UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACION

"IMPLEMENTACION DE CONTEXT-AWARE ASPECTS EN REFLEX Y EVALUACION EN UNA APLICACION CONTEXT-AWARE"

MEMORIA PARA OPTAR AL TÍTULO
DE INGENIERO CIVIL EN COMPUTACIÓN

ALEXIS HERRERA ORDENES

PROFESOR GUIA:
ERIC PIERRE TANTER

MIEMBROS DE LA COMISION:
LUIS MATEU BRULE
NANCY HITSCHFELD KAHLER

SANTIAGO DE CHILE
2 0 0 7

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACION

"IMPLEMENTACION DE CONTEXT-AWARE ASPECTS EN REFLEX Y EVALUACION EN UNA APLICACION CONTEXT-AWARE"

MEMORIA PARA OPTAR AL TÍTULO
DE INGENIERO CIVIL EN COMPUTACIÓN

ALEXIS HERRERA ORDENES

PROFESOR GUIA:
ERIC PIERRE TANTER

MIEMBROS DE LA COMISION:
LUIS MATEU BRULE
NANCY HITSCHFELD KAHLER

SANTIAGO DE CHILE
2 0 0 7

RESUMEN

Cada vez es más necesario contar con aplicaciones capaces de autoadaptarse y modificar su comportamiento de acuerdo a información de su contexto que ellas mismas recopilan e interpretan. Las soluciones propuestas tradicionalmente para implementar dichas aplicaciones son muy limitadas, especialmente porque no logran separar su comportamiento básico de su capacidad de adaptarse a distintos contextos. Una solución resulta de utilizar trozos de código condicionales (*if-else*) relacionados con contextos en todas las partes donde sea necesario. Otro enfoque resulta de utilizar el polimorfismo de la orientación a objetos para permitir seleccionar entre varios comportamientos. Esto se consigue utilizando el patrón de diseño *Strategy*, que permite separar los atributos de los métodos en un objeto.

Ambas soluciones propuestas padecen los mismos problemas, ya que no logran separar la lógica de adaptación de la lógica básica de funcionamiento. Estas decisiones de diseño hacen que la preocupación por el contexto esté dispersa (*scattering*) por varias partes del código que se va generando, obteniendo como resultado un código muy enredado(*tangling)* que es muy difícil desarrollar y mantener. Los problemas de este tipo, que presentan clara dificultad de ser encapsulados, tienen su origen en que atraviesan (*crosscut*) toda la funcionalidad básica del sistema, de ahí que sean llamados *intereses transversales* (*crosscutting concerns*).

Para implementar aplicaciones dependientes de su contexto, resulta necesario implementar intereses transversales en forma modular, problema que justamente resuelve la programación orientada a aspectos (AOP). AOP es un paradigma cuyo objetivo fundamental es lograr la implementación modular de intereses transversales dentro de aplicaciones, logrando una efectiva separación de intereses (*SoC: Separation of Concerns*).

Contando con: 1. los requisitos que debería cumplir una implementación de aspectos que se aplican bajo ciertas condiciones contextuales, 2. un framework de AOP, como es Reflex y 3. un framework de implementación de aplicaciones context-aware, como es WildCAT; es necesario integrar todas estas partes en un sólo framework que haga transparente toda la creación y manejo de distintos contextos, logrando efectiva separación de intereses.

El objetivo general de este trabajo consiste en extender el framework Reflex para soportar *context-aware aspects*. Este nuevo framework permitirá integrar sensores de contexto implementados con *WildCAT* para el desarrollo de una aplicación modelo que requiera aspectos *context-aware*.

La aplicación del framework implementado a una aplicación context-aware es el resultado más importante. En su desarrollo no se aprecian solamente la implementación de comportamientos adaptables a distintos contextos, sino que también se aprecia que dicha implementación se encuentra completamente separada de la lógica de negocio que tiene la aplicación. Otro aspecto importante está en que el framework fue construido de manera tal que su utilización es simple y completamente inserta dentro de Reflex. Para lograr esto fue necesario realizar aportes al *API* de Reflex, de manera tal que facilitara su utilización desde el framework.

La extensión más importante a este trabajo se centra en la posibilidad de contar con varios contextos que podrían estar vigentes al mismo tiempo y en un mismo lugar. Dado que componer contextos implicaría componer aspectos y que en Reflex ya existe la posibilidad de componer estos últimos (definiendo un orden y una jerarquía de aplicación) se podría tomar como ejemplo dicha implementación y a partir de ella definir un sistema basado en reglas que permita asignar una jerarquía y orden de utilización de los contextos.

Agradezco a todos quienes
hicieron posible este sueño,
especialmente a mi familia,
mi polola y mis amigos.
Y a Dios sea el Honor y la Gloria,
por los siglos de los siglos. Amén.

Índice general

Capítulo I

Introducción

En la actualidad resulta de mucha utilidad contar con sistemas auto-adaptables[8] y autónomos[7], capaces de autogestionarse y de tomar decisiones acerca de cómo tener un mejor desempeño. Esto conlleva la capacidad de tener un claro conocimiento del entorno de ejecución del sistema (*runtime environment*) y de su evolución[2]. Para estos sistemas resulta importante saber en qué condiciones está el contexto de ejecución para seguir funcionando de una manera o para cambiar el funcionamiento a una nueva situación, por ejemplo, en el caso de un dispositivo móvil resulta muy útil conocer qué tipo de red está disponible para poder comunicarse a través de ella, o qué dispositivos están dentro del alcance para interactuar con el usuario.

Este tipo de aplicaciones tienen que ser capaces de [2]:

- Explorar su mundo para descubrir bajo qué circunstancias están operando. Puede ser con respecto a recursos locales de hardware (memoria, disco, dispositivos E/S,etc) en que opera, datos de las redes disponibles, información proveniente de sensores especializados (temperatura, ubicación, etc) o datos del usuario (preferencias, características y estado actual)
- A partir de la información recuperada, deben ser capaces de tomar decisiones al respecto, como cambio de comportamiento, cambio de estado o gatillando eventos.
- Además, las aplicaciones deben ser notificadas si ocurre algún evento que pueda ser importante de analizar o tener en cuenta. Estos eventos pueden ser el corte de la red, caída de un servidor que utilize, etc.

Una aplicación que se comporte de esta manera se llama una aplicación *context-aware*. Si deseáramos implementar esta tecnología necesitaríamos dos cosas:

1. Otorgarles la capacidad de percibir su entorno.
2. Darles la flexibilidad necesaria para adaptarse al entorno.

Para enfrentar lo planteado en 1, se han implementado diversos *frameworks* que permiten conocer el estado del entorno de ejecución. Estos pueden otorgar toda la información necesaria para que la aplicación sea conciente de su entorno. Wildcat[2] es uno de ellos, es un framework Java extensible pensado para facilitar la creación de aplicaciones context-aware. WildCat provee un simple pero poderoso modelo dinámico para representar el contexto de ejecución de una aplicación. WildCAT se encuentra integrado con Linux, y consiste básicamente en una interfaz que permite ver el directorio `\proc` del sistema de archivos.

Para el punto 2 sería necesario agregar comportamiento a la aplicación con la introducción de preguntas condicionales (*if*), que sean capaces de dirigir el flujo de la aplicación frente a distintos contextos. Otra solución sería utilizar un patrón de diseño *Strategy*[4], que por medio del polimorfismo logra separar el comportamiento y los atributos de los objetos para que éste pueda poseer varios comportamientos y no sólo uno, pudiendo elegir el necesario para enfrentar el contexto actual.

En ambos casos estamos frente a soluciones invasivas, que implican la modificación de código y más grave aún introducen intereses transversales que generan problemas de separación de intereses (*SoC*), ya que habrá que agregar la misma preocupación en distintas partes. Esto hará difícil reutilizar código dada la especialización a la que lo sometemos y además introducirá un interés transversal (*crosscutting concern*[6]) que atraviesa toda la aplicación respecto a que deberá manejar su dependencia del contexto de ejecución en distintas partes.

1. Motivación

La Programación Orientada a Aspectos (AOP) ha sido una alternativa de solución a la *SoC*. La introducción de aspectos para la implementación modular de intereses transversales a la aplicación otorga una clara separación de intereses que es imposible lograr a través de una implementación orientada a objetos[3]. Por lo mismo, AOP se focaliza en la modularización y encapsulamiento de intereses transversales (*crosscutting concerns*).

Una buena implementación en AOP busca encapsular estos intereses transversales introduciendo un nuevo concepto llamado aspecto. Un aspecto puede cambiar el comportamiento del código por medio de la introducción de nuevos comportamientos (*advice*) en alguna parte del flujo del programa (*join point*). Un conjunto de *join points* conforman un punto de corte (*pointcut*) del programa y se refiere directamente a la transversalidad de intereses[6].

Dado que al implementar aplicaciones *context-aware* aparecen intereses transversales involucrados, es posible plantear una solución con un enfoque AOP. Esta no será invasiva, ya que utilizará aspectos que sean *context-aware*. De aquí surge el término *context-aware aspects*[12], que es la fusión de todas las ideas AOP con la necesidad de contar con aplicaciones *context-aware*.

2. Objetivo General

En [12] se ha planteado el diseño de un framework para *context-aware aspects*. Esto lo logra agregando a *Reflex*[13], un framework creado para la experimentación en AOP, el soporte de *context-aware*. Este trabajo falta cerrarlo con una implementación y experimentos que sirvan para mostrar su funcionamiento real y comparar los beneficios obtenidos.

El objetivo general de este trabajo consiste en extender el framework Reflex para soportar *context-aware aspects*, luego integrar *WildCAT* para definir contextos dependientes del ambiente de ejecución, concluyendo con el desarrollo de una aplicación modelo que requiera aspectos *context-aware*.

3. Objetivos Específicos

Los objetivos específicos de este trabajo son:

1. Implementar un framework de *context-aware aspects* en Reflex.
2. Definir una interfaz que permita integrar WildCat y Reflex.

3. Diseñar y desarrollar una aplicación real, que necesite ser *context-aware*, utilizando la plataforma *Reflex-WildCAT* de modo de ilustrar los beneficios que representan los *context-aware aspects*.

4. Comparar la solución obtenida con otras alternativas de desarrollo, de tal manera de poder validar los resultados obtenidos.

4. Cómo Leer esta Tesis

Esta tesis expone la concreción de la implementación del framework para context-aware aspects en Reflex definido en [12].

Para facilitar la lectura, evitando entorpecer el desarrollo de las ideas expuestas, en la capítulo II se explica el caso de ejemplo que será utilizado en todos los capítulos siguientes. El caso del *e-Shop*, como lo llamaremos, es capaz de ilustrar todos los posibles escenarios contra los que el framework será probado.

En el capítulo III serán mostrados los conceptos básicos que hay que tener en cuenta para entender y contextualizar el problema y la solución implementada para los *context-aware aspects*.

En el capítulo IV está el núcleo de esta tesis. En él se muestra como efectivamente fue implementado el framework, indicando de forma detallada cada parte y las decisiones que fueron tomadas a la hora de aterrizar el diseño planteado en [12].

Para cerrar el caso de ejemplo, en el capítulo V es presentada la aplicación del framework sobre dicho caso. Mostrando los principales escenarios a los que se puede enfrentar dependiendo de los contextos y bajo que condiciones se aplican.

Finaliza la tesis con el capítulo VI que muestra las conclusiones obtenida del trabajo realizado. No se enmarcan tan sólo en el ámbito de todo lo que fue posible de lograr, sino que también destacando los puntos débiles y las perspectivas de evolución de este mismo trabajo.

Capítulo II

Caso de Estudio

Para ilustrar el trabajo desarrollado presentaremos a continuación un caso de estudio o ejemplo motivador. A lo largo de toda la tesis aparecerá este mismo ejemplo visto desde diferentes ángulos y de diversas implementaciones.

En la sección 1 de este capítulo se presentará el funcionamiento del *e-Shop*. Luego, en la sección 2, se detallará un breve análisis de las funciones que están involucradas, ahondando en tres puntos que son importantes como son el manejo de promociones, la aplicación de descuentos y los tipos de clientes.

1. E-Shop en Acción

Este ejemplo trata de la utilización de una tienda, llamada *e-Shop*. El *e-Shop* funciona como una tienda convencional, ya sea virtual (Sitio web, aplicación de ventas, etc) o real (supermercado, bazar, etc). Por esto mismo los usuarios podrán acceder al *e-Shop* en forma directa o a través de un Web Service. En él es posible comprar toda un lista de productos y después pagar el monto total de la compra. Posee un *shopping cart* (carrito de compras) donde los productos son agregados, para que al momento de pagar (*check out*) se pueda calcular el monto total como la suma del valor de cada producto.

Para cumplir la tarea para la cual el *e-Shop* fue pensado, este tiene una serie de funciones que llevan a cabo el ciclo de vida del *shopping cart*:

1. Tomar *shopping cart*: obtiene un *shopping cart* nuevo, este comienza vacío.
2. Agregar producto: Coloca un nuevo producto en el *shopping cart*.
3. Eliminar producto: Quita un producto que fue agregado al *shopping cart*.
4. Pagar: Se calcula el total del monto de la compra como la suma del valor de todos los productos. Esta operación es conocida como *check out*.

Las funciones son pocas y muy sencillas, porque de hecho son las típicas que realiza cualquier usuario de un sitio web de ventas o en un supermercado. En la figura II.1 vemos el diagrama de casos de uso del *e-Shop* para dos tipos de usuarios: usuarios web y usuarios normales.

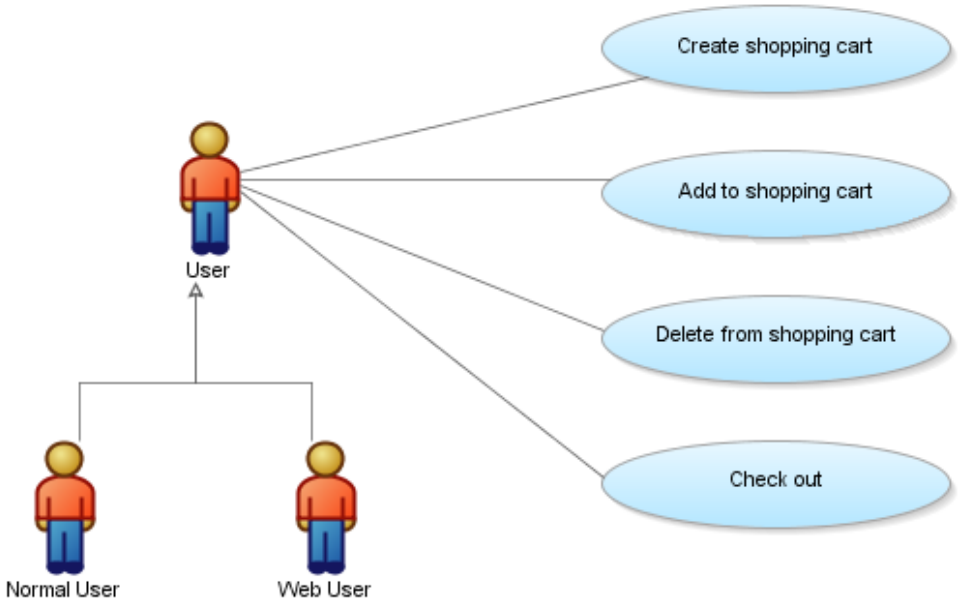

Figura II.1: Diagrama de Casos de Uso del e-Shop

2. Análisis de las Funcionalidades

El diagrama de casos de uso lleva inmediatamente a pensar en el diagrama de clases del *e-Shop* (figura II.2). El *shopping cart* posee dos atributos: `productList`, que contiene el detalle de cada producto comprado y `client` que identifica al cliente dueño del *shopping cart*. Los métodos principales son: `addProduct()`, `deleteProduct()` y `getAmount()`. Los dos primeros agregan y eliminan un producto a la `productList` respectivamente, mientras que el último es el encargado de calcular la suma del valor total de los productos en la `productList` y retornarlo como el monto de la compra.

Las otras clases dan completitud al caso de estudio. Ellas son: `Store`, `Client` y `Product`. Estas se encargan de identificar la tienda, los clientes y los productos, respectivamente.

Con esta definición dejamos contenido lo esencial del *e-Shop*. Es necesario destacar que otros elementos quedan fuera por dos motivos, agregan complejidad innecesaria al modelo o agregan otras preocupaciones que no hacen parte del núcleo del *shopping cart*.

Algunos elementos extra a considerar son las promociones, tipo de clientes y descuentos. A continuación detallamos cada uno de estos.

2.1. Manejo de Promociones

El *shopping cart* vive dentro de una tienda, en este caso, dentro del *e-Shop*. Por lo tanto, está sujeto a todos los fenómenos que en ella aparecen. Para el *shopping cart*, estos fenómenos aparecen ante él como contextos que pueden o no ser tomados en cuenta para modificar su comportamiento.

Ejemplo de estos fenómenos son las promociones. Los negocios de ventas en general venden todos los mismos productos. Por esto, las promociones son fundamentales porque son el medio por el cuál se diferencian unos de otros. Desde el punto de vista administrativo, una promoción no es simplemente un descuento, sino que posee una definición (qué promoción es) y un conjunto de

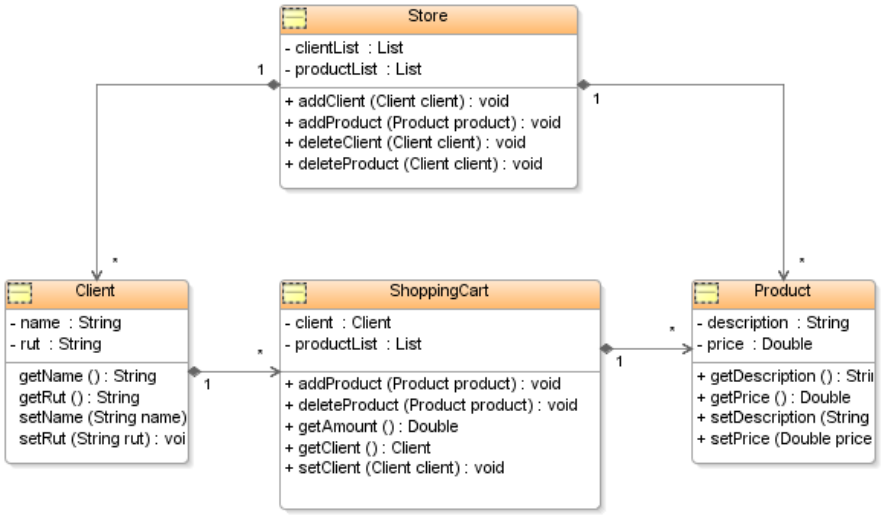

Figura II.2: Diagrama de Clases del e-Shop

restricciones (cuándo se aplica). Por ejemplo, es posible lanzar una promoción para todos los clientes que compren a través de un Web Service, aplicándoles un descuento de un 5 % sobre su compra (definición), la promoción es válida sólo al momento de hacer el *check out* (restricción).

Las promociones en si mismas son un problema porque implican tener que modificar código de manera tal que permita identificarlas y aplicarlas cuando corresponda sin equivocación.

2.2. Tipo de Cliente: Normal/Web Service

En el *e-Shop* existe la posibilidad de acceder de dos formas, a través de un web service o directamente. Esto define dos tipos de uso del *e-Shop*, ya que se podrá distinguir entre la utilización del *e-Shop* por un usuario web o por un usuario normal.

Como ya fue mencionado, una aplicación context-aware adapta su comportamiento a un contexto. En la sección anterior fue explicada una forma de lograr esto en el caso concreto de las promociones.

Si el cliente compra directamente o a través de un web service también define un contexto para el *e-Shop*. Como ya dijimos, es posible definir una promoción especial para los usuarios que prefieran comprar a través de la web.

Junto con los dos ejemplos ya mostrados es posible definir muchos otros casos de contextos que son posibles de definir para el *e-Shop*. Para este caso de estudio consideraremos los siguientes:

1. promoción para los clientes que compran a través de un web service.
2. promoción para los clientes que compran en una franja de tiempo.
3. promoción para los clientes que compran en un departamento específico de la tienda.
4. promoción para los clientes que compran cuando el servidor de ventas está con baja carga.

2.3. Aplicación de Descuentos

Como ya fue mencionado con anterioridad un contexto es tomado por una aplicación *context-aware* para que su comportamiento se adapte a él. Dado que una promoción es un contexto para el *e-Shop*, este tiene que modificar su comportamiento de alguna forma para adaptarse a ella. Esta modificación la logra a través del manejo de *descuentos*.

No es inmediata la decisión de donde agregar el manejo de descuentos. Es posible agregar un descuento a la lógica del `getAmount()` del *shopping cart*, o colocarlo como una etapa intermedia entre la generación de la factura (o boleta) y la llamada al `getAmount()`.

Existen diversas formas por las cuales un descuento se puede relacionar con una promoción. Por ejemplo, el descuento puede ocurrir cuando hay una promoción y (1)el cliente hace *check out*, (2) el cliente entra al *e-Shop* y el *shopping cart* es creado(3) es agregado un producto al *shopping cart*[12].

Las soluciones pueden ser variadas, lo importante es considerar que no es un tema menor y que existen formas mejores o peores de enfrentar su implementación.

3. Resumen

El *e-Shop* será el caso de estudio que servirá para ejemplificar todo el desarrollo realizado en la tesis.

Su implementación no está exenta de problemas, ya que temas como el manejo de contextos en los cuales puede estar el *e-Shop* (por ejemplo, las promociones) y definir la forma en que la aplicación modificará su comportamiento para adaptarse a los contextos (por ejemplo, el *e-Shop* modifica su comportamiento aplicando distintos descuentos) son desafíos que hay que tener en cuenta a la hora de llevar a la realidad sistemas similares al *e-Shop*.

Capítulo III

Aspectos y Contexto

Existe toda una serie de términos y definiciones que serán empleadas en la explicación de la implementación del framework de context-aware aspects. Muchos de ellos son comúnmente empleados, por lo que es necesario darles un alcance que permita comprender el sentido con que serán utilizados.

La definición de contexto es central dentro de la temática expuesta. Por esto, la primera sección se dedica completamente a ella. Luego, en la sección 2, esta definición es vista desde el punto de vista de la computación, mostrando un tipo específico de aplicaciones cuyo comportamiento depende del contexto en el cual se encuentran.

Las aplicaciones dependientes de contextos tienen requisitos que cumplir para que sean reconocidas como tales. Estos requisitos son expuestos en la sección 3.

En la sección 4 son detalladas dos de las alternativas de solución que comúnmente han sido utilizadas. Contraponiéndose a estas soluciones, en la sección 5 se encuentra explicada la base sobre la cual se construye la solución planteada en esta tesis. Son mostradas algunas nociones y los elementos fundamentales de la programación orientada a aspectos.

En la sección 6 son presentadas las dos herramientas que serán utilizadas para implementar el framework. La primera es Reflex, encargada de brindar una implementación de programación orientada a aspectos, y la segunda es WildCAT, framework que permite crear aplicaciones dependientes de contextos en forma simple.

Este capítulo es concluido mostrando la aspectos fundamentales que deberá contener el framework para que sea realmente una herramienta que permita resolver de forma estándar el problema del contexto.

1. Noción de Contexto

La palabra *contexto* es aplicada a distintos ámbitos y en formas muy diversas. A pesar de esto existe un rasgo común en todas estas acepciones: *contexto* siempre hace referencia a una *relación* entre una *entidad* y *elementos externos* a la misma.

Dentro del área de la computación, desde el punto de vista de las aplicaciones (*software*), la noción de contexto es entendida como el entorno (*environment*) dentro del cual una aplicación está ejecutándose. Esto ilustra la figura III.1, donde se realiza un paralelo en forma esquemática entre lo que significa un contexto en términos generales y la forma en que esta idea se concreta en una aplicación. Los contextos pueden ser muy variados. Un caso muy simple es el de otras aplicaciones que están ejecutándose al mismo tiempo, y que pueden relacionarse con la aplicación original a través de alguna interfaz conocida. Otro contexto para una aplicación puede ser el *hardware* que

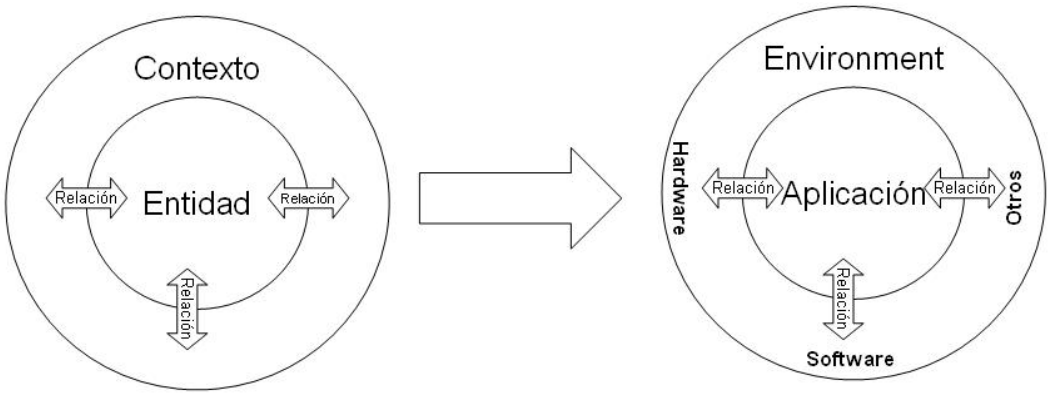

Figura III.1: Contextos en general y en una aplicación.

la contiene, su entorno físico donde se encuentra el hardware (ubicación, temperatura, etc.). Por último un contexto puede ser definido para una aplicación, o sea, es posible que los usuarios quieran colocar a la aplicación en un contexto específico para que esta se comporte de cierta forma.

El desafío entonces está en cómo mantener y propiciar la relación que existe entre los contextos y las aplicaciones, dado que ella determina la capacidad de adaptación de las herramientas a sus contextos

2. Aplicaciones Context-aware

Dado que las aplicaciones se encuentran insertas dentro de un contexto es posible definir relaciones entre ambos. Aquellas aplicaciones que tienen relaciones de dependencia con el contexto, y que son capaces de modificar su comportamiento de acuerdo a él, son llamadas aplicaciones *context-aware*[1] (dependientes del contexto).

Es necesario contar con este tipo de aplicaciones, porque el valor agregado que generan es mucho mayor que el de una aplicación sin comportamiento context-aware. Las aplicaciones *context-aware* tiene que ser capaces de:

- Explorar su mundo para descubrir bajo que circunstancias están operando.
- A partir de la información recuperada, deben ser capaces de tomar decisiones al respecto.
- Además, estas deben ser notificadas si ocurre algún evento que pueda ser importante de analizar o tener en cuenta.

Esas labores clásicamente eran otorgadas a usuarios, pero dado los crecientes avances tecnológicos, ya existen sensores capaces de proveer estos datos a un computador, y por lo tanto quedan accesibles a las aplicaciones context-aware.

Las aplicaciones context-aware son entonces capaces de tener un comportamiento más *inteligente*, manteniendo una relación dinámica con su contexto sin necesidad de otro agente.

Por las razones expuestas, el desarrollo y profundización de la temática del *contexto* ha ido de la mano del tema de la computación ubicua[5] (*ubiquitous computing*), del desarrollo de sistemas auto-adaptables[8] y de sistemas autónomos[7].

La noción de contexto también es muy amplia dentro del marco de la computación. Existen distintos enfoques que otorgan varias formas de concebir aplicaciones context-aware. Para [5] el

contexto es útil para permitir a las aplicaciones elegir distintas fuentes de datos, manejarlos en forma distinta, proveyendo de esta forma un soporte para escalabilidad, mobilidad y confiabilidad. Por otra parte, para [8] existe una desafío con respecto a la correctitud en la toma de decisiones que tienen que hacer los sistemas adaptables con respecto al mundo físico altamente dinámico al que pertenecen. Aquí maneja la idea de utilizar *sensores* (ya sean de hardware o software) que sirvan de entrada al sistema de toma de decisiones. Por último, en [7] aparece un elemento que ha acompañado al desarrollo de aplicaciones desde hace mucho tiempo, nos referimos a la interacción con el usuario. Esta interacción también es parte del contexto y nos muestra que el sólo hecho de tener interfaces con los humanos ya nos plantea en sí mismo un problema de contexto.

3. El Problema del Context-awareness

Para implementar aplicaciones context-aware necesitaríamos proveer dos funcionalidades básicas[2]:

1. Otorgarles la capacidad de percibir su entorno.
2. Darles la flexibilidad necesaria para adaptarse al entorno.

Hay que tener en cuenta que el desarrollo de estas funcionalidades va junto con las funcionalidades de la aplicación por si misma, o sea, de su lógica de negocio interna.

Agreguemos a la capacidad de percepción y adaptación una gestión de contextos. Estos podrían ser varios, estar disponibles o no, o aplicarse a distintas partes de la aplicación.

Para definir un contexto es importante tener en cuenta que este debe tener asociado un estado (*stateful*). El estado de un contexto tiene dos elementos. Por una parte tiene un estado interno (o parámetros del contexto), que indica por ejemplo en el caso de una promoción basada en una ventana de tiempo, cuales son las horas iniciales y finales de validez dentro del día. Este estado interno está directamente relacionado con la implementación del contexto. Por otro lado, tiene asociado un estado público (externo) accesible, que lo diferencia de otros contextos. Es deseable que este estado puede cambiar dinámicamente si es requerido.

Además, es deseable que los contextos se puedan componer(*composable*), es decir, a partir de algunos contextos básicos se puedan crear contextos más complejos. Por ejemplo, se podría crear un contexto que se aplicara si el shopping cart fuera creado a través de una web service en un cierto horario. Así mezclamos dos contextos, obteniendo uno nuevo.

Por último, los contextos tienen que ser parametrizables. Esto quiere decir que los contextos tienen que ser definidos genéricamente, para que dinámicamente se pueda modificar sus parámetros de funcionamiento.

4. Alternativas de Solución

Para los problemas antes nombrados existen varias posibilidades de solución. Obviamente algunas soluciones serán mejores que otras, dependiendo de la flexibilidad que otorguen, la facilidad para mantener las aplicaciones desarrolladas y el nivel de modularidad que permitan.

Existen diversos frameworks, herramientas o tecnologías que permiten capturar contextos, por lo tanto ya hay avances en esta área. Por otro lado, la lógica de la aplicación base es siempre un problema para el cual existen diversas alternativas que van de la mano de la Ingeniería de Software.

El problema que no tiene una clara solución es el de la adaptabilidad de las aplicaciones. ¿Cómo lograr que la aplicación modifique su comportamiento?. A continuación presentamos dos alternativas existentes.

4.1. Acercamiento IF-ELSE

Un acercamiento *naive* lleva a plantear una solución que maneje condiciones de contexto *(if)*. Como por ejemplo:

```
public Double getAmount(){
    \\ obtain amount from productList
    amount = ...
    ...
    discount = 0;
    if (inWebService()){
        discount = getDiscountWSUser();
    }
    else if (isCurrentlyInTimeFrameContext()){
        discount = getDiscountCurrentlyTimeFrame();
    }
    else if (wasCreatedInTimeFrameContext()){
        discount = getDiscountCreatedInTimeFrameContext();
    }
    else if (CPULoad()>2){
        discount = getDiscountCPULoad();
    }
    else if (userLocation().equals("Sport_Department")){
        discount = getDiscountInSportDepartment();
    }
    return amount*(1-discount);
}
```

Esta parecería una solución correcta y aplicable, pero bajo el supuesto que sea una aplicación restringida a esos contextos y que no fuese a ser sometida a cambios en el tiempo. Claramente, los posibles estados del contexto son estáticos, o sea, no se pueden agregar más estados sin modificar el código de la aplicación. El peor caso ocurriría si fuese necesario estar constantemente agregando nuevos contextos o nuevos parámetros al método, aquí empezarían los problemas típicos de la dispersión de código que produce esta solución, dado que habría que cambiar todas las llamadas al método `getDiscount*()`. Esto presenta serios problemas de reutilización y de evolución del código, dado que el cálculo de descuentos no se encuentra encapsulado en una sola parte, sino que por el contrario hay cinco funciones a cargo de dicha labor. Además no aprovecha paradigmas como la orientación a objetos.

Respecto de la implementación misma, se puede observar que el problema de la adaptación del `getAmount()` al contexto está absolutamente implementado a mano, con *if/else*. Esto tiene claros problemas de separación de intereses (*separation of concerns - SoC*) ya que, inserto dentro del código que se encarga de obtener el monto de la compra, hay toda una serie de líneas que agregan validaciones de contexto necesarias para que la operación pueda ser *context-aware*, pero que no tienen nada que ver con la lógica interna del método `getAmount()` que sólo debería recorrer la `productList` para calcular `amount`.

4.2. Acercamiento con Patrón Strategy

Un segundo acercamiento más interesante resulta de utilizar el polimorfismo de la orientación objetos, a través de una implementación realizada con el patrón de diseño *Strategy*[4]. Este patrón busca separar el comportamiento de los objetos para que este pueda poseer varios comportamientos y

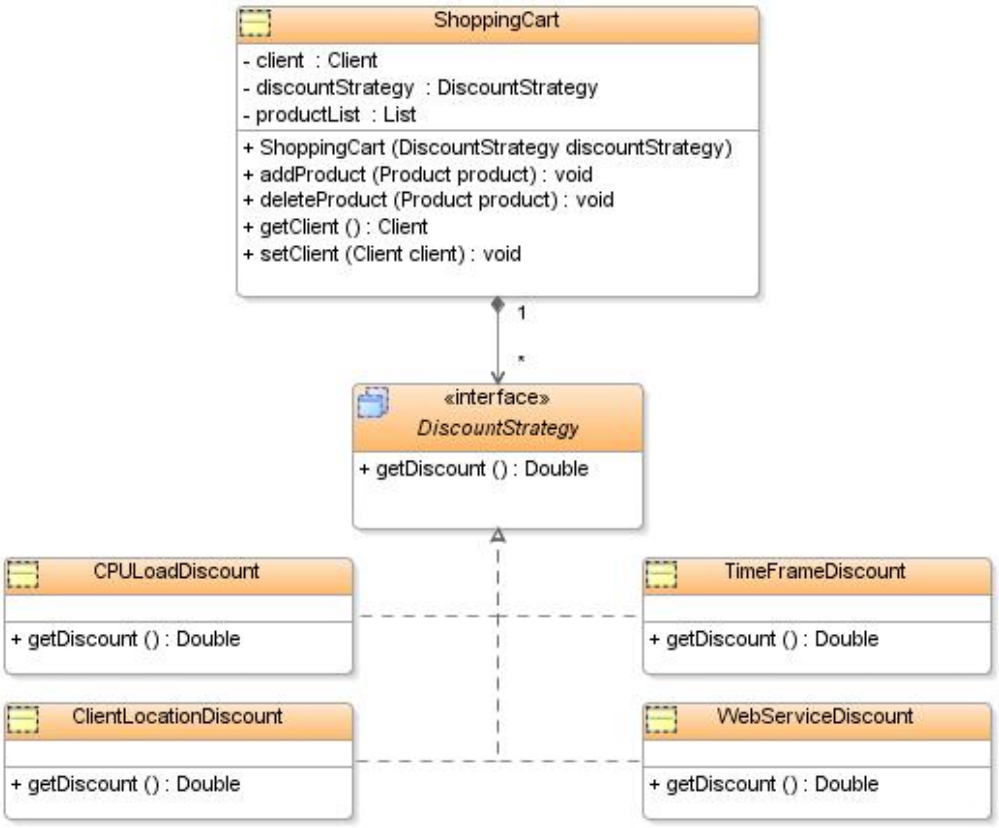

Figura III.2: Diagrama de Clases siguiendo Patrón de Diseño Strategy

no sólo uno, pudiendo elegir el necesario para enfrentar un contexto en particular. Retomando el caso de ejemplo, según la figura II.2, la clase `Client` tendría que realizar compras en el `ShoppingCart`, y al final hacer un `getAmount()` para pagar.

Al realizar el `getAmount()` es necesario obtener el descuento de acuerdo a la promoción (contexto) que esté vigente. Para cada una de las promociones habrá una forma de obtener el descuento correspondiente.

Siguiendo la implementación *Strategy* de este caso tenemos una clase `ShoppingCart` idéntica a la mostrada en la figura II.2, salvo por su atributo `discountStrategy` y sus correspondientes métodos `get` y `set`.Este atributo pertenece a la interfaz `DiscountStrategy`, cuyo único objetivo es dar acceso a la implementación del método que se encargará de obtener el descuento. Las implementaciones de `DiscountStrategy` son realizadas en las clases `WebServiceDiscount`, `TimeFrameDiscount`, `CPULoadDiscount` y `SportDepartmentDiscount`, donde cada una se corresponde con los distintos contextos que pueden ocurrir. Un diagrama de clases de este ejemplo aparece en la Figura III.2.

Al iniciar el `ShoppingCart` en su constructor, se define cual será el comportamiento a utilizar. Este puede ser modificado utilizando su respectivo método `set`, logrando de esta forma la capacidad de adaptarse modificando su comportamiento. Cuando se realice un *check out*, será llamado el `getAmount` y entonces se obtendrá el descuento con `discountStrategy.getDiscount()`. Por el polimorfismo sabemos que esta llamada irá directamente a la implementación del `DiscountStrategy` que haya sido asignada al atributo `discountStrategy`.

Esta implementación, si bien es cierto es mucho más elaborada que la anterior, dado que utiliza la orientación a objetos, sirviéndose del polimorfismo para implementar la capacidad de obtener varios comportamientos, no logra dar una solución que logre implementar intereses transversales en forma modular. Esto dado que de todas formas está inserta dentro de las preocupaciones del `ShoppingCart` el conocer quien es el que implementará su `discountStrategy`.

Comparando las implementaciones con *if-else* y el patrón *Strategy* se puede apreciar que en ambos casos se está realizando lo mismo, sólo que en uno se realiza en el mismo objeto y en el otro parte de esta lógica es traspasada a otros objetos. Estas soluciones tratan los intereses transversales en forma distinta, pero no logran modularizarlos, ya que el interés de aplicar los descuentos es dependiente de otros intereses y no está concentrado en una parte aislada y separada de otros.

5. Aspect Oriented Programming

5.1. Antecedentes

La programación orientada a objetos (OOP) ha sido presentada como un paradigma que ayuda en el desarrollo de aplicaciones complejas, para que sean flexibles y mantenibles. Para lograr estos objetivos se basa fuertemente en la modularización.

Durante el ciclo de desarrollo de aplicaciones simples o sistemas más complejos, resulta una buena idea utilizar OOP. Lamentablemente se pueden encontrar muchos problemas donde las técnicas propuestas por OOP no son suficientes para ayudar a tomar importantes decisiones de diseño que el programador debe implementar.

Rápidamente aparecen requerimientos que no pueden ser descompuestos en comportamientos únicos, quedan mezclados con otras funcionalidades completamente distintas. OOP tiene claras dificultades para *localizar* las *preocupaciones* que están referidas a restricciones globales del sistema o de comportamientos muy recurrentes. Tiene problemas para separar apropiadamente distintos intereses en el código[3]. Este problema es conocido como *crosscutting concerns*, y a los puntos esenciales a los que están dirigidas las decisiones de diseño que causan estos problemas se les llama *aspectos*.

La *programación orientada a aspectos*[6] (AOP) ha sido una alternativa de solución a *SoC*. La introducción de aspectos para la implementación modular de intereses transversales a la aplicación otorga una clara separación de intereses que es imposible lograr a través de una implementación orientada a objetos. Por lo mismo, AOP se focaliza en la modularización y encapsulamiento de intereses transversales (*cross-cutting concerns*).

Un aspecto puede cambiar el comportamiento del código por medio de la introducción de nuevos comportamientos (*advice*) en alguna parte del flujo del programa (*join point*). Un conjunto de join points conforman un punto de corte (*pointcut*) del programa y se refiere directamente a la transversalidad de intereses. Como fue presentado en el ejemplo con *if-else*, un descuento es un aspecto que es dependiente de los distintos contextos promocionales, por esta razón presentamos un ejemplo básico de implementación de un aspecto de descuento en *AspectJ*[1]:

```
aspect Discount{
    double rate = 0.1;
    pointcut amount(): execution(double ShoppingCart.getAmount());
    double around(): amount(){ return proceed()*(1-rate);
}
```

Este ejemplo muestra para un aspecto de descuento el pointcut que será interceptado (llamada a `ShoppingCart.getAmount()`), el método que realizará las modificaciones al comportamiento (`amount()`) y la definición de este método. El orden de ejecución es:

- Al llamar al `ShoppingCart.getAmount()` en la aplicación base, el aspecto toma el control.
- Reemplaza (*around*) la llamada al método `ShoppingCart.getAmount()` por la función `amount()` definida en el aspecto.
- Ejecuta `amount()` y retorna su resultado.

Hay que hacer notar que `amount()` lo único que hace es aplicar el descuento al resultado del `ShoppingCart.getAmount()`. Esto lo logra llamando a la función `proceed()`.

5.2. AOP y el Problema del Contexto

Retomando el caso de ejemplo, es claro ver que para el *e-Shop* las promociones son contextos que lo hacen adaptarse por medio de diferentes aplicaciones de descuentos. Además, la implementación de ellas a través de *if/else* o siguiendo el patrón *strategy* ofrecen claros problemas de separación de intereses, dado que fuerzan al shopping a preocuparse por problemas que afectan en otras partes del código (*crosscutting concerns*).

Podemos decir entonces que, a partir del ejemplo presentado, es factible plantear la idea que durante la construcción de aplicaciones resulta ser una preocupación transversal (*crosscutting*) el prepararlas para que tengan un comportamiento dependiente del contexto. Esto se debe fundamentalmente a que estas preocupaciones no se encuentran aisladas en una parte, sino que se encuentran dispersas por el código, atravesándolo por completo. Esto nos indica que estamos frente a un problema que puede ser tratado como un aspecto.

Esto hace plantear inmediatamente la idea de que los descuentos pueden ser tratados como aspectos, que en términos generales significa que los aspectos, dependiendo de los contextos, modificarán el comportamiento de la aplicación context-aware. Estos aspectos son los llamados *context-aware aspects* o aspectos atentos al contexto.

[1] *AspectJ* es una extensión orientada a aspectos para el lenguaje de programación *Java*.

6. Herramientas Disponibles

Existen dos herramientas que serán utilizadas para las secciones posteriores, es conveniente presentarlas en este momento. La primera es un implementación de AOP en Java, su nombre: Reflex. La segunda, WildCAT, permite a cualquier aplicación Java acceder a su ambiente de ejecución (predefinidos o creados por el programador) de diversas formas y a través de un API única.

6.1. AOP: Reflex

En esta sección se hará una introducción a Reflex, la herramienta que implementa AOP en Java y que será utilizada a lo largo de toda esta tesis.

Para comprender cuales son los fundamentos y las funcionalidades que Reflex provee será necesario comenzar con elementos que fundamentan la programación orientada a aspectos, para pasar luego a ver como son implementados en Reflex.

La primera parte de esta sección está dedicada a la definición de metaprogramación y reflexión. En la siguiente sección serán presentadas las herramientas que ofrece Reflex para la implementación de ambas. Por último, se hará explícita la relación que existe entre metaprogramación y AOP, mostrando la correspondencia de los elementos presentes en ambos. Esto lleva indiscutiblemente a la conclusión de que Reflex permite implementar AOP y por lo tanto puede ser usado para implementar los *context-aware aspects*.

6.1.1. Metaprogramación y Reflexión

La metaprogramación es la capacidad de un programa de razonar y actuar sobre otro programa. Un programa que tiene esta capacidad es llamado *metaprograma*. Su mismo nombre denota que se encuentra por sobre un programa. A este último se le llama *programa base*. Por consiguiente, al espacio donde se encuentra el programa base[2] será llamado *nivel base* y el espacio donde se encuentra el metaprograma será llamado *metanivel*.

Dado que Reflex se encuentra bajo el contexto de la orientación a objetos y que se ha implementado en Java, estos conceptos tienen que ser llevados a ese ámbito. En primer lugar, la aplicación original estará conformado por muchos *objetos* y el metaprograma estará formado por mucho *metaobjetos*.

Como ya fue enunciado un metaprograma (un metaobjeto) tiene la capacidad de *razonar* y *actuar* sobre otro programa (otro objeto). Esto implica que puede realizar dos acciones sobre el programa:

- *Instrospection*, que consiste en examinar la aplicación original.
- *Intercession*, que consiste en modificar la aplicación original.

Ambas acciones se aplican a un nivel estructural y de comportamiento. En el caso de los objetos y dada la implementación de Java, esto significa que es posible:

1. A Nivel estructural:
    - Añadir un campo a una clase.
    - Añadir un método a una clase.
    - Modificar la declaración de una clase (agregar o modificar su superclase, añadir una interfaz, etc).

[2]También llamado programa original o aplicación original.

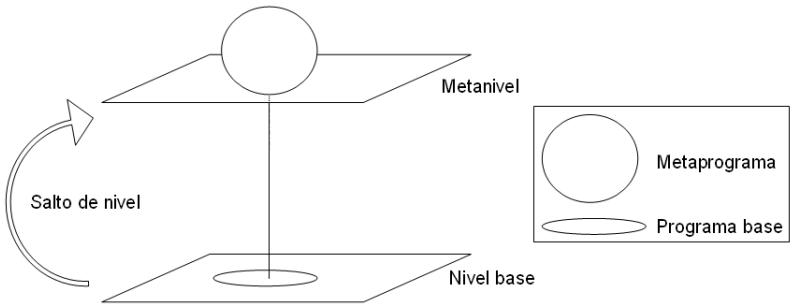

Figura III.3: Esquema de conceptos de metaprogramación

2. A Nivel de comportamiento:
   - Modificar el comportamiento de un método.
   - Eliminar o modificar alguna invocación a un método, instanciación, acceso a variable, etc.

Para que estas acciones se puedan realizar es necesario que en algún momento el flujo de ejecución pase desde el nivel base al metanivel. Al hablar de objetos y metaobjetos este salto no es automático, sino que es necesario definirlo y se representa como un *salto al metanivel*.

Todos los conceptos definidos acerca de metaprogramación pueden ser observados en forma esquemática en la figura III.3.

Para que el salto ocurra, es necesario definir primero en qué lugar se llevará a cabo, este punto es llamado *corte*. Un corte especifica en que lugar el metanivel tomará el control de la aplicación.

Para que el metaobjeto pueda realizar las acciones antes descritas será necesario que éste reciba información acerca del nivel base. Ejemplos de esta información serán el objeto sobre el cuál se realiza la acción, la acción que se está realizando y cuales son los argumentos de dicha acción. Este proceso de reunir toda esta información para entregarla al metanivel es llamado *reificación*.

Además de definir en que punto se realizará el salto, es necesario definir desde donde y hacia donde se realizará. Esta operación es un *binding* entre objetos y metaobjetos.

Teniendo en mente la definición y todos los conceptos relacionados con metaprogramación, la definición de reflexión es muy simple de enunciar. *Reflexión* significa la capacidad de un programa de *razonar* y *actuar* sobre si mismo, o sea, estamos frente a una caso particular de metaprogramación, donde el programa base y el metaprograma son el mismo[9].

6.1.2. Metaprogramación con Reflex

Reflex implementa todos los conceptos vistos anteriormente de una forma particular:

1. Los metaobjetos son definidos como clases de primer nivel.
2. Se puede especificar en forma transparente el paso al metanivel.
3. Es posible configurar la información recibida en el metanivel.
4. El vínculo entre el nivel base y el metanivel es explícito.

A continuación se detalla brevemente cada uno de estos elementos.

Definición de metaobjetos

Al definir los metaobjetos como clases de primer nivel, el concepto de metaobjeto se hace explícito, ya que se concretiza de la misma forma que cualquier otra clase de primer nivel. Además, esto permite la existencia de metaniveles sobre metaniveles, dado que al ser clases de primer nivel, un metaobjeto es un objeto que a su vez puede tener otro metaobjeto asociado. Esto es útil, ya que agrega ciertas características deseables para una herramienta de metaprogramación:

- Evita que el comportamiento asociado al nivel base se mezcle con el metanivel.
- Provee la capacidad de cambiar el metaobjeto asociado al metaprograma al ser éste una entidad por si sola.

Si retomamos el ejemplo del *e-Shop*, y deseamos hacer un descuento al valor devuelto por el `getAmount()` del `ShoppingCart` a través de un metaobjeto implementado en la clase `DiscountMO`, en Reflex se hace de la siguiente forma:

```
MODefinition theMetaobject = new MODefinition.MOClass("DiscountMO");
```

Reflex utiliza la clase `MODefinition` como un envoltorio para la clase `DiscountMO`. De esta manera es posible realizar configuraciones al alcance del metaobjeto[3].

Paso al Metanivel

La especificación del paso al metanivel, o sea, el punto en que el flujo de ejecución de la aplicación va a pasar al metanivel se define como un *hook*. Un hook define un punto en la aplicación base donde ocurre una operación sobre la cual hay interés. Las operaciones definidas por Reflex son:

- Envío de mensaje (invocación a un método).
- Recepción de mensaje (invocación a un método).
- Acceso de lectura o escritura a una variable.
- Instanciación de clase (objeto devuelto por el constructor).
- Creación de objeto (ejecución del constructor de una clase).

En Reflex es posible expresar que un metaobjeto especifica varios *hooks*. Esto permite definir un conjunto de *hooks*, llamado *hookset*.

En el ejemplo del descuento aplicado al `getAmount()` del `ShoppingCart` se podría definir con Reflex el siguiente *hookset*:

```
Hookset theHookset = new PrimitiveHookset(
                        MsgReceive.class,
                        new NameCS("ShoppingCart"),
                        new NameOS("getAmount"));
```

Los tres parámetros indican que la operación es una recepción de mensaje (`MsgReceive.class`), que aplica a la clase `ShoppingCart` (`new NameCS("ShoppingCart")`), y que el mensaje es `getAmount()` (`new NameOS("getAmount")`).

Reificación

Para configurar la información que va a ser pasada al metanivel, o sea para realizar el proceso de *reificación* de esta información, se emplea la interfaz `Parameter` provista por Reflex:

[3]Por ejemplo, si se va a compartir el metaobjeto, o existirá una instancia para cada clase que lo utilice.

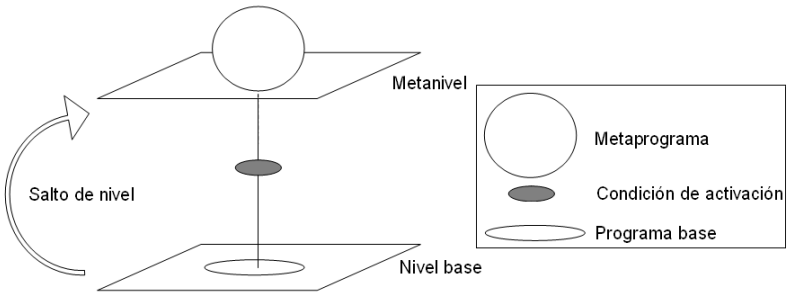

Figura III.4: Esquema de conceptos de metaprogramación

```
public interface Parameter{
  public String evaluate(Operation aOperation);
  public String getType(Operation aOperation);
}
```

Esta interfaz define dos métodos:

- `evaluate()`: especifica que valor tomará la variable.
- `getType()`: especifica el tipo de la variable.

El parámetro que recibe el método es de tipo `Operation` y representa la operación para la cual se generó el *hook*.

Existen varias implementaciones de esta interfaz disponibles en Reflex. Por ejemplo, es posible obtener el objeto sobre el cual se ha aplicado el *hook*, los parámetros pasados a la operación interceptada, etc.

Definición del Vínculo entre el Nivel Base y el Metanivel

Por último, el *binding* que se realiza entre un *hookset* y un metaobjeto es explícito y se realiza a través de un *link*.

De igual forma que es posible configurar algunas opciones sobre un metaobjeto dado que es explícito, es posible configurar el *link*, o sea, configurar como se realizará el salto al metanivel.

Para el presente trabajo sólo son relevantes los siguientes atributos configurables:

1. Tipo de control: se refiere a cuando saltar al metanivel en relación a la operación seleccionada.
   - BEFORE: antes de que se ejecute la operación.
   - AFTER: después de que se ejecute la operación.
   - AROUND: que el salto reemplace la ejecución de la operación.
2. Condición de activación: dado que el proceso de reificación (invocación a un método y la recolección de datos) es caro en tiempo de ejecución, Reflex permite minimizar dicho tiempo agregando una etapa previa a la realización del salto, que verifica si el *link* está activo o no. Una visión esquemática de la ubicación de la condición de activación está en la figura III.4.
3. Descriptor de llamada: dado que el paso al metanivel se reduce a la invocación al metaobjeto asociado, Reflex permite definir dicha invocación indicando el método del metaobjeto que será llamado y los parámetros que serán pasados.

Para el ejemplo del descuento, estos tres elementos se definen a través de Reflex de la siguiente forma:

```
BLink applicationLink = API.links().createBLink(theHookset, theMetaobject);
applicationLink.setControl(Control.AROUND);
applicationLink.addActivation(theActivationCondition);
applicationLink.setCall("DiscountMO",
       "amount",
       new Parameter[]{
           Parameter.CLOSURE
       });
```

Con `API.links().createBLink(...)` se crea el *link*. Sus parámetros son el *hookset* y el metaobjeto que definimos anteriormente. En la siguiente línea de código se define el tipo de control, que en este caso es `AROUND`. Las condiciones de activación se agregan con `addActivation()` de la interfaz `BLink`, su parámetro es alguna condición de activación (en el siguiente capítulo ahondaremos este tema). Y por último se define el descriptor de llamada. En este caso la llamada al metaobjeto se realiza llamando al método `amount()` de la clase `DiscountMO` y pasándole como parámetro el objeto al que se aplicó el *hook* (Parameter.CLOSURE realiza esta acción y es proveído por Reflex).

6.1.3. Reflex como herramienta AOP

Como se mencionó en la sección 5 del capítulo III (Aspectos y Contextos), AOP se presenta como una alternativa para solucionar el problema de *SoC*. Dicha alternativa consiste en la identificación de aspectos desde la aplicación original, para luego llevar a cabo su implementación de forma separada y finalmente, insertar esta implementación de vuelta en la aplicación original.

En el caso de esta tesis, es relevante considerar el modelo *Pointcut-Advice*[10] para AOP que define para su funcionamiento los siguientes conceptos:

- Jointpoint: es un punto de ejecución de la aplicación original.
- Advice: es el código que es ejecutado cuando se alcanza un *jointpoint*.
- Pointcut: es un estructura que permite agrupar joinpoints y exponer cierto contexto presente en ellos al *advice*.

De las definiciones anteriores, es posible desprender que un aspecto es la unión de un *pointcut* y un *advice*.

A continuación iremos identificando los conceptos definidos para metaprogramación con estos conceptos definidos para AOP.

Advices y Métodos de los Metaobjetos

En AOP un *advice* es el código asociado al aspecto que está siendo extraído de la aplicación original para lograr la eliminación de aspectos no funcionales o transversales en ella.

Recordemos que en metaprogramación un metaobjeto es una entidad que se posiciona por sobre la aplicación original para poder actuar y razonar sobre ella. Dichas acciones se realizan en los métodos del metaobjeto.

La relación es clara, y se establece ya que un advice puede ser implementado como un método en el metaobjeto que está sobre la aplicación base. Por lo tanto, la función del metaobjeto es implementar un aspecto que se desea resolver.

Jointpoint y Hooks

Si bien es cierto un *jointpoint* esta definido como un punto de ejecución, mientras que un *hook* esta definido como un punto en el programa base (en el código), ambos son lo mismo. Esto se debe a que en el momento de ejecutarse el programa base los puntos definidos en el código pasan a ser puntos de ejecución, y por lo tanto un *jointpoint* se puede identificar con un *hook*.

Pointcuts, Hooksets y Links

Como ya dijimos, un *pointcut* es una conjunto de *joinpoints*. Además, un *pointcut* permite la especificación de la información acerca de los *jointpoints* que será expuesta a los *advice*.

Considerando que un *hookset* es un conjunto de *hooks* y que está relacionado a un *link*, y que además este link permite especificar la condición de activación del salto al metanivel y la información que se pondrá a disposición del metaobjeto asociado, podemos decir que se cumple:

$$pointcut = hookset + activacion + descriptor\_de\_llamada$$

Dado que podemos considerar que para un link existe un único metaobjeto asociado, tenemos que:

$$link = hookset + activacion + descriptor\_de\_llamada + metaobjeto$$

Utilizando ambas igualdades, junto con la conclusión del apartado anterior, que nos indicaba que un conjunto de *advices* es un metaobjeto, podemos concluir que:

$$\begin{aligned} link &= pointcut + metaobjeto \\ &= pointcut + conjunto\_de\_advices \\ &= aspecto \end{aligned}$$

Esta igualdad se tiene para una aspecto simple, pero en el caso de aspectos complejos se tiene:

$$link = n \cdot aspecto$$

De todas formas Reflex permite abstraer los aspectos complejos para que de todas formas la primera igualdad sea cierta.

6.1.4. Resumen

Reflex permite realizar metaprogramación y reflexión sobre aplicaciones originales o programas base. Para este efecto, Reflex provee toda una serie de herramientas que implementan todos los elementos necesarios para hacer metaprogramas en un metanivel y conectarlos con el nivel base.

Teniendo en cuenta el modelo de *Pointcut-Advice*[10] para AOP, es posible mapear todos los conceptos de metaprogramación y reflexión a los conceptos de AOP. Por lo mismo, dado que Reflex permite realizar metaprogramación y reflexión, nos permitirá entonces implementar aplicaciones con el paradigma AOP, y en particular implementar el framework de *context-aware aspects*.

6.2. Context-awareness: WildCAT

6.2.1. Fundamentación de WildCat

El desarrollo de aplicaciones *auto-adaptables*[8] o de sistemas autónomos[7] requiere de un gran conocimiento del *entorno de ejecución* del sistema y de su *evolución*. El entorno de una aplicación, en una primera aproximación, puede comprender recursos de *hardware* y *software*, pero además puede incluir información de otros ámbitos, como *posición geográfica*, *opciones del usuario*, *capacidad del sistema* y su *actividad*.

En un entorno de aplicaciones móviles el manejo de esta información es crítico, ya que en base a ella tiene que ser capaz de adaptar su comportamiento. Por ejemplo, en el caso de la comunicación a través de las redes habilitadas es necesario conocer los anchos de banda disponibles para elegir el protocolo apropiado de comunicación. En el caso de la interfaz con el usuario esto también es importante. Si el usuario esta usando su dispositivo móvil como reproductor de mp3, resulta inútil enviar mensaje a través de la pantalla (porque es muy probable que no la esté viendo), por el contrario sería muy útil dar una señal de audio o de movimiento para inmediatamente captar la atención del usuario. Esta habilidad de una aplicación de conocer el entorno en el cual está ejecutándose es conocida como *context awareness*.

Resulta muy importante entonces conocer el contexto dinámicamente, o sea, en todo momento, obteniendo información certera (que nos diga *realmente* como está el entorno) y actual (que nos diga como está *ahora*). Esta información obtenida *tiene* que ser *interpretada*, o sea, hay que razonar con ella. Y por último la aplicación tiene que estar alerta frente a hechos o eventos que ocurran, y no cualquiera, sino que sólo los que sean de interés.

La mayoría del código que es necesario para realizar todas las tareas descritas requieren interactuar a bajo nivel con el *Sistema Operativo* o en ciertos casos directamente con el *hardware*. Y aquí surgen los problemas, ya que no resulta útil hacer que los desarrolladores de software programen esta clase de código, especialmente si éste resulta siempre muy parecido en muchas aplicaciones.

6.2.2. Objetivos de WildCat

Por estas razones fue desarrollado WildCAT[2], un *toolkit* para *context awareness*, que además de ser liviano en su ejecución es extensible. Este provee a los desarrolladores de aplicaciones en Java una forma sencilla de hacer que sus aplicaciones sean *context-aware*

WildCAT provee un modelo de datos simple y dinámico que representa el contexto de ejecución de la aplicación. Ofrece un API muy simple para acceder a esta información, ya sea en forma *síncrona* o *asíncrona*[4]. Este *framework* esta diseñado para facilitar la obtención y el manejo de la información referente al contexto, además de permitir crear nuevos dominios de contexto (similares a los ya nombrados de hardware, red, configuración del usuario, etc) que sean relevantes para muchas otras aplicaciones.

La Figura III.5 representa el modelo lógico utilizado por WildCAT. Este modelo no corresponde exactamente al implementado en la versión final de WildCAT, pero sirve para ilustrar la visión de los clientes del *framework* que sólo verán la clase *Context*. Esta clase sigue el patrón de diseño *Singleton*[4], o sea, sólo existe una instancia de ella para cada aplicación.

[4]Cuando es necesario conocer inmediatamente algún atributo del contexto hay que utilizar la forma *síncrona*, en cambio, cuando la aplicación necesita ser avisada del acaecimiento de un evento hay que utilizar la forma *asíncrona*

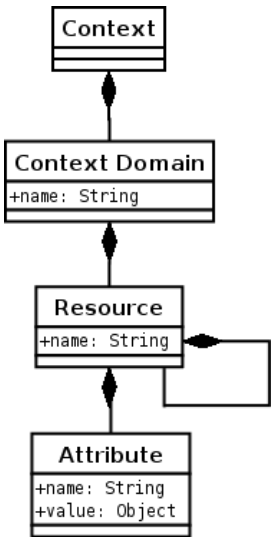

Figura III.5: Modelo lógico de datos de WildCAT

6.2.3. Dominios de Contextos

Para WildCAT el contexto está compuesto de varios *dominios* cada uno de ellos está representado por un objeto *ContextDomain*. Esta visión permite separar los distintos tipos de contexto de ejecución, cada uno con su propia implementación, permitiendo mejorar la *performance* (por que cada tipo de contexto es manejado en forma directa y no genérica) o para tener interoperabilidad con sistemas ya existente.

Cada dominio de contexto tiene un conjunto de *recursos* (objetos *Resource*) que a su vez son descritos por atributos (objetos *Attribute*). Esta es una estructura lógica muy sencilla y fácil de comprender por los programadores que usen WildCAT (clientes), pero no es más que un modelo lógico. Actualmente la implementación usa un modelo más complejo, pero en general es similar al ya presentado y por lo demás sirve para comprender como funciona WildCAT.

Otro elemento muy importante dentro de WildCAT son los *eventos*. Cada cambio ocurrido en el contexto es representado por un evento, o sea, representan la dinamicidad del contexto. Los posibles tipos de eventos son:

- `RESOURCE_ADDED` y `RESOURCE_REMOVED` gatillado cuando un recurso aparece o desaparece dentro del contexto. Podría ser gatillado si aparece un nuevo dispositivo de entrada o si es sacado el cable de un servicio de red.
- `ATTRIBUTE_ADDED` y `ATTRIBUTE_REMOVED` gatillado cuando aparece o desaparece un atributo en un recurso.
- `ATTRIBUTE_CHANGED` gatillado cuando cambia el valor de un atributo, tal como el número de paquetes TCP perdidos por una interfaz de red.
- `EXPRESSION_CHANGED` gatillado cuando el valor de una *expresión sintética*[5] del contexto cambia, por ejemplo el valor máximo de espacio libre en disco.
- `CONDITION_OCCURRED` gatillado cuando una expresión sintética se vuelve verdadera

[5]Una *expresión sintética* se refiere a la forma específica de indicar al contexto como monitorear el valor de un atributo, por ejemplo, que la temperatura no exceda los $30°C$.

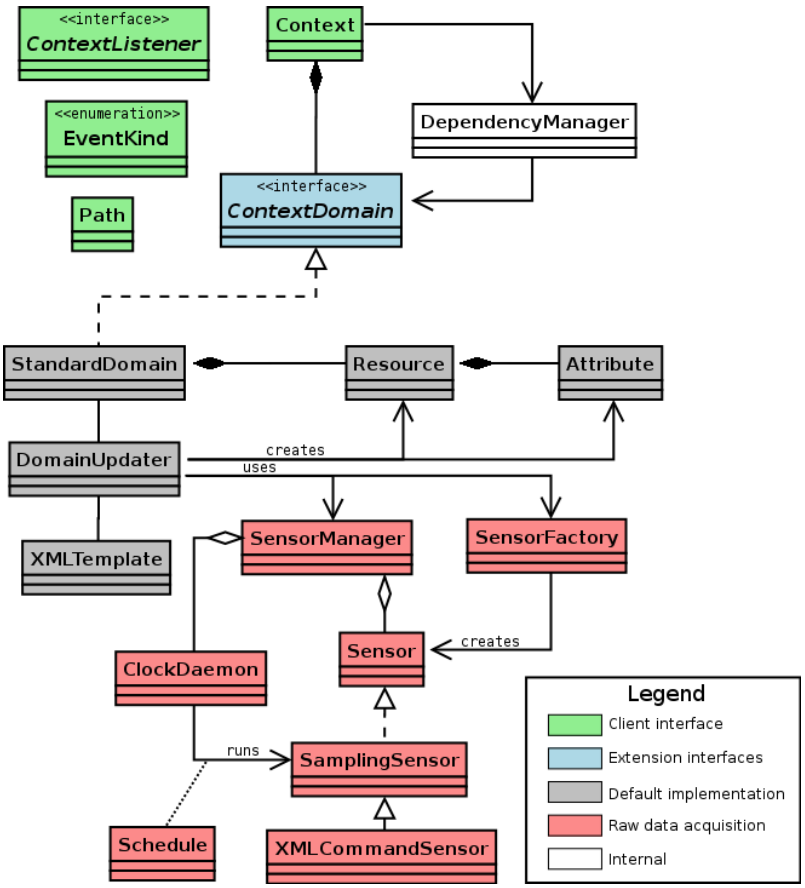

Figura III.6: Diagrama de clases de WildCAT

Wildcat utiliza una sintaxis inspirada en las *URIs* (*Uniform Resource Identifier*) para identificar elementos en el contexto. La sintaxis general es:

`domain://path/to/resource#attribute`

El *path* indica un recurso, un atributo, o todos los subrecursos o subatributos de un recurso. Hay que hacer notar que a pesar de que un *path* esté sintácticamente bien escrito, éste puede no existir. Además permite usar un comodín, el asterisco (∗). Veamos algunos ejemplos de la sintaxis:

- `sys://storage/memory` //Un recurso
- `sys://storage/disks/hda#removable` //Un atributo
- `sys://devices/input/*` //Dispositivos de entrada
- `sys://devices/input/mouse#*` //Atributos del Mouse

El diagrama UML de la Figura III.6 representa la estructura interna de WildCAT. El *framework* puede ser descompuesto en varias partes (identificadas por colores). La parte del cliente (en verde) ya ha sido descrita.

La interfaz *ContextDomain* es el principal punto de extension de WildCAT. Nuevas implementaciones de dominios de contexto pueden ser creadas si los dominios que ya existen no satisfacen las necesidades de una aplicación en particular.

6.2.4. Obtención de Datos

Toda la parte relativa a la obtención de los datos del contexto se encuentran en color rosado (*Data acquisition*) y toda ella gira en torno a la noción de *sensor*. A través de la clase *SensorManager* es posible organizar todos los sensores. Cada uno está identificado por un nombre y puede ser activado/desactivado. WildCAT distingue dos tipos de sensores: *Activos* y *Pasivos*. Los *activos* implementan la interfaz *Sensor* y corren en su propio *thread*. Ellos son responsables de enviar los datos recopilados al *SensorManager*. Los pasivos son creados asociando un *Sampler* y un *Schedule* para crear un *SamplingSensor*. En este caso el *SensorManager* es el encargado de ir preguntando al sensor los datos recopilados.

Para facilitar la integración con otros sistemas, WildCAT provee la clase *XMLCommandSensor*. Esta clase implementa la interfaz *Sensor* invocando un programa externo a WildCAT. La salida del programa externo tiene que ser un XML válido que represente un objeto *SampleSet*. Este XML será parseado y retornado como objeto por parte del *XMLCommandSensor*. Esto puede ser útil para obtener información más bien estática, como la versión del sistema operativo o de otro programa. Por razones de *performance* no se recomienda utilizar esto para monitoreo de sensores muy frecuentes, esto debido al sobrecosto de la llamada a programas externos.

Finalmente, WildCAT provee una implementación estándar de *ContextDomain* en la clase *StandardDomain* (Figura III.6, color gris). Esta implementación sigue el modelo lógico presentado, con clases para recursos y atributos. La creación y el mantenimiento de *ContextDomain* está a cargo de *DomainUpdater*, el cual agregará/removerá recursos y atributos de acuerdo a la especificación del contexto provista por *XMLTemplate*. La obtención de los datos queda delegada directamente a la estructura que revisamos en los párrafos anteriores.

La especificación de un *ContextDomain* a través de un XML resulta de mucha utilidad, ya que al ser genérica, fácilmente puede ser definida a través de dicho archivo. A continuación presentamos un ejemplo:

```
<?xml version='1.0' encoding='ISO-8859-15'?>
<context-domain name="sys">
    <resource name="load">
        <sensor name="load" class="org.obasco.wildcat.sensors.LoadSensor">
            <schedule>
                <periodic period="10000"/>
            </schedule>
        </sensor>
    </resource>
<context-domain/>
```

El elemento principal del XML es el `context-domain` e indica el nombre del dominio que será definido. Dentro de él encontramos los recursos identificados por sus nombres. Los recursos son asociados con sensores definidos por una clase Java que los implementa. Estos pueden contener información acerca del intervalo de tiempo en que será revisado.

Recordemos que un recurso puede tener subrecursos. Esto es simple de hacer en XML:

```
<resource name="storage">
  <resource name="disks">
    <sensor name="hda" class="org.obasco.wildcat.sensors.HardDriveSensor">
      <schedule><on-create/></schedule>
      <configuration>
        <device>/dev/hda</device>
        <mode>static</mode>
```

```
      </configuration>
    </sensor>
  </resource>
</resource>
```

En este caso los recursos *storage* y *disks* son abstractos y sirven sólo para organizar los recursos internos. En el ejemplo anterior resulta importante destacar la etiqueta *configuration*, debido a que cada Sensor que pueda ser configurado tendrá su propio formato de configuración y será responsable de entender el contenido de la misma. La única exigencia hasta este punto es que sea un XML bien formado.

Además es posible definir *atributos sintéticos*, como en el ejemplo siguiente:

```
<resource name="network">
  <attribute name="nb_interfaces">count(*)</attributes>
  <resource name="eth0">
    <sensor name="nic" class="org.obasco.wildcat.sensors.NICSensor">
      <schedule><periodic period="1000"/></schedule>
      <configuration><device>eth0</device</configuration>
    </sensor>
    <attribute name="error_rate">#dropped_packets / #received_packets</attribute>
  </resource>
</resource>
```

En este caso aparece el atributo *error_rate*, y se define explicitamente por que es distinto a la información típica obtenida a través de los sensores. El valor de estos atributos se define a través de expresiones sintéticas (iguales a las indicadas para monitorear el valor de un atributo). Estas pueden estar referidas a cualquier parte del contexto, WildCAT se encargará de ubicarlas a través del *DependencyManager*. En este caso se refieren a atributos locales. Lo importante es que para los clientes estos atributos son exactamente iguales a los demás.

Es claro ver que WildCAT resulta ser una herramienta fácil de utilizar, y sobretodo que cumple su cometido de ser liviana, lo suficientemente genérica para los dominios básicos y fácil de extender a nuevos dominios de contexto apropiados para muchas otras aplicaciones.

7. Context-aware Aspects

Teniendo en cuenta que la aplicación de un descuento sobre el *shopping cart* es un aspecto, es posible entonces empezar a formular los requerimientos que debería cumplir una implementación de context-aware aspects. Esto implica indicar de qué forma un aspecto es restringido a un contexto en particular.

Para poder restringir un aspecto a un contexto particular se requiere la posibilidad de referir a la definición del contexto en la definición del *pointcut*. Complementando el ejemplo en *AspectJ* sería:

```
pointcut amount(): execution(double ShoppingCart.getAmount())
                   && inContext(PromotionCtx);
```

Además, puesto que el porcentaje de descuento (`rate`) puede variar en la promoción utilizada, el *advice* (`amount`) puede necesitar acceder algún estado externo del contexto. El ejemplo quedaría entonces:

```
aspect Discount{
    pointcut amount(double rate): execution(double ShoppingCart.getAmount())
                        && inContext(PromotionCtx(rate));
    double arount(double rate): amount(rate){ return proceed()*(1-rate);
}
```

En este ejemplo se asume que el contexto de promoción expone un atributo `rate` (a través de su `getRate()` correspondiente), distinto al caso original en que estaba definido como variable del aspecto (sección 5.1 de este capítulo). Además de exponer el estado del contexto, tiene que ser posible parametrizar un contexto cuando se exprese una referencia a él, por ejemplo, el aspecto `Discount` podría depender de una `PromotionCtx` basado en una ventana de tiempo, y un `StockOverloadCtx` parametrizado con rango de sobre stock del 80 %[6] Este ejemplo se vería en *AspectJ* como sigue:

```
pointcut amount(double rate): execution(double ShoppingCart.getAmount())
                    && inContext(PromotionCtx(rate))
                    && inContext(StockOverloadCtx[.8]);
```

El restrictor `inContext`[7] restringe el aspecto basado en alguna condición (contexto) que es verificada en el momento.

Pero debe también ser posible restringir un aspecto basado en si la aplicación estaba en algún otro tipo de contexto, y no necesariamente un contexto que se verifica inmediatamente. Dos ejemplos de restrictores diferentes son presentados a continuación.

7.1. Restrictor CreatedInCtx

Debe también ser posible restringir un aspecto basado en si la aplicación estaba en algún contexto previamente. Existe una dependencia del pasado que es muy útil: conocer el contexto en el cual un objeto fue creado.

Al utilizar un restrictor apropiado, llamado `createdInCtx`, se logra que el aspecto `Discount` se aplique a cualquier *shopping cart* que fue creado cuando el `PromotionCtx` estaba activo, sin considerar si la promoción está aún activa cuando se realiza el *check out*. El ejemplo quedaría:

```
pointcut amount(): execution(double ShoppingCart.getAmount())
                    && createdInCtx(PromotionCtx)
```

7.2. Restrictor PutInCartCtx

Resulta además deseable poder definir restricciones de contextos de dominio específico o referentes a una aplicación específica. En el caso de ejemplo esto significa poder referirse al contexto cuando un producto es agregado al *shopping cart*, asumiendo el respectivo restrictor `putInCartCtx`.

```
pointcut amount(): execution(double Item.getPrice())
                    && putInCartCtx(PromotionCtx)
```

Al manejar contextos con referencia al pasado y contextos con estado (*stateful)* es necesario mantener un seguimiento de los contextos pasados y sus estados asociados. Esto significa que por

[6]La sintaxis (...) indica exposición de estado y [...] indica parametrización del contexto.

[7]Se llama restrictor, porque restringe el *pointcut*, y por lo tanto la ejecución del *advice*.

un lado los contextos pueden tener un estado que puede variar en el tiempo, y por otro la aplicación del descuento va a depender de un contexto de promoción que ya pasó. El proceso de mantener un seguimiento de los contextos pasados y sus estados lo llamaremos *context snapshotting* y a la foto del estado de un contexto en un tiempo determinado será un *context snapshot*. Un *context snapshot* global será un *snapshot* de todos los contextos definidos en un momento determinado.

Por razones de implementación, especialmente de uso de memoria, es imposible hacer un *context snapshot* global a cada momento. Es necesario considerar que se debe hacer un *context snapshot* sólo cuando sea necesario. Además sería útil asociar el *context snapshot* al objeto al que está relacionado para no tener que mantener un repositorio global de *context snapshots*, que seguramente se volverá un cuello de botella. Por último, la posibilidad de definir restricciones de contextos de dominio específico o referentes a una aplicación específica implica que la capacidad de hacer *snapshooting* (si es necesario) tiene que poder ser definida por el programador.

En resumen, para los *context-aware aspects*, es necesario lo siguiente:

- Contextos y los *context-aware aspects* tienen que estar separados.
- Los contextos tienen que usar parámetros, tener un estado y ser componibles.
- El estado de contexto tiene que ser visible en el pointcut (para poder ser utilizado en el meta nivel).
- Se debe poder expresar una dependencia con contextos pasados.
- Tiene que permitir definir contextos relacionados con otros dominios.
- El snapshotting de contextos tiene que ser astuto.

8. Resumen

Teniendo en cuenta los avances en sistemas autoadaptables[8] y sistemas autónomos[7], resulta cada vez es más necesario incorporar en las aplicaciones la capacidad de modificar su comportamiento de acuerdo al contexto en que se encuentran.

Lo complicado de esta situación es que no resulta fácil definir, diseñar e implementar una aplicación con estas características. Existen diversas alternativas de solución, pero en general no dejan cerrado el tema, sino que por el contrario abren nuevas interrogantes acerca de como lograr realmente una separación de intereses con respecto a los aspectos que implementan los contextos.

AOP resulta ser una ayuda importante a la hora de resolver problemas con SoC. Queda entonces planteada la posibilidad de implementar el manejo de contextos a través de herramientas que implementen AOP (como Reflex) y que permitan crear aplicaciones context-aware (como WildCAT).

Los context-aware aspects son justamente esto, la respuesta desde AOP al problema de la implementación, administración y aplicación de aspectos basados en contextos.

Capítulo IV

Implementación de un Framework de Context-aware Aspects

En este capítulo se concentra la parte más importante de la tesis. La implementación del framework significa en si mismo un desafío, ya que no es inmediato que para enfrentar la implementación de alguna solución estándar se pueda crear un framework que facilite la labor.

En la sección 1 será definida la estructura o división del framework. Esta ya muestra que incluso dentro del mismo framework se optó por separar los intereses.

La forma de definir contextos es presentada en la sección 2, seguida de la definición de restricciones.

Una sección aparte (4) tiene un tipo de restricciones especiales, aquellas que dependen de promociones que estuvieron activas (pasado).

La sección 5 entrega los elementos fundamentales de la integración con WildCAT, indicando incluso como es posible definir nuevos sensores.

Este capítulo concluye con una sección dedicada especialmente a presentar como se utiliza el framework, indicando principalmente la configuración que es necesaria realizar para tener funcionado un *context-aware aspect*.

1. Estructuración del Framework

Para cumplir con los requisitos definidos para un framework de *context-aware aspects*, es necesario contar con dos elementos fundamentales:

- Tiene que permitir definir contextos.
- Se debe poder aplicar restricciones de activación sobre los aspectos basados en contextos.

La implementación de este framework se separa entonces en dos packages: `reflex.caa.definition` para la definición y `reflex.caa.restriction` para las restricciones. El diagrama de clases de esta implementación se encuentra en la figura IV.1. Para no entorpecer la lectura no se indicará en las secciones siguientes que sea revisado este diagrama. Se asume que en caso de duda se puede consultar inmediatamente.

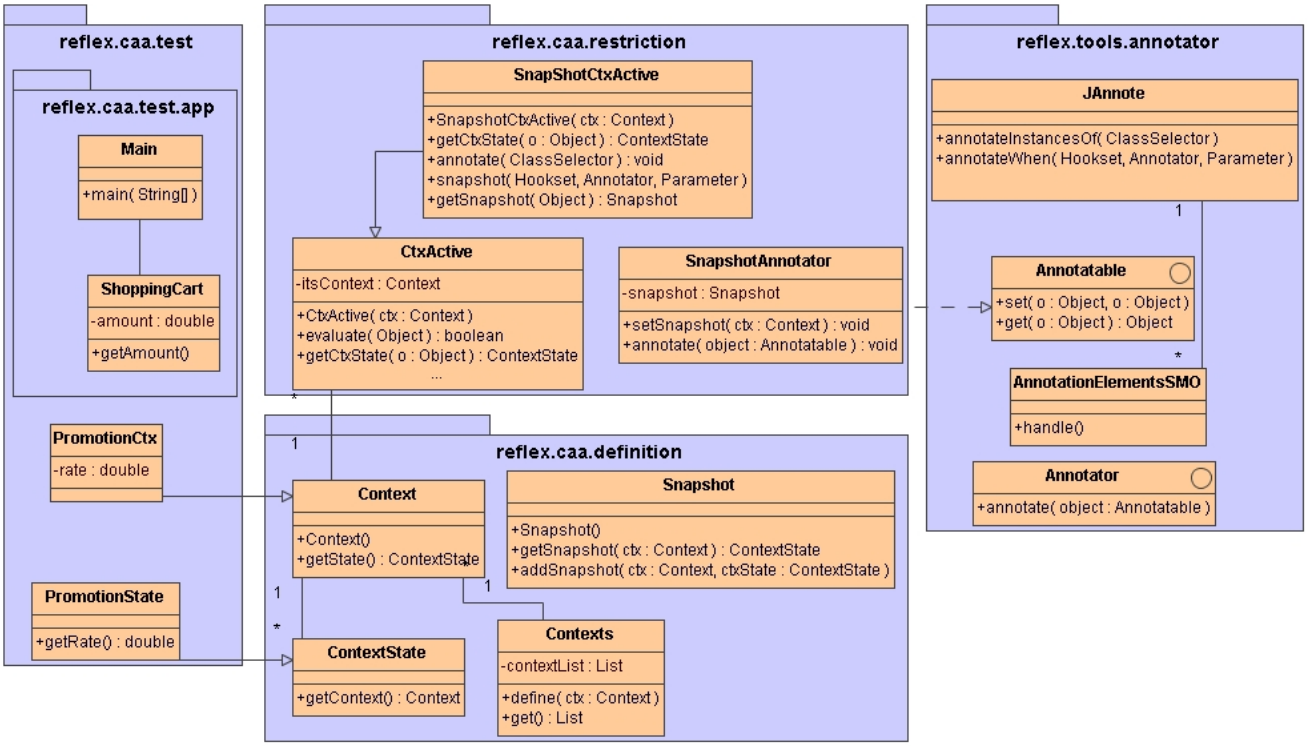

Figura IV.1: Diagrama de Clases del framework de context-aware aspects

2. Definición de Contextos

Reflex soporta la implementación de *pointcuts* de primera clase, a través de hooksets y condiciones de activación[11]. Ya con esto es posible pensar en un primer acercamiento a la implementación de un contexto de promoción aplicable a un `ShoppingCart`, sólo si este se encuentra dentro de un Web service. Lo llamaremos *WSPromotionCtx*:

```
public class WSPromotionCtx {
    CFlowExposer wsCFlow = CFlowFactory.getCflow(new PrimitiveHookset(
                                MsgReceive.class,
                                new NameCS("reflex.caa.demo.app.WebServiceRequest"),
                                new NameOS("perform")));
    public boolean active(){
        return wsCFlow.in();
    }
}
```

El `CFlowExposer` expone el flujo de control. Al enviar el mensaje `in()` al objeto, éste retorna verdadero si se encuentra dentro del flujo de control definido por el hookset que va de parámetro a su *factory*: `CFlowFactory`.

Para completar los requerimientos, es necesario definir la forma de almacenar los snapshot[1] , con el fin de acceder a ellos con posterioridad. Sería posible definir los contextos como clonables,

[1]Los *snapshot* son utilizados para almacenar contextos con referencia al pasado, o sea, contextos que estuvieron activos en un momento y en la actualidad es necesario manipular.

permitiendo de esta forma que al almacenar un snapshot se copien todos los contextos disponibles. Esto genera problemas ya que habría que empezar a controlar la profundidad de clonación, además de que se estarían generando muchos contextos siendo que es muy probable que se utilicen sólo algunos.

Esto lleva a definir el framework de manera tal que obligue al programador que implementa el contexto a preocuparse del problema del snapshot. Para obligar a esto los contextos deberán extender la clase `Context`, teniendo que implementar los métodos que utilizará para conocer el estado.

La forma de proceder si el estado está activo o no es totalmente genérica, por lo tanto, se ha implementado esta funcionalidad en `getState()`. De esta forma se recupera el estado del contexto como un objecto `CtxState` [2]. Para asegurar que este objeto está amarrado a su contexto será implementado como una *inner class* dentro de la clase `Context`. La implementación es la siguiente:

```
public abstract class Context {
    public Context() {
        Contexts.define(this);
    }
    public ContextState getState(){
        if (!active())
            return null;
        else
            return capture();
    }
    public class ContextState {
        public Context getContext() {
           return Context.this;
        }
    }
    public abstract ContextState capture();
    public abstract boolean active();
}
```

Otro elemento a considerar es la parametrización del contexto. Para el caso del `WSPromotionCtx` se cuenta con un porcentaje de descuento (`rate`) que debería ser definido al crear el contexto y recuperado al obtener el estado.

Si el contexto está activo el método `active()` retorna *true*. Es abstracto, porque depende de cada contexto la definición de cuando está activo o no. Si está activo, a través de `capture()` se obtiene el `ContextState` correspondiente al contexto activo en el momento.

Para finalizar hay que mencionar que el framework almacena un snapshot con todos los contextos que se crean. Esto se realiza a través de un `Contexts.define()` en el constructor de `Context`.

Dado que todas los contextos de promoción son muy similares, podemos implementar un `PromotionCtx` abstracto, que factoriza todos los elementos comunes. El diagrama de clases resultante para `PromotionCtx` y los contextos que lo extienden se encuentran en la figura IV.2.

```
public abstract class PromotionCtx extends Context {
    protected double rate;
```

[2]Para facilitar la implementación de la parte encargada de las restricciones se ha definido que esta *inner class* creada para cada contexto tiene que tener el mismo nombre que su clase padre concatenado con "State". Por ejemplo para `WSPromotionCtx` su clase `ContextState` respectiva será `WSPromotionCtxState`, siempre y cuando la implemente.

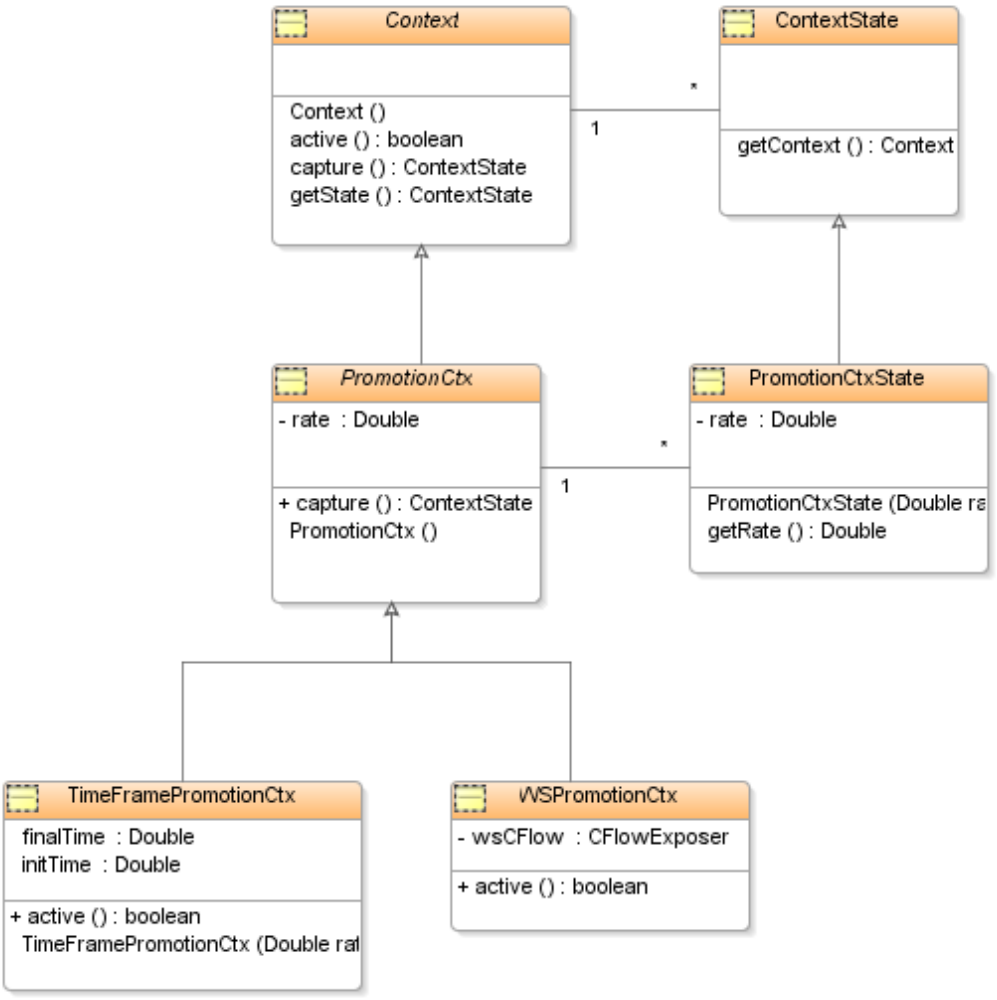

Figura IV.2: Diagrama de Clases de PromotionCtx y las clases que la extienden

```
    public PromotionCtx(double rate){
        this.rate = rate;
    }
    public class PromotionState extends ContextState{
        double rate;
        public PromotionState(double rate){
            this.rate = rate;
        }
        public double getRate(){
            return rate;
        }
    }
    public ContextState capture(){
        return new PromotionState(rate);
    }
}
```

Como ya fue mencionado, los contextos de promoción y sus estados poseen un parámetro de descuento `rate`. Este se inicializa al crear el `PromotionCtx`, y es traspasado al retornar el nuevo `ContextState` creado al momento de verificar si está activo el aspecto de descuento.

Dado que todos las clases que extiendan `PromotionCtx` tendrán exactamente el mismo `ContextState`, sólo el método `capture()` es implementado. En cambio, `active()` seguirá dependiendo de la implementación específica de cada contexto.

La implementación completa del ejemplo del contexto de promoción del Web Service mostrado sigue a continuación. Además se presenta la implementación de un contexto basado en una ventana de tiempo (`TimeFramePromotionCtx`).

2.1. Ejemplo: Web Service PromotionCtx

```
public class WSPromotionCtx extends PromotionCtx {
    CFlowExposer wsCFlow = CFlowFactory.getCflow(new PrimitiveHookset(
                                MsgReceive.class,
                                new NameCS("reflex.caa.demo.app.WebServiceRequest"),
                                new NameOS("perform")));
    public WSPromotionCtx(double rate){
        super(rate);
    }
    public boolean active(){
        return wsCFlow.in();
    }
}
```

Dado que la mayoría de las características del `WSPromotionCtx` están implementadas en `PromotionCtx`, sólo hace falta implementar el método `active()`.

En este caso, dadas las características del contexto, el retorno queda definido por el método `in()` del `CFlowExposer`. Este retorna verdadero si efectivamente el flujo de ejecución actual es el mismo flujo de ejecución (*CFlow*) del método `perform` del `WebServiceRequest`.

2.2. Ejemplo: Time Frame PromotionCtx

Este contexto se encuentra vigente si la hora actual se encuentra dentro del rango definido por una ventana horaria.

```
public class TimeFramePromotionCtx extends PromotionCtx {
    double initTime;
    double finalTime;
    public TimeFramePromotionCtx(double rate, double initTime, double finalTime){
        super(rate);
        this.initTime = initTime;
        this.finalTime = finalTime;
    }
    public boolean active(){
        Double actualTime = Clock.getTime();
        return (initTime<=actualTime && actualTime<=finalTime);
    }
}
```

Los atributos extras (`initTime` y `finalTime`) son utilizados para definir la ventana horaria (rango activo).

Al igual que en el caso anterior, sólo se definirá `active()`. Su implementación es simple, ya que sólo valida que la hora actual se encuentre dentro del rango definido por `initTime` y `finalTime`. Para facilitar las pruebas, se encapsuló la definición de *hora actual* dentro de la clase `Clock`.

Ambos ejemplos muestran dos formas distintas de definir contextos. Conviene finalizar recalcando que el trabajo de implementación de un contexto se centra fundamentalmente en definir correctamente sus métodos `active()` y `capture()`.

3. Restricciones de Activación de los Aspectos dependientes de contextos

Las restricciones definidas en la sección 3.7 y que se aplican sobre los aspectos dependientes de contextos se pueden expresar en Reflex a través de las condiciones de activación que tienen los *links*. Estas se implementan a través de la interfaz `Active` disponible en Reflex.

Para esto definiremos una clase abstracta que implemente la interfaz `Active`, y que representará la condición de activación de un aspecto:

```
public abstract class CtxActive implements Active {
    protected Context itsContext;
    public CtxActive(Context c){
        itsContext = c;
    }
    public boolean evaluate(Object o){ // método requerido por Interfaz Active
        return getCtxState(o) != null;
    }
    public abstract ContextState getCtxState(Object o);
}
```

En el constructor se almacena, en una variable de instancia, el contexto asociado. El método `evaluate()` retorna verdadero siempre y cuando el *link* asociado esté activo. Esto lo realiza a través de `getCtxState()`, método abstracto que se encarga de obtener el estado del contexto. No se define inmediatamente porque, como veremos más adelante, existen aspectos cuya activación depende de contextos que estuvieron vigentes en el pasado (snapshots).

A continuación, en la siguiente subsección, será presentada la implementación de una condición de activación de este tipo. Como ya están definidos los contextos y se ha definido por lo menos una condición de activación, se presentará un análisis detallado con la ayuda de diagramas de secuencia.

3.1. Ejemplo de Condición de activación: CurrentlyInCtx

Cuando se busca evaluar si un contexto se encuentra activo exactamente en el momento de requerirlo estamos en presencia de un contexto cuya restricción es estar `CurrentlyInCtx` (*currently in context*). La implementación de este contexto es simple ya que no tiene dependencias con el pasado, y como ya fue mencionado, este parte como una extensión a `CtxActive`:

```
public class CurrentlyInCtx extends CtxActive {
   public CurrentlyInCtx(Context c){
       super(c);
```

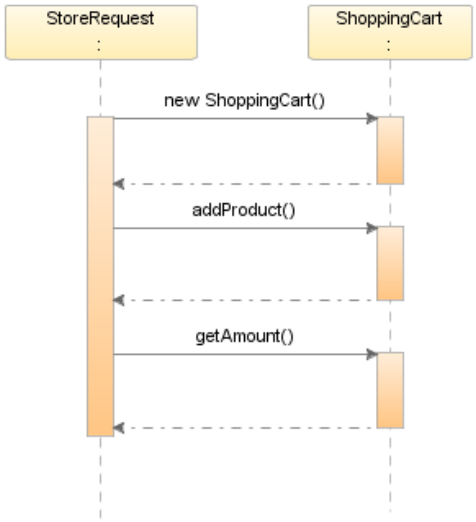

Figura IV.3: Diagrama de secuencia de una compra en el e-Shop

```
    }
    public Context.ContextState getCtxState(Object o){
        return itsContext.getState();
    }
}
```

Efectivamente la implementación de `getCtxState()` es muy simple ya que lo único que hace es obtener el estado del contexto que esta definido para esta condición de activación.

3.2. Análisis del diagrama de secuencia

Para entregar más luces que permitan entender como se van ligando los contextos y las condiciones de activación de los aspectos asociados será analizado con diagramas de secuencia el caso de una `TimeFramePromotionCtx` y su condición de activación `CurrentlyInCtx`.

El nivel base se presenta en la figura IV.3. Este caso utiliza el *shopping cart* desde una clase `StoreRequest` que simula ser una tienda comercial cualquiera.

Desde `StoreRequest` se crea el *shopping cart*, se agregan productos y al final se obtiene el total a pagar. Justamente en esta parte es donde se aplicará la condición de activación.

Cuando se obtiene el monto final llamando al `getAmount()` se gatilla el diagrama de secuencia de la figura IV.4.

Al hacer el `getAmount()` se invoca inmediatamente la linea de ejecución del `ApplicationLink`. Esta capa para el framework es una caja negra, porque está implementada por Reflex. En esta capa se obtienen los parámetros del contexto (el `rate`), para pasarlos al `DiscountMO`[3]. Antes de realizar esto es necesario validar si el contexto está vigente o no. Para esto se realiza el `getCtxState()` al `CurrentlyInCtx`, que a su vez llamará al `getState()` del `TimeFramePromotionCtx`.

[3]La obtención de parámetros se implementa en la clase `ContextParameter`. Esta posee un método `evaluate` que genera una linea de código (en un `String`) que será agregada al objeto que está siendo manipulado (en este caso el `ShoppingCart`) y que en tiempo de ejecución será llamada para que retorne el parámetro (en este caso el `rate`).

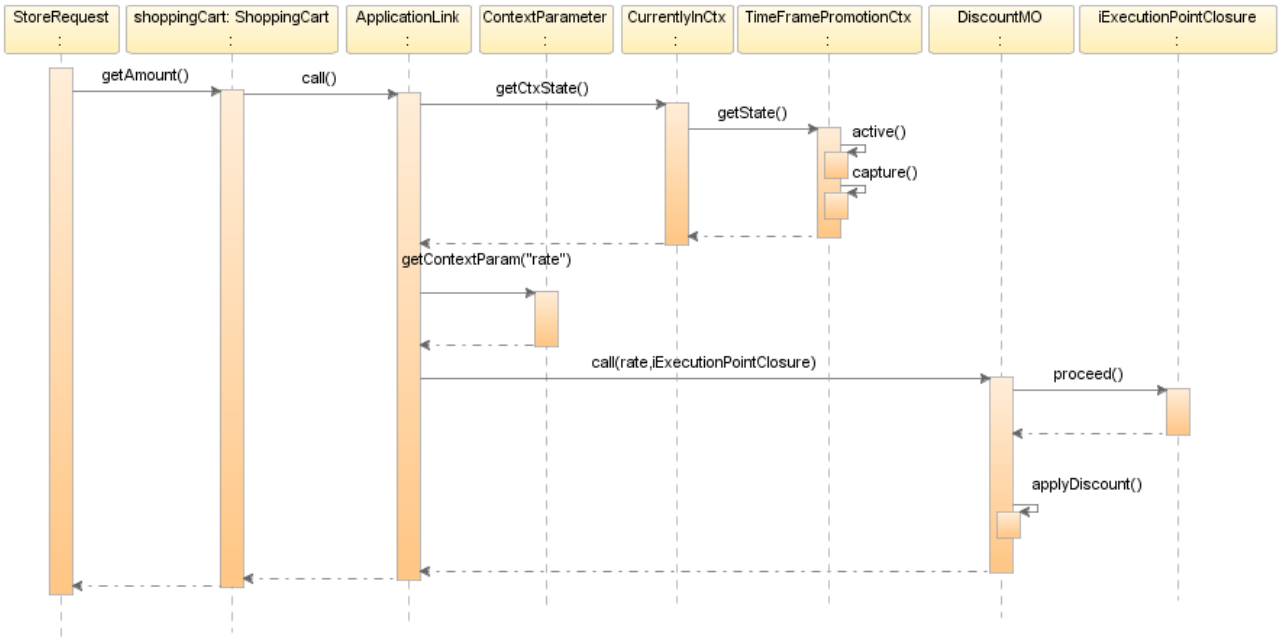

Figura IV.4: Diagrama de secuencia de aplicación de un descuento bajo un TimeFramePromotionCtx

Luego de esto se aplica el descuento en `DiscountMO`. Para obtener el objeto en ejecución se utiliza un objeto `IExecutionPointClosure` proveído por Reflex, y que se utiliza para obtener el valor original que tiene que retornar `getAmount()` (se obtiene con `proceed()`). Finalmente se retorna todo y `StoreRequest` recibe el monto de la compra con los descuentos aplicados.

4. Dependencia con el Pasado

Se pueden definir condición de activación que permitan rescatar el estado de un contexto que en algún momento estuvo vigente. Para el caso del *shopping cart* esto serviría para controlar un descuento que se aplica si este fue creado(`CreatedInCtx`) o si se le agrega un producto (`PutInCartCtx`) cuando una promoción estaba vigente. Esto resulta útil, porque en ambos casos el *check out* se produce con un desfase temporal con respecto a los eventos de crear el *shopping cart* o agregar un producto. Por esta razón, hay que diseñar una forma que permita mantener guardados estos contextos (*snapshot*) y su `ContextState` respectivo en el momento en que se produjeron dichos eventos.

El soporte para definir estas dependencias se encuentra contenido en la clase abstracta `SnapshotCtxActive`.

```
public class SnapshotCtxActive extends CtxActive {
    public SnapshotCtxActive(Context c){
        super(c);
    }
    public ContextState getCtxState(Object o){
        Snapshot snapshot = getSnapshot(o);
        return (Context.ContextState)snapshot.get(itsContext);
    }
```

```
    ...
}
```

Cuando se evalúa el estado del contexto a través de `getCtxState()` se extrae el snapshot correspondiente al objeto que está actualmente en ejecución. Este snapshot es consultado entonces sobre el estado asociado al contexto que contenía.

Existen básicamente dos opciones para almacenar los snapshot: manejar un repositorio global (por ejemplo un `Map` de objetos) de snapshots o agregar a los objetos un atributo extra que va a contener un *snapshot*. La elección de la segunda opción radica fundamentalmente en que ésta elimina el cuello de botella generado al tener un repositorio central. Otra razón no menos importante es que si se almacenan todos los contextos en un repositorio, jamás serán recogidos por el *Garbage Collector*, porque siempre existirá un referencia a ellos. La opción tomada supone la implementación de un *framework* de manejo de anotaciones a nivel de objetos, basado en Reflex, especialmente en la capacidad que tiene de realizar cambios estructurales a las clases.

Para manejar la dependencia del pasado, la clase `SnapshotCtxActive` implementa también los métodos:

- void annotate(ClassSelector cs): activa la capacidad de recibir snapshots a las clases seleccionadas.
- void snapshot(Hookset hs, Parameter p): al activarse el hookset almacena el snapshot del contexto dentro del parámetro p.
- Snapshot getSnapshot(Object o): retorna el snapshot almacenado en el objeto o.

La implementación de estos métodos faltantes en el `SnapshotCtxActive` es:

```
public class SnapshotCtxActive extends CtxActive {
     public void annotate(ClassSelector cs){
        JAnnote.annotateInstancesOf(cs);
    }
    public BLink snapshot(Hookset snapshotHookset,Annotator aAnnotator,
                          Parameter[] parameters){
       return JAnnote.annotateWhen(snapshotHookset, aAnnotator,parameters);
    }
    Snapshot getSnapshot(Object o){
        Snapshot snapshot =(Snapshot)((Annotatable)o).get(itsContext);
        return snapshot;
    }
}
```

Antes de pasar a un ejemplo de un aspecto dependiente de una condición pasada será presentado en forma breve el pequeño framework de anotaciones implementado para manejar el almacenamiento de *snapshots*.

4.1. JAnnote: Framework de Manejo de Anotaciones

Una anotación sobre una clase consiste en dejar almacenado en ella una nota, representada por un objeto. Este objeto puede ser accedido por los correspondientes métodos `get()` y `set(Object o)`.

Para generalizar la noción de *anotable*, es necesario que los objetos que pueden recibir anotaciones implementen la interfaz `Annotatable` cuyo único fin es dar posterior acceso a los métodos `get()` y `set()`.

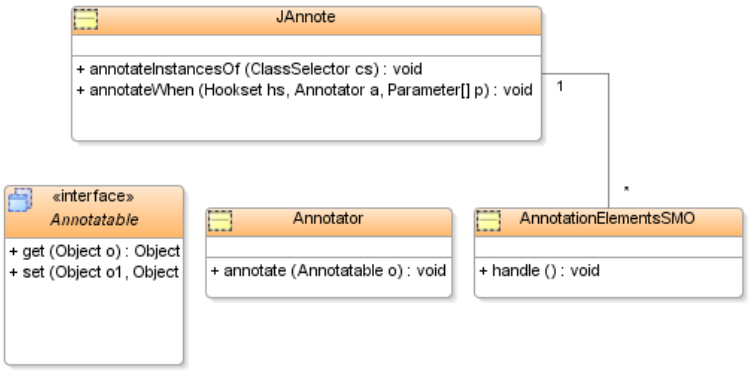

Figura IV.5: Diagrama de Clases de JAnnote

Este es un problema recurrente, por esta razón se implemento de forma separada del framework de context-aware aspects.

Las funcionalidades esperadas de este framework son las siguientes:

- Indicar que clases serán capaces de recibir anotaciones.
- Agregar el atributo y sus correspondientes métodos `get()` y `set()`.
- Definir bajo que circunstancias se va a producir la anotación.

Estos objetivos son logrados por *JAnnote*, el framework de anotaciones implementado en Reflex. Un ejemplo de uso puede ser apreciado en la implementación de la clase `SnapshotCtxActive` presentada anteriormente.

El diagrama de clases de *JAnnote*, en la figura IV.5, presenta la estructura completa del framework de anotaciones. La clase principal es `JAnnote`, en ella se proveen dos métodos:

- `annotateInstancesOf(ClassSelector cs)`: convierte en implementaciones de `Annotatable` a las clases indicadas por el `ClassSelector`. Para realizar esto utiliza el meta objeto estructutal `AnnotationElementsSMO`.
- `annotateWhen(Hookset hs, Annotator a, Parameter[] p)`: Indica sobre que objeto (`Parameter`) y para que operación (*hookset*) el `Annotator` va a guardar la anotación.

Dentro de estos dos métodos se implementan todas las funcionalidades necesarias para el framework de anotaciones.

4.2. Ejemplo de Condición de activación: CreatedInCtx

Para el caso de ejemplo mostrado en la sección 2, un uso concreto de una condición de activación que hace referencia a un contexto pasado es preguntar acerca de si al momento de crear el shopping cart existía un contexto activo. Este es la condición de activación `CreatedInCtx`. Es una extensión de `SnapshotCtxActive`, tomando de ella el método de validación de su condición de activación.

En la definición de una restricción del tipo `CreatedInCtx` es necesario primero dejar como `Annotatable` todos los objetos que van a estar sometidos al contexto y definir la condición de aplicación de Snapshot, indicando específicamente sobre que objeto será aplicado.

Su implementación es la siguiente:

```
public class CreatedInCtx extends SnapshotCtxActive {
    public CreatedInCtx(Context c, BLink l){
        super(c);
        ClassSelector cs = l.getHookset().getClassSelector();
        annotate(cs);
        Hookset snapshotHookset =new PrimitiveHookset(Instantiation.class,
                AllCS.getInstance(),
                new TargetsTypeOS(cs));
        SnapshotAnnotator aAnnotator = new SnapshotAnnotator(c);
        snapshotLink = snapshot(snapshotHookset, aAnnotator,
                new Parameter[]{ new Parameter.TypeWrapper(Annotatable.class,
                                                           Parameter.RESULT)});
    }
}
```

Los métodos `annotate()` y `snapshot()` que ya provee `SnapshotCtxActive` son utilizados para dejar anotables las mismas clases a las que `BLink` aplicará el contexto, y para definir sobre qué objeto y qué metaobjeto (`SnapshotAnnotator`) será el encargado de realizar el *snapshot*. `TargetsTypeOS` se implementó de manera tal de rescatar todas las clases apuntadas por el `ClassSelector` independiente de la implementación del hookset que haya sido utilizada.

5. Incorporación de WildCAT

En la sección 3.6.2 fue presentado WildCAT. Es importante destacar que uno de sus objetivos es otorgar un API simple y fácil de extender.

En la primera parte se presentará cuál es la lógica que maneja WildCAT para obtener los datos del contexto. Este posee dos medios, a través de *Samplers* o a través de *Sensores*.

La definición de sensores será presentada a través de dos ejemplos: `CPULoadSensor` y `LocationSensor`. El primer sensor ya es proveído por WildCAT, mientras que el segundo fue implementado específicamente para aplicarlo al caso de ejemplo. Los samplers no fueron utilizados, porque como veremos su funcionamiento no aplica a los comportamientos esperados por el caso de ejemplo.

5.1. Obtención de Datos con WildCAT

WildCAT permite obtener la información de los contextos de dos formas: a través de *sensores* y a través de *samplers* (muestreos).

En el primer caso es posible pedir a WildCAT que devuelva un valor específico en el momento en que es requerido. Esto significa que el sensor toma una muestra en el momento que se lo solicitan o automáticamente cuando se satisface una condición. En este último caso se puede manipular el contexto sólo cuando ocurra, en caso contrario nunca se toma el control de la ejecución porque nunca se dispara la llamada.

Por otro lado, un *sampler* es capaz de ir haciendo muestreos programados, definidos mediante un *timer*. Estos datos son retornados para su manipulación como un bloque y posteriormente a que hayan ocurrido.

Para la activación de los contextos que están siendo ejemplificados a lo largo de esta tesis, es necesario contar en cualquier momento (en la medida que es requerido) con la información que indique si un contexto esta activo o no. Esto lleva inmediatamente a descartar los *samplers* como opción para obtener información. Esto se debe fundamentalmente a que los *samplers* funcionan en

base a la toma de muestras sucesivas (un sensor se ejecuta muchas veces) y el resultado es retornado a un objeto que está esperando dichas respuestas.

Los sensores automáticos (con condición de activación) tienen el inconveniente de que no siempre devuelven información, ya que el objeto que espera su respuesta nunca se activa si este no es llamado directamente. Para el caso de un sensor de localización implementado de esta forma, podríamos saber cuando un cliente compra en el departamento de deportes, pero nunca sabríamos si compra en otra parte.

Si bien es cierto los comportamientos son distintos, de igual modo se puede acceder a los datos del contexto directamente, excepto en el caso de los sensores automáticos, ya que estos sólo dejan información cuando se activan y no en todo momento. Por eso se implemento un capturador de contexto en base a sensores y otro en base a un sampler.

5.2. Sensores

En primer lugar presentaremos un sensor capaz de indicar el nivel de carga de la máquina que lo esta corriendo. La implementación de este sensor fue tomada directamente de WildCAT.

Posteriomente presentaremos un sensor de localización, específicamente diseñado para el caso de ejemplo, ya que simula el efecto de estar en un lugar u otro.

5.2.1. Utilizando un Sensor WildCAT: CPULoadSensor

Dentro de un servidor Linux (o cualquier servidor basado en Unix) la forma de conocer el nivel de carga es a través del comando *top*. Este comando lo que hace es ordenar la información contenida en el directorio `/proc` del *filesystem* del sistema operativo. La información específica del procesador está en el archivo `/proc/cpuinfo`. Dentro de esta información aparece la carga del procesador.

El CPULoadSensor lo que hace es *reificar* el archivo `/proc/cpuinfo` entregando la información del procesador como información relativa al contexto de ejecución.

Dado que la información viene directamente de un archivo, este sensor se encarga de permitir acceder a la misma información, pero en forma directa. En Linux esto resulta de mucha utilidad, dado que mucha información del *environment* se encuentra especificada en archivos.

Algunos de los tipos de datos que son posibles de obtener con este sensor son:

- Identificación del procesador (fabricante y nombre).
- Características técnica (familia, cantidad de núcleos, velocidad, tamaño del cache, etc).
- Promedio de carga del procesador.
- Cantidad de tareas en ejecución, entre otros.

La obtención de los datos de muestra se realiza de la siguiente forma:

```
SampleSet sSet= new CPULoadSensor().getSample();
```

Un objeto `SampleSet` contiene contiene todos los datos de una muestra. Para extraer un dato hay que conocer la llave que lo identifica. Por ejemplo para extraer la carga promedio dentro del último minuto transcurrido se utiliza "*avg1*":

```
Double cpuLoad=(Double)sSet.getSampleValue("avg1");
```

Con `getSampleValue("llave")` se extrae desde la muestra el valor indicado por *llave*.

5.2.2. Definiendo un Sensor en WildCAT: LocationSensor

En este caso fue utilizado un sampler para implementar la captura del contexto. Para simular el cambio de ubicación se toma dicha información desde un objeto `LocationManager` con varias ubicaciones posibles. Al solicitar una muestra la información de la ubicación debe ser actualizada y entregada para su utilización.

La implementación en WildCAT es la siguiente

```
public class LocationSensor implements Sampler {
    private String place;
    public SampleSet sample() {
        SampleSet samples = new SampleSet(TimeStamp.now());
        place = LocationManager.getPlace();
        samples.addSample("place", place);
        return samples;
    }
}
```

En el método `sample()` lo que se hace crear el `SampleSet` correspondiente a este sensor. Para agregar valores a la muestra se utiliza el método `addSample()` del `SampleSet`. Los parámetros son la *llave* y el valor que corresponde.

Desde el otro lado, para obtener los resultados desde este sensor, basta con utilizar las siguientes líneas de código:

```
SampleSet s=new UbicationSensor().sample();


 String place = (String)s.getSampleValue("Current user location"));
```

Al igual que en el CPULoadSensor obtenemos el valor de la medición con `getSampleValue()`.

Ambos ejemplos son capaces de ilustrar que la utilización de WildCAT está bien encapsulada y es directa. Con una simple llamada puede ser obtenida información instantánea acerca del contexto de ejecución.

5.3. Integración de WildCAT con el Framework CAA

En la sección 4.2 ya fue establecido que el método `getState()` de las clases que implementan `Context` es el encargado de devolver el estado de un contexto cualquiera. Modificando la condición que aparece dentro de este método se logra definir si un contexto se encuentra activo o no.

Por otra parte, en la sección anterior se mostró cual es la forma de utilizar WildCAT para obtener información del contexto. Resulta ser una forma simple y directa.

Juntando ambos elementos no es difícil concluir que la forma natural de integrar ambos framework ocurre directamente en el getState().

Los siguientes 2 ejemplos muestran como se utilizan los dos sensores definidos, mostrando además como quedan fácilmente integrados al framework de context-aware aspects.

Dado que ambos casos son también `PromotionCtx`, la lógica de definición de contextos ya fue presentada(ver figura IV.6). Entonces sólo se presentará la implementación del método `active()` para ambos casos.

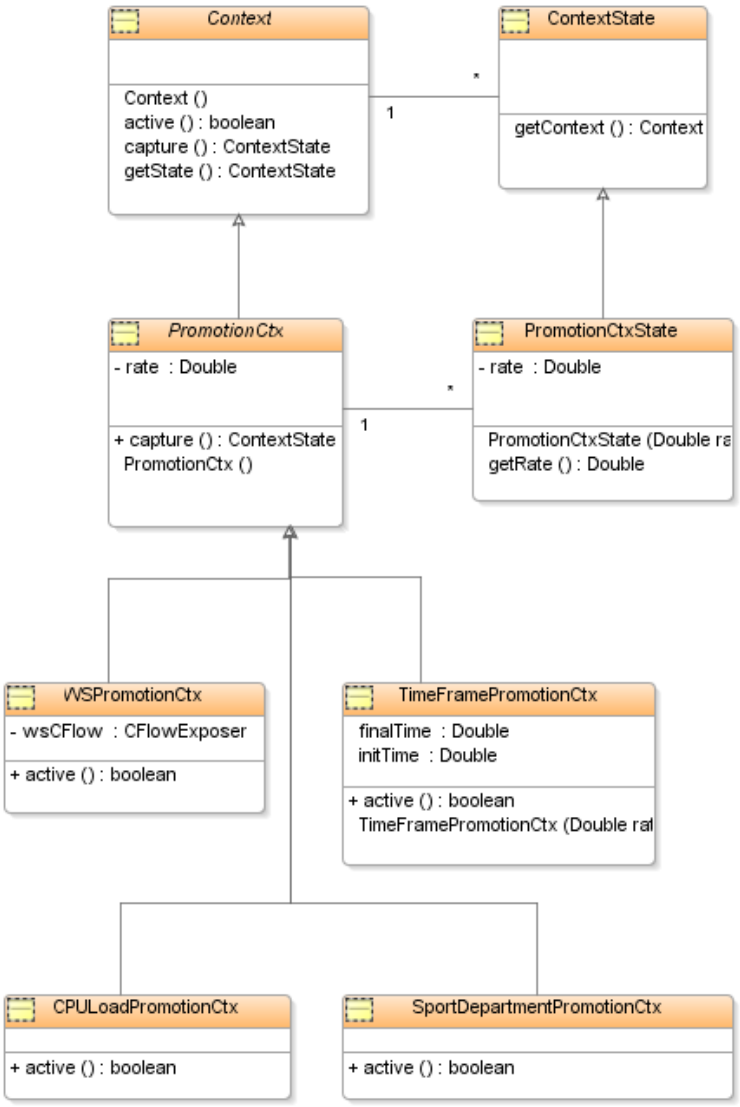

Figura IV.6: Diagrama de Clases de PromotionCtx, junto con todas las clases que lo extienden

5.3.1. Ejemplo Utilizando un CPULoadSensor

Este contexto de promoción premia la realización de compras en horarios de poco público. Esto podría ser implementado con un `TimeFramePromotionCtx`, indicando específicamente las horas de baja concurrencia. Pero una forma mucho más directa es reconocer una baja concurrencia por la baja carga del servidor. Como fue presentado anteriormente, el `CPULoadSensor` es capaz de informar justamente acerca de esto. Entonces, el `CPULoadPromotionCtx` estará activo siempre y cuando la carga se encuentre dentro bajo un umbral definido.

La implementación de su método `active()` sería:

```
public boolean active(){
    SampleSet sSet= new CPULoadSensor().getSample();
    return ((double)(Double)(sSet.getSampleValue("avg1"))>MAX_CPU_LOAD);
}
```

5.3.2. Ejemplo Utilizando un UserLocationSensor

Por diversos motivos puede ser necesario estimular las ventas de un departamento de una tienda comercial. Para esto se pueden crear promociones especiales que premien este comportamiento. En particular, es presentado un contexto de promoción que premia la compra en el departamento de *Deportes*: `SportDepartmentPromotionCtx`.

Al igual que en caso anterior, solamente definimos su método `active()`:

```
public boolean active(){
    SampleSet sSet= new UserLocationSensor().getSample();
    return "Sport".equals(sample);
}
```

Ambos ejemplos ilustran de forma clara que independiente de la forma en que los sensores recuperen la información, dada el API definido para WildCAT, es posible rescatar de forma casi directa una muestra relativa al contexto definido, para cualquier caso, en particular, para el `CPULoadPromotionCtx` y para `SportDepartmentPromotionCtx`

5.4. Configuración del Framework

La configuración de los context-aware aspects es simple. Se divide de la misma forma en que se divide el framework: una parte de definición de contextos y otra de las condiciones de activación.

5.4.1. Configuración: Definición de Contextos

Para definir un contexto basta con obtener una instancia de su clase.

Para el ejemplo del `WSPromotionCtx` hacemos:

```
WSPromotionCtx wsPromo = new WSPromotionCtx(customRate);
```

Los parámetros del constructor dependerán específicamente de la implementación del contexto. Para el `TimeFramePromotionCtx` estos cambian dado que necesita la hora inicial y final de la ventana de tiempo:

```
TimeFramePromotionCtx tfPromo =
                  new TimeFramePromotionCtx(customRate,initialHour,finalHour);
```

Con esto ya contamos con un contexto definido. Hay que proceder a generar las condiciones de activación.

5.4.2. Configuración: Condiciones de Activación

Las condiciones de activación requieren un poco más de trabajo para ser definidas. Pero en general los pasos son los mismos. Esta puede hacerse en varios lugares. Es decisión del implementador de ella decidir donde será mejor ubicarla.

Para el `TimeFramePromotionCtx` mostraremos las dos configuraciones, con `CurrentlyInCtx` y con `CreatedInCtx`.

A continuación se presenta el caso con `CurrentlyInCtx`. Primero serán presentados los elementos básico antes de pasar a la configuración de la condición de activación.

Los elementos básicos son comunes a ambas condiciones de activación, y son:

```
//Obtenemos instancia del metaobjeto de descuento
DiscountMO tfMO = new DiscountMO();
MODefinition theMO = new MODefinition.SharedMO(tfMO);
//Creamos el hookset que hará la intercepción de la llamada al getAmount
Hookset theHookset = new PrimitiveHookset(
                       MsgReceive.class,
                       new NameCS("reflex.caa.demo.app.ShoppingCart"),
                       new NameOS("getAmount"));
//Creamos el Link entre el metaobjeto y el hookset
BLink applicationLink = API.links().createBLink(theHookset, theMO);
//se pueden definir algunas otras propiedades del link
applicationLink.setControl(Control.AROUND);
applicationLink.setName("ApplicationLink");
```

Esta primera parte es completamente proveída por Reflex. No es necesario entrar en más detalles sobre ella.

La condición de activación `CurrentlyInCtx` se realiza de la siguiente forma:

```
// creamos la condición de activación
CtxActive currentlyInPromo
            = new CurrentlyInCtx(
                     newTimeFramePromotionCtx(.2,initTime,finalTime)
                                );
// agregamos la condición de activación al link
applicationLink.addActivation(currentlyInPromo);
// configuramos la llamada al metaobjeto
applicationLink.setCall("reflex.caa.demo.mo.DiscountMO",
       "amount",
       new Parameter[]{
           ((CurrentlyInCtx)currentlyInPromo).getCtxParam("rate"),
           Parameter.CLOSURE
       });
```

Casi la totalidad de estos pasos son proveídos por el framework. Sólo el último (`setCall()`) es parte básica de Reflex.

Para crear una condición de activación `CreatedInCtx` basta con hacer:

```
CtxActive createdInPromo
            = new CreatedInCtx(
                     newTimeFramePromotionCtx(.2,initTime,finalTime)
                                );
```

```
applicationLink.addActivation(createdInPromo);
applicationLink.setCall("reflex.caa.demo.mo.DiscountMO",
       "amount",
       new Parameter[]{
           ((CreatedInCtx)createdInPromo).getCtxParam("rate"),
           Parameter.CLOSURE
           });
```

Las llamadas son muy similares, sólo cambia la clase que implementa la condición de activación.

6. Resumen

Dado que los problemas que aparecen al implementar aplicaciones context-aware son muy similares, y que se ha definido una estructura estándar para implementarlas a través de aspectos, es posible construir un framework que unifica y encapsula todas estas definiciones. Este framework se divide en dos partes: definición de contextos e implementación de condiciones de activación.

Teniendo en cuenta el caso de ejemplo del *shopping cart*, se pueden definir diversos contextos de promoción, como `WSPromotionCtx` o `TimeFramePromotionCtx`. A su vez es posible definir condiciones de activación como: `CurrentlyInCtx` o `CreatedInCtx`.

Si es necesario obtener información del contexto de ejecución para que una promoción se active (como la ubicación o niveles de carga del servidor) es posible incorporar WildCAT. Este posee diversos sensores que pueden ser utilizados en forma sencilla. Además provee los elementos básicos que permiten construir sensores nuevos.

Capítulo V

Aplicación del Framework al Caso de Estudio

Teniendo presente los requisitos para la implementación del framework de context-aware aspects, y la consiguiente implementación que ha sido mostrada, es posible definir para los contextos creados y las dos condiciones de activación las pruebas que permitan validar su funcionamiento, junto con los resultados esperados para cada caso.

Las pruebas tomarán el caso de ejemplo presentado en el capítulo II. Serán aplicados los diversos contextos y las condiciones de activación definidas bajo un ambiente controlado por una aplicación de simulación de ventas (nivel base) y un manejador de promociones (meta nivel). El shopping cart será intervenido con el framework para que al momento de hacer un *check out* se apliquen los descuentos si corresponden.

La sección 1 de este capítulo se encargará justamente de definir estos casos de prueba, tanto para las promociones simples (secciones 1.1 y 1.2) como para aquellas que usan WildCAT (secciones 1.3 y 1.4). En la siguiente sección será revisado el software de pruebas, tanto en el nivel base como en el meta nivel. Por último en la sección 3 serán presentados los resultados obtenidos para cada caso.

1. Definición de Casos de Prueba

Existen varias configuraciones posibles para probar los contextos y las condiciones de activación. De estos elementos contamos hasta el momento con:

Condiciones de activación:

- CreatedInCtx: objeto creado cuando el contexto estaba activo.
- CurrentlyInCtx: actualmente contexto está activo.

Contextos:

- WSPromotionCtx: contexto activo cuando esta dentro del flujo de ejecución del Web Service.
- TimeFramePromotionCtx: contexto activo si la hora actual está dentro de una ventana de tiempo.
- CPULoadPromotionCtx: contexto activo si la carga del procesador es baja.

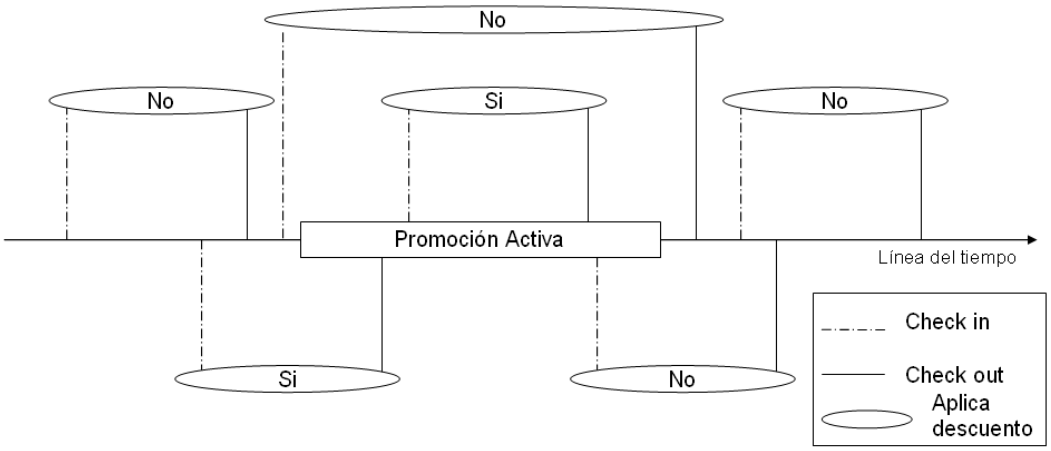

Figura V.1: Diagrama de Casos para CurrentlyInCtx

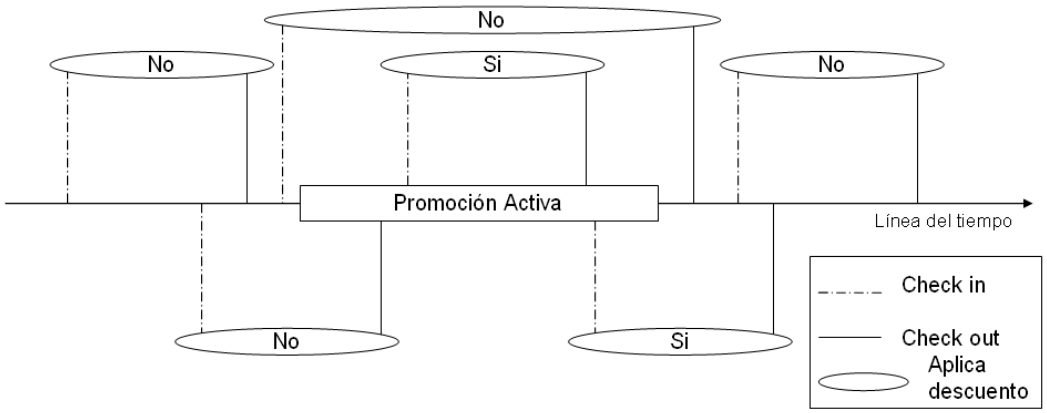

Figura V.2: Diagrama de Casos para CreatedInCtx

- SportDepartmentPromotionCtx: contexto activo si la ubicación actual es en el departamento de deportes.

La combinación de contextos y condiciones crean un total de $4*2 = 8$ casos base. Posteriormente analizaremos la aplicabilidad de cada uno.

A estos 8 (4 para cada condición de activación) casos base hay que agregarle otra variable: antes, durante y después que el contexto esté activo. Esto se interpreta en forma distinta para el `CurrentlyInCtx` y el `CreatedInCtx`.

Para el `CurrentlyInCtx` lo que importa siempre es el *durante*, o sea, que el *check out* del shopping cart se produzca *durante* una promoción. Es independiente del momento de inicio de la compra. Un esquema que muestra todos los casos posibles se encuentra en la figura V.1.

Esto agrega 6 nuevos casos de prueba, lo que da un total de $4*6 = 24$ casos de prueba sólo para el `CurrentlyInCtx`.

Para el `CreatedInCtx` el mismo esquema se desarrolla en la figura V.2. Las posibilidades de aplicar el descuento cambian. El `CreatedInCtx` es claramente más restrictivo, porque efectivamente el shoppingCart tiene que haber sido creado en el contexto, con lo que todo lo que ocurra *antes* no tiene ninguna validez. También se agregan 6 nuevos casos de prueba, lo que da un total de $4*6 = 24$

casos de prueba sólo para el `CreatedInCtx`.

Por lo tanto, existen $24+24 = 48$ casos de prueba que aseguran haber probado todos los posibles escenarios. De todas maneras, estos 48 casos pueden ser disminuidos, porque hay que analizar si para cada `PromotionCtx` implementado tiene sentido utilizar ambas condiciones de activación. Además, como la implementación de los contextos es independiente de la implementación de las condiciones de activación, será aceptable incluso probar con un sólo contexto las 12 posibles situaciones de activación (con `CreateInCtx` y `CurrentlyInCtx`) siempre y cuando tengan sentido.

El detalle del número de pruebas para cada `PromotionCtx` se detalla en las siguientes secciones.

1.1. Pruebas para Promociones de WebService

Esta promoción indica que si al hacer el *check out* el shopping cart está dentro del flujo de ejecución del Web Service, entonces tiene descuento.

Dada la configuración del Web Service que se está utilizando, en le mismo método `perform()` se crea el shopping cart y se realiza el check out, por lo tanto un `CreatedInCtx` y un `CurrentlyInCtx` son exactamente iguales.

Las pruebas para el Web Service se realizarán entonces en el `CreatedInCtx`, para los casos especificados en la tabla 2(6 casos).

1.2. Pruebas para Promociones de Time Frame

Esta promoción está activa siempre y cuando ocurra dentro de una ventana de tiempo definida.

Contrario al caso anterior, el caso del `TimeFramePromotionCtx` tiene sentidos distintos en el `CurrentlyInCtx` y en el `CreatedInCtx`. De hecho, los comportamientos esperados son diferentes.

Manejando la hora actual, el `CurrentlyInCtx` y el `CreatedInCtx` se pueden simular perfectamente sus 12 casos de prueba, especificados en las tablas 1 y 2.

1.3. Pruebas para Promociones de CPU Load

Dado que esta prueba se refiere a la carga instantánea del procesador tiene sentido realizarla para el `CurrentlyInCtx`.

Teniendo en cuenta que este contexto no es fácil de simular, dado que la cantidad de carga máxima es muy relativa y no se puede asegurar su estado, se opta por realizar una prueba con un límite bajo de carga máxima (para que se comporte como que estuviera muy cargado) y una con un límite muy alto de carga (para que se comporte como si no tuviera mucha carga). Con esto, definimos que habrán 2 casos de prueba para `CPULoadPromotionCtx`

1.4. Pruebas para Promociones de Location

Este último caso aporta más que nada en las pruebas de la integración con WildCAT. Serán realizadas para dar completitud y no dejar dudas acerca de la implementación correcta del framework. Teniendo esto en cuenta se probarán sólo dos casos.

Este `PromotionCtx` tiene características similares al anterior. Dado que es relevante que el *check out* se produzca en el departamento de deportes, independiente de donde se haya empezado a comprar. Por este motivo, y para no obligar al `PromotionCtx` a comportarse en un escenario forzado, los dos casos serán los dos primeros del `CurrentlyInCtx` de la tabla 1.

Figura V.3: Interfaz del e-Shop

2. Software de Prueba

Las pruebas se realizarán sobre una pequeña aplicación que tiene tres vistas: *e-Shop*, *Configurador e-Shop* y *Manejador de Promociones*. El *e-Shop* implementará las funciones principales de una tienda con su shopping cart. El Configurador *e-Shop* permitirá configurar las opciones del sistema, simulando el contexto actual (hora, ubicación y tipo de usuario). Mientras que el Manejador de Promociones permitirá para cada promoción, activarla y desactivarlas a gusto del usuario.

A continuación presentaremos cuál es el objetivo de construir esta pequeña aplicación y luego será descrito cada uno de los niveles.

2.1. Objetivo

El objetivo será automatizar el proceso de pruebas dentro una sola aplicación de nivel base y un único meta nivel, cubriendo todos los escenarios definidos en los casos de prueba.

Los objetivos específicos de cada nivel son:

- Nivel Base: Presentar en una aplicación las funcionalidades básicas de una tienda con su shopping cart: entrar al local, agregar productos, comprar y recibir detalle de la compra.
- Meta Nivel: Intervenir el nivel base aplicando o eliminando los contextos y condiciones de activación definidos en los casos de prueba.

2.2. Aplicación e-Shop

Esta interfaz implementa una aplicación de ventas que simula una tienda o local.

Los usuarios tienen que poder ser reconocidos como usuarios web (pensando en `WSPromotionCtx`) o usuarios presentes en la tienda (cualquier otro contexto). Además tiene que existir la posibilidad de modificar el reloj de la tienda, para las promociones de `TimeFramePromotionCtx`. Estos dos elementos serán agregados a la interfaz del simulador de tienda (ver figura V.3).

Si se realiza alguna compra, la aplicación se comporta de forma esperada, agregando los productos al shopping cart, pagando, y obteniendo un detalle. En este último aparece indicado un descuento, que para este nivel siempre será cero, ya que no es posible aplicarlo a través de la aplicación.

Figura V.4: Interfaz del Configurador del e-Shop

Figura V.5: Interfaz del Manejador de Promociones

2.3. Configurador del ambiente

Como se puede ver en la figura V.4, en esta interfaz se permiten realizar configuraciones del entorno del *e-Shop*, como el tipo de usuario y el reloj.

Además, en esta interfaz es posible seleccionar la ubicación del usuario, necesaria para utilizar el `SportDepartmentPromotionCtx`. Esto es así, porque los sensores se encuentran absolutamente en el meta nivel. La aplicación base ni siquiera sabe que la ubicación del usuario es monitoreada.

2.4. Manejador de Promociones

El administrador de promociones tiene su propia interfaz, separada del nivel base. Esta completamente implementado utilizando el framework de context-aware aspects para manejar las promociones.

La interfaz permite seleccionar las promociones disponibles, cada una con su respectiva condición de activación. No fueron separadas, para no complicar innecesariamente la aplicación de pruebas. Mantenerlas juntas o separadas no agregan ningún valor al comportamiento esperado. El aspecto de la interfaz puede ser visto en la figura V.5

3. Resultados Obtenidos

La totalidad de las pruebas fueron implementadas en un *test JUnit*, de tal manera de poder realizarlas multiples veces para todos los escenarios. Al ejecutar el test se corroboró que los resultados esperados fueron obtenidos.

Pero además de los resultados obtenidos al aplicar los distintos escenarios de prueba, es posible decir que se han obtenidos resultados a nivel de la aplicación implementada. Esto es mostrado en las secciones 3.1 y 3.2 de este capítulo. En la sección 3.3 se muestra cual fue el nivel de administración alcanzado para los contextos en la implementación del framework.

3.1. Efectiva Separación de Intereses

Con la implementación hecha es posible decir que se logró SoC a varios niveles.

En primer lugar, dentro del mismo framework. La separación de los contextos y de las condiciones de activación efectivamente hace que se separen las preocupaciones del framework de *context-aware* aspects. Esto fue pensado desde un principio y fue una de las razones para dividir el framework en una parte de *definición* de contextos y otra de *restricciones* sobre aspectos que implementan contextos.

En segundo lugar, pensando en el caso de estudio, se logra efectivamente separar el shopping cart de las promociones. El framework permite implementar un meta nivel de aplicación de aspectos dependientes de contextos que van a influir realmente sobre el nivel base. Esto ha sido corroborado con la implementación de la aplicación de prueba y del administrador de contextos.

3.2. Codificación

Al implementar la aplicación de prueba se logró el objetivo específico para el cuál fue pensada. De hecho podía ser utilizada por si sola, obteniendo el comportamiento esperado, acorde con la implementación realizada. Esta aplicación se puede concebir como una cápsula que no posee dependencias externas de ningún tipo.

En el desarrollo de la aplicación del meta nivel no fue necesario intervenir ninguna línea de código del nivel base para modificar su comportamiento o lograr algún tipo de conexión con el meta nivel. Esto efectivamente muestra que en la práctica (en la codificación) y no tan sólo en un nivel abstracto (de diseño) se logra SoC.

3.3. Administración de un Contexto por Objeto

El framework permite administrar contextos de forma individual sobre un mismo objeto. Esto es una limitante de la implementación desarrollada. Dar el siguiente paso agrega una alta complejidad. El salto a tener *composición de contextos* no es trivial y hace parte de un proyecto de investigación en sí mismo.

Si bien es cierto puede estar atento un solo contexto a la vez, los otros pueden estar disponibles para ser utilizados en cualquier momento. Esta implementación se realizó en el manejador de promociones y demostró un correcto funcionamiento. Otra alternativa es manejar una sola promoción con varias condiciones de activación. Esto último también se puede hacer con el framework.

4. Resumen

Las pruebas definidas colocaron al caso de ejemplo en todos los escenarios posibles. Un análisis posterior saca a la luz que también es posible definir contextos sin sentido, que no aplican y sobre

los cuales es inútil definir pruebas.

La construcción del software de prueba entregó flexibilidad y rapidez para para la realización de las pruebas. La clara separación entre un nivel base y un metanivel son también una prueba de la correcta implementación del framework, ya que se logra una efectiva SoC entre ambos niveles.

Más que importar los valores obtenidos en los escenarios de prueba, resulta de vital importancia reconocer que los resultados más valiosos están en la efectiva SoC conseguida, la facilidad de codificación del framework y la posibilidad de administrar los contextos.

Capítulo VI

Conclusiones

Las últimas observaciones acerca del trabajo realizado serán separadas en diversas secciones que ilustren en forma separada los puntos más importantes para concluir.

En la sección 1 se muestra una evaluación del trabajo realizado, en cuanto a las limitaciones detectadas y los logros obtenidos.

En la sección 2 se especifican las contribuciones hechas por este trabajo. Finalmente en la sección 3 son presentadas algunas elementos pendientes que pueden ser extensiones al trabajo realizado.

1. Evaluación

1.1. Limitaciones

El framework implementado permite manejar una sola promoción a la vez sobre un objeto. Si bien es cierto, típicamente los modelos de negocio utilizados en los descuentos son excluyentes (una sola promoción), no está de más pensar en que hubiera sido deseable alcanzar a revisar los temas relativos a la composición de aspectos que subyace a esta problemática.

Este problema de la composición de contextos pudo ser resuelto utilizando un *rule-based systems*, similar al empleado por Reflex para componer aspectos. Esto implica la declaración de reglas lógicas que indican como se componen los contextos, definiendo una jerarquía y un orden de dependencia de los mismos.

Se podrían haber realizado más aportes de optimización al framework y a Reflex. Durante el desarrollo se encontraron diversas dificultades, que por inexperiencia o por tiempo se mantuvieron hasta el final o simplemente no fueron solucionadas.

1.2. Logros

Es importante destacar que el API del framework de context-aware aspects es simple y sencilla. Se incorpora naturalmente a Reflex y permite realizar todas las funciones para las cuales fue pensado. Además, si es necesario implementar otros contextos más complejos, se ha presentado claramente cuales son los puntos (métodos) donde hay que dedicarse con mayor detalle y que contienen toda la lógica del contexto.

Así mismo, si se va a incorporar WildCAT no es necesario complicar el código de los contextos, ya que toda la lógica de los sensores queda completamente encapsulada dentro de ellos, y éstos dejan visibles los métodos que permiten acceder a los datos que reifican el contexto.

El hecho de lograr que el framework maneje contextos en forma genérica y que permita adaptar aplicaciones según contextos basados en aspectos son logros muy importante; independientemente que sólo haya sido mostrado el caso del contexto de promoción, el ejemplo es claro en separar sus propias funcionalidades, de las que corresponden a los contextos basados en aspectos y de las proveídas por el framework.

El ejemplo utilizado a lo largo de toda la tesis resultó muy interesante. Si bien es cierto está hecho a una escala manejable este mismo problema se ve en la vida cotidiana de las casas comerciales. En la industria es un problema real el manejo de las promociones. Si las aplicaciones de venta de las casas comerciales se realizaran utilizando este tipos de tecnologías, el *time to market* de las promociones se disminuiría notablemente. Esto resulta muy importante ya que las grandes tiendas ya no se diferencian por productos, sino que más bien por promociones. Si una empresa es más rápida en presentar sus nuevas promociones, tiene una ventaja comparativa clara sobre su competencia.

2. Contribución

Si bien es cierto Reflex fue utilizado como base para construir el framework, de todas maneras fue necesario adentrarse en su mundo. Nunca dejó de sorprender la cantidad de funcionalidades y capacidades que ofrece Reflex. La implementación del framework provocó que surgieran aportes hacia Reflex, incorporando sobre todo nuevos métodos que facilitaran su uso. De hecho el framework de *context-aware aspects* y de anotaciones (JAnnote) son un aporte directo a la comunidad Reflex.

Un segundo elemento que ha significado un aporte es la construcción misma de un framework. Esto porque siempre un framework es la culminación de un largo trabajo de abstracción de un problema y sistematización de una solución. En este caso esta labor ha pasado por el descubrimiento del problema, su desarrollo y el trabajo de abstracción que se realizó en [12].

La aplicación de administración de promociones resultó ser un elemento muy útil para demostrar las capacidades del framework y por sobretodo para ejemplificar su utilización. Dadas la cantidad de casos de prueba y la posibilidad de utilizarla a gusto del usuario, fue posible ver en vivo cada uno de los comportamientos, incluso aquellos para los cuales no estaba preparado el framework.

3. Perspectivas

Indudablemente el paso siguiente a este trabajo pasa por agregar la capacidad de contar con varios contextos al mismo tiempo sobre la misma aplicación. Este tema está directamente relacionado con la composición de aspectos. Este problema tiene dos aristas. Por un lado está la composición de contextos (ya explicado en las limitaciones, sección 1.1) y por otro lado está la composición de los metaobjetos que utilizan los aspectos para modificar el comportamiento. Este desafío no es menor, y radica fundamentalmente en la implementación de *operaciones* entre metaobjetos, o sea, por ejemplo, si hay dos promociones activas y los correspondientes metaobjetos van a calcular el descuento, ¿Cómo se maneja este doble retorno?. Para el caso de ejemplo, cada uno de los metaobjetos aplicaría el descuento, pero ¿Cómo se unen?.

Otro tema que surge como trabajo futuro es el relacionado con contextos distribuidos. La lógica que hay detrás es distinta a la lógica de los context-aware aspects implementados en este framework. Una visión simplista podría dar ideas donde los *links* y los *hooksets* se encuentran distribuidos, pero el problema va más allá, ya que bajo este paradigma lo importante no es tanto que la aplicación esté distribuida, sino que lo realmente importante es que el contexto es quien está distribuido.

Finalmente, un tema relacionado que aparece luego de este trabajo es la implementación de la extensión sintáctica para *AspectJ* que fue utilizada como ejemplo en los primeros capítulos. Es posible proponer una implementación en *AspectJ* donde los contextos de promoción sean objetos

y los descuentos sean aspectos. Un problema de realizar esta implementación bajo *AspectJ* es la definición de como manejar los *snapshot*, dado que como en *AspectJ* no es posible analizar los *pointcuts* de los aspectos, todo aspecto de descuento tiene que poseer un advice a cargo del *snapshot*, que será utilizado incluso cuando no sea necesario, con la consiguiente sobrecarga para la aplicación. Esto junto a otros problemas limitan las posibilidades de extender *AspectJ* por si mismo, en forma reusable y modular. Por este motivo y dado que Reflex en la actualidad es un kernel multi-lenguaje para AOP, es posible a través de él, proveer a *AspectJ* de todas las extensiones sintácticas necesarias para *context-aware aspects*.

Bibliografía

[1] A.K.Dey and G.D.Abowd. Towards a better understanding of context and context awareness. In *Workshop on the What, Who, Where, When and How of puting Systems (CHI 2000)*, The Hague, The Netherlands, April 2004.

[2] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceeding of MPAC'05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Grenoble, France, November 2005.

[3] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.

[4] E. et al Gamma. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] G.Chen and D.Kotz. Context aggregation and dissemination in ubiquitous computing systems. In IEEE Computer Society, editor, *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and applications*, page 105, Washington, DC, USA, 2002.

[6] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.V.Lopes, J.M.Loingtier, and J.Irwin. Aspect-oriented programming. In M.Aksit and S.Matsuoka, editors, *Proceedings of the 11th European Conference on Onject-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskulä, Finland, June 1997. Springer-Verlag.

[7] J.Kephart. The vision of autonomic computing. In *Onward! Track at OOPSLA*, pages 13–36, Seattle, WA, USA, 2002.

[8] Philip K.McKinley, Seyed Masoud Sadjadi, Eric P.Kasten, and Betty H.C.Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.

[9] Pattie Maes. *Computational reflection*. Phd thesis, Artificial intelligence laboratory, Vrije Universiteit Brussels, Brussels, Belgium, 1987.

[10] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60, 2003.

[11] Éric Tanter, Noury Bouraqadi, and Jacques Noyé. Reflex - towards an open reflective extension of java. In *Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'01)*, LNCS 2192, pages 25–43, Kyoto, Japan, September 2001. Springer-Verlag, A. Yonezawa & S. Matsuoka (Eds.).

[12] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, LNCS 4089, Vienna, Austria, March 2006. Springer-Verlag.

[13] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In Robert GlŸck and Mike Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *lncs*, pages 173–188, Tallinn, Estonia, september/october 2005. sv.