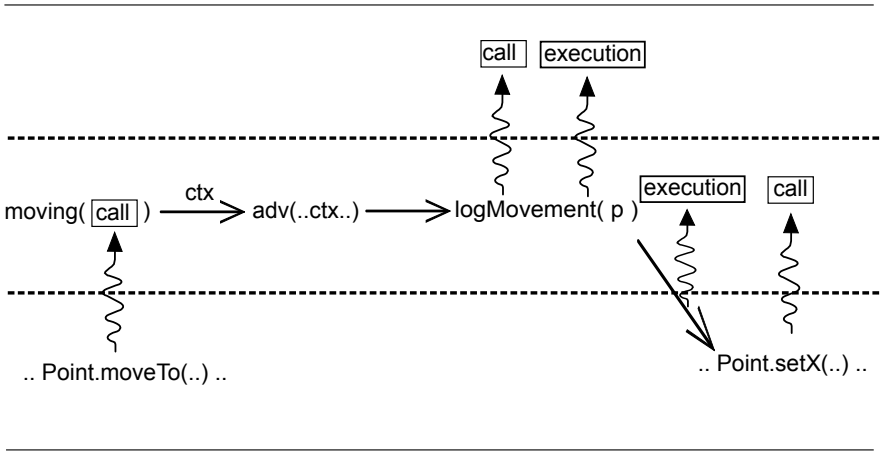

Figura 3.1: Cambio de posición de un punto.

Around advice y `proceed`

Como se mencionó en la subsección 2.4.3, un advice puede ser ejecutado antes, después o alrededor de un join point. En un advice around se puede continuar con la ejecución del join point original mediante la invocación a `proceed`.

El siguiente ejemplo es una versión modificada del anterior. El advice ahora se escribió para ser ejecutado alrededor, así que para mantener el mismo comportamiento incluye un proceed.

```
1  public aspect Activity{
2
3     pointcut moving(Point p): call(* Point.moveTo(..)) && this(p);
4
5     void around(Point p): moving(p) {
6       logMovement( p );
7       proceed();
8     }
9
10    protected static void logMovement(Point p){
11      /*  codigo para registrar la actividad del punto  */
12    }
13 }
```

Cuando se ejecuta `proceed()` dentro de un around advice la ejecución del join point anterior es resumida, pero con el esquema presentado ocurrirá que este join point se ejecutará en el mismo nivel que el advice. Esto es incorrecto, pues en tal caso el comportamiento del programa base se verá influenciado por la existencia del advice: el nivel de la ejecución en el que se ejecuta dicho join point será un nivel mayor o menor dependiendo de la presencia o no del advice.

En el ejemplo, la ejecución del cuerpo del método `moveTo` se realiza en un nivel incorrecto. La figura 3.2 muestra esta situación.

Se deben realizar cambios de niveles apropiados antes de suspender la ejecución de un advice, debido a un `proceed()`, y al reanudarla. Inmediatamente antes de la ejecución del join point original se debe disminuir el nivel, y antes de retomar la ejecución del advice se debe aumentar el nivel.

En la figura 3.3 se puede ver el ejemplo con el cambio de nivel que se debe hacer. Notar que sólo se ve el descenso de nivel descrito; si el advice tuviera más instrucciones luego del proceed, la ejecución volvería al nivel superior.

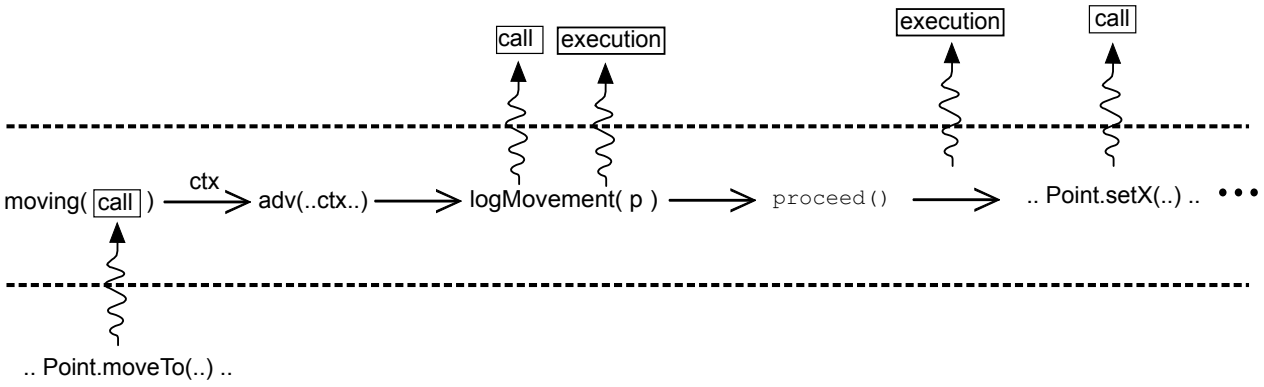

Figura 3.2: Comportamiento no deseado de un `proceed()`.

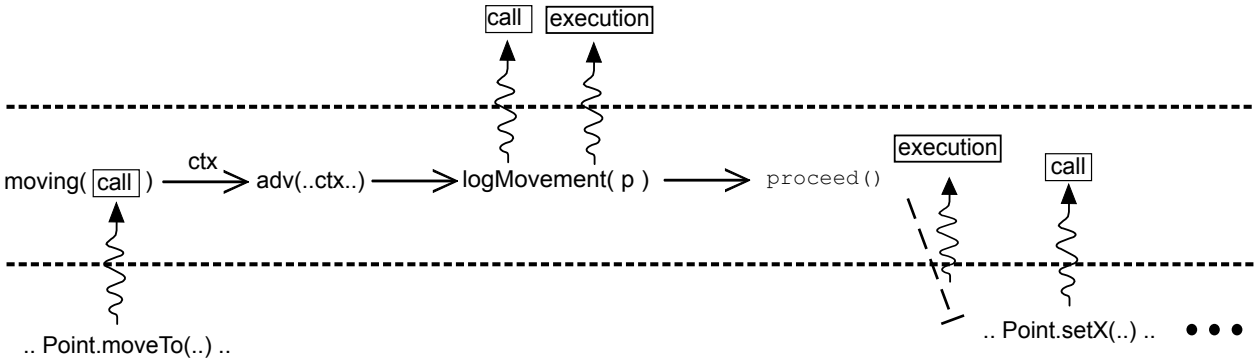

Figura 3.3: Bajar nivel antes de continuar con la ejecución del programa base.

Operadores `up` y `down`

Para permitir el control explícito del nivel de ejecución, se introducen los operadores `up` y `down` [1]. Estos cambian el nivel de ejecución al nivel inmediatamente superior o inferior, respectivamente, con respecto al nivel actual.

En el código siguiente, un método es ejecutado en dos niveles de ejecución distintos, por lo que los join points producidos en cada caso aparecen en niveles distintos. La sintaxis utilizada para el operador `up` es ficticia, pensada para facilitar las explicaciones en este capítulo.

```
1  class UpExample{
2
3    static public void main(String[] args){
4      foo(0);
5
6      up {
7          foo(0);
8      }
```

```
9     }
10
11    int foo(int n){
12      return n;
13    }
14  }
```

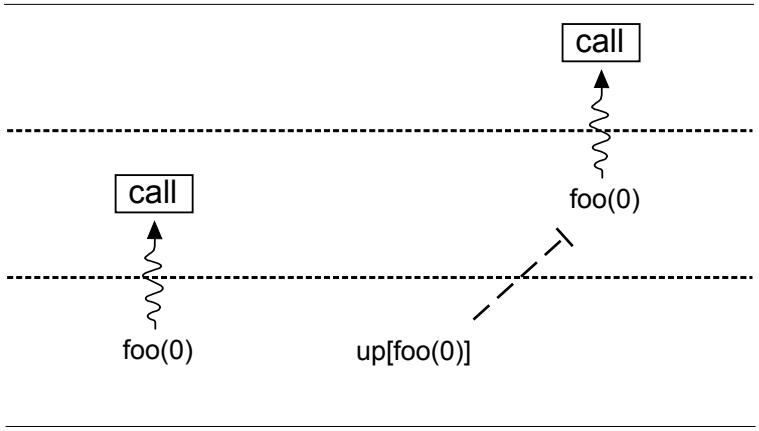

Figura 3.4: Operador up.

En la figura 3.4 se muestran los join points producidos.

Estos operadores pueden ser utilizados para cambiar de nivel de ejecución de cualquier computación, incluso dentro de un pointcut o un advice.

Desplegar aspectos en un nivel distinto

En AspectJ, basta con declarar un aspecto para que este sea aplicado. No existe un mecanismo de *deployment* que permita activar y desactivar selectivamente aspectos durante la ejecución del programa. Por ello, se utilizará una sintaxis ficticia para representar el deployment de aspectos. El listado de código siguiente muestra el deployment de un aspecto utilizando esta sintaxis.

```
1   aspect LogCalls{
2
3     pointcut call() : call( * *.*(..) );
4
5     before(obj) : call() && target(Object obj) {
6       Logger.info( "llamada al objeto: "+ obj );
7     }
8
9   }
10
11  class DeployExample{
12
13    static public void main(String[] args){
14      deploy new LogCalls();
```

```
15
16      (new DeployExample).bar();
17    }
18
19    void bar(){
20      /* ... */
21    }
22  }
```

Como se mencionó anteriormente, el código se ejecuta por defecto en el nivel 0. Al ejecutar `deploy` en el nivel 0, el aspecto es desplegado en el nivel 1. Este aspecto en el nivel 1 observará la ejecución ocurrida en el nivel 0, o lo que es lo mismo, su pointcut sólo podrá calzar join points en el nivel 1. De forma general, un aspecto que está desplegado en el nivel $n + 1$ observa la ejecución que ocurre en el nivel n.

Los aspectos pueden ser desplegados en un nivel superior cambiando el nivel en que ejecuta `deploy`. Para ello se puede utilizar el operador `up`. Con él se puede cambiar la ejecución de `deploy` del nivel n al nivel $n + 1$, con lo que el aspecto, en vez de ser desplegado en el nivel $n + 1$, es desplegado en el nivel $n + 2$.

En el listado de código siguiente aparece el ejemplo modificado para desplegar el aspecto en el nivel 2. Ahora la instancia del aspecto observa los join points correspondientes a la ejecución del nivel 1. Al ejecutarlo, no se imprimiría ningún mensaje.

```
1   /* ... */
2   class DeployExample{
3
4     static public void main(String[] args){
5       up{
6         deploy new LogCalls();
7       }
8
9       (new DeployExample).bar();
10    }
11
12    void bar(){
13      /* ... */
14    }
15  }
```

De forma análoga, es posible desplegar un aspecto en un nivel inferior al que se obtiene por defecto utilizando `deploy` junto con `down`.

3.4.2. Aprovechando los niveles de ejecución

cflow sensible al contexto

Hay pointcuts que además del join point actual, utilizan información del contexto para evaluar. El pointcut `cflow` es uno de ellos. `cflow` inspecciona el stack de ejecución, pero no toma en cuenta el nivel de ejecución en que se produjo cada entrada del stack.

Se introduce un nuevo `cflow`, sensible a la información de los niveles que ahora está disponible en el stack. Este nuevo pointcut `l-cflow` es muy similar a `cflow`; la diferencia es que este sólo considera los join points que pertenecen al mismo nivel de ejecución que el join point actual.

3.5. Implicancias de la semántica

3.5.1. Eliminación de varios casos de reentrancia

La separación de la ejecución en niveles impide casi totalmente que se produzcan problemas de reentrancia de aspectos. Como se dijo al comienzo del capítulo 3, para que se presente reentrancia, es necesario que la ejecución del código de un aspecto genere un join point tal que luego este sea calzado por su pointcut. Pero como los join points generados por la ejecución del advice o del pointcut aparecen en el nivel superior, el pointcut no tendrá acceso a ninguno de ellos; no es posible que el pointcut encuentre un calce de estos puntos.

Retomemos los ejemplos mostrados en las subsecciones 3.1.1 y 3.1.2. En ambos casos, la reentrancia se soluciona al introducir niveles de ejecución. El ejemplo de la subsección 3.1.1 debía la reentrancia a que dentro del flujo de control del advice se generaba un join point para el método `Point.toString()`. Este es calzado por el pointcut `active`, lo que causa rentrancia del aspecto. El aspecto del ejemplo original está a continuación.

```
1  public aspect Activity{
2
3     // Pointcut que selecciona los metodos de Point
4     pointcut active(Point p): execution(* Point.*(..)) && this(p);
5
6     before(Point p): active(p) && !cflow(within(Activity)){
7       System.out.println(
8        "Se ha producido actividad en el punto:"+p);
9     }
10 }
```

El join point de ejecución `Point.String()` que aparece al obtener la representación co-

mo String del punto activado ya no es calzado; ni tampoco ninguno de los sucesivos join points que aparecen luego, como los correspondientes a llamadas y ejecuciones de los métodos `Point.getX()` y `Point.getX()`. Esto es debido a que, si el nivel en el que está desplegado el aspecto `Activity` es $n+1$, los join points que se generan debido a su ejecución aparecen en el nivel $n+2$; lo que implica que estos join points ya no pueden ser calzados por los pointcuts de `Activity`, pues está desplegado en el nivel $n+1$.

El ejemplo de la subsección 3.1.2 está copiado a continuación.

```
1   public aspect Activity {
2
3     Area area = /* referencia a un area */
4
5     //Pointcut que selecciona los metodos de Point de interes
6     pointcut activeAndInArea(Point p):
7       execution(* Point.*(..)) && this(p) && if(p.isInside(area));
8
9     before(Point p): activeAndInArea(p)
10      {
11        System.out.println(
12          "Se ha producido actividad en un punto dentro del area");
13      }
14  }
```

La reentrancia era producida debido a que durante la evaluación de `activeAndInArea` se generan join points de ejecución del método `Point.isInside(Area)`, los cuales volvían a ser evaluados por el mismo pointcut. Con la introducción de niveles de ejecución estos join points no son tomados en cuenta, pues aparecen en el nivel superior.

3.5.2. `down` y control de reentrancia

El ejemplo siguiente es una versión aumentada de uno mostrado en la subsección 3.1.1.

```
1  public aspect Activity {
2     Display display = /* codigo para obtener referencia al display */;
3
4     //Pointcut que selecciona los metodos de Point
5     pointcut active(Point p): execution(* Point.*(..)) && this(p);
6
7     before(Point p): active(p) {
8        System.out.println("Se ha producido actividad en el punto" +p);
9        Display.refresh(p);
10    }
11 }
12
13 public class Display{
14    protected static refresh(Point p){/*...*/}
15    protected static optimizedRefresh(List<Point> pts){/*...*}
16 }
```

El advice del aspecto `Activity` actúa antes de la ejecución de cualquier método de los objetos que representan puntos. Imprime un mensaje que da cuenta de la actividad del punto, tal como el ejemplo original; pero además llama a un método que redibuja una pantalla (`refresh`).

Es importante notar que los métodos `refresh` y `optimizedRefresh` necesitan acceder a las coordenadas de los puntos, y para ello deben realizar llamadas a los métodos `Point.getX()` y `Point.getY()`.

Existe además un aspecto capaz de posponer varias actualizaciones al display con el objetivo de realizarlas luego de forma eficiente, en una sola operación.

```
1  public aspect OptimizeRefresh{
2     Display display = /* codigo para obtener referencia al display */;
3     List<Point> pendingRefreshes = new List();
4
5     //Selecciona las llamadas (ineficientes) para actualizar un display
6     pointcut aRefresh(Point p): call(* Display.refresh(Point p));
7
8     void around(Point p): aRefresh(p) {  pendingRefreshes.add(p) };
9
10    /* ... */
11 }
```

Este aspecto posee un pointcut que selecciona todos los join points correspondientes a las llamadas a `refresh`, el método no óptimo de actualización. El advice asociado guarda la referencia correspondiente al punto que debe ser actualizado. Como el advice es de tipo around, la actualización no es realizada en este momento. Las actualizaciones pendientes acumuladas son realizadas todas juntas utilizando el método eficiente `optimizedRefresh`, por ejemplo, cada intervalos de tiempo regulares utilizando un temporizador. El aspecto `OptimizeRefresh` se encuentra desplegado en el nivel 1 para optimizar actualizaciones al display realizadas en el código base.

Sería deseable que `OptimizeRefresh` se encargara de optimizar también las actualizaciones al display realizadas en `Activity`. Esto se puede lograr usando el operador `down` para que la actualización del display ocurra en el nivel base. Es decir, utilizar `down` en la invocación al método `Display.refresh` (linea 9 del primer listado de código de esta subsección). La utilización de `down` no introduciría ningún caso de reentrancia, la llamada a `refresh` es interceptada correctamente por el aspecto `OptimizeRefresh` y postergada.

Sin embargo, es posible introducir reentrancia de esta forma. Si se disminuye el nivel de ejecución de parte de la ejecución del advice, los join points generados durante la ejecución de

esta parte se verán en el mismo nivel en el cual ocurre la ejecución de una parte del aspecto. Debido a esto, el aspecto podría activarse nuevamente si es que su pointcut calza con alguno de estos join points. Por ejemplo, si se hubiese utilizado `down` en la linea que imprime el mensaje (linea 8 del primer listado de esta subsección). Esto está ilustrado en la figura 3.5.

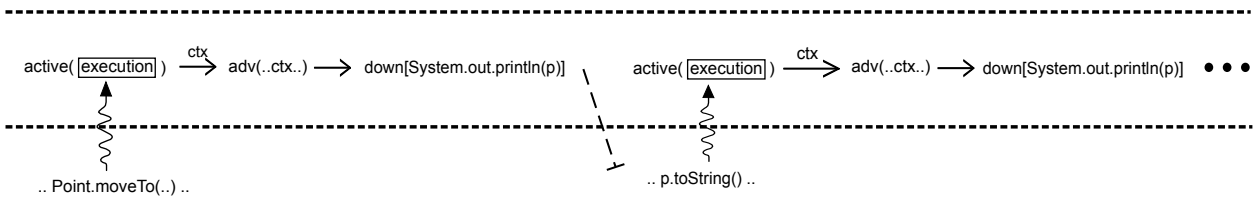

Figura 3.5: Mover la impresión del mensaje al nivel base. El aspecto entra en un búcle infinito

Como se menciona en [1], para evitar la ocurrencia de reentrancia debido a un `down` se requiere introducir un *control de reentrancia*. Antes de la evaluación de cada pointcut de un aspecto se debe chequear que no se esté ya en el flujo de control del mismo; si se detecta esta situación el pointcut no será evaluado, y se considerará que no calza [2].

Volviendo al ejemplo, el control de reentrancia impide el segundo calce del pointcut `active` que se ve a la derecha de la figura 3.5. El join point causante de la reentrancia aún es emitido, y en el mismo nivel, pero no es calzado por el pointcut `Active`. En la figura 3.6 está retratada la ejecución con el control de reentrancia incorporado.

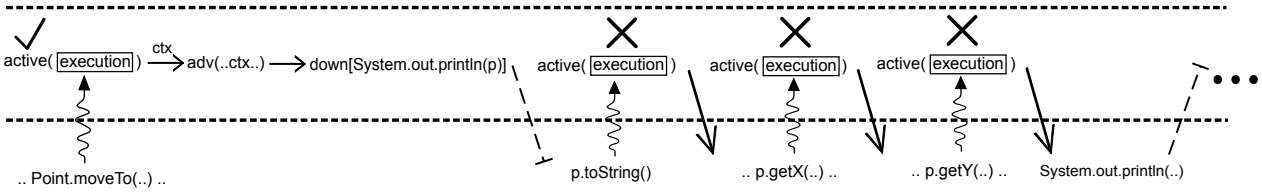

Figura 3.6: Control de reentrancia. El pointcut `active` no calza mientras se esté dentro del flujo de control del aspecto.

[2] En [1] no se explicita esta semántica para el control de reentrancia; fue explicada en conversaciones ocurridas durante el desarrollo del trabajo

Capítulo 4

Implementación

Para implementar la semántica descrita en el capítulo anterior se ha elegido el compilador abc [5]. Este ha sido concebido pensando en que pueda ser extendido con facilidad, por lo que resulta apropiado para este trabajo. Sin duda, hubiese sido factible implementar la semántica expuesta en el capítulo anterior en el compilador del proyecto AspectJ [12] (ajc), pero habría requerido de un período de implementación mucho más prolongado, dado que ajc no fue diseñado para ser extendido.

A pesar de la extensibilidad de abc, gran parte del tiempo dedicado a la implementación fue utilizado en comprender el diseño del compilador y a identificar los lugares que debían ser modificados. Los cambios eran semánticamente simples, pero el compilador debía ser modificado de forma precisa, considerando los efectos que cada cambio produciría en el código compilado.

4.1. Niveles en AspectJ

Es importante hacer hincapié en que hay una sección de la propuesta que no puede ser implementada debido a las características de AspectJ. El no tener funciones de primera clase impide que podamos implementar *level capturing functions* [1] tal como se han diseñado. Una adaptación adecuada a las características de AspectJ es un desafio aún no resuelto.

`deploy`

La forma de declarar aspectos, análoga a la declaración de clases en Java, implica que no existe scoping al desplegar aspectos. Por esto, no es posible desplegar dinámicamente un aspecto en un cierto nivel de ejecución relativo al nivel actual. Para entregar una funcionalidad

similar, se ha implementado un sistema inspirado en *Stratified aspects* [4] que entrega la capacidad de especificar qué nivel de ejecución será afectado por cada aspecto declarado.

En la declaración de un aspecto se puede indicar el nivel de ejecución que debe observar. En caso de que no se especifique ningún nivel, el aspecto observará por defecto los join points correspondientes a la ejecución ocurrida en el nivel 0. El código a continuación muestra la sintaxis de las declaraciones.

```
1 aspect AspectA {} //desplegado en el nivel 1
2
3 onlevel 1 aspect AspectB {} //desplegado en el nivel 1
4
5 onlevel 2 aspect AspectC {} //desplegado en el nivel 2
```

`up` y `down`

Como se explicó en el capítulo de semántica (3.3), los operadores `up` y `down` pueden ser aplicados a una expresión para modificar el nivel de ejecución en el que será ejecutada. En esta implementación, `up` y `down` reciben objetos, cuyo código es ejecutado en el nivel pedido.

Los objetos pasados como argumento a `up` y `down` deben implementar la interfaz `java.util.concurrent.Callable`. Esta interfaz define a `call`; el código en el cuerpo de este método es ejecutado en el nivel pedido.

El ejemplo en el listado siguiente ilustra la utilización de `up` y `down`. La clase anónima entre las lineas 6 y 10 implementa la interfaz `Callable` y es pasada como argumento a `up`. El método `call` simplemente imprime una linea de texto.

```
1  class Example{
2    static public void main(String[] args){
3
4      try {
5        Level.up(
6          new Callable<Object>() {
7            public Object call() throws Exception {
8              System.out.printf( "some text" );
9              return null;
10           }
11         })
12     } catch(Exception e) {}
13
14   }
15 }
```

`cflow` sensible al contexto

`l-cflow`, introducido en la subsección 3.4.2, puede ser utilizado de la misma forma que el pointcut `cflow()` existente en AspectJ: recibe como argumento a otro pointcut.

En el lenguaje, se ha reemplazado el pointcut `cflow()` por el pointcut `lcflow()`. Sin embargo, se puede acceder a la implementación original de `cflow()` escribiendo el pointcut `gcflow()`.

4.2. AspectBench Compiler

abc es un compilador alternativo para AspectJ. Es actualmente la implementación de AspectJ con fines investigativos de mayor avance.

abc tiene dos objetivos principales: estar diseñado con una arquitectura que facilite la extensión de la sintaxis y semántica del lenguaje y por otro lado, facilitar el desarrollo de optimizaciones en la generación bytecode que aumenten la eficiencia de los programas AspectJ compilados.

La versión actual de este compilador soporta las extensiones introducidas en la versión 1.5 de Java, como *generics* y *autoboxing*. Sin embargo, no soporta la sintaxis de AspectJ basada en anotaciones [13].

4.2.1. Arquitectura de abc

Frontend

Existen dos frontends del compilador abc. El primero está basado en Polyglot [6, 5] y es el utilizado por la versión inicial del compilador. Carece de soporte para las características nuevas en Java 5. El otro frontend, que es el utilizado en el trabajo, fue desarrollado posteriormente y está basado en Beaver [14] y JastaddJ [8].

Beaver es un generador de *parsers*. El parser de abc es generado con Beaver desde un conjunto de especificaciones. Las especificaciones incluidas pueden ser modificadas para incorporar extensiones al lenguaje. JastAddJ es un compilador desarrollado utilizando JastAdd [7,9]. JastAdd es un framework para la escritura de compiladores extensibles.

Backend

El backend de abc está implementado utilizando Soot [10,11]. Soot es un framework para la optimización de bytecode Java. Dispone de tres representaciones distintas del bytecode: Baf, Jimple y Grimp. Baf es una representación casi directa del bytecode y al igual que en él, las instrucciones trabajan sobre un stack. A diferencia del bytecode de Java, en el que existen instrucciones que no tienen tipos, todas las instrucciones de Baf están tipadas.

Las intrucciones de Jimple, a diferencia de como ocurre en Baf, no trabajan sobre un stack. Las instrucciones operan sobre variables, las cuales son obtenidas analizando el uso del stack. Adicionalmente, estas variables son declaradas con un tipo, por lo que las instrucciones que en Baf poseen versiones distintas por cada tipo de operando corresponden a una sola instrucción en Jimple. Las instrucciones en Jimple siempre poseen dos operandos y un resultado.

La representación Grimp del bytecode es muy similar a Jimple. La principal diferencia es que en Grimp se agregan las operaciones, es decir, es permitido tener expresiones que representan árboles de operaciones.

La estrategia estándar de soot es aplicar optimizaciones en cada una de estas distintas representaciones.

abc utiliza principalmente la representación Jimple del bytecode. El bytecode entregado por JastAddJ es manipulado luego de que ha sido convertido en Jimple. Las instrucciones correspondientes a la evaluación de pointcuts, invocación de advices y otras relacionadas con AspectJ son generadas en Jimple, las que luego son insertadas en el Jimple correspondiente al bytecode obtenido de JastAddJ.

Proceso de compilación en abc

En la figura 4.1 se puede apreciar un esquema del proceso de compilación de abc.

El código dentro de pointcuts `if()` y advices no puede ser transformado a Jimple directamente. Por ello, durante la conversión del código a Jimple se generan nuevos métodos. En los cuerpos de estos nuevos métodos se colocan el código de los pointcuts `if()` y de los advices. Luego se accede al Jimple de estos métodos nuevos.

La información sobre los aspectos extraidos del código es mantenida en objetos del paquete `abc.weaving.aspectinfo`, algunos de los cuales se muestran en la figura 4.2.

Estos objetos modelan las distintas construcciones que existen en AspectJ, como aspectos

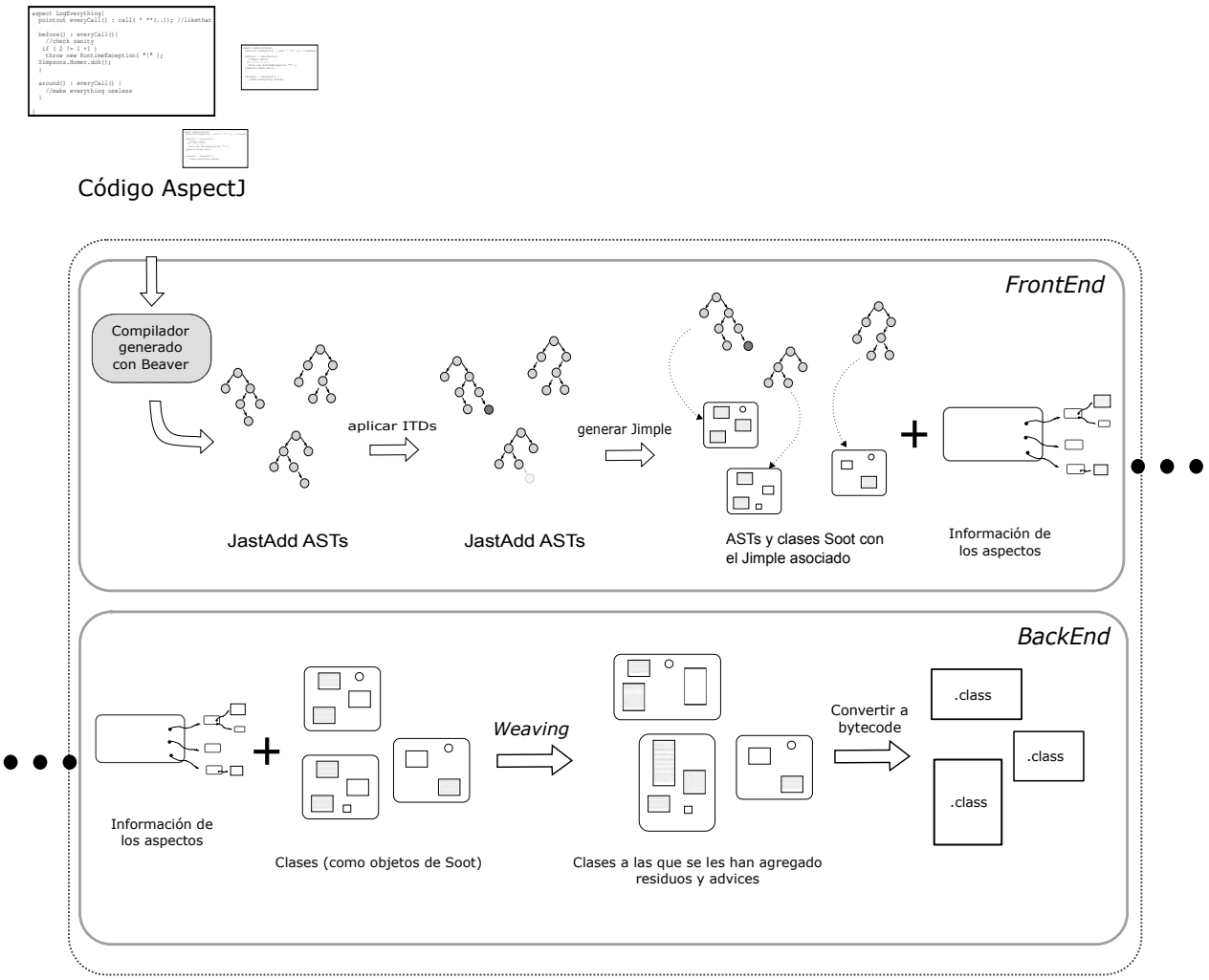

Figura 4.1: Proceso de compilación

y pointcuts. Los objetos son luego utilizadas para hacer el weaving de aspectos. Para ello, algunas clases poseen métodos que permiten acceder al bytecode (en forma de Jimple) correspondiente a las entidades que modelan. Como por ejemplo `AdviceDecl`, que modela la declaración de un advice dentro de un aspecto: esta contiene el método `makeAdviceExecutionStmts` cuya función es la de generar una cadena de sentencias que puede ser utilizada para invocar al advice que representa.

Residuos

En tiempo de compilación es posible evaluar por completo algunos pointcuts con la información disponible. Por ejemplo, se puede evaluar en tiempo de compilación el calce de un pointcut que seleccione la ejecución de un método. En cambio, si el calce de un pointcut depende del estado de los objetos del sistema, este no puede ser decidido durante la compi-

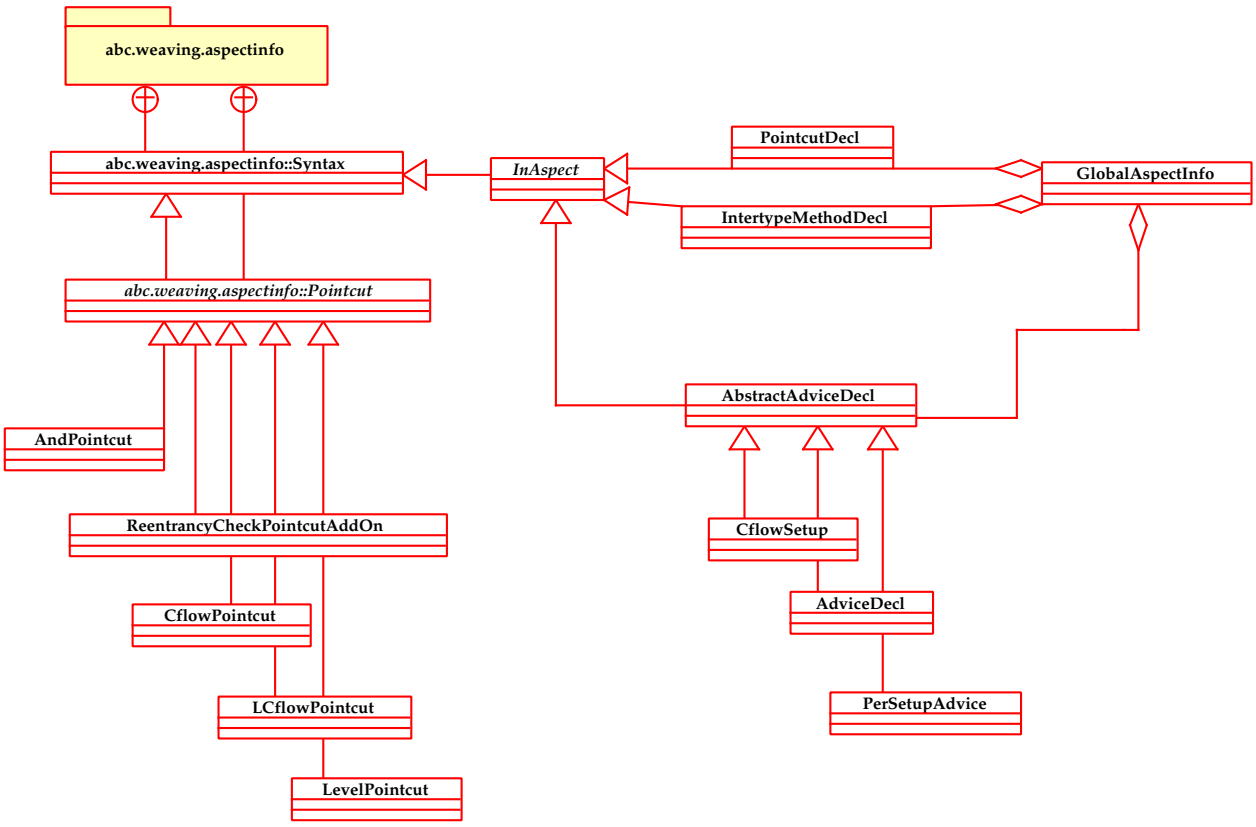

Figura 4.2: Algunas clases en el paquete AspectInfo

lación. Un ejemplo de esto es un pointcut `if` que pruebe el valor de un atributo `boolean` de un objeto.

Cuando no es posible evaluar un pointcut completamente en tiempo de compilación, parte de la evaluación se debe posponer hasta tiempo de ejecución. El código generado para completar esta evaluación es llamado *residuo*. Los residuos son insertados por el compilador en los lugares en que se identifica un potencial calce del pointcut.

En tiempo de ejecución el residuo evalúa el resto del pointcut. En caso de encontrarse un calce, el advice correspondiente es ejecutado en el momento apropiado: antes, después o alrededor del join point actual. Terminada la ejecución del advice (y dependiendo del advice mismo también del join point original), se continuará con la ejecución del programa. Por otro lado si el residuo determina que no existe un calce, no se ejecutará el advice asociado y el flujo de control normal del programa no se verá afectado.

En abc los residuos son modelados por las clases del paquete `abc.weaving.residues`.

El siguiente es un esquema de los residuos generados por abc, en pseudo-código:

```
1  ...
```

```
2  ;codigo correspondiente a un residuo
3  INSTRUCCION_1_RESIDUO
4  INSTRUCCION_2_RESIDUO
5  ...
6  ;si el residuo evalua como falso, no ejecutar advice.
7  IF condicion_no_calce GOTO :Fail
8  ;fin del residuo
9
10 ;codigo correspondiente a la invocacion del advice.
11 INVOCAR_ADVICE
12
13 :Fail
14 ; continua el programa base
15 ...
```

Cada objeto correspondiente a un residuo en abc tiene un método que devuelve el código correspondiente a él: `codeGen()`. Entre los parámetros que el método recibe se encuentran referencias a la sentencia a la que el residuo debe saltar en caso de que el pointcut no calce, y la sentencia a partir de la cual se debe insertar el código del residuo.

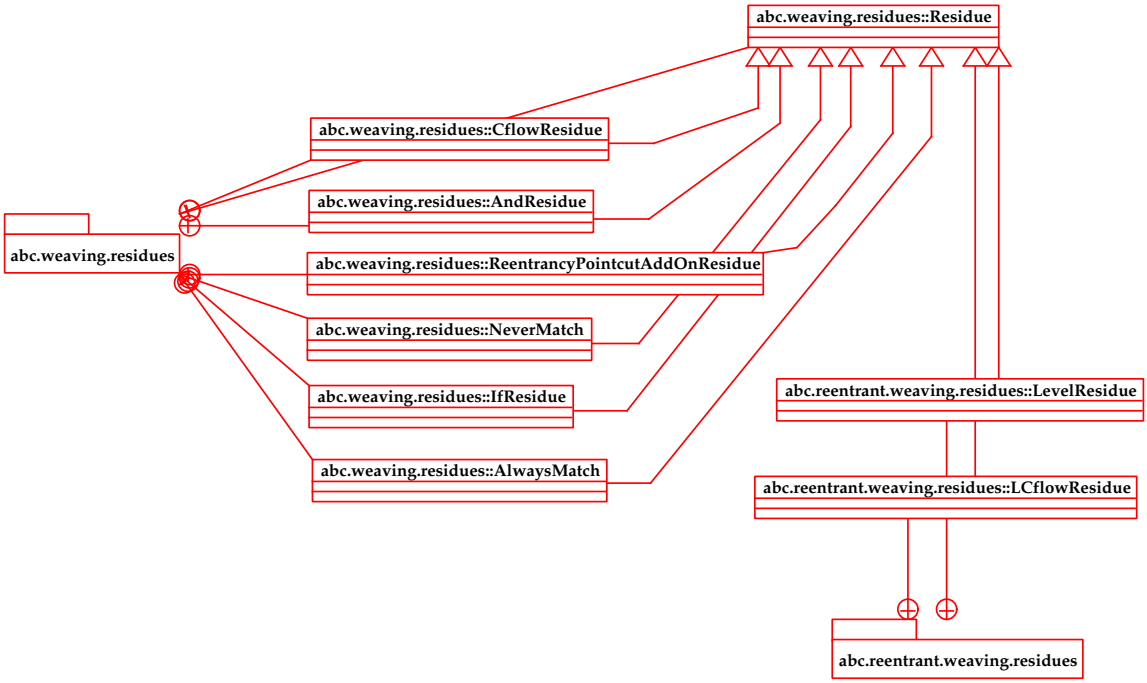

Figura 4.3: Algunas clases que modelan residuos relevantes

Para los casos en que abc es capaz de determinar estáticamente el valor de un pointcut en un cierto punto del programa base, se utilizan dos residuos especiales. `AlwaysMatch` significa que el pointcut calzará siempre, y `NeverMatch` que el pointcut nunca calzará. El residuo de `AlwaysMatch` es vacío, con lo que la estructura del código no se ve alterada por él al

incorporar los residuos al programa compilado. El residuo de `NeverMatch` consiste en un salto incondicional al lugar de la cadena de instrucciones que corresponde al programa original.

4.2.2. Extender abc

El principal mecanismo para modificar el comportamiento o aumentar las capacidades del compilador es una *extensión*. Estas extensiones pueden ser activadas luego al ejecutar el compilador, pasándole un argumento adicional.

En la siguiente figura se aprecian algunas clases que componen la extensión desarrollada. Las clases principales de esta extensión pertencen al paquete `abc.ja.reentrant`. Pueden apreciarse además otros paquetes correspondientes a otras extensiones: `abc.ja.eaj` corresponde a una versión extendida de AspectJ que incorpora funcionalidad adicional, y `abc.ja` que corresponde a la extensión que reimplementa el frontend utilizando JastAddJ, agregando soporte para Java 5.

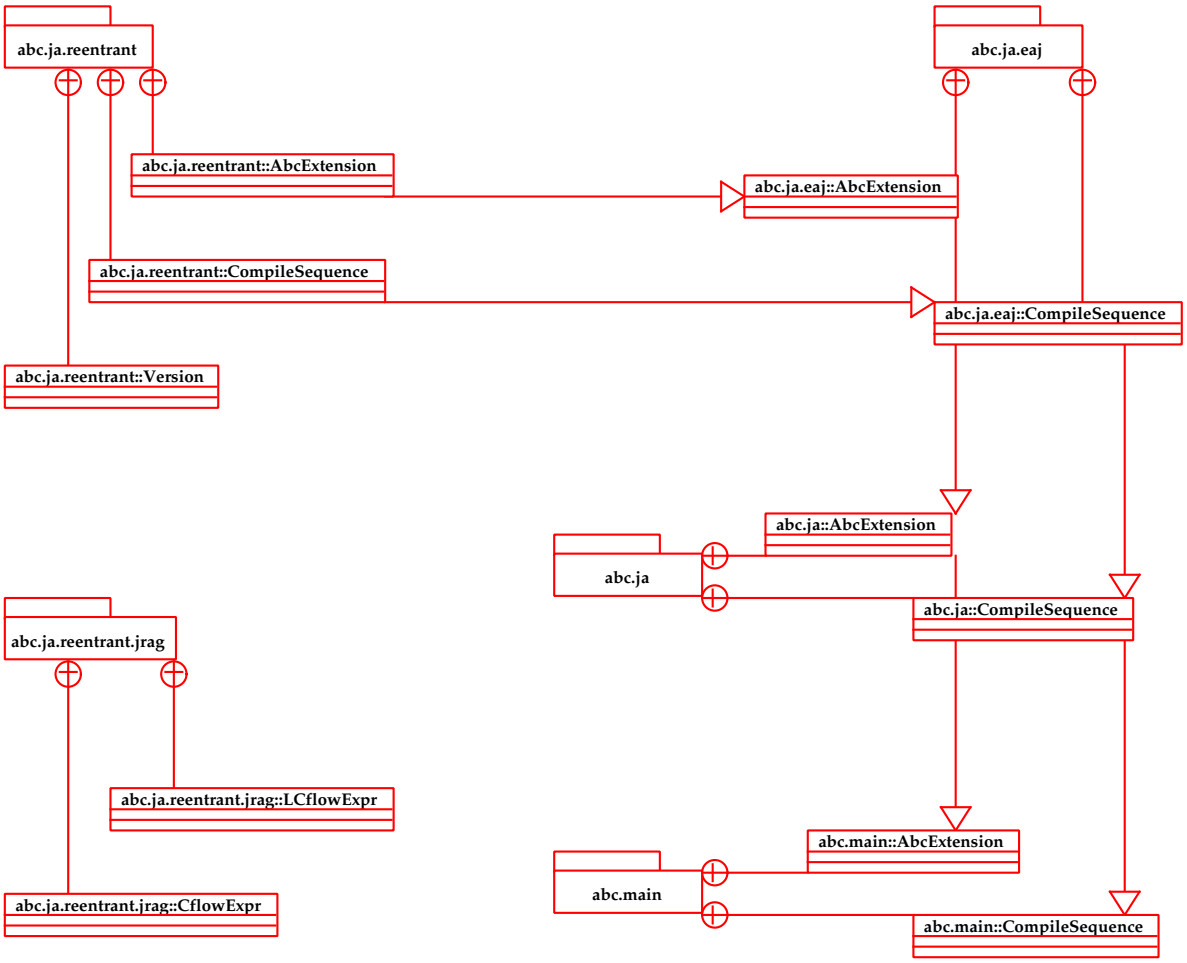

Figura 4.4: Paquetes y clases involucradas en la extensión

El paquete `abc.ja.reentrant.jrag` contiene las clases generadas por JastAdd. Estas modelan el AST utilizado por el frontend del compilador extendido.

Una extensión debe contener por lo menos una clase llamada `AbcExtension`. Esta clase debe heredar de `abc.main.AbcExtension`. En ella están definidos métodos que especifican el comportamiento del compilador, con el fin de que puedan ser redefinidos en una extensión.

4.2.3. Inserción de código

Como se mencionó anteriormente, abc trabaja sobre el código del programa transformado en Jimple. El código está disponible como una cadena de sentencias, en las cuales es posible insertar otras nuevas.

Para realizar inserciones de código dentro de cadenas generadas por abc, en algunos casos fue necesario modificar clases del compilador. En las siguientes páginas se mostrarán estos casos y las estrategias que se utilizaron.

4.3. Niveles

La implementación de niveles de ejecución es realizada por un conjunto de funcionalidades. Primero, la implementación de los **niveles de ejecución**. Para ello se tiene un objeto que modela el nivel de ejecución actual, el que tiene métodos apropiados para la manipulación de estos. Además, se modificó abc para que en los programas compilados por él ocurran los cambios de nivel dictados por la semántica (vista en el capítulo 3.3), de forma automática; y la modificación de los pointcuts de los programas compilados, para que los aspectos sólo se ejecuten cuando corresponda según el nivel en que son desplegados. Con esto, la ejecución de los programas compilados con esta extensión está organizada en niveles de ejecución.

Segundo, la implementación de un **control de reentrancia**; con lo que se elimina toda posibilidad de reentrancia de aspectos.

Y la implementación del **cflow sensible al contexto** (`l-cflow`) a partir del cflow tradicional de AspectJ.

En las subsecciones siguientes se detalla la implementación de cada una de estas funcionalidades.

4.3.1. Runtime para mantener el nivel

La clase `Level` posee un mecanismo para mantener el valor del nivel de ejecución y métodos para manipularlo. Esta se encuentra en el paquete `abc.reentrant.runtime`, como se puede ver en la figura 4.5.

`Level` implementa el patrón singleton, de forma que las peticiones relacionadas con el nivel de ejecución son servidas durante la ejecución por un único objeto. El valor es guardado dentro de este objeto único como un valor `ThreadLocal`, para evitar interferencias entre los valores de los niveles de ejecución de distintos threads.

`Level` implementa los métodos `up` y `down` mencionados en la sección 4.1. El usarlos garantiza que antes, durante y después de ser ejecutado el código especificado, los niveles de ejecución en los que corre el programa serán los apropiados.

Además, `Level` posee los métodos `forceUnmanagedUp` y `forceUnmanagedDown`, que cambian directamente el valor del nivel de ejecución. Estos son utilizados para poder implementar los cambios automáticos de nivel introducidos en los programas compilados, y para implementar `up` y `down`.

4.3.2. Inserción de código para el cambio de nivel en abc

Para lograr que el nivel de ejecución cambie sin intervención del programador, el compilador debe insertar instrucciones que realizen estos cambios en lugares especiales dentro del código del programa. El código adicional que se debe incluir en los programas compilados debe realizar un par de invocaciones estáticas a los métodos que cambian el nivel de ejecución. El código que genera el Jimple correspondiente a las invocaciones es el siguiente:

```
1  //subir nivel:
2  Stmt levelUpStmt = Jimple.v().newInvokeStmt(
3   Jimple.v().newStaticInvokeExpr(
4    Scene.v().getMethod(
5     "<abc.reentrant.runtime.Level: void forceUnmanagedUp()>").makeRef()
6   ));
7
8  //bajar nivel
9  Stmt levelDownStmt = Jimple.v().newInvokeStmt(
10  Jimple.v().newStaticInvokeExpr(
11   Scene.v().getMethod(
12    "<abc.reentrant.runtime.Level: void forceUnmanagedDown()>").makeRef()
13  ));
```

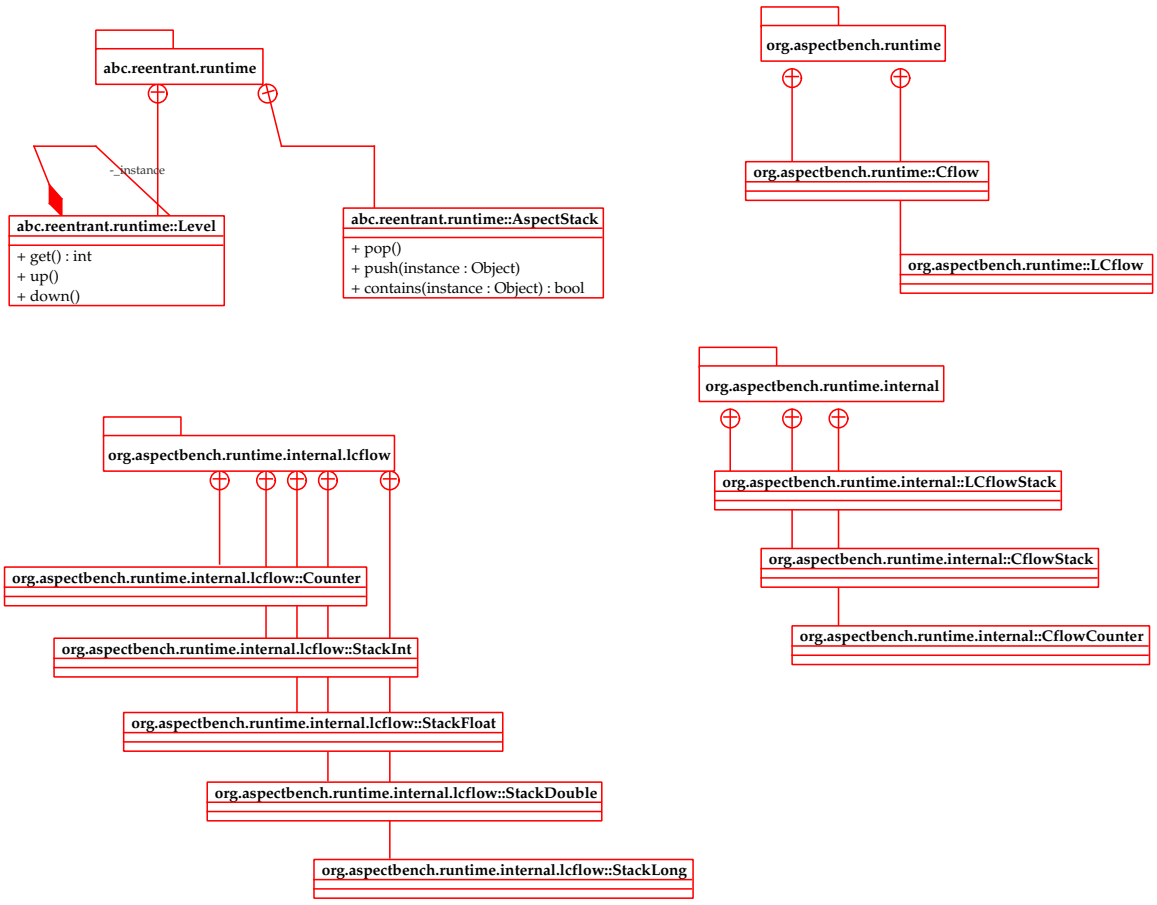

Figura 4.5: Algunas de las clases pertenecientes a las librerias de runtime de la extensión

Estas sentencias son luego insertadas en las cadenas de código Jimple que corresponden a los lugares del programa en los cuales es necesario provocar los cambios de nivel.

Como se vió en el capítulo de semántica (3.3), la ejecución del código de un aspecto debe ocurrir en un nivel superior. En los siguientes parrafos se detallan los cambios de nivel que ocurren en distintos momentos de la ejecución de los aspectos.

Código dentro de un pointcut

Se necesita que todo el código ejecutado en un pointcut esté en un nivel superior al del join point con el que se evalúa. Es decir, cambiar al nivel superior antes de ejecutar el código, y una vez finalizada la ejecución del pointcut cambiar al nivel inferior. En el caso de AspectJ, el único pointcut capaz de ejecutar código es el pointcut `if()`. Por ello, los cambios de nivel deben ocurrir inmediatamente antes y después de la ejecución del código contenido en el pointcut `if()`.

Para implementar estos cambios de nivel, se modificó la clase que genera el residuo co-

rrespondiente a este pointcut: `abc.weaving.residues.IfResidue`. El método encargado de la generación de código para el residuo fue modificado para cambiar la cadena de sentencias que produce.

Como se mencionó en el apartado que describe la arquitectura de abc, el código dentro de cada pointcut `if()` se coloca en el cuerpo de un método introducido por el compilador. El residuo contiene una invocación a este método, pues requiere ejecutar este código para evaluar el pointcut. En la cadena de sentencias de este residuo, inmediatamente antes y después de esta invocación se agregaron nuevas sentencias que realizan los cambios de nivel. Estas sentencias son invocaciones a los métodos `Level.forceUnmanagedUp` y `Level.forceUnmanagedDown`.

Ejecución del cuerpo de un advice

En el caso de los advice, la modificación se realizó en la clase `AdviceDecl` (perteneciente al paquete `abc.weaving.aspectInfo`), dentro del método `makeAdviceExecutionStmts`. Como las sentencias de la cadena generada por este método son ejecutadas cada vez que se debe ejecutar el advice que interesa, es un lugar adecuado para introducir los cambios de nivel. Las sentencias que realizan la invocación estática de los métodos se insertaron al principio y al final de la cadena que genera este método.

Invocación de proceed() dentro de un advice around

Para el caso de la invocación de `proceed()` en un advice around se modificó la clase `abc.weaving.weaver.around.ProceedInvocation`. Los objetos de esta clase representan la invocación de un proceed dentro de un around advice, y contienen la lógica para la generación del código que continuará con la ejecución del join point sobre el cual aplique el advice.

El método `generateProceed()` genera la cadena de sentencias necesarias para realizar un proceed. Esta cadena fue modificada para contener dos sentencias adicionales de invocaciones, rodeando al código responsable de que continúe la ejecución del join point original. La primera invocación provoca que la ejecución baje de nivel, y la segunda que suba, con lo que vuelve a quedar en la situación original.

4.3.3. Nivel observado por un aspecto

La implementación busca agregar a cada pointcut la condición de que para calzar, el nivel de ejecución actual debe ser el especificado en el aspecto que lo contiene. El AST es

modificado de modo de combinar, con una conjunción lógica (&&), cada uno de los pointcuts que aparece en la declaración de los advices con otro adicional: `Level()`. Este nuevo pointcut recibe de argumento un número que será comparado, en tiempo de ejecución, con el número del nivel de ejecución actual y calzará si son iguales.

En el siguiente listado de código se puede ver un aspecto que se desplegará en el nivel 2.

```
1 onLevel 2 aspect ExampleAspect{
2   pointcut foo() : call(* *.bar(..));
3
4   before() : foo(){ ... }
5 }
```

Luego de ser reescrito, el AST resultante es tal, que puede ser generado a partr del código a continuación.

```
1 aspect ExampleAspect{
2   pointcut foo() : call(* *.bar(..));
3
4   before() : Level(2) && ( foo() ) { ... }
5 }
```

Las modificaciones en el AST se realizan aprovechando las facilidades para reescribirlo que entrega JastAdd. Los cambios necesarios están escritos como reglas de transformación, cuyo orden y momento de aplicación son solucionados por JastAdd.

La regla para restringir los pointcuts al nivel en el que el aspecto contenedor ha sido desplegado puede ser vista en el fragmento de código siguiente.

```
1  rewrite AdviceDecl {
2    when ( !this.alreadyRewritten )
3    to AdviceDecl {
4
5     AspectDecl aspDecl = (AspectDecl)this.getParent().getParent();
6
7     int intentedLevel = asptDecl.level;
8
9     PointcutExpr oldPcExpr = this.getPointcutExpr();
10
11    LevelPointcutExpr levelPcExpr = new LevelPointcutExpr();
12
13    levelPcExpr.setlevel(intentedLevel);
14
15    AndPointcutExpr newPcExpr = new AndPointcutExpr(
16                                levelPcExpr, oldPcExpr);
17
18    this.setPointcutExpr(newPcExpr);
19
20    this.alreadyRewritten = true;
21
22    return this;
```

```
23
24      }
25  }
```

El código contenido en la regla obtiene el número del nivel al que el pointcut se restringirá (lineas 5 a 7). Luego se genera un nuevo nodo correspondiente a un pointcut `&&` y un nuevo nodo correspondiente al pointcut `Level` que se quiere introducir (lineas 11 a 16). Al ser creado el nodo correspondiente a `&&` se le asignan como hijos el pointcut `Level` y el pointcut original. La parte esencial de la regla es el cambio del pointcut original por el nuevo pointcut `&&`, en la linea 18.

En la segunda linea del código se realiza un chequeo al atributo `alreadyRewritten`. Esto es para asegurar que la reescritura se realiza sólo una vez.

Es importante comentar que debido a la regla de reescritura anterior, todos los pointcuts del programa compilado tendrán residuo pues, por lo menos, tendrán como residuo el correspondiente al pointcut `Level`. Una forma de minimizar los lugares en que se inserta este residuo es parte del trabajo futuro (sección 6.2).

4.4. Control de reentrancia

Como se mencionó al final de la subsección 3.5.2, para implementar el control de reentrancia se requiere poder determinar en cada momento si es que se está dentro del flujo de control de un aspecto dado. Antes de ejecutar el código penteneciente a cualquier pointcut de un cierto aspecto, se chequea que no se esté ya en el flujo de control del mismo. Para ello se necesita una estructura similar a un stack en el que se van apilando identificadores de los aspectos: al entrar en el flujo de control se apila el identificador del aspecto correspondiente, y al salir se desapila. Esta puede ser luego examinada para determinar si es que se está en el flujo de control: si es que se está en el flujo de control de un aspecto en particular, su identificador debe encontrarse en la estructura.

El stack de aspectos requerido debe ser actualizado en tiempo de ejecución a medida que los aspectos se ejecutan. Para ello fue necesario agregar en los programas compilados código adicional que se encargara de actualizarlo. Este fue insertado en los mismos puntos en que fue necesario colocar el código para la implementación de los niveles.

El stack de aspectos contiene, más precisamente, referencias a instancias de aspectos.

Esto porque en AspectJ los aspectos son similares a los objetos, contienen comportamiento y estado, lo que hace necesario el poder distinguir entre las (posibles) instancias distintas de ellos.

La estructura de datos es implementada por la clase `AspectStack`. Soporta las operaciones normales de un stack pero adicionalmente puede ser consultado de forma eficiente por si contiene una cierta instancia de un aspecto. Dado su uso frecuente, esta operación de consulta es implementada utilizando una *tabla de hash* auxiliar. En la implementación, esta tabla de hash es actualizada simultáneamente junto al stack de aspectos[1].

Para implementar el control de reentrancia se introdujo un nuevo pointcut, de nombre `ReentrancyCheckPointcutAddOn`. Este envuelve al pointcut anónimo presente en cada una de las declaraciones de advice del programa, de forma que antes de que se ejecute este pointcut, se comprueba la condición de que este no es reentrante. El residuo del nuevo pointcut primero chequea la condición de reentrancia y falla inmediatamente si no se cumple, con lo que la parte del residuo correspondiente al pointcut que envuelve no es ejecutado. En caso de no ser reentrante, se ejecuta el residuo del pointcut envuelto.

El pointcut es introducido con una regla de reescritura análoga a la mostrada en la subsección 4.3.3.

Para evaluar la condición, se comprueba el stack de aspectos: si la instancia del aspecto actual ya se encuentra en el stack, el pointcut no calza. La consulta al stack se realiza invocando al método `contains()` con la referencia a la instancia del aspecto.

En la subsección 4.3.3 se explica que todos los pointcuts tienen por lo menos un residuo, correspondiente a `Level`. Con la adición de `ReentrancyCheckPointcutAddOn`, todos los residuos de los pointcuts además poseen el código correpondiente al control de reentrancia. Insertar el residuo sólamente en los lugares en que sea necesario es parte de los planes para trabajo futuro, en la sección 6.2.

4.5. Cflow sensible al nivel de ejecución

El *runtime* del cflow en abc tiene dos implementaciones. Una que utiliza stacks y otra que utiliza contadores. La implementación preferida por el compilador es la de contadores, pero en

[1]Basta con una tabla de hash gracias a que el mismo control de reentrancia garantiza que nunca habrán dos o más instancias idénticas dentro del stack de aspectos

Clases	Descripción
`LCflow`, `LCflowBelow`	Clases AspectInfo correspondientes al `lcflow()`
`LCflowResidue`	Clase correspondiente al residuo de este pointcut
`LCflowSetup`, `GlobalLCflowSetupFactory`	Relacionadas con el advice artificial que actualiza el estado necesario para cada `lcflow()`
`LCflowCodeGenUtils`	Contiene la lógica de generación de código necesario para consultar y actualizar el estado de un `lcflow()`

Cuadro 4.1: Clases introducidas en el compilador para implementar `l-cflow`

caso de que sea necesario mantener bindings del cflow se hace necesario utilizar el stack. Esto ocurre cuando el pointcut al interior del `cflow()` expone información del contexto utilizando pointcuts como `this()` o `target()`. Para este trabajo, el nuevo runtime sólo contiene una implementación con stacks, pues la eficiencia no era una preocupación importante y con ello basta para todos los casos de uso.

En cuanto a la implementación del `l-cflow`, correspondiente al pointcut `lcflow()` de AspectJ, esta posee la misma estructura de clases que existe para el cflow normal. Se extendió cada una de las clases de los originales, agregando la funcionalidad nueva en estas nuevas clases. Fue necesario cambiar la visibilidad (a *public* en la mayor parte de los casos) de varios miembros de las clases originales e incluso de las clases mismas. Los nombres de las clases pueden ser vistos en la tabla 4.1.

La implementación del runtime para el `lcflow()` requiere una operación adicional, con respecto a la implementación del cflow normal en abc. Esta permite preguntar por el estado del cflow para un nivel de ejecución en particular. Cada celda del stack se anota con el nivel en el que se estaba en el momento en que se agregó ésta celda. Entonces, la operación realiza un recorrido por las celdas del stack ignorando toda celda que no corresponda al nivel por el cual se está preguntando.

4.5.1. Reemplazo del cflow normal

Finalmente, el nuevo cflow sensible al nivel de ejecución debe reemplazar al cflow normal. Esto es, que al utilizar el pointcut `cflow()` dentro de un aspecto se obtenga la semántica de un `l-cflow`. El cflow normal, estará disponible a través un pointcut llamado `gcflow` (*global cflow*). El cflow sensible al contexto podra ser utilizado escribiendo `lcflow()`.

El cambio interno de pointcut `cflow()` a `lcflow()` se logró utilizando una regla de

reescritura de JastAdd. La regla se puede ver más abajo.

```
1  public boolean CflowPointcutExpr.declaredAsCflow = true;
2
3  rewrite CflowPointcutExpr {
4      when ( this.declaredAsCflow )
5      to LCflowPointcutExpr {
6          return new LCflowPointcutExpr( this.getPointcut() );
7      }
8  }
```

La regla simplemente crea un nuevo nodo del tipo que representa a un `lcflow()` utilizando el contenido del pointcut original. Existe una regla análoga para el caso de los pointcuts `cflowbelow()` y `lcflowbelow()`.

En el caso de `gcflow()` se requirieron dos cambios. El primero es agregar un token nuevo para `gcflow()` (y `gcflowbelow()`) de modo que el parser lo reconozca. El segundo es asociar estos tokens a un nodo del AST: `gcflow()` (`gcflowbelow()`) se asocia a un nodo de tipo `cflow()` (`cflowbelow()`) del AST. El código para ello está a continuación.

```
1  PointcutExpr basic_pointcut_expr =
2
3    PC_GCFLOW LPAREN pointcut_expr.internal_pc RPAREN
4      {:
5          ParserTrace.parserTrace("GCFLOW pointcut");
6          CflowPointcutExpr pc =
7                          new CflowPointcutExpr(internal_pc);
8          pc.declaredAsCflow = false;
9          return pc;
10     :}
11
12   | PC_GCFLOWBELOW LPAREN pointcut_expr.internal_pc RPAREN
13     {:
14         ParserTrace.parserTrace("GCFLOWBELOW pointcut");
15         CflowBelowPointcutExpr pc =
16                         new CflowBelowPointcutExpr(internal_pc);
17         pc.declaredAsCflow = false;
18         return pc;
19     :}
20   ;
```

Este código es parte de la definición del parser para la extensión desarrollada. El estilo del código es similar a la declaración de una gramática. `PC_GCFLOW`, `PC_GCFLOWBELOW`, `LPAREN` y `RPAREN` son símbolos terminales y corresponden a los tokens `cflow`, `cflowbelow`, `(` y `)` respectivamente. Mientras que `basic_pointcut_expr` y `pointcut_expr` son símbolos no-terminales. Entre `{:` y `:}` se encuentra código Java que debe ser ejecutado en cada caso. La expresión `pointcut_expr.internal_pc` asigna el nombre de variable `internal_pc` a el objeto que fue encontrado por el parser en esa posición, con el cual puede ser manipulado

dentro del código.

Cada vez que el parser encuentra una expresión de `gcflow()` bien formada ejecuta el código incrustado, lo que crea un nodo del AST correspondiente a un `cflow()`: `CflowBelowPointcutExpr`. Notar que se establece el valor de `declaredAsCflow` como `false`, de modo que el nodo creado no sea afectado después por la primera regla de reescritura mostrada en esta sección. Para el caso de `gcflowbelow()` se procede de la misma forma, pero con los tipos de nodos apropiados.

Capítulo 5

Validación

Se prepararon pruebas de dos tipos. Las primeras muestran la correctitud de la implementación. El otro grupo corresponde a benchmarks sobre los que se realizó un ligero análisis de eficiencia.

5.1. Correctitud

5.1.1. Tests

En paralelo con el desarrollo de la implementación se confeccionó un grupo de pruebas. Las pruebas están agrupadas de acuerdo a la funcionalidad a la que están dirigidas. El código correspondiente a estas pruebas está incluido en los anexos.

Runtime para niveles de ejecución (`BasicLevel`)

Estas pruebas están dirigidas a comprobar que los métodos para el cambio de nivel (`Level.up()` y`Level.down()`) funcionen correctamente.

Cambios de niveles automáticos (`BasicLevel`)

Para comprobar que el nivel se eleve, sin intervención del programador, cada vez que se ejecute código de un aspecto. Además de bajar correctamente cuando un `proceed()` sea invocado dentro de un advice around.

Deploy (`BasicLevelDeploy`)

Aseguran que los aspectos son aplicados acorde con el nivel en que estos hayan sido desplegados.

Actualización automática del stack de aspectos (`AspectStackUpdating`)

Comprueban que el stack de aspectos utilizado por el control de reentrancia siempre esté actualizado correctamente. Debe contener en el tope una referencia al aspecto dentro del cual se está llevando a cabo la ejecución, de haber alguno.

Control de reentrancia (`BasicReentrancyControl`)

Para asegurar que el control de reentrancia funciona al utilizar el operador `down`. En el contexto de un aspecto se generan join points que debieran ser calzados por uno de sus pointcuts, y se comprueba que estos no calzan.

`cflow` sensible al contexto (`BasicLCflow`)

Esta prueba cuida que `lcflow()` implemente correctamente el chequeo que comprueba la relación entre los niveles del join point actual y el(los) joinpoint(s) definido(s) por el pointcut interno del `lcfow()`. Se comprueba que `lcflow()` calza cuando el nivel es el mismo y que no calza si es que son distintos.

5.1.2. Aplicaciones

AJHotDraw

AJHotDraw [16] es un *refactor* del framework gráfico JHotDraw [17]. JHotDraw fue desarrollado como un ejercicio de diseño, y utiliza fuertemente patrones de diseño de programación orientada a objetos.

AJHotDraw viene acompañado de programas de ejemplo que aprovechan el framework. Uno de ellos es un programa de dibujo llamado *javadraw*, capaz de abrir, grabar y editar imágenes.

AJHotDraw está escrito en 20339 lineas de código[1], de las cuales, 1191 corresponden a código escrito en AspectJ. El programa de dibujo de ejemplo está escrito en 647 lineas de código Java.

abc con la extensión para control de reentrancia activada compila correctamente el framework y la aplicación de dibujo, sin que haya sido necesario modificar el código en ninguna forma relevante[2]. El programa de dibujo funciona correctamente al ejecutarlo.

[1] Sólo el framework, sin contar comentarios ni lineas en blanco. El conteo se hizo con el programa cloc [15]

[2] Se debió modificar el *build file* de *ant* utilizado para compilar y un aspecto(un pointcut no válido que no

	referencia	extensión	%
bean	0,31	6,65	2112,2 %
bean_gregor	0,23	6,79	2952,2 %
figure	1,62	28,28	1743,4 %
hello	0,05	0,05	100 %
nullcheck-sim	0,81	25,1	3098,7 %
nullcheck-sim_after	0,75	3,26	433,2 %
nullcheck-sim_notwithin	0,79	24,9	3151,9 %
nullcheck-sim_orig	1,4	55,3	3950 %
quicksort_gregor	2,91	20,41	702,2 %
quicksort_oege	3,2	22,73	710,3 %

Cuadro 5.1: Tiempos de ejecución, en segundos, para cada benchmark.

5.2. Performance

A pesar de que no es parte de los objetivos de este trabajo, se realizaron pequeñas pruebas de rendimiento para dimensionar la magnitud del sobrecosto que poseen los programas al ser compilados con la extensión desarrollada. Esto es de interés debido a que *todos* los pointcuts de los programas compilados con esta versión extendida del compilador poseen residuos.

En [18], se encuentra disponible un grupo de benchmarks implementados en AspectJ.

Cada benchmark fue ejecutado 3 veces y se calculó el promedio del tiempo que demoró su ejecución. Se compararon los tiempos de los mismos al ser compilados con una versión sin modificar de abc[3] y luego compilados con la extensión activada. No se pudo utilizar todas las pruebas incluidas en el benchark debido a problemas de compatibilidad de abc; adicionalmente, se omitieron los benchmarks cuyo código fuente no contiene aspectos. Los resultados están en el cuadro 5.1.

`quicksort_gregor` y `quicksort_oege` consisten en aplicar quicksort a un conjunto de datos de tamaño considerable mientras un aspecto cuenta el número de intercambios y particiones realizados. Cada vez que se va a realizar un intercambio o una nueva partición, se realizan los nuevos chequeos de reentrancia y nivel, junto con las actualizaciones a las estructuras de datos que estos chequeos consultan. La inclusion de estos chequeos aumenta el tiempo que toma el test aproximadamente unas 7 veces.

`figure` muestra un gran cambio en el tiempo de ejecución, unas 18 veces. La idea del

se usaba tuvo que ser removido).

[3]Específicamente, versión 1.3 con frontend JastaddJ y con soporte para `eaj`, que es exáctamente la versión en que se basa la extensión.

benchmark es muy similar al ejemplo de puntos utilizado en capítulos anteriores. Se tienen puntos y rectas en el espacio que puede ser movidos y un aspecto que, cada vez que alguno de ellos se mueve, aumenta un contador. El benchmark consiste en realizar gran cantidad de veces un par de operaciones de desplazamiento sobre un punto y una linea.

La lógica de este benchmark es bastante simple: el movimiento de los objetos es simplemente modificar las coordenadas de los mismos, y el aspecto lo único que hace es aumentar el contador; en contraste, los residuos de chequeo de reentrancia y nivel realizan operaciones más complicadas: el primero un chequeo a una tabla que contiene aspectos de referencias ,y el segundo obtiene el valor del nivel y luego efectúa una comparación con él. Además, de las respectivas rutinas de actualizacion de nivel y del stack de aspectos al ejecutar el código del advice. La simpleza del código base y el advice al compararlo con el código que es agregado por la extensión explica este aumento en el tiempo de ejecución.

`bean` y `bean_gregor` es similar a `figure`. Hay un objeto que modela puntos, el cual es aumentado con nuevas capacidades mediante un aspecto. La prueba consiste en realizar varias operaciones sobre un punto. En este caso el deterioro de performance es más notorio; esto probablemente pueda ser explicado observando las características del advice que introduce el aspecto de este ejemplo. A diferencia de como ocurre en `figure`, el advice contiene un `proceed()`, lo que requiere la inserción de código adicional para mantener las estructuras utilizadas en los chequeos.

Los benchmarks cuyo nombre comienzan con `nullcheck-sim` contienen un aspecto que posee un pointcut que calza todas las llamadas a métodos que retornen un objeto. El advice asociado a este pointcut examina el valor de retorno para detectar cuando sea igual a `null`. `nullcheck-sim`, `nullcheck-sim_notwithin` y `nullcheck-sim_orig` utilizan un advice de tipo around para esto, y muestran el mayor sobrecosto. En cambio, `nullcheck-sim_after` utiliza un advice de tipo (after returning), por lo que no tiene una llamada a `proceed()`; y esto a su vez significa que existe menos código adicional para actualizar el estado del stack de aspectos y del nivel de ejecución.

Gran parte del sobrecosto podría ser eliminado en estos ejemplos. Cada uno de ellos ejecuta adecuadamente en ausencia de niveles de ejecución y el mecanismo de control de reentrancia asociado. Un análisis de código automatizado podría detectar la falta de necesidad de incluir la funcionalidad asociada a niveles de ejecución, e idealmente omitiría la introduc-

ción de gran parte, sino de todo, el código relacionado. La investigación y desarrollo de un mecanismo que permita este tipo de optimizaciones es parte del trabajo futuro (sección 6.2).

Capítulo 6

Conclusiones y perspectivas

6.1. Conclusiones

Se ha conseguido una implementación funcional de los conceptos introducidos en [1]. Esta implementación, una extensión para el compilador abc, introduce en AspectJ niveles de ejecución junto con el control de reentrancia(para el caso explicado en la sección 3.5.2).

El uso del compilador desarrollado elimina la posibilidad de que se produzca reentrancia en aspectos, liberando al programador de la responsabilidad de evitar su aparición; aunque a un costo que debe ser considerado, pues aumenta a medida que mayor es el alcance de los pointcuts de la aplicación.

El compilador extendido mantiene un alto nivel de compatibilidad, pues es capaz de compilar proyectos de tamaño mediano. Y gracias a la semántica definida para las nuevas características incorporadas al lenguaje, no es necesario realizar modificaciones al código fuente. El control de reentrancia puede ser aprovechado por proyectos en desarrollo simplemente cambiando al compilador presentado.

6.2. Trabajo futuro

Con respecto a la implementación, las clases para tiempo de ejecución podrían ser optimizadas. El stack de aspectos podría ser reimplementado con multiples stacks, uno para cada nivel de ejecución; esto permitiría responder a las consultas con un algoritmo que toma tiempo $O(1)$.

También podría realizarse un estudio del desempeño de la extensión. No se sabe cómo las distintas partes de la implementación contribuyen al sobrecosto mostrado en las pruebas de rendimiento en la sección 5.2. Los resultados ayudarían a priorizar las secciones que necesiten

ser optimizadas; las herramientas desarrolladas durante el estudio servirían además para medir los avances logrados por optimizaciones futuras.

El compilador extendido durante este trabajo, inserta los residuos y pointcuts que forman la implementación de niveles de ejecución en todos los lugares en que podrían ser utilizados. Queda pendiente la investigación sobre los posibles análisis que se puedan realizar sobre el código, de forma de encontrar oportunidades en las que estos chequeos puedan ser omitidos.

Una vía posible de optimización para el chequeo de reentrancia podría resultar de contrastar los pointcuts de un aspecto con los join points que pueden ser producidos por él. Dado un aspecto, el conjunto de los posibles lugares en los que un pointcut puede producir un calce puede ser separado en dos subconjuntos: algunos de estos lugares pueden ser accedidos desde el flujo de control de este aspecto, mientras que el resto no. En los lugares correspondientes al segundo subconjunto nunca se producirá un calce que origine reentrancia. Los residuos correspondientes al chequeo de control de reentrancia pueden ser removidos en estos lugares.

Otra posibilidad es desarrollar un análisis estático del código que determine los distintos niveles de ejecución en los que es posible que cada lugar del código se ejecute. Esta información puede ser luego contrastada con los lugares en los que el compilador ha detectado posibles calces de un pointcut y el nivel en que el aspecto que lo contiene debe ser desplegado; la inserción de residuos puede omitirse completamente en cada lugar en el que no se encuentre coincidencia entre los niveles.

Sería interesante realizar un estudio empírico sobre la utilización de chequeos adhoc para evitar la reentrancia. Determinar la frecuencia con la que aparece reentrancia de aspectos, y la dificultad de la detección de problemas de reentrancia y de la implementación de una solución. El estudios sobre la frecuencia podría enfocarse en proyectos existentes. Experimentos con programadores de distintos niveles de destreza en AspectJ podrían dar luces sobre los otros puntos sugeridos.

Bibliografía

[1] Éric Tanter. Execution Levels for Aspect-Oriented Programming. In Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010), Rennes and Saint Malo, France, Mar. 2010. ACM Press. To Appear.

[2] Éric Tanter. Controlling Aspect Reentrancy. In Journal of Universal Computer Science, 14(21):3498–3516, 2008.

[3] Dutchyn, C., Tucker, D. B., and Krishnamurthi, S. 2006. Semantics and scoping of aspects in higher-order languages. Sci. Comput. Program. 63, 3 (Dec. 2006), pp. 207-239.

[4] E Bodden, F Forster, F Steimann "Avoiding infinite recursion with stratified aspects". In GI-Edition Lecture Notes in Informatics "NODe 2006 GSEM 2006", Robert Hirschfeld, Andreas Polze and Ryszard Kowalczyk (Editors), pp. 49-64.

[5] Avgustinov, P., et al. abc: An extensible AspectJ compiler. Proceedings of the 4th international conference on Aspect-oriented software development (2005), pp. 87-98.

[6] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In Compiler Construction, 12th International Conference, CC 2003, volume 2622 of LNCS, pages 138–152. Springer, 2003.

[7] Hedin, G. and Magnusson, E. 2003. JastAdd: an aspect-oriented compiler construction system. Sci. Comput. Program. 47, 1 (Apr. 2003), pp. 37-58.

[8] Ekman, T. and Hedin, G. 2007. The jastadd extensible java compiler. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (Montreal, Quebec, Canada, October 21 - 25, 2007). OOPSLA '07. ACM, New York, NY, 1-18.

[9] Jastadd and JastaddJ website: http://www.jastadd.org/ .

[10] Vallie-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. 1999. Soot - a Java bytecode optimization framework. In Proceedings of the 1999 Conference of the Centre For Advanced Studies on Collaborative Research (Mississauga, Ontario, Canada, November 08 - 11, 1999). S. A. MacKay and J. H. Johnson, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 13.

[11] Soot website: http://www.sable.mcgill.ca/soot/.

[12] AspectJ website http://www.eclipse.org/aspectj/.

[13] AspectWerkz website: http://aspectwerkz.codehaus.org/ .

[14] Beaver website: http://beaver.sourceforge.net/ .

[15] Counting lines of code Website: http://cloc.sourceforge.net/.

[16] Un refactor orientado a aspectos de JHotDraw. AJHotDraw website: http://ostatic.com/ajhotdraw.

[17] Framework Java para interfaces gráficas para el usuario para gráficos técnicos y estructurados. JHotDraw website: http://www.jhotdraw.org/.

[18] Benchmarks en el sitio web de abc: http://abc.comlab.ox.ac.uk/benchmarks.

Apéndices

A . Código de los tests

BasicLevel

```
1  package abc.reentrant.test.tests;
2
3  import abc.reentrant.runtime.Level;
4  import junit.framework.TestCase;
5
6  /**
7   *
8   * @author carlos
9   */
10 public class BasicLevel extends TestCase {
11
12     public void test_baseLevelAdvicing() {
13         /*Here we'll be testing that the level gets automatically changed*/
14         System.out.println(
15             "\n-Testing execution Levels: correct execution level updating");
16         BasicLevel.assertLevelValue( 0 , "------before all advices------");
17         doSomething();
18         BasicLevel.assertLevelValue( 0 , "------after before  advices------");
19         doSomethingElse();
20         BasicLevel.assertLevelValue( 0 , "------after around advices------");
21         doSomethingMore();
22         BasicLevel.assertLevelValue( 0 , "------after if pointcut------");
23
24     }
25
26     public void test_basicRuntimeChecks() {
27         System.out.println(
28             "\n-Testing execution Levels: runtime functionality");
29         BasicLevel.assertLevelValue( 0 , "--before a manual level up--");
30         Level.forceUnmanagedUp();
31         BasicLevel.assertLevelValue( 1 , "--after a manual level up--");
32         Level.forceUnmanagedDown();
33         BasicLevel.assertLevelValue( 0 , "--after a manual level down--");
34     }
35
36     //base level code
37      void doSomething() {
38          BasicLevel.assertLevelValue( 0 , "inside method: doSomething()");
39     }
40
41      void doSomethingElse() {
42         BasicLevel.assertLevelValue( 0 , "inside method: doSomethingElse()");
43     }
44
45     private void doSomethingMore() {
46         BasicLevel.assertLevelValue( 0 , "inside method: doSomethingMore()");
47     }
48
49     static boolean doSomeCheck(){
50         //this method serves as a condition for an if() pointcut below
51         BasicLevel.assertLevelValue(1,
```

```
52              "inside method: doSomeCheck(), (if() poincut condition)");
53          return true;
54      }
55
56
57      //helper method, prints a message and asserts the level value
58      static public void assertLevelValue(int correct_value, String message) {
59          System.out.print(message + " expected level ->"+ correct_value +"<-, ");
60          System.out.println("current level ->"+ Level.get()+"<-");
61          assertTrue( Level.get() == correct_value );
62      }
63  }
64
65  //Aspects for testing
66  aspect PcBaseLevelApplication{
67
68      pointcut match_1() : call(void BasicLevel.doSomething()) ;
69
70      before() : match_1() {
71          BasicLevel.assertLevelValue( 1 , "asp1-match1");
72      }
73
74      pointcut match_2() : call(void BasicLevel.doSomethingElse()) ;
75
76      void around() : match_2() {
77          System.out.println("*advice start*");
78          BasicLevel.assertLevelValue( 1 , "asp1-match2-preProceed");
79          proceed();
80          BasicLevel.assertLevelValue( 1 , "asp1-match2-postProceed");
81          System.out.println("*advice end*");
82      }
83
84      pointcut match_3(): call(void BasicLevel.doSomethingMore())
85                          && if(BasicLevel.doSomeCheck()) ;
86
87      after() : match_3() {
88          BasicLevel.assertLevelValue( 1 ,
89              "after match_3 advice (it's poincut has an if pointcut)");
90      }
91
92      void around() : match_3() {
93          BasicLevel.assertLevelValue( 1 ,
94              "around match_3 advice (it's poincut has an if pointcut), before proceed()");
95          proceed();
96          BasicLevel.assertLevelValue( 1 ,
97              "around match_3 advice (it's poincut has an if pointcut), after proceed()");
98      }
99
100     before() : match_3() {
101         BasicLevel.assertLevelValue( 1 ,
102             "before match_3 advice (it's poincut has an if pointcut)");
103     }
104
105 }
106
107 aspect PcBaseLevelApplicationCopy{
108     //this is just a copy of a part of PcBaseLevelApplication
109
110     pointcut match_1() : call(void BasicLevel.doSomething()) ;
111
112     before() : match_1() {
113         BasicLevel.assertLevelValue( 1 , "asp2-match1");
114     }
115
116     pointcut match_2() : call(void BasicLevel.doSomethingElse()) ;
117
118     void around() : match_2() {
119         System.out.println("*advice start*");
120         BasicLevel.assertLevelValue( 1 , "asp2-match2-preProceed");
121         proceed();
122         BasicLevel.assertLevelValue( 1, "asp2-match2-postProceed");
```

```
123         System.out.println("*advice end*");
124     }
125 }
126
127
128 aspect PcBaseLevelApplicationCopy2{
129     //this is just a copy of a part of PcBaseLevelApplication
130
131     pointcut match_1() : call(void BasicLevel.doSomething()) ;
132
133     before() : match_1() {
134         BasicLevel.assertLevelValue( 1, "asp3-match1");
135     }
136
137     pointcut match_2() : call(void BasicLevel.doSomethingElse()) ;
138
139     void around() : match_2() {
140         System.out.println("*advice start*");
141         BasicLevel.assertLevelValue( 1, "asp3-match2-preProceed");
142         proceed();
143         BasicLevel.assertLevelValue( 1, "asp3-match2-postProceed");
144         System.out.println("*advice end*");
145     }
146 }
```

BasicLevelDeploy

```
1  package abc.reentrant.test.tests;
2
3  import abc.reentrant.runtime.Level;
4  import junit.framework.TestCase;
5
6  /**
7   *  Test correct level matching.Uses 3 identical aspects,
8   *  except for that they are declared to apply to different levels.
9   *
10  * @author carlos
11  */
12 public class BasicLevelDeploy extends TestCase {
13     public static boolean flag1 = false;
14     public static boolean flag2 = false;
15     public static boolean flag3 = false;
16
17     @Override
18     public void setUp(){
19         flag1 = false;
20         flag2 = false;
21         flag3 = false;
22     }
23
24     public void test_baseLevelAdvicing() {
25         System.out.println("\n-Testing level matching when deployed on level 1");
26         doNothing();
27         assertTrue(flag1);
28         assertTrue(flag2);
29         assertFalse(flag3);
30     }
31
32     public void test_level1Advicing() {
33         System.out.println("\n-Testing level matching when deployed on level 2");
34         Level.forceUnmanagedUp();
35         doNothing();
36         assertFalse(flag1);
37         assertFalse(flag2);
38         assertTrue(flag3);
39         Level.forceUnmanagedDown();
40     }
41
```

```
42      public void doNothing(){}
43  }
44
45    //should work exactly as if it had been declared as 'onlevel 1'
46    aspect DeployOnDefaultLevel{
47        pointcut match_test() : execution(void BasicLevelDeploy.doNothing());
48
49        before() : match_test() {
50            BasicLevelDeploy.flag1 = true;
51        }
52    }
53
54    onlevel 1 aspect DeployOnLevelOne{
55        pointcut match_test() : execution(void BasicLevelDeploy.doNothing());
56
57        before() : match_test() {
58            BasicLevelDeploy.flag2 = true;
59        }
60    }
61
62    onlevel 2 aspect DeployOnLevelTwo{
63        pointcut match_test() : execution(void BasicLevelDeploy.doNothing());
64
65        before() : match_test() {
66            BasicLevelDeploy.flag3 = true;
67        }
68    }
```

AspectStackUpdating

```
1   package abc.reentrant.test.tests;
2
3   import junit.framework.TestCase;
4
5
6   /**
7    *
8    * @author carlos
9    *
10   */
11  public class AspectStackUpdating extends TestCase{
12
13      public void test_aspectStack(){
14          System.out.println("\n-Testing AspectStack correct updating");
15          doSomething();
16      }
17
18      private void doSomething() {
19      }
20
21      static public void assertTopElementIs(Object reference, String message){
22          Object currentTopElement = abc.reentrant.runtime.AspectStack.peek();
23          System.out.println(message+" top element in the aspect stack is: " +
24                             currentTopElement + " expected: "+reference);
25          assertEquals(reference, currentTopElement);
26      }
27
28      static public void assertTopElementIsNot(Object reference, String message){
29          Object currentTopElement = abc.reentrant.runtime.AspectStack.peek();
30          System.out.println(message+" asserting top element is not equal to: "+reference+
31                             " actual top element is: "+currentTopElement);
32          assertNotSame( currentTopElement, reference );
33      }
34
35      static public boolean returnTrue(){
36          return true;
37      }
38
```

```
}

aspect BasicReentrancyAdvice{

    pointcut intercept_doSomething() : call( void AspectStackUpdating.doSomething() ) ;
    pointcut trivial_if() : if( true );
    pointcut static_call_if() : if( AspectStackUpdating.returnTrue() );

    void around() : intercept_doSomething() && trivial_if() && static_call_if() {
        AspectStackUpdating.assertTopElementIs( this, "before proceed" );
        proceed();
        AspectStackUpdating.assertTopElementIs( this, "after proceed" );
    }
}
```

BasicReentrancyControl

```
package abc.reentrant.test.tests;

import junit.framework.TestCase;

/**
 *
 * @author carlos
 *
 * IDEA: trigger reentrancy by:
 *  - using level down inside an advice, and then (indirectly) generating a
 *   join point that should be matched by a pointcut inside the same aspect
 *
 *
 */
public class BasicReentrancyControl extends TestCase{

    public static boolean flag1;

    @Override
    public void setUp() {
        flag1 = false;
    }

    public void test_reentrancyControlLoops(){
        System.out.println( "\n-Testing reentrancy control: pointcut loops" );
        foo();
        //no assert here: if the test doesn't works, it degenerates into a infinite loop
    }

    public void test_reentrancyControlAdviceReentrancy(){
        System.out.println( "\n-Testing reentrancy control: same advice reentering" );
        bar();

        assertFalse(flag1);

    }

    public static void foo(){
        //do nothing
    }

    public static void bar(){
        //do nothing
    }

    public static void bar2(){
        //do nothing
    }

}

```

```
52 aspect ReentrantAspect{
53
54     pointcut matchFoo() : execution(void BasicReentrancyControl.foo());
55
56     after() : matchFoo() {
57         //this advice will generate a reentrant join point by setting the level down
58         //and calling Foo. In fact, this would cause a loop because
59         //the execution would be adviced again
60
61         abc.reentrant.runtime.Level.forceUnmanagedDown();
62         BasicReentrancyControl.foo(); //this would cause an infinite loop
63         abc.reentrant.runtime.Level.forceUnmanagedUp();
64
65     }
66
67
68     pointcut matchBar() : execution(void BasicReentrancyControl.bar());
69
70     pointcut matchBar2() : execution(void BasicReentrancyControl.bar2());
71
72     before() : matchBar() {
73         abc.reentrant.runtime.Level.forceUnmanagedDown();
74         BasicReentrancyControl.bar2();
75         abc.reentrant.runtime.Level.forceUnmanagedUp();
76     }
77
78     before() : matchBar2() {
79         //this shouldn't execute
80         BasicReentrancyControl.flag1 = true;
81     }
82
83
84 }
```

BasicLCflow

```
1  package abc.reentrant.test.tests;
2
3  import junit.framework.TestCase;
4  import abc.reentrant.runtime.Level;
5
6  /**
7   *
8   * @author carlos
9   */
10 public class BasicLCflow extends TestCase{
11     static public boolean flag1 = false;
12     static public boolean flag2 = false;
13
14     public void test_works(){
15         System.out.println("\n-Testing LCflow: basic level based matching tests");
16
17         m();
18         i();
19
20         assertTrue("lcflow activation flag", flag1);
21         assertFalse("lcflow activation flag2 should remaing in false: there" +
22                     "is a levelUp() inserted that should prevent lcflow from matching"
23                     , flag2);
24     }
25
26     static public void m() {
27         n();
28     }
29
30     static public void n(){
31
32     }
```

```
33
34      static public void i() {
35          Level.forceUnmanagedUp();
36          j();
37          Level.forceUnmanagedDown();
38      }
39
40      static public void j() {
41      }
42  }
43  aspect BasicLCflowTestASpect{
44
45      void around() : lcflow( call( void m() ) ) && call( void n() ) {
46          System.out.println("aspect1 activated, lvl:" + Level.get());
47          BasicLCflow.flag1 = true;
48          proceed();
49      }
50
51      before() : lcflow( call(void i()) ) && call( void j() ) {
52          System.out.println("aspect2 activated, lvl:" + Level.get());
53
54          BasicLCflow.flag2 = true;
55      }
56
57  }
```