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

EL LENGUAJE DE ASPECTOS AMISTOSO CASPER

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MIGUEL ENRIQUE CAMPUSANO ARAYA

PROFESOR GUÍA:
JOHAN FABRY

MIEMBROS DE LA COMISIÓN:
NANCY HITSCHFELD KAHLER
LUIS MATEU BRULE

SANTIAGO DE CHILE
ABRIL 2013

Resumen

Mientras los programas se vuelven más complejos, nacen nuevos problemas. Uno de ellos es la poca modularidad que se tiene con las llamadas funcionalidades transversales. Estas funcionalidades están dispersas por toda la aplicación y no pueden separarse mediante el paradigma de la programación orientada a objetos. Para solucionar este problema nace el paradigma de la programación orientada a aspectos.

En la programación orientada a aspectos se modularizan las funcionalidades transversales en una entidad llamada *aspecto*. El aspecto se ejecuta donde corresponde gracias a que, conceptualmente, el programa se está monitoreando por el aspecto en toda su ejecución, decidiendo cuando se debe ejecutar una acción provista por el aspecto. Lamentablemente, el monitoreo de la aplicación conlleva a un sobrecosto que, muchas veces, los desarrolladores no están dispuestos a pagar.

En este trabajo se presenta Casper, un lenguaje orientado a aspectos construido sobre Pharo Smalltalk. Las características más importantes buscadas por Casper son la simplicidad de uso, la explicitud de lo que sucede con los aspectos y bajar el sobrecosto de la ejecución de los programas que utilicen Casper.

Casper basa sus funcionalidades en PHANtom. PHANtom es un lenguaje de aspectos construidos sobre Pharo Smalltalk. Ambos lenguajes tienen sintaxis similares, pero las implementaciones son muy diferentes. Casper busca mejorar los sobrecostos generados al usar PHANtom. Para ello Casper propone compilar las funcionalidades dadas por los aspectos junto con los métodos originales. Gracias a la compilación, Casper expone las funcionalidades al desarrollador, en cambio, PHANtom esconde las funcionalidades y nunca muestra de forma explícita donde se ejecuta una funcionalidad transversal.

Se comprobó el uso de Casper refactorizando SPY. SPY es un framework para el análisis dinámico de programas. SPY instrumentaliza los métodos que se quieren analizar. Casper provee la instrumentalización de SPY con aspectos. Se realizaron pruebas sobre SPY versus SPY con Casper versus SPY con PHANtom. Se tomaron los tiempos sobre la preparación de la aplicación misma y lo que demora en ejecutar. Estos resultados comprueban un sobrecosto del uso de aspectos en la aplicación. Sin embargo, también demuestra que la ejecución de la refactorización de SPY con Casper es mucho menos costosa que la refactorización de SPY usando PHANtom.

Gracias por apoyarme en mis decisiones, de a poco... volveré.

Agradecimientos

A mis padres que siempre respetaron y me apoyaron en mis estudios. Me dieron el tiempo necesario y el espacio que necesitaba para la realización de este trabajo.

A .::MB::. y al Morocho. Siempre me apoyaron en los momentos más complicados de este trabajo y mis estudios. También quiero agradecer a mis amigos de la Universidad y departamento, sin su ayuda no solo este trabajo, sino todo el camino recorrido en la universidad habría sido mucho más difícil. No puedo dejar de mencionar a los Talentosos Campesinos, siempre recordaré los logros que tuvimos y que casi tuvimos.

Y finalmente, pero no menos importante, quiero agradecer a mi profesor guía, Johan Fabry, que sin su ayuda y constante apoyo esta memoria nunca habría salido adelante.

Tabla de Contenido

Capítulo 1

Introducción

La programación orientada a aspectos es un paradigma de programación que busca incrementar la modularidad intentando solucionar el problema de las funcionalidades transversales, esto es, funcionalidades que se encuentran dispersas en muchos sectores de código de la aplicación. El problema surge ya que el actual paradigma de programación orientada a objetos no logra una buena capa de abstracción frente a este tipo de funcionalidades. Un ejemplo clásico de una funcionalidad transversal es cuando se quiere hacer un registro de lo que sucede en un programa. Esto es, realizar un logger sobre eventos que se definen importantes. La forma de hacer esto es escribir esta funcionalidad dentro de todos los eventos que se deseen registrar. La solución que emplean los aspectos es modularizar esta funcionalidad en una sola entidad: un aspecto, que se llama automáticamente en los eventos que se están registrando.

La programación orientada a aspectos logra modularizar esto porque, conceptualmente, los aspectos están monitoreando la ejecución del programa y deciden cuando deben o no ejecutar un nuevo comportamiento o modificar el comportamiento del programa mismo.

Actualmente existen muchos lenguajes orientados a aspectos que se construyen sobre otros lenguajes. En Java tenemos AspectJ [8] y CaesarJ [2], Eos-U [9] para C#, AspectScheme [4] para Scheme entre otros. Uno de los lenguajes de aspectos más estudiados para la realización de este trabajo es PHANtom [6], que está construido sobre Pharo Smalltalk.

En este trabajo se presenta el diseño y la construcción de un lenguaje de aspectos escrito sobre Pharo Smalltalk. Smalltalk es un lenguaje de programación orientado a objetos con un gran soporte para realizar metaprogramación. Pharo es un ambiente para poder programar en Smalltalk. El lenguaje presentado en este trabajo se basa en la sintaxis propuesta por PHANtom, buscando la simpleza y la facilidad para el programador. El lenguaje que se desarrolla en este trabajo es llamado Casper.

1.1. Terminología

Los lenguajes orientados a aspectos introducen nuevas estructuras y terminologías para soportar la infraestructura que ofrecen.

- **Join point** Es un punto de ejecución del programa que se considera importante por el sistema de aspectos.
- **Join point Shadow** Es la proyección estática de un join point en el código fuente. Esto es, es el sector del código fuente que genera ese join point cuando se corre.
- **Pointcut** Es la estructura del lenguaje que captura los join points mediante expresiones regulares.
- **Advice** Es la funcionalidad que se agrega al sistema. Usualmente esta asociado a un pointcut que define en que momento se ejecutará.
- **Aspect** Representa una funcionalidad transversal en el sistema. Se compone de pointcuts y advices.
- **Weaving** Es el proceso de tomar todos los aspectos e integrarlos en el programa.

1.2. Motivación

La industria y el mundo científico han comenzado a poner atención a los lengajes orientados a aspectos. Pero aun así el mundo industrial no esta familizarizado del todo con estas herramientas. El paradigma de la programación orientada a aspectos es algo complejo, incluso muchos programadores no entienden completamenten el funcionamiento de esta metodología. Con tan solo unas pocas lineas de código, la programación orientada a aspectos puede relizar funcionalidades de forma muy compleja. Además, la programación orientada a aspectos siempre viene con un aumento del tiempo de ejecución de los programas, debido a las mayores funcionalidades que se ejecutan y el cómo se ejecutan.

Por eso es interesante minimizar la complejitud del lenguaje orientado a aspectos, haciendo visible y entendible el comportamiento del lenguaje de aspectos en el sistema. También es muy importante minimizar los tiempos de ejecución de los programas que utilizan la programación orientada a aspectos. Así las personas y las industrias tendrán mayores herramientas para la realización de programas con mayor modularización y abstracción.

1.3. Objetivos

El objetivo principal de este trabajo es desarrollar Casper, un lenguaje orientado a aspectos construido sobre Pharo Smalltalk. El objetivo de Casper es ayudar a las

personas a escribir programas y entender el código que escriban en el paradigma de la programación orientada a aspectos. Para lograrlo se propone hacer modificaciones del código fuente de sectores identificados por los join point shadows donde habrán aspectos que se ejecutan, así el programador podrá ver exactamente lo que sucede en el sistema que él construye.

Otro objetivo fundamental de Casper es mejorar la velocidad de ejecución del lenguaje. Para ello también se propone compilar los métodos indentificados por los pointcuts, en lugar de usar la estrategia de los *Method Wrappers*. Los *method wrappers* permiten capturar un método cualquiera del sistema y agregarle mayor información y funcionalidades. PHANtom utiliza la estrategia de los *method wrappers* como una forma simple de lograr el *weaving* del lenguaje, pero a costa de bajar el rendimiento en tiempo de ejecución.

En la actualidad PHANtom no soluciona los dos problemas fundamentales anteriores que se atacan con Casper. Debido a la utilización de method wrappers, la ejecución de un programa modularizado con PHANtom es más lenta que la ejecución de un programa sin modularizar, esto es, con funcionalidades transversales repartidas en el código fuente del programa. Además, PHANtom se aprovecha de la metaprogramación para cambiar la estructura interna del programa, sin dejar a la vista lo que sucede. En Casper, siempre se puede ver, en el código fuente, lo que se ha modificado.

Objetivos específicos

- Diseñar y construir las especificaciones de los siguientes elementos del lenguaje de aspectos Casper: pointcuts, advices, aspectos y modificadores de clases.
- Hacer que el rendimiento de Casper sea mejor que PHANtom. Para ello se propone compilar el código en lugar de interpretar al nivel meta del lenguaje
- Facilitar que los programadores y las empresas decidan utilizar el paradigma de la programación orientada a aspectos en sus proyectos. Así se podrá llevar el mundo de los aspectos de un ámbito netamente científico al ámbito industrial. Para lograrlo Casper se crea de forma de ser un lenguaje fácil de usar y explícito en los cambios que realiza en el sistema.

1.4. Estructura

Primero en la sección 2 se mostrará todo el trabajo previo de investigación que fue ocupado para la realización de Casper. Luego en la sección 3 se presentará Casper, su sintaxis y formas de uso. En la sección 4 se verán todos los detalles de la implementación de Casper, los cuales son totalmente diferentes con la implementación de PHANtom. Luego, en la sección 5 se presentará un caso de estudio, una aplicación real para Casper. Finalmente en la sección 6.1 se verán las conclusiones de este trabajo y en la sección 6.2 se hablará sobre las posibilidades de cómo continuar este trabajo.

Capítulo 2

Trabajo Relacionado

Actualmente hay varios lenguajes de aspectos desarrollados, algunos más avanzados que otros. Para lograr una buena implementación de Casper, se tomaron como estudio e inspiración otros lenguajes orientados a aspectos.

2.1. AspectJ

AspectJ [8] es uno de los lenguajes orientados a aspectos más conocidos hoy en día, convirtiéndose en un ejemplo de muchos de los lenguajes de aspectos que hay en la actualidad. Esto debido a la simplicidad y usabilidad de AspectJ para el usuario final. AspectJ es una implementación de aspectos que trabaja sobre el lenguaje de programación Java. Además, se encuentra bien integrado con el IDE Eclipse, tanto para mostrar la sintaxis como para mostrar las funcionalidades transversales que se están capturando. Las funcionalidades más importantes de AspectJ son: uso de patrones para la definición de pointcuts (sección 2.1.1), soporte para modificadores de clase (llamados *Inter-type declarations*, sección 2.1.3) y definición de reglas de precedencia para los aspectos (sección 2.1.4).

En AspectJ el *weaving* de los aspectos se realiza a tiempo de compilación o carga de clases. Lo que hace es cambiar directamente el *bytecode* del programa original en la máquina virtual de Java, generando otro programa válido que puede correr en la máquina virtual de Java.

Los puntos más importantes del modelo de join points de AspectJ son los siguientes [1]:

- **call(Signature)** corresponde a cada llamada de un método o constructor que es capturado por el patrón definido en *Signature*
- **execution(Signature)** corresponde a cada ejecución de un método o constructor que es capturado por el patrón definido en *Signature*
- **get(Signature)** cada referencia a cualquier variable que es capturada por el

patrón definido en *Signature*

- **set(Signature)** cada asignación a cualquier variable que es capturada por el patrón definido en *Signature*
- **handler(TypePattern)** cada manejo de excepción para cualquier tipo *Throwable* en *TypePattern*
- **adviceexecution()** cualquier ejecución de cualquier *advice*

En las siguientes secciones se hablará sobre las funcionalidades importantes de AspectJ. Todos los ejemplos que se presentarán son obtenidos de *The AspectJ Programming Guide* [1]

2.1.1. Pointcut

En AspectJ, existen dos tipos de pointcuts: *name-based crosscuting* y *property-based crosscuting*. Los *name-based crosscuting* son pointcuts basados en dar la firma de un método de forma explícita. El siguiente ejemplo muestra un pointcut que captura la llamada al método *setX* que recibe un argumento de tipo *int* y que no retorna ningún argumento (su retorno es *void*), de la clase *Point*.

```
call(void Point.setX(int))
```

Los pointcuts del tipo *property-based crosscuting* proveen mecanismos que especifican join points mediante sus propiedades y no por su nombre explícito. Para ello, se usan patrones. En el siguiente ejemplo se muestra esta funcionalidad, acá se capturan todas las llamadas a métodos públicos de la clase *Figure*, donde los métodos pueden retornar cualquier cosa y recibe cualquier argumento y cualquier cantidad de argumentos.

```
call(public * Figure.*(..))
```

Los patrones de este tipo de pointcuts están especificados en la tabla 2.1.

Comodín	**Significado**
*	Captura cualquier secuencia de caracteres, exceptuando "."
..	Captura cualquier secuencia de caracteres, incluyendo "."
+	Captura el tipo especificado en el patrón, más todas las subclases o subinterfaces de aquel tipo

Tabla 2.1: Comodines para patrones de la captura de pointcuts en AspectJ

Composición

En AspectJ se permite componer pointcuts usando operadores booleanos. Los operadores permitidos para componer pointcuts son: **and** (&&), **or** (||) y **not** (!). Por

ejemplo, se puede definir el pointcut que captura todos los join points que se generan cuando una instancia de la clase *FigureElement* se mueve, de la siguiente forma:

```
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));
```

Exposición del Contexto

Los pointcuts tienen una interfaz. Esta interfaz permite exponer ciertas partes del contexto cuando está ocurriendo la ejecución de un join point capturado. Gracias a esto, se puede tener mucha más información y ser usada después en el advice o en los llamados "pointcuts condicionales". La forma de exponer variables al contexto es usando el sistema de tipo de Java junto con el nombre de los parámetros. Así AspectJ hace una designación uno a uno con los nombres de los parámetros.

Además, se usan tres palabras claves distintas para exponer el contexto deseado. Estas palabras son:

- **this** Se expone el objeto que se está ejecutando en el join point
- **target** Se expone el objeto que está siendo llamado o la variable que está siendo accedida en el momento de la ejecución del join point
- **args** Expone los argumentos que recibe el objeto que está siendo ejecutado o que será llamado

Por ejemplo, el siguiente pointcut captura todos los métodos que reciben un argumento de tipo *int*. El pointcut expone el argumento en la variable "i".

```
call(public *.*(int)) && args(i);
```

Named Pointcut

Los pointcuts en AspectJ pueden ser nombrados usando la palabra reservada *pointcut*. A diferencia de los pointcuts anteriores, los cuales son llamados pointcut anónimos, los pointcuts que pueden ser nombrados se llaman *named pointcut*. Además, un *named pointcut* pública el contexto que expone para que pueda ser usado por el advice o por otro *named pointcut*.

En el siguiente ejemplo se define el pointcut *setXY*, el cual se compone del pointcut *call(void FigureElement.setXY(int, int))* más las exposiciones del contexto de los argumentos y de la clase. El contexto expone la clase llamada por *call*. Esta clase queda guardada en la variable *fe*. Además el contexto también expone los argumentos en las variables x e y.

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

Verificación Dinámica

En AspectJ es posible tener verificación dinámica de pointcuts. Para lograr esto se puede usar la forma condicional por medio de la expresión *if* o por medio del flujo de ejecución del programa.

Cuando se usa la expresión *if* se hablan de pointcuts **Condicionales**. La expresión *if* puede recibir cualquier expresión booleana válida en Java. En el siguiente ejemplo se crea un pointcut que captura cualquier llamada sobre cualquier clase de un método que tenga un argumento de tipo *int*, siempre y cuando este argumento sea mayor que cero.

```
pointcut someCallWithIfTest(int i):
  call(* *.*(int)) && args(i) && if(i > 0);
```

Para verificar un pointcut por medio del flujo de ejecución del programa se usan las siguientes expresiones:

- **cflow(Pointcut)** Captura todos los join points que están dentro del flujo de ejecución de los join points P capturados por *Pointcut*, incluyendo los mismos P. Por ejemplo, si P es la ejecución de un método, *cflow(Pointcut)* captura todos los join points que ocurren dentro de la ejecución de este método, incluyendo la ejecución del método.

- **cflowbelow(Pointcut)** Captura todos los join points que están dentro del flujo de ejecución de los join points P capturados por *Pointcut*, excluyendo los join points P. Siguiendo el ejemplo anterior, si P es la ejecución de un método, *cflow(Pointcut)* captura todos los join points que ocurren dentro de la ejecución de este método, pero excluye la ejecución misma del método.

En el siguiente ejemplo, se desean capturan todas las llamadas que se producen dentro de la ejecución del pointcut definido por *setXY*. Esta captura se hace incluso cuando los join points que son capturados lanzan una excepción. Gracias a esto se puede saber si se producen llamadas internas que no corresponden con la lógica de los métodos que son capturados por el pointcut *setXY*

```
cflow(setXY(fe, x, y)) && call(* *.*(..))
```

2.1.2. Advice

En AspectJ existen tres tipos de advice, los cuales son definidos dependiendo de cuando se ejecutará el advice. Estos tipos son:

- **before** El advice se ejecuta antes de la ejecución del join point

- **after** El advice se ejecuta después de la ejecución del join point, además se puede elegir si ejecutar el advice cuando el join point retorno de forma normal o a través de una excepción

- **around** El advice se ejecuta en lugar de la ejecución del join point, además este advice puede decidir cuando será ejecutado o no el código original por medio de la expresión *proceed*

Los advice en AspectJ pueden ser definido por medio de pointcuts anónimos o *named pointcut*. Para graficar tipos de pointcuts se escribirán dos ejemplos. Si se define el pointcut *move()* como el pointcut que captura cuando una instancia de la clase *FigureElement* se mueve de la siguiente forma:

```
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));
```

Entonces se define el advice que escribirá en consola cuando el elemento está a punto de moverse de la siguiente forma:

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

También se define el advice que escribe en consola cuando el elemento se movió de forma exitosa de la siguiente forma:

```
after() returning: move() {
    System.out.println("just successfully moved");
}
```

La expresión *returning* se utiliza cuando la ejecución del join point capturado termina de forma normal. Para ejecutar el advice cuando el join point termina debido a una excepción se usa la expresión *throwing*.

Advice y exposición del contexto

Como se explicó, los pointcuts no solo capturan join points, sino también exponen el contexto de ejecución de los join points que capturan. Los valores que se exponen en el pointcut pueden ser usados en la firma del advice para así poder ocuparlos en el cuerpo del mismo. Por ejemplo si se toma el pointcut *setXY(FigureElement fe, int x, int y)* que se definió antes, se pueden usar las variables expuestas de la siguiente forma:

```
after(FigureElement fe, int x, int y) returning: setXY(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
}
```

Donde cada vez que se cambia la posición de *FigureElement*, se imprime después del cambio el elemento que se cambio y los puntos hacia donde se movió.

2.1.3. Inter-type Declaration

Los *inter-type declarations* en AspectJ se utilizan para agregar variables, métodos y constructores. Incluso, se puede cambiar la relación de herencia entre clases. Los cambios producidos por los *inter-type declarations* se verifican estáticamente, en tiempo de compilación.

Los *inter-type declarations* buscan, entre otras cosas, solucionar el problema de la limitación que se tiene de cambiar las clases que ya son parte del sistema de clases. En lugar de crear una nueva clase que extienda de la clase del sistema, se puede simplemente agregar funcionalidades a la clase del sistema en un único lugar, mediante *inter-type declarations*. Por ejemplo, se pueden agregar *observers* a una clase que no define el patrón *Observer*, como lo es la clase *Point*, de la siguiente forma:

```
private Vector Point.observers = new Vector();

public static void addObserver(Point p, Screen s) {
  p.observers.add(s);
}
public static void removeObserver(Point p, Screen s) {
  p.observers.remove(s);
}
```

En el ejemplo se agrega a la clase *Point* una variable de instancia llamada *observers*. También se le agregan los métodos necesarios para agregar y remover observers. Si a esto se le combinan los advices para hacer la actualización de los observers, se implementa el patrón *Observer* en una clase de sistema que no está definida con ese patrón, sin necesidad de cambiar el código de la clase *Point*.

2.1.4. Reglas de Precedencia

Cuando se declaran muchos aspectos, es probable que dos o más aspectos tengan advices que actúen sobre el mismo join point. Cuando esto sucede, es deseable decir, de alguna u otra forma, cuando se quiere ejecutar un advice de un aspecto antes o después que otro. Esto se logra con las reglas de precedencia que implementa AspectJ. Para especificar el orden de ejecución de los aspectos se utiliza la expresión:

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

Las reglas de precedencia depende de los tipos de advices que se ejecutan en el join point de la siguiente forma:

- **Before Advice** Se ejecutan los *before advices* del aspecto A y luego los de B
- **After Advice** Se ejecutan los *after advices* del aspecto B y luego los de A
- **Around Advice** Se ejecuta el *around advice* de A, luego si este tiene un llamado a *proceed*, se sigue ejecutando otro de A. Si siguen las llamadas a *proceed* hasta acabar los *around advice* de A, se comienza recién a llamar a los *around advice* de B.

AspectJ además verifica si la declaración de precedencia de los aspectos es correcta a momento de compilar. Una declaración es incorrecta cuando se declara un ciclo de precedencia, esto se puede ver en el siguiente ejemplo:

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

En el ejemplo se ve como se declara que el aspecto A se ejecute antes que B, luego se declara que B se ejecute antes que A, generando un conflicto. AspectJ ve todas las declaraciones de precedencia de todos los aspectos, y si se produce uno de estos conflictos no permite la compilación del programa. Pero se pueden declarar cosas como esta:

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

Lo que es perfectamente legal en AspectJ, a menos que A y B compartan un join point.

2.1.5. Aspectos

Los aspectos permiten combinar todos los pointcuts, advices e inter-type declarations en una unidad modular. Un aspecto tiene una sintaxis parecida a la de las clases de Java. Los aspectos además pueden tener métodos, campos e inicializadores. Un aspecto en AspectJ puede ser instanciado, pero no mediante la expresión *new* de Java. Por defecto, cada aspecto es un singleton, esto es, solo puede existir una instancia por aspecto.

Un aspecto en AspectJ se muestra en la figura 2.1.

```
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 2.1: Ejemplo de un aspecto en AspectJ

2.2. Eos-U

Eos-U [9] es un lenguaje orientado a aspectos desarrollado sobre el lenguaje *C#*. Eos-U nace buscando combinar el concepto de clases con el de aspectos, inventando el concepto de *classpect*. Esto lo hacen porque separar la implementación de clases con los aspectos puede hacer más difícil de entender y usar un programa orientado a aspectos en el largo plazo; y que la asimetría solo contribuye a dañar la modularidad de los programas. Gracias a esto, los aspectos en Eos-U son entidades de primera clase en el lenguaje.

2.2.1. Pointcut

El modelo de pointcuts es similar al de AspectJ. Esto es porque en Eos-U se basa en las buenas funcionalidades de AspectJ. Los pointcuts son especificados por medio de la palabra clave *pointcut*. Los pointcuts en Eos-U soportan el uso de patrones a lo AspectJ y la exposición del contexto, también a lo AspectJ.

2.2.2. Advice

En Eos-U no existe el concepto de advice. En cambio, se utiliza un método normal de una clase de *C#* que funciona como advice. Para que el método funcione como

advice se le debe indicar mediante operadores especiales. Además se tiene que indicar si el método funcionará como *before*, *after* y *around*. Por ejemplo, si existe declarado el método *Foo()* en la clase que funcionará como aspecto, se puede escribir el advice de la siguiente forma:

```
static int around execution(public int *.Bar()): call Foo();
```

También se le puede entregar al advice el código directamente, en lugar de indicarlo con un método, de la siguiente forma:

```
int around(): execution(public int *.Bar()){
      /* Advice Code Here */
}
```

2.2.3. Aspecto

Como se mencionó antes, un aspecto en Eos-U es simplemente una clase. Una clase funcionará como aspecto si existe declarado algún método como advice. En la figura 2.2 se ve una implementación de aspectos en Eos-U.

```
public class Aspect{
  void foo(){
    ...
  }

  static int after execution(public int *.Bar()): call foo();
}
```

Figura 2.2: Ejemplo de un aspecto en Eos-U

2.3. PHANtom

PHANtom [6] es un lenguaje nuevo desarrollado para Pharo Smalltalk. PHANtom combina las mejores funcionalidades de los lenguajes que le preceden e incorpora las nuevas investigaciones de control de reentrancia. La reentrancia es un problema común cuando se trabaja con un lenguaje orientado a aspectos. La reentrancia sucede cuando un aspecto captura un evento que es producido por un advice que se ejecuta en sí mismo. Una vez que un aspecto se captura a sí mismo, el programa termina en un ciclo infinito de ejecución.

Además, al igual que Smalltalk, PHANtom es diseñado para ser un lenguaje de aspecto dinámico. Ser dinámico significa que PHANtom puede cambiar los aspectos de forma dinámica, ya sea agregando, eliminando o incluso cambiando el orden de ejecución de los aspectos, todo esto en tiempo de ejecución del programa.

En PHANtom, el weaving de los aspectos se realiza en tiempo de ejecución con los llamados *method wrappers* [6]. En Smalltalk se pueden hacer cambios en el diccionario de métodos de una clase, agregando, eliminando y reemplazando cualquier método. Con los *Method Wrappers*, uno puede reemplazar métodos del diccionario por otros creados por el usuario de forma sencilla.

El modelo de join points que usa PHANtom consiste solamente en ejecuciones de métodos. Esto debido a que en Smalltalk, todo es un objeto (incluso las clases), y conceptualmente todo se maneja mandando mensajes hacia estos objetos.

2.3.1. Pointcut

Un pointcut en PHANtom es simplemente una instancia de la clase *PhPointcut*. La instancia se encarga de guardar la información necesaria para la captura de los join points. En el siguiente ejemplo se puede ver como un pointcut captura el mensaje *assert:* que se envía a una instancia de la clase *TestCase*.

```
PhPointcut receivers: 'TestCase' selectors: 'assert:'.
```

PHANtom tiene un sistema de patrones que permite capturar un conjunto de join points. Los patrones aceptan comodines diferentes tanto para la captura de clases como para la captura de métodos. En las tablas 2.2 y 2.3 se presentan los comodines aceptados por PHANtom.

Comodín	**Significado**
*	Captura cualquier secuencia de caracteres
+	Captura la clase y todas sus subclases

Tabla 2.2: Comodines para patrones de la captura de Clases en Phantom

Comodín	**Significado**
#()	Todos los selectores
_	Una *Keyword* de un método

Tabla 2.3: Comodines para patrones de la captura de Selectores en Phantom

Cabe destacar que el comodín "*" no captura las metaclases. Por ejemplo, el patrón *Test** en la sección de *receivers:* captura todas las clases que contengan la palabra *Test* en el comienzo de su nombre, pero no captura la clase *TestCase class*. Para capturar metaclases, se debe escribir explícitamente la palabra *class* después de un comodín o un nombre de clase. Por ejemplo, para capturar la clase *TestCase class*, se puede usar el patrón *Test* class*.

Además de un patrón, PHANtom también soporta una lista de patrones. En el siguiente ejemplo se capturan todas las posibles llamadas a los métodos de *asserts* que son usados por las pruebas unitarias en Smalltalk, pero acotados a las subclases de *ACTestCase*.

```
PhPointcut receivers: 'ACTestCase+'
           selectors: #('assert:' 'assert:_:' 'assert:_:_:')
```

Este pointcut captura las llamadas *assert* que reciben uno, dos y tres argumentos en las instancias de la clase *ACTestCase* y las instancias de todas las subclases.

Composición

En PHANtom la composición de pointcut se realizan mediante las operaciones booleanas **and** (&), **or** (|) y **not** (not). Todas estas expresiones se realizan como un llamado a un pointcut, recibiendo otro pointcut como argumento, finalmente, retornando un tercer pointcut que es la composición de ambos. En el siguiente ejemplo, se capturan todas las llamadas al método *position:* en la clase *Morph* y todas sus subclases, exceptuando los llamados de ese método en la clase *ImageMorph*, que es una subclase de *Morph*.

```
p1 := PhPointcut receivers: 'Morph+' selectors: 'position:'.
p2 := PhPointcut receivers: 'ImageMorph' selectors: 'position:'.
p3 := (p1 & (p2 not)).
```

Exposición del Contexto

La exposición del contexto en PHANtom se realiza junto con la declaración de pointcuts. La exposición de uno o más elementos del contexto permite ser usado después para los "pointcut condicionales" o en el advice. Los elementos que se pueden exponer son los siguientes:

- **receiver** Expone el objeto que recibe el mensaje
- **sender** Expone el objeto que envía el mensaje
- **selector** Expone el selector del mensaje
- **arguments** Expone los argumentos que son entregados al mensaje
- **proceed** Permite, para un *around advice*, ejecutar el comportamiento original
- **advice** Permite obtener y cambiar el orden de ejecución de los advice.

Estos elementos se pueden exponer dando como argumento una lista de lo que se desea exponer al método *context:* de la clase *PhPointcut*.

Volviendo al ejemplo de los métodos *asserts* acotados a las subclases de *ACTestCase*, si además se desea exponer el contexto del objeto que recibe el mensaje, se debe escribir:

```
PhPointcut receivers: 'ACTestCase+'
           selectors: #('assert:' 'assert:_:' 'assert:_:_:')
           context: #(receiver)
```

Restricción por Package

En Pharo Smalltalk, las clases se ordenan por unas entidades llamadas *Packages*. PHANtom puede restringir los join point que captura un pointcut por estos *packages* dando el mensaje *restrict:* a la clase *PhPointcut*. Siguiendo con el ejemplo anterior, si sabemos que solo queremos correr los test del package *Tests-Mine*, basta con escribir el pointcut:

```
PhPointcut receivers: 'TestCase+'
           selectors: #('assert:' 'assert:_:' 'assert:_:_:')
           context: #(receiver)
           restrict: #(Test-Mine)
```

Verificación Dinámica

En PHANtom se puede tener verificación dinámica del pointcut ya sea entregando una expresión booleana o por medio de un mensaje de flujo de ejecución del programa.

Los pointcuts que reciben una expresión booleana son llamados *guarded pointcuts*. Estos son generados simplemente mandando el mensaje *if:* a la clase *PhPointcut* con un bloque que retorna un valor booleano. El bloque recibe como argumento un objeto que representa la exposición del contexto en el pointcut. Cuando el join point que captura el pointcut de forma estática es ejecutado, se comprueba el bloque condicional para ver si el pointcut finalmente captura el join point en tiempo de ejecución. Por ejemplo, tomando el pointcut de la sección de restricción por package, se creará un pointcut que, dinámicamente, verifique si se encuentra en el package *Test-Mine*.

```
(PhPointcut receivers: 'TestCase+'
           selectors: #('assert:' 'assert:_:' 'assert:_:_:')
           context: #(receiver))
    if:[:ctx| ctx receiver category = #Test-Mine].
```

La verificación por el flujo de ejecución del programa se realiza por medio del mensaje *inCflowOf:*. Este mensaje recibe un pointcut, y retorna un tercer pointcut que verifica, en tiempo de ejecución, si captura o no el join point. El pointcut capturará el join point siempre y cuando se encuentre dentro de la ejecución del pointcut entregado como argumento. En el siguiente ejemplo se muestra como capturar los join points cuando una figura que se mueve, pero solo cuando se mueve por medio de una interfaz gráfica.

```
p1 := PhPointcut receiver: 'Figure+' selectors: 'move:'.
p2 := PhPointcut receiver: 'Window+' selectors: 'show'.
p3 := p1 inCflowOf: p2.
```

Parser Personalizados

PHANtom, además de entregar patrones predefinidos para la captura de join points, permite al programador definir sus propios patrones. Esto es posible gracias al uso de *PetitParser* [6]. *PetitParser* es un framework que permite generar parsers. Estos parsers se pasan como argumento ya sea para *receivers:* como para *selectors:*. Los parser generados puede obtener información completa de la clase para capturar otros join points. Por ejemplo, el siguiente pointcut captura los mensajes *count* y *count:* de las instancias de las clases que tienen una variable de instancia llamada *count*:

```
PhPointcut
  receivers: (#any asParser
              plusGreedy: 'instanceVariableNames: ''count''' asParser)
  selectors: ('count' asParser , ':' asParser optional).
```

2.3.2. Advice

En PHANtom, un advice es una instancia de la clase *PhAdvice*. Esta instancia posee el pointcut que captura los join points donde se ejecutará el advice, el tipo de advice y una acción. PHANtom posee tres tipos de advices y dos tipos de acciones diferentes.

Los tres tipos de advice en PHANtom son: *before*, *after* y *around*. Cada uno con su respectiva palabra clave *before:*, *after:* y *around:*. Cada uno de estas palabras claves reciben como argumento un pointcut. Los tipos de advice de PHANtom funcionan de la misma forma que los tipos de advices en AspectJ.

En PHANtom se aceptan dos tipos de acciones, un bloque o un mensaje a un objeto. Para ambos tipos de acciones, dependerá del tipo de advice si se ejecuta antes, después o en lugar del código original. Cuando se pasa un bloque al advice, este bloque tiene un argumento, el contexto. Por ejemplo, si existe un pointcut *pc*, se puede crear un advice de la siguiente forma:

```
PhAdvice before: pc advice:[:ctx| Transcript show: 'before execution'; cr].
```

El segundo tipo de acción de un advice permite mandar un mensaje a un objeto. Al igual que el primer tipo de acción, este mensaje debe recibir un argumento, el contexto que fue expuesto. Adentro, en el cuerpo del mensaje, se pueden hacer las llamadas al contexto que se deseen, siempre y cuando este sea expuesto en el pointcut. Por ejemplo, si se tiene un pointcut *pc*, un objeto *printer*, el cual acepta el mensaje *printAfter:*, se puede escribir un advice de este tipo como:

```
PhAdvice after: pc send: #printAfter: to: printer.
```

Lo que hará será enviar, después del join point capturado por pc, el mensaje *printAfter:* al objeto *printer*.

2.3.3. Modificadores de Clase

En PHANtom se puede modificar la estructuras de las clases del sistema. Funcionan de forma parecida a los *inter-type declarations* de AspectJ. En PHANtom uno puede modificar las clases agregando variables y métodos de instancias. Además, también se pueden agregar variables y métodos en el lado de la clase. Las clases que se modifican son aquellas que son capturadas por el pointcut que se entrega como parámetro a un modificador de clase. Los modificadores de clase son instancias de la clase *PhClassModifier*. Esta clase soporta los siguientes mensajes:

- **addNewInstanceVar:** recibe el nombre de una variable de instancia que será agregada
- **addNewInstanceMethod:** recibe un método, pero como string, que será el código fuente de un nuevo método de instancia
- **addNewClassVar:** funciona igual que *addNewInstanceVar:*, solo que agrega la variable al lado de la clase
- **addNewClassMethod:** funciona igual que *addNewInstanceMethod:*, solo que agrega el método en el lado de la clase

Por ejemplo, suponiendo que existe un pointcut *pc*, se quiere agregar la variable *phacount*, y los métodos que servirán para acceder a esta variable de instancia. Esto se hace de la siguiente forma:

```
cm1 := PhClassModifier on: pc addIV: 'phacount'.
cm2 := PhClassModifier on: pc addIM:
  'phacount
    ^phacount ifNil:[phacount := 0]'.
cm3 := PhClassModifier on: pc addIM:
  'phacount: anObject
    phacount := anObject'.
```

2.3.4. Reglas de Precedencia

De la misma forma que en AspectJ, en PHANtom es posible declarar el orden en los que son ejecutados los advices de diferentes aspectos. Las reglas de precedencia son iguales a las de AspectJ, mencionadas en la sección 2.1.4.

En PHANtom, se pueden declarar tres tipos de precedencia:

- **PhAspect>>precedence:** Agrega orden de aspectos a nivel global
- **PhPointcut>>precedence:** Agrega orden de aspectos, pero a nivel del pointcut
- **PhPointcut>>overridePrecedence:** Ignora cualquier orden global y solo toma el orden local

2.3.5. Aspectos

Los aspectos combinan todos los cambios que pueden ser realizados tanto por los modificadores de clase como por los advices. Un aspecto es una instancia de la clase *PhAspect*. A esta instancia se le pueden agregar advices mediante el mensaje *add:* y se le pueden agregar modificadores de clase mediante el mensaje *addClassModifier:*. Un aspecto no basta con declararlo, para que funcione debe ser instalado en el sistema, para ello se le manda el mensaje *install* a la instancia del aspecto que se desea instalar. Si ya no se requiere más un aspecto, este puede ser desinstalado con el mensaje *uninstall*.

PHANtom mantiene un control de los aspectos instalados en la clase *PhAspectWeaver*. Para ver los aspectos instalados en el sistema basta ejecutar el código:

```
PhAspectWeaver installedAspects.
```

Lo que retorna una lista de los aspectos instalados en el sistema. Además se pueden desinstalar todos los aspectos que se han instalado mediante la ejecución del código:

```
PhAspectWeaver uninstallAll.
```

2.3.6. Membranas Computacionales

Las membranas computacionales se crean como un mecanismo para acotar el alcance de los join points emitidos en la computación [10]. Las membranas computacionales son desplegadas alrededor de alguna computación, capturando uno o más join points. La idea que hay por detrás es que los aspectos, al ser instalados en el sistema, solo pueden ver la computación en la membrana donde es instalado.

PHANtom utiliza el potencial de las membranas computacionales para controlar los problemas de reentrancia que aparecen con la utilización de aspectos. En el código de la figura 2.3 se ve un ejemplo clásico de problemas de reentrancia. El pointcut *pc* captura la llamada que se ejecuta cuando se imprime en el *Transcript*. El advice *adv* tiene como acción una llamada al método *show:* de *Transcript*, esto es, otra llamada que imprime algo en el *Transcript*. Luego, el aspecto *asp*, que contiene el advice *adv* es instalado en el sistema. Finalmente se ejecuta una llamada para imprimir un texto en el *Transcript*. Cuando se ejecuta la última llamada, primero es ejecutado el advice *adv*, pero el advice tiene una llamada que es capturada por el mismo aspecto *asp*, llevando a un ciclo infinito de llamadas al advice *adv*.

PHANtom se encarga de este problema instalando una membrana por cada aspecto que instala. Por lo mismo, el aspecto es ejecutado en su propia membrana, luego, el join point generado al ejecutar el advice no es capturado por el aspecto. Además en PHANtom es posible definir manualmente estas membranas e instalarlas en el sistema. Cuando se desea que un aspecto vea solo la ejecución que se produce en una membrana, el aspecto es instalado en la membrana que se crea.

```
pc  := PhPointcut
          receivers: 'TranscriptModel'
          selectors: 'show:'.
adv := PhAdvice
          before: pc
          advice: [:ctx| Transcript show: 'Before Advice... '].
asp := (PhAspect new) add: adv.
asp install.
Transcript show: 'Reentrancy '; cr.
```

Figura 2.3: Ejemplo de reentrancia en PHANtom. Gracias a las membranas no se produce ejecución infinita del advice

Capítulo 3

Casper: Descripción del Lenguaje

Esta sección describe el lenguaje de aspectos desarrollado, Casper. Casper es un lenguaje desarrollado para Pharo Smalltalk[1] y se basa en otro lenguaje para esta plataforma, PHANtom [6]. El modelo de join points usado en Casper es el mismo que en PHANtom. Cada mensaje es recibido por un objeto es un join point. Además, la sintaxis es muy parecida en ambos lenguajes, intentando incluso que las clases importantes del sistema tengan los mismos nombres.

En la sección 3.1 se describen los pointcuts, sus características y como exponer el contexto para los advices. La sección 3.2 se trata el tema de los advices y como conectarlos con un pointcut. En la sección 3.2.3 se describe la exposición del contexto, su significado y como usarlo en un advice. En la sección 3.3 se explica como funcionan los modificadores de clase en Casper y como crearlos. En la sección 3.4 nos referiremos a los aspectos en si y como poder guardar diferentes configuraciones de advices y modificadores de clases en los aspectos.

La particularidad de Casper sobre los demás lenguajes orientados a aspectos se encuentra en su explicitud. Al momento de instalar una nueva funcionalidad en el sistema ésta aparece explícitamente en el ambiente de Pharo con un nombre especial. Así el programador puede estar seguro que realiza los cambios que estaba buscando. Junto con ello el código mismo se compila. Esto se hace para lograr la explicitud y para hacer que el tiempo del programa a nivel de ejecución sea menor, ya que se carga este tiempo a nivel de compilación.

Aunque PHANtom y Casper tienen sintaxis y funcionalidades muy parecidas, ambos difieren fundamentalmente en la implementación. PHANtom posee otras funcionalidades, como parser personalizados y la implementación de membranas para controlar el alcance que tienen los aspectos en el sistemas. Casper busca la simplicidad de la implementación y la velocidad a nivel de ejecución del programa.
Además Casper reestructura la forma en que se manejan los modificadores de clases. Mientras PHANtom usa strings para agregar nuevos métodos, Casper usa el ambiente de Pharo para escribir los métodos. Así se aprovecha el manejo de errores a nivel de

[1] `http://www.pharo-project.org`

Comodín	Significado
*	Especifica cualquier secuencia de caracteres válida
+	Captura la clase y todas sus subclases.

Tabla 3.1: Comodines para patrones de la captura de Clases en Casper

escritura del código mismo y al nivel de compilación.

3.1. Pointcuts

Como ya se mencionó, un pointcut es la definición de una captura de un join point o un conjunto de estos. En las siguiente secciones se explica el uso y se dan ejemplos de los pointcuts de Casper.

3.1.1. Uso Básico

Al igual que PHANtom, en Casper se define un pointcut cuando se le da el receptor del mensaje y el mensaje mismo que se desea capturar. En Casper esto se hace enviando mensaje *receivers:selectors:* a la clase *PhPointcut*, tomando como argumento el nombre del receptor del mensaje y el mensaje mismo. Al recibir este mensaje, la clase *PhPointcut* retorna una instancia de sí misma con la información recibida.

En el siguiente ejemplo, capturamos el join point que se genera al enviar el mensaje *stop* a la clase *Vehicle*

```
PhPointcut receivers:'Vehicle' selectors:'stop'
```

Casper se inspira en PHANtom que, a su vez, se inspira en AspectJ para usar patrones que especifican join points que se quieren capturar. Para el receptor del mensaje los patrones pueden usar los comodines "*" y "+". El comodín "*" se usa en Casper para especificar cualquier secuencia de caracteres válida en el nombre de una clase y el comodín "+" se pone al final del patrón para especificar las clases que captura el patrón, además de todas sus subclases.

Por ejemplo, si se desea capturar todos los join point que envían el mensaje *speed:* a una instancia que contenga *Vehicle* y a todas sus subclases, se escribe:

```
PhPointcut receivers:'*Vehicle*+' selectors:'speed:'
```

Todos los comodines se resumen en la tabla 3.1.

Al igual que los receptores del mensaje, el mensaje mismo también acepta patrones para especificar conjunto de join points. Los comodines que aceptan los patrones de los mensajes se basan en los comodines de PHANtom. Actualmente el patrón de los

Comodín	Significado
_	Especifica todos los mensajes que no reciben argumentos
_:	Especifica todos los mensajes que reciben argumentos
#()	Especifica todos los mensajes

Tabla 3.2: Comodines para patrones de la captura de Selectores en Casper

selectores acepta el comodín "_" que se usa para reemplazar el nombre de un mensaje para que capture cualquier nombre. Por ejemplo, _ captura todos los mensajes que no reciben argumentos, *_:* captura todos los mensajes que reciben 1 argumento, *_:_:* captura todos los mensajes que reciben 2 argumentos, y así. Además, se pueden hacer combinaciones del estilo *foo:_:*. Este mensaje captura todos los métodos de dos argumentos, donde el nombre del mensaje que recibe el primer argumento es *foo*, y el segundo puede ser cualquier nombre.
Sin embargo, "_" no funciona como "*" para los receptores del mensaje. Esto es, "_" no se usa para especificar cualquier secuencia válida de caracteres. Por ejemplo, _ captura todos los mensajes que no reciben argumento, *get_* solo captura el mensaje *get_* y no el mensaje *getPoint*. Esto pasa porque el comodín "_" funciona cuando se ocupa sin ningún otro caracter en su nombre. Finalmente, se pueden capturar todos los mensajes de un receptor dando como argumento a *selectors:* un arreglo vacío, *#()*

En el siguiente ejemplo se capturarán todos los mensajes de un argumento que son enviados a una instancia de la clase *Vehicle*

```
PhPointcut receivers:'Vehicle' selectors:'_:'
```

Los comodines para los mensajes se resumen en la tabla 3.2

En PHANtom se pueden pasar múltiples patrones tanto para los receptores del mensaje como para el mismo mensaje. Esta es una funcionalidad importante que se mantiene en Casper. Para ambos casos solo se debe entregar una lista de patrones. Cuando se entrega una lista de patrones, el pointcut captura todos los mensajes que se han especificado en la lista de patrones de *selectors:* que son recibidos por alguna clase que se haya capturado en la lista de patrones en *receivers:* . Por ejemplo, si queremos capturar los mensajes *run* y *stop* de las instancias de las clases *Motocycle* y *Car* se debe escribir el siguiente pointcut.

```
PhPointcut receivers:#('Motocycle' 'Car') selectors:#('run' 'stop')
```

Si un selector es capturado por un patrón en Casper, entonces se captura el mensaje que entiende la clase, esto significa, que el patrón captura el selector incluso cuando estos son definidos en alguna clase padre. Para capturar solamente los selectores que define la clase que se captura, se puede usar el mensaje *localSelectors:*. Por ejemplo si la clase *Car* extiende de la clase *Vehicle*, puedo escribir el siguiente pointcut para capturar todos los selectores que entiende una instancia de la clase *Car*, pero que estén definidos solamente en la clase *Car*:

```
PhPointcut receivers: 'Car' localSelectors:'*'
```

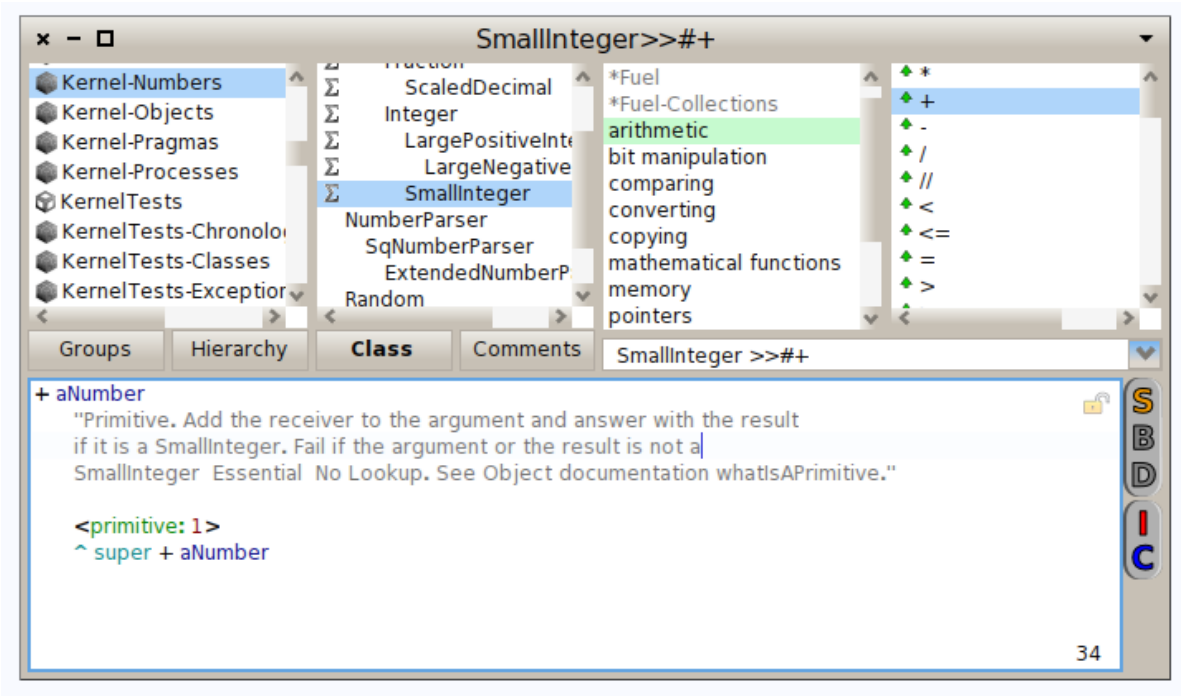

Figura 3.1: Método para sumar de la clase SmallInteger

3.1.2. Pragmas

En Smalltalk, los pragmas se usan para marcar algunos métodos del sistema con un *tag* especial, como las anotaciones en Java[2]. Estos tags pueden ser utilizados para expresar funcionamiento especial del método marcado. Por ejemplo, en el método para sumar números se utiliza un pragma para decir que es una función primitiva del sistema, como se ve en la imagen 3.1. En Casper se pueden capturar join points que estén marcados con pragmas, siendo una funcionalidad incorporada en Casper que no se encuentra presente en PHANtom.

Para capturar un join point que posea una anotación se envía el mensaje *receivers:selectors:pragmas:* a la clase *PhPointcut*. Los argumentos de *receivers:* y *selectors:* son los mismos mencionados anteriormente. Los pragmas en Smalltalk tienen la misma estructura que un selector, solo que los argumentos que recibe no pueden ser variables. Por esto, en Casper el patrón que captura los pragmas es muy parecido al de los selectores, solo que también se pueden especificar el argumento que recibe de forma explícita. En el siguiente ejemplo, capturamos el mensaje que pasa un número entero a uno flotante usando una primitiva en Smalltalk.

```
PhPointcut receivers:'SmallInteger' selectors:#() pragmas:'primitive:40'
```

Si se desea capturar un pragma que recibe como argumento un string se debe utilizar dos comillas (") cuando se declara el string. Esto debido a que en Smalltalk una comilla dentro del string cerraría la primera comilla del string. Para evitarlo, se utilizan do

[2] `http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html`

comillas. Por ejemplo, si se desea capturar el pragma *foo:* con argumento *'kaz'* se debe escribir: *pragma: 'foo: "kaz"'*.

Para especificar patrones para los pragmas, usamos el comodín "_", tanto para el nombre del mensaje como para el argumento, por ejemplo, para capturar todas las primitivas usamos el patrón *primitive:_* y para capturar todos los pragmas que no reciben argumentos usamos _. También se pueden entregar una lista de patrones para capturar un conjunto de pragmas, por ejemplo, si queremos capturar las primitivas para sumar y para convertir a flotante un número usamos el patrón *#('primitive:1' 'primitive:40')*.

3.1.3. Restricción por Package

En Smalltalk, un Package es un contenedor de clases. Los Package se usan para llevar un orden de los proyectos y programas que se instalan en este lenguaje, por ejemplo, Casper esta contenido dentro del Package *Casper-Core*. Casper toma el ejemplo de PHANtom para restringir por packages los join points donde pueden capturar los pointcuts. Para esto, se le envía el mensaje *receivers:selectors:restrict:* a la clase *PhPointcut*, donde el mensaje que recibe *restrict:* es una lista con el nombre de todos los package que se desean restringir en la captura de join points.

En el siguiente ejemplo, se capturarán todos los mensajes de las clases que extiendan de *Test*, pero solo dentro del *Package* donde se encuentran los tests de Casper.

```
PhPointcut receivers:'TestCase+' selectors:#() restrict: #('Casper-Test')
```

3.1.4. Exposición del Contexto

Casper toma el ejemplo de PHANtom para lograr exponer el contexto de los join points que captura con la definición de los pointcuts. Así los advices que se ejecuten en los join points capturados por el pointcut dado pueden usar más información. Por defecto no se saca ninguna información del contexto para evitar hacer trabajo innecesario. La información que se puede obtener es la siguiente:

- #**receiver** expone el objeto receptor del mensaje
- #**sender** expone el objeto que envió el mensaje
- #**selector** expone el selector que representa el mensaje capturado
- #**arguments** expone los argumentos que recibe el mensaje capturado
- #**thisAdvice** expone el advice que esta siendo ejecutado en la captura del mensaje
- #**proceed** expone la siguiente ejecución para los advices del tipo *around*

Para exponer uno o más de estos contextos, se debe entregar el mensaje *context:* a una instancia de la clase *PhPointcut*. Lo que recibe este mensaje es un arreglo con

uno o más de los símbolos listados anteriormente. Por ejemplo, para exponer el objeto receptor y los argumentos, debemos mandar el mensaje *context:#(receiver arguments)* a una instancia de la clase *PhPointcut*.

En PHANtom además uno puede exponer todos los advices que ejecutan en un join point. Esto puede ser usado para realizar operaciones sobre los advices, incluso cambiándolos de orden de forma dinámica. En cambio, en Casper se decidió ir por la simplicidad, tanto en el lenguaje como en su uso para el programador. Por eso en Casper solo se expone un advice a la vez. Este advice es aquel que se está ejecutando en el join point y que el código es especificado por el advice.

3.1.5. Composición de Pointcuts

En Casper también es posible componer pointcuts. La sintaxis es la misma que en PHANtom. La Composición de un pointcut crea un nuevo pointcut. Para componer pointcuts se usan los operadores booleanos "or" ||, "and" & y "not" *not*, que se definen de forma natural como:

- **pc1 || pc2** El pointcut generado captura tanto lo que captura pc1 como lo que captura pc2
- **pc1 & pc2** El pointcut generado captura todo lo que captura pc1 y pc2 juntos.
- **pc1 not** El pointcut generado captura todo lo que no captura pc1.

Si se tiene una clase *Vehicle* de la cual heredan muchas clases que representan vehículos, en el siguiente ejemplo se capturará todos los mensajes que no reciben argumentos de la clase *Vehicle* y sus subclases, exceptuando los mensajes de la clase *Car* (que hereda de vehículo) que no reciben argumentos.

```
pc1 := PhPointcut receivers:'Vehicle+' selectors:#().
pc2 := PhPointcut receivers:'Car' selectors:'_'.
pc3 := (pc1 & (pc2 not)).
```

3.1.6. Pointcuts Condicionales

Una característica avanzada de los lenguajes orientados a aspectos es permitir o negar la ejecución de advices bajo un criterio que se evalúa en tiempo de ejecución. Casper se basa en PHANtom para exponer dos formas de hacer pointcuts condicionales. Estas dos formas son: con una condición explicita que se entrega a una instancia de la clase *PhPointcut* por el mensaje *if:* o por medio de una operación de pointcuts llamada *cflow*. La llamada *cflow* genera un nuevo pointcut. Ambas condiciones no son excluyentes, esto es, un *cflow pointcut* puede tener también una condición del tipo *if:*.

El mensaje *if:* de un pointcut recibe un bloque que acepta como argumento el contexto (explicado en la sección3.1.4) y entrega un valor booleano. Este valor determina,

en tiempo de ejecución, si el advice que esta definido en este pointcut se ejecutará o no. En el siguiente ejemplo, se capturarán todos los mensajes de la clase *Vehicle*, siempre y cuando el objeto que envía el mensaje se encuentra en un *package* distinto al objeto que recibe el mensaje.

```
pc := PhPointcut receivers:'Vehicle'
                 selectors:#()
                 context:#(receiver sender).
pc if:[:ctx| ctx receiver class category = ctx sender class category]
```

La operación *cflow* esta soportada por Casper de la forma: *pc1 inCflowOf: pc2*. Lo que significa que, en tiempo de ejecución, el advice que esta definido en el pointcut pc1 se ejecutará solo si la ejecución del programa se encuentra dentro del flujo de control del pointcut pc2. En el siguiente ejemplo, se capturará los join points de los autos que compiten en una carrera. Esto se representará capturando el mensaje *run* de las instancias de la clase *Car*, siempre y cuando estén dentro del flujo de ejecución del mensaje *start* de una instancia de la clase *Race*.

```
pc1 := PhPointcut receivers:'Car' selectors:'run'.
pc2 := PhPointcut receivers:'Race' selectors:'start'.
pc3 := pc1 inCflowOf: pc2.
```

3.1.7. Restricciones

Debido a la forma de instalar aspectos en Casper, se puede generar problemas en el sistema, incluso dejándolo inutilizable. Para evitar esto, los pointcuts por defecto no capturan dos tipos de métodos. Los primeros, son métodos que están definido en una clase dentro de los packages que comiencen por *Kernel* o *Collections*. Los segundos, son métodos que no tengan un pragma del tipo *primitive:*. Estos dos tipos de métodos son importantes en el sistema, ya que Smalltalk los ocupa para compilar los métodos o realizar operaciones especiales con ellos. Como Casper recompila los métodos que son capturados por los pointcut, el tratar de recompilar un método que puede ser utilizado por el compilador es una operación peligrosa para el sistema. En Casper esto se puede traducir en fallas en el programa, incluso congelando el programa.

Aun así, el programador puede saltar estas restricciones si está seguro de que no causará problemas en el sistema. Para evitar la primera restricción hay que mandar el mensaje *noPackages:* y dar como argumento un arreglo vacío. También le puede dar cualquier arreglo para evitar que los pointcuts capturen otros packages. Por ejemplo, si se desea que por ningún motivo el pointcut debe capturar un mensaje dentro de los packages que comiencen con *AST*, se debe mandar el mensaje *noPackages:* y dar como argumento *#('AST')*.

Para saltar la segunda restricción, solo basta con mandar el mensaje *withoutPrimitives:* al pointcut, con argumento *false*.

3.1.8. API Pointcut

Los pointcuts en Casper pueden tener la información que se presenta en la tabla 3.3. La API de la clase *PhPointcut* para la creación de un pointcut es la siguiente:

- *#receivers:*
- *#receivers:restrict:*
- *#receivers:selectors:*
- *#receivers:selectors:context:*
- *#receivers:selectors:context:if:*
- *#receivers:selectors:context:if:restrict:*
- *#receivers:selectors:context:restrict:*
- *#receivers:selectors:restrict:*
- *#receivers:selectors:pragmas:*
- *#receivers:selectors:pragmas:context:*
- *#receivers:selectors:pragmas:context:if:*
- *#receivers:selectors:pragmas:context:if:restrict:*
- *#receivers:selectors:pragmas:context:restrict:*
- *#receivers:selectors:pragmas:restrict:*
- *#receivers:localSelectors:*
- *#receivers:localSelectors:context:*
- *#receivers:localSelectors:context:if:*
- *#receivers:localSelectors:context:if:restrict:*
- *#receivers:localSelectors:context:restrict:*
- *#receivers:localSelectors:restrict:*
- *#receivers:localSelectors:pragmas:*
- *#receivers:localSelectors:pragmas:context:*
- *#receivers:localSelectors:pragmas:context:if:*
- *#receivers:localSelectors:pragmas:context:if:restrict:*
- *#receivers:localSelectors:pragmas:context:restrict:*
- *#receivers:localSelectors:pragmas:restrict:*

3.2. Advices

Un advice es la acción que se ejecutará en los join points que captura el pointcut. En Casper, un advice es una instancia de la clase *PhAdvice*. Una instancia de la clase *PhAdvice* tiene que tener como información el pointcut, la acción misma y si se ejecutará antes de la acción original, después o reemplazando la acción original.

Selector	Descripción
#receivers:	Especifica un patrón (con un string) o varios patrones (lista de strings) para capturar nombres de clases
#selectors:	Especifica un patrón (con un string) o varios patrones (lista de strings) para capturar nombres de métodos
#pragmas:	Especifica un patrón (con un string) o varios patrones (lista de strings) para capturar pragmas
#context:	Especifica una lista de strings que representan la exposición del contexto
#restrict:	Especifica una lista de strings con los packages donde se podrán capturar las clases
#receivers:	Especifica un patrón (con un string) o varios patrones (lista de strings) para capturar nombres de clases
#if:	A través de un bloque que retorna un argumento booleano, especifica, en tiempo de ejecución, si el pointcut se ejecutará o no

Tabla 3.3: Resumen de la información de PhPointcut

3.2.1. Tipos de Advices

Como en AspectJ y en PHANtom, en Casper existen tres tipos de advices:

- #**before** Se usa para especificar que la acción del advice se ejecutará antes que la acción original capturada por el pointcut
- #**after** Se usa para especificar que la acción del advice se ejecutará después que la acción original capturada por el pointcut
- #**around** Se usa para especificar que la acción del advice reemplazará a la acción original capturada por el pointcut

En Casper, para cada uno de los tipos, a la clase *PhAdvice* se le puede mandar uno de los siguientes mensajes: *before:advice:*, *after:advice* y *around:advice*. Los argumentos de cada uno de estos mensajes son un pointcut y una acción representada por un bloque (del cual hablaremos en detalle en la siguiente sección).

3.2.2. Acción de Advices

Al igual que PHANtom, Casper tiene dos tipos de acciones que puede realizar un advice.

El primer tipo acción de un advice es un bloque de un argumento. A una instancia de la clase *PhPointcut* se le manda el mensaje *advice:* con el bloque como argumento. El argumento que recibe el bloque representa la exposición del contexto a través del pointcut. En Smalltalk los bloques capturan las variables del contexto donde son declaradas, por ejemplo si el bloque posee una variable x que esta definida fuera del bloque, al momento de declarar el bloque también captura el valor de la variable x. Al buscar

expresividad y simpleza en Casper, los bloques que son pasados al mensaje *advice:* no capturan el contexto de Smalltalk en ese momento.

Para mostrar este funcionamiento se creará un advice que mostrará cuando un auto cambia de velocidad. Esto se representará capturando el mensaje *speed:* a una instancia de la clase *Car*. El mensaje *speed:* recibe la nueva velocidad del auto como argumento. Para el comportamiento se creara una instancia de la clase *PhAdvice* donde, con el mensaje *advice:*, se le entregará un bloque con la lógica para imprimir que la velocidad ha cambiado. Este advice se ejecutará antes de que el auto cambie de velocidad.

```
pc := PhPointcut receivers:'Car' selectors:'speed:'.
adv := PhAdvice
            before:pc
            advice:[:ctx| Transcript show: 'Changing speed'].
```

El segundo tipo de acción de un advice se centra en el uso de la instancia de otra clase y un mensaje hacia esta instancia. A una instancia de la clase *PhAdvice* se le puede entregar el mensaje *send:to:* que representa este tipo de acción del advice. El primer argumento de este mensaje es un selector y el segundo argumento puede ser una instancia de otro objeto o un bloque que retorna un objeto. Cuando el advice ejecuta esta acción, lo que hace es mandar el mensaje representado por el selector que se mando como primer argumento, al objeto que fue entregado como segundo argumento del mensaje *send:to:*. Cuando el segundo argumento es un bloque, lo primero que se hace es evaluarlo para que se convierta en un objeto, luego se maneja como si originalmente se haya recibido un objeto como segundo argumento. El mensaje enviado es un método que recibe como argumento la representación de la exposición del contexto a través del pointcut.

El ejemplo anterior se modificará para usar esta otra forma de tener acciones en un advice. Si se supone que se tiene un objeto *printer* que acepta el mensaje *speedChange:*, entonces se puede modificar el ejemplo anterior de la siguiente forma:

```
pc := PhPointcut receivers:'Car' selectors:'speed:'.
adv := PhAdvice before:pc send:#speedChange: to:printer.
```

Donde el método *speedChange:* imprime que la velocidad ha cambiado.

3.2.3. Exposición del Contexto

Cuando un pointcut expone el contexto con los símbolos mostrados antes, estos se pueden ocupar en la acción del advice. La sintaxis para la exposición del contexto es tomada de PHANtom. Para poder exponer el contexto, este es representado por una instancia de la clase *PhContext*. Cuando se pide exponer alguna información del contexto, esta se vuelve disponible en la instancia de *PhContext*. Para ilustrar esto veamos el siguiente ejemplo:

```
pc := PhPointcut receivers:'Vehicle'
                 selectors:'run'
                 context:#(selector).
adv := PhAdvice
            before:pc
            advice:[:ctx| Transcript show: ctx selector].
```

Como en el pointcut del ejemplo se expone al contexto el selector que representa el mensaje donde captura el pointcut, podemos usar en el advice la línea de código *ctx selector*, que entregará el selector del mensaje para que en el advice se utilice como se desee, en este caso, imprimir el selector.

3.2.4. API de Advice

En la tabla 3.4 se presenta un resumen de los mensajes que puede recibir la clase *PhAdvice* y su funcionalidad

Selector	Descripción
#before:advice:	Recibe un pointcut y un bloque. El bloque se ejecutará antes de la ejecución original del mensaje especificado en el pointcut
#before:send:to:	Recibe un pointcut, un mensaje (representado en un selector) y un objeto. El objeto ejecutará el mensaje especificado antes de la ejecución del mensaje original especificado en el pointcut
#after:advice:	Recibe un pointcut y un bloque. El bloque se ejecutará después de la ejecución original del mensaje especificado en el pointcut
#after:send:to:	Recibe un pointcut, un mensaje (representado en un selector) y un objeto. El objeto ejecutará el mensaje especificado después de la ejecución del mensaje original especificado en el pointcut
#around:advice:	Recibe un pointcut y un bloque. El bloque se ejecutará en lugar de la ejecución original del mensaje especificado en el pointcut
#around:send:to:	Recibe un pointcut, un mensaje (representado en un selector) y un objeto. El objeto ejecutará el mensaje especificado en lugar de la ejecución del mensaje original especificado en el pointcut

Tabla 3.4: Resumen API de PhAdvice

3.3. Modificadores de Clase

En Casper es posible agregar nuevo comportamiento a las clases del sistema. Casper se basa en la funcionalidad de PHANtom para realizar esto, el cual se basa en lo que se puede realizar con los Inter-type declarations de AspectJ. En Casper la forma de crear modificadores de clases es un poco más compleja que en PHANtom, pero esta complejidad es recompensada con el uso de Smalltalk para comprobar la sintaxis de los métodos que se quieran agregar. No es así en PHANtom, donde esto se realiza

entregando un String con un formato de método, siendo esta última opción muy poco amigable con el programador.

Para usar los modificadores de clase se debe crear una clase que herede de *PhClassModifier*. Las instancias de estas clases pueden recibir los siguientes mensajes:

- **pointcut:** específica las clases que serán alteradas, las cuales son capturadas por un pointcut.
- **addNewInstanceVar:** específica el nombre de una nueva variable de instancia
- **addNewInstanceMethod:** específica el nombre de un método de instancia nuevo, el cual debe ser definido en la misma clase
- **addNewClassVar:** específica el nombre de una nueva variable de clase
- **addNewClassMethod:** específica el nombre de un método de clase nuevo, el cual debe ser definido en la misma clase

Los nombres de las nuevas variables y métodos se deben especificar como un símbolo. Si la modificación consiste en agregar un nuevo método (de instancia o de clase), se debe crear una nueva clase que herede de *PhClassModifier* y que tenga definido los nuevos métodos a agregar.

En el siguiente ejemplo, se definirá un modificador de clase que funcionará como una extensión de funciones matemáticas para la clase *Number*. Para realizar esto se crea la clase *PhNumberModifier* que extiende de *PhClassModifier* (figura 3.2) y se definirá en esta clase el método *fibonacci* (figura 3.3). Luego se agregará esta modificación a la clase *Number* de la siguiente manera:

```
pc := PhPointcut receivers:'Number'.
numModifier := (PhNumberModifier new)
                  pointcut: pc;
                  addNewInstanceMethod: #fibonacci.
```

```
PhClassModifier subclass: #PhNumberModifier
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Casper-Test'
```

Figura 3.2: Definición de la clase PhNumberModifier

```
PhNumberModifier>>fibonacci
  self == 0 ifTrue:[^0].
  self == 1 ifTrue:[^1].

  ^ ((self - 1) fibonacci) + ((self - 2) fibonacci).
```

Figura 3.3: Definición del método fibonacci de la clase PhNumberModifier

Con esto se pueden seguir definiendo más funciones importantes para la clase *Number*.

3.4. Aspectos

En Casper, un aspecto es una instancia de la clase *PhAspect*. Un aspecto posee toda la información necesaria para instalar en el sistema advices y modificadores de clase. Un advice o un modificador de clase por si solo no hacen nada, es el aspecto quien logra reunir los datos para luego instalarlos en el sistema para su funcionamiento.
Una instancia de *PhAspect* acepta los siguientes mensajes:

- **add:** permite agregar un nuevo advice representado por una instancia de la clase *PhAdvice*
- **addClassModifier:** permite agregar un nuevo modificador de clase representado por una instancia de *PhClassModifier* o alguna subclase.
- **install** permite instalar el aspecto en el sistema
- **uninstall** permite desinstalar el aspecto revirtiendo todos los cambios en el sistema.

Una vez que se hayan agregado advices y/o modificadores de clases, el aspecto puede ser instalado en el sistema para activar el comportamiento y desinstalarlo para desactivarlo. Además si se modifican las clases del sistema una vez instalados el aspectos, estos nuevos join points también serán capturados si corresponde.

En el siguiente ejemplo se ilustra como se instala un aspecto. Primero se crea un pointcut que captura todos los mensajes que no reciben argumentos de la clase *Mock*. Luego se definen dos acciones con la forma de dos advices distintos. Los advices solo imprimen antes y después de la ejecución del método original. Finalmente todo esto se guarda en un aspecto que se instala en el sistema.

```
pc := PhPointcut receivers:'Mock' selectors: '_'.
advBefore := PhAdvice
                before:pc
                advice:[:ctx| Transcript show: 'Before execution'].
advAfter := PhAdvice
               after:pc
               advice:[:ctx| Transcript show: 'After execution'].
asp := (PhAspect new)
          add: advBefore;
          add: advAfter.
asp install.
```

3.4.1. Múltiples Advices y Modificadores de Clases

Los aspectos son un contenedor de advices y modificadores de clases, los cuales son agregados al sistema cuando el aspecto se instala. En Casper, uno puede mandar el mensaje *add:* tantas veces como uno quiera para agregar tantos advices como se desee. Esto también es válido para los modificadores de clases con el mensaje *addClassModifier:*. Uno puede agregar tantos modificadores de clases como se desee. Al momento de instalar el aspectos con múltiples advices y/o múltiples modificadores de clases se producen en el sistema todos los cambios que los advices y los modificadores de clases producen, y al momento de desinstalar también se revierten todos los cambios hechos por todos los advices y modificadores de clases.

Ya sabiendo que un aspecto puede tener muchos advices y modificadores de clases a la vez solo queda discutir como interactúan los dos mundos en el sistema. En Casper, los métodos que son agregados por los modificadores de clases también son parte del sistema y se comportan como todos los otros métodos del sistema. En particular, estos métodos generan join points que pueden ser capturados por otros pointcuts y así ejecutar advices que son instalados por otros aspectos o por el mismo aspecto que instaló los nuevos métodos.

Aspectos Instalados

En Casper, al igual que PHANtom, hay una clase en el sistema que posee toda la información de los aspectos instalados y como instalarlos. Esta clase se llama *PhAspectWeaver*. Esta clase contiene información sobre los modificadores de clase, definiciones de métodos y, en particular, de los aspectos instalados. La clase *PhAspectWeaver* acepta varios mensajes, donde se destacan dos de ellos, *installedAspects* y *uninstallAll*. El mensaje *installedAspects* devuelve todos los aspectos que han sido instalados en el sistema. Con esto se puede mantener informado de los aspectos que se han instalados, dándole la posibilidad al programador de tomar los aspectos y desinstalarlos. El otro mensaje, *uninstallAll*, desinstala todos los aspectos que han sido instalados en el sistema.

3.4.2. Ejemplo de Uso

Los aspectos en Casper son representados por la clase *PhAspect*. Como cualquier clase del sistema, *PhAspect* puede ser extendida para crear un aspecto específico. La información del aspecto puede ser entregada en el inicializador y así, después de instanciar la clase que hereda de *PhAspect*, solamente faltaría instalar y desinstalar el aspectos. Incluso se podría mandar esta instancia del aspecto como parámetro al método *send:to:* de la clase *PhAdvice*.

Para entender mejor esto se extenderá el ejemplo de fibonacci de la sección 3.3 para optimizar la función. La función de fibonacci es una función muy fácil de definir, pero que es muy lenta con la definición formal. En cambio uno puede optimizar esta función

usando *memoization*. *Memoization* es una técnica de optimización, donde se guardan los resultados de las llamadas a una función, dado ciertos parámetros, para evitar tener que repetir los cálculos de aquella función cuando se llama con los mismos parámetros. En el ejemplo, se instalará la función original de fibonacci y luego se capturará para que use la versión optimizada.

Primero se define la clase *PhNumberAspect* que extiende de *PhAspect* (figura 3.4). La clase definida *PhNumberAspect* tiene una variable de instancia llamada *fibCache* que guarda un caché de resultados de fibonacci. Luego se define el método *initialize*, el cual se ejecutará cuando la clase *PhNumberAspect* reciba el mensaje *new*. Este método tendrá todos los pointcuts, aspectos y modificadores de clases que se necesiten para instalar el aspectos. En particular, se sabe que existe la clase *PhNumberModifier* en el sistema con el método fibonacci, ambos definidos respectivamente en la figura 3.2 y en la figura 3.3.

```
PhNumberAspect>>initialize
  |pcFib pcCm cm adv|
  super initialize.

  pcFib := PhPointcut
             receivers:'Number'
             selectors:'fibonacci'
             context:#(proceed receiver).
  pcFib noPackages: #().

  pcCm := PhPointcut receivers:'Number'.
  pcCm noPackages: #().
  cm := (PhNumberModifier new)
              pointcut: pcFib;
              addNewInstanceMethod:#fibonacci.

  adv := PhAdvice around:pcFib send:#dinamicFibonacci: to:self.

  fibCache := Dictionary new.
  fibCache at:0 put:0.
  fibCache at:1 put:1.

  self addClassModifier:cm.
  self add:adv.
```

Para que esto funcione, se debe crear el método *dinamicFibonacci*. Este método se encargará de usar el caché cuando corresponda.

```
PhNumberAspect>>dinamicFibonacci: ctx
  |exist|
  exist := fibCache includesKey: ctx receiver.
  exist
    ifFalse:[fibCache at: (ctx receiver) put: (ctx proceed)].
  ^fibCache at:(ctx receiver)
```

En este método cuando no se encuentra un valor guardado en el caché de fibonacci se utiliza la instrucción *ctx proceed*. Esta instrucción continua con la ejecución normal de fibonacci, entregando un valor definido por la función de fibonacci original. Ahora si el valor ya se encontraba guardado en el caché, simplemente se retorna este valor sin pasar de nuevo por la ejecución de fibonacci. Esto es posible gracias a que se define un advice del tipo *around*. En este caso, el advice que se define como la ejecución de *dinamicFibonacci* en una instancia de *PhNumberAspect* reemplazará a la ejecución original del método *fibonacci*. Si, dentro de la ejecución del advice, se llama a la instrucción *ctx proceed*, se puede tomar la ejecución original de *fibonacci*.

```
PhAspect subclass: #PhNumberAspect
  instanceVariableNames: 'fibCache'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Casper-Test'
```

Figura 3.4: Definición de la clase PhNumberAspect

3.5. Resumen

En esta sección se presentó el lenguaje de aspectos desarrollado, Casper.

En la sección 3.1 se describieron los pointcuts y como se definen en Casper. Se explicó como capturar con un solo pointcut un conjunto de join points con el uso de patrones. Además también se mostró como se pueden capturar join points con el uso de pragmas. También se mostró como se pueden componer los pointcuts y como se definen los pointcuts condicionales. Luego, se explicó como se puede restringir la captura de los join points a un determinado package de Pharo. Finalmente se mostró como se puede exponer distintos elementos del contexto de los pointcuts que pueden ser usado después por los advices.

En la sección 3.2 se describieron los advices en Casper y como definirlos. Se explicaron los tres tipos diferentes de advices: before, after y around. También se explicaron los dos tipos de acciones de los advices y el uso de elementos del contexto que fueron expuestos en el pointcut donde la acción del advice será ejecutada.

En la sección 3.2.3 se explicó sobre la exposición del contexto y la representación del contexto. Además se mostraron ejemplos de advices que usan información del contexto.

En la sección 3.3 se explicó lo que son los modificadores de clase en Casper y como crearlos. También se mostró como con esta herramientas se puede agregar nuevo comportamiento a las clases del sistema.

En la sección 3.4 se explicaron lo que son los aspectos en si. Como poder agregar advices y modificadores de clase en una instancia de un aspecto. También se explicó como los modificadores de clase pueden ser capturados por un pointcut y definir acciones a través de un advices que es instalado por el mismo aspecto o por otro. Luego se explica como revisar los aspectos instalados del sistema y como poder desinstalar todos los aspectos del sistema. Finalmente se explicó como usar los aspectos de forma avanzada, concluyendo con un ejemplo.

La sintaxis de Casper se basa en la sintaxis de PHANtom, excepto por algunos casos especiales. Los casos especiales se decidieron atacarlos de forma más simple o de forma diferente. Las grandes diferencias son que cambian algunas formas de exponer el contexto (sección 3.1.4) y la forma en que se escriben los modificadores de clase (sección 3.3)

Capítulo 4

Casper: Implementación

En esta sección se describe la implementación de Casper. Primero se presenta un análisis general de todas las clases para luego detallar las clases más importantes del sistema. Aunque la sintaxis del lenguaje es muy parecida a la de PHANtom, la implementación de Casper es totalmente diferente a la de PHANtom.

En PHANtom se utilizan *Method Wrappers*. Los method wrappers además de guardar información permiten realizar los cambios en los join point shadows capturados sin utilizar el compilador de Pharo Smalltalk. Esto hace que las acciones extras que se ejecutan se deciden en tiempo de ejecución. Casper busca bajar el tiempo de ejecución de los programas que usan un enfoque orientado a aspectos. Es por esto que Casper usa un enfoque totalmente distinto. En Casper las acciones extras de los aspectos se compilan, junto con los join point shadows. La idea es cargar más tiempo de la infraestructura de aspectos a la compilación y no a la ejecución.

4.1. Visión General de las Clases de Casper

Pharo Smalltalk es un lenguaje conocido por sus propiedades de metaprogramación. Estas propiedades son usadas por Casper para buscar en tiempo de ejecución las clases del sistema y los mensajes que reciben. Cuando los pointcuts capturan join points en Smalltalk se modifican los métodos que representan los join point shadows. Por eso, las propiedades de metaprogramación son imprescindibles para la implementación de Casper.

Las clases más importantes del sistema se presentan a continuación:

- **PhMethodDefinition 4.2** - Guarda información sobre los advices que se ejecutan en un método.
- **PhContext 4.4** - Se usa para obtener la información de la exposición del contexto.
- **PhSourceBuilder 4.3** - Clase importante para reescribir los códigos fuentes

originales

- **PhPragmaMatcher 4.5** - Se usa para matchear los comodines sobre los pragmas
- **PhSelectorMatcher 4.5** - Se usa para matchear los comodines sobre los nombre de los selectores
- **PhReceiverMatcher 4.5** - Se usa para matchear los comodines sobre las clases
- **PhPointcut 4.5** - Representa la información de un pointcut en Casper
- **PhAdvice 4.6** - Representa la información de un advice en Casper
- **PhClassModifier 4.7** - Representa la información de un modificador de clase en Casper
- **PhAspect 4.8** - Representa la información de un aspecto en Casper
- **PhAspectWeaver 4.8** - La clase que realiza el weaving y la instalación de los aspectos

4.2. Información Extra de los Métodos

Como se explicó en el capítulo 3, los join points en Casper corresponden al envío de un mensaje a un objeto. Cada vez que se instala un aspecto, se verifican los join point shadows que corresponden a un join point capturado. Luego, por cada uno de los join point shadows se genera una instancia de la clase *PhMethodDefinition*. Los objetos de la clase *PhMethodDefinition* se encargan de guardar toda la información extra que se necesita para modificar el comportamiento del join point shadow. Aunque todo sucede en tiempo de ejecución, la carga de la información se genera en el momento mismo de la instalación del aspecto. Así no se influye en el tiempo de ejecución del programa si los aspectos son instalados antes de la ejecución.

Una instancia de la clase *PhMethodDefinition* guarda la información de todos los advices que se ejecutarán y que se podrían ejecutar (pointcut condicionales) en el join point shadow. Además guarda un nombre nuevo de un método que usará el generador de código más adelante (ver sección 4.3).

Una de las complicaciones de trabajar con código fuente se produce cuando se captura un join point en una subclase. Por ejemplo, existe un pointcut que captura el mensaje *speed:* de la clase *Car*, pero este mensaje esta definido en la clase *Vehicle*, que es el padre de la clase *Car*. Para abordar este problema se guarda más información en la instancia de *PhMethodDefinition* que se genera con el join point shadow. Se guarda la clase que contiene la definición del método original. Luego, al momento de instalar el aspecto, se replica el método *speed:* en la clase *Car*, así se maneja como si el método siempre existió en la clase *Car*. Cuando el aspecto se desinstala, este método replicado también se elimina, para así volver todo a la normalidad.

4.3. Generador de Código Fuente

La implementación de Casper se basa en modificar los códigos fuentes de los join point shadows que corresponden a los join point que son capturados. Para esto se utiliza la clase *PhSourceBuilder*. Por cada uno de los join point shadows se genera una instancia de la clase *PhSourceBuilder*. Las instancias de esta clase se encargan de crear los nuevos códigos fuentes de los join point shadow que son capturados. Los códigos fuentes son generados en base al join point shadow y a la nueva información que entrega el *PhMethodDefinition*. Los advices que entrega la instancia de *PhMethodDefinition* se compilan en el código fuente generado. Cuando el advice tiene como acción un bloque (no es del tipo *send:to:*), Casper escribe este bloque de forma explícita y no como una referencia al bloque. Esto último se hace para que el usuario vea que la acción de forma explícita y sepa lo que sucede en el método que ha capturado.

El código fuente original se guarda en un método nuevo el código fuente del join point shadow con un nombre donde se le antepone el nombre original el texto *casperXXXX*, con *XXXX* un número de cuatro cifras. Por ejemplo si se captura el mensaje *speed:* de la clase *Vehicle*, el método original se guarda en *casper1000Speed:*. Si luego se captura el mensaje *stop* de la clase *Vehicle*, se genera el método *casper1001Stop*. Estos nuevos métodos se guardan en la categoría "casper autogenerated", así se le muestra explícitamente al programador donde se encuentran estos métodos autogenerados.

Lo anterior no funciona para métodos que tienen como nombre un símbolo, como la suma (+). Esto es porque en Smalltalk solo se permiten símbolos como nombre cuando tan solo se escriben símbolos, esto hace que falle cuando queremos escribir un método con el nombre autogenerado por Casper de forma normal, como sería *casper1000+:*. Para evitar esto, en Casper definimos una serie de símbolos que pueden ser usados. Los símbolos que pueden ser capturados por Casper hasta ahora son: $. + - * > | \& = < /$. Si se desea agregar más símbolos solo basta modificar la lista de caracteres en "*PhSelectorMatcher class>>specialCharacters*". Si se encuentra un símbolo, este se reemplaza por una letra "c" y el número que es su representación en ASCII. Por ejemplo, si capturamos el mensaje ">" de la clase *Integer*, en Casper el nombre del método autogenerado será: *casper1000c62:*. Si hay dos o más caracteres especiales, se siguen concatenando la letra "c" y el número de la representación ASCII por cada uno de los caracteres. Por ejemplo para el mensaje ">>", el nombre del método autogenerado será: *casper1000c62c62:*

El Generador de Código Fuente modifica el join point shadow y crea un nuevo código fuente con más acciones e información. El código original del join point shadow se guarda en otro método con un nombre especificado por el *PhMethodDefinition*. Esto se hace para poder llamar al método original en el método generado cuando corresponda y para poder revertir los cambios realizados por el generador de código. El código generado se puede dividir en ocho secciones:

- Firma e información
- Pre procesamiento
- Before advices

- Método original
- Around advices
- Ejecución
- After advices
- Retorno

Ahora vamos a detallar cada una de estas secciones

4.3.1. Firma e Información

El nombre del método es el mismo del join point shadow, lo que cambia son el nombre de las variables. Para los nombres de las variables se usa la misma convención que para el nombre del método que se generará con el código original. Esto es porque en Smalltalk generalmente se usa una convención para denotar el tipo de los nombres de variables. Por ejemplo, si el método *speed:* recibe un número, en Smalltalk la firma del método es:

```
speed: aNumber
```

Debido a esta convención y a que el advice se escribe de forma explícita, es posible que los nombres de variables del método original choquen con algún nombre de variable que el usuario usa en un advice, por eso se cambia el nombre de las variables. Tomando el ejemplo anterior, si el método *speed* es capturado, el código fuente nuevo de la firma se escribe:

```
Vehicle>>speed: casper1002ANumber
```

Luego de la firma se escribe un comentario explicando donde se puede encontrar el método original. Para el mismo método *speed:*, se tiene:

```
Vehicle>>speed: casper1002ANumber
  "This method has been modified by Casper.
  The original implementation can be found in casper1002Speed:"
```

4.3.2. Pre procesamiento

En esta sección del código fuente se realiza todo el pre procesamiento para lograr que las siguientes secciones del código funcionen. En Casper, el único tipo de pre procesamiento que se define es agregar variables y asignar valores a estas variables. Las variables más importantes que se definen son:

- **phOriginalMethod** Representa un bloque que contiene a la llamada del método original

- **phMethodDefinitions** Representa la instancia de *PhMethodDefinition* que es usada en este join point
- **phCtx** Representa el contexto que esta expuesto. El contexto global es un conjunto de todas las exposiciones del contexto hechas por los pointcuts
- **phArgsVar** Representa un arreglo con todos los argumentos que recibe el método. Se usa tanto para el contexto como para la llamada del método original.
- **phReturn** Es el valor de retorno del método original o del advice de tipo *around*
- **phAroundAdvice** Representa un bloque con el llamado del advice que reemplaza a la ejecución original.
- **phAdvice** Representa un advice que se usa para llamadas especiales sobre él, como saber si ese advice es de un tipo condicional y si se debe ejecutar o no en el join point, entre otras.

4.3.3. Before Advices

En esta sección se escriben explícitamente en el código los advices que se ejecutan antes del código original y un comentario al comienzo diciendo que se encuentra en las sección de *before advices*. Si la acción del advice viene del mensaje *advice:*, esto significa que la acción es un bloque. Cuando la acción es un bloque este se escribe directamente en el nuevo código fuente y se evalúa inmediatamente con el contexto global. Por ejemplo, volviendo al método *speed:*, si existe un advice que tiene como bloque *[:ctx| Transcript show: 'Before advice']*, esto se verá de la siguiente forma en el código:

```
Vehicle>>speed: casper1002ANumber
  ...
  "Before Advices"
  [:ctx| Transcript show: 'Before advice'] value: phCtx.
```

Si la acción del advice viene del mensaje *send:to:*, en el código se ejecuta el selector que se entrega en *send:* al objeto que se entrega en *to:*. Se supone nuevamente que se captura el método *speed:*, pero esta vez, el advice que tiene como acción viene del mensaje *send:to:*, con argumentos *#speedChange:* y *printer*, donde *printer* es un objeto que se definió en su momento. Todo esto se verá de la siguiente manera en el código:

```
Vehicle>>speed: casper1002ANumber
  ...
  "Before Advices"
  phAdvice := phMethodDefinition advices before at:1.
  phAdvice objectSend perform: #speedChange: with: phCtx.
```

Como se ve, primero se debe obtener el advice que corresponde a la ejecución del mensaje *send:to:*. Para ello se necesita tener la información extra del join point para luego obtener el advice que se quiere ejecutar. Al tener el advice se puede obtener el objeto que fue entregado al advice con el mensaje *objectSend*, y luego, con el mensaje *perform:with:*, se le puede entregar el mensaje a ejecutar a tal objeto. Además se le

entrega el contexto como argumento a la ejecución de este mensaje.

Cuando hay más de un advice del tipo *before* que se ejecuta en el mismo join point, estos solos se evalúan uno después de otro.

4.3.4. Método Original

Luego de escribir los before advices en el código fuente, se escribe la ejecución del código original, pero capturada en un bloque dentro de una variable. La idea es tener guardada la ejecución, pero no ejecutarla de inmediato ya que aun queda saber si hay advices del tipo *around*. Los advices del tipo *around* reemplazan la ejecución del método original, pero también puede ser posible ejecutar el método original desde un advice del tipo *around* con una llamada al método *proceed* del contexto. Debido a esto, capturar le ejecución tiene dos objetivos. Primero, no ejecutar el código inmediatamente y segundo, guardarla para una posible ejecución que se genere en una llamada a *proceed*.

En el siguiente ejemplo, se verá como se escribe todo lo dicho anteriormente en el código fuente del método *speed:*.

```
Vehicle>>speed: casper1002ANumber
  ...
  "Original Method"
  phOriginalMethod :=
    [:args|
      self perform: #casper1002Speed: withArguments: args.
    ].
```

Se puede notar como se usa la variable *phOriginalMethod* para guardar un bloque que captura la ejecución del método original. Como se explica en la sección 4.2, el método original se guarda en otro método. En el ejemplo, se guarda en el método de nombre *casper1002Speed:*.

4.3.5. Around Advices

Los advices del tipo *around* son los más complicados de escribir de los tres tipos. Los advices de tipo *before* y *after* basta con evaluarlos todos juntos en el lugar que les correspondes, los advices del tipo *around* funcionan de forma diferente. Si se definen varios advices de tipo *around* en un mismo join point, solo se ejecuta uno de ellos. Si el que se ejecuta llama al método *proceed*, se ejecutará en ese momento otros de los advices del tipo *around*. Esto es posible hasta que se ejecuta el método original por medio de la llamada a *proceed*.

Para soportar todo esto se realizan manejos de los contextos de cada advice de tipo *around* y capturas en bloque de la ejecución. Primero se captura toda la ejecución de los advices del tipo *around* en un bloque. Luego dentro de este bloque se definen 2N, donde

si hay N advices del tipo *around*, se definen N variables por cada advice y por cada contexto. Cada uno de los contexto es igual al contexto global, excepto si el pointcut que captura el join point expuso la variable *proceed*. Si se expone la variable *proceed*, para el último advice que se ejecuta, se guarda el bloque que captura la ejecución original y los argumentos. Para los demás advices, se guarda el siguiente bloque que se ejecutaría si se llama a proceed y el contexto. Finalmente, se evalúa el primer advice que es ejecutado.

En el ejemplo se podrá observar el código generado en el join point *speed:*, donde se capturan dos advices.

```
Vehicle>>speed: casper1002ANumber
  ...
  "Around Advice - in reverse order of execution"
  phAroundAdvice := [
    |phCtx2 phAroundAdvice2 phCtx1 phAroundAdvice1 |
    phCtx2 := phCtx copy.
    phCtx2 proceedBlock: phOriginalMethod.
    phCtx2 isOriginal: true.
    phCtx2 originalArgs: phArgsVar.
    phAroundAdvice2 :=
      [:ctx | Transcript show: 'Around 2'. ctx proceed].

    phCtx1 := phCtx copy.
    phCtx1 proceedBlock: phAroundAdvice2.
    phCtx1 nextContext: phCtx2.
    phAroundAdvice1 :=
      [:ctx | Transcript show: 'Around 1'. ctx proceed].

    phAroundAdvice1 value: phCtx1.
  ].
```

En el ejemplo se puede ver como es capturado toda la ejecución de los advices del tipo *around* en la variable *phAroundAdvice*. Luego aparecen las variables auxiliares, al ser dos advices los que ejecutan, son cuatro variables las que se definen. Los advices se escriben en el orden inverso que serán ejecutados. Primero se escribe el último advice que será ejecutado, en la variable de su contexto (*phCtx2*) se guarda que la llamada a proceed es el bloque que captura la ejecución del método original, y en la variable *phAroundAdvice2* se guarda la acción que ejecutará este advice. Entonces se escribe el siguiente advice, que en este caso es el primero en ejecutarse. En este lugar se guarda en su contexto lo que será la ejecución si existe un proceed - con el mensaje *proceedBlock:* - y además el siguiente contexto que se usará - con el mensaje *nextContext:* - ambos recibiendo el bloque y el contexto del advice que se escribió primero. Finalmente se hace una evalúa el primer advice que será ejecutado, en este caso, *phAroundAdvice1*.

4.3.6. Ejecución

En esta sección se hace la ejecución del método original o del advice del tipo *around* según corresponda. Esta ejecución se realiza antes de evaluar cualquier advice de tipo *after*. Si el join point no tiene advices del tipo *around*, se escribe:

```
Vehicle>>speed: casper1002ANumber
  ...
  "Execution"
  phReturn := phOriginalMethod value: phArgsVar.
```

Donde *phOriginalMethod* guarda la ejecución del método original. Este se evalúa con los argumentos originales, los cuales fueron guardados en la etapa de Pre procesamiento (sección 4.3.2) en la variable *phArgsVar*.

Si el join point tiene advices del tipo *around*, entonces se escribe:

```
Vehicle>>speed: casper1002ANumber
  ...
  "Execution"
  phReturn := phAroundAdvice value.
```

Donde *phAroundAdvice* es el bloque que contiene la ejecución de todos los advices del tipo *around*. Aquel bloque contiene una referencia a los argumentos originales por si se decide realizar una llamada del tipo *proceed*.

Además, en ambos casos, se guarda en la variable *phReturn* el valor de retorno que entrega la ejecución del método original o de los advices. Este no puede ser retornado, porque falta por ejecutar los advices del tipo *after*

4.3.7. After Advices

En esta sección se escriben explícitamente los advices que se ejecutan después de la ejecución del método original. Acá se escriben de la misma forma que en la sección de *Before Advices*. En el siguiente ejemplo vemos como se escribe un advice del tipo *around* con bloque *[:ctx| Transcript show: 'After advice']*.

```
Vehicle>>speed: casper1002ANumber
  ...
  "After Advices"
  [:ctx| Transcript show: 'After advice'] value: phCtx.
```

4.3.8. Retorno

La última sección del código es donde se produce el retorno de la función. El valor de retorno es definido en la sección donde se produce la ejecución. En Pharo Smalltalk, cuando una función no define valor de retorno por defecto retorna *self*, esto es, su misma instancia. Por ende, aunque la función original no defina un valor de retorno, el retorno existe por defecto. Por lo tanto el valor de retorno es bien definido por el retorno de la función original. El valor de retorno esta definido en la variable *phReturn*.

En el siguiente ejemplo, se concluirá con el análisis del código fuente de *speed:*, definiendo el valor de retorno:

```
Vehicle>>speed: casper1002ANumber
  ...
  "Return Value"
  ^phReturn.
```

4.4. Contexto

Como se explicó en 3.1.4, una instancia de *PhContext* guarda información dependiendo de lo que el usuario de Casper decida. Cada una de las exposiciones del contexto guarda distinta información para lograr la funcionalidad deseada en los advices. Las distintas exposiciones del contexto se ven de la siguiente forma en el código fuente:

- #**receiver** *phCtx receiver: self*
- #**sender** *phCtx sender: thisContext sender receiver*
- #**selector** *phCtx selector: thisContext method selector*
- #**arguments** *phCtx arguments: phArgsVar*
- #**thisAdvice** *phCtx thisAdvice: phMethodDefinition advices [before|around|after] at:n* (con n un número entero)
- #**proceed** ver sección 4.3.5

En Pharo Smalltalk existe una palabra reservada que entrega información sobre el contexto de la ejecución del método, esta palabra se llama *thisContext*. Casper aprovecha la información que entrega esta variable para obtener datos sobre el contexto. Esto se ve cuando se exponen los datos de *sender* y *selector*. Para obtener los argumentos, en la sección de pre procesamiento se obtienen estas variables y se guardan en la variable *phArgsVar*. Para obtener el advice por medio de *thisAdvice* se necesitan varias cosas. Primero, se tiene que obtener los advices por medio de *phMethodDefinition*, conociendo el tipo del advice y también el lugar en la ejecución. Esto es, si se toma *phCtx thisAdvice: phMethodDefinition advices before at:1* como ejemplo, se ve que se obtiene el advice de los advices de tipo *before*, además se ve que es el primero en la ejecución - por el valor del argumento "1" que se le pasa a *at:* -.

4.5. Pointcuts y Captura de Join Point

La instancias de la clase *PhPointcut* se encargan de guardar la información necesaria para obtener capturar los join points. En Casper, al igual que PHANtom, un join point es una tupla de tipo {Objeto receptor, mensaje}. La captura de join points se realiza siempre cuando se llame al mensaje *getPointcuts* y depende de los patrones dados tanto en el mensaje *receivers:*, *selectors:* y *pragmas:*. Para ver los join points que capturan los patrones, se usan las clases *PhReceiverMatcher*, *PhSelectorMatcher* y *PhPragmaMatcher*. Estas tres clases se dedican a revisar el sistema para capturar los join points.

La clase *PhReceiverMatcher* se encarga de entregar las clases del sistema que son capturadas por el patrón recibido en el mensaje *receivers:*. Lo que hace es buscar en todas las clases del sistema y verifica cual de ellas son capturadas por el patrón. Si se restringe las clases que captura el pointcut a algunos *packages* - con el mensaje *restrict:* -, en lugar de buscar por todo el sistema, solo busca en las clases que están en los *packages* definidos.

La clase *PhSelectorMatcher* hace lo mismo que la clase *PhReceiverMatcher*, pero con los métodos de la clase. La clase *PhSelectorMatcher* recibe una lista de clases y entrega un par {Objeto receptor, mensaje}, cuando el mensaje es capturado por el patrón entregado en *selectors:*. La lista de clases que recibe *PhSelectorMatcher* es la misma lista que retorna *PhReceiverMatcher*. Los mensajes que se aceptan no solo son los mensajes que se definen en la clase, sino todos los mensajes que entiende, esto es, incluso los mensajes de las super clases y más.

La clase *PhPragmaMatcher* recibe una lista de pares {Objeto receptor, mensaje} y se encarga de retornar solo aquellos pares donde el mensaje tenga definido un pragma capturado por el patrón que se entre en *pragmas:*. Un mensaje puede tener más de un pragma definido, pero solo basta que uno sea capturado por el patrón para aceptar este mensaje.

4.5.1. Composición

La composición de pointcuts se maneja de forma sencilla usando conjuntos. La operación "and" toma dos pointcuts y realiza una intersección de los dos conjuntos de join points que generan los pointcuts. La operación "or" toma los dos pointcuts y realiza una unión de los dos conjuntos de join points. En cambio la operación "not" es algo más complicada. Primero toma todas las clases del sistema que no son capturadas, junto a todos sus mensajes, luego toma todos los mensajes que no son capturados, ni por el patrón del mensaje ni por el del pragma, de las clases que si son capturadas. Esta combinación entrega todos los join points de la operación "not".

4.5.2. Condicional

Como se habló en la sección 3.1.6 los pointcuts condicionales son de dos tipos, los que tienen el mensaje *if:* y los que se definen con *cflow:*. Los join point shadows que generan ambos tipos de pointcuts son los mismos sino estuviera la parte condicional, esto es porque la parte condicional se determina en tiempo de ejecución y un join point shadow es una representación estática del join point capturado de forma condicional. Es en el advice y en la escritura del código fuente donde se determina si la parte condicional afecta o no la ejecución del advice en el join point capturado por el pointcut. Esto lo detallamos en la sección 4.6

4.6. Advices

La clase *PhAdvice* contiene la información necesaria que define un advice. En esta clase se guarda la acción a realizar y el pointcut que captura los join points. Este pointcut además entrega información extra, como la exposición del contexto y si es un pointcut condicional o no. Los advices son guardados en la definición extra del método, representado por la clase *PhMethodDefinition*. Los advices se guardan bajo la instancia de la clase *PhAdviceGroup*. Lo que hace esta clase es separar los advices en sus tres tipos. Esto se hace para que la búsqueda de los advices cuando se tienen que llamar en el código fuente generado en el join point shadow sea mucho más fácil y directa.

Cuando existen pointcut condicionales, el advice hace llamadas extras a nivel de ejecución del programa para saber si se debe ejecutar o no. Estas llamadas se ven reflejadas en el código fuente a través de la llamada *shouldExecuteOn:*. Esta llamada retorna un valor booleano que dice si se debe ejecutar el advice o no. La forma de calcular este valor depende del tipo de condicional que es el pointcut que captura el join point. Si el pointcut posee el bloque *if:*, entonces se evalúa este bloque con el contexto que se entrega como argumento a *shouldExecuteOn:*.

Cuando el pointcut que captura el join point es del tipo *cflow:* se debe hacer un trabajo mayor. Si el pointcut que recibe como argumento se llama *pc2*, entonces se debe revisar si el flujo de ejecución que ha permitido llegar hasta la ejecución del join point alguna vez pasa por un join point que es capturado por *pc2*. Esto se puede lograr gracias a la variable *thisContext* que entrega Pharo Smalltalk. Lo que se hace en Casper es buscar, por medio de la variable *thisContext*, si se ha ejecutado algún join point que captura *pc2*. Si efectivamente captura uno, el advice se ejecuta. Esta condición puede ser muy costosa a tiempo de ejecución si la llamada que es capturada por el join point esta muy al comienzo de todas las llamadas, esto es, este muy cercano al fondo de la pila de llamadas.

Cuando el pointcut tiene los dos tipos de condicionales, primero se verifica el bloque *if:* y luego la condición de *cflow:*. Así, sino se verifica el bloque *if:* no debe hacer toda la verificación de *cflow:*, siendo que esta última eventualmente puede ser muy costosa.

Si suponemos que el join point de *speed:* es capturado por pointcut condicionales, la forma en como se escribe en el código fuente es la siguiente:

```
Vehicle>>speed: casper1002ANumber
  ...
  phAdvice := phMethodDefinition advices before at:1.
  (phAdvice shouldExecuteOn: phCtx)
  ifTrue:[
    [:ctx | Transcript show: 'some advice'] value: phCtx.
  ].
  ...
```

Cuando el advice es del tipo around, si el pointcut es condicional, al momento de hacer *proceed* se debe verificar la condición, si la condición no verifica, se debe seguir llamando a *proceed*, para que el siguiente advice o el método original sea llamado. Esto puede seguir muchas veces si, eventualmente, los siguientes advices también dependen de un pointcut condicional. Aunque estas llamadas no son infinitas, porque al menos se debe terminar con la ejecución del método original.

4.7. Modificadores de Clase

Para lograr las modificaciones de clase, Casper toma como ejemplo PHANtom y se aprovecha de las capacidades reflexivas de Pharo Smalltalk. Cuando se utilizan los métodos *addInstanceVarNamed:* y *addClassVarNamed:* de la clase *PhClassModifier*, se agregan nuevas variables de instancia y de clase respectivamente. Estas variables se agregan a las clases que son capturadas por el pointcut *pointcut:*. Esto funciona de la misma manera con los métodos *addInstanceMethod:* y *addClassMethod:*, para agregar métodos de instancia y de clase respectivamente. Sin embargo estos métodos deben estar definidos en una subclase de *PhClassModifier*. Cuando se instalan estos métodos, se toma el código fuente del método y se compila en la clase donde se desea instalar. Es por esto si el método ocupa variables de instancia o de clases, estas también deben de agregarse con los métodos correspondientes.

Cuando se agregan variables o métodos, tanto de instancia como de clase, primero se verifica la existencia de alguno de ellos anteriormente. Esto se hace para evitar modificar la clase agregando variables o métodos que ya habían sido definidos con anterioridad.

Los métodos que se agregan a la clase se guardan bajo la categoría "casper class modifiers", así quedan explícitos para el programador y pueden encontrarlos sin ningún problema. La infraestructura de Casper asegura que estos métodos no pueden ser eliminados como los métodos normales y que la única forma de eliminarlos es desinstalando el aspecto que agregó los métodos. Además estos métodos agregados por Casper quedan en el diccionario de clases, siendo posible la captura a través de un pointcut de un aspecto.

4.8. Aspectos

Al instalar los aspectos, toda la información necesaria queda guardada en la clase *PhAspectWeaver*. La instalación de un aspecto genera tantas instancias de *PhMethodDefinition* como join point shadows que son capturados en el sistema. Cada una de estas instancias guarda información del método, del método autogenerado por Casper donde se encuentra el código original y de todos los advices que podrían ser ejecutados en el join point shadow. Además en la clase *PhAspectWeaver* también se guardan todos los modificadores de clase que actúan en el sistema.

Cuando un aspecto se instala, primero se hacen efectivos los cambios que se producen por los modificadores de clase, así un pointcut puede capturar los cambios hechos por los modificadores de clase sin ningún problema. Detalladamente, lo que sucede cuando un aspecto es instalado es:

- Se desinstalan todos los aspectos que han sido instalados en el sistema
- Se instalan todos los modificadores de clase, tanto de los antiguos aspectos como del nuevo
- Se genera la nueva información dada por los join point shadows del aspecto instalado y, si es necesario, nuevas instancias de *PhMethodDefinition*
- Se instalan todos los *PhMethodDefinition* que se han generado en el paso anterior

Todos los cambios hechos por los aspectos en los códigos fuentes no pueden ser modificados de la forma normal, ya que estos cambios son interceptados por Casper. Las modificaciones que se pueden hacer se listan a continuación:

- El método generado por Casper - método que tiene en el nombre *casperXXXX* - se puede modificar normalmente
- El método original, cuyo código fuente ha sido cambiado, no se puede modificar. Las modificaciones no se verán reflejadas al momento de guardarlas
- Si se elimina el método original o el método autogenerado por Casper, ambos se eliminan
- No se puede cambiar de categoría un método autogenerado por Casper o uno instalado por los modificadores de clase
- No se puede eliminar un método instalado por los modificadores de clase
- No se puede quitar una variable instalada por los modificadores de clase

4.9. Test

Para comprobar el funcionamiento de Casper, se usó el sistema de test unitarios provisto por Pharo Smalltalk. Ya que Casper basa su sintaxis en PHANtom se usaron los test que ya están escritos en PHANtom. Esto permitió partir con un buen grado de cobertura de tests [6].

Los tests que se realizaron se pueden dividir en tres tipos: tests sobre pointcuts, tests sobre aspectos y advices, y tests sobre modificadores de clase. Los tests sobre pointcuts comprueban la correcta correspondencia entre un pointcut y los join points que captura. Se comprueban el correcto funcionamiento de los patrones en los selectores y en los receptores del mensaje. Además se comprueba el correcto funcionamiento de la captura de pointcuts para un mensaje en una clase o en las clases padres. Se comprueba también el funcionamiento de la composición de pointcuts y los pointcuts condicionales. Finalmente se comprueba, con un aspecto funcionando, la exposición del contexto que puede generar un pointcut.

Los tests sobre aspectos y advices comprueban que un advice se ejecute en los join points que le corresponde. Se realizan tests sobre un advice de tipo *before*, *after* y *around*. Se comprueba también múltiples tipos de advices ejecutando sobre el mismo join point. Finalmente se comprueba también el correcto funcionamiento de los dos tipos de acciones de advice, un advice como un bloque de código o como un mensaje hacia una instancia de una clase.

Los tests sobre los modificadores de clase se reescribieron para que puedan soportar la nueva sintaxis provista por Casper.

Para las nuevas funcionalidades provistas por Casper, como la captura de join points a través de pointcuts que definen patrones a nivel de pragmas, se escribieron nuevos tests para comprobar el correcto funcionamiento de las nuevas funcionalidades.

Además se eliminaron tests de PHANtom que prueban funcionalidades que solo se encuentran en ese lenguaje, como es el uso de membranas.

4.10. Resumen

En esta sección se presentó cómo se implementó el lenguaje desarrollado, Casper. Se describieron sus clases importantes y como Casper cambia el código fuente de los join point para ejecutar los advices.

En la sección 4.2 se describieron los *PhMethodDefinition* y cómo estos guardan la información necesaria para realizar los cambios en el código fuente.

En la sección 4.3 se describió como se realizan los cambios en el código fuente de los join point shadows. Se presentaron todas las secciones y se mostró explícitamente el código generado en varios ejemplos.

En la sección 4.4 se describió el contexto y como se representa la exposición del contexto en el código fuente.

En la sección 4.5 se presentó como los pointcuts permiten capturar los join point por medio de otras clases que se dedican a esto.

En la sección 4.6 se describió como contienen la información necesaria para ser usados en el código fuente.

En la sección 4.7 se presentaron nuevamente de los modificadores de clase y como se usan las capacidad reflexivas de Pharo Smalltalk para poder agregar nueva información a las clases.

En la sección 4.8 se presentó como los aspectos guardan información y los cambios que se pueden hacer con los códigos generados.

En la sección 4.9 se describió como se hicieron los test para validar el lenguaje desarrollado.

Capítulo 5

Caso de Estudio: SPY

En esta sección se describe una aplicación que se le puede dar al lenguaje Casper. SPY [3] es una herramienta de profiling en Smalltalk. Al igual que PHANtom, se modificó la herramienta SPY para que utilice Casper en la instrumentalización del sistema.

5.1. SPY

SPY [3] es un framework que permite construir diferentes profilers y visualizar la información entregada por los profilers. Un profiler es una herramienta que permite obtener distinta información de un programa en tiempo de ejecución, como el tiempo y uso de memoria. SPY permite construir los profilers que uno desee de acuerdo a la necesidad del programa que se desea analizar.

Para crear un profiler con SPY, se deben crear subclases de las cuatro clases más importantes de SPY: PackageSpy, ClassSpy, MethodSpy y Profiler. Estas clases se explican a continuación:

- **PackageSpy** Contiene la información de hacer profiling en un package. Por cada una de las clases del package, es creada una instancia de ClassSpy.
- **ClassSpy** Contiene la información de hacer profiling en un clase. Por cada método dentro de la clase, se crea una instancia de MethodSpy.
- **MethodSpy** Instrumentaliza un método en Pharo Smalltalk para poder obtener la información de ejecución del código del programa que se esta evaluando. Para poder obtener la información se usan los métodos *before:run:with:in*, *after:run:with:in* y *run:with:in*
- **Profiler** Esta clase contiene las herramientas necesarias para obtener la información en tiempo de ejecución sobre un bloque de ejecución de código.

5.1.1. Creación de un Profiler

Como se comentó anteriormente, para crear un profiler con SPY se debe crear subclases de PackageSpy, ClassSpy, MethodSpy y Profiler, además se deben redefinir algunos métodos de estas subclases para que SPY entiende que instrumentalizar y como hacerlo. Los métodos que se deben redefinir son:

- Profiler class>>spyClassForPackage
- Profiler>>visualizeOn:
- PackageSpy class>>spyClassForClass
- ClassSpy class>>spyClassForMethod
- MethodSpy>>beforeRun:with:in
- MethodSpy>>afterRun:with:in
- MethodSpy>>run:with:in

Para la subclase de Profiler, se debe sobreescribir el método de clase *spyClassForPackage*. Este método retorna la clase que extiende de PackageSpy. Además, para producir la visualización que uno desee, se debe sobreescribir el método *visualizeOn:* y ahí escribir como uno desee que el profiler visualice la información.

Para la subclase de PackageSpy, se hace algo similar que con la clase que extiende de Profiler. En el lado de la clase, se sobreescribe el método *spyClassForClass* que retorna la clase que extiende de ClassSpy.

Para la subclase de ClassSpy sigue siendo similar. En el lado de la clase se sobreescribe el método *spyClassForMethod* que retorna la clase que extiende de MethodSpy.

La subclase de MethodSpy es donde se escriben los métodos que permitirán hacer la instrumentalización necesaria. En esta clase se definen las variables de instancia necesarias para hacer el profiling. Dependiendo del nivel de instrumentalización que se necesite, se pueden usar estos tres métodos: *before:Run:with:in*, *after:Run:with:in* y *run:with:in*. El método *before:run:with:in* se llama antes de cada ejecución del método original, *after:Run:with:in* se llama después de la ejecución del método original y *run:with:in* se llama en lugar del método original.

Para hacer profiling de un bloque de código solo basta con ejecutar *viewProfiling:* de la subclase de Profiler. A este método se le pasa como argumento un bloque de código que será el que se instrumentalizará.

5.2. Refactoring SPY con Casper

Para probar el funcionamiento Casper, se cambió la forma original en que SPY instrumentaliza los métodos del sistema. Se modificó de tal forma en que ocupe Casper para permitir las inserciones de código necesarias para instrumentalizar lo que requieren

los profilers.

Originalmente, SPY obtiene información dinámica del sistema haciendo un wrapper de los métodos compilados, similar a lo que ocurre en PHANtom. La información que obtiene SPY la va acumulando de la ejecución del bloque que se esta instrumentalizando.

Para instrumentalizar el código, SPY cambia el método original por otro que reifica el mensaje original, esto es, genera un nuevo método teniendo como información el código fuente del método original. El nuevo método es en realidad una copia de un template que tiene SPY para poder generar el método correcto dependiendo de la cantidad de argumentos que tiene el método original. Gracias a esto, SPY asocia cada método a una instancia de MethodSPY.

Para poder usar SPY con Casper debemos hacer una serie de cambios en la infraestructura de SPY.

Primero, se agregan tres métodos a la clase MethodSPY: *afterRun:*, *beforeRun:* y *aroundRun:*. Cada uno de estos métodos está pensado para funcionar de la misma forma que *after:Run:with:in*, *before:Run:with:in* y *run:with:in* respectivamente. La idea es que luego se usarán advices con la acción *send:to:* con el pointcut respectivo. Así se podrá llamar a los métodos que obtienen los datos en tiempo de ejecución del bloque de código que se esta instrumentalizando.

Segundo, se modifica el código del método de clase *installOnBehavior:* de la clase Profiler. Ahora en lugar de cambiar los CompiledMethod como lo hace SPY, lo que se hace es crear un aspecto que tiene advices que como acción mandan el mensaje *beforeRun:*, *afterRun:* y *aroundRun:* a un objeto que es una instancia de una clase que extiende de MethodSpy. A modo de ejemplo desde ahora en adelante solo se usará la instrumentalización que en SPY se logra con el método *run:with:in*, por lo que en Casper el código que instrumentaliza se ve de esta forma:

```
Profiler class>>installOnBehavior: aBehavior
  ...
asp := (PhAspect new)
            add: (PhAdvice around: pc send: #aroundRun: to: methodSpy).
  PhAspectWeaver addAspect: asp.
  ...
```

La instalación de los aspectos en Casper requiere recompilar los códigos fuentes de los join point shadow, lo cual es un proceso muy costoso en tiempo. Como en SPY se recompilan una gran cantidad de métodos, lo que se hace en este método es primero agregar todos los aspectos que se deseen instalar. Luego se instalan todos juntos. Esto reduce considerablemente el tiempo de instalación de los aspectos.

Para soportar una instalación masiva de los aspectos, también se modificó el método de clase *profile:* de la clase Profiler. Esto es porque, justo después de agregar los métodos debemos instalarlos con la llamada *installAspect* directamente sobre la clase PhAspectWeaver. Esto se se ve así:

```
Profiler class>>profile: aBlock
  ...
  self installOnPackagesNamed: tally allInvolvedPackageNames.
  PhAspectWeaver installAspect.
  ...
```

Finalmente, se debe modificar una serie de métodos que permiten extraer la información de los MethodSpy y visualizarla. Estos métodos permiten saber si un CompiledMethod tiene un MethodSpy y permiten obtener el MethodSpy de un CompiledMethod. Como la forma de instalar los MethodSpy cambia, también lo hace la forma de saber si son MethodSpy. Los métodos que hay que cambiar son tres que agrega SPY a CompiledMethod: *hasMethodSpyAsLiteral*, *isSpy* y *methodSpy*; además de cambiar un método de Profiler, *fillClassType:with:*.

5.2.1. Pruebas y Resultados

Para probar si la instrumentalización de SPY con Casper fue correcta se probó con un profiler que viene incluido en SPY, KaiProfiler.

KaiProfiler se encarga más que nada de revisar los tiempos de ejecución de un programa. Para ello analiza los tiempos que toma cada método, el número de veces que cada método ha sido llamado y el número de diferentes receptores. Gracias el método *visualizeOn:* se puede ver toda esta información. La figura 5.1 representa una sección de la visualización que se genera al correr KaiProfiler sobre el bloque *[RBFormatterTests suite run]*. El bloque representa correr todos los test de RBFormatterTests, que son los tests sobre las clases responsables de formatear código.

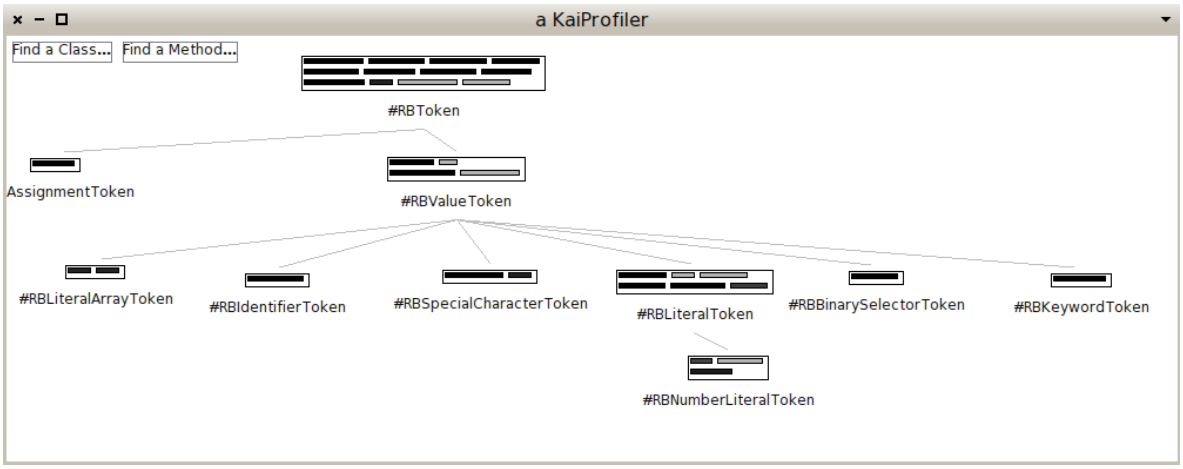

Figura 5.1: Sección KaiProfiler SPY sin Casper

En la figura, los cuadrados grandes representan las clases y los pequeños los métodos. Las aristas representan herencia de clases, donde la que esta arriba es superclase y las de más abajo son subclases. El alto de los métodos representa el tiempo de ejecución. El ancho representa el número de veces que el método se ha ejecutado y el color es el

número de diferentes receptores que el método ha ejecutado, mientras más oscuro el método, más receptores a ejecutado.

En la figura 5.2 se puede ver el mismo extracto de la visualización de KaiProfiler usando Casper. Se puede ver que hay muy pocas diferencias entre ambas imágenes, donde las únicas diferencias son algunos colores de métodos. En la figura con SPY sin Casper se pueden ver algunas zonas un poco mas oscuras que en SPY con Casper, como en la clase *#RBNumberLiteralToken*. Esto puede ser debido a que en Casper no se instrumentalizan muchos métodos que se consideran peligrosos y que pueden congelar el programa.

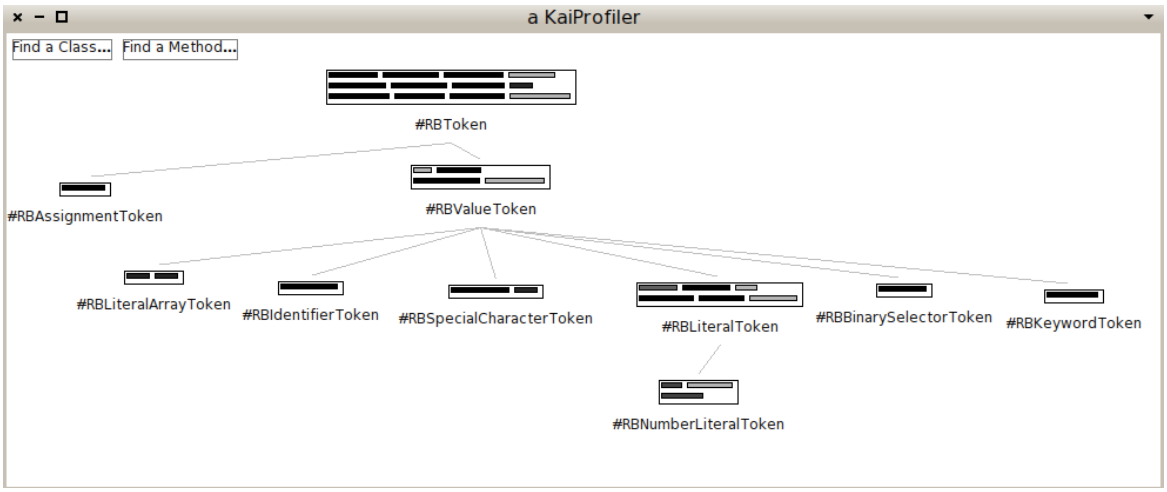

Figura 5.2: Sección KaiProfiler SPY con Casper

5.2.2. Benchmark

Una vez comprobado el funcionamiento, se midió el tiempo que toma realizar un profiling con SPY sin Casper y con Casper. Además también se implementó el funcionamiento de SPY con PHANtom. No se utilizó la implementación original de SPY con PHANtom [7], ya que no se pudo reproducir de la misma forma. Como alternativa, se implementó SPY con PHANtom de la misma forma que sale explicado en la sección 5.2, esto se logró ya que Casper y PHANtom comparten una sintaxis similar.

Para medir el tiempo se tomó el siguiente bloque de código:

- **[RBFormatterTests suite run]** Ejecuta los test de las clases responsables de formatear el código, es el mismo bloque que se usa para los ejemplos de la sección 5.2.1.

Se tomaron dos tiempos diferentes. El tiempo de instalación de la instrumentalización y el tiempo de ejecución del bloque de código una vez instrumentalizado. En la tabla 5.1 se pueden ver los resultados. La diferencia más grande entre SPY original con las implementaciones con aspectos es claramente el tiempo de instalación. Esto es esperable

porque Casper vuelve a compilar todos los métodos que son instrumentalizados, en este caso, alrededor de 5000 métodos. En PHANtom [7] usan una optimización para reducir este tiempo de gran forma, sin embargo en este trabajo se desea comparar los tiempos de ejecución de PHANtom y Casper, por lo tanto el análisis se enfocará más en ese resultado.

El tiempo de ejecución de SPY original es el más rápido de las tres implementaciones. Luego, casi el doble más lento, viene la implementación con Casper. Finalmente el tiempo de ejecución de la implementación con PHANtom es casi treinta veces más lento.

	SPY original	**Con Casper**	**Con PHANtom**
Instalación	0.706 sec	23.66 min	24.31 min
Ejecución	29.4 sec	57.6 sec	27.82 min

Tabla 5.1: Comparación de tiempo entre SPY sin Casper y con Casper

5.3. Discusión de Resultados

En la sección 5.2.1 se vio como fue posible usar Casper de forma exitosa para implementar la instrumentalización que realiza SPY. También logramos realizar una comparación de ambas implementación.

Una de las ventajas de usar Casper o PHANtom es la modularización que viene gracias al usar un lenguaje orientado a aspectos. SPY originalmente cambia los CompiledMethod que necesita instrumentalizar con CompiledMethod generados a través de un template. Esto hace que sea muy difícil llevar la idea de SPY a otro lenguaje, ya que usa una funcionalidad muy especifica de Pharo Smalltalk. Al usar Casper o PHANtom, se podría portar de forma más sencilla SPY a otros lenguajes que tengan una extensión orientada a aspectos, como lo es Java con AspectJ.

Una de las grandes desventajas es el tiempo que demora la implementación de SPY con Casper sobre la implementación original, sobre todo por la forma en como se instalan en el sistema para poder instrumentalizar. Para compensar, los tiempos de ejecución de ambas formas son prácticamente los mismos. Esto es debido a que Casper nace como una forma de optimizar la ejecución del programa, aunque esto sea subir mucho el tiempo de compilación. SPY logra tiempos de ejecución mejores que con su modularización con Casper, debido al manejo interno de los CompiledMethod. Por todo esto, Casper no es una buena solución para un framework como SPY, ya que Casper es pensado para que el programa se compile una sola vez durante el flujo normal del programa, pero en SPY la instrumentalización hace que se tenga que compilar nuevamente las clases del sistema para diferentes tipos de programas que se quieran instrumentalizar.

Casper vs PHANtom

Claramente los resultados muestran como Casper es más rápido en la instalación y en la ejecución de la instrumentalización. Sin embargo cabe destacar que para instrumentalizar con Casper se uso una optimización en la instalación de la instrumentalización. La optimización consiste en no instalar los aspectos por separado, como se hace con PHANtom, sino instalarlos todos juntos, esto permite que no haya una etapa de desinstalación de los aspectos cada vez que se instalan por separado. En la implementación original de la instrumentalización con PHANtom [7], se logra una instalación mucho más eficiente. Por lo tanto, aunque en este experimento Casper sea más rápido que PHANtom en la instalación de la instrumentalización, se sabe que PHANtom se puede optimizar para dar un tiempo de instalación mucho menor, del orden de tan solo algunos segundos.

Sin embargo, lo que no se pudo conseguir usando la implementación presentada en este trabajo o la implementación original, fue un tiempo de ejecución eficiente. PHANtom utiliza methods wrappers para instrumentalizar los métodos [6], lo que hace que la ejecución de la instrumentalización sea mucho más lento que la ejecución con Casper, pasando de casi una media hora con PHANtom a tan solo un minuto con Casper. En resumen, para una misma implementación, sin utilizar herramientas de optimización propias de la implementación del lenguaje orientado a aspectos, Casper es más rápido en la ejecución de los programas, y en la instalación tienen tiempos muy parecidos.

5.4. Resumen

En esta sección se presentó un caso de estudio, que es usar Casper en la herramienta de profiling SPY.

Primero se presentó SPY en la sección 5.1 y la forma en como se pueden realizar profiling gracias a la estructura que tiene SPY.

Luego, en la sección 5.2, se mostró como se modificó SPY para que soporte la instrumentalización con Casper, además se mostró los métodos necesarios para lograr que SPY funcione con Casper.

En la sección 5.2.1 se mostró como la implementación de Casper permitió correr de forma exitosa KaiProfiler. Además se mostró las diferencias de los tiempos de ejecución entre la instrumentalización normal de SPY versus la instrumentalización de SPY con Casper.

Finalmente se discutieron los resultados obtenidos por el experimento de modificar SPY en la sección 5.3

Capítulo 6

Conclusión y Trabajo Futuro

6.1. Conclusión

En este trabajo se presentó cómo se logró desarrollar e implementar Casper, un lenguaje orientado a aspectos que funciona sobre Pharo Smalltalk. Casper se inspira en las funcionalidades básicas de otros lenguajes de aspectos. Además, Casper toma la sintaxis del lenguaje de aspectos sobre Pharo Smalltalk, PHANtom, y mejora la especificación de los modificadores de clase. Casper usa el IDE Pharo Smalltalk para crear los métodos y variables que se desean instalar en otra clase. El IDE verifica la correcta sintaxis de los método que se instalarán con los modificadores de clase. Con esto Casper ayuda al programador a que no cometa errores de sintaxis en la escritura del método, no como en PHANtom que el método es representado en un string. Otra nueva funcionalidad de Casper es que se pueden especificar join points con los pragmas de un método.

Con la implementación de Casper se busca solucionar básicamente dos problemas: la dificultad de entender lo que sucede con un programa que ha sido modificado con aspectos y el rendimiento del programa. Casper ataca ambos problemas cambiando el código fuente de los métodos que son capturados por los aspectos. Casper luego compila el comportamiento que es generado por los aspectos en otros métodos del sistema. El compilar el comportamiento en un método en Pharo Smalltalk hace que este método sea visible por el usuario en el IDE. Además, con la compilación del comportamiento de los aspectos, Casper busca equiparar el tiempo de ejecución del programa como si no usara aspectos, pero manteniendo toda la modularidad que permite una implementación orientada a aspectos.

Es importante hacer notar que hay métodos en el sistema que al recompilarlos con cualquier otra funcionalidad que provea un aspecto puede congelar el sistema. Para evitar esto, por defecto Casper no captura los métodos que son clave para el correcto funcionamiento del sistema.

Casper tiene una sintaxis simple de usar gracias a que se basa en la sintaxis de

PHANtom. Además, gracias a que las sintaxis son similares, Casper puede utilizar la batería de tests construidos para PHANtom. Para las funcionalidades que tienen diferente sintaxis se reescribieron los tests de PHANtom para que soporten la nueva sintaxis de Casper. Las nuevas funcionalidades de Casper se crearon con sus propios tests para asegurar su correcto funcionamiento.

La validación de Casper fue reimplementar SPY. SPY es un framework que analiza dinámicamente otros programas. Se hicieron pruebas donde se comparó el comportamiento de SPY tanto en su versión original, como con la versión reimplementada con Casper. El usar Casper en SPY se logra un mejor nivel de modularización de SPY. SPY en su versión original se encuentra muy acoplado con el lenguaje sobre el que esta construido, en cambio, SPY en su versión con Casper solo depende de las funcionalidades del lenguaje orientado a aspectos, las cuales son transversales para cualquier lenguaje orientado a aspectos. Gracias a esto se puede portar la metodología de SPY a otro lenguaje que tenga soporte con aspectos. Sin embargo, la instalación de Casper sobre SPY original toma demasiado tiempo. SPY originalmente toma solo unos pocos segundos, en cambio SPY modularizado con Casper puede llegar a tardar 20 minutos. Además, usando Casper, SPY no es significativamente más rápido en tiempo de ejecución. Esto último gracias al manejo de los métodos que posee SPY, que logra optimizar el tiempo de ejecución de la instrumentalización.

Junto con la validación de Casper al reimplementar SPY, también se reimplementó SPY usando PHANtom. Las benchmarks presentados fueron favorables para Casper. Con la implementación presentada se muestre que Casper es tan solo un poco más rápido que PHANtom en la instalación de la instrumentalización, aunque el tiempo de diferencia es tan pequeño que se puede adjudicar a la capacidad del computador en ese momento. Por el otro lado, el tiempo de ejecución de la instrumentalización con Casper es mucho menor que con PHANtom, donde Casper demoró tan solo 1 minutos y PHANtom llegó casi a los 30 minutos. Esto se debe a que la implementación de Casper compila el código fuente capturado por los join points shadows, en cambio PHANtom interpreta los cambios al nivel meta del lenguaje.

Casper no está pensado para programas que requieren constante instalación de los aspectos dentro de la ejecución del programa. Casper está pensado para ser ejecutado una vez al comienzo de todo programa, así se logra reducir los tiempos de ejecución del programa, gracias a que Casper es compilado en los métodos originales.

6.2. Trabajo Futuro

En AspectJ y en PHANtom, cuando dos o más aspectos ejecutan un advice en un join point, es posible declarar el orden de los aspectos. Esta funcionalidad es deseable en muchos lenguajes de aspectos incluido Casper, que por factores de tiempo no se alcanzó a desarrollar y sería ideal tenerla en una siguiente versión.

El costo de instalación de los aspectos es muy alto. Esto sucede ya que en Casper,

cuando un aspecto se instala en un join point shadow, el método que representa este join point shadow se recompila con las nuevas funcionalidades. Pharo Smalltalk posee propiedades que permiten la metaprogramación, como es la reflexión y la reificación del sistema. Sería bueno usar estas propiedades, parecido a como lo hace SPY [3], para así bajar los costos de la instalación de los aspectos, tener el código explícito en el sistema y no aumentar el tiempo en la ejecución del programa. SPY no recompila los métodos que se instrumentalizan, lo que hace SPY es tomar el método instrumentalizado y lo reemplaza con un nuevo CompiledMethod que reifica el mensaje enviado e invoca la instancia de MethodSpy asociada. En el futuro Casper podría usar esta estrategia para reducir los tiempos de instalación de los aspectos, cambiando los CompiledMethod por uno nuevo que reifica las funcionalidades que instala el aspecto en el join point shadow.

El parsing de las clases y métodos del sistema con expresiones regulares en los pointcuts es un proceso costoso. Ésta también es una de las razones por la que el proceso de instalación es lento. También el proceso de composición de pointcuts puede llegar a ser muy lento dependiendo de los join points que matchean los pointcuts por separado. Se puede investigar como optimizar en el futuro los procesos de matching de clases y de la composición de pointcuts.

Una forma que se hizo en Casper para dejar el código explícito fue compilarlo junto con el método original para mostrar los cambios en el IDE de Pharo Smalltalk. Es posible cambiar el IDE para que muestre más información. Podría ser interesante mostrar los aspectos de otra forma, modificando el IDE donde se muestran los métodos. Con esto además podríamos mostrar la información que realmente le interesa al usuario y no cosas como el sector de pre-procesamiento (ver sección 4.3).

Cuando un nuevo aspecto es instalado, se desinstalan todos los aspectos que estaban antes instalados, luego se regenera toda la información de los aspectos para poder instalar la nueva información que se genera con el nuevo aspecto. Este proceso puede ser optimizado, solamente regenerando la información que nuevo aspecto modifica, sin tener que desinstalar toda la información antigua, solo aquella que cambia.

Aunque Casper se crea para ayudar al usuario, este lenguaje no está integrado con las herramientas de depuración existentes en Pharo Smalltalk. Cuando se produce un error, las herramientas de depuración muestran las referencias a la infraestructura interna de Casper. El usuario aún puede depurar el código que se ha generado, pero se muestra mucha más información que al mismo usuario quizás no le sea fácil de ver para depurar el programa. Es interesante manejar lo que el usuario puede ver para ayudarle a depurar un código que ha sido modificado por Casper.

Junto con lo anterior, en este trabajo no se muestran pruebas con usuario para saber si realmente Casper logra ser más entendible que PHANtom. Para esto se propone realizar pruebas finales con usuarios. Las pruebas que se proponen son modularizar un programa con aspectos e implementar uno nuevo también con aspectos. Ambas pruebas se dividirán entre usuarios que programen con Casper y otros con PHANtom. Al final de la prueba se les consultará a los usuarios cual lenguaje fue más fácil de utilizar para ellos y se medirá el tiempo que tarden en realizar ambas pruebas.

Finalmente Casper busca ser un paso fundamental para la gente que no sabe o tiene miedo a programar en un lenguaje orientado a aspectos. Por esto es importante dar mayores facilidades al usuario, traducidas en nuevas funcionalidades como: generador de pointcuts donde matchean solo los métodos que el usuario selecciona; muestra visual de donde los aspectos están matcheando en el sistema, algo como Aspectmaps [5]; entre muchas otras funcionalidades que los mismos usuarios de un lenguaje orientado a aspectos encuentren necesarias para que todo sea mucho más fácil para ellos mismos.

Bibliografía

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

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[3] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Computer Languages, Systems Structures*, 38(1):16 – 28, 2012.

[4] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207 – 239, 2006. Special issue on foundations of aspect-oriented programming.

[5] J. Fabry, A. Kellens, and S. Ducasse. Aspectmaps: A scalable visualization of join point shadows. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 121 –130, june 2011.

[6] Johan Fabry and Daniel Galdames. PHANtom: a modern aspect language for Pharo Smalltalk. *Software—Practice and Experience*, 2012. To Appear.

[7] Daniel Andrés Galdames Grünberg. Diseño e implementación de PHANtom, un lenguaje de aspectos para Pharo Smalltalk - Memoria para optar al título de Ingenierio Civil en Computación, Septiembre 2011.

[8] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM.

[9] H. Rajan and K.J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 59 – 68, may 2005.

[10] Éric Tanter, Nicolas Tabareau, and Rémi Douence. Taming aspects with membranes. In *Proceedings of the eleventh workshop on Foundations of Aspect-Oriented Languages*, FOAL '12, pages 3–8, New York, NY, USA, 2012. ACM.