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

DISEÑO E IMPLEMENTACIÓN DE PHANtom, UN LENGUAJE DE ASPECTOS PARA PHARO SMALLTALK

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DANIEL ANDRÉS GALDAMES GRÜNBERG

SEPTIEMBRE 2011

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

DISEÑO E IMPLEMENTACIÓN DE PHANtom, UN LENGUAJE DE ASPECTOS PARA PHARO SMALLTALK

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DANIEL ANDRÉS GALDAMES GRÜNBERG

PROFESOR GUÍA:

JOHAN FABRY

MIEMBROS DE LA COMISIÓN:

ÉRIC TANTER

LUIS MATEU BRULE

SANTIAGO DE CHILE

SEPTIEMBRE 2011

A mis padres, que me han apoyado siempre

Agradecimientos

A mi padres, por su incondicional apoyo en todo lo que necesité, y por respaldarme en las decisiones tomadas, aún cuando no las compartieran. Sin ellos, este trabajo no habría sido posible.

A mi hermano, por estar ahí para ayudarme cuando lo necesité, generalmente con buenos consejos :p.

A mi abuelita Rosario, por todo su amor y cariño.

A mis gatas: Sasha, Luna, Kiki, Piti, Lita, Lupita y a mi perro Rayito, por hacer mi vida más feliz.

A mi profesor guía, Johan Fabry, por ayudarme a realizar esta memoria de buena forma, estando siempre disponible durante todo el proceso.

Finalmente, a mis amigos, quienes ayudaron a formar los buenos recuerdos que siempre tendré de mi paso por la universidad.

Resumen

La programación orientada a aspectos es un paradigma de programación que intenta solucionar el problema de las funcionalidades transversales, esto es, funcionalidades de la aplicación que están dispersas por muchas áreas del código, y no pueden separarse en forma eficiente usando el paradigma de programación orientada a objetos.

Un aspecto representa una funcionalidad transversal de la aplicación. Éste incluye en su definición un *pointcut*, que representa un conjunto de puntos en la ejecución de la aplicación que van a ser capturados por el aspecto, y un *advice*, que representa la funcionalidad del aspecto, esto es, el código que va a ser ejecutado en los puntos capturados.

En este trabajo se diseñó e implementó un lenguaje de aspectos para el lenguaje de programación Pharo Smalltalk, donde se incluyeron características destacadas de otros lenguajes de aspectos, junto a nuevas funcionalidades que le otorgan un mayor control y flexibilidad al lenguaje desarrollado.

El lenguaje desarrollado incluye un sistema de definición de patrones para la definición de los *pointcuts*, reglas de precedencia globales y a nivel de *pointcut*. También contiene modificadores de clases similares a las *inter-type declarations* de AspectJ, y un sistema de control dinámico en el orden de ejecución de los *advice*.

Para el lenguaje desarrollado se implementó el concepto de membranas computacionales. Éstas son una forma de controlar el alcance que tienen los aspectos en el sistema, permitiendo controlar problemas como la reentrancia en los aspectos, esto es, cuando un aspecto captura un evento desencadenado por sí mismo. Junto a esto, las membranas computacionales son capaces de controlar la visibilidad que los aspectos tienen sobre el sistema donde son instalados.

Se creó una suite de test usando el framework para test unitarios de Pharo Smalltalk, para comprobar el correcto funcionamiento del lenguaje desarrollado, además, se comprobó el grado de cobertura de la suite de test usando el software Hapao.

Finalmente, se comprobó el funcionamiento del lenguaje de aspectos desarrollado, usándolo para refactorizar algunas partes del software de análisis dinámico de código Spy. Se realizaron algunos benchmark para comprobar el sobrecosto generado por la infraestructura de aspectos desarrollada, donde se pudo apreciar un sobrecosto variable, siendo elevado en algunos casos.

Índice

1. Introducción **6**

1.1. Objetivos 7

1.2. Estructura de la memoria 8

2. Trabajo relacionado **9**

2.1. AspectJ 9

2.1.1. Pointcuts 10

2.1.2. Advice 12

2.1.3. Inter-type declaration 14

2.1.4. Especificación de reglas de precedencia 14

2.1.5. Definición de un aspecto 15

2.1.6. Resumen elementos destacados 15

2.2. Eos-U 16

2.2.1. Pointcuts 16

2.2.2. Advice 16

2.2.3. Aspectos 16

2.2.4. Resumen de elementos destacados 17

2.3. AspectS 17

2.3.1. Pointcuts 18

2.3.2. Advice 18

2.3.3. Definición de un aspecto 20

2.3.4. Resumen de elementos destacados 21

2.4. AspectScheme 21

2.4.1. Pointcuts 21

2.4.2. Advice 22

2.4.3. Aspectos 22

2.4.4. Resumen de elementos destacados 22

2.5. Dynamic AspectJ 23

2.5.1. Ejemplo del problema 23

2.5.2. Solución del problema propuesta por Dynamic AspectJ 24

2.5.3. Ejemplo de uso 25

2.6. Membranas 25

2.6.1. Propuesta de membranas 26

2.6.2. Ejemplos de estructuras de membranas 26

2.7. Resumen 27

3. PHANtom - Descripcion del lenguaje **29**
3.1. Pointcuts . . . 29
3.1.1. Custom Parsers . . . 31
3.1.2. Exposición del contexto . . . 32
3.1.3. Restricción por Package . . . 32
3.2. Advice . . . 33
3.2.1. PhContext . . . 33
3.2.2. Acción y tipos de advice . . . 34
3.2.3. Ejemplos de advice . . . 34
3.3. Modificadores de Clase . . . 37
3.4. Aspectos . . . 38
3.5. Reglas de precedencia . . . 40
3.6. Modificación dinámica de advice . . . 43
3.7. Uso de membranas . . . 45
3.7.1. Membranas sobre flujos de ejecución . . . 46
3.7.2. Membranas sobre objetos . . . 48
3.8. Resumen . . . 49

4. Implementación de PHANtom **50**
4.1. Visión general . . . 50
4.2. Method Wrappers y CompiledMethods . . . 51
4.3. Pointcuts . . . 53
4.4. Advice . . . 54
4.5. Modificadores de Clase . . . 54
4.6. Aspectos . . . 55
4.7. Grafo de precedencias . . . 55
4.8. Weaving de los aspectos/membranas . . . 57
4.9. Estructura de membranas . . . 58
4.10. Tests . . . 60
4.11. Resumen . . . 61

5. Aplicación **63**
5.1. Spy . . . 63
5.2. Refactoring de Spy con PHANtom . . . 64
5.3. Resultados . . . 64
5.3.1. KaiProfiler . . . 65
5.3.2. MemoryProfiler . . . 67
5.3.3. Benchmark . . . 68

5.4. Discusión 69
5.5. Resumen 69

6. Conclusiones **71**

7. Trabajo futuro **72**

Índice de figuras

1. Resumen sintaxis de patrones de AspectJ, obtenido de [1] 10
2. Ejemplo de aspecto en AspectJ 15
3. Ejemplo de Aspecto en Eos-U 17
4. Aspecto en AspectS 20
5. Métodos de AspectGroup en Dynamic AspectJ, obtenido de [2] 25
6. Ejemplo de membranas sobre una computación, extraído de [3] 26
7. Estructuras de membranas, imagen extraída de [3] 27
8. Definición de una clase en Smalltalk 31
9. Grafo de precedencia de aspectos 43
10. Diagrama UML de las clases de PHANtom 51
11. Instalación MethodWrapper 52
12. Problemas en la instalación de MethodWrappers 53
13. Orden topológico de nodos 56
14. Ejemplo grafo de membranas 59
15. Test coverage con Hapao 61
16. KaiProfiler Spy 65
17. Extracto KaiProfiler Spy 66
18. Extracto KaiProfiler PHANtom-Spy 66
19. Extracto KaiProfiler MethodWrappers-Spy 66
20. Extracto MemoryProfiler Spy 67
21. Extracto MemoryProfiler PHANtom-Spy 67
22. Extracto MemoryProfiler MethodWrappers-Spy 68
23. Benchmark Spy-PHANtom 69

1. Introducción

El paradigma de programación orientada a objetos consiste en una forma de abstracción de datos y de encapsular comportamiento, con el objetivo de modularizar funcionalidades en el software que se está desarrollando. Para esto, se modelan entidades llamadas objetos, en base a elementos del mundo real o del dominio del software que se esta desarrollando. Estos objetos reúnen los datos y el comportamiento de los elementos modelados. Sin embargo, existen ciertas funcionalidades que son transversales dentro de la aplicación y no pueden separarse de forma eficiente usando este paradigma. Un ejemplo de funcionalidad transversal es el registro de eventos dentro de la aplicación (logger), que corresponde a una funcionalidad que debe colocarse en cada parte del código que genere un evento que se quiera registrar.

Una solución que se plantea para este problema es el paradigma de programación orientado a aspectos (AOP) [4]. En este paradigma se tiene un aspecto, que representa una funcionalidad transversal de la aplicación. El aspecto esta compuesto por un advice y un pointcut. El advice representa la funcionalidad del aspecto, esto es, el código a ejecutarse por el aspecto. El pointcut corresponde a la identificación de un conjunto de puntos específicos dentro de la ejecución de la aplicación. A estos puntos se les llama join points. El último componente importante es el weaver. El weaver se encarga de tomar los aspectos desarrollados e integrarlos dentro del programa, de tal forma de obtener el programa completo. De esta forma es posible encapsular estas funcionalidades transversales de la aplicación en el código, permitiendo una mejor modularización de la aplicación.

Por ejemplo, para el registro de eventos en una aplicación se podría crear un aspecto "logger", cuyo pointcut definiera todos los puntos de interés para el logger, como las llamadas a métodos de registro de usuarios y llamadas a métodos que modifiquen datos. Luego, el advice de este aspecto incluiría el código necesario para registrar estos eventos en un archivo determinado. De esta forma, al realizar el weaving del aspecto en el sistema, todos estos puntos de interés van a registrar los eventos que se requieren. Con esto se logra modularizar una función del software, que de otra forma, tendría que haberse incluido en el código de cada una de las funciones que el logger deseara registrar, duplicando código y haciendo más compleja la mantención de la aplicación.

Al aumentar la complejidad de un software, también empiezan a aparecer en mayor grado las funcionalidades transversales a la aplicación. Por esto se han desarrollado lenguajes de aspectos en otros lenguajes como AspectJ[5] para Java, Eos-U[6] para C#, AspectScheme[7] para Scheme, entre otros.

En esta memoria se presenta el diseño e implementación de un lenguaje de aspectos para Pharo Smalltalk. Smalltalk es un lenguaje de programación orientado a objetos, dinámicamente tipado y con un fuerte soporte para reflexión. Pharo [8] es un dialecto moderno de Smalltalk, open-source, y en actual desarrollo.

En Smalltalk existe AspectS[9], que es un lenguaje de aspectos para Squeak [10], un dialecto de Smalltalk del cual deriva Pharo. El desarrollo de AspectS no fue continuado, por lo cual no es posible usarlo en versiones actuales de Squeak. Además, actualizar AspectS implicaría quedar limitados por el diseño de base de AspectS, haciendo más difícil la implementación de los objetivos planteados. También, para este trabajo se quería diseñar un lenguaje que fuera simple y claro para su uso. Por lo cual, se optó por diseñar un nuevo lenguaje, en vez de actualizar AspectS.

Para el lenguaje desarrollado se han tomado en consideración distintos lenguajes de aspectos, rescatándose las características más destacadas de éstos, incorporando además nuevas funcionalidades, con el fin de otorgarle al usuario mayor flexibilidad y control a la hora de aplicarlo.

Las funcionalidades más destacadas de PHANtom son:

- Control de precedencia de aspectos a nivel global y a nivel de pointcut, permitiendo un mayor grado de control en el orden de los aspectos.
- Control dinámico de los advice, permitiendo que un aspecto pueda acceder a la lista de advice a ejecutarse en un join point determinado, pudiendo agregar, eliminar y cambiar el orden de los advice para ese join point.
- Uso de membranas computacionales, aumentando el control que se tiene sobre el alcance de los aspectos en el sistema, permitiendo resolver problemas como el de reentrancia de aspectos.
- Sistema de definición de pointcut flexible, que permite a usuarios usar sus propios parser para definir un pointcut.

1.1. Objetivos

En este trabajo se pretende implementar un lenguaje de aspectos para AOP en Pharo Smalltalk, integrando características destacadas de otros lenguajes de aspectos actuales, junto a nuevos avances en el área de AOP, logrando un lenguaje de aspectos flexible y que permita al usuario especificar de manera clara los aspectos implementados.

Objetivos específicos

- Diseñar las especificaciones del lenguaje para aspectos, pointcuts y advice
- Revisar otros lenguajes de aspectos e incorporar elementos destacados de éstos al lenguaje diseñado.
- Implementar el weaving de los aspectos, incluyendo elementos de control de reentrancia.

- Aplicar el lenguaje desarrollado y analizar los resultados

1.2. Estructura de la memoria

En la sección 2 se presentará el trabajo relacionado con el realizado en esta memoria, y los aspectos de éste que fueron usados para el trabajo desarrollado. En la sección 3 se presentará el lenguaje que fue desarrollado durante este trabajo, junto a ejemplos de uso. En la sección 4 se mostrarán detalles sobre la implementación del lenguaje. En la sección 5 se expondrá una experiencia de uso del lenguaje, junto a algunas pruebas realizadas. En la sección 6 se presentan las conclusiones obtenidas de esta trabajo, y finalmente, en la sección 7 se presentan posibilidades de continuación del trabajo desarrollado.

2. Trabajo relacionado

Para el diseño del lenguaje, se tomaron como inspiración ciertas características importantes de otras implementaciones de aspectos en diferentes lenguajes. A continuación se explica cada uno de los lenguajes considerados. En la sección 2.1 se presentará el lenguaje AspectJ, uno de los lenguajes de aspectos más usados actualmente. Luego en la sección 2.2 presentaremos Eos-U, un lenguaje de aspectos similar a AspectJ, que rompe la asimetría entre clases y aspectos. A continuación en la sección 2.3 se mostrará AspectS, el primer lenguaje en presentar aspectos dinámicos que pueden ser desplegados y retirados en tiempo de ejecución. Seguido a lo anterior, en la sección 2.4 se verá AspectScheme, una implementación de aspectos para el lenguaje Scheme. Para continuar, en la sección 2.5 se expondrá Dynamic AspectJ, una extensión de AspectJ que permite controlar los aspectos que serán ejecutados en un join point de forma dinámica. Finalmente, en la sección 2.6, se presentará el concepto de membranas computacionales, y su aplicación en lenguajes de aspectos.

2.1. AspectJ

AspectJ [4, 5] es uno de los lenguajes de aspectos más conocidos actualmente, y consiste en una implementación de aspectos para el lenguaje de programación Java. Dentro de las principales características de AspectJ destacan: la posibilidad de usar patrones para definir los pointcuts, permitir establecer reglas de precedencia para las interacciones entre aspectos, y la capacidad para agregar nuevos métodos y variables de estado a las clases existentes (*inter-type declarations*).

El weaving de los aspectos en AspectJ se realiza en tiempo de compilación. En el modelo de join points de AspectJ, los principales puntos son [1]:

- **method call/execution:** corresponde a la llamada/ejecución de un método.
- **constructor call/execution:** corresponde a la llamada/ejecución de un constructor.
- **field reference:** referencia a una variable de instancia.
- **field set:** asignación de un valor a una variable de instancia.
- **handler execution:** ejecución de una excepción.
- **advice execution:** ejecución de un advice.

A continuación se realizará una breve descripción del lenguaje y algunos ejemplos de uso. Los ejemplos aquí presentados han sido extraídos de *The AspectJ Programming Guide [1]*

2.1.1. Pointcuts

Para definir los join point que un determinado pointcut debe capturar, AspectJ utiliza un sistema de patrones, permitiendo especificar un conjunto de joinpoints para los cuales el patrón coincide.

Un patrón está formado por elementos de la firma de los elementos a capturar (especificados por el modelo de join points), junto a comodines que permiten ampliar el rango de elementos que coinciden con el patrón.

En esta sección sólo vamos a hablar sobre call y execution de métodos y constructores, ya que son los más relevantes para el lenguaje de aspectos diseñado en este trabajo.

```
MethodPattern =
   [ModifiersPattern] TypePattern
         [TypePattern . ] IdPattern (TypePattern | ".." , ... )
         [ throws ThrowsPattern ]
ConstructorPattern =
   [ModifiersPattern ]
         [TypePattern . ] new (TypePattern | ".." , ...)
         [ throws ThrowsPattern ]
FieldPattern =
   [ModifiersPattern] TypePattern [TypePattern . ] IdPattern
ThrowsPattern =
   [ ! ] TypePattern , ...
TypePattern =
     IdPattern [ + ] [ [] ... ]
    | ! TypePattern
    | TypePattern && TypePattern
    | TypePattern || TypePattern
    | ( TypePattern )
IdPattern =
  Sequence of characters, possibly with special * and ..
     wildcards
ModifiersPattern =
  [ ! ] JavaModifier  ...
```

Figura 1: Resumen sintaxis de patrones de AspectJ, obtenido de [1]

La forma más simple de pointcut corresponde a la definición con la firma exacta del método que se quiere capturar, por ejemplo: `call(void Point.setX(int))`

Usando comodines para partes de la firma del método, se puede ampliar el alcance del pointcut que se esta especificando. Para esto se pueden utilizar los siguientes comodines:

- * Este comodín captura cualquier secuencia de caracteres, exceptuando el separador "."
- + Este comodín captura el tipo especificado en el sub patrón que lo precede, más todos los subtipos de éste.
- .. Este comodín se usa para especificar cero o más parámetros en un método

A continuación algunos ejemplos de uso de patrones en un pointcut.

- Ejecución de cualquier método que no reciba parámetros, y que retorne un *int*: `execution (int *())`
- Llamada a un método llamado "SetY" que toma como parámetro un *long*, independiente del tipo de retorno: `call(* setY(long))`
- Llamada a cualquier constructor de la clase "Point", independiente del número de argumentos: `call(Point.new(..))`
- Llamada a cualquier constructor de la clase "Point" y sus subclases: `call(Point+.new())`

Composición de Pointcuts

AspectJ permite componer pointcuts usando operadores booleanos. Los operadores permitidos son: **or** (`||`), **and** (`&&`) y **not** (`!`). Por ejemplo, en el siguiente pointcut identificamos los llamados a cualquier método "SetX" que reciba un *int* o cualquier método "SetY" que reciba un *int*: `call(void setX(int)) || call(void setY(int))`.

El designador *within* puede usarse para especificar que el código que se va a ejecutar, está definido en el tipo especificado. En el siguiente ejemplo, definimos un pointcut que captura un llamado a cualquier método que esté definido en la clase Line o en la clase Point: `call(* *(..)) && (within(Line) || within(Point))`.

Exposición del contexto

AspectJ provee herramientas para exponer ciertos elementos del contexto en el cual están ocurriendo los join points capturados y, de esta manera, permitir su uso en un advice. El designador *this* representa el objeto que se esta ejecutando actualmente, y *target* representa al objeto al cual se transferirá el control. El designador *args* puede ser usado para especificar el tipo de argumentos que debe recibir una método.

Por ejemplo, el siguiente pointcut especifica una llamada a cualquier método que retorne un int, y que no reciba argumentos, cuyo receptor sea una instancia de la clase Point: `target(Point) && call(int *())`.

Llamada a cualquier método que retorne un int, que reciba cero o más argumentos, y cuyo receptor no sea una instancia de la clase Point: `!this(Point) && call(int *(..))`.

Llamada a cualquier método que retorne un int, que reciba cero o más argumentos, y que los argumentos sean de tipo *int*: `call(int *(..)) && args(int)`.

Named pointcuts

A diferencia de los pointcuts de los ejemplos anteriores, llamados pointcut anónimos, un pointcut puede ser definido en una clase o un aspecto, usando el designador *pointcut*. De esta forma, el pointcut puede ser referenciado con un nombre, para ser utilizado luego en la definición de un advice. Este pointcut así definido, pasa a formar parte de la clase o aspecto. Usando esta definición de pointcut, se pueden usar los identificadores especificados en *target, this,* y *args*. De esta forma, es posible usarlos luego en el cuerpo del advice que use este pointcut.

En el siguiente ejemplo se define un pointcut llamado "setXY" y los identificadores "fe" de tipo *FigureElement*, "x" de tipo *int* e "y" de tipo *int*. Luego, usando *target,* se liga el identificador "fe" con el objeto sobre el cual se está llamando el método, usando *args* se liga x e y con los argumentos del método.

```
pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);
```

2.1.2. Advice

Un advice en AspectJ puede ejecutarse de tres formas distintas:

- **before:** El advice se ejecuta antes de ejecutar el código capturado por el pointcut.
- **after:** Después de ejecutado el código capturado por el pointcut.
- **around:** El advice es ejecutado reemplazando el código capturado por el pointcut, además, se puede usar la instrucción *proceed*, para ejecutar la acción que iba a ser realizada originalmente.

La definición del advice puede realizarse usando pointcuts anónimos, o pointcuts previamente definidos usando *pointcut* como se explicó anteriormente.

Ejemplo de before advice:

En este ejemplo se define un pointcut llamado "setter". Este pointcut captura todas las llamadas a métodos con la firma `void setX(int)` o `void setY(int)`, en el cual el objeto al cual se realiza la llamada sea de tipo `Point`. Usando la definición del pointcut, se liga el símbolo *p1* con el objeto al cual se está realizando la llamada, y el símbolo *newval* con el

argumento del método. Luego se define un advice usando el pointcut previamente definido, en el cual se utilizan los valores del contexto expuestos para imprimir un mensaje haciendo referencia al objeto en el cual se está realizando el llamado, y los argumentos que se usarán.

```
pointcut setter(Point p1, int newval): target(p1) && args(newval)
        (call(void setX(int)) ||
         call(void setY(int)));

  before(Point p1, int newval): setter(p1, newval) {
      System.out.println("About to set something in " + p1 +
                         " to the new value " + newval);
  }
```

Ejemplo de after advice:

En este ejemplo se usa un pointcut anónimo, que captura las llamadas al método con la firma `void setX(int)`, en el cual el objeto al que se realiza la llamada sea de tipo `Point`. Usando la definición, se liga el símbolo p con el objeto al que se realiza la llamada, y el símbolo x con el argumento del método. Luego en el cuerpo del advice se verifica realizando un llamado al método *assertX* del objeto p, y según el resultado de este llamado, lanzar una excepción.

```
after(Point p, int x): target(p) && args(x) && call(void setX(int)
   ) {
      if (!p.assertX(x)) throw new PostConditionViolation();
  }
```

Ejemplo de around advice:

Un around advice, al ser ejecutado en reemplazo del código que está capturando, debe especificar un tipo de retorno que sea compatible al del código capturado y debe retornar un objeto con ese tipo.

En este ejemplo se están capturando los llamados al método con la firma `void setX(int)`, en los cuales el objeto que recibe el llamado sea de tipo `Point`. Usando la definición, se liga el símbolo p con el objeto que recibe el llamado, y el símbolo x con el argumento del método. Luego se realiza un llamado al método *assertX* del objeto p, si el resultado de esto es *true*, se realiza un llamado a *proceed* lo que ejecutará el llamado al método *setX* que había sido capturado. Luego se realizará un llamado al método *releaseResources* del objeto p.

```
void around(Point p, int x): target(p)
            && args(x)
```

```
            && call(void setX(int)) {
    if (p.assertX(x)) proceed(p, x);
    p.releaseResources();
}
```

2.1.3. Inter-type declaration

Esta es una forma que provee AspectJ para agregar variables, métodos y constructores a una clase. También es posible declarar que alguna clase implementa una interfaz determinada, o hacer que extienda a alguna otra.

A continuación se presentarán algunos ejemplos de inter-type declaration.

Declarar que cada instancia de "Server" tiene una variable de instancia llamada "disabled" inicializada como *false*: `private boolean Server.disabled = false;`

Declarar que cada instancia de la clase "Point" tiene un método llamado "getX()" que retorna una variable del objeto llamada "x": `public int Point.getX() { return this.x; }`

Declarar que la clase "Point" extiende a la clase "GeometricObject": `declare parents: Point extends GeometricObject;`

2.1.4. Especificación de reglas de precedencia

Es posible que varios aspectos activos en el sistema tengan advice que comparten un join point, por lo cual puede ser necesario declarar algún orden de precedencia, para que algunos advice siempre se ejecuten antes que otros. AspectJ permite especificar un orden de precedencia para los aspectos, de tal forma que los advice que pertenecen a los aspectos con mayor precedencia, se ejecuten antes que aquellos con menor precedencia.

Para declarar la precedencia, se utiliza la orden *declare precedence : TypePattern list.*

Es posible utilizar patrones para especificar la precedencia, ver *TyperPattern* en figura 1.

Por ejemplo, lo siguiente declara que todos los aspectos que tengan "Security" como parte de su nombre tiene la precedencia más alta, luego están los aspectos que sean de tipo Logging o subtipos de éste, y finalmente, el resto de los aspectos. Los aspectos con precedencia más alta ejecutan los advice de tipo *before* y *around* antes que los de precedencia más baja, y los de tipo *after* después de los con precedencia más baja.

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

AspectJ verifica que no se creen conflictos en la declaración de precedencia dentro de un aspecto, por ejemplo: `declare precedence: A, B, A` no esta permitido, ya que esto genera un ciclo. Es posible que dos o más aspectos activos en el sistema, tengan reglas de precedencia

que estén en conflicto entre ellos, siempre que no tengan algún advice que comparta un join point.

2.1.5. Definición de un aspecto

En AspectJ los aspectos se definen de una manera similar a una clase de Java, sin embargo, los aspectos son entidades separadas a las clases de Java. AspectJ se encarga de instanciar los aspectos, según las especificaciones de éste.

Un aspecto de AspectJ puede contener métodos y variables de instancia, al igual que una clase de Java, pero junto a esto, incluye la definición de los pointcuts, advice e inter-type declarations.

```
aspect Logging {
    OutputStream logStream = System.err;

        pointcut move():
        call(void FigureElement.setXY(int,int)) ||
        call(void Point.setX(int))              ||
        call(void Point.setY(int))              ||
        call(void Line.setP1(Point))            ||
        call(void Line.setP2(Point));

    before(): move() {
        logStream.println("about to move");
    }
}
```

Figura 2: Ejemplo de aspecto en AspectJ

2.1.6. Resumen elementos destacados

Los aspectos más destacados de AspectJ, que sirvieron de inspiración para la creación del nuevo lenguaje son:

Uso de patrones

El uso de patrones en los pointcuts de AspectJ permite capturar en una expresión compacta, una gran cantidad de join points, permitiendo una definición clara de lo que se quiere capturar al escribir un pointcut.

Inter-type declarations

La capacidad para agregar variables y comportamiento a clases existentes del sistema,

modularizado dentro de un aspecto, es otra de las características importantes que se tomaron en consideración para la creación del lenguaje.

Reglas de precedencia
La especificación de reglas de precedencia para los aspectos, junto al uso de patrones para estas reglas, permite especificar en cada aspecto la forma en que debe interactuar con otros, respecto a su orden de ejecución.

2.2. Eos-U

Eos-U [6] es una implementación de aspectos en C#, similar a AspectJ, que introduce el concepto de *classpects*, con el fin de evitar la separación entre clases y aspectos que presenta AspectJ. De esta forma, los aspectos pasan a ser entidades de primera clase en el lenguaje, permitiendo que un aspecto pueda ser instanciado, y pueda controlar lo que hacen otros aspectos, de manera similar a como en AspectJ un aspecto puede controlar a las clases.

2.2.1. Pointcuts

En Eos-U los pointcuts son similares a lo presentados en AspectJ, permitiendo el uso de patrones, junto a información del contexto tanto para exponerlo al advice o para especificar los join point a capturar.

2.2.2. Advice

En Eos-U no existen los advice como un concepto separado, en cambio, cualquier método de la clase puede ser un advice. Lo que se hace para usar algún método como advice, es ligar un método junto a la especificación de cuando este método debe ser ejecutado (pointcut). Al igual que en AspectJ, se puede especificar la ejecución de código como *after, before* y *around*.

Como se puede ver en la figura 3, se está especificando que cuando se captura una ejecución de un método "bar" que reciba cero o más argumentos y que retorne cualquier tipo, en la clase "Foo", se llame al método "foo()" después de la ejecución.

2.2.3. Aspectos

En Eos-U, los aspectos son clases del sistema (llamadas *classpects*). Una clase actuará como aspecto, si internamente tiene definido algún advice.

Como se puede ver en este ejemplo, un aspecto se declara de forma similar a cualquier otra clase.

```
public class G {
  void foo() {...}
  static void before execution(* Foo.bar(..)) : call foo();
}
```

Figura 3: Ejemplo de Aspecto en Eos-U

2.2.4. Resumen de elementos destacados

Lo que más destaca en Eos-U es que rompe la asimetría entre aspectos y clases, existente en lenguajes como AspectJ. De esta forma, los aspectos pasan a ser entidades de primera clase en el lenguaje, pudiendo tener aspectos que observan a otros aspectos, de la misma forma que lo hacen con las otras clases del sistema.

2.3. AspectS

AspectS [9] es una implementación de aspectos para Squeak Smalltalk. A diferencia de lo que ocurre en AspectJ y Eos-U, en AspectS el weaving de los aspectos ocurre en forma dinámica en tiempo de ejecución y no en tiempo de compilación. AspectS es el primera instancia en que se tienen aspectos dinámicos, donde los aspectos pueden ser desplegados y retirados en forma dinámica en tiempo de ejecución, a diferencia de lo que ocurre en AspectJ. Para realizar esto, AspectS hace uso de las capacidades reflexivas de Smalltalk, utilizando el *meta object protocol*.

Una de las características interesantes de AspectS, es el uso de *method wrappers*[11]. Los *method wrappers* son una forma de "envolver" un método compilado en Smalltalk y colocar el *method wrapper* en el diccionario de métodos de la clase. De esta forma, cuando se realiza la búsqueda del método en el diccionario de la clase, lo que se devuelve es el *method wrapper*, permitiendo interceptar la ejecución del código original, y modificar el comportamiento.

Tomando en cuenta que en Smalltalk, los objetos del sistema interactúan a través del envío de mensajes, el modelo de join point de AspectS se basa en la recepción de un mensaje por un objeto.

A continuación se realizará una breve descripcion del lenguaje, junto con algunos ejemplos. Los ejemplos aqui presentados han sido extraídos desde *AspectS - Aspect-Oriented Programming with Squeak [9]*, y desde los ejemplos incluidos con el paquete de AspectS para Squeak.

2.3.1. Pointcuts

Para definir un pointcut en AspectS, se debe crear una instancia de la clase AsJoinPointDescriptor, especificando el mensaje (*selector*) y la clase que lo recibirá. A diferencia de AspectJ, aquí no existe un sistema de patrones para generalizar los join points que el pointcut debe capturar, estos deben ser entregados explícitamente, haciendo referencia a las clases del sistema que definen a los receptores del mensaje. En otras palabras, para obtener el conjunto de join points que se quiere definir se deben usar las facilidades reflexivas del lenguaje Smalltalk, haciendo consultas al sistema.

Por ejemplo, si se quiere capturar todas las clases de Morph, y sus sub clases, que responden al mensaje "position:": Primero, se le pide a la clase Morph que entregue el listado de todas sus sub clases, incluyéndose, lo que genera una colección de referencias a las clases. Luego usando el método "select:thenCollect:" implementado en las colecciones de Smalltalk, usando el método "includesSelector:" se le pregunta a la clase si su diccionario de métodos incluye el método "#position:". Esto genera una colección con las clases que contienen ese método en sus diccionarios de métodos. Finalmente, usando esta colección de clases, se genera una nueva colección con las instancias de la clase AsJoinPointDescriptor.

```
Morph withAllSubclasses
        select: [:each | each includesSelector: #position:]
        thenCollect: [:each | AsJoinPointDescriptor
                        targetClass: each
                        targetSelector: #position:]]
```

2.3.2. Advice

Un advice es una instancia de la clase AsAdvice. En AspectS existen cuatro tipos de AsAdvice:

- **AsBeforeAfterAdvice:** Permite la ejecución de código antes o después de la ejecución del método capturado por el join point.
- **AsAroundAdvice:** Permite la ejecución de código en lugar del código capturado.
- **AsIntroductionAdvice:** Permite agregar nuevos métodos y variables a las clases capturadas.
- **AsHandlerAdvice:** Permite capturar excepciones.

A continuación se presentan algunos ejemplos de advice:

BeforeAfter advice

Este advice captura los llamados al método "mouseLeave:" en la clase "Morph" y todas sus sub clases, e imprime un mensaje especificando el evento ("MouseLEAVE"), el receptor del mensaje y los argumentos.

```
AsBeforeAfterAdvice
 qualifier: (AsAdviceQualifier
  attributes: { #receiverClassSpecific. #projectSpecific. })
 pointcut: [
  Morph withAllSubclasses
   select: [:each | each includesSelector: #mouseLeave:]
   thenCollect: [:each |
    AsJoinPointDescriptor
     targetClass: each
     targetSelector: #mouseLeave:]]
 beforeBlock: [:receiver :arguments :aspect :client |
  self showHeader: '<<< MouseLEAVE <<<'
   receiver: receiver
   event: arguments first]
```

Around advice

En este advice se captura la llamada al método "new" en la clase "AsSingleton class", y se genera un error para especificar que la clase es un Singleton y no debe ser instanciada con new.

```
AsAroundAdvice
 qualifier: (AsAdviceQualifier attributes: { #
    receiverClassSpecific. })
 pointcut: [{ AsJoinPointDescriptor
  targetClass: AsSingleton class
  targetSelector: #new. }]
 aroundBlock: [:receiver :arguments :aspect :client :clientMethod
    |
  receiver error: 'Singleton. Use #soleInstance instead.']
```

Introduction advice

En este advice se agrega el método "soleInstance" a la clase "AsSingleton class".

```
AsIntroductionAdvice
```

```
qualifier: (AsAdviceQualifier attributes: { #
    receiverClassSpecific. })
pointcut: [{ AsJoinPointDescriptor
 targetClass: AsSingleton class
 targetSelector: #soleInstance. }]
 introBlock: [:receiver :arguments :aspect :client |
 aspect annotationsFor: receiver
  at: #soleInstance
  ifAbsentPut: receiver basicNew initialize]
```

2.3.3. Definición de un aspecto

En AspectS los aspectos son como cualquier otra clase del sistema. Los aspectos pueden ser instanciados, y la activación y desactivación de éstos es controlada a través del envío de mensajes a sus instancias.

Para crear un aspecto en AspectS, se debe crear una sub clase de AsAspect. Esta clase puede contener variables y métodos como cualquier otra clase del sistema. Para desplegar un aspecto en el sistema, se le envía el mensaje *install* a una instancia del aspecto. Para retirarlo del sistema, se le envía el mensaje *uninstall* a una instancia de un aspecto instalado.

Para definir los advice que el aspecto incluye, se deben crear métodos que no reciban argumentos y cuyo nombre comience por "advice". Este método debe retornar un objeto de tipo AsAdvice.

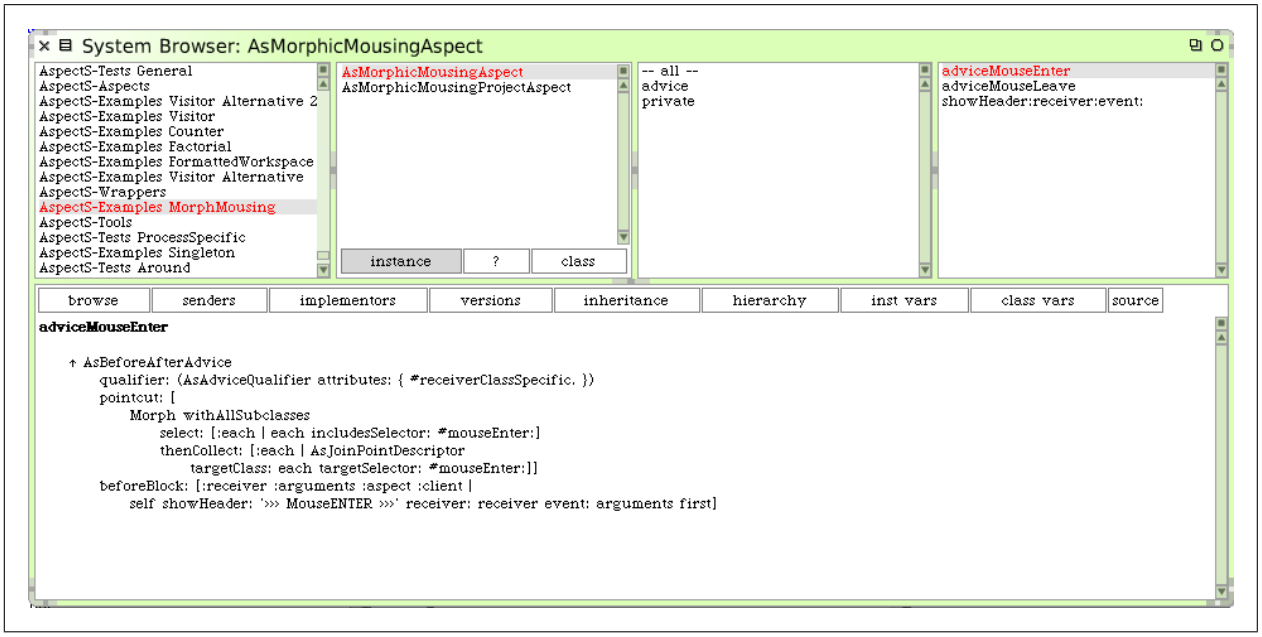

Figura 4: Aspecto en AspectS

2.3.4. Resumen de elementos destacados

Los elementos importantes que se tomaron en consideración de AspectS son: El uso de *method wrappers* para realizar el weaving de los aspectos en tiempo de ejecución, permitiendo el despliegue de los aspectos en forma dinámica dentro del sistema. Junto a esto, el uso de las capacidades reflexivas del lenguaje Smalltalk para realizar cambios dentro de éste, permitiendo desplegar y sacar los aspectos en forma dinámica, enviando los mensajes *install/uninstall* respectivamente.

2.4. AspectScheme

AspectScheme [7] es una implementación de aspectos en el lenguaje funcional Scheme, un dialecto de Lisp. En AspectScheme, aspectos, pointcuts y advice son valores de primera clase en el lenguaje. Esto permite que se puedan usar las funciones de orden superior existentes en Scheme para manipular esos elementos, dándole una gran flexibilidad al lenguaje.

En AspectScheme, al ser un lenguaje funcional, el modelo de join point corresponde a la aplicación de una función (*procedure*), la cual puede estar en el contexto de otra aplicación de función.

A continuación se presentará brevemente el lenguaje, junto a algunos ejemplos. Los ejemplos han sido extraídos desde *Semantics and Scoping of Aspects in Higher-Order Languajes*[7].

2.4.1. Pointcuts

Tomando en cuenta el modelo de join point propuesto, un pointcut en AspectScheme, es una función que toma como parámetro una lista de join points, retornando un *true* o *false*, dependiendo si quiere capturar ese join point o no.

Por ejemplo, un pointcut que verifica que la aplicación de función actual es "close-file", y que está dentro de la aplicación de la función "write-contents"

```
(lambda (jp*)
 (and (eq? close-file (first jp*))
      (not (empty? (rest jp*)))
      (eq? write-contents (second jp*))))
```

De esta forma pueden ser definidos pointcuts similares a los de otros lenguajes, por ejemplo el de llamado a una función:

```
(define ((call f) jp*)
  (eq? f (first jp*)))
```

2.4.2. Advice

Un advice va a tomar como parámetros el join point (que es una función) y sus argumentos, y va a aplicar una función en ese contexto.

AspectScheme define la primitiva *app/prim* que tiene la misma semántica que *proceed* de los lenguajes mencionados anteriormente.

A continuación presentamos un ejemplo de advice que verifica si el argumento es menor a 0, y en ese caso retorna un string vacío, o en caso contrario aplica la función original.

```
(lambda (jp)
 (lambda (a)
   (if (<= a 0)
     ""
     (app/prim jp a))))
```

2.4.3. Aspectos

En un aspecto en AspectScheme, se va tomar un pointcut y un advice, y éste va a ser aplicado sobre un *body*. Para esto, AspectScheme define la expresión (AROUND *pointcut advice body*).

A continuación, un ejemplo de un aspecto que captura la llamada a la función "open-file" y aplica el advice "trace-advice" que imprime el mensaje "Calling open-file" y luego continua con la aplicación de la función original.

```
(let ([open-file (lambda(f ) ...)]
  [trace-advice (lambda (jp)
   (lambda (a)
    (printf "Calling open-file")
    (app/prim jp a)))])
 (around (call open-file) trace-advice
  (list (open-file "vancouver")
        (open-file "whistler"))))
```

2.4.4. Resumen de elementos destacados

De AspectScheme destaca el hecho de que todos los elementos de los aspectos (aspectos, advice, pointcuts y joinpoints) son elementos de primera clase del sistema, lo que da una gran flexibilidad y poder para combinar cada uno de estos elementos.

2.5. Dynamic AspectJ

Dynamic AspectJ [2] es una extensión a AspectJ para permitir controlar el orden de ejecución de los aspectos en forma dinámica.

Uno de los problemas que se presentan al tener múltiples aspectos actuando en el sistema, es que existen ocasiones en que es necesario poder controlar en forma dinámica cuales aspectos se ejecutaran en un determinado pointcut, y en que orden se ejecutarán. En estos casos, a veces no es suficiente poder especificar un orden estático, como en el caso de las reglas de precedencia que provee AspectJ.

2.5.1. Ejemplo del problema

A continuación se presentará un ejemplo de estos casos, extraído de [2]. Supongamos que se tiene una aplicación cliente-servidor, para alojamiento de archivos. En esta aplicación, los clientes envían archivos llamando al método *send*. El servidor por su parte, revisa los archivos recibidos en busca de virus, llamando al método *virusCheck*.

Se desea extender esta aplicación con el fin de hacerla más eficiente y segura, para lo cual se crean los siguientes aspectos:

- **CompressUpload:** Este aspecto se encargará de comprimir los archivos que el cliente envía, descomprimiéndolos luego en el servidor para que puedan ser revisados por virus.
- **SecureUpload:** Este aspecto se encargará de encriptar los archivos que el cliente envía, haciendo más segura la transmisión de éste. En el servidor, el archivo será desencriptado para realizar la búsqueda de virus.

```
public aspect CompressUpload {
  before(File f):
    call(* Client.send(..)) && args(f) {
      zip(f);
    }
  before(File f):
    call(* Server.virusCheck(..)) && args(f) {
      unzip(f);
    }
}

public aspect SecureUpload {
  before(File f):
    call(* Client.send(..)) && args(f){
```

```
      encrypt(f);
    }
  before(File f):
    call(* Server.virusCheck(..)) && args(f) {
      decrypt(f);
    }
}
```

El orden en el que los aspectos son ejecutados tiene un alto impacto en lo que la aplicación está realizando. Supongamos que establecemos una regla de precedencia en la cual *CompressUpload* siempre se ejecuta antes que *SecureUpload*. Esto significa que los archivos en el lado del cliente, son primero comprimidos, y luego encriptados. Sin embargo, en el lado del servidor, los archivos serán primero descomprimidos y luego desencriptados, lo que es incorrecto para lo que la aplicación requiere.

2.5.2. Solución del problema propuesta por Dynamic AspectJ

Para solucionar el problema anterior, Dynamic AspectJ permite obtener la lista de aspectos que se van a ejecutar en un join point particular, usando la variable de instancia *thisAspectGroup*, dentro de un advice. Usando esta lista, se provee de una serie de métodos para manipularla, y de esta forma controlar los aspectos que serán ejecutados en ese punto. En la figura 5 se pueden ver los métodos provistos en AspectGroup.

- **skipAll:** Permite saltarse todos los aspectos que se iban a ejecutar a continuación.
- **skip:** Permite quitar de la lista de aspectos a ejecutarse al aspecto con el nombre especificado.
- **skipFirst:** Permite quitar de la lista al aspecto que se encuentra al inicio de la lista.
- **setFirst:** Permite colocar al aspecto especificado al inicio de la lista de aspectos a ejecutarse.
- **member:** Permite preguntar si el aspecto especificado está en la lista (esto es, si va a ser ejecutado).

```
public void skipAll () { ... }
public boolean skip ( String name ) { ... }
public void skipFirst () { ... }
public boolean setFirst ( String name ) { ... }
public boolean member ( String name ) { ... }
```

Figura 5: Métodos de AspectGroup en Dynamic AspectJ, obtenido de [2]

2.5.3. Ejemplo de uso

A continuación de presenta un ejemplo de uso, para resolver el problema planteado anteriormente.

En este ejemplo, se crea un aspecto con los mismos pointcuts que *SecureUpload* y *CompressUpload*, luego usando *thisAspectGroup* se establece que cuando el cliente quiera subir un archivo, siempre se ejecutará primero *CompressUpload*. Tomando esto en cuenta, en el lado del servidor, siempre se ejecutará primero *SecureUpload*.

Con esto, cada vez que un cliente envíe un archivo, éste será comprimido y luego encriptado. En el lado del servidor, un archivo siempre será primero descencriptado y luego descomprimido, antes de ser revisado en busca de virus.

```
public aspect SecureCompression {
  before ( File f ) :
    call (* Client.send (..) ) && args ( f ) {
      thisAspectGroup . setFirst ( "CompressUpload") ;
  }

  before ( File f ) :
    call (* Server.virusCheck (..) ) && args ( f ) {
      thisAspectGroup . setFirst ( "SecureUpload") ;
    }
}
```

2.6. Membranas

Uno de los problemas que tienen los aspectos, es el alcance que tienen dentro del sistema una vez que éstos son instalados. Uno de los problemas más comunes que esto genera, es el problema de la reentrancia, esto ocurre por ejemplo, cuando un aspecto captura un join point generado por la ejecución del mismo aspecto.

2.6.1. Propuesta de membranas

Las membranas computacionales [3] son una forma de controlar el alcance que tienen los aspectos, envolviendo partes de la computación de un programa dentro de ellas, y de esta forma controlando la visibilidad que la computación dentro de ella tiene. Cuando se coloca una membrana sobre una computación, los join points que esta membrana genera serán visibles sólo para las membranas que estén conectadas a ella, por tanto, no serán visibles para si misma, a menos que se establezca una conexión hacia si misma.

En la figura 6 se puede ver un ejemplo de membranas sobre una computación, en este ejemplo se tienen 3 membranas, *m1*, *m2* y *m3*. La membrana *m1* esta desplegada sobre la computación de x e y, la membrana *m2* sobre la computación de y y z, y el aspecto A está registrado en la membrana *m3*. La membrana *m3* está conectada a *m1*. Los join points que se generen dentro de *m1* serán visibles para el aspecto que está en *m3*, mientras que los join points generados en la parte de *m2* que no se comparte con *m1*, no serán visibles para *m3*.

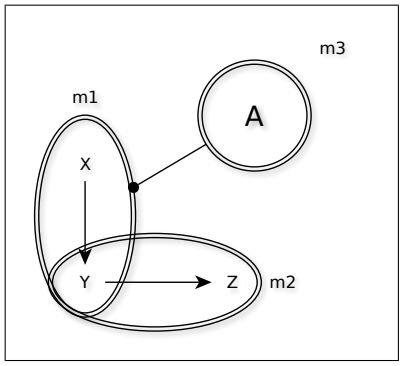

Figura 6: Ejemplo de membranas sobre una computación, extraído de [3]

Una de las aplicaciones de esto es controlar la reentrancia que se produce en los aspectos, ya que al envolver la computación de un aspecto dentro de una membrana, los join points que se generen dentro de la computación de éste, no serán visibles para si mismo. Además las membranas proveen una gran flexibilidad en cuanto a la estructura que puede generarse para controlar la visibilidad de una computación, ya que el sistema de membranas permite generar estructuras de grafos sobre una computación.

2.6.2. Ejemplos de estructuras de membranas

En la figura 7 se pueden ver distintos ejemplos de las estructuras que se pueden formar usando membranas. En este ejemplo se tiene un Browser, un aspecto *Cache* que realiza el cache de las páginas que Browser visita, y el aspecto *Quota* que restringe la cantidad de espacio en disco que se puede usar.

En *a* se tiene una membrana desplegada sobre la computación de *Browser*, otra membrana sobre ésta contiene al aspecto *Cache*, y sobre ésta, otra conteniendo al aspecto *Quota*. De

esta forma se logra que *Cache* solo vea los join points generados por *Browser*, y el aspecto *Quota* solamente captura los join points generados por *Cache*, sin afectar a los join points de *Browser.*

En b se tienen dos membranas con un aspecto *Quota*, una observando a Browser y otra observando a *Cache.* Esto permite que se controle una cuota en disco distinta para lo que hace *Cache* y para lo que hace *Browser*

Finalmente en c se tiene una sola membrana con el aspecto *Quota* que observa tanto a Browser como a *Cache.* Esto permite que tanto lo que hace *Browser*, como *Cache*, sume para el control de la cuota que realiza *Quota*.

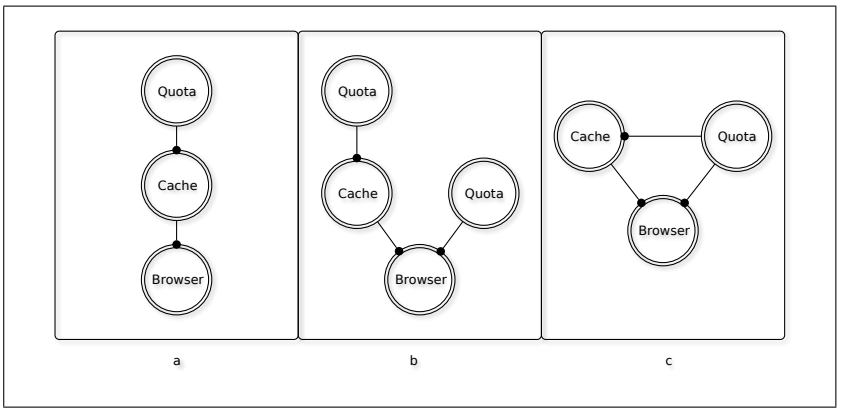

Figura 7: Estructuras de membranas, imagen extraída de [3]

2.7. Resumen

En esta sección se revisaron los lenguajes de aspectos considerados importantes para el lenguaje que se desarrolló.

En primer lugar se presentó el lenguaje AspectJ, que corresponde a una implementación de aspectos para Java, destacándose su sistema de patrones para *pointcuts*, permitiendo describir en expresiones compactas un gran número de *join points*. También se destacaron las *inter-type declarations* que permiten agregar variables y comportamiento a clases del sistema, desde un aspecto. Finalmente se describieron las reglas de precedencia de AspectJ, permitiendo definir un orden en la ejecución de los aspectos, y para las cuales también es posible hacer uso del sistema de patrones.

El siguiente lenguaje presentado fue Eos-U, una implementación de aspectos en C#, donde se destacó el romper la asimetría entre clases y aspectos, pasando los aspectos a ser entidades de primera clase dentro del lenguaje.

A continuación se revisó AspectS, una implementación de aspectos en Smalltalk, donde fue destacado el uso de *Method Wrappers* para realizar el *weaving* de los aspectos en tiempo de ejecución, y el uso de las capacidades reflexivas de Smalltalk, permitiendo activar y desactivar los aspectos en forma dinámica.

El ultimo lenguaje presentado en esta sección fue AspectScheme, una implementación de aspectos en el lenguaje funcional Scheme, destacando que todos los elementos (aspectos, advice, pointcuts y join points) son entidades de primera clase en el lenguaje, entregando una gran flexibilidad.

En esta sección también se describió Dynamic AspectJ, una extensión para AspectJ que permite controlar en forma dinámica el orden de ejecución de los aspectos.

Finalmente se presentó el concepto de membranas computacionales, que corresponden a una forma de control sobre el alcance de los aspectos en el sistema, envolviendo partes de la computación de un programa, y de esta forma, controlando la visibilidad que ésta tiene.

3. PHANtom - Descripcion del lenguaje

En esta sección se presentará el lenguaje de aspectos desarrollado en este trabajo. En 3.1 se presentarán las características de los pointcuts en el lenguaje desarrollado. En 3.2 los advice, y la forma de usarlos con los elementos del contexto. En 3.3 la forma que tiene el lenguaje desarrollado para agregar nuevas funcionalidades a clases del sistema. En 3.4 los aspectos en el lenguaje, como se componen, y la forma de instalarlos en el sistema. En 3.5 el sistema de reglas de precedencia existente en el lenguaje. En 3.6 la forma que tiene el lenguaje de realizar modificaciones dinámicas a la ejecución de los advice. Finalmente, en 3.7 el sistema de membranas desarrollado, y su uso.

El nombre del lenguaje hace referencia a **PH**aro **A**spect la**N**guage, ya que el lenguaje fue desarrollado para *Pharo*[8], un dialecto de Smalltalk open-source, derivado de *Squeak*[10].

El modelo de join points usado en PHANtom, al ser un lenguaje desarrollado para Smalltalk, se basa en la recepción de un mensaje por un objeto determinado, mismo modelo usado en AspectS, el lenguaje de aspecto para Squeak comentado en 2.3.

3.1. Pointcuts

En PHANtom, como ya se mencionó, un join point corresponde a la recepción de un mensaje por un objeto determinado. Un pointcut en PHANtom es un objeto de la clase *PhPointcut* que se encargará de especificar el conjunto de join points que el pointcut define. Para definir un pointcut, se debe dar el receptor del mensaje, y el mensaje que se desea capturar. Para especificar el receptor del mensaje, se le envía el mensaje *receivers:* a una instancia de la clase *PhPointcut*, tomando como argumento el nombre del receptor del mensaje. Para especificar el mensaje que se desea capturar, se envía el mensaje *selectors:* , tomando como argumento el nombre del mensaje (selector).

En el siguiente ejemplo, creamos una instancia de la clase *PhPointcut*, y especificamos que queremos capturar todos los llamados a "position:" en la clase "Morph":

```
(PhPointcut new)
        receivers: 'Morph';
        selectors: 'position:'.
```

Tomando como inspiración a AspectJ, en PHANtom es posible usar patrones para especificar los join points que se quieren capturar. Para el receptor del mensaje, se pueden usar los comodines "*****" y "+". El comodín * puede ser usando para describir cualquier secuencia de caracteres válidas para una clase en Pharo, y + sirve para especificar que se quieren capturar todas las clases que defina el patrón anterior, junto a las sub clases de éstas.

Por ejemplo, si se quiere capturar todos los llamados a "position:", en las instancias de clases que contengan "Morph" en su nombre, junto a sus sub clases, se puede definir el

siguiente pointctut:

```
(PhPointcut new)
        receivers: '*Morph*+';
        selectors: 'position:'.
```

Para el caso de los selectores, se puede usar el comodín "_" para especificar cualquier selector y "**_:**" para cualquier selector que reciba argumentos. Por ejemplo si se quieren capturar los llamados a métodos que reciban tres argumentos, en las instancias de la clase "Morph", se puede definir el siguiente pointcut:

```
(PhPointcut new)
        receivers: 'Morph';
        selectors: '_:_:_:'.
```

Tanto para receptores, como para selectores, es posible entregar como parámetro un arreglo de patrones que los especifiquen. Por ejemplo, si se quieren capturar todos los llamados a los métodos "extent:" y "position:", en "ImageMorph" y todas sus subclases, y en "ImagePreviewMorph", se puede definir el siguiente pointcut:

```
(PhPointcut new)
        receivers: #('ImageMorph+' 'ImagePreviewMorph');
        selectors: #('extent:' 'position:').
```

Si se envía el mensaje *selectors:* con un arreglo vacío como argumento (*#()*), esto representa a todos los mensajes. Por ejemplo, un llamado a cualquier método de la clase "Morph":

```
(PhPointcut new)
        receivers: 'Morph';
        selectors: #().
```

Similar al designador *within* de AspectJ, en PHANtom se le puede enviar el mensaje *localSelectors:* a una instancia de la clase *PhPointcut*, para especificar que los selectores deben estar definidos dentro de la clase que recibe el mensaje. Los argumentos que recibe *localSelectors:* son de la misma forma que los ya descritos para *selectors:*.

Composición de Pointcuts

Es posible componer pointcuts usando operadores booleanos. Los operadores permitidos para componer pointcuts son: **or** (*por:*), **and** (*pand:*) y **not** (not). En el siguiente ejemplo, se quiere capturar todas las llamadas al método "position:" en Morph y sus sub clases, en las cuales "position:" no esté definido localmente. Para esto tenemos *p1* que captura a todas las llamadas a "position:" en "Morph" y sus sub clases, y *p2* que captura todas las llamadas a "position:" en "Morph" y sus sub clases que definan "position:" localmente. Tomando estos dos pointcuts, definimos *p3*, que va a capturar lo que queremos.

```
p1 := (PhPointcut new)
        receivers: 'Morph+';
        selectors: 'position:'.
p2 := (PhPointcut new)
        receivers: 'Morph+';
        localSelectors: 'position:'.
p3 := (p1 pand: (p2 not)).
```

3.1.1. Custom Parsers

Junto a la especificación de receptores y selectores usando patrones, PHANtom permite usar parser definidos por el usuario para especificar los pointcuts. Para esto se usa PetitParser[12] para definir un parser, el cual será luego usado en la definición del pointcut. PetitParser es un framwork para la creación de parser en Smalltalk.

Para el caso de los receptores de un mensaje, el parser intentará hacer un match con la definición de la clase de Smalltalk para definir si corresponde o no al join point que se está especificando. En el caso de los selectores, el parser realizará la correspondencia con el nombre del selector como string.

```
Morph subclass: #ImageMorph
        uses: TAbleToRotate
        instanceVariableNames: 'image'
        classVariableNames: 'DefaultForm'
        poolDictionaries: ''
        category: 'Morphic-Basic'
```

Figura 8: Definición de una clase en Smalltalk

En la figura 8 se puede ver un ejemplo de la definición de una clase en Pharo Smalltalk. Por ejemplo, el siguiente parser define una clase que tiene una sola variable de instancia en su definición, llamada "image". En este ejemplo generamos primero un parser que acepta cualquier carácter. Este parser se adjunta a un segundo parser que acepta el string "instanceVariableNames: 'image"'. Estos dos parser se adjuntan usando el mensaje *plusGreedy:* que señala que el primer parser va a seguir aceptando caracteres sólo hasta que el segundo parser acepte la cadena que sigue a continuación. El ejemplo completo: `(#any asParser plusGreedy: 'instanceVariableNames: ''image''' asParser)`. Luego este parser puede ser usado en la definición de un pointcut. En el siguiente ejemplo, utilizamos el parser descrito anteriormente, junto a un parser que va a aceptar cualquier selector.

```
(PhPointcut new)
  receivers: (#any asParser
    plusGreedy: 'instanceVariableNames: ''image''' asParser);
  selectors: #any asParser.
```

3.1.2. Exposición del contexto

Es posible exponer ciertos elementos del contexto del join point capturado, para que luego pueda ser usado por los advice que usen el pointcut definido. En PHANtom se permite definir los siguientes elementos para luego ser expuestos en un advice:

- #**receiver** permite exponer el objeto receptor del mensaje capturado.
- #**sender** permite exponer al objeto que envío el mensaje que se está capturando.
- #**selector** permite exponer el selector que representa el mensaje que se está capturando.
- #**arguments** permite exponer los argumentos usados.
- #**proceed** permite exponer el método original que está siendo capturado en un advice de tipo around.
- #**advice** permite exponer a los advice que van a ser ejecutado en el join point actual.

Para definir en un pointcut que se quieren usar estos elementos, se le debe enviar el mensaje *context:* al objeto de la clase *PhPointcut* que se usará para definir el pointcut. Como argumento de este mensaje se debe usar una lista, cuyos elementos son los símbolos listados anteriormente. Por ejemplo `context: #(#receiver #selector #arguments)` expondrá al receptor del mensaje, el selector que representa el mensaje del join point capturado, y los argumentos usados.

3.1.3. Restricción por Package

En Pharo Smalltalk las clases y métodos son organizados usando Packages, los cuales pueden ser versionados usando Monticello, que es la herramienta de control de versiones usada en Pharo Smalltalk. PHANtom permite restringir la definición de un pointcut sólo a las clases que pertenecen a un determinado Package. Esto puede resultar útil, tanto por motivos de eficiencia (no se examinan todas las clases en el sistema, sino sólo aquellas que pertenecen a un Package determinado), como para restringir el alcance que tiene la definición de un pointcut. Para esto enviamos el mensaje *restrictToPackages:* a la instancia de *PhPointcut*

que define al pointcut, usando como argumento una lista con los nombres de los Packages a los que se debe restringir la definición.

Por ejemplo, capturar todas las llamadas al método "position:" en la clase "Morph" y sus sub clases, que pertenezcan al Package "Polymorph-Widgets":

```
(PhPointcut new)
        receivers: 'Morph+';
        selectors: 'position:';
        restrictToPackages: #('Polymorph-Widgets')
```

3.2. Advice

Un advice en PHANtom es una instancia de la clase *PhAdvice*. Se debe definir el pointcut que usará el advice, cuál es la acción que este ejecutara, y cuál es el tipo de advice. Para definir el pointcut a usar, se le envía el mensaje *pointcut:* a una instancia de la clase *PhAdvice*, entregando como argumento un objeto de la clase *PhPointcut*. Si el pointcut utilizado especifica la exposición de elementos del contexto, el bloque o el mensaje que la acción defina, deben recibir un parámetro, que corresponderá a un objeto de tipo *PhContext*.

3.2.1. PhContext

Una instancia de *PhContext* representará el contexto expuesto por el join point. Este objeto contiene los elementos que fueron especificados en el join point usando el mensaje *context:* tal como fue descrito anteriormente. Una instancia de *PhContext* responde a los siguientes mensajes:

- **receiver** retorna el objeto receptor del mensaje capturado en el join point.
- **sender** retorna el objeto que envió el mensaje que desencadenó el join point capturado.
- **selector** retorna el símbolo que representa al mensaje capturado en el join point actual.
- **arguments** retorna una lista con los argumentos usando en el mensaje capturado en el join point.
- **proceed** retorna el resultado de continuar con la ejecución del join point actual.
- **proceed:** permite cambiar los argumentos entregados al join point actual, retornando el resultado.
- **beforeAdvice** retorna la lista de los advice que serán ejecutados antes del código del join point capturado.

- **afterAdvice** retorna la lista de los advice que serán ejecutados después del código del join point capturado.
- **aroundAdvice** retorna la lista de los advice que serán ejecutados en lugar del código del join point capturado.
- **currentAdvice** retorna la lista de los advice que serán ejecutados en el grupo al cual el advice que lo envía pertenece (after, before o around).

3.2.2. Acción y tipos de advice

En PHANtom se presentan dos formas en las que se puede definir la acción que será ejecutada en el advice, enviando uno de los siguientes mensaje a una instancia de *PhAdvice*.

- **advice:** este mensaje recibe como argumento un bloque, el cual sera ejecutado como acción del advice.
- **send:to:** este mensaje recibe dos argumentos, el primero corresponde al objeto al cual se le enviará un mensaje cuando se ejecute el advice, el segundo argumento es el selector del mensaje que será enviado.

Finalmente, se debe especificar el tipo de advice que se está generando, para ello se le debe enviar el mensaje *type:* a la instancia de *PhAdvice*, usando como argumento uno de los siguientes símbolos:

- #**before** se usa para especificar que el advice será ejecutado antes del join point capturado.
- #**after** se usa para especificar que el advice será ejecutado después del join point capturado.
- #**around** se usa para especificar que el advice será ejecutando reemplazando al código que se iba a ejecutar en ese join point.

3.2.3. Ejemplos de advice

A continuación, crearemos un advice que imprima el mensaje “Position about to change!”, cada vez que “SystemWindow” o alguna de sus sub clases recibe el mensaje “position:”, antes que la ventana se mueva. Para esto crearemos un pointcut que capture el llamado a “position:” en “SystemWindow” y todas sus sub clases. Para el comportamiento, usaremos el mensaje *advice:* usando como argumento un bloque que imprima el mensaje deseado al ser evaluado. Finalmente, usando *type:* definiremos que el advice debe ser ejecutado antes de la llamada al método “position:”.

```
pc := (PhPointcut new)
        receivers: 'SystemWindow+';
        selectors: 'position:'.
(PhAdvice new)
        pointcut: pc;
        advice: [Transcript show: 'Position about to change!'; cr
           ];
        type: #before.
```

Ahora ampliaremos el ejemplo anterior para que el mensaje impreso incluya al objeto que recibe el mensaje, quién lo envía, y la posición a la cual se va a mover. Para esto, tal como se mencionó anteriormente, definimos en el pointcut los elementos que se quieren exponer, usando el mensaje *context:*. En el advice, el bloque especificado debe recibir un argumento, el que corresponderá al objeto context, que finalmente usamos para componer el mensaje deseado.

```
pc := (PhPointcut new)
        receivers: 'SystemWindow+';
        selectors: 'position:';
        context: #(#receiver #sender #arguments).
(PhAdvice new)
        pointcut: pc;
        advice: [:context |
                Transcript
                        show: 'Position about to change!';cr;
                        show: ('sender: ', context sender asString
                           ); cr;
                        show: ('receiver: ', context receiver
                           asString);cr;
                        show: ('arguments: ', context arguments
                           asString);cr];
        type: #before.
```

Proceed

A continuación presentaremos algunos ejemplos del uso de *proceed* dentro de un advice.

En el siguiente ejemplo capturamos el mismo pointcut del ejemplo anterior. En el pointcut declaramos que se exponga el valor de los argumentos, y *proceed*. Usando esta información, comparamos la posición a la cual se esta moviendo la ventana, sólo si la posición a la que se quiere mover es menor que el límite especificado, continuamos con el movimiento, de lo

contrario, la ventana no se moverá:

```
pc := (PhPointcut new)
                receivers: 'SystemWindow+';
                selectors: 'position:';
                context: #(#proceed #arguments).
(PhAdvice new)
        pointcut: pc;
        advice: [ :context |
                | pos |
                pos := context arguments at: 1.
                pos x < 800 & (pos y < 600)
                        ifTrue: [ context proceed ] ];
        type: #around.
```

Ahora mejoramos el ejemplo anterior para que si se intenta mover la ventana, fuera del limite establecido, la ventana se mueva hasta el máximo permitido. Para esto creamos un objeto de la clase Rectangle, que representará el límite de 800x600 píxeles. Luego, enviando el mensaje *adhereTo:* al punto al que se quería mover la ventana y usando como argumento el rectángulo que representa nuestro límite, hacemos que el punto quede como máximo al borde del rectángulo límite. Finalmente, usando *proceed:* continuamos con la ejecución del método capturado, usando como nuevo argumento el punto recién generado.

```
(PhAdvice new)
  pointcut: pc;
  advice: [ :context |
    | limit |
    limit := Rectangle origin: 0 @ 0 corner: 800 @ 600.
    context proceed:
      (Array with: ((context arguments at: 1) adhereTo: limit)) ];
  type: #around.
```

Finalmente, supongamos que tenemos un objeto "positionChecker", que es el encargado de verificar la posición de la ventana usando el método "check:", de la misma forma que nuestro bloque anterior. Entonces podemos definir el siguiente advice, usando el mensaje *send:to:* para enviar el mensaje "check" al objeto "positionChecker":

```
(PhAdvice new)
  pointcut: pc;
  send: #check: to: positionChecker;
  type: #around.
```

3.3. Modificadores de Clase

En PHANtom, es posible agregar nuevo comportamiento a las distintas clases, de manera similar a las Inter-type declarations de AspectJ. Para realizar esto, se debe crear una instancia de la clase *PhClassModifier*, en la cual se definirán las nuevas características que serán agregadas a una clase. PHANtom permite agregar nuevas variables de instancias, variables de clase, métodos de instancia y métodos de clase.

PhClassModifier provee los siguientes métodos:

- **on:** permite especificar la clase que va a ser alterada.
- **addNewInstanceVar:** permite especificar el nombre de una nueva variable de instancia.
- **addNewClassVar:** permite especificar el nombre de una nueva variable de clase.
- **addNewInstanceMethod:** permite definir un nuevo método de instancia.
- **addNewClassMethod:** permite definir un nuevo método de clase.

Las modificaciones a la clase especificada serán efectuadas cuando el aspecto que contiene la instancia de *PhClassModifier* sea instalado en el sistema. Estos cambios serán eliminados cuando el aspecto sea desinstalado.

En el siguiente ejemplo, agregaremos la variable de instancia "phTestCaseCounter" a la clase "TestCase", y métodos para acceder y modificar esta variable. El método "phTestCaseCounter" retornará la variable de instancia "phTestCaseCounter" y en caso de que no haya sido inicializada, la inicializará con el valor 0. El método "phTestCaseCounter:" permite asignar el valor especificado como argumento, a la variable "phTestCaseCounter".

```
(PhClassModifier new)
        on: TestCase;
        addNewInstanceVar: 'phTestCaseCounter';
        addNewInstanceMethod: 'phTestCaseCounter
                ^phTestCaseCounter ifNil: [phTestCounter := 0]';
        addNewInstanceMethod: 'phTestCaseCounter: anInteger
                phTestCaseCounter := anInteger'.
```

Usando las modificaciones anteriores, se podría crear el siguiente advice que cuente cuantas veces se ha ejecutado el método "assert:" en una instancia de TestCase o cualquiera de sus sub clases, y lo imprima:

```
pc := (PhPointcut new)
  receivers: 'TestCase+';
```

```
  selectors: 'assert:';
  context: #(#receiver).

(PhAdvice new)
  pointcut: pc;
  advice: [ :context |
    context receiver phTestCaseCounter: context receiver
       phTestCaseCounter + 1.
    Transcript
      show: 'ran: ', context receiver phTestCaseCounter asString
         , ' cases ';
      cr ];
  type: #after.
```

3.4. Aspectos

Un aspecto en PHANtom es una instancia de la clase *PhAspect*. Un aspecto en PHANtom puede contener instancias de *PhAdvice* y de *PhClassModifier*.

PhAspect acepta la siguiente lista de mensajes:

- **add:** permite agregar un nuevo advice, recibe como argumento la instancia de *PhAdvice* que se quiere agregar.
- **remove:** permite quitar un advice de un aspecto. Recibe como argumento la instancia de *PhAdvice* que se quiere eliminar.
- **addClassModifier:** permite agregar una instancia de *PhClassModifier* para que se generen las modificaciones definidas por ésta, cuando se instale el aspecto. Recibe como argumento la instancia de *PhClassModifier* que se quiere agregar.
- **removeClassModifier:** permite eliminar una instancia de *PhClassModifier* contenida en el aspecto. Recibe como argumento la instancia de *PhClassModifier* que se quiere eliminar.
- **install** instala el aspecto en el sistema.
- **uninstall** desinstala el aspecto del sistema, revirtiendo todos los cambios que el aspecto haya realizado.

Si se crea una nueva clase o método en el sistema, que es capturado por algún pointcut de un aspecto activo en el sistema, estos nuevos join points también serán capturados.

A continuación se presenta un ejemplo de creación de un aspecto, usando el mismo pointcut, advice y modificador de clase, descrito en la sección 3.3.

```
mod := (PhClassModifier new)
        on: TestCase;
        addNewInstanceVar: 'phTestCaseCounter';
        ...

pc := (PhPointcut new)
        receivers: 'TestCase+';
        ...

adv := (PhAdvice new)
        pointcut: pc;
        ...

aspect := (PhAspect new)
        add: adv;
        addClassModifier: mod.
aspect install.
```

Es posible crear una sub clase de *PhAspect*, de esta forma declarar los advice y modificadores de clase que el aspecto va a contener dentro del método de inicialización. Además las acciones que los advice van a realizar pueden ser métodos del mismo aspecto, para lo cual se usa el método *send:to:* como se mencionó en la sección 3.2.2.

A continuación se presenta un ejemplo de lo anterior. Se tiene el aspecto "TestCaseCounter" sub clase de PhAspect. El método "initialize" de "TestCaseCounter" crea una instancia de *PhClassModifier* con las modificaciones descritas anteriormente, un pointcut y un advice, y los agrega a si mismo. El advice creado hace un llamado al método "countAndPrint:" en la instancia del aspecto que se crea. El método "countAndPrint:" realiza la misma acción que antes realizaba un bloque:

```
PhAspect subclass: #TestCaseCounterToSelf
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Phantom-Examples'.

TestCaseCounter>>initialize
        | adv pc mod |
```

```
        super initialize.
        mod := (PhClassModifier new)
                on: TestCase;
                addNewInstanceVar: 'phTestCaseCounter';
                ...

        pc := (PhPointcut new)
                receivers: 'TestCase+';
                ...

        adv := (PhAdvice new)
                pointcut: pc;
                send: #countAndPrint: to: self;
                type: #after.

        self addClassModifier: mod.
        self add: adv

TestCaseCounter>>countAndPrint: context
  context receiver
    phTestCaseCounter: context receiver phTestCaseCounter + 1.
  Transcript
    show: 'ran: ', context receiver phTestCaseCounter asString ,
       ' cases ';
    cr.
```

3.5. Reglas de precedencia

PHANtom permite especificar reglar de precedencia para controlar el orden de ejecución de los aspectos, de manera similar a lo que realiza AspectJ. Para establecer estas reglas, se le debe enviar el mensaje *precedence:* a la instancia de *PhAspect* en la cual se quiere especificar la precedencia. El argumento del mensaje debe ser una lista de patrones, igual a los usados para especificar los receptores de un mensaje en un pointcut, descrito en la sección 3.1.

El orden de los patrones en la lista especifica la precedencia, con los aspectos de mayor precedencia a la izquierda de la lista y los de menor precedencia a la derecha. Los aspectos con mayor precedencia, ejecutarán sus advice de tipo before y around antes que los aspectos con menor precedencia, y los advice de tipo after después de aquellos con menor precedencia.

Por ejemplo si se quiere especificar que todos los aspectos que contengan "Security" de-

ben tener precedencia sobre los de la clase "Logging" y sus sub clases, y estos a su vez, deben tener precedencia sobre el resto, se debe enviar el siguiente mensaje: `precedence: #('*Security*' 'Logging+' '*')`.

Al igual que en AspectJ, se supone que no pueden ocurrir ciclos en la precedencia especificada, por ejemplo `precedence: #('A' 'B' 'A')` generará un error. No obstante, si dos aspectos activos en el sistema tienen reglas de precedencia que están en conflicto, se sigue la misma regla que en AspectJ, se permite siempre y cuando no tengan advice que compartan un join point.

La precedencia establecida en los aspectos genera un orden global en la ejecución de los aspectos en el sistema, sin embargo, en PHANtom es posible especificar reglas de precedencia en un pointcut. Para esto se le debe enviar un mensaje a una instancia de *PhPointcut* que especifique las reglas de precedencia para ese pointcut. Una instancia de *PhPointcut* responderá a los siguientes mensajes:

- **precedence:** este mensaje establecerá reglas de precedencia adicionales para los advice que se ejecuten en los join points especificados. Estas reglas de precedencia tendrán mayor prioridad a las establecidas en un aspecto, esto quiere decir que, en caso de conflicto, se respeta el orden establecido en el pointcut por sobre el orden establecido en los aspectos.

- **overridePrecedence:** este mensaje establecerá un nuevo conjunto de reglas de precedencia para los advice que se ejecuten en los join points especificados. Estas reglas reemplazarán a aquellas establecidas en los aspectos, por tanto, para los join points especificados en el pointcut, el orden global no será tomado en cuenta.

A continuación presentaremos algunos ejemplos de lo anterior.

En el siguiente ejemplo se tiene el pointcut *pc1* que captura los llamados al método "extent:" en la clase "Morph" y todas sus sub clases. El pointcut *pc2* captura las llamadas al método "extent:" en la clase "ImageMorph", que es una sub clase de "Morph". El aspecto *aspectC* establece una regla de precedencia que dice que los aspectos de la clase AspectA tienen mayor precedencia que los de la clase AspectB, y estos, a su vez, mayor precedencia que los de la clase AspectC. El pointcut *pc2* establece una regla de precedencia que dice que los aspectos de la clase "AspectB" tienen mayor precedencia que aquellos de la clase "AspectA". En la figura 9 se pueden ver los grafos que representan cada regla, junto al grafo de precedencia global resultante, y el grafo de precedencia para los join points capturados por el pointcut *pc2*. Dado lo anterior, para los advice de tipo around, si se hace un llamado al método "extent:" en una instancia de la clase "Morph", el orden de ejecución de los advice por aspecto sera: AspectA->AspectB->AspectC, pero si se hace un llamado al método "extent:" en una instancia de la clase "ImageMorph", el orden de ejecución de los advice por aspecto

será: AspectB->AspectA->AspectC. Esto porque el orden especificado en el pointcut *pc2* tiene una prioridad mayor al orden global.

```
pc1 := (PhPointcut new)
  receivers: 'Morph+';
  selectors: 'extent:';

pc2 := (PhPointcut new)
  receivers: 'ImageMorph';
  selectors: 'extent:';
  precedence: #('AspectB' 'AspectA').

aspectA := (AspectA new)
  add: ( (PhAdvice new)
    pointcut: pc2;
    advice: ...;
    type: #around)
  ...

aspectB := (AspectB new)
  add: ( (PhAdvice new)
    pointcut: pc1;
        advice: ...;
    type: #around)
  ...

aspectC := (AspectC new)
  add: ( (PhAdvice new)
    pointcut: pc1;
    advice: ...;
    type: #around);
  ...;
  precedence: #('AspectA' 'AspectB' 'AspectC').
```

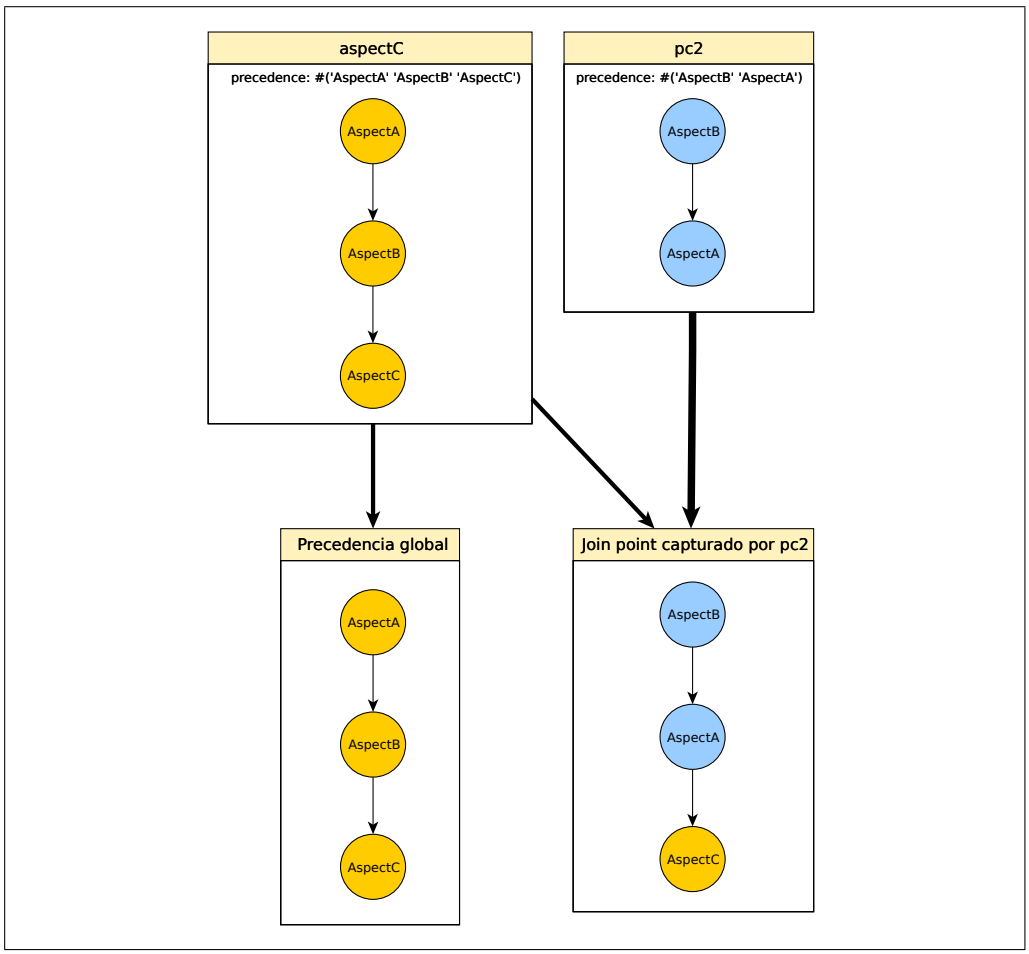

Figura 9: Grafo de precedencia de aspectos

3.6. Modificación dinámica de advice

Siguiendo una idea similar a la propuesta por Assaf y Noyé [2], presentada en la sección 2.5, en PHANtom es posible, desde un advice, controlar la ejecución de otros advice en el join point capturado. Para esto, se debe declarar en el pointcut que se quieren exponer los advice, tal como se mencionó en la sección 3.1.2. A diferencia de lo que ocurre en Dynamic AspectJ, en PHANtom se pueden cambiar advice específicos, y no todo el grupo de advice que pertenecen a un aspecto particular.

Como se mencionó en 3.2.1, para acceder a los advice se debe usar una instancia de la clase *PhContext*. *PhContext* provee 4 mensajes para acceder a los grupos de advice (*beforeAdvice, aroundAdvice y currentAdvice*), junto a lo anterior, provee los siguientes métodos para alterar estas listas de advice:

- **beforeAdvice:** recibe como argumento una nueva lista de advice, que reemplazará a la existente para los advice de tipo *before*.
- **aroundAdvice:** recibe como argumento una nueva lista de advice, que reemplazará a

la existente para los advice de tipo *around*.

- **afterAdvice:** recibe como argumento una nueva lista de advice, que reemplazará a la existente para los advice de tipo *after*.
- **currentAdvice:continueAt:** recibe como argumentos una nueva lista de advice, que reemplazará a los que se están ejecutando actualmente, y un índice para indicar en que lugar de la nueva lista se debe continuar la ejecución. Si se utiliza alguno de los mensajes anteriores para cambiar la lista de advice a la cual el mismo advice pertenece, ese cambio es ignorado.
- **currentAdviceIndex** retorna el índice del advice que se está ejecutando actualmente, en la lista de *currentAdvice*.

A continuación presentaremos un ejemplo de modificación dinámica de advice. En el ejemplo dado para Dynamic AspectJ en 2.5.1, se tenía una aplicación cliente-servidor, donde los clientes subían archivos al servidor, y el servidor buscaba virus al recibir los archivos. Además se tenían dos aspectos actuando en el sistema, "CompressUpload" que comprimía los archivos antes de enviarlos, y los descomprimía antes de ser revisados por virus, y "SecureUpload", que encriptaba los archivos antes de subirlos, y los desencriptaba antes de revisar por virus. Ahora crearemos un aspecto que controle el orden de ejecución de estos aspectos, de manera similar a lo presentado para Dynamic AspectJ.

En este ejemplo, creamos un pointcut que captura los llamados a *send:* en la clase "Client", y expone los advice que se ejecutarán en ese pointcut. Luego, usando este pointcut, creamos un advice que tomará la lista de around advice, y seleccionará todos aquellos que no pertenecen al aspecto "CompressUpload", después seleccionará todos aquellos que pertenecen al aspecto "CompressUpload". Finalmente, reemplazará la lista de around advice, componiendo ambas lista anteriores, colocando los advice que pertenecen a "CompressUpload" al principio. De forma similar, se creará un advice que haga los mismo para el join point que captura los llamados a *virusCheck:* en la clase "Server", pero colocando los advice que pertenecen a "SecureUpload" al inicio de la lista.

```
sendPc := (PhPointcut new)
  receivers: 'Client';
  selectors: 'send:';
  context: #(#advice).

sendAdvice := (PhAdivce new)
  pointcut: sendPc;
  advice: [:context |
    | adviceList compressUploadAdvice |
```

```
    adviceList := (context aroundAdvice
      select: [:adv | adv owner ~= CompressUpload])
        asOrderedCollection.
    compressUploadAdvice := context aroundAdivce
      select: [:adv | adv owner = CompressUpload].
    context aroundAdvice:
      (adviceList addAllFirst: compressUploadAdvice) asArray];
  type: #before.

virusCheckPc := (PhPointcut new)
  receivers: 'Server';
  selectors: 'virusCheck:';
  context: #(#advice).

virusCheckAdvice := (PhAdivce new)
  pointcut: virusCheckPc;
  advice: [:context |
    | adviceList secureUploadAdvice |
    adviceList := (context aroundAdvice
      select: [:adv | adv owner ~= SecureUpload])
        asOrderedCollection.
    compressUploadAdvice := context aroundAdivce
      select: [:adv | adv owner = SecureUpload].
    context aroundAdvice:
      (adviceList addAllFirst: secureUploadAdvice) asArray];
  type: #before.
```

3.7. Uso de membranas

Como se explicó en la sección 2.6, las membranas sirven para controlar el alcance que tienen los aspectos en el sistema, permitiendo envolver partes de la computación de un programa dentro de ellas. PHANtom hace uso de las membranas para controlar los aspectos que actúan en el sistema. Todos los aspectos que se instalen en el sistema estarán dentro de una membrana, que es generada en forma automática. Lo anterior permite controlar los problemas de reentrancia que ocurren, por ejemplo, cuando un aspecto ejecuta dentro de un advice, código que es capturado por el mismo aspecto.

En el siguiente ejemplo se tiene un aspecto con un advice cuyo pointcut captura los llamados al método "show:" en una instancia de la clase "TranscriptModel". Cuando el llamado

es capturado, el advice imprime el mensaje "Showing: ", enviando el mensaje "show:" al objeto Transcript, el cual es una instancia de TranscriptModel. Cuando se instala este aspecto, y se ejecuta el código `Transcript show: 'Reentrancy'`, el join point es capturado, y el advice se ejecuta, esto podría causar un problema de reentrancia, ya que al imprimir el mensaje del advice, el join point sería capturado nuevamente. Sin embargo, en PHANtom, el aspecto es ejecutado en su propia membrana, por tanto, el join point generado al ejecutar el advice, no es visto por el aspecto.

```
asp := PhAspect new
  add:
    ((PhAdvice new)
      pointcut:
        ((PhPointcut new)
          receivers: 'TranscriptModel';
          selectors: 'show:');
      advice: [ Transcript show: 'Showing: ' ];
      type: #before).
asp install.
Transcript show: 'Reentrancy'; cr.
```

3.7.1. Membranas sobre flujos de ejecución

Además de lo anterior, en PHANtom es posible crear estructuras de membranas propias. Una membrana en PHANtom es una instancia de la clase *PhMembrane*. Una instancia de *PhMembrane* responde a los siguientes mensajes:

- **pointcut:** recibe como argumento una instancia de *PhPointcut* y establece que la membrana envolverá la computación que se genere al ocurrir alguno de los join points establecidos en el pointcut, los que podrán ser capturados por las membranas que estén observándola.
- **advise:** toma como argumento una instancia de *PhMembrane* y establece que la membrana podrá ver los join points que provengan de la membrana que se reciba como argumento.
- **unAdvise:** recibe como argumento una instancia de *PhMembrane*, y establece que la membrana dejará de ver los join points que provengan de la membrana que se recibe como argumento.
- **install** instala la membrana en el sistema.

- **uninstall** desinstala la membrana del sistema.

Un aspecto puede ser registrado en una membrana enviándole el mensaje *registerOn:* usando como argumento la instancia de *PhMembrane* en la cual se quiere registrar el aspecto.

A continuación se presentará el ejemplo de la figura 7 c, adaptado del ejemplo presentado en "*Exploring Membranes for Controlling Aspects*"[3]. En este ejemplo se tiene a la membrana "l0" envolviendo la computación que se genera al ocurrir un llamado a "browse" en una instancia de la clase "Browser". La membrana "l1" y "l2" verán los join points que se generen dentro de "l0", además la membrana "l2" verá los join points que se generen dentro de "l1". El aspecto "cache", que captura los llamados al método "getUrl:" en instancias de la clase "Browser" y sus sub clases, estará registrado en "l1", capturando los join points provenientes de "l0". El aspecto "quota", que captura las llamadas al método "accessDisk" en instancias de la clase "DiskAccessor", estará registrado en "l2", capturando los join points que provengan tanto de "l0" como de "l1".

```
l0 := PhMembrane new pointcut: (
                PhPointcut new
                        receivers: 'Browser+';
                        selectors: 'browse').
l1 := PhMembrane new advise: l0.
l2 := PhMembrane new advise: l0.
l2 advise: l1.
cache := PhAspect new
        add: (PhAdvice new
                pointcut: (PhPointcut new
                        receivers: 'Browser+';
                        selectors: 'getUrl:');
                advice: ...;
                type: #before).
quota := PhAspect new
        add: (PhAdvice new
                pointcut: (PhPointcut new
                        receivers: 'DiskAccessor';
                        selectors: 'accessDisk');
                advice: ...;
                type: #before).
cache registerOn: l1.
quota registerOn: l2.
l0 install.
```

```
Browser new browse
```

3.7.2. Membranas sobre objetos

También es posible desplegar una membrana sobre algún objeto particular, de esta forma, todos los join points que se generen en la computación de ese objeto estarán dentro de la membrana. De esta forma, los join points que sean generados dentro de esa membrana podrán ser vistos por las membranas que la estén monitorizando. Para realizar esto, se debe enviar el mensaje *deployOn:* a una instancia de PhMembrane, este mensaje recibe como argumento el objeto sobre el cual se quiere desplegar la membrana.

A continuación presentamos un ejemplo de lo anterior. En éste ejemplo tenemos una estructura similar a la del ejemplo en 3.7.1, con dos diferencias. La primera es que a la membrana "l0" no le enviamos el mensaje *pointcut:*, ya que esta membrana será desplegada sobre un objeto especifico. La segunda diferencia es que para desplegar la membrana sobre el objeto especifico, se le envía el mensaje *deployOn:* que recibe como argumento "browser" que es la instancia de la clase "Browser" sobre la cual se quiere desplegar la membrana. Con esto se logra que esta estructura de membranas solamente este activa para el objeto "browser", pero no para nuevas instancias de "Browser" a también se despliegue sobre estas nuevas instancias.

```
l0 := PhMembrane new.
l1 := PhMembrane new advise: l0.
l2 := PhMembrane new advise: l0.
l2 advise: l1.
cache := PhAspect new
        add: (PhAdvice new
                pointcut: (PhPointcut new
                        receivers: 'Browser+';
                        selectors: 'getUrl:');
                advice: [Transcript show: 'Cache';cr.
                                DiskAccessor new accessDisk];
                type: #before).
quota := PhAspect new
        add: (PhAdvice new
                pointcut: (PhPointcut new
                        receivers: 'DiskAccessor';
                        selectors: 'accessDisk');
                advice: [Transcript show: 'Quota';cr];
                type: #before).
cache registerOn: l1.
```

```
quota registerOn: 12.

browser := Browser new.
10 deployOn: browser.
browser browse.
```

3.8. Resumen

En esta sección se presentó PHANtom, el lenguaje desarrollado. En 3.1 se describió el uso de *pointcuts* en el lenguaje, y el uso de patrones para especificar los mensajes y receptores que definen a un *pointcut* en PHANtom. También, se describió como pueden componerse los *pointcuts* definidos, para generar nuevos *pointcuts*, y la posibilidad que ofrece PHANtom para crear *parsers* personalizados para describir un *pointcut*. Finalmente se describió como pueden exponerse distintos elementos del contexto en un *pointcut*, para luego ser usados por un *advice*, y como se puede restringir la definición de un *pointcut* para que solo capture las clases en un determinado *package* de Pharo.

En 3.2 se describieron los *advice* en PHANtom, los distintos tipos de *advice* que PHANtom provee, y el uso de elementos del contexto dentro de éstos. Además se mostraron algunos ejemplos de uso de *advice*.

En 3.3 se presentaron las herramientas que provee PHANtom para agregar nuevo comportamiento a las clases del sistema, siendo posible agregar nuevos métodos y variables a clases e instancias.

En 3.4 se describieron los aspectos en PHANtom, como estos se componen con los elementos anteriormente descritos, y la forma de aplicarlos en el sistema. Además se presentan ejemplos de uso de éstos.

En 3.5 se presentaron las reglas de precedencia que pueden usarse para determinar el orden en que los aspectos se ejecutarán en el sistema. Estas reglas pueden hacer uso de patrones, como los descritos en 3.1. Además pueden definirse en un aspecto, determinando un orden global, y a nivel de *pointcut*, permitiendo especificar un orden de precedencia para el *pointcut*, teniendo estas reglas a nivel de *pointcut*, mayor prioridad que aquellas a nivel global.

En 3.6 se describen las formas en que PHANtom puede modificar el orden de ejecución de los *advice* en forma dinámica, en el *join point* capturado.

En 3.7 se describe el uso de membranas computacionales en PHANtom, como éstas son usadas para controlar el alcance de los aspectos, y como pueden ser usadas por los usuarios de PHANtom para generar sus propias estructuras de membranas.

4. Implementación de PHANtom

En esta sección se presentará la implementación del lenguaje, presentaremos una visión, y luego los puntos más relevantes de la implementación.

4.1. Visión general

PHANtom se desarrolló utilizando las capacidades reflexivas del lenguaje Smalltalk, usando MethodWrappers, presentados en la sección 4.2, para capturar la recepción de los mensajes por parte de un objeto. En la figura 10 se puede ver un diagrama con las principales clases de PHANtom, y sus relaciones.

PHANtom ha sido implementado sin realizar cambios en el compilador de Pharo Smalltalk, o en la máquina virtual.

A continuación se presenta una lista con las clases más importantes de PHANtom:

- PhAspect: Ver secciones 3.4, 4.6 y 4.8.
- PhAdvice: Ver secciones 3.2 y 4.4.
- PhPointcut: Ver secciones 3.14.3
- PhReceiverParser: Ver sección 4.3
- PhSelectorParser: Ver sección 4.3.
- PhContext: Ver sección 3.2.1.
- PhClassModifier: Ver secciones 3.3 y 4.5.
- PhMembrane: Ver secciones 3.7, 4.8 y 4.9.
- PhMethodWrapper: Ver secciones 4.2 y 4.8.
- PhAdviceRunner: Ver sección 4.4.
- PhAdviceGroup: Ver sección 4.4.
- PhAspectWeaver: Ver sección 4.8.
- PhPrecedenceRulesProcessor: Ver sección 4.7.
- AspectGraph: Ver sección 4.7.

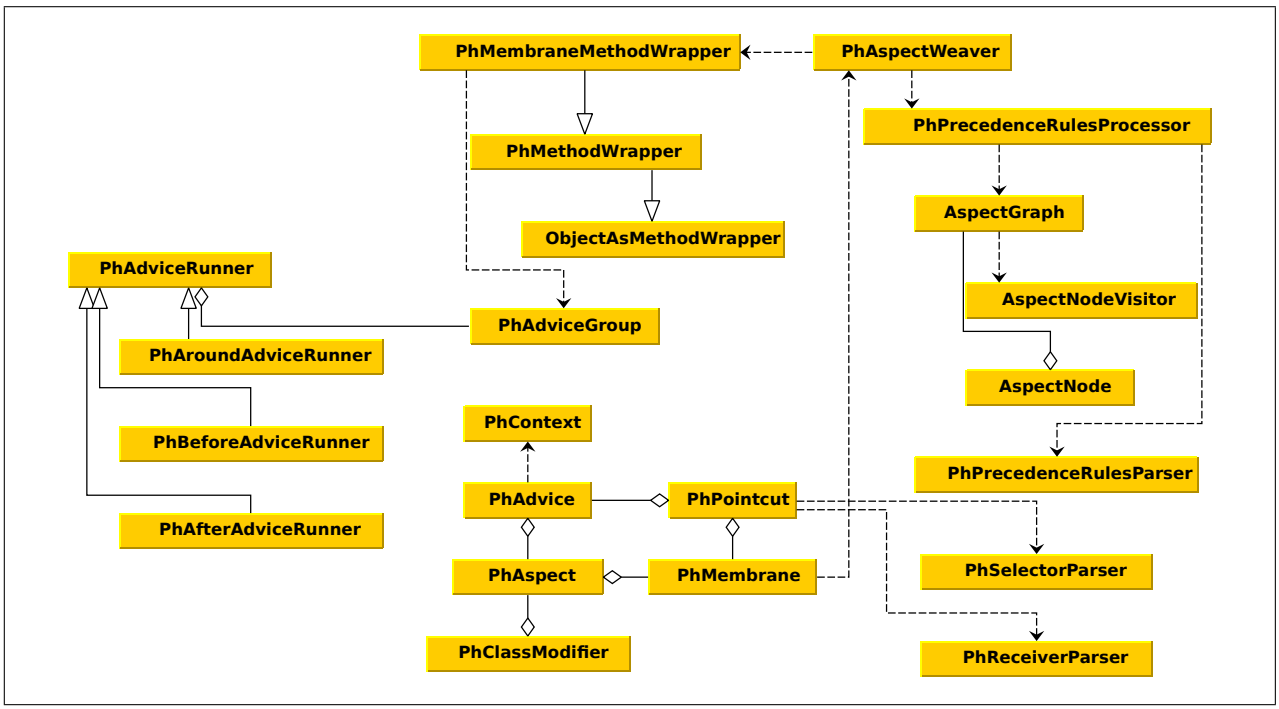

Figura 10: Diagrama UML de las clases de PHANtom

4.2. Method Wrappers y CompiledMethods

Para la captura de los join points en el sistema, se optó por el uso de Method Wrappers (MW). Una de las principales razones para el uso de MW, es que permiten realizar las modificaciones requeridas para capturar los join points, y realizar el weaving de los aspectos, sin necesidad de realizar cambios en el compilador de Pharo Smalltalk o en la maquina virtual. Esto permitió que se lograra tener un lenguaje con las funcionalidades que se requerían, en el tiempo destinado a esta memoria.

En Pharo Smalltalk, los métodos que define cada clase son compilados a objetos del tipo *CompiledMethod*, los que son interpretados por la máquina virtual. Los MW implementados en Pharo Smalltalk, reemplazan al objeto del tipo *CompiledMethod* en el diccionario de métodos de la clase, por un objeto del tipo *ObjectAsMethodWrapper*. Cuando la máquina virtual encuentra un objeto que no es un *CompiledMethod*, le envía el mensaje *run:with:in:* al objeto, ejecutándose el código que este objeto defina en ese método. En la figura 11, se puede ver un ejemplo de lo anterior.

Para la implementación de PHANtom, se modificaron los *ObjectAsMethodWrapper*, para resolver dos problemas que estos presentaban al instalarse en el sistema, lo que presentamos a continuación.

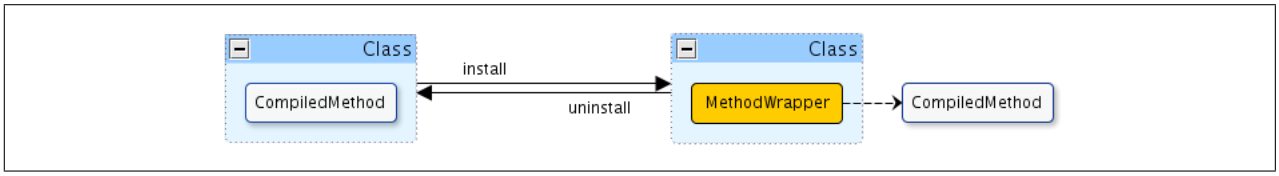

Figura 11: Instalación MethodWrapper

El primer problema presentado corresponde a la instalación de un MW en un método que sea heredado de alguna super clase, ilustrado en la figura 12a. Cuando se instala un MW en un método heredado, el MW sigue la cadena de búsqueda hasta encontrar la clase donde se implementa el método (la clase que tiene el *CompiledMethod* en el diccionario de métodos de la clase), obtiene la referencia al *CompiledMethod*, y luego se instala localmente, en el diccionario de método de la clase sobre la que se está instalando el MW. Al desinstalar el MW, éste coloca la referencia al *CompiledMethod* capturado en la instalación, en el diccionario de métodos de la clase sobre la que estaba instalado, como se puede ver en la figura 12a. Esto genera un problema, ya que una clase que antes heredaba un método desde una super clase, ahora lo implementa localmente. Para solucionar este problema, se modificó el protocolo de instalación del MW, para guardar un registro si el método sobre el cual se estaba instalando el MW estaba siendo heredado de una super clase, y de esta forma, limpiar el diccionario de métodos de ser necesario.

El segundo problema presentado corresponde a la instalación de un MW en un método heredado de una super clase que ya tiene un MW instalado en ese método, presentado en la figura 12b. Por el protocolo de instalación de los MW, el MW busca el *CompiledMethod* en la super clase, si encuentra un MW, lo elimina para encontrar el *CompiledMethod* que implementa el método, y copia la referencia como ya fue explicado, esto se puede ver en la figura 12b. El problema de esto es que elimina el MW de la super clase, y solo deja el de la sub clase. Por lo anterior, se modificó el protocolo de instalación del MW para tomar esto en consideración.

Para el caso de nuestra implementación, es necesario tener un MW en cada una de las clases sobre la cual se instala un MW, y no sólo en la super clase en casos de métodos heredados, esto debido a que era necesario capturar al objeto que recibe el mensaje. Por ejemplo, supongamos que tenemos una clase *Foo*, y *SubFoo*, que es una sub clase de *Foo*, y el metodo "bar" que esta implementado en *Foo.* Si se quiere capturar las llamadas al método "bar" en *SubFoo*, se necesita cambiar el comportamiento cuando se recibe el mensaje "bar" en *SubFoo*, pero no se debe hacer nada si se recibe el mensaje "bar" en *Foo*.

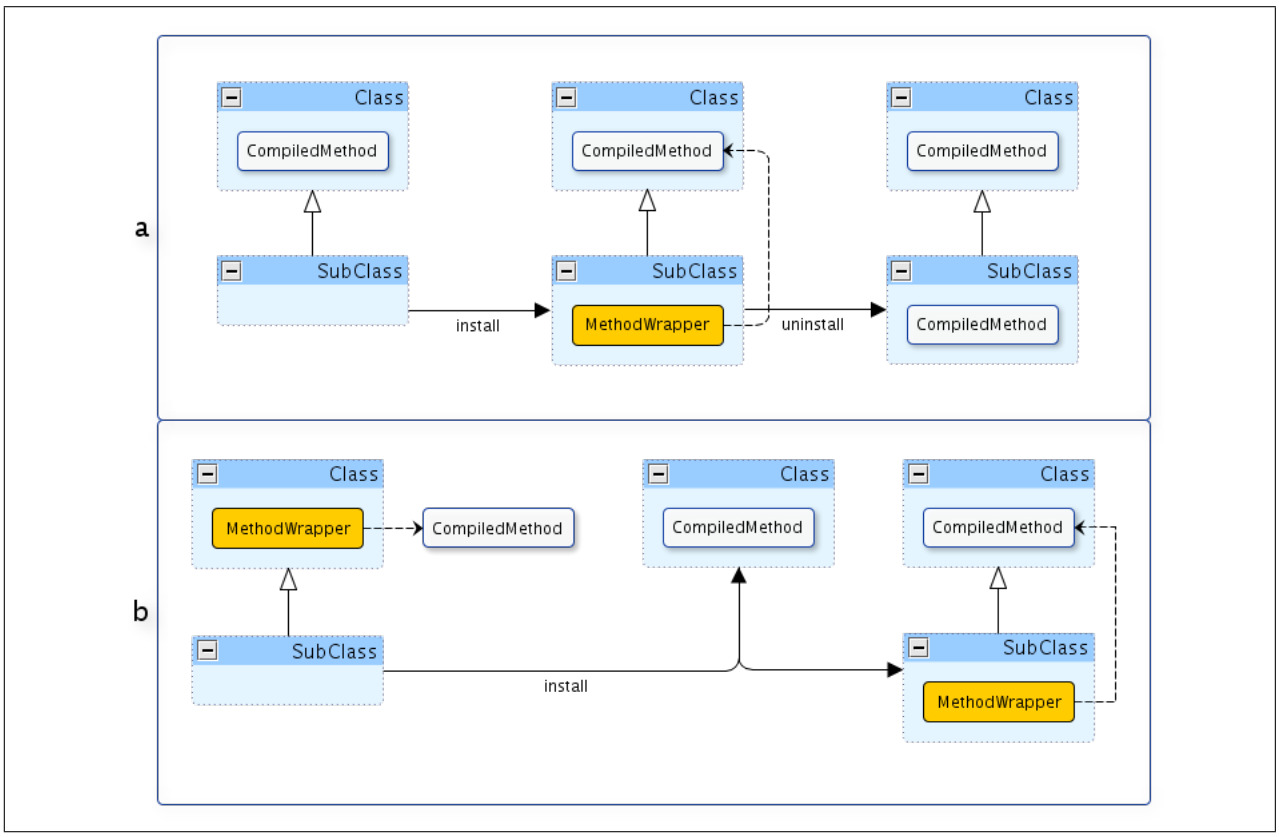

Figura 12: Problemas en la instalación de MethodWrappers

4.3. Pointcuts

Un join point, tal como se hablo en la sección 3.1, corresponde a una tupla del tipo (Objeto receptor, mensaje). La clase *PhPointcut* se encarga de generar una colección de estas tuplas, en base a los patrones dados para receptores (*receivers:*) y selectores (*selectors:*). Para esto, se recorren todas las clases del sistema y sus métodos, y se verifica si pertenecen al conjunto de join points que el pointcut define.

Para el sistema de patrones se optó por usar PetitParser[12], debido a la flexibilidad que presenta para crear nuevas gramáticas, combinando distintos tipos de parsers.

Las clases *PhReceiverParser* y *PhSelectorParser* son parser creados usando las funcionalidades de PetitParser para crear gramáticas. Para el caso de *PhReceiverParser*, este parser toma el patrón entregado en una instancia de *PhPointcut* para definir a los receptores de un mensaje, y genera un nuevo parser que es usado para filtrar las clases que se quiere capturar, usando al definición de clase como entrada. De forma similar, *PhSelectorParser* toma el patrón entregado para definir los mensajes que se quieren capturar, generando un nuevo parser que es usado para filtrar los mensajes a los que responden los receptores especificados, usando como entrada el nombre del mensaje (selector).

El uso de PetitParser para definir los patrones permite extender el sistema de patrones

especificados de forma simple, usando las funcionalidades que entrega PetitParser. Además, al usar PetitParser internamente para filtrar las clases, permite que un usuario de este sistema pueda definir sus propios parser para filtrar las clases, como se explicó en la sección 3.1.1.

4.4. Advice

PhAdvice es la clase que define un advice. Cuando se realiza el weaving de los aspectos, todos los advice que deben ser ejecutados en un join point determinado, son agrupados en la clase *PhAdviceGroup* que se encarga de separar los advice según su tipo (before, around y after), controlar la ejecución de los grupos de advice, y controlar la modificación que un advice realiza sobre los advice que van a ser ejecutados (modificación dinámica de advice).

Los grupos de advice (before, after, around), son ejecutados por una instancia de la clase *PhAdviceRunner*, que ejecuta cada uno de los advice individuales dentro de la secuencia, controlando las modificaciones que éstos hacen sobre su mismo grupo (llamados a *changeAdvice:continueAt:* sobre una instancia de la clase *PhContext*). Además, se encarga de manejar los llamados a *proceed* y *proceed:*, controlando el encadenamiento de estos llamados.

Junto a lo anterior, *PhAdviceRunner* se encarga de manejar la relación que tiene cada advice que se va a ejecutar con las membranas activas en el sistema.

4.5. Modificadores de Clase

Para las modificaciones que pueden realizar los aspectos a las clases del sistema, explicadas en la sección 3.3, se optó por usar las capacidades reflexivas del lenguaje. De esta forma la clase *PhClassModifier* se encarga de agregar nuevas variables a clases, usando las capacidades de reflexión en Pharo Smalltalk, utilizando los métodos: `addInstVarNamed: varName`, que permiten agregar nuevas variables de instancia a una clase, o `addClassVarNamed: varName`, para agregar una nueva variable de clase. Se optó por verificar previamente la existencia de estas variables en las clases especificadas, y si están previamente definidas, informar al usuario con un mensaje de error. La razón para realizar lo anterior es que, si el usuario del lenguaje esta especificando agregar una nueva variables, probablemente sea con el fin de usarla para algo específico del aspecto que contiene a este modificador de clase, y por tanto, usarla dentro de la ejecución del aspecto, entrando en conflicto con el uso que esta variable tenía dentro del comportamiento de la clase. Otra opción habría sido permitir agregar variables sólo para el aspecto específico, sin embargo, esto habría aumentado la complejidad del lenguaje, y en la etapa actual, se quería tener un lenguaje simple y funcional.

Para el caso de los métodos, se optó por una solución similar. Los nuevos métodos son compilados en la clase especificada, colocándose dentro de la categoría "*phantom-generated-method", utilizando llamados del tipo: `modifiedClass compile: aMethodString classified: '*phantom-generated-method'` para el caso de las clases o `modifiedClass`

`theMetaClass compile: aMethodString classified: '*phantom-generated-method'` para las meta clases. El string que se entrega como argumento es parseado para comprobar que sea una expresión valida del lenguaje, y obtener el selector correspondiente, ésto al momento de declarar el modificador de clase. En los casos en que el nuevo método ya existe en la clase especificada, se le informa al usuario con un mensaje de error. La racionalidad para esto es similar a la dada para las variables. Además, los nuevos métodos, al ser métodos compilados dentro del diccionario de clases, simplifica el que estos puedan ser capturados por algún pointcut de un aspecto.

4.6. Aspectos

Como se mencionó en la sección 3.7, cuando se instala un aspecto, el protocolo de instalación crea una membrana que vea el mismo pointcut en el cual está interesado el aspecto, y luego se registra en esta membrana. De esta forma es posible usar los aspectos sin necesariamente tener presente la existencia de las membranas, y al mismo tiempo, evitar los problemas de reentracia que podrían presentarse sin su uso. En las secciones 4.8 y 4.9 se habla en más profundidad del weaving de los aspectos y el uso de membranas.

4.7. Grafo de precedencias

Para la aplicación de las reglas de precedencia en los aspectos, se optó por crear grafos dirigidos acíclicos para realizar el análisis de precedencia. La clase *PhPrecedenceRulesProcessor* se encarga de tomar una regla de precedencia, procesarla, y generar un grafo dirigido acíclico que representa la regla establecida. Para el procesamiento de las reglas se usa la clase *PhPrecedenceRulesParser*, que es un parser desarrollado usando PetitParser, que toma un patrón y lo convierte en un conjunto de nodos que representan al patrón. Usando estos nodos, *PhPrecedenceRulesProcessor* crea un grafo que representa la regla que se esta procesando. Los grafos están representados por la clase *AspectGraph*, los nodos de este grafo son instancias de *AspectNode*. Además se creó la clase *AspectNodeVisitor* para recorrer el grafo. Cada grafo generado detecta si existen ciclos dentro de él, y en caso de detectar un ciclo, se le informa al usuario que existe un error con las reglas definidas. Además, los grafos pueden ser combinados, esto dado que puede existir más de una regla actuando en un join point, por lo cual se generan los grafos para cada una de las reglas que actúan en ese punto y luego se combinan para crear el grafo completo.

Una vez que el grafo ha sido generado, se obtiene el orden topológico de los aspectos usando un algoritmo de búsqueda en profundidad. En la figura 13a se puede ver un ejemplo resultante de la unión de las siguientes reglas:

- 1 con mayor precedencia que 2 y 3

- 3 con mayor precedencia que 2, 4 y 5.
- 4 con mayor precedencia que 2
- 5 con mayor precedencia que 4
- 6 con mayor precedencia que 1, 3 y 5.

Para el algoritmo de ordenamiento usado, se recorre el grafo desde los nodos con menor precedencia (aquellos que no tienen hijos) hacia arriba, como se puede ver en la figura 13b. Luego, usando el algoritmo que se presenta a continuación se obtiene el orden deseado, que se puede ver en la figura 13c.

```
NodosOrdenados := Se inicializa una lista vacía para los nodos
    ordenados.
N := Conjunto de nodos que no tienen hijos (el que necesariamente
    debe existir ya que el grafo no tiene ciclos).
Para cada nodo n en N:
        visitar(n)

visitar(Nodo n):
        si n no ha sido visitado:
                marcar n como visitado
                para cada nodo d que precede a m:
                        visitar(d)
                agregar n a la lista de nodos ordenados
```

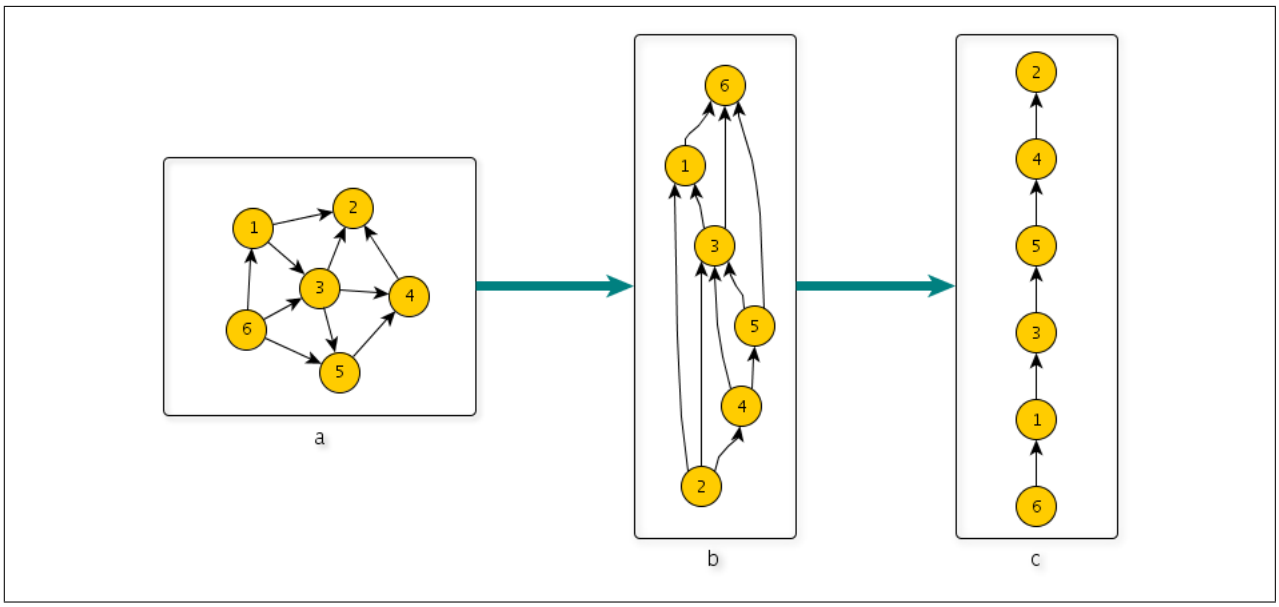

Figura 13: Orden topológico de nodos

4.8. Weaving de los aspectos/membranas

La clase *PhAspectWeaver* se encarga de manejar el weaving de los aspectos. Para esto, cada vez que se instala una nueva membrana en el sistema, se analizan todas las membranas instaladas, junto a la estructura que éstas contienen (membranas que ven a las membranas instaladas), obteniéndose la colección de todos los pointcuts que las membranas, o sus aspectos registrados, están capturando. Luego se instala una instancia de la clase *PhMembraneMethodWrapper* en cada uno de los join points definidos por los pointcuts anteriores, con el conjunto de membranas que están definidas para esos join points.

Junto a lo anterior, la clase *PhAspectWeaver* está registrada con el sistema de notificación de eventos de Pharo Smalltalk, de esta forma, captura los cambios que se producen en el sistema, actuando en consecuencia.

Los eventos capturados actualmente son:

- Clases
  - Creación de una nueva clase: se eliminan los MW instalados y se vuelven a instalar las membranas tomando en cuenta la nueva clase creada.
  - Modificación: se eliminan los MW instalados y se vuelven a instalar las membranas tomando en cuenta la nueva clase creada.
  - Renombrado: se restablecen las referencias en los MW hacia la clase que cambio de nombre (para evitar un estado inconsistente del sistema) y luego se limpia el sistema de MW y se vuelven a instalar las membranas.
  - Eliminación: se eliminan los MW instalados y se vuelven a instalar las membranas tomando en cuenta la nueva clase creada.
- Métodos
  - Creación de un nuevo método en una clase: se eliminan los MW instalados y se vuelven a instalar las membranas tomando en cuenta el nuevo método creado.
  - Modificación: al modificarse un método, el sistema elimina el antiguo CompiledMethod (CM), y coloca el nuevo CM creado al modificarse el método, en el diccionario de métodos de la clase. Si el método modificado estaba siendo capturado por PHANtom, se actualiza el MW que estaba instalado en el diccionario de método de la clase, para hacer referencia al nuevo CM, y vuelve a instalar el MW.
  - Eliminación: Si el método eliminado estaba siendo capturado por PHANtom, se elimina el MW de la lista de MW instalados.

Para el caso de las membranas que son desplegadas sobre objetos, *PhAspectWeaver* instala una instancia de *PhMembraneMethodWrapper* en cada uno de los métodos del objeto sobre el cual la membrana ha sido desplegada. Esto, ya que es necesario capturar cada una de las llamadas a métodos de este objeto, para verificar si corresponde a la instancia sobre la cual la membrana fue desplegada, activándola.

Para evitar la reificación de la pila de llamados, en el contexto donde se captura un join point, se optó por mantener una pila propia, de esta forma, cada vez que se captura un join point, se agrega el objeto receptor a la pila, y se saca cuando se sale de esa llamada. Para mantener la pila se está usando una estructura de diccionario con referencias débiles en las llaves. La llave del diccionario corresponde al objeto que representa el proceso (thread) en el cual se está realizando la llamada, y el valor guardado corresponde a la pila de objetos para ese proceso.

4.9. Estructura de membranas

Para el manejo de las membranas se usa un sistema de grafos, de esta forma, cada vez que se captura un join point, esto ocurre en el contexto de una o más membranas. Las membranas activas en el join point que se está capturando son guardadas en un diccionario con referencias débiles por proceso, tal como el mencionado en la sección 4.8, para la pila de objetos. Luego, los join point que se generen dentro de las membranas activas, serán vistos por las membranas que están visualizando a las activas en el grafo.

En el siguiente ejemplo, se tiene la clase *PhBrowser*, y las subclases de ésta, *PhAdminBrowser, PhUserBrowser y PhGuestBrowser*, como se puede ver en la figura 14. Se crea la membrana "adminUserMembrane" que captura la computación que se inicia cuando *PhAdminBrowser* o *PhUserBrowser* reciben el mensaje "get:", y la membrana "userGuestMembrane" que captura la computación que se inicia cuando *PhUserBrowser* o *PhGuestBrowser* reciben el mensaje "get:". Junto a esto se crea el aspecto "cacheAspect" y se registra en una membrana que observa a "adminUserMembrane", y el aspecto "logAspect", y se registra en una membrana que observa a "userGuestBrowser", tal como se aprecia en la figura 14. De esta forma, si se produce un llamado al método "get:" en *PhAdminBrowser*, los join points que se generen serán vistos por la membrana donde esta registrado "cacheAspect", pero no por la membrana donde esta registrado "logAspect", ya que la membrana activa es "adminUserMembrane". Sin embargo, si se produce un llamado al método "get:" en *PhUserBrowser*, los join points que se generen podrán ser vistos por la membrana donde está registrado "cacheAspect" y también por la membrana donde está registrado "logAspect", ya que tanto "adminUserMembrane" como "userGuestMembrane" estarán activas.

```
cacheAspect := PhAspect new
  add: (...).
```

```
logAspect := PhAspect new
  add: (...).

adminUserMembrane := PhMembrane new
  pointcut:  ((PhPointcut new)
    receivers: #('PhAdminBrowser' 'PhUserBrowser');
    selectors: 'get:').

userGuestMembrane := PhMembrane new
  pointcut: ((PhPointcut new)
    receivers: #('PhGuestBrowser' 'PhUserBrowser');
    selectors: 'get:').

cacheAspect
  registerOn: (PhMembrane new advise: adminUserMembrane).
logAspect
  registerOn: (PhMembrane new advise: userGuestMembrane).
```

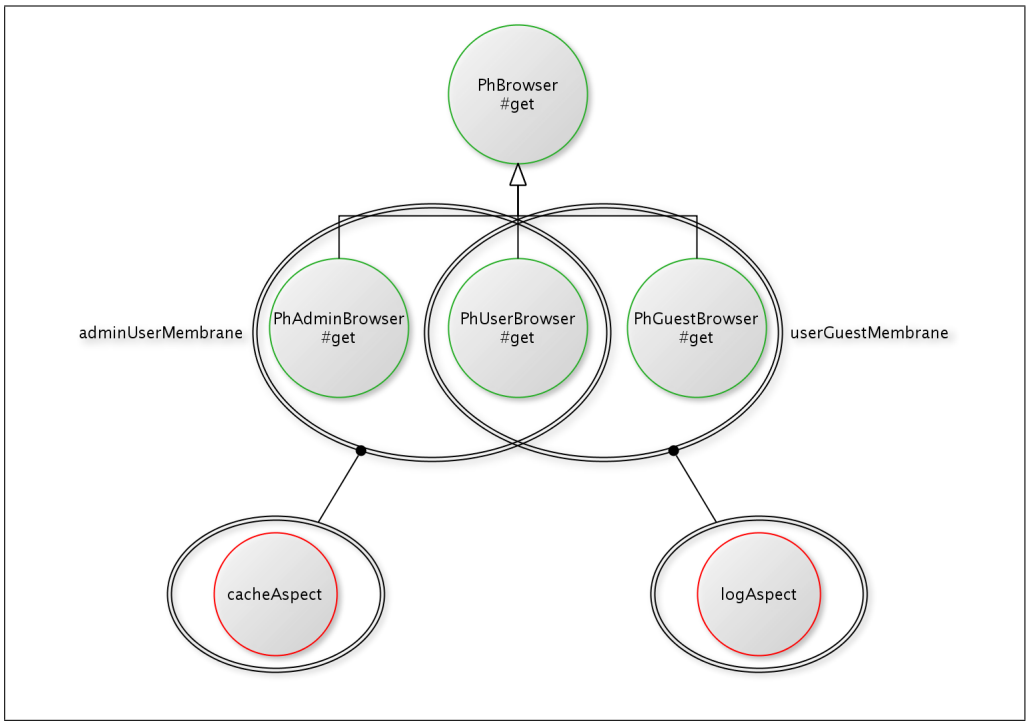

Figura 14: Ejemplo grafo de membranas

4.10. Tests

Para comprobar el funcionamiento del lenguaje, se usó el sistema de test unitarios existente en Pharo Smalltalk. Se construyó un total de 155 tests, agrupados en 13 categorías principales.

El grado de alcance de los test fue comprobado usando la herramienta Hapao [13], que permite visualizar el grado de cobertura que los test hacen sobre el código del programa. En la figura 15 se puede visualizar la cobertura que los test están realizando en la implementación de lenguaje. Los rectángulos más grandes representan a la clases del programa que participaron en los tests, las aristas entre estos rectángulos representan herencia entre esas clases. Los rectángulos más pequeños representan métodos dentro de las clases, y las aristas de éstos representan llamados entre ellos. El alto de los rectángulos de métodos, representa la complejidad ciclomática del método, el ancho representa el número de método diferentes que lo llaman, y la intensidad de gris representa el número de veces que el método ha sido ejecutado durante el test. Los rectángulos rojos representan métodos que no fueron ejecutados, los azules métodos abstractos, y los verdes representan a los métodos de test. Los rectángulos de clases, sin métodos al interior, que pueden verse en la esquina superior derecha de la figura 15, corresponden a clases que fueron creadas solamente para la realización de algunos tests.

El grado de cobertura obtenido fue de un 90.37 %, lo que está dentro de los rangos de un código bien probado [14].

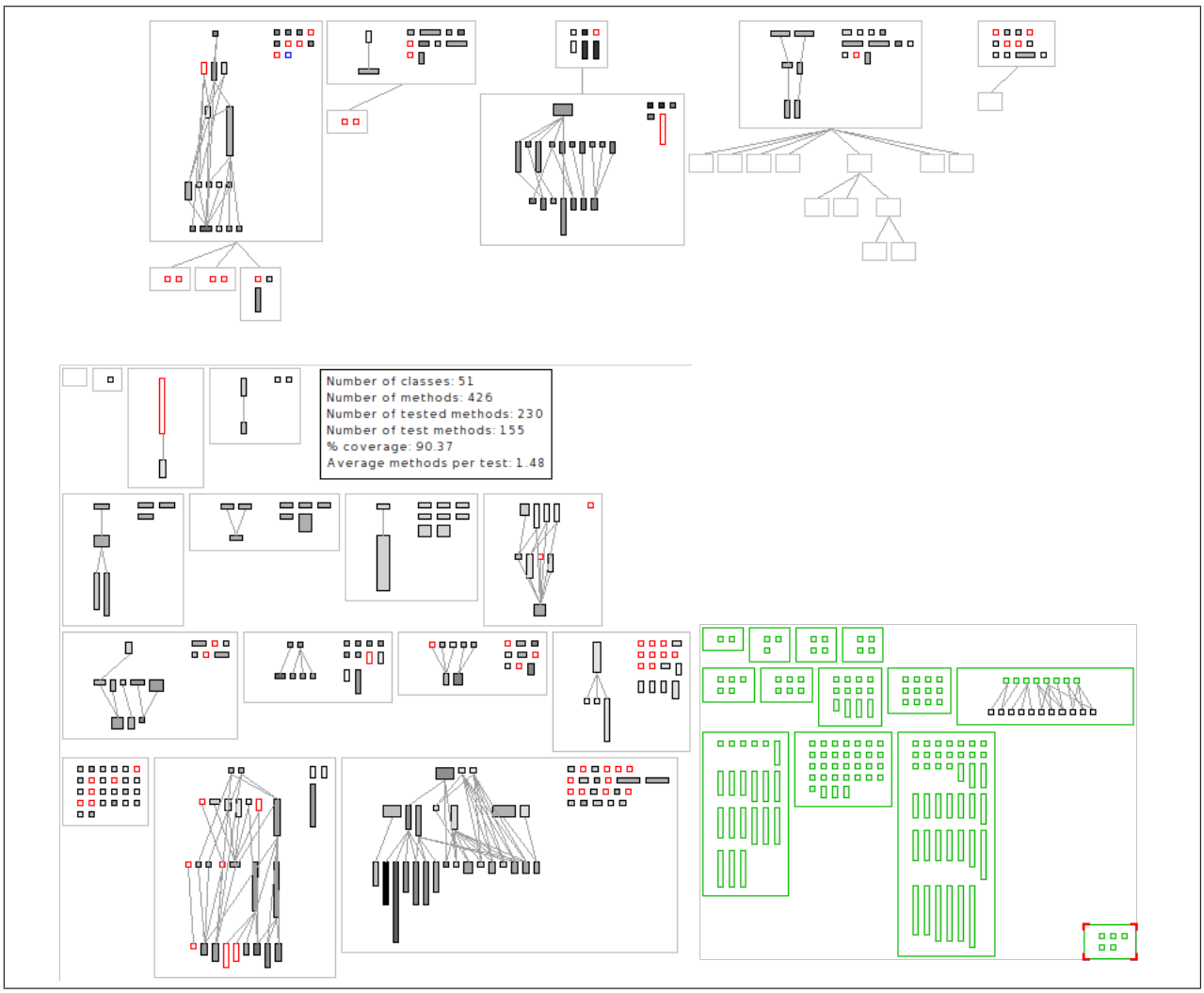

Figura 15: Test coverage con Hapao

4.11. Resumen

En esta sección se presentó la implementación de PHANtom, describiendo sus clases más importantes: PhAspect, PhAdvice, PhPointcut, PhReceiverParser, PhSelectorParser, PhContext, PhClassModifier, PhMembrane, PhMethodWrapper, PhAdviceRunner, PhAdviceGroup, PhAspectWeaver, PhPrecedenceRulesProcessor y AspectGraph. En 4.2 se presentaron los MethodWrappers, como estos son usados para capturar los *join points*, sin la necesidad de modificar el compilador o la máquina virtual, y los problemas que fué necesario solucionar para permitirlo. En 4.3 se describió la implementación usada para los *pointcuts*, junto al uso de PetitParser [12] para la implementación del sistema de patrones, facilitando la modificación y extensión de éste. En 4.4 se describió la implementación de los *advice*, y como estos están organizados para permitir la modificación dinámica del orden de ejecución. En 4.5 se presentó la implementación de los modificadores de clases en PHANtom, usando las capacidades reflexivas de Smalltalk. En 4.6 se presentó la relación que tienen los aspectos con

el sistema de membranas implementado. En 4.7 se describe el sistema de grafos implementado para el manejo de las reglas de precedencia, usando grafos dirigidos acíclicos, y orden topológico. En 4.8 se describió la implementación del *weaving* de los aspectos/membranas en el sistema, y como se interactúa con el sistema de notificaciones de Pharo Smalltalk, para manejar los cambios que ocurren al crear, eliminar o modificar las clases del sistema, por parte de los usuarios. También se describió la forma en que se maneja la aplicación de una membrana sobre un objeto, usando un diccionario con referencias débiles para el manejo de la pila de llamados. En 4.9 se presentó el manejo de las estructuras de membranas generadas, junto a algunos ejemplos, describiendo los grafos de membranas y como éstas controlan la visibilidad que tienen los aspectos de los *join points* que se generan durante la ejecución del programa. Finalmente, en 4.10 se describió el sistema de test usado, que consta de 155 tests unitarios, agrupados en 13 categorías. Además se presentó el resultado obtenido usando Hapao [13] para analizar la cobertura de los test, alcanzando un 90.37 % de cobertura, lo que es considerado como un código bien probado [14].

5. Aplicación

En esta sección se presentará la experiencia realizada al modificar la herramienta de profiling Spy [15] con el propósito de utilizar PHANtom para la instrumentación del sistema. En 5.1 se presentará Spy, y las modificaciones realizadas, y en 5.3 se presentarán los resultados obtenidos.

5.1. Spy

Spy [15] es un framework para el análisis dinámico de programas. Spy ha sido usado para la creación de profilers de programas y test coverage, entre otras aplicaciones.

Las principales clases de Spy son:

- Profiler: Esta clase esta encargada de instrumentar el código sobre el cual actuará el profiler, ejecutar el código, desinstrumentarlo y entregar una instancia de Profiler. La instancia entregada contiene la estructura del programa, usando instancias de PackageSpy, ClassSpy y MethodSpy.
- PackageSpy: Representa un Package del código instrumentado, conteniendo información de profiling del package que representa. Internamente contendrá instancias de ClassSpy.
- ClassSpy: Representa una clase del código instrumentando, conteniendo información de profiling de la clase que representa. Contiene las instancias de MethodSpy que representan los métodos de esta clase.
- MethodSpy: Representa a los métodos de las clases instrumentadas, y es la encargada de tomar la información de ejecución del código que se esta evaluando.

Para usar Spy, el usuario debe crear sub clases de Profiler, PackageSpy, ClassSpy y MethodSpy, redefiniendo algunos métodos de estas clases para obtener la información que requiere para su profiler. Por ejemplo, KaiProfiler, que es uno de los profilers incluidos en Spy, redefine el método run:with:in de la clase MethodProfiler, para capturar el número de llamadas que se realizan a un método. El método run:with:in es llamado cada vez que se realiza un llamado al método sobre el cual está actuando el profiler:

```
KaiProfilingMethodSpy>>run: aSelector with: aListOfArguments in:
    aReceiver
...
numberOfCalls := numberOfCalls + 1.
...
```

5.2. Refactoring de Spy con PHANtom

Con el fin de probar el lenguaje de aspectos desarrollado, se modificó la forma en que Spy instrumenta las clases del sistema, para que se usara PHANtom para el proceso de instrumentación.

Para realizar la instrumentación de las clases, Spy originalmente reemplazaba los CompiledMethod por un MethodSpy que actuaba como MethodWrapper. Por razones de optimización, actualmente modifica los CompiledMethod de cada uno de los métodos en el diccionario de métodos de las clases, cambiando las referencias internas del CompiledMethod. Con esto se delega la ejecución del método hacia el MethodSpy. Es la instancia de MethodSpy la que captura la información que el profiler requiere, y ejecuta el código original del método, llamando al CompiledMethod original.

Para PHANtom en Spy, se modificaron los métodos de instrumentación de las clases, y la forma en que se recolecta la información una vez realizado el profiler. Estos cambios se realizaron de manera que no cambiara la forma en que lo usuarios de Spy crean sus profilers. Para esto se crearon las siguientes clases:

- SpyPointcut: Una sub clase de PhPointcut que permite especificar los pares receptor-mensaje en forma directa, sin realizar la búsqueda de coincidencias sobre todas las clases del sistema. Ésto fue necesario ya que Spy instrumenta todos los métodos de todas las clases involucradas en el código que se está evaluando, y realizar el parsing de todas ellas usando PetitParser puede tomar bastante tiempo cuando son muchas las clases involucradas.

- SpyMembrane: Una sub clase de PhMembrane, que se encarga de generar una instancia de SpyMembrane para cada instancia de un Profiler que se ejecute. La instancia de SpyMembrane envolverá la computación sobre la que está interesado el profiler. Las membranas donde se registrarán los aspectos que realizarán el profiling observaran a ésta membrana.

- SpyAspect: Una sub clase de PhAspect. Se crea una instancia de SpyAspect por cada clase que se quiere instrumentar. Esta instancia contiene un advice por cada método de la clase, el cual realiza el llamado a la instancia correspondiente de MethodSpy que realizará la captura de los datos en los cuales está interesado el profiler. El código que ejecuta MethodSpy es ejecutado dentro de la membrana creada por SpyMembrane, esto, debido a que es necesario que los join points generados por la ejecución del método original, sean vistos por los aspectos.

5.3. Resultados

Se probaron dos de los profilers que vienen junto a Spy:

- KaiProfiler: Realiza un profiling sobre la ejecución de un programa, analizando el tiempo que toma cada método, las relaciones entre ellos, el número de veces que cada método ha sido llamado y el número de diferentes receptores en que el método ha sido ejecutado. Junto a lo anterior, también se puede ver el grafo de llamadas de cada método.

- MemoryProfiler: Realiza un profiling sobre la ejecución de un programa, analizando el tiempo de ejecución de cada método, el número de bytes asignados en memoria por cada método, y el valor de retorno.

5.3.1. KaiProfiler

Se aplicó KaiProfiler sobre la ejecución de los test en *RBFormatterTest*, que corresponde a test sobre las clases responsables de formatear código. En la figura 16 se puede apreciar una vista general de la visualización obtenida usando KaiProfiler, en la figura 17 se puede ver un extracto de la visualización, los rectángulos grandes representan las clases, las aristas entre ellos representan la herencia. Los rectángulos pequeños representan los métodos de las clases, la altura de éstos representa el tiempo que tomó su ejecución, y el ancho representa el número de veces que ha sido ejecutado, finalmente la intensidad del color representa el número de distintos receptores en que ha sido ejecutado. En la figura 18 se puede ver el mismo extracto, pero usando PHANtom para realizar la instrumentación de las clases. Como se puede apreciar, existen diferencias entre ambas visualizaciones, sin embargo, en la figura 19 se puede apreciar el mismo extracto, cuando se cambia la forma de instrumentación de Spy de las clases para usar solamente MethodWrappers. Se puede ver que no existe gran diferencia entre la visualización realizada usando PHANtom, con aquella realizada usando solamente MethodWrappers. Las diferencias obtenidas entre la figura 17 y 18 pueden ser atribuidas al uso de MethodWrappers, y la forma en que la máquina virtual maneja la ejecución de estos.

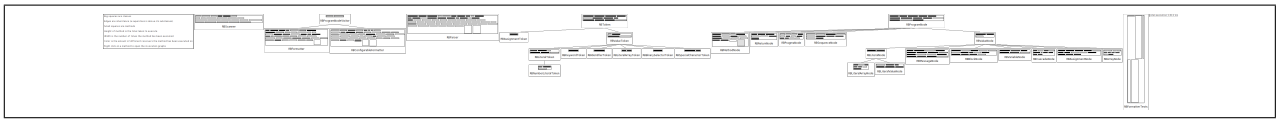

Figura 16: KaiProfiler Spy

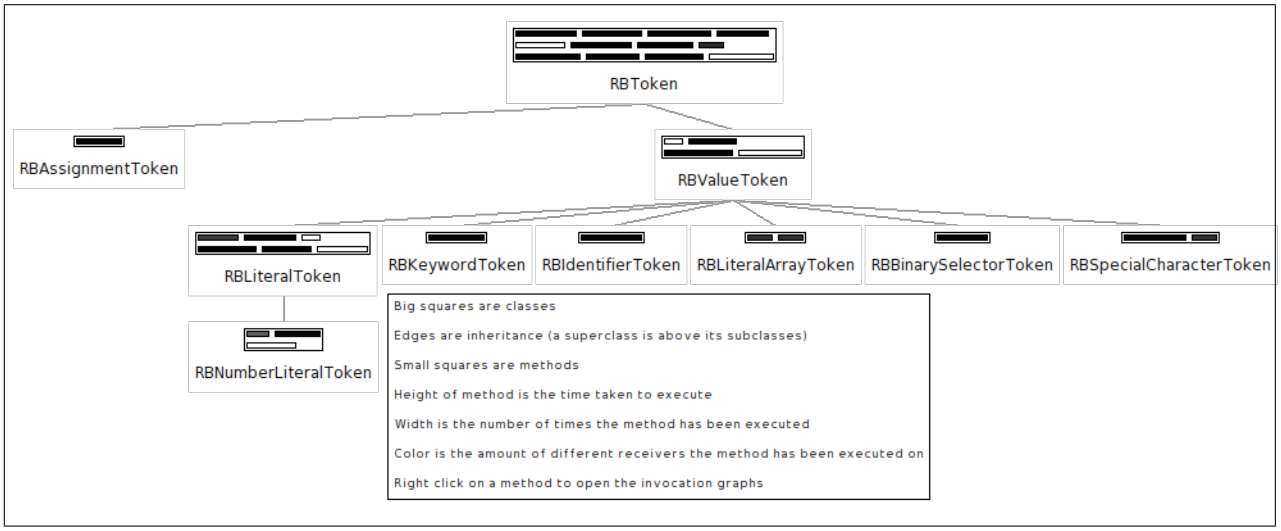

Figura 17: Extracto KaiProfiler Spy

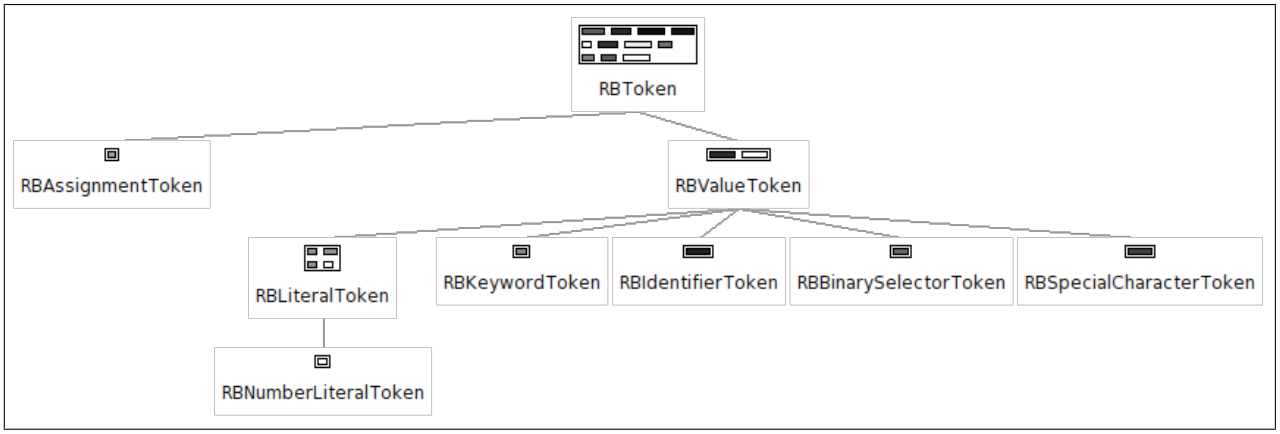

Figura 18: Extracto KaiProfiler PHANtom-Spy

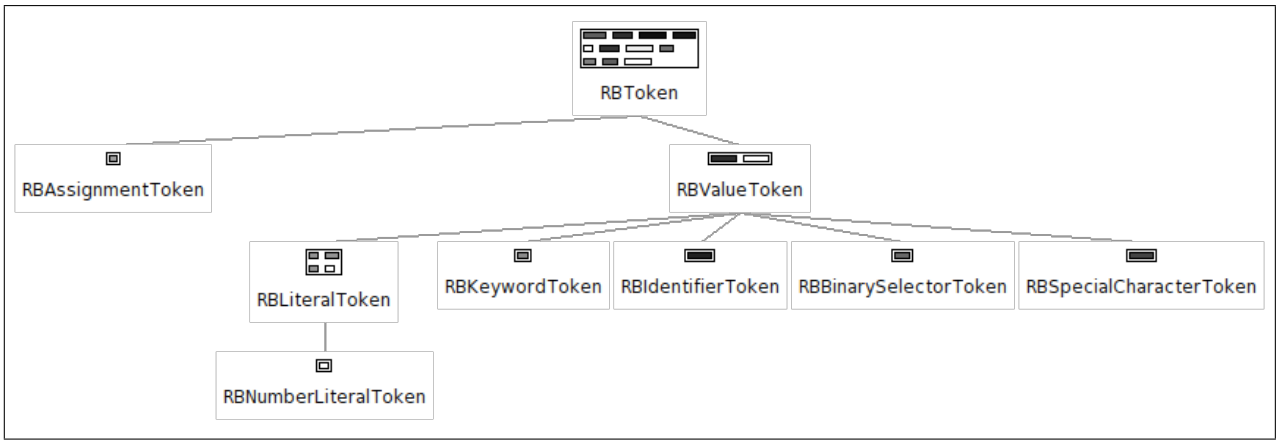

Figura 19: Extracto KaiProfiler MethodWrappers-Spy

5.3.2. MemoryProfiler

Se aplicó MemoryProfiler sobre los mismos test realizados en 5.3.1. En la figura 20 se puede ver un extracto de la visualización obtenida usando MemoryProfiler. El rectángulo grande representa una clase, los rectángulos más pequeños representan métodos de esa clase. La altura de los rectángulos pequeños representa el tiempo de ejecución del método, el ancho representa los bytes asignados en memoria. Un rectángulo pequeño de color gris representa que el valor de retorno es el objeto sobre el que se llamó el método, un rectángulo de color naranja representa que el valor de retorno es constante por receptor del mensaje. En la figura 21 se observa la misma visualización, pero usando PHANtom para realizar la instrumentación del código. Se puede observar que existe una diferencia entre ambas, principalmente en el ancho de los rectángulos, que representan la memoria asignada para cada uno de los métodos, eso puede ser explicado por la diferencia en la forma de instrumentación, lo que genera diferencias en la memoria necesaria para los métodos de las clases. En la figura 22 se puede ver el mismo extracto, pero usando los MethodWrappers de Spy para realizar la instrumentación, similar a lo ocurrido en 5.3.1, la visualización es más parecida a lo obtenido usando PHANtom, las diferencias son mayores a lo ocurrido en 5.3.1 ya que este profiler mide el espació alocado en memoria para cada método, y PHANtom necesita alocar en memoria una serie de estructuras para poder manejar las estructuras de membranas que se generan, por lo cual se observa una diferencia notoria comparado a la memoria que se usa solamente para los MethodWrappers.

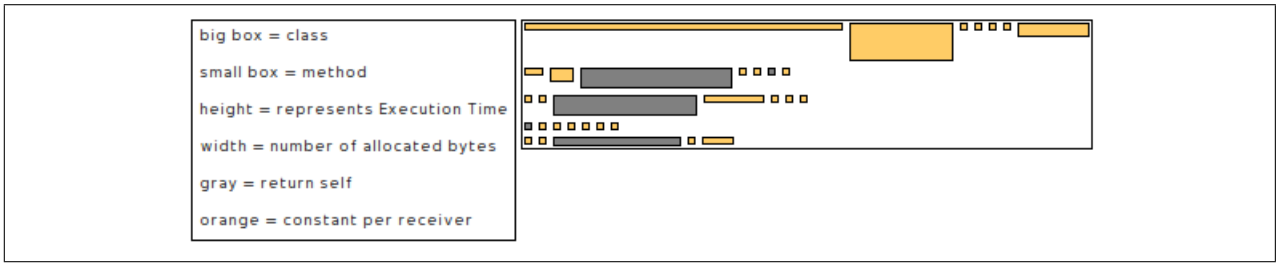

Figura 20: Extracto MemoryProfiler Spy

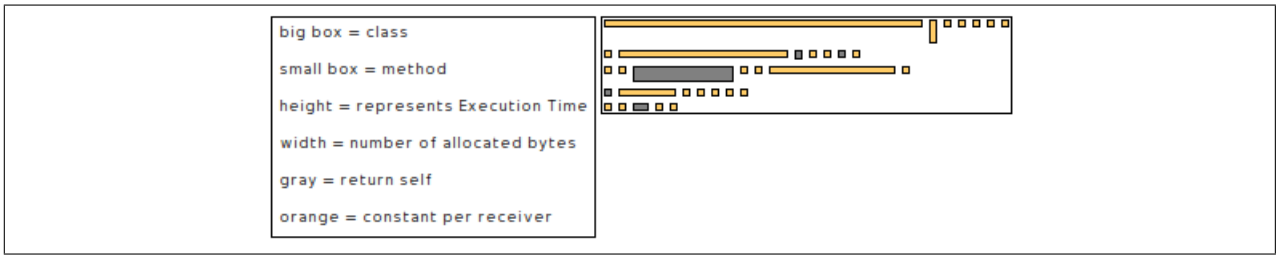

Figura 21: Extracto MemoryProfiler PHANtom-Spy

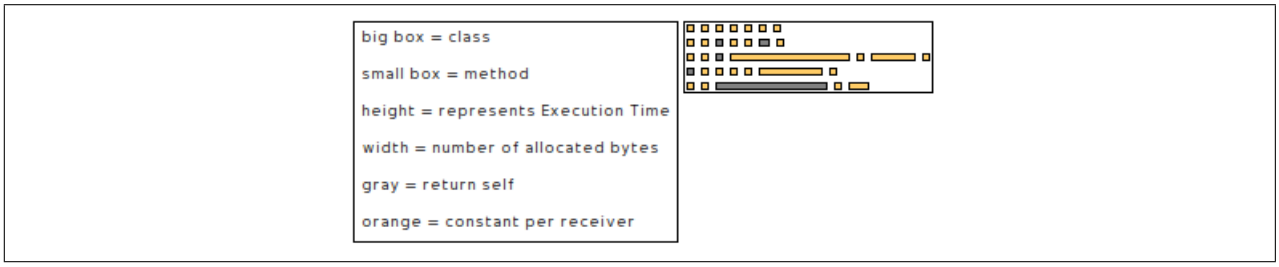

Figura 22: Extracto MemoryProfiler MethodWrappers-Spy

5.3.3. Benchmark

Se midió el tiempo que toma realizar un profiling para lo siguientes bloques de código:

- A: `[RBFormatterTests suite run]` que ejecuta los test a los que se hace referencia en 5.3.1 y 5.3.2.
- B: `[| view |`
  `view := MOViewRenderer new.`
  `view nodes: (1 to: 10) forEach: [:each | view nodes: (1 to: 10)].`
  `view root applyLayout]` que genera la visualización de 10 rectángulos grandes, cada uno conteniendo en su interior 10 rectángulos pequeños, usando el motor de visualizaciones que usa Spy para generar la visualización de los profilers.

Se tomó el tiempo para Spy, Spy usando MethodWrappers, y Spy usando PHANtom. Como se puede ver en la figura 23, existe una gran variación en el tiempo que toma realizar el profiling de estos dos bloques de código. Comparando Spy con MW vs Spy con PHANtom, se puede ver que usar PHANtom aumenta el tiempo de ejecución del profiler, lo que se explica por el control en los llamados a método que debe generar PHANtom para mantener la estructura de membranas. La diferencia en el tiempo de ejecución entre los bloques A y B, se debe a que en el bloque B se genera una gran cantidad de invocaciones entre métodos, aumentando la influencia generada por PHANtom en el tiempo de ejecución.

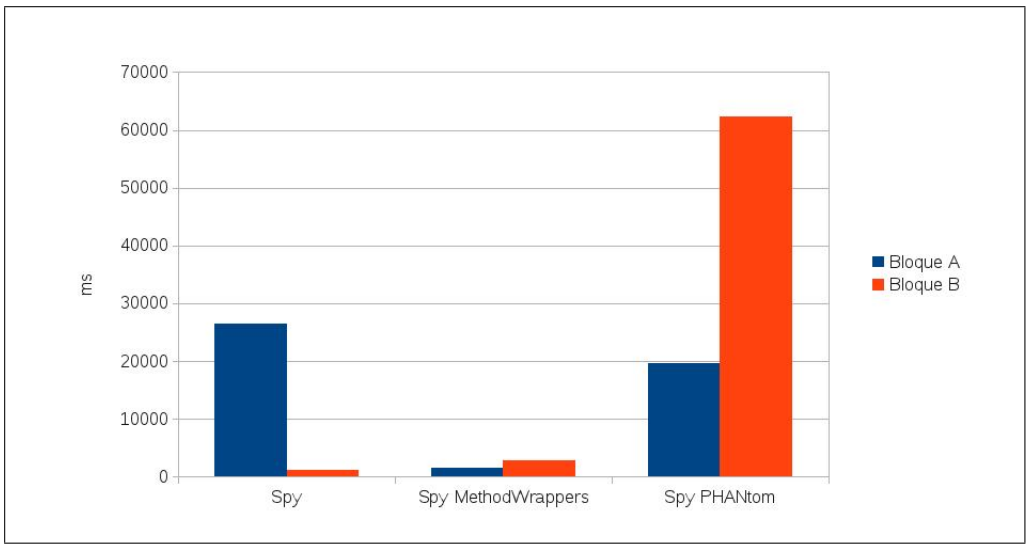

Figura 23: Benchmark Spy-PHANtom

5.4. Discusión

Como se vió en 5.3 es posible utilizar PHANtom para realizar la instrumentación del código que requiere Spy para funcionar. Las ventajas que presenta el uso de PHANtom tal como fue implementado para estas pruebas, son una mayor modularización en la forma en que se instrumenta el código, y un mejor control de los métodos instrumentados, ya que estos se encuentran dentro de un aspecto por clase, y todo el conjunto de aspectos en una membrana por instancia de profiler.

Un problema que se presenta con la versión actual, es cuando un profiler intenta inspeccionar la ejecución de si mismo, esto puede provocar problemas de reentrancia, al evaluar la infraestructura de Spy y PHANtom. Por esto, un siguiente paso que podría realizarse en la integración de PHANtom con Spy, es la separación de la ejecución del método original contenido en el MethodSpy, de la ejecución de los métodos necesarios para realizar el profiling del código. Sin embargo, para realizar esta separación es necesaria modificar Spy de forma más profunda que el alcance de esta memoria.

Las desventaja que se presenta es principalmente la pérdida en la eficiencia del profiler, debido al sobre costo que requiere la mantención de la estructura de membranas que utiliza PHANtom, junto al mayor costo en memoria que requieren éstas estructuras.

5.5. Resumen

En esta sección se presentó la experiencia realizada al usar PHANtom en la herramienta de profiling Spy [15], con el fin de utilizar PHANtom para realizar la instrumentación del sistema. Para esto se modificaron las clases de Spy responsables de la instrumentación del sistema, y de recolección de la información luego de realizado el profiling. Estas modificaciónes

se realizaron de manera tal que no afectara la forma de uso de Spy. Estas modificaciones se probaron en KaiProfiler, un profiler que analiza el tiempo que toman los métodos en ejecutarse, las relaciones entre ellos y número de llamados y receptores; y en MemoryProfiler, que analiza el tiempo que toman los métodos en ejecutarse, los bytes asignados en memoria, y el valor de retorno. Junto a esto se realizó un benchmark, donde se apreció una gran variabilidad en los tiempos que toman en ejecutarse los distintos profilers, dependiendo del código al cual son aplicados. Se pudo apreciar que el uso de PHANtom presenta las ventajas de permitir una mejor modularización en la forma en que se instrumenta el código y un mejor control de los métodos instrumentados, sin embargo, esto tiene un costo en eficiencia, tanto en tiempo de ejecución como en memoria.

6. Conclusiones

Durante éste trabajo se logró desarrollar un lenguaje de aspectos sobre Pharo Smalltalk, integrando las funcionalidades básicas encontradas en otros lenguajes de aspectos. Además la implementación realizada incluye una serie de funcionalidades avanzadas, tales como:

- Control de precedencia a nivel de pointcut, que permite un grado mayor de granularidad en el orden de los aspectos, logrando también una mayor flexibilidad al tener la posibilidad de combinar el orden global, entregado por la precedencia definida en los aspectos, junto al orden definido en los pointcuts.

- Control dinámico del orden de los advice, que permite a un aspecto acceder al conjunto de advice que van a ejecutarse en un join point determinado, con lo que se puede eliminar, agregar y cambiar el orden de ellos. Esto otorga una gran flexibilidad al lenguaje, permitiendo crear aspectos que controlen la ejecución de advice generados por otros aspectos.

- Uso de membranas computacionales, que otorgan un mayor control sobre el alcance de los aspectos en el sistema, permitiendo entre otras cosas resolver el problema de la reentrancia de los aspectos. Este sistema fue implementado de una forma transparente para los usuarios, permitiendo el uso del lenguaje de aspectos sin hacer referencia directa al sistema de membranas, pero permitiendo que si los usuarios así lo desean, puedan definir sus propias estructuras de membranas.

Además, se integró PetitParser para la definición de los pointcuts, permitiendo una mayor flexibilidad para la especificación de éstos, con lo cual los usuarios pueden utilizar sus propios parsers. Junto a esto, el sistema de patrones fue también desarrollado usando PetitParser, lo que otorga mayor facilidad para extender este sistema de patrones para permitir definiciones más especificas para los receptores y selectores.

Se realizó una prueba del lenguaje, usándolo para refactorizar algunas partes del software de análisis dinámico Spy, comprobando luego el funcionamiento de dos de los profilers que están incluidos con Spy. Con esto se obtuvo una mejor modularización de las partes de Spy encargadas de instrumentar código, sin embargo, se pudo ver en algunas de las pruebas realizadas, que se generó un sobre costo en tiempo y memoria al usar PHANtom.

El uso de las herramientas de test fue un factor muy importante durante el desarrollo de este trabajo, lográndose una batería de test que comprueban más del 90 % de la infraestructura del lenguaje desarrollado, otorgando una mayor seguridad sobre la estabilidad y funcionamiento del lenguaje. Esto fue comprobado usando la herramienta Hapao, como se mencionó en la sección 4.10.

7. Trabajo futuro

Una funcionalidad que no fue implementada corresponde a la posibilidad de especificar un join point en base al *control flow*, esto es, especificar que el join point debe ocurrir dentro de un flujo de ejecución determinado. Esta funcionalidad puede ser implementada directamente en Pharo Smalltalk, ya que el lenguaje permite hacer consultas al contexto en el cual se está ejecutando un determinado método, e inspeccionar la pila de llamadas. Sin embargo, la reificación de la pila de llamadas es un proceso costoso. Por ésto nos parece importante contar con ésta funcionalidad dentro de la especificación de los pointcuts, permitiendo además realizar optimizaciones y evitando la reificación de la pila de llamados cuando no es necesario.

Como se pudo ver en la sección 5, existen un sobre costo importante al usar PHANtom dentro de una aplicación. Por esto hay cuatro cosas importantes que pueden ser optimizadas:

- El parsing de las clases y métodos del sistema, para encontrar las coincidencias que los pointcuts describen, es un proceso costoso, el cual puede ser optimizado para mejorar la velocidad de instalación de los aspectos.

- Los protocolos de actualización de los aspectos frente a cambios en el sistema, como nuevos métodos y clases, puede ser mejorado para que se actualicen solamente las partes afectadas.

- El manejo de las estructuras de membranas, donde puede ser usado un sistema de cache para los advice que deben activarse frente a un conjunto determinado de membranas activas, y para un join point especifico, reduciendo los tiempos de ejecución.

- Realizar parte del weaving de los aspectos en tiempo de compilación, colocando el código necesario para la infraestructura de aspectos y membranas, directamente en el CompiledMethod que genera el compilador de Pharo Smalltalk. Esto podría ayudar a disminuir el sobre costo que se genera, realizando análisis estático del programa en tiempo de compilación.

Finalmente, nos parece necesario integrar el lenguaje de aspectos desarrollado con las herramientas de depuración existentes en Pharo Smalltalk, ya que cuando se generan errores en el código de los aspectos desarrollados, las herramientas de depuración de Pharo Smalltalk hacen referencias a la infraestructura de PHANtom, por lo que la información presentada al usuario tiende a ser poco clara, no haciendo referencia directa al código que el usuario esta desarrollando.

Referencias

[1] The AspectJ Programming Guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`, 2011.

[2] Ali Assaf and Jacques Noyé. Dynamic aspectj. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 8:1–8:12, New York, NY, USA, 2008. ACM.

[3] Éric Tanter, Nicolas Tabareau, and Rémi Douence. Exploring membranes for controlling aspects. Technical Report TR/DCC-2011-8, University of Chile, June 2011.

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. Springer-Verlag, 1997.

[5] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. In Jorgen Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001.

[6] Hridesh Rajan and Kevin J. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *In ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68. ACM Press, 2005.

[7] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, 63:207–239, December 2006.

[8] Pharo Smalltalk. `http://pharo-project.org/home`, 2011.

[9] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[10] Squeak Smalltalk. `http://www.squeak.org`, 2011.

[11] John Brant, Brian Foote, Ralph Johnson, and Donald Roberts. Wrappers to the rescue. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 396–417. Springer Berlin / Heidelberg, 1998.

[12] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, jun 2010.

[13] Vanessa Peña Araya and Alexandre Bergel. Test coverage with hapao. In *Proceedings of the 5th Workshop on Dynamic Languages and Applications*, 2011.

[14] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 291–301, Washington, DC, USA, 2009. IEEE Computer Society.

[15] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. In *Smalltalks 2010*, 2010.