Tesis de Licenciatura

“Acceso al contexto en un framework de AOP Semántico”

Alumnos : Marcelo Rodríguez – Juan Vaccari
Director : Nicolás Kicillof
Fecha : Noviembre 2006

Departamento de Computación
Facultad de Ciencias Exactas
Universidad de Buenos Aires


¿Ésta Es La Llave Para El Acceso Al Contexto En Un Framework De AOP Semántico?
Bárbaro.
Me Gustan Las Cosas Complicadas.

ABSTRACT

La evolución de los sistemas implementados mediante AOP (Aspect Oriented Programming) puede verse severamente obstaculizada por el acoplamiento existente entre el código base y los aspectos que se le aplican. Este acoplamiento es difícil de evitar en las plataformas orientadas a aspectos tradicionales (como AspectJ), que definen los aspectos en términos de la estructura del código base. Este problema ha sido atacado originalmente por la tesis de Setpoint. Se propuso un framework orientado a aspectos que permite describir los pointcuts en un nivel de abstracción que los separa del código base, haciendo uso de un modelo conceptual de vistas del sistema. Este trabajo continúa la línea de Setpoint, proponiendo un nuevo modelado de vistas, y mecanismos que permiten el acceso al contexto de ejecución desde los pointcuts y advices en términos de dichas vistas. De esta manera se logra una separación aún mayor entre aspectos y código base.

AGRADECIMIENTOS

El trabajo realizado en esta tesis es el resultado de un esfuerzo conjunto, con lo cual no podemos dejar de agradecer a todas las personas involucradas, quienes además no dejaron de brindarnos su apoyo:

A Nico, quien confió en nosotros desde un primer momento para llevar a cabo la Tesis, guiándonos, aconsejándonos y marcándonos el rumbo. Incluso durante sus viajes al exterior siempre estuvo en contacto con nosotros, haciendo que su ausencia prácticamente no se notara.

A Alan, que nos dio una mano enorme para darle forma a la tesis, colaborando en casi todas las dimensiones que la conforman, tanto en lo conceptual como en lo concerniente a la implementación de los cambios efectuados en Setpoint.

A Fer, que compartió con nosotros su conocimiento en el área de AOP, ayudándonos durante la investigación, sugiriéndonos papers y distintos tipos de bibliografía que nos fueron muy útiles para madurar los conceptos de esta disciplina.

A los demás integrantes de Dependex (Víctor, Diego G., Lucia, Diego P., Andran, Guido, Jorge) con quienes compartimos varios meses en el grupo de investigación de la salita 704, y de donde nos llevamos muy lindos recuerdos.

A nuestros padres, abuelos, hermanos y amigos, sin cuya constante compañía y contención durante toda la carrera probablemente nada de esto hubiera sido posible.

INDICE

1 Introducción

1.1 Problemática enfrentada

La programación orientada a aspectos (Aspect Oriented Programming, AOP) es una serie de tecnologías y conceptos que extienden las técnicas de programación tradicionales. Su objetivo es encapsular en un nuevo tipo de módulo, denominado *aspecto* las denominadas *incumbencias transversales* (*crosscutting concerns*) que generalmente se encuentran dispersas a través de varios módulos del sistema. Esto se realiza a través de dos nuevos conceptos: *advices* y *pointcuts*. Estos definen, respectivamente, nuevos comportamientos y donde deben aplicarse. El advice es un elemento parecido a un método que define el comportamiento que debe aplicarse, y está vinculado a un pointcut, el cual identifica un conjunto de puntos de ejecución del sistema, denominados joinpoints, en los cuales debe aplicarse el advice. A su vez se cuenta con el denominado *weaver*, que se encarga de componer en un sistema ejecutable la funcionalidad del sistema base con los comportamientos definidos mediante aspectos.

Sin embargo AOP como se lo entiende actualmente en su corriente más conocida, es decir, los frameworks del estilo definido por [ASPECTJ] -extensión de Java para programar aspectos-, tienen características que llevan a una situación paradójica. El uso de AOP tiene como fin último permitir una mejor separación de incumbencias para obtener así un diseño de mayor calidad que permita al sistema de software ser comprendido, construido, reutilizado y modificado durante toda su evolución. Sin embargo la forma en que AOP plantea la definición de los aspectos hace que estos estén altamente acoplados al código base del sistema, dificultando estas tareas, al punto de resultar incluso un trabajo más arduo, complejo y propenso a errores que en los sistemas convencionales.

Para atacar el problema existen varios enfoques que reemplazan la forma directa de vinculación de aspectos con el código base, desacoplándolos mediante la utilización de una capa intermedia ([SETPOINT], [CAZZOLA04], [KELLENS06], [NAGY05]). Esta capa que hace de nexo se trata en este caso de representaciones conceptuales de distintas vistas del sistema, cada vista reflejando un dominio relevante del problema. Este enfoque deja abierto el camino para poder definir pointcuts en una forma más abstracta, en términos relacionados a las vistas del sistema y no a su implementación. Esto trae dos consecuencias beneficiosas: por un lado las definiciones de los pointcuts son más naturales al hablar un lenguaje cercano al de los dominios del sistema; por otro lado se logra esa distancia necesaria entre aspectos y código base que facilita la evolución y reutilización de las distintas partes del sistema.

1.2 Objetivo de la tesis

En esta tesis se continúa profundizando sobre el modelo de representación y los mecanismos utilizados por [SETPOINT], lo que permite avanzar en el enfoque planteado por el framework. Más específicamente se determinan mecanismos que permitan acceder al contexto de ejecución siguiendo los lineamientos de Setpoint, es decir que se acceda de manera semántica, en términos de los conceptos planteados en las vistas del sistema.

Esta tesis hace dos aportes fundamentales en el camino delineado por la tesis original de Setpoint. Por un lado refina el modelo de representación de la capa intermedia definiendo

un esquema basado en eventos de alto nivel y por otro lado permite una mayor separación entre aspectos y código base gracias a la definición de los mecanismos de acceso al contexto mencionados.

A través de estos aportes se allana el camino para facilitar un trabajo futuro en el que se planea contar con una ejecución abstracta en dominios específicos de discurso, ampliando el concepto de vista. Esta idea es desarrollada en más detalle en las conclusiones del trabajo.

1.3 Organización del informe de tesis

El informe se encuentra organizado en varias secciones, las cuales se recomienda seguir en el orden establecido por este documento. A continuación enumeramos estas secciones, con una breve descripción del objetivo que se intenta alcanzar en cada una de ellas:

- **Motivación:** Introducción a la Programación Orientada a Aspectos, presentación del problema conocido como aspect fragility y de la propuesta original de Setpoint.
- **Acceso al contexto – Solución al advice fragility:** Desarrollo de los cambios efectuados sobre Setpoint para lograr el acceso al contexto semántico.
- **Casos de estudio**: Aplicación de aspectos con la nueva versión de Setpoint en los ejemplos de "cuentas bancarias" del "pipe and filter".
- **Conclusiones**: Conclusiones del trabajo realizado.
- **Anexo I - Preweaver 2.0**: Detalles de implementación de la nueva versión del preweaver.
- **Anexo 2 – Gramática LENDL 2.0**: Gramática de la nueva versión del lenguaje LENDL.

2 Motivación

2.1 ¿Qué es AOP?

En Ingeniería de Software, la técnica de Programación Orientada a Aspectos (Aspect Oriented Programming, AOP) procura ayudar a los programadores en la separación de incumbencias [DIJK76], o en la separación de un programa en distintas piezas que se superpongan lo menos posible en cuanto a funcionalidad. En particular, AOP se centra en la modularización y el encapsulamiento de las incumbencias transversales. Una incumbencia se llama transversal cuando afecta a distintas partes de un programa que no están relacionadas en el código.

Las incumbencias transversales se traducen en repetición de código en distintos módulos del sistema no relacionados, lo que tiene como consecuencia directa la aparición de código enredado (tangled code) y disperso (scattered code). Se llama código enredado cuando se realizan llamados a distintos servicios (logging, locking, presentación, transporte, autenticación, seguridad, etc), desde una misma operación que además cumple con su función específica. Se llama código disperso cuando el código de una incumbencia se encuentra distribuido por todo el programa.

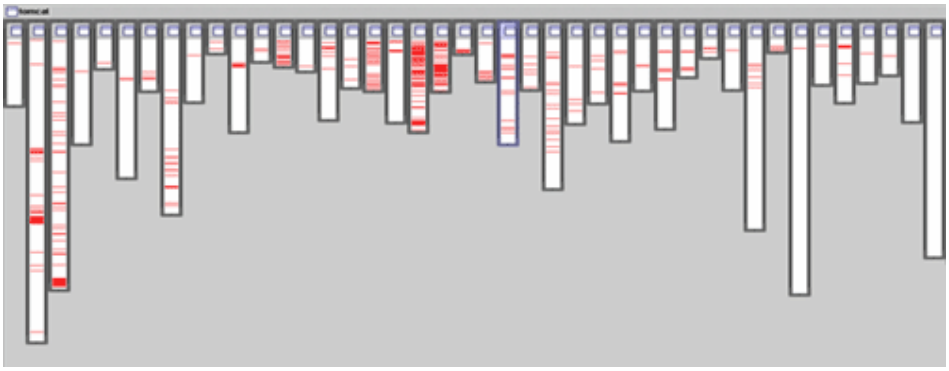


Figura 1 - Logging en el Apache Tomcat [KICZ01]

En el ejemplo de la figura, cada barra representa un módulo del sistema. Las líneas de código que invocan el servicio de *logging* están en rojo.

La fuente de problemas reside en que cada aplicación tiene una política sobre dónde se requiere cada uno de estos servicios; pero esa política no es explícita, está oculta como parte de la estructura del programa. Esto hace que sea difícil de entender, razonar sobre ella y mantenerla.

El logging es el ejemplo típico de una incumbencia transversal, porque una estrategia de logging necesariamente afecta cada parte a loguear del sistema. El logging por lo tanto atraviesa las distintas componentes del sistema que necesitan ser logueadas.

Típicamente una implementación de un lenguaje de AOP trata de encapsular estos tipos de incumbencias transversales introduciendo un nuevo concepto llamado aspecto [KICZ01]. Un aspecto puede alterar el comportamiento del código base (la parte no "aspectualizada" de un programa) mediante la aplicación de un advice (comportamiento adicional) sobre una cuantificación de joinpoints (puntos en la estructura o ejecución del programa) llamada pointcut (una descripción lógica de un conjunto de joinpoints).

En muchos lenguajes de AOP orientados a objetos, las ejecuciones de métodos y referencias a campos ejemplifican los llamados joinpoints. Un pointcut puede consistir, por ejemplo, en todas las referencias a un conjunto particular de campos.

2.1.1 Quantification y obliviousness

En [FILMAN00] se resaltan dos características importantes que deben estar presentes en la programación orientada a aspectos: *quantification* y *obliviousness*. Se entiende por quantification a la capacidad de identificar al conjunto de join points que deben responder a una determinada incumbencia transversal, sin necesidad de enumerarlos uno por uno. Por otro lado, el principio de obliviousness está relacionado con la habilidad de mantener al programador del código base prescindente de las incumbencias transversales que afectan a su código, librándolo de la preocupación de tener que contemplarlas cada vez que tenga que programar una funcionalidad específica de su aplicación. Según este trabajo, "AOP puede entenderse como el deseo de realizar sentencias cuantificadas sobre el comportamiento de los programas, y lograr que esas sentencias se cumplan en los programas escritos por programadores que las desconocen".

2.2 Ejemplo: Sistema Bancario

En esta sección presentamos el ejemplo que acompañará la lectura de esta tesis, y que servirá como referencia a la hora de describir los problemas que serán planteados, así como también los mecanismos elegidos para encararlos. Cabe aclarar que este ejemplo es una simplificación de una aplicación del mundo real; sin embargo, consideramos que es lo suficientemente ilustrativo para cumplir con los propósitos anteriormente citados.

Supongamos que contamos con una aplicación bancaria, la cual está conformada básicamente por *cuentas* y *operaciones* que se efectúan sobre las *cuentas*. A continuación se muestra el diagrama de clases que modela esta aplicación:

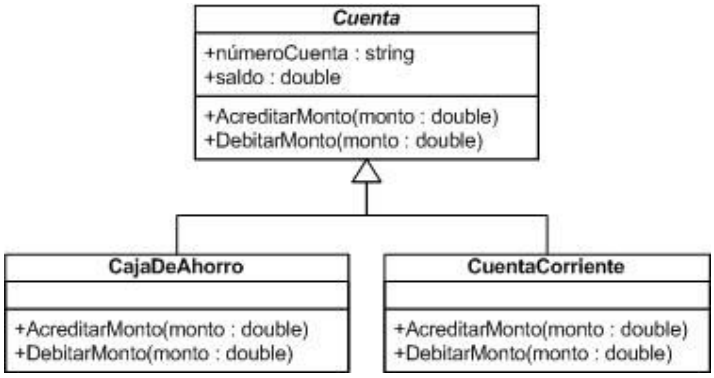


Figura 2 - Ejemplo de aplicación bancaria

En este modelo contamos con la clase abstracta *Cuenta*, la cual agrupa el comportamiento de todas las cuentas bancarias de esta aplicación. Los métodos *AcreditarMonto()* y *DebitarMonto()* representan las operaciones que incrementan y decrementan el saldo de la cuenta bancaria, respectivamente. Debido a que la forma en la que se realizan estas operaciones bancarias es inherente al tipo de la cuenta, las subclases CajaDeAhorro y CuentaCorriente cuentan con sus propias implementaciones para cada uno de esos métodos.

Supongamos que eventualmente surge un requerimiento que consiste en lo siguiente:

Luego de cada operación que modifique el saldo de una caja de ahorro, si el saldo de esa caja de ahorro es negativo, deberá registrarse el número de cuenta y el saldo resultante.

Este requerimiento no es otra cosa que una incumbencia transversal, ya que el mismo afecta tanto a la operación de débito como a la operación de crédito de la caja de ahorro, y a posibles futuras operaciones. Es por esto que resulta conveniente encapsular su implementación en un módulo separado del resto de las componentes de la aplicación, mediante el uso de *aspectos.*

Siguiendo la sintaxis de AspectJ, procedemos entonces a implementar el aspecto *LoggingAspect* que cumple con el requerimiento solicitado:

```
public aspect LoggingAspect {

      public pointcut ModificaciónDeSaldo(CajaDeAhorro ca):
            target(ca)
            && call(void *Monto(..));
```

```
      after(CajaDeAhorro ca): ModificaciónDeSaldo(ca) {
            if (ca.saldo < 0)
            {
                  Output.PrintLine("Cuenta: %s, Saldo: %f",
                                    ca.númeroCuenta, ca.saldo);
            }
      }
}
```

Este aspecto esta conformado por un pointcut llamado ModificaciónDeSaldo y un advice que ejecuta un código asociado luego de la ocurrencia de join points pertenecientes a este pointcut. El pointcut captura las llamadas a los métodos de la clase CajaDeAhorro cuyo nombre finaliza en "Monto". El advice chequea que el saldo de la caja de ahorro sea mayor a cero e imprime un mensaje con la información requerida.

2.3 Aspect fragility

A pesar de las ventajas prometidas por AOP, varias investigaciones en el impacto que el uso de esta disciplina tiene sobre las aplicaciones demostraron que, contrariamente a lo que se creía inicialmente, las tecnologías AOP del estilo AspectJ no facilitan la evolución de estas aplicaciones y muchas veces inclusive la dificultan aún más. Este hecho es particularmente nocivo si se tiene en cuenta que la evolución de los sistemas implica la mayor parte del tiempo de desarrollo de una aplicación [PARADOX]. Tampoco permiten la reusabilidad de los aspectos.

A continuación describiremos las razones por las que ocurre esta situación, lo que se denomina el pointcut fragility problem en [FRAGILITY], y que motivó en gran medida la propuesta original de Setpoint como alternativa a las soluciones AOP encontradas hasta ese momento. Una vez expuesto este problema, se explicará la forma en que Setpoint consigue resolverlo. Y a continuación se mostrará otro problema, el advice fragility problem, que agrupado con el anterior definen el aspect fragility problem, problema que debería ser resuelto por cualquier propuesta AOP para permitir el reuso de aspectos y una evolución de sistemas sin inconvenientes. A estas características positivas las denominamos reusabilidad y modificabilidad respectivamente.

2.3.1 Pointcut fragility

Para describir el pointcut fragility problem nos guiaremos a continuación en la exposición original del problema en [FRAGILITY]. Utilizaremos como ejemplo a AspectJ, pero los comentarios, observaciones y críticas realizadas son aplicables a lenguajes similares a AspectJ.

Los pointcuts son definidos en AspectJ mediante una serie de pointcut designators, los cuales permiten predicar sobre el código base, por ejemplo refiriéndose a llamadas a métodos o a acceso a propiedades de un objeto. Estos elementos atómicos pueden ser combinados mediante operadores lógicos para obtener pointcuts más expresivos.

Se produce por este motivo un problema de alto acoplamiento, debido a que en estos pointcuts es necesario hacer explícitos nombres de elementos pertenecientes al código base. Esto dificulta notablemente la reusabilidad de aspectos.

Como forma para reducir el acoplamiento, AspectJ propone el uso de wild-cards. Esto lleva el problema a un nuevo nivel. Por un lado sigue existiendo un factor sintáctico que acopla los pointcuts al código base debido a que finalmente se trata de expresiones regulares definidas a partir de cadenas de caracteres que se pueden encontrar en el programa. Por otro lado, lo que no queda explícitamente nombrado, por ejemplo con el uso del asterisco, implica que el código base debió ser desarrollado teniendo en cuenta convenciones de código. Esto trae varios inconvenientes, descritos en [PARADOX]:

1. Puede existir la necesidad de seguir varias convenciones de código simultáneamente, lo que puede llevar a conflictos.
2. Las convenciones no son chequeadas por el compilador, por lo que no son garantizadas. Se debe por tanto tener mucho cuidado al definir un pointcut para evitar atrapar joinpoints que no deberían pertenecer a este, o no atrapar otros joinpoints que sí deberían pertenecer.
3. No siempre se puede cumplir con las convenciones, por diversos motivos: presión por deadlines, falta de documentación, complejidad de la aplicación, etc.

Y existe aún otro problema, que tiene que ver con la fragilidad de los pointcuts en el siguiente sentido: cambios en el código base pueden afectar, de manera difícil de detectar, los conjuntos de join points que atrapan los pointcuts definidos en los aspectos. Por ejemplo un programador podría realizar una refactorización sobre el código. Por lo general los entornos de desarrollo modernos se encargan de chequear que esta refactorización se realice sobre las referencias involucradas, pero no tienen soporte para manejar el impacto que tienen estos cambios en los aspectos. En general existen varios cambios triviales no locales que podrían afectar la semántica de los pointcuts en cuanto a los join points atrapados:

- **Renombrado:** Renombrar clases, métodos o campos influencia la semántica de matcheo de *call*, *execution*, *get*/*set* y otros pointcuts. Los wild-cards proveen una protección parcial contra estos efectos indeseables.

- **Mover métodos/clases:** Los pointcuts pueden seleccionar join points por su posición léxica mediante *within* o *withincode*. Mover clases obviamente cambia la semántica de matcheo para tales pointcuts.

- **Agregar/Borrar métodos/campos/clases:** También se modifica la semántica si se agregan o remueven elementos del programa. Elementos nuevos pueden (y a veces deberían) ser matcheados por pointcuts disponibles, pero en general al escribir los pointcuts no se puede prever los futuros agregados. Eliminar elementos del programa naturalmente se traduce en join points "perdidos".

Siguiendo con el ejemplo anteriormente expuesto, supongamos que, como consecuencia de una refactorización del código de nuestra aplicación bancaria, los métodos `AcreditarMonto()` y `DebitarMonto()` son renombrados como `Acreditar()` y `Debitar()` respectivamente. Nótese cómo este pequeño cambio sintáctico en el código de la aplicación tiene un considerable impacto sobre la semántica del pointcut `ModificaciónDeSaldo`, ya que ahora ninguno de los join points que modifican el saldo de la caja de ahorro matchearán con dicho pointcut:

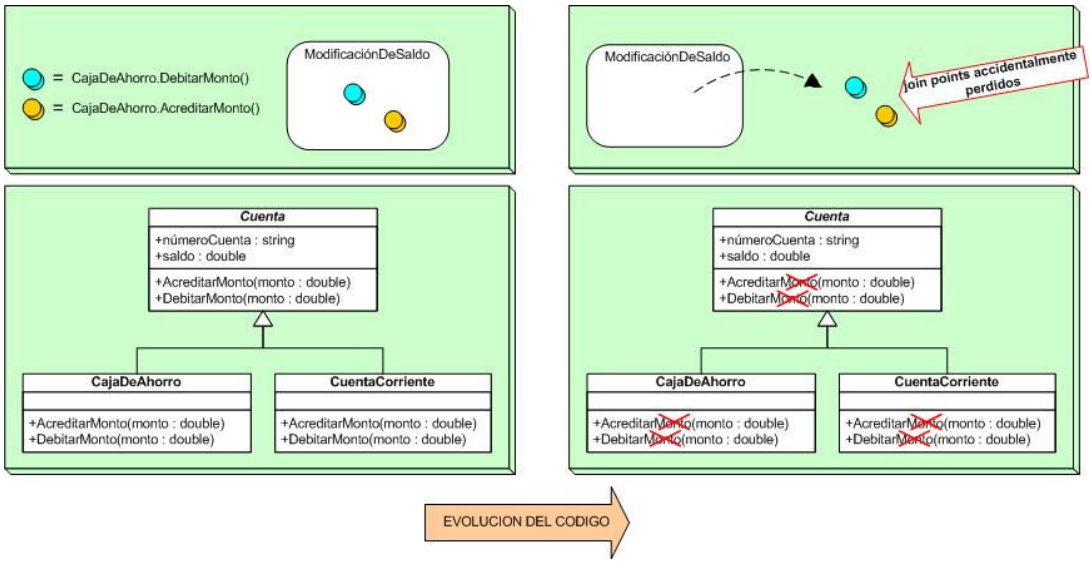


Figura 3 – Ejemplo de pointcut fragility en la aplicación bancaria

Para contrarrestar este problema, será necesario revisar la definición del pointcut `ModificaciónDeSaldo` de forma de poder "atrapar" nuevamente a los join points que se perdieron:

```
public pointcut ModificaciónDeSaldo(CajaDeAhorro ca):
       target(ca)
       && call(void *Monto(..));
       && call(void Acreditar(..)
       && call(void Debitar(..));
```

Claramente esta metodología de trabajo en cuanto a la definición de pointcuts es muy propensa a errores difícilmente detectables y corregibles.

2.3.2 Setpoint original

La problemática descrita motivó el desarrollo de la tesis original de Setpoint, cuyos fundamentos se explican a continuación.

El problema surge básicamente, como se mencionó, en el hecho de que en la mayoría de los enfoques de AOP los pointcuts se definen en función del código base, de una manera absolutamente atada a los aspectos sintácticos de este código. Se trata de un acoplamiento que afecta la evolución del sistema, a la vez que impide la reutilización del código de aspectos.

La solución planteada por Setpoint consistió en utilizar un modelo conceptual del sistema, que permitiría hacer las veces de capa intermedia entre el código base y el código de aspectos. De esta manera el código base se vincula con conceptos que describen el sistema en términos de diferentes **vistas**, como podría ser la vista de arquitectura o de un dominio particular del negocio. Estas vistas reflejan así la descomposición del sistema en distintas dimensiones del problema, explicitando la semántica asociada al sistema. Los pointcuts se describen en términos de estos conceptos, logrando dos objetivos: desacoplar los aspectos del código base y a la vez permitir predicar sobre el sistema en términos más abstractos y

por tanto más adecuados a la hora de definir el modelado de dimensiones como las mencionadas. Se trata entonces de **pointcuts semánticos**, denominados *setpoints*.

Estas ideas fueron llevadas a la práctica definiendo en primer lugar el lenguaje en el que se expresarían estas dimensiones del problema. En vez de optar por lenguajes específicos a dominios del problema, se utilizó un lenguaje general que fuera lo suficientemente amplio para permitir expresar las dimensiones que pudieran surgir. El lenguaje elegido fue OWL[OWL], un lenguaje proveniente de otra área de las ciencias de la computación, la web semántica, que permite definir ontologías conceptuales, así como proveer información sobre instancias de estos conceptos, definiendo una base de conocimiento que abarca las diferentes vistas del sistema y la relación del código con ellas. Este lenguaje esta basado en Description Logics[DL], y permite utilizar sobre él motores de inferencia y realizar consultas sobre la base de conocimiento.

Por otro lado fue necesario vincular de alguna manera los modelos definidos con OWL al código base. Para ello se utilizaron anotaciones en el código base, que vinculaban elementos del programa con conceptos definidos en las vistas del sistema.

Las anotaciones eran posteriormente traducidas de manera automática en términos de OWL como relaciones entre elementos del programa y conceptos definidos en las vistas. Esto permitía incorporar este conocimiento y predicar sobre él en un lenguaje de consulta especialmente diseñado para OWL. De esta forma, los pointcuts se definían como consultas a esta base de conocimiento, sobre las cuales posteriormente se determinaba la pertenencia o no de un joinpoint a un pointcut semántico, o setpoint.

Con el agregado de un lenguaje para definir los pointcuts y los advices en forma declarativa (llamado LENDL) quedaba entonces definido el framework de aspectos con un enfoque semántico que permitía atacar el pointcut fragility problem.

El framework de Setpoint fue desarrollado en .NET. Se decidió utilizar esta plataforma básicamente por dos razones fundamentales: por un lado, el soporte nativo de metadatos permite expresar las mencionadas anotaciones semánticas mediante el uso de atributos y, por el otro, la potencial llegada a una comunidad de usuarios masiva permitirá recibir feedback y validar la factibilidad de aplicación de pointcuts semánticos.

Veamos ahora cómo la propuesta de Setpoint resuelve el pointcut fragility problem en nuestro ejemplo de la cuenta bancaria. Se debe seguir una serie de pasos:

1. Definir el modelo conceptual de la aplicación bancaria mediante RDF/OWL. Para ello, debemos identificar los conceptos que intervienen en el dominio del problema a modelar, así como también la relación entre ellos:

   - Existen `[cuentas]`
   - Las `[cuentas]` tienen un `[saldo]`
   - La `[cajaDeAhorro]` es una `[cuenta]`
   - Existen `[operaciones]`
   - `[modificaciónDeSaldo]` es una `[operación]` que modifica el `[saldo]`

2. Anotar los elementos del programa con los conceptos anteriormente definidos:

```
[ProgramAnnotation("semántica://cuentasBancarias#cajaDeAhorro")]
public class CajaDeAhorro
{
…
}

[ProgramAnnotation("semántica://cuentasBancarias#modificaciónDeSaldo")]
public void AcreditarSaldo(double monto)
{
…
}
```

Las anotaciones se realizan mediante el uso de atributos de .NET. En este caso, la clase `CajaDeAhorro` se encuentra anotada con el concepto *semántica://cuentasBancarias#cajaDeAhorro*, mientras que el método `AcreditarSaldo` es asociado a *semántica://cuentasBancarias#modificaciónDeSaldo*

3. Definir el pointcut. El siguiente es el código LENDL utilizado para definir el pointcut `ModificaciónDeSaldo`:

```
pointcut ModificaciónDeSaldo {
   receiver is [semántica://cuentasBancarias#cajaDeAhorro];
   message is [semántica://cuentasBancarias#modificaciónDeSaldo];

}
```

Esta expresión de LENDL declara al pointcut `ModificaciónDeSaldo` como el conjunto de join points tales que el objeto que envía el mensaje esté relacionado con el concepto *semántica://cuentasBancarias#cajaDeAhorro*, y que el mensaje enviado esté relacionado con el concepto *semántica://cuentasBancarias#modificaciónDeSaldo*

4. Escribir el aspecto de *logging*. Para ello, utilizamos nuevamente el lenguaje LENDL

```
aspect LoggingAspect{
   event logAccount;
}

advice LogEvents : LoggingAspect{
   trigger logAccount after {ModificaciónDeSaldo};
}
```

Aquí se declaran los *eventos* de los que dispone el aspecto, que representan las incumbencias transversales. Por ejemplo, el evento `logAccount` es la incumbencia transversal de *logging*. Los eventos son programados en un módulo independiente.

Los *triggers* son elementos que asocian los eventos de un aspecto con los pointcuts que los disparan, y especifican además si los eventos deben ejecutarse antes o después del correspondiente join point.

Puede verse ahora que luego de renombrar los métodos que modifican el saldo (*AcreditarMonto()* y *DebitarMonto()*) la definición del pointcut no se ve afectada, y por lo tanto no hay necesidad de efectuar cambios del lado de los aspectos.

2.3.3 Advice fragility

El desarrollo conceptual de Setpoint permitió vislumbrar nuevos inconvenientes en cuanto a la reusabilidad y modificabilidad de las aplicaciones aspectualizadas.

En la sección anterior se mostró que cuando el código base evoluciona muchos de esos cambios llevan a los pointcuts a no capturar el mismo conjunto de joinpoints original, cambiando por tanto la semántica implícita.

¿Pero qué ocurre con los advices al evolucionar el código base? Lo que se puede observar es que la decisión de mantener al código base oblivious con respecto a los aspectos lleva en última instancia a que también los advices queden excesivamente acoplados al mismo código base. La aparente ventaja que conlleva poder agregar aspectos sobre código base sin modificar este último tiene consecuencias que se aprecian al mirar el sistema resultante desde una perspectiva más global. Es desde esta perspectiva que se hace necesario considerar las consecuencias del obliviousness sobre la forma en que quedan expresados los aspectos y su evolución al evolucionar el sistema completo.

Los advices vinculan los pointcuts con código que debe ejecutar comportamiento relacionado a incumbencias transversales. Sin embargo, este código, a pesar de tratarse de incumbencias transversales, depende del contexto de ejecución para realizar sus tareas.
De hecho existen ataduras relacionadas con el contrato implícito utilizado por los advices para poder utilizar este mismo contexto. La evolución del código base puede romper este contrato.

Existen varias formas en las que, al acceder el contexto, un advice se percibe como frágil. Una primera situación que se presenta tiene que ver con el estado que se espera que tengan los objetos que el advice recibe del contexto. El advice podría por ejemplo requerir que los objetos estuvieran inicializados, o que se cumplieran cierta relación entre ellos, como por ejemplo que un objeto pasado como parámetro fuera menor a otro. Esta situación, que también se presenta en la programación habitual, es más peligrosa en este contexto, debido a la obliviousness por parte del programador de código base, que desconoce las consecuencias no locales de cambiar la semántica en su propio código.

Otra situación se presenta en el uso de reflection, el cual es fundamental en muchos frameworks de AOP. Si por ejemplo en el código base se cambia el nombre de un método o se lo borra, se podría estar llamando a un método inexistente en el advice. Una tercera situación tiene que ver con la aplicación de casting sobre objetos. El uso de este mecanismo sintáctico de especialización de comportamiento en los advices presenta problemas similares al de la situación anterior: un cambio en la estructura jerárquica de clases en el código base podría fácilmente dejar a dichos advices en un estado de inconsistencia (esta vez arrojando un error de casting).

Remontémonos nuevamente al ejemplo de las cuentas bancarias, ahora aspectualizado sobre la primera versión de Setpoint. A continuación presentamos el extracto de código correspondiente a la incumbencia transversal de *logging*:

```
public class LoggingAspect : IAspect
{
   public void logAccount (IJoinPoint jp)
```

```
    {
       CajaDeAhorro ca = (CajaDeAhorro)
                         receiver.asMethodInvokeReference();

       if (ca.saldo < 0)
       {
          CajaDeAhorro ca = (CajaDeAhorro)
                            receiver.asMethodInvokeReference();
          System.Console.Writeline("Cuenta: %s, Saldo: %f",
                                   ca.númeroCuenta, ca.saldo);
       }
    }
}
```

En Setpoint los aspectos deben ser programados en un módulo independiente y esperan como argumento los denominados *join points de bajo nivel*, que son elementos predefinidos por el framework que reifican los eventos en tiempo de ejecución, tales como las llamadas a métodos, constructores, getters, setters, etc. Estos contienen el *emisor* del mensaje (sender), *receptor* del mensaje (receiver), los argumentos y el *selector* del mensaje. En este caso, el objeto `CajaDeAhorro` se obtiene inspeccionando al miembro `receiver` del join point de bajo nivel.

Para hacer más gráfico el problema de los advices frágiles, mostraremos ahora dos escenarios en los cuales se ven reflejados los inconvenientes de modificabilidad y reusabilidad en la aplicación bancaria, analizando el impacto que tienen los mismos sobre los advices.

2.3.3.1 Escenario 1: impacto en la modificabilidad

Supongamos que la aplicación bancaria es modificada de forma tal que ahora el saldo de las cuentas es expresado como la diferencia entre los montos acreditados y los montos debitados:

$$saldo = total\ \ acreditado - total\ \ debitado$$

Para ello debemos modificar la clase `Cuenta` agregando los atributos `totalAcreditado` y `totalDebitado` que acumularán los montos acreditados y debitados de la cuenta:

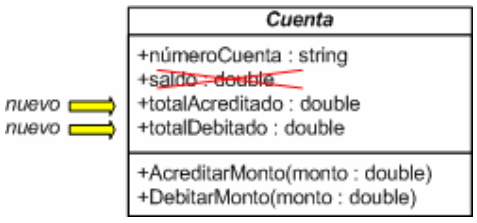


Figura 4 – Nueva representación del saldo

Este cambio en la aplicación impacta directamente sobre el código de nuestra incumbencia transversal de *logging*, en donde deben efectuarse los cambios necesarios para poder contemplar la nueva forma de obtener el saldo de las cuentas:

```
public class LoggingAspect : IAspect
{
   public void logAccount (IJoinPoint jp)
   {
```

```
        CajaDeAhorro ca = (CajaDeAhorro) jp.receiver;

        if (ca.saldo < 0)
        if (ca.totalAcreditado < ca.totalDebitado)
        {
           System.Console.Writeline("Cuenta: %s, Saldo: %f",
                                    ca.númeroCuenta, ca.saldo);
           System.Console.Writeline("Cuenta: %s, Saldo: %f",
                                ca.númeroCuenta,
                                ca.totalAcreditado - ca.totalDebitado);
        }
     }
  }
```

2.3.3.2 Escenario 2: impacto en la reusabilidad

Veamos ahora un caso en el que la aplicación bancaria está compuesta por tres sub-módulos independientes, cada uno de los cuales efectúa operaciones de modificación de saldo sobre sus propias implementaciones de las cuentas. Debido a que la forma de representar las cuentas y sus miembros (número de cuenta, saldo, etc.) es diferente en cada uno de estos sub-módulos, será necesario construir el aspecto de *logging* para cada uno de ellos.

Este impacto –evidentemente negativo– en la reusabilidad atrae las siguientes consecuencias en el mantenimiento y evolución de la aplicación:

- La incorporación de un *nuevo submódulo* que implemente las cuentas requerirá la creación de un advice específico para dicho submódulo.
- La incorporación de un *nuevo aspecto* requerirá la creación de un advice específico para cada uno de los submódulos
- En caso que se desee reemplazar, por ejemplo, el mecanismo utilizado para efectuar *logging*, será necesario modificar todos los advices de los distintos sub-módulos.

Es correcto apreciar además que este problema rompe en cierto grado con el propósito inicial de los aspectos: encapsular las incumbencias transversales en un único lugar. Es decir, si bien las incumbencias se encuentran separadas del código base, ahora están *dispersas* en distintos componentes.

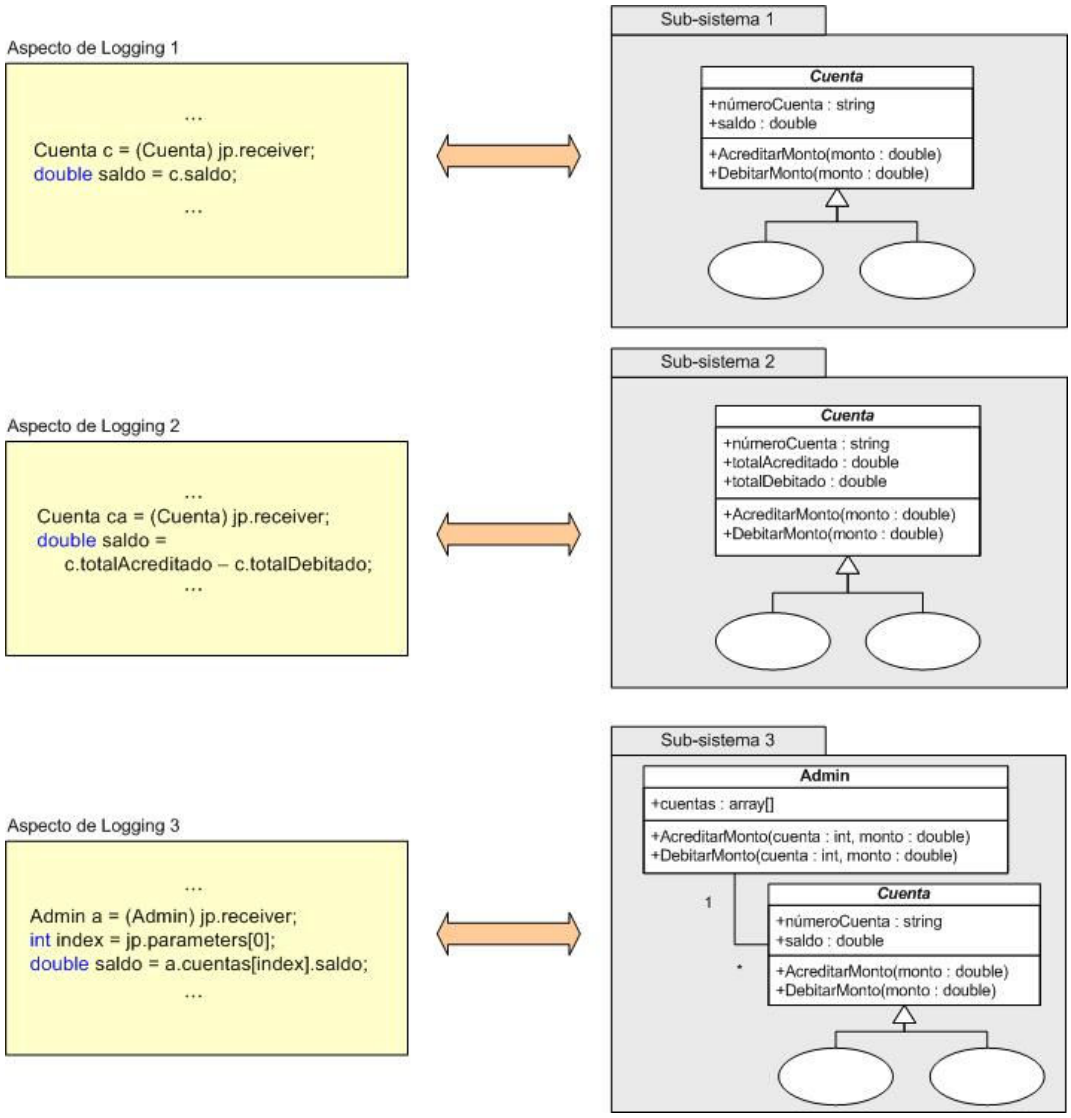


Figura 5 - Cada implementación de las cuentas se asocia a un aspecto de logging específico

3 Acceso al contexto – Solución al advice fragility

3.1 Introducción

Como se mencionó anteriormente, la propuesta original de Setpoint incorpora un modelo conceptual del dominio como capa intermedia entre los aspectos y el código base para solucionar el problema del pointcut fragility. Sin embargo, el uso de OWL para la representación del dominio manifestaba ciertas limitaciones, como por ejemplo la imposibilidad de hacer uso de dicha capa intermedia para acceder a información del contexto de ejecución. Esto en particular daba lugar entonces a la eventual problemática del advice fragility, dado que no se contaba con un mecanismo que permitiera, desde los advices, abstraerse del código base.

Es por esto que surge la necesidad de replantear la representación conceptual del dominio así como también de incorporar los mecanismos necesarios de forma de encontrar una

solución integral; es decir, una solución que abarque tanto al problema del pointcut fragility como al del advice fragility.

En esta sección explicamos en detalle la solución y los principales elementos que componen una nueva versión de Setpoint, desarrollada para permitir el acceso al contexto en forma semántica. En primer lugar se detallan los lineamientos generales necesarios que una solución al problema debería considerar. Luego se presenta la solución en sí. Se describe el nuevo modelado del dominio utilizando interfaces en vez de OWL, a la vez que se justifican los motivos para realizar el cambio de lenguaje de representación. Se presenta la arquitectura de la implementación, en la que se ven los componentes de la solución y su interrelación. Se presenta la nueva versión de LENDL, el lenguaje en el que se especifican los pointcuts, aspectos y advices. Se explica la utilización de mappers, mecanismo por el cual se vincula el código base a las vistas del sistema, así como las anotaciones en el código base, a partir de las cuales se generan los mappers de manera automática. Finalmente se analiza las propiedades de quantification y obliviousness con respecto a la solución presentada, se menciona el trabajo relacionado y se realizan algunas conclusiones finales. Como último punto se presenta el trabajo futuro.

3.2 Lineamientos generales para alcanzar una solución

Así como en el Setpoint original se realizó un salto semántico para resolver el pointcut fragility problem, lo que se pretende en esta tesis es avanzar por ese mismo camino semántico para resolver el advice fragility problem. Este avance consiste en definir los mecanismos por los cuales se puede acceder de manera semántica al contexto, lo cual requerirá la existencia de un contrato semántico explícito entre las partes involucradas (código base y aspectos).

Analicemos entonces cuales son los lineamientos necesarios para cumplir con este objetivo:

- *Capacidad para expresar información estructural en el lenguaje de la capa intermedia*. Interesa expresar la estructura de elementos de las vistas, los cuales se modifican durante la ejecución del sistema. Se busca con esto poder predicar desde diferentes vistas sobre elementos del contexto de ejecución. La información estructural brindará a su vez la explicitación necesaria del contrato semántico.

- *Mapeo de elementos del código base a conceptos en las vistas*. Para poder obtener desde el código base la información necesaria para componer los conceptos en las vistas se necesita algún mecanismo que el programador pueda explicitar. En este mismo sentido vale aclarar que alguna información existente en el código base no es factible de ser obtenida sin realizar un procesamiento previo, el cual podría implicar cierta complejidad. Esta complejidad nos permite pensar que lo más adecuado sería utilizar el mismo lenguaje de programación con el que se produce el código base. Esto por un lado facilita la compatibilidad a la hora de conseguir datos que se encuentran expresados en el mismo lenguaje, no requiere aprendizaje extra por parte del programador y aprovecha toda la complejidad inherente al lenguaje utilizado, que en este caso es C# (aunque podría ser cualquier otro lenguaje soportado por .NET).

- *Capacidad de referencia y acceso desde pointcuts y advices a elementos de representación intermedia*. Una vez que los conceptos en las vistas tengan una existencia propia, mediante las condiciones explicadas más arriba, es necesario

poder acceder a la información que estos contienen de manera de poder utilizarla en pointcuts y advices.

3.3 Modelado del dominio mediante interfaces

3.3.1 Esquema de acciones, roles, entidades y propiedades

El modelado de los distintos dominios involucrados en el sistema es llevado a cabo mediante orientación a objetos, más específicamente interfaces. Aunque bien podría utilizarse libremente la expresividad provista por el paradigma de objetos para modelar los dominios, se establecieron ciertas reglas básicas a seguir de forma que el modelo pueda interactuar tanto con las vistas como con el código. Para ello se definió un meta-modelo en el que se prescribe la forma en que deben estructurarse los elementos del dominio dentro del modelo.

Como puede verse en el diagrama UML de la figura 6, este meta-modelo cuenta con *conceptos*, los cuales se distinguen principalmente en dos clases: *acciones* y *entidades*. Las acciones serían el equivalente a un join point en el nivel conceptual. Estas representan eventos que se producen en el sistema y que son relevantes desde alguna perspectiva. Las entidades, por su parte, representan elementos estructurales pertenecientes a alguna perspectiva.

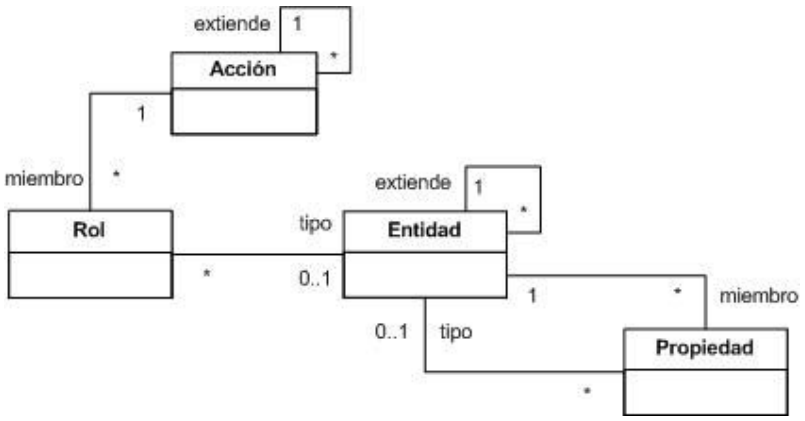


Figura 7 - Meta-modelo del concepto de acciones y entidades

Las acciones contienen *roles*, que pueden ser entidades cumpliendo determinadas funciones en el contexto de una determinada acción. Los roles también pueden ser cumplidos por tipos de datos primitivos. Las entidades poseen *propiedades*, en el sentido tradicional de orientación a objetos. Estas propiedades pueden también ser a su vez entidades, o bien estar representadas por tipos primitivos. El meta-modelo también permite establecer herencia entre acciones y herencia entre entidades, lo cual proporciona un mayor grado de expresividad al modelo.

Un dominio del sistema es modelado entonces mediante un conjunto de interfaces, las cuales representan acciones y entidades. Estas interfaces son interfaces comunes de C#, conteniendo getters y setters para las propiedades y roles, mecanismo por el que se accede a la estructura de los objetos del dominio.

Retomando nuestro ejemplo de las cuentas bancarias, veamos cómo quedaría modelado el dominio del problema haciendo uso de las interfaces:

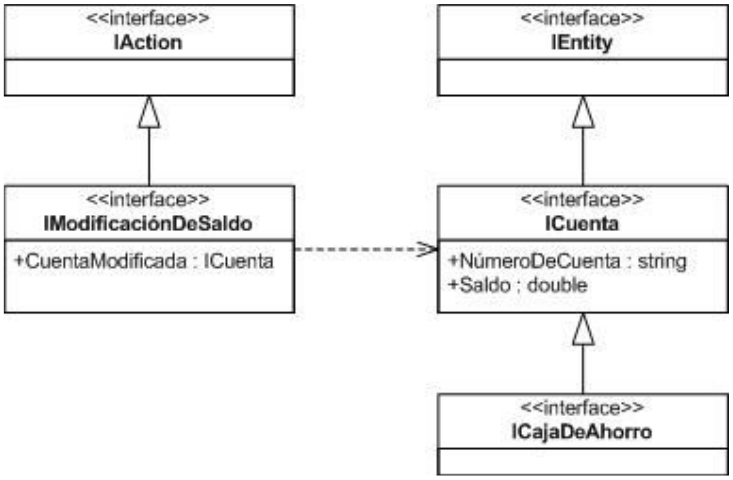


Figura 8 - Modelo de las cuentas bancarias usando interfaces

En este modelo, `IModificaciónDeSaldo` es la acción que representa las operaciones que modifican el saldo de una cuenta. Dicha acción posee el rol `CuentaModificada`, que corresponde a la entidad `ICuenta` y representa a las cuentas bancarias. La entidad `ICuenta` contiene las propiedades `NúmeroDeCuenta` (de tipo `string`) y `Saldo` (de tipo `double`). Finalmente, se establece que la entidad `ICajaDeAhorro` especializa a la entidad `ICuenta`, definiendo de esta forma que una caja de ahorro es un tipo particular de cuenta.

3.3.2 Justificación del cambio de OWL por interfaces

Dado que el modelado de la capa intermedia se realizaba originalmente en Setpoint mediante OWL comenzamos nuestra búsqueda de soluciones para los nuevos requerimientos utilizando este lenguaje.

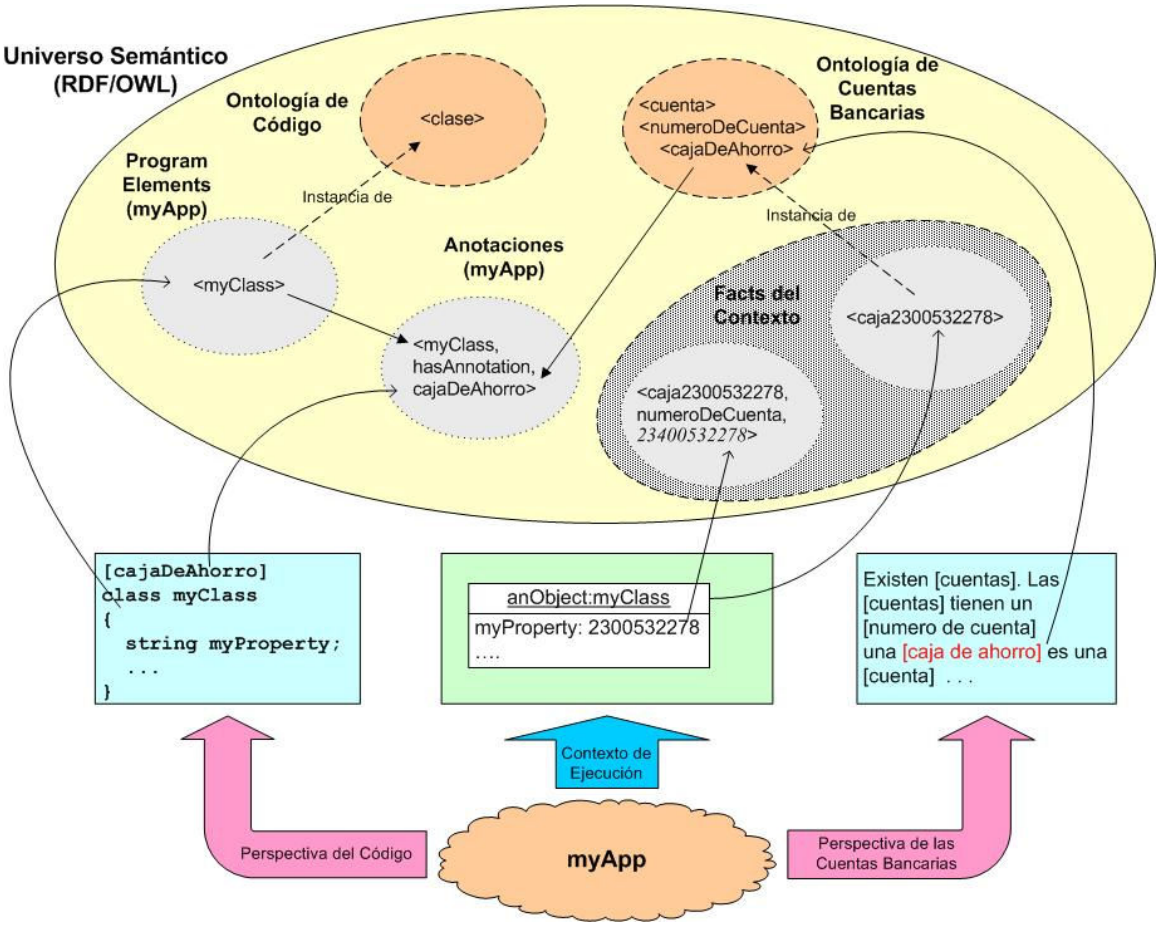
Como se mencionó anteriormente, OWL es un lenguaje que permite expresar ontologías. Una ontología en este contexto se entiende como la definición del significado de términos en un vocabulario y las relaciones entre estos términos. Toda esta información se expresa mediante axiomas. A su vez OWL permite expresar información sobre instancias del dominio descrito. Los enunciados que componen una ontología en OWL se expresan mediante RDF, por lo que la ontología esta conformada por un conjunto de triplas de la forma <sujeto, predicado, objeto>, a las que se denomina facts (hechos). Como resultado queda definida una teoría axiomática acerca de un dominio específico, sobre la que se puede realizar consultas e inferencias lógicas.

Las propuestas de soluciones orbitaban alrededor del siguiente planteo: nuestra intención era que al llamarse al weaver, el framework generase el contexto a nivel semántico agregando facts dentro de una *ontología de contexto*. Esta ontología, junto al resto de las ontologías involucradas, definiría un contexto sobre el que se podría inferir y hacer consultas. Esto permitiría acceder de manera homogénea y coherente con el uso de ontologías desde pointcuts y aspectos.

Como ejemplo, supongamos que parte del modelado del dominio de las cuentas bancarias consiste en especificar un recurso RDF para identificar las cajas de ahorro, y asignarle propiedades, como el número de la cuenta y el saldo. Luego, asumamos que en un determinado momento de la ejecución de la aplicación bancaria existirá, en el contexto de ejecución, una caja de ahorro con el número de cuenta 23400532278 y con un saldo de 1450 pesos. El framework de Setpoint debería agregar al universo semántico los siguientes facts:

<unaCajaDeAhorro, nroDeCuenta, 23400532278>
<unSaldo, valorDelSaldo, 1450>

Por otro lado, ya que las ontologías definen los tipos de los individuos y de las relaciones que se quieren acceder, no debería haber problema para el programador de aspectos en cuanto a la especificación de lo accesible. Gracias a las ontologías, conocería lo que va a recibir cuando lo pida. El chequeo de condiciones para verificar si un join point pertenece a un pointcut y las consultas de información sobre el contexto se podrían realizar mediante el lenguaje de consulta SeRQL.



Sin embargo la experiencia que tuvimos al utilizar OWL nos permitió sacar algunas conclusiones negativas con respecto a su aplicabilidad en el contexto de Setpoint, que nos llevaron a cambiarlo por un lenguaje de objetos para representar la capa intermedia. Describiremos estos problemas a continuación:

3.3.2.1 Problemas de expresividad:

Previamente describimos los lineamientos generales que deberíamos seguir para poder llevar a cabo el acceso al contexto de manera semántica y resolver así los inconvenientes presentados con el nombre de advice fragility problem. En particular la utilización de OWL como lenguaje de representación nos trajo inconvenientes para poder cumplir con el primero de dichos lineamientos, es decir, la capacidad para expresar información estructural en el lenguaje de la capa intermedia.

OWL tiene 3 versiones del lenguaje, OWL Lite, DL y Full, con crecientes niveles de expresividad. La mayor expresividad trae aparejada una mayor complejidad a la hora de realizar razonamientos y derivar conclusiones. De hecho la versión Full no tiene garantías de ser computable. Es decir que no se cuenta con software que soporte razonamientos sobre la totalidad de las capacidades de expresión de esta versión del lenguaje. Se trata de una restricción fuerte que nos limita a las versiones Lite y DL. La versión Lite provee básicamente una jerarquía de clasificación y permite restricciones simples. Por ejemplo, provee restricciones de cardinalidad pero mientras los valores de cardinalidad se mantengan entre 0 y 1. Debido a esta falta de expresividad nos concentramos en la versión DL.

Sin embargo OWL DL no resultó tampoco ser lo suficientemente expresivo para cumplir con nuestros objetivos. Necesitábamos instanciar el metamodelo mencionado en la sección anterior: era necesario poder expresar que existen lo que denominamos acciones y que éstas están relacionadas con entidades que cumplen roles en el contexto de estas acciones. Para ello era necesario definir en OWL ciertas relaciones que permitieran vincular las acciones con otros conceptos, que representaban entidades. Estas entidades se representan en OWL mediante clases que definen a su vez los elementos que componen su estructura. Para vincular las acciones con las entidades era necesario entonces definir relaciones entre clases pero tratándolas como instancias, de manera de poder expresar que cierta acción esta vinculada a ciertas entidades. Este tratamiento de las entidades como clases a la hora de definir su estructura (necesario para luego acceder a esta información de contexto) y como instancias a la hora de asociarlas vía una relación con acciones, es una capacidad existente en OWL, pero no en la versión DL. Para poder expresarlo debe utilizarse OWL Full, lo que nos llevaba nuevamente a los inconvenientes mencionados previamente.

3.3.2.2 Problemas de usabilidad:

Por otro lado la experiencia al intentar expresar los requerimientos en este lenguaje nos mostró que se encontraba a una distancia conceptual grande con respecto a los lenguajes que suelen encontrar los desarrolladores de software. Una de las diferencias fundamentales entre el enfoque de lenguajes tradicionales en la industria del software (nos referimos a lenguajes de programación dentro del paradigma de objetos y a otras tecnologías vinculadas, como XML o las bases de datos relacionales) y lenguajes relacionados con aspectos lógicos es que están basados en una hipótesis diferente sobre la realidad. El primer caso se basa en el **closed world assumption** y el segundo en el **open world assumption**. Básicamente el primero da por sentado que todo lo que no se sabe es falso porque considera que todo el conocimiento que existe está disponible, mientras que el segundo lo consideraría no computable o desconocido, ya que podría existir información de la que no se dispone actualmente.

Mostramos dos ejemplos para clarificar. Supongamos que tenemos expresada en un lenguaje la siguiente información:

Contamos con una relación *perteneceA* que asigna cuentas de un banco a clientes y con los siguientes datos:

cajaDeAhorro1 ***perteneceA*** *cliente1*

cajaDeAhorro1 ***perteneceA*** *cliente2*

cajaDeAhorro2 ***perteneceA*** *cliente3*

Si se consultara al sistema qué clientes no tienen asignada la cajaDeAhorro1, en un entorno basado en el close world asumption se daría como respuesta al cliente3 (se sabe que el cliente3 no está asociado a la cajaDeAhorro1), mientras que en un entorno basado en el open world asumption la respuesta seria que se desconoce la información (se desconoce si cliente3 está asociado a la cajaDeAhorro1).

Veamos otro ejemplo:

cajaDeAhorro1 ***perteneceA*** *cliente1*

Ahora agregamos una condición sobre la relacion *perteneceA*:

perteneceA ***tieneCardinalidad*** *1*

Agreguemos ahora a la anterior información el siguiente dato:

cajaDeAhorro1 ***perteneceA*** *cliente2*

En un lenguaje tradicional bajo el closed world assumption, el sistema arrojaría un error indicando que la cardinalidad no esta siendo respetada. Por otro lado, bajo un open world assumption, como en OWL, la presentación de este nuevo dato llevaría a generar otra información que indicara que cliente1 y cliente2 son la misma entidad, ya que esa interpretación permite hacer consistente el conjunto de datos presentes.

Estas diferencias en el enfoque de los lenguajes nos permiten pensar que la utilización de un framework combinando ambos enfoques podría implicar errores, una curva de aprendizaje muy lenta, y/o abandono de la tecnología.

Estos mencionados problemas de expresividad y de usabilidad sumados al hecho de no haber encontrado ejemplos que justificaran la necesidad de un motor de inferencias tan fuerte como el asociado a OWL hicieron poner en duda la continuidad de OWL como lenguaje de representación. Nos concentramos entonces en una representación de la capa intermedia mediante la utilización de un lenguaje orientado a objetos como el que se usa para desarrollar la aplicación base.

3.4 Arquitectura del nuevo Setpoint

Para llevar a cabo la solución al problema del advice fragility fue necesario rediseñar la versión original de Setpoint, lo cual implicó modificar algunos componentes y alterar el comportamiento del motor de Setpoint de forma de poder interactuar con la nueva

representación del dominio. El siguiente esquema presenta la arquitectura resultante (se resaltan los elementos incorporados en la nueva versión del framework), y los apartados sucesivos detallan la función de cada uno de los componentes y procesos.

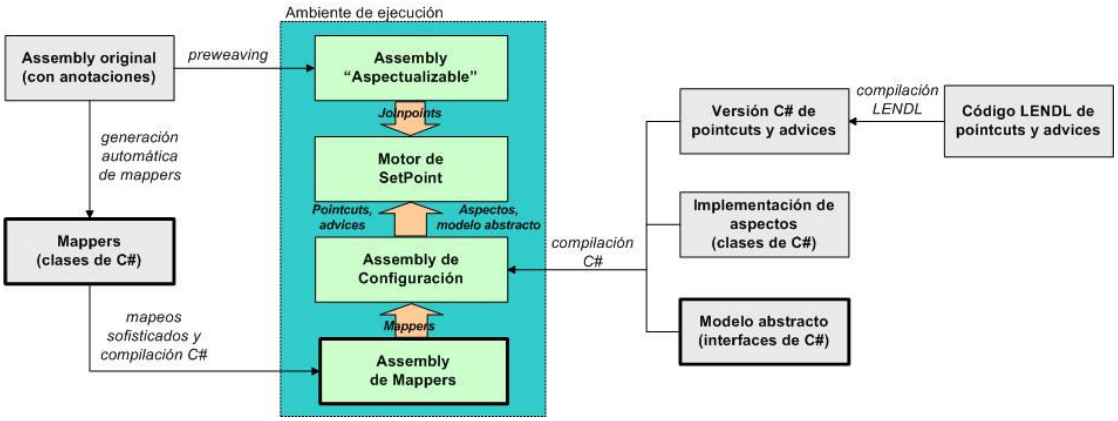


Figura 9 - Arquitectura del nuevo Setpoint

3.4.1 Preweaving

Este procedimiento tiene como objetivo preparar un assembly cualquiera para que pueda funcionar bajo el framework orientado a aspectos. Para ello, debe ejecutarse la tarea de *preweaving*, que consiste en instanciar en cada envío de mensaje el correspondiente join point y delegar su posterior ejecución al weaver.

El proceso de preweaving no presenta variantes respecto de la versión original, pero su implementación fue modificada para trabajar con Phoenix [PHOENIX], una herramienta de Microsoft orientada al análisis y generación de código[1]. La tecnología utilizada en la versión original para la instrumentación de assemblies [PERWAPI] pertenecía a terceras partes, y no era estable con las nuevas versiones de .NET. Phoenix provee mecanismos nativos para trabajar con código MSIL, y sus librerías eventualmente formarán parte de la plataforma de desarrollo de .NET. El apéndice del informe cuenta con una sección dedicada al uso esta nueva tecnología en el proceso de preweaving.

En la versión original de Setpoint existía una tarea adicional de procesamiento del assembly: la *semanticación*. El propósito de la semanticación consistía en agregar la perspectiva del código de la aplicación a aspectualizar dentro del universo semántico, repositorio del motor de inferencias de Setpoint. Para ello, se construía una representación RDF del assembly y se la incluía como recurso embebido. Debido a que en la nueva versión del framework cambió la forma de representar y de interactuar con el universo semántico, esta fase de procesamiento fue descartada.

3.4.2 Creación de mappers

La creación de mappers consta de tres etapas. En la primera etapa se utiliza una herramienta que genera automáticamente un archivo de código fuente, en un lenguaje compatible con .NET, con la definición de las clases que corresponden a los mappers. Para esto, se

[1] Esta herramienta aún se encuentra en una fase de desarrollo, pero lo suficientemente madura como para ser utilizada.

necesita como input el assembly original con las anotaciones efectuadas en el código base, y por medio de las anotaciones realizadas en el assembly original se determinan los mappers a definirse y las interfaces que éstos deben implementar.

La generación automática sólo resuelve mapeos directos entre el código base y el dominio semántico. Es por ello que en una segunda etapa se deben agregar manualmente aquellos mapeos sofisticados que no pudieron ser contemplados mediante las anotaciones. Finalmente, en la tercera etapa se compila el código resultante y se genera el assembly de los mappers.

3.4.3 Generación de configuración

El proceso de generación de configuración toma los pointcuts y advices definidos mediante LENDL, y junto con código correspondiente al modelo abstracto y el de la implementación de los aspectos construye el assembly de configuración, el cual es utilizado por el motor de Setpoint para decidir sobre la aplicación de los aspectos en los distintos join points.

Previamente a la generación del assembly de configuración, los pointcuts y advices deben ser traducidos mediante el compilador de LENDL a código compatible con .NET. El lenguaje LENDL fue extendido como resultado de algunos cambios que se hicieron en los pointcuts y advices para proveer acceso al contexto semántico. Además se robusteció la expresividad de los pointcuts con el agregado de expresiones lógicas. La nueva sintaxis es explicada en detalle en las siguientes secciones.

Como fue explicado en secciones anteriores, las interfaces son el nuevo medio de representación del modelo de abstracción en esta versión del framework. Las mismas deben ser definidas en cualquier lenguaje compatible con .NET.

Al igual que en la versión anterior, la implementación del código de los aspectos es realizada en un lenguaje compatible con .NET. Durante la implementación de los aspectos deben respetarse ciertas convenciones de naming en las clases y métodos que lo conforman, de manera de asociarlos con los advices definidos mediante LENDL.

En esta versión se eliminan algunos elementos de la configuración que existían en la versión original asociados al uso de ontologías, como ser el modelo semántico definido en OWL y las reglas de inferencia. Estos elementos, junto con la información generada por el proceso de semanticación, conformaban el antiguo repositorio del universo semántico.

Otro cambio respecto de la versión original consiste en la integración de la implementación de los aspectos dentro del assembly de configuración (anteriormente los aspectos se encontraban en un assembly independiente).

3.4.4 Motor de Setpoint

Los assemblies anteriormente mencionados son coordinados por este componente, el cual se conforma por el assembly *setpoint.dll*. Luego que el assembly aspectualizable efectúa la primera llamada a un método, el weaver toma el control y son cargados en memoria todos los elementos necesarios para la toma de decisiones sobre la aplicación de aspectos: advices, pointcuts, aspect factories, mappers. Luego se determina a qué pointcuts pertenece

el join point en curso, se verifican los triggers asociados y finalmente se instancian y ejecutan los aspectos correspondientes.

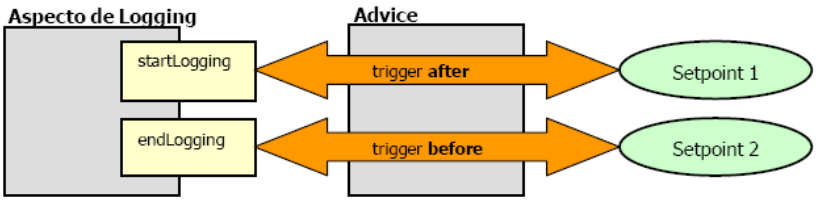
El comportamiento anteriormente expuesto es el mismo que utiliza el motor de Setpoint de la versión original. La diferencia se encuentra en la nueva forma de representar el universo semántico: ontologías y reglas de inferencia son reemplazadas por el uso de interfaces y mappers, los cuales son referenciados e inspeccionados por los pointcuts, advices y aspectos para obtener acceso al contexto en la forma de las vistas que fueron definidas en dicho universo.

3.5 Nuevos pointcuts, aspectos y advices en LENDL

Como se mencionó anteriormente, decidimos cambiar el lenguaje de modelado de perspectivas, pasando de OWL a orientación a objetos. Esto conlleva cambios en varios puntos de la solución propuesta: el modelado en sí de los dominios, la forma en que se especifican los pointcuts y advices y la forma en que se anota el código fuente.

Dado que el manejo de los conceptos reificados de la disciplina con respecto a advices y aspectos varía levemente con respecto a AspectJ daremos una breve explicación de cómo se aplican en este entorno antes de detallar su uso en particular.

Básicamente lo que ocurre es que en Setpoint se amplió el modelo para incorporar un nuevo elemento llamado trigger, el cual es un mensaje que indica al aspecto sobre un evento que le es relevante. De esta forma, un advice se encarga de agrupar los triggers que relacionan cada evento con los pointcuts que lo disparan. A continuación se muestra un diagrama que refleja este nuevo esquema para un ejemplo de logging:



A continuación describiremos la nueva forma de especificar pointcuts, advices y aspectos en LENDL, el lenguaje específico definido en la versión original de Setpoint, creado para definir estos elementos de manera declarativa.

3.5.1 Pointcuts

El tipo de eventos relevantes que se pueden identificar utilizando los pointcuts semánticos son representados mediante acciones, que a su vez se implementan mediante interfaces.

Primero veamos como se definían originalmente los pointcuts en LENDL. La idea consistía en especificar las asociaciones que deberían existir entre elementos del contexto (emisor, receptor y mensaje) y conceptos definidos en alguna ontología OWL. A continuación mostramos un ejemplo:

```
pointcut unPointcutComoAntes {
```

```
      sender is [semantics://anOntology#concept1];
      receiver is [semantics://anOntology#concept2];
      message is [semantics://anOntology#concept3];
}
```

Lo que se puede ver en esta definición de pointcut es cómo se asocia el emisor (`sender`) con el concepto `anOntology#concept1`. La asociación se realiza a través de la palabra clave **is**, que simboliza la relación hasAnnotation. Esto se traducirá posteriormente a una consulta OWL que determinará si el emisor tiene una anotación utilizando `anOntology#concept1`. Algo análogo se realiza con el receptor (`receiver`) y el mensaje (`message`). Este ejemplo muestra cómo se especificaban los pointcuts. Con esta definición se podía verificar la existencia en la ontología representada en OWL de la tupla <elemento del programa, hasAnnotation, concepto> mediante el motor de inferencia.

En el nuevo esquema, el chequeo de correspondencia entre conceptos y contexto se realiza de otra forma: lo que interesa es verificar si elementos del contexto cumplen con ciertas interfaces definidas en alguna vista del sistema. Pero los elementos del contexto con los que contamos ahora no están directamente vinculados al contexto de ejecución tradicional (emisor, receptor, mensaje y parámetros), sino que está determinado por el modelado de las acciones y los roles que diferentes objetos cumplen ante ellas.

Por ejemplo, podemos contar con un pointcut como el siguiente:

```
pointcut unPointcutConMensaje():
      message is IAcción1 && message.rol1 is IEntidad1 and message.rol2 is
IEntidad2;
```

Lo que indica este pointcut es que el mensaje debe cumplir la interfaz *IActión1* y que los roles *rol1* y *rol2* (roles definidos en la interfaz *IAcción1*) deben cumplir las interfaces *IEntidad1* e *IEntidad2* respectivamente.[2]

Desde esta perspectiva del programador de aspectos, la evaluación parecería producirse de manera directa, pero en realidad se realizan algunas operaciones que es importante resaltar. Al evaluar este pointcut, lo que se hace es instanciar un mapper para el join point que permita trabajar a partir de ese momento a un nivel de abstracción adecuado para la vista involucrada (el concepto de mapper es explicado en detalle más adelante). Este mapper cumple con la interfaz *IAcción1*, lo que permite acceder a los distintos roles (*rol1* y *rol2* en este caso) y realizar chequeos sobre ellos. Por otro lado cada chequeo sobre un rol implica una invocación a un método definido en el mapper para cumplir con la interfaz. Recién cumplidos estos pasos es posible realizar los chequeos sobre los roles.

Al igual que en AspectJ, los nuevos pointcuts definidos en LENDL permiten exportar parámetros, que luego podrán ser utilizados en diferentes aspectos. Ejemplificamos con el siguiente pointcut, una variación del anterior:

```
pointcut unPointcutConParámetros(string parámetro1, int parámetro2) :
      message is IAcción1 && message.rol1 is parámetro1 and message.rol2 is
parámetro2;
```

[2] Un mensaje ó un rol cumplen con una determinada interfaz si cumplen con alguna de sus sub-interfaces (desde el punto de vista de herencia entre interfaces).

En este caso, luego de obtener un mapper para el mensaje, se realizan los chequeos correspondientes a los roles, con la diferencia de que el chequeo es con respecto al tipo de la variable con la que esta asociada el rol. A su vez, una vez verificado el tipo, se realiza el binding con la variable en sí, lo cual permite la exportación.

La definición de los nuevos pointcuts permite componer expresiones mediante los conectivos lógicos and, or y not, permitiendo un mayor poder expresivo. A continuación un ejemplo de su uso:

```
pointcut unPointcutConConectivosLógicos(string parámetro1, int parámetro2):
      message is IAcción1 && not (message.rol1 is parámetro1) or message.rol2
is parámetro2;
```

Otra capacidad agregada consiste en la definición de pointcuts utilizando cflow y cflowbelow, con la semántica conocida de AspectJ, es decir, identificando join points basado en si ocurren o no en el contexto dinámico de otros joinpoints. Esta capacidad sigue teniendo sentido en el contexto de pointcuts semánticos excepto que ya no está atada a aspectos del código base sino que es una definición más conceptual. Un ejemplo de su uso:

```
pointcut otroPointcut():
      cflow(unPointcutConMensaje);
```

Por otro lado, queríamos conservar la posibilidad de predicar desde la perspectiva de código. Sabemos que si esta posibilidad es usada en forma exclusiva para predicar sobre el sistema lleva a aspectos acoplados con baja reutilización y difícil mantenimiento, pero a la vez creemos que podría aportar, usada con precaución, cierta luz sobre el sistema que otras vistas no pueden brindar. Para ello permitimos predicar sobre el emisor y el receptor, pudiendo chequear si estos pertenecen a cierta clase o cumplen determinada interfaz. A continuación mostramos un ejemplo de su uso:

```
pointcut unPointcutSobreCodigo():
      sender is UnaClase and not (receiver is OtraClase);
```

3.5.2 Aspectos

Los aspectos en Setpoint se definen mediante un nombre, la especificación de sus eventos relevantes y eventualmente el nombre de un factory. Los eventos definen su protocolo y serán vinculados en el advice a un conjunto de pointcuts y a un momento de ejecución (after, before). Mostramos a continuación un ejemplo de definición de aspecto:

```
aspect unAspecto builtby DefaultSingletonAspectFactory {
      event unEvento(int unParametro, string otroParametro);
      event otroEvento();
}
```

Esta definición sirve para validar los advices (ver siguiente sección), y no genera código para los aspectos. El código especifico de los aspectos debe ser desarrollado íntegramente por el programador de aspectos, en un modulo independiente y en un lenguaje compatible a .NET. En dicho modulo debe implementarse la clase que llevara el nombre del aspecto (en el caso del ejemplo, el nombre de la clase será `unAspecto`), la cual contendrá los eventos especificados en la definición del aspecto como métodos de la clase. En la definición del aspecto, los eventos tienen asociados la declaración de parámetros que corresponden a estos

métodos. El programador del modulo de aspectos deberá respetar dicha declaración de parámetros en la implementación de los eventos.

La cláusula opcional *builtby* permite indicar el factory que se encargará de la instanciación del aspecto. En caso de no especificarse se asumirá por defecto para la instanciación de aspectos el comportamiento de un Singleton (es decir, se creará una única instancia para el aspecto).

3.5.3 Advices

Por último definiremos la forma de notar los advices. Estos se encargan de asociar a través de los mencionados triggers a conjuntos de pointcuts con eventos pertenecientes a un aspecto.

Mostramos nuevamente un ejemplo para ilustrar la idea:

```
advice myAdvice : myAspect {
      trigger unEvento(p1, p2) after {unPointcutConParámetros(p2, p1)};
      trigger otroEvento() before { unPointcutConParámetros(p2, p1)};
}
```

La forma de declarar un advice consiste en especificar el nombre del advice y el aspecto asociado a los eventos que se mencionan dentro del bloque. Los eventos y los pointcuts que se asocian a ellos se anotan con los argumentos, de manera de mapear los parámetros exportados por los pointcuts en el conjunto asociado al evento.

En el ejemplo se ve cómo en el primer trigger el primer parámetro exportado por *unPointcutConParámetros* se asigna como segundo parámetro en *unEvento*. De la misma forma el segundo parámetro se asigna como primero en el evento.
En el segundo trigger ninguno de los parámetros del pointcut es necesario para *otroEvento*, por lo que no son asignados.

3.5.4 Pointcuts, aspectos y advices para la aplicación bancaria

Primero se define, en lenguaje LENDL, el pointcut `ModificaciónSaldoEnCajaDeAhorro`. Este pointcut estará conformado por aquellos métodos anotados con la interfaz de acción `IModificaciónDeSaldo`, en los que el rol `CuentaModificada` sea una caja de ahorro (es decir, cumpla con la interfaz `ICajaDeAhorro`). Para esos casos se transmitirá la información del contexto mediante el binding entre el rol `CuentaModificada` de dicha acción y la variable `ca`.

```
pointcut ModificaciónSaldoEnCajaDeAhorro(ICajaDeAhorro ca) :
    message is IModificaciónDeSaldo && message.CuentaModificada is ca;
```

A continuación presentamos la clase `NotificaciónSaldoNegativo`, que debe cumplir con la interfaz `IAspect` (predefinida por el framework). Aquí se programa el aspecto solicitado, en el método `NotificarDatosCuenta`. La información del contexto necesaria (es decir, el número de la cuenta y el saldo resultante) es recibida como parámetros del método.

```
public class NotificaciónSaldoNegativo  : IAspect
{
```

```
  void NotificarDatosCuenta (ICuenta cuenta)
  {
    if (cuenta.Saldo < 0)
    {
      Console.WriteLine("saldo resultante de la cuenta {0} es {1}.",
               cuenta.NúmeroDeCuenta,
               cuenta.Saldo);
    }
  }
}
```

Finalmente se especifica, nuevamente en código LENDL, el aspecto que debe contemplarse (en este caso, el aspecto `NotificaciónSaldoNegativo` definido anteriormente) y los eventos que lo conforman (en este caso, el método `NotificarDatosCuenta`). También se define el advice `Registrar`, en el cual se asocia el aspecto `NotificaciónSaldoNegativo` con el pointcut `ModificaciónSaldoEnCajaDeAhorro`. En dicha relación se establece la política de ejecución de los eventos del aspecto (triggers) y el binding de la información del contexto solicitada (los datos de la cuenta) entre los parámetros exportados por el pointcut y los parámetros del aspecto. En particular, se establece que el evento `NotificarDatosCuenta(cuenta)` deberá ejecutarse **luego** de la ejecución de aquellos join points que pertenezcan al pointcut `ModificaciónSaldoEnCajaDeAhorro`.

```
aspect NotificaciónSaldoNegativo  {
    event NotificarDatosCuenta(ICuenta cuenta);
}

advice Registrar: NotificaciónSaldoNegativo  {
    trigger NotificarDatosCuenta(cuenta) after {ModificaciónSaldoEnCajaDeAhorro(cuenta)};
```

3.6 Mappers

Necesitamos ahora un mecanismo que permita vincular la nueva representación del dominio con el código base, pero de forma que sea posible desacoplar la colaboración que existe entre los objetos del código base y el mundo semántico. Como dijimos, el mundo semántico es ahora representado por medio de interfaces, con lo cual debe existir un intermediario que permita abstraer a los "usuarios" de las interfaces (pointcuts y advices) del código. En el ambiente de la programación orientada a objetos, la forma más conocida para lograr esto es mediante el uso del *adapter pattern*. Lo que hace un adaptador es ubicarse en medio de una colaboración, con el fin de desacoplar ambos extremos. Llamaremos *mapper* al encargado de realizar esta tarea, y el mismo debe cumplir los siguientes objetivos:

- Agrupar aquellos objetos del código base necesarios para representar el comportamiento de cada una de las interfaces de *entidades* y *acciones*.
- Implementar las *propiedades* y *roles* de dichas interfaces.

Surge entonces una evidente clasificación que diferencia dos tipos de mappers: por un lado están los *mappers de acciones*, los cuales resuelven el mapeo de los roles de las acciones, y por otro lado los *mappers de entidades*, que hacen lo propio con las propiedades de la entidad.

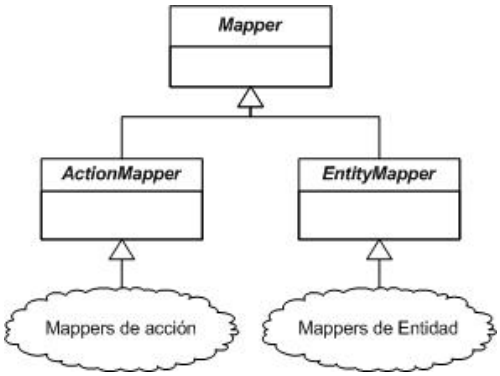


Figura 10 - Jerarquía de mappers en Setpoint

3.6.1 Mappers de acciones

Básicamente, los mappers de acciones envuelven los join points de bajo nivel y saben cómo acceder a los elementos que los conforman. Los join points de bajo nivel son elementos que reifican los eventos en tiempo de ejecución, tales como las llamadas a métodos, constructores, getters, setters, etc. Estos contienen el emisor del mensaje (sender), receptor del mensaje (receiver), los argumentos y el *selector* del mensaje.

En apartados previos modelamos, mediante el uso de intefaces, el dominio semántico correspondiente a nuestro ejemplo del sistema bancario, donde la interfaz de acción `IModificaciónDeSaldo` representa una operación que modifica el saldo de una cuenta. A continuación presentamos la definición del mapper correspondiente a una operación de modificación de saldo:

```
public class MapperModificaciónDeSaldo : ActionMapper, IModificaciónDeSaldo
{
  private ICuenta mCuentaModificada;

  public MapperModificaciónDeSaldo(IJoinPoint aJoinpoint) : base(aJoinpoint)
  {}

  public virtual ICuenta CuentaModificada {
    get {
      return EntityMapper.GetMapper<ICuenta>(new object[] { this.Receiver });
    }
  }
}
```

Como dijimos antes, los mappers se encargan de establecer el vinculo entre el mundo semántico (las interfaces) y el mundo sintáctico (el código base), razón por la cual los mappers necesariamente tienen que hacer referencia a elementos de ambos mundos. Vemos, por ejemplo, que en la definición del rol `CuentaModificada` correspondiente al mapper `MapperModificaciónDeSaldo` aparece tanto el concepto `ICuenta` del modelo de representación del dominio de las cuentas (mundo semántico) como el receptor del mensaje, que será representado por algún objeto del código base (mundo sintáctico).

Los roles de las acciones son implementados mediante el uso de getters. En muchos casos, los roles de las acciones no serán otra cosa que las entidades de nuestro modelo semántico, y por ello estarán "tipadas" como una interfaz de entidad. Por lo tanto, al acceder a un "rol de entidad" de una determinada acción, el mapper asociado a dicha acción devolverá el mapper de entidad correspondiente a ese rol. Un ejemplo de esto es el rol

`CuentaModificada` del mapper `MapperModificaciónDeSaldo`, en el cual se retorna un objeto que corresponde a un mapper de la entidad `ICuenta`.

Para la obtención de los mappers de entidad, los mappers de acción utilizan el método `GetMapper<I>` de la clase que engloba a los mappers de entidad (`EntityMapper`). Este método se encarga de devolver el mapper adecuado para los *mappees* especificados como parámetro. El método `GetMapper<I>` hace uso de una nueva facilidad que provee .NET conocida con el nombre de *generics*, que permite especificar en el momento de la llamada a dicho método la interfaz que debe implementar el mapper solicitado, y al mismo tiempo determinar el tipo que debe retornar el método. El uso de este mecanismo provee dos ventajas:

- Desacoplar los mappers de acción de los mappers de entidad. De esta forma es posible utilizar, por ejemplo, el mismo mapper de acción `MapperModificaciónDeSaldo` ya sea si el objeto que recibió el mensaje es una caja de ahorro ó una cuenta corriente. La búsqueda dinámica del mapper se encargará de devolver el mapper adecuado.
- Mediante el uso de generics es posible garantizar tipado seguro en tiempo de compilación.

Los mappers de acciones deben ser subclases de la superclase `ActionMapper`, la cual colabora con el join point de bajo nivel para proveer los siguientes servicios:

`Sender`	Devuelve el objeto que envió el mensaje
`Receiver`	Devuelve el objeto que recibió el mensaje
`GetParameterValue(n)`	Devuelve el n-ésimo argumento del mensaje
`Message`	Devuelve el selector del mensaje
`Arguments`	Devuelve un arreglo con los argumentos del mensaje
`GetReturnValue()`	Devuelve el valor de retorno del mensaje

3.6.2 Mappers de entidades

Los mappers de entidades envuelven un objeto (o un grupo de objetos) del código base. Los mismos tienen acceso a los miembros públicos de los objetos con los que colabora, con el fin de implementar las propiedades de las entidades con las que se encuentran asociados.

En el siguiente extracto de código, el mapper `MapperCajaDeAhorro` implementa la interfaz correspondiente a la entidad `ICajaDeAhorro` (perteneciente al mundo semántico), a partir de un objeto de tipo `CajaDeAhorro` (perteneciente al mundo sintáctico).

```
public class MapperCajaDeAhorro : EntityMapper, ICajaDeAhorro
{
  private CajaDeAhorro mCA;

  public virtual string NúmeroDeCuenta {
    get {
      return mCA.númeroDeCuenta;
    }

  public virtual double Saldo {
    get {
      return mCA.montoAcreditado - mCA.montoDebitado;
```

```
    }
  }

  public override object[] Mappee {
    set {
      this.mCA = (CajaDeAhorro)value[0];
    }
  }

  public override bool IsMapperFor(object[] mappee) {
    CajaDeAhorro ca = mappee[0] as CajaDeAhorro;
    return (ca != null);
  }
}
```

Adicionalmente, los mappers de entidad deben heredar de la superclase `EntityMapper`, y por lo tanto deberán implementar el método `IsMapperFor()` para validar si es un mapper válido para una determinada colección de mappees, así como también el setter de la propiedad `Mapee` para setear los mapees de dicho mapper.

3.7 Nuevas anotaciones

Las anotaciones son atributos de .NET (almacenados como metadatos) que relacionan los elementos del código base (métodos, clases, parámetros) con el modelo semántico.

En esta nueva versión de Setpoint se extiende el mecanismo de anotaciones que provee la versión original, ya que ahora no sólo se asocian los elementos del código con los conceptos del modelo, sino que además se debe indicar la relación entre los elementos del código con los mappers de entidades y de acciones.

Los atributos de .NET son un tipo especial de clases, con lo cual puede definirse una jerarquía de atributos con un comportamiento diferente en cada uno de sus miembros. Se aprovechará entonces esta facilidad que provee el lenguaje para crear cuatro tipos de anotaciones:

Anotaciones de entidad. Permite anotar una clase del código base con la interfaz de entidad y el mapper correspondiente. Es posible que varias clases del código se asocien con una misma entidad y con un mismo mapper, así como también es posible que una clase colabore con distintas entidades (y por lo tanto con distintos mappers).

Anotaciones de propiedad. Las anotaciones de propiedad asocian a los miembros de una clase del código base con la propiedad de una interfaz de entidad determinada. Los elementos que se pueden anotar deben ser de acceso público.

La caja de ahorro de nuestra aplicación bancaria se anotará de la siguiente forma para asociarla a la entidad `ICajaDeAhorro`, al mapper `MapperCajaDeAhorro` y a las propiedades `Saldo` y `NúmeroDeCuenta`:

```
[Entity(entityInterface="ICajaDeAhorro", mapper="MapperCajaDeAhorro")]
public class CajaDeAhorro {
  [Property(entityInterface="ICajaDeAhorro", property="Saldo")]
  public double saldo;

  [Property(entityInterface="ICajaDeAhorro", property="NúmeroDeCuenta")]
  public string númeroDeCuenta;

  ...
```

```
  }
```

Anotaciones de acción. Los métodos del código base tienen una correspondencia directa con las acciones del modelo semántico. Por ello las anotaciones de acción se encargan de asociar las interfaces de acciones con los métodos. Aquí también debe especificarse el mapper que implementa la interfaz de acción.

Anotaciones de rol. Las anotaciones de rol asocian un elemento del programa con el rol de una interfaz de acción determinada. Los elementos del programa que se pueden anotar son los siguientes:

- *Métodos*: En este caso debe especificarse un flag adicional para indicar la siguiente información:
  - si el rol se asocia con el objeto llamador del método (sender) o con el objeto que lo posee (receiver) en el contexto de ejecución de dicho método.
  - si se utilizará el nombre del método para definir un determinado rol (por ejemplo, un rol que represente el nombre de la acción).
- *Parámetros*: permite asociar un rol con los parámetros de los métodos

Veamos entonces cómo quedaría anotado el método `DebitarMonto()` de la caja de ahorro en la aplicación bancaria para asociarlo a la acción `IModificaciónSaldo` y a los roles `CuentaModificada` y `Monto`:

```
...
  [Action(actionInterface="IModificaciónDeSaldo", mapper="MapperModificaciónDeSaldo")]
  [Role(actionInterface="IModificaciónDeSaldo", role="CuentaModificada",
   roleElement=RoleElement.Receiver)]
  public void DebitarMonto(double monto)
  {
      ...
  }
...
```

Puede apreciarse en este ejemplo que la forma de anotar el rol `CuentaModificada` especificando el parámetro `roleElement=RoleElement.Receiver` implica que dicho rol debe ser cumplido por el objeto que recibe el mensaje al momento de efectuarse la llamada del mismo (en este caso, será un objeto de tipo `CajaDeAhorro`).

Como parte de la nueva versión de setpoint, hemos desarrollado una herramienta que, a partir del código base anotado y las interfaces del modelo semántico, genera automáticamente la estructura de los mappers (con todos sus miembros) y completa el código de las propiedades y roles en los casos donde el mapeo entre los conceptos y los elementos del código base es directo (por ejemplo, el mapeo entre la propiedad *NúmeroDeCuenta* de la entidad *ICajaDeAhorro* y el miembro `númeroDeCuenta` de la clase `CajaDeAhorro`).

Cabe aclarar que pueden existir casos en los que no haya un mapeo directo entre los elementos del código base y las anotaciones de roles y/o propiedades. En dichos casos estos mapeos deberán explicitarse manualmente en el módulo de los mappers.

3.8 Quantification y obliviousness en Setpoint

3.8.1 Quantification

En la nueva versión de Setpoint se predica sobre conceptos de la capa intermedia. Esto permite que diferentes elementos del código base puedan estar asociados al mismo concepto, como también puede ocurrir que un mismo elemento de código se corresponda con diferentes conceptos de la capa intermedia (por ejemplo podría tratarse de una clase que representa un cierto concepto de la vista del dominio del problema, a la vez que se asocia a cierto componente de la vista de arquitectura de software). Esta capacidad de abstracción se mantiene con respecto a la versión anterior de Setpoint, a pesar de los cambios realizados en la representación de la capa intermedia. También se cuenta con la utilización de conectivos lógicos para combinar restricciones sobre join points semánticos. El uso de *cflow* y *cflowbelow*, adoptado de AspectJ, permite al framework predicar y tener control sobre la dinámica de la pila de ejecución pero a nivel semántico, ya que sobre lo que se predica es sobre el orden de aparición de eventos de alto nivel (acciones)

La herencia es un mecanismo que provee en cierta forma una noción de quantification, ya que:

- Los métodos de subclases heredarán las anotaciones de acciones de sus ancestros.
- Las subclases heredarán las anotaciones de entidades de sus ancestros.
- Es posible definir una jerarquía de herencia en el modelo de acciones y entidades para conseguir un mayor grado de generalización en la definición de pointcuts.

3.8.2 Obliviousness

La propiedad de obliviousness es considerada valiosa porque libera a los programadores del código base del esfuerzo de hacer que los mecanismos de AOP funcionen correctamente. Este enfoque, defendido en el paper [FILMAN00], fue considerado prioritario durante un tiempo dentro de la comunidad de aspectos. En éste los autores reconocían de todas formas la necesidad en ciertos sistemas de que la propiedad de obliviousness no se cumpla por completo por cuestiones funcionales (mencionaban un ejemplo donde la aplicación necesitaba comunicarse con los aspectos para informarles sobre la prioridad de las tareas para poder proveer a las de mayor prioridad con una mayor calidad de servicio a través de aspectos). Sin embargo algunos autores [PARADOX] [CLIFTON03] [XPI] van más allá de esta afirmación, señalando que la propiedad de obliviousness podría perjudicar la evolución de los sistemas basados en aspectos como también su comprensión y la capacidad de debugging.

El objetivo de este trabajo no es determinar la importancia de obliviousness en AOP o en que grado o forma debería respetarse, por lo que nos limitaremos simplemente a describir el framework con respecto a esta propiedad. Nuestro enfoque no es disruptivo con respecto a obliviousness, como podría serlo XPI y solo requiere, en su uso más elemental, de anotaciones sobre los métodos para marcar su correspondencia con eventos de más alto nivel. El resto de las anotaciones son meramente para facilitar la generación de los mappers, pero estos últimos podrían escribirse sin anotaciones en el código. En su uso más intrusivo (poniendo anotaciones sobre el modelo) no vincula directamente el código fuente con los aspectos sino con vistas del sistema, que pueden ayudar a la trazabilidad con respecto a las vistas del sistema.

4 Casos De Estudio

En esta sección presentamos dos casos de estudio donde se aplican aspectos mediante la solución propuesta. Sirven por un lado para mostrar el uso del framework y por otro para resaltar ciertos inconvenientes en el enfoque que deben ser considerados tanto en su uso como para trabajo futuro sobre Setpoint.

En los casos de estudio se presentan varios escenarios que reflejan distintas implementaciones para un mismo problema. Para ellos se define una representación del dominio sobre la que luego se articulan los pointcuts, advices y aspectos. Todos estos elementos son comunes para todos los escenarios. Por otro lado se muestran las anotaciones en el código base y la definición de los mappers de acciones y entidades para los distintos escenarios.

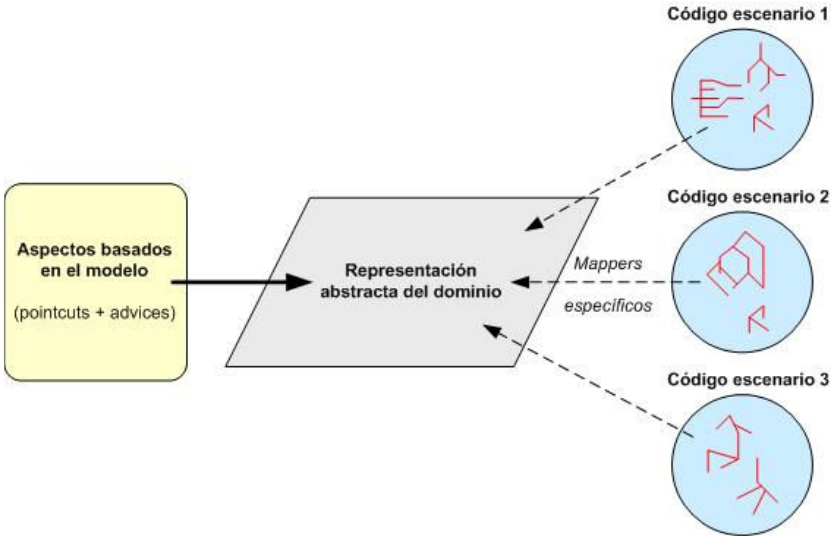


Figura 11 – Distintos escenarios del mismo dominio comparten un único modelo de representación

4.1 Caso de Estudio 1: Cuentas Bancarias

4.1.1 Introducción

En esta sección presentamos el problema de las cuentas bancarias utilizado durante el desarrollo de nuestra tesis, contemplando además varios escenarios con distintas variantes en el código base. Luego se modela el dominio del problema y se definen todos los elementos necesarios (poinctuts, advices, aspectos, anotaciones, mappers) que deben ser considerados para el nuevo framework de Setpoint.

4.1.2 Conocimiento del dominio

Existen [cuentas]
Cada [cuenta] tiene un [número de cuenta]
El [número de cuenta] es un [string]
Cada [cuenta] tiene un [saldo]
El [saldo] es un [número racional]
Existen [operaciones] que [modifican el saldo de una cuenta]*
Cada [operación] tiene un [nombre]*
El [nombre] es un [string] *

Las [cajas de ahorro] son cuentas

* Esta información podría ser implícita, porque todas las aplicaciones tienen operaciones con nombre que modifican sus datos

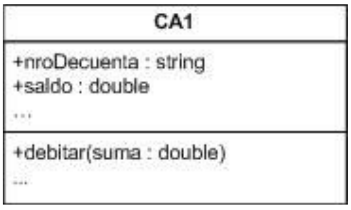
4.1.3 Requerimiento

Después de cada [operación] que [modifique el saldo de una caja de ahorro], si el [saldo] de esa [caja de ahorro] es negativo, registrar el [número de cuenta], el [nombre] de la [operación] y el [saldo].

4.1.4 Código base

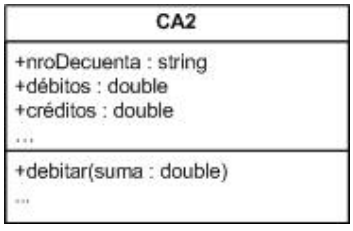
4.1.4.1 Escenario 1

En este escenario presentamos uno de los casos más sencillos de implementación de las cuentas bancarias. Consiste en la caja de ahorro `CA1` que posee, entre otras cosas, el atributo `nroDeCuenta` para representar el número de la cuenta de dicha caja de ahorro, el atributo `saldo` para almacenar el valor del saldo que posee la cuenta actualmente, y el método `debitar()`, que decrementa el saldo de la cuenta según la suma de dinero indicada como parámetro.



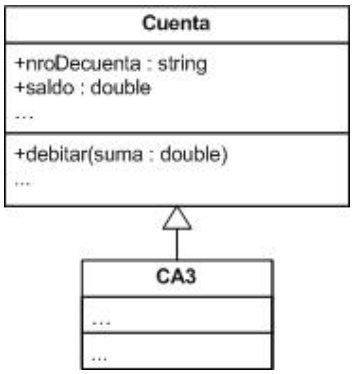
4.1.4.2 Escenario 2

En este escenario definimos la caja de ahorro `CA2`, la cual posee una variante respecto de la caja de ahorro `CA1` del escenario anterior; aquí no existe el atributo `saldo`, pero aparecen dos nuevos atributos: `débitos` y `créditos`. El atributo `débitos` almacena los montos acumulados de las operaciones de débito efectuadas sobre la cuenta, mientras que el atributo `créditos` representa los montos acumulados de las operaciones de débito. Luego, la forma de obtener el saldo es ahora en función de dichos atributos, es decir, *`créditos - débitos`*.



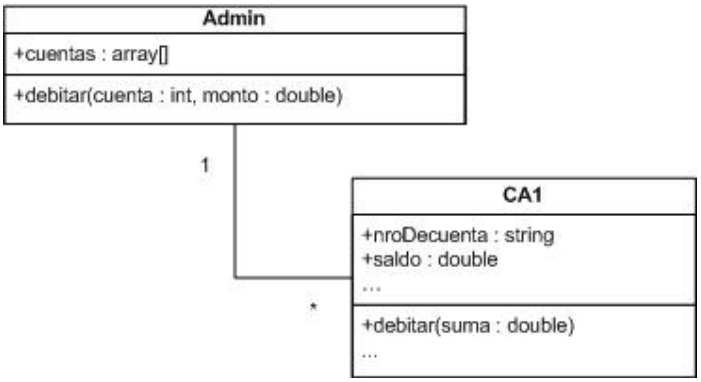
4.1.4.3 Escenario 3

En este escenario introducimos una jerarquía de clases, donde existe una clase genérica que engloba el comportamiento a las cuentas bancarias. Dicha clase, llamada `Cuenta`, está conformada por los mismos miembros de la clase del escenario 1. Las cajas de ahorro son ahora una especialización de la clase `cuenta`, y están representadas por la clase `CA3`.



4.1.4.4 Escenario 4

Este escenario consiste en un administrador de las cajas de ahorro del escenario 1. La clase `Admin` representa básicamente una colección de cajas de ahorro de tipo `CA1`. Este administrador cuenta con un método `debitar`, el cual recibe la suma de dinero a debitar y un índice para identificar la caja de ahorro dentro de la colección sobre la que se desea efectuar la operación.



4.1.5 Definiciones

A continuación se enumeran las definiciones que se deben realizar para aspectualizar el problema de las cuentas bancarias:

1. Representación del conocimiento del dominio
2. Mappers de acciones y entidades
3. Anotaciones semánticas en el código
4. Pointcuts
5. Aspecto a ejecutar
6. Advice

4.1.6 Representación del conocimiento del dominio

En este punto se modelan los elementos que conforman el dominio del problema mediante el uso de interfaces:

```
namespace SemanticWorld
{
  public interface ICuenta {
```

```
    public string NúmeroDeCuenta { get; }
    public double Saldo { get; }
  }

  public interfaceICajaDeAhorro : ICuenta {
  }

  public interface IModificaciónDeSaldo {
    public ICuenta CuentaModificada { get; }
    public string Nombre { get; }
  }
}
```

`IModificaciónDeSaldo` es la acción que representa las operaciones que modifican el saldo de una cuenta. Dicha acción posee dos roles: `Nombre`, que corresponde al nombre de la operación que modifica el saldo de una cuenta y `CuentaModificada`, que corresponde a la entidad `ICuenta` y representa a las cuentas bancarias. La entidad `ICuenta` contiene las propiedades `NúmeroDeCuenta` (de tipo `string`) y `Saldo` (de tipo `double`). Finalmente, se establece que la entidad `ICajaDeAhorro` especializa a la entidad `ICuenta`, definiendo de esta forma que una caja de ahorro es un tipo particular de cuenta.

4.1.7 Pointcut

Aquí se define, en lenguaje LENDL, el pointcut `ModificaciónSaldoEnCajaDeAhorro`. Este pointcut estará conformado por aquellos métodos anotados con la interfaz de acción `IModificaciónDeSaldo`, en los que el rol `CuentaModificada` sea una caja de ahorro (es decir, cumpla con la interfaz `ICajaDeAhorro`). Para la transferencia de la información del contexto se producirá un binding entre los roles `Nombre` y `CuentaModificada` de dicha acción y las variables `nombre` y `ca` respectivamente.

```
pointcut ModificaciónSaldoEnCajaDeAhorro(string nombre, ICajaDeAhorro ca) :
    message is IModificaciónDeSaldo && message.Nombre is nombre and message.CuentaModificada
is ca;
```

4.1.8 Aspecto a ejecutar

Definimos ahora la clase `NotificaciónSaldoNegativo`, que debe cumplir con la interfaz `IAspect` (predefinida por el framework). Aquí se programa el aspecto solicitado, en el método `NotificarDatosCuenta`. La información del contexto necesaria (es decir, el nombre de la operación, el número de la cuenta y el saldo resultante) son recibidas como parámetros del método.

```
public class NotificaciónSaldoNegativo : IAspect
{
  public void NotificarDatosCuenta (string nombre, ICuenta cuenta)
  {
    if (cuenta.Saldo < 0)
    {
      Console.WriteLine("La operación {0} dejó en {1} el saldo de la cuenta {2}.",
              nombre,
              cuenta.Saldo,
              cuenta.NúmeroDeCuenta);
    }
  }
}
```

4.1.9 Advice

En este fragmento de código LENDL se especifican los aspectos que deben contemplarse en los casos de estudio (en este caso, el aspecto `NotificaciónSaldoNegativo` definido anteriormente) y los eventos que lo conforman (en este caso, el método `NotificarDatosCuenta`). También se define el advice `Registrar`, en el cual se asocia el aspecto `NotificaciónSaldoNegativo` con el pointcut `ModificaciónSaldoEnCajaDeAhorro`. En dicha relación se establece la política de ejecución de los eventos del aspecto (triggers) y el binding de la información del contexto solicitada (el nombre de la operación y la cuenta) entre los parámetros exportados por el pointcut y los parámetros del aspecto. En particular, se establece que el evento `NotificarDatosCuenta(nombre, cuenta)` deberá ejecutarse luego de la ejecución de aquellos join points que pertenezcan al pointcut `ModificaciónSaldoEnCajaDeAhorro`,

```
aspect NotificaciónSaldoNegativo {
  event NotificarDatosCuenta(string nombre, ICuenta cuenta);
}

advice Registrar: NotificaciónSaldoNegativo {
  trigger NotificarDatosCuenta (nombre, cuenta)
    after {ModificaciónSaldoEnCajaDeAhorro(nombre, cuenta);
}
```

4.1.10 Mappers y anotaciones

A continuación se presenta la definición de los mappers de acciones y entidades que implementan el dominio del problema modelado con las interfaces, indicando además las anotaciones que deben efectuarse en el código.

Para cada escenario de código base existe una definición de mappers específicos. Los mappers fueron generados previamente en forma automática por la herramienta de generación de mappers construida por nosotros. Dependiendo del caso, y por motivos explicados anteriormente, algunos getters de los mappers debieron ser completados manualmente luego de la generación (esta distinción se aclara mediante comentarios en el código presentado).

Respecto de las anotaciones, deben anotarse aquellos métodos que modifiquen el saldo de las cajas de ahorro, como también aquellas entidades, roles y propiedades que puedan ser identificadas de forma directa en el código. Para cada entidad y acción anotada se especifica la interfaz del dominio semántico a la que representan y el mapper que la implementa.

4.1.10.1 Mappers y anotaciones del Escenario 1

A continuación se presentan las anotaciones semánticas efectuadas en la clase `CA1`:

```
[Entity(entityInterface="ICajaDeAhorro", mapper="MapperCA1")]
public class CA1 {
  [Property(entityInterface="ICajaDeAhorro", property="Saldo")]
  double saldo;

  [Property(entityInterface="ICajaDeAhorro", property="NúmeroDeCuenta")]
  string númeroDeCuenta;

  [Action(actionInterface="IModificaciónDeSaldo", mapper="MapperModificaciónDeSaldo")]
  [Role1(actionInterface="IModificaciónDeSaldo", role="CuentaModificada",
   roleElement=RoleElement.Receiver)]
  [Role2(actionInterface="IModificaciónDeSaldo", role="Nombre",
```

```
   roleElement=RoleElement.MessageName)]
  void Debitar (double suma)
  {
    this.saldo -= suma;
  }

  ...
}
```

El siguiente mapper de acción corresponde la implementación de las operaciones de modificación de saldo que se efectúan sobre las cuentas: `MapperModificaciónDeSaldo`. Este mapper, que implementa la interfaz de acción `IModificaciónDeSaldo`, posee las propiedades `Nombre` y `CuentaModificada`. La propiedad `Nombre` devolverá el nombre del mensaje interceptado, en este caso, "Debitar". La propiedad `CuentaModificada` retornará el mapper para el objeto de tipo `CA1 (MapperCA1)`. El objeto `CA1` es obtenido inspeccionando el receptor del mensaje (receiver).

```
class MapperModificaciónDeSaldo : ActionMapper, IModificaciónDeSaldo {

  // **Generado automáticamente**
  private ICuenta mCuentaModificada;

  public MapperModificaciónDeSaldo(IJoinPoint aJoinpoint) : base(aJoinpoint)
  {}

  public string Nombre {
    get {
      // **Generado automáticamente**
      return this.Message.name;
    }
  }

  public virtual ICuenta CuentaModificada {
    get {
      // **Generado automáticamente**
      if (mCuentaModificada == null) {
        mCuentaModificada = EntityMapper.GetMapper<ICuenta>(new object[] { this.Receiver });
      }
      return mCuentaModificada;
    }
  }
}
```

El mapper de entidad `MapperCA1` implementa la interfaz de entidad `ICajaDeAhorro` a partir de un objeto de tipo `CA1`:

```
public class MapperCA1 : EntityMapper, ICajaDeAhorro {

  // **Generado automáticamente**
  private CA1 mCA1;

  public virtual string NúmeroDeCuenta {
    get {
      // **Generado automáticamente**
      return mCA1.númeroDeCuenta;
    }
  }

  public virtual double Saldo {
    get {
      // **Generado automáticamente**
      return mCA1.saldo;
    }
  }

  public override object[] Mappee {
    set {
      // **Generado automáticamente**
      this.mCA1 = (CA1)value[0];
    }
```

```
  }

  public override bool IsMapperFor(object[] mappee) {
    // **Generado automáticamente**
    CA1 ca = mappee[0] as CA1;
    return (ca != null);
  }
}
```

4.1.10.2 Mappers y anotaciones Escenario 2

A continuación se presentan las anotaciones semánticas efectuadas en la clase `CA2`:

```
[Entity(entityInterface="ICajaDeAhorro", mapper="MapperCA2")]
public class CA2 {
  double débitos;
  double créditos;

  [Property(entityInterface="ICajaDeAhorro", property="NúmeroDeCuenta")]
  string númeroDeCuenta;

  [Action(actionInterface="IModificaciónDeSaldo", mapper="MapperModificaciónDeSaldo")]
  [Role1(actionInterface="IModificaciónDeSaldo", role="CuentaModificada",
   roleElement=RoleElement.Receiver)]
  [Role2(actionInterface="IModificaciónDeSaldo", role="Nombre",
   roleElement=RoleElement.MessageName)]
  void Debitar (double suma) {
    this.débitos += suma;
  }

  ...
}
```

Los mappers de este escenario son similares a los mappers del escenario 1. La principal diferencia radica en que la forma de representar el saldo en esta clase de cajas de ahorro es distinta a la del escenario anterior, ya que `saldo` no existe explícitamente como un atributo de la clase, sino que debe ser determinado como la diferencia entre los atributos `créditos` y `débitos`. Por este motivo deberá definirse un mapper de entidad específico para la clase de la caja de ahorro correspondiente a este escenario (`CA2`). El mapper de acción de este escenario es igual al mapper de acción del escenario anterior.

```
class MapperModificaciónDeSaldo : ActionMapper, IModificaciónDeSaldo {

  // **Generado automáticamente**
  private ICuenta mCuentaModificada;

  public MapperModificaciónDeSaldo(IJoinPoint aJoinpoint) : base(aJoinpoint)
  {}

  public string Nombre {
    get {
      // **Generado automáticamente**
      return this.Message.name;
    }
  }

  public virtual SemanticWorld.ICuenta CuentaModificada {
    get {
      // **Generado automáticamente**
      if (mCuentaModificada == null) {
        mCuentaModificada = EntityMapper.GetMapper<ICuenta>(new object[] { this.Receiver });
      }
      return mCuentaModificada;
    }
  }
}
```

```
public class MapperCA2 : EntityMapper, ICajaDeAhorro {
```

```
  // **Generado automáticamente**
  private CA2 mCA2;

  public virtual string NúmeroDeCuenta {
    get {
      // **Generado automáticamente**
      return mCA2.númeroDeCuenta;
    }
  }

  public virtual double Saldo {
    get {
      // **Agregado manualmente**
      return mCA2.creditos - mCA2.debitos;
    }
  }

  public override object[] Mappee {
    set {
      // **Generado automáticamente**
      this.mCA2 = (CA2)value[0];
    }
  }

  public override bool IsMapperFor(object[] mappee) {
    // **Generado automáticamente**
    CA2 ca = mappee[0] as CA2;
    return (ca != null);
  }

}
```

4.1.10.3 Mappers y anotaciones Escenario 3

A continuación se presentan las anotaciones semánticas efectuadas en la clase `Cuenta` y en la subclase `CA3`:

```
[Entity(entityInterface="ICuenta", mapper="MapperCuenta")]
public class Cuenta {
  [Property(entityInterface="ICuenta", property="Saldo")]
  double saldo;

  [Property(entityInterface="ICuenta", property="NúmeroDeCuenta")]
  string númeroDeCuenta;

  [Action(actionInterface="IModificaciónDeSaldo", mapper="MapperModificaciónDeSaldo")]
  [Role1(actionInterface="IModificaciónDeSaldo", role="CuentaModificada",
   roleElement=RoleElement.MessageName)]
  [Role2(actionInterface="IModificaciónDeSaldo", role="Nombre",
   roleElement=RoleElement.MessageName)]
  void Debitar (double suma) {
    this.saldo -= suma;
  }
}
```

```
[Entity(entityInterface="ICajaDeAhorro", mapper="MapperCA3")]
public class CA3 : Cuenta {
  ...
}
```

La particularidad que puede encontrarse en la definición de los mappers de este escenario radica en la jerarquía de clases que se establece entre los mappers de entidad: el mapper `MapperCA3`, que implementa la interfaz de entidad `ICajaDeAhorro`, es además una subclase del mapper `MapperCuenta`, el cual implementa la interfaz de entidad `ICuenta`. De esta forma se aprovechan las propiedades que ofrece el mecanismo de herencia para reutilizar en la subclase `MapperCA3` el comportamiento definido en la superclase `MapperCuenta`. Nuevamente, el mapper de acción de este escenario es igual al mapper de acción de los escenarios anteriores.

```
class MapperModificaciónDeSaldo : ActionMapper, IModificaciónDeSaldo {

  // **Generado automáticamente**
  private ICuenta mCuentaModificada;

  public MapperModificaciónDeSaldo(IJoinPoint aJoinpoint) : base(aJoinpoint)
  {}

  public string Nombre {
    get {
      // **Generado automáticamente**
      return this.Message.name;
    }
  }

  public virtual ICuenta CuentaModificada {
    get {
      // **Generado automáticamente**
      if (mCuentaModificada == null) {
        mCuentaModificada = EntityMapper.GetMapper<ICuenta>(new object[] { this.Receiver });
      }
      return mCuentaModificada;
    }
  }
}
```

```
public class MapperCuenta : EntityMapper, ICuenta {

  // **Modificado manualmente**
  protected Cuenta mCuenta;

  public string NúmeroDeCuenta {
    get {
      // **Generado automáticamente**
      return mCuenta.númeroDeCuenta;
    }
  }

  public double Saldo {
    get {
      // **Generado automáticamente**
      return mCuenta.saldo;
    }
  }

  public override object[] Mappee {
    set {
      // **Generado automáticamente**
      this.mCuenta = (Cuenta)value[0];
    }
  }

  public override bool IsMapperFor(object[] mappee) {
    // **Generado automáticamente**
    Cuenta cta = mappee[0] as Cuenta;
    return (cta != null);
  }
}
```

```
public class MapperCA3 : MapperCuenta, ICajaDeAhorro
{
  // **Generado automáticamente**
  private CA3 mCA3;

  ...

  public override object[] Mappee {
    set {
      // **Modificado manualmente**
       base.Mappee = value;
       this.mCA3 = (CA3)value[0];
    }
  }
```

```
  public override bool IsMapperFor(object[] mappee) {
    // **Generado automáticamente**
    CA3 ca = mappee[0] as CA3;
    return (ca != null);
  }
}
```

4.1.10.4 Mappers y anotaciones Escenario 4

A continuación se presentan las anotaciones semánticas efectuadas en la clase `Admin`:

```
public class Admin
{
  CA1[] cuentas;

  [Action(actionInterface="IModificaciónDeSaldo", mapper="MapperModificaciónDeSaldoAdmin")]
  [Role(actionInterface="IModificaciónDeSaldo", role="Nombre",
   roleElement=RoleElement.MessageName)]
  void Debitar (int cuenta, double monto)
  {
    ...
  }

  ...
}
```

El mayor grado de complejidad de mapeo que se ha dado hasta ahora puede encontrarse en este escenario; específicamente en la propiedad `CuentaModificada` del mapper de acción `MapperModificaciónDeSaldoReceiverAdmin`. Puede observarse que el objeto receptor del mensaje correspondiente al join point es una colección de cajas de ahorro, y la forma de obtener la caja de ahorro cuyo saldo es modificado es accediendo a la posición de la colección indicada por el argumento del mensaje (que corresponde al índice que determina la caja de ahorro dentro de la colección).

```
public class MapperModificaciónDeSaldoAdmin : ActionMapper, IModificaciónDeSaldo
{
  // **Agregado manualmente**
  private ICuenta mCuentaModificada;

  public MapperModificaciónDeSaldoAdmin(IJoinPoint aJoinpoint) : base(aJoinpoint)
  {}

  public string Nombre {
    get {
      // **Generado automáticamente**
      return this.Message.name;
    }
  }

  public virtual ICuenta CuentaModificada
  {
    get
    {
      // **Agregado manualmente**
      if (mCuentaModificada == null)
      {
        int indice = ((int)(this.GetParameterValue(0)));
        Admin admin = (Admin)this.Receiver;
        mCuentaModificada =
               EntityMapper.GetMapper<ICuenta>(new object[] { admin.cuentas[indice] });
      }
      return mCuentaModificada;
    }
  }
}
```

```
public class MapperCA1 : EntityMapper, ICajaDeAhorro {
```

```
  // **Generado automáticamente**
  private CA1 mCA1;

  public virtual string NúmeroDeCuenta {
    get {
      // **Generado automáticamente**
      return mCA1.númeroDeCuenta;
    }
  }

  public virtual double Saldo {
    get {
      // **Generado automáticamente**
      return mCA1.saldo;
    }
  }

  public override object[] Mappee {
    set {
      // **Generado automáticamente**
      this.mCA1 = (CA1)value[0];
    }
  }

  public override bool IsMapperFor(object[] mappee) {
    // **Generado automáticamente**
    CA1 ca = mappee[0] as CA1;
    return (ca != null);
  }
}
```

4.2 Caso de estudio 2: Pipe and Filter

En esta sección mostramos un segundo caso de estudio, con el objetivo de reflejar en un dominio la existencia de una arquitectura definida mediante el estilo "pipe and filter", sobre la cual se puedan especificar aspectos. Se definieron dos implementaciones que permitieran analizar la flexibilidad de los mecanismos y del modelo de la capa intermedia en distintos escenarios. Presentamos dos escenarios. El primero presenta un comportamiento asincrónico, y está conformado por una serie de clases representando al pipe, al filter, a los paquetes, y al DataSource y DataSink, elementos para representar la entrada y salida de la arquitectura, a la vez que sirven de conectores intermedios entre pipes y filters. El segundo es sincrónico y simple. Se cuenta con una clase que modela el pipe and filter, junto a otra clase que se encarga de modelar el comportamiento particular de un filter, de manera de poder componerlos.

4.2.1 Conocimiento del dominio

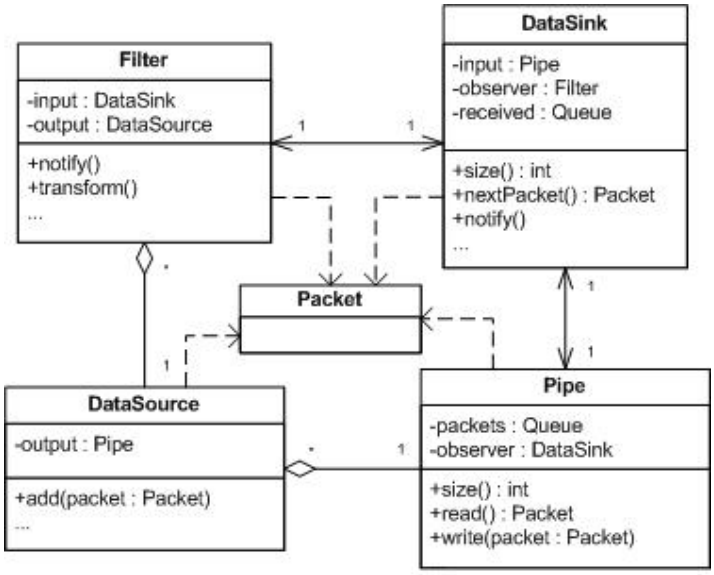
1. Existen [filters]
2. Cada [filter] tiene un [nombre]
3. El [nombre] es un [string]
4. Existen [pipes]
5. Los [filters] se conectan a través de [pipes]
6. Los [paquetes] son la unidad de información
7. Existe un [proceso] en el [filter] que transforma el [paquete]
8. Existen operaciones que toman el [paquete] del [pipe] y lo envían al [filter]
9. Existen operaciones que toman el [paquete] del [filter] y lo envían al [pipe]

4.2.2 Requerimiento

Cada vez que un [paquete] llega a un [filter] imprimir el [nombre] del [filter] y el [paquete] antes y después de ser procesado por el [filter]

4.2.3 Código Base

4.2.3.1 Escenario 1



4.2.3.2 Escenario 2

```
public abstract class Filter {
    public string name;
    public abstract string ProcessStream(string stream);

    public Filter (string name) {
        this.name = name;
    }
}

public class PipeAndFilter {
    public Filter[] pipeLine;

    public void Add(Filter filter) {
        pipeLine.Add(filter);
    }

    public string Process(string stream) {
        string s = stream;
        foreach(Filter f in pipeLine)
            s = f.ProcessStream(s);

        return s;
    }
}

public class Filter1 : Filter {
    public string ProcessStream(string stream) {
        ...
    }
}

public class Filter2 : Filter {
    public string ProcessStream(string stream) {
        ...
    }
}
```

4.2.4 Representación del conocimiento del dominio

Las interfaces reflejan la existencia de filters, pipes y paquetes de información. Los filters están conectados con pipes y los pipes con filters. A su vez se definen dos acciones, que representan el pasaje de un paquete de información desde un filter hacia un pipe, y desde un pipe hacia un filter. Estas acciones cuentan con un paquete y con un pipe o un filter, que es el elemento de salida del paquete, y del cual puede obtenerse el elemento vinculado en la arquitectura hacia donde se dirige el paquete.

```
public interface IPaquete {
    string Dato { get; }
}
```

```
public interface IFilter {
    string Nombre { get; }
    public IPipe Entrada { get; }
    public IPipe Salida { get; }
}
```

```
public interface IPipe {
    public IFilter Entrada { get; }
    public IFilter Salida { get; }
}
```

```
public interface IPasajePaqueteDelPipeAlFilter {
    public IPipe Pipe { get; }
    public IPaquete Paquete { get; }
}
```

```
public interface IPasajePaqueteDelFilterAlPipe {
    public IFilter Filter { get; }
    public IPaquete Paquete { get; }
}
```

4.2.5 Pointcuts

```
pointcut PasajePaqueteDelPipeAlFilter(IPipe pipe, IPaquete paquete) :
  message is IPasajePaqueteDelPipeAlFilter && message.Pipe is pipe and message.Paquete is
paquete;
```

```
pointcut PasajePaqueteDelFilterAlPipe(IFilter filter, IPaquete paquete) :
  message is IPasajePaqueteDelFilterAlPipe && message.Filter is filter and message.Paquete
is paquete;
```

4.2.6 Aspectos a ejecutar

La definición de aspectos es tal que recibe como parámetros el paquete que se esta transmitiendo, y el filtro involucrado, como receptor o como emisor del paquete. Los métodos se encargan de imprimir el filtro involucrado, así como el dato transmitido.

```
class LoggingAspect : IAspect {
    public void LogLlegadaPaquete (IPipe pipe, IPaquete paquete) {
          Console.WriteLine(
          String.Format(
             "El filtro {0} recibió del pipe de entrada el siguiente paquete: {1}.",
              pipe.Salida.Nombre, paquete.Dato));
    }

    public void LogSalidaPaquete (IFilter filter, IPaquete paquete) {
          Console.WriteLine(
          String.Format(
            "El filtro {0} depositó en el pipe de salida el siguiente paquete: {1}.",
             filter.Nombre, paquete.Dato));
    }
}
```

4.2.7 Advice

En el advice Logging se puede ver el vínculo establecido entre los pointcuts *PasajePaqueteDelPipeAlFilter* y *PasajePaqueteDelFilterAlPipe* y los métodos a ejecutar. En este vínculo se explicitan también el momento de la ejecución de los aspectos (*after*, *before*) y el pasaje de parámetros exportados por los pointcuts a los métodos asociados.
Más adelante en este caso de estudio se verá que a pesar de que en principio el momento de ejecución de aspectos visto desde esta perspectiva no altera el resultado final, detalles de implementación fuerzan decisiones, mostrando que existen casos en los que la independencia del código base a la hora de definir aspectos semánticos aún no es total.

```
aspect LoggingAspect {
    event LogLlegadaPaquete (IPipe pipe, IPaquete paquete);
    event LogSalidaPaquete (IFilter filter, IPaquete paquete);
}

advice Logging: LoggingAspect {
    trigger LogLlegadaPaquete(filter, paquete) before {PasajePaqueteDelPipeAlFilter(p,
paquete)};
    trigger LogSalidaPaquete(filter, paquete) after {PasajePaqueteDelFilterAlPipe(filter,
paquete)};}
```

4.2.8 Mappers y anotaciones en el código

4.2.8.1 Escenario 1

```
[Entity(entityInterface="IPaquete", mapper="MapperPaquete")]
public class Packet
{
    private Object dato;

    public Packet(Object dato)
    {
        this.dato = dato;
    }

    public override string ToString()
    {
        return this.dato.ToString();
    }
}

public class DataSource
{
    private Pipe output;

    ...

    [Action(actionInterface="IPasajePaqueteDelFilterAlPipe",
     mapper="MapperPasajePaqueteDelFilterAlPipe")]
    [Role(actionInterface="IPasajePaqueteDelFilterAlPipe", role="Filter",
     roleElement=RoleElement.Sender)]
    public virtual void add
        (
            [Role(actionInterface="IPasajePaqueteDelFilterAlPipe", role="Paquete")]
            Packet packet
        )
    {
        if (output != null)
        {
            this.output.write(packet);
        }
        else
        {
            throw new NoOutputPipeException();
        }
    }
```

```
    ...
}


public class DataSink
{
    private Pipe input;
    private Filter observer;
    protected Queue received;

    ...

    [Action(actionInterface="IPasajePaqueteDelPipeAlFilter",
     mapper="MapperPasajePaqueteDelPipeAlFilter")]
    public void notify()
    {
        while (this.input.size() > 0)
        {
            this.received.Enqueue(this.input.read());
        }
        if (observer != null)
        {
            this.observer.notify();
        }
    }

    [Role(actionInterface="IPasajePaqueteDelPipeAlFilter", role="Pipe")]
    public Pipe getInput()
    {
        return this.input;
    }

    /**
      * Este método es requerido por el mapper MapperPasajePaqueteDelPipeAlFilter
      * para inspeccionar el próximo paquete en la cola, sin removerlo
      */
     public Packet peekNextPacket() {
       if (this.packets.Count > 0) {
         return (Packet)this.packets.Peek();
       }
       else {
         return null;
       }
     }

     ...
}


[Entity(entityInterface="IPipe", mapper="MapperPipe")]
public class Pipe
{
    private Queue packets;
    private DataSink observer;

    ...
}

[Entity(entityInterface="IFilter", mapper="MapperFilter")]
public class Filter
{
    DataSource output;
    DataSink input;

    private string name;

    [Property(entityInterface="IFilter", property="Nombre")]
    public string Name
    {
        get { return this.name; }
    }

    public Filter(string name)
    {
        this.name = name;
```

```
    }

    ...
}
```

```
public class MapperPaquete : EntityMapper, IPaquete
{
    private Packet mPacket;

    public virtual string Dato {
        get {
            // **Agregado Manualmente**
            return this.mPacket.ToString();
        }
    }

    public override object[] Mappee
    {
         // **Generado Automáticamente**
        set { this.mPacket = (Packet)value[0]; }
    }

    public override bool IsMapperFor(object[] mappee)
    {
         // **Generado Automáticamente**
        Packet packet = mappee[0] as Packet;
        return (packet != null);
    }
}
```

```
public class MapperFilter : EntityMapper, IFilter
{
    private PipeAndFilter.Filter mFilter;

    public virtual string Nombre {
        get {
            // **Generado Automáticamente**
            return this.mFilter.Name;
        }
    }

    public virtual SemanticWorld.IPipe Entrada {
        get {
            // **Agregado Manualmente**
            return EntityMapper.GetMapper<IPipe>(new
                                    object[] { this.mFilter.getDataSink().getInput() });
            }
        }

    public virtual SemanticWorld.IPipe Salida {
        get {
            // **Agregado Manualmente**
            return EntityMapper.GetMapper<IPipe>(new
                                object[] { this.mFilter.getDataSource().getOutput() });
            }
        }

    public override object[] Mappee {
             // **Generado Automáticamente**
            set { this.mFilter = (Filter)value[0]; }
        }

        public override bool IsMapperFor(object[] mappee)
        {
            // **Generado Automáticamente**
            PipeAndFilter.Filter filter = mappee[0] as Filter;
            return (filter != null);
        }
    }
```

```
public class MapperPipe : EntityMapper, IPipe {

    private PipeAndFilter.Pipe mPipe;

    public virtual SemanticWorld.IFilter Entrada {
```

```
        get {
            // **Agregado Manualmente**
            return null; // no puede obtenerse la referencia necesaria a partir de la
                         implementación
            }
        }

    public virtual SemanticWorld.IFilter Salida {
        get {
            // **Agregado Manualmente**
            return EntityMapper.GetMapper< IFilter>(new object[] {

    this.mPipe.getObserver().getObserver()});
            }
        }

    public override object[] Mappee {
         // **Generado Automáticamente**
        set { this.mPipe = (PipeAndFilter.Pipe)value[0]; }
    }

    public override bool IsMapperFor(object[] mappee) {
         // **Generado Automáticamente**
        PipeAndFilter.Pipe pipe = mappee[0] as PipeAndFilter.Pipe;
        return (pipe != null);
    }
}
```

En el mapeo anterior no es posible conseguir la referencia al filtro de entrada. A pesar de que la implementación en este caso cuenta con una descripción muy rica de clases que representan distintos elementos (pipes, filters, conectores), la conexión entre ellos no es tan completa como la que se especifica en la definición conceptual, por lo que no es posible obtener el filtro anterior desde el pipe actual. Por otro lado en esta situación puede apreciarse que a pesar de que el modelo planteado conceptualmente permite una navegación a través de los distintos elementos que componen la arquitectura, en la práctica se reduce lo navegable a lo accesible en el contexto de ejecución. Esto supone limitaciones a la visión de los aspectos sobre el sistema, ya que lo único sobre lo que pueden tener información es sobre el contexto semántico de ejecución, pero no tener una perspectiva más global del sistema. Por ejemplo, podría interesar loguear los paquetes de un filter si el filter anterior tiene cierta característica, para lo cual es necesario tener un alcance mayor desde los aspectos. Esto podría conseguirse si se utilizaran mecanismos que permitan definir, actualizar y consultar perspectivas más amplia del sistema, con un alcance mayor al del contexto de ejecución actual y agregando la noción de estado.

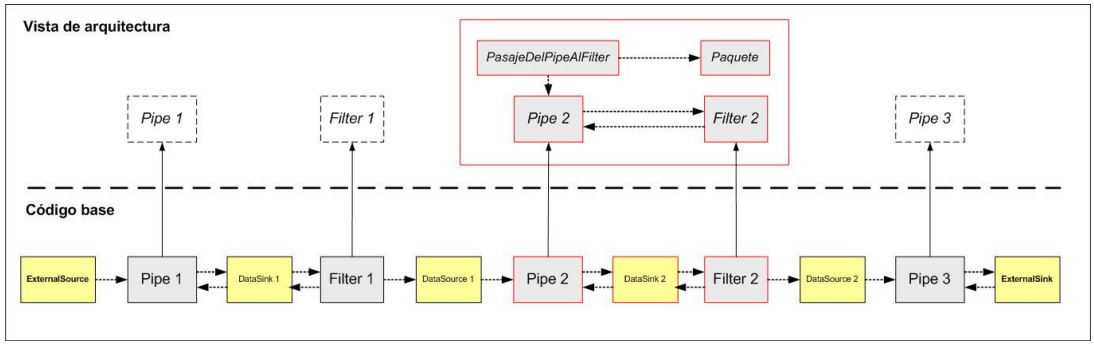


Figura 12 – Visión limitada de la arquitectura en el contexto de la acción *IPasajeDelPipeAlFilter*

```
public class MapperPasajePaqueteDelPipeAlFilter : ActionMapper,
                                                  IPasajePaqueteDelPipeAlFilter
{
```

```
    private IPaquete mPaquete;
    private IPipe mPipe;

    public MapperPasajePaqueteDelPipeAlFilter(IJoinPoint aJoinpoint) : base(aJoinpoint)
    {}

    public virtual IPipe Pipe {
        get {
            // **Agregado Manualmente**
            if (mPipe == null) {
                mPipe = EntityMapper.GetMapper<IPipe>(new
                            object[]{((DataSink)this.Receiver).getInput()});
            }
            return mPipe;
        }
    }

    public virtual IPaquete Paquete {
        get {
            // **Agregado Manualmente**
            if (mPaquete == null)
            {
                Pipe p = ((DataSink)this.Receiver).getInput();
                mPaquete = EntityMapper.GetMapper<IPaquete>(new object[] {
                                                        p.peekNextPacket()});
            }
            return mPaquete;
        }
    }
}
```

En este último mapper aparecen dos inconvenientes relativos a la obtención del paquete de información.

Por un lado aparece una cierta dificultad al acceder a los datos. Esta se debe a las características del objeto finalmente consultado, que es la implementación de una cola. La clase que la implementa es `Queue`, donde un objeto de esta clase quita un elemento de cola al invocar al método `read()`. Si la llamada al método es realizada desde un aspecto se produce una alteración del estado del sistema original, lo que se traduce en este caso en una pérdida de información. Para solucionarlo debe agregarse un método en la clase `Pipe` del código base, llamado `peekNextPacket()`, que obtiene el próximo paquete de la cola sin removerlo de la misma. Claramente, la solución presentada no respeta obliviousness del lado del código base. De todas maneras, este problema no es inherente a nuestra propuesta, sino que aparece también en otros frameworks de aspectos, como AspectJ.

Por otro lado existe cierta sensibilidad del mapper en relación al advice. Lo que ocurre es que dependiendo de si se utiliza el mapper vinculado a un *before advice* o a un *after advice* el paquete se encuentra en distinto lugar. De hecho si el aspecto es ejecutado posteriormente a la ejecución del método asociado, el paquete de información no es alcanzable, ya que una serie de métodos llamados en cadena se encargan de llevar el paquete hacia un objeto fuera de alcance. Es por este motivo que el programador de aspectos se ve obligado a utilizar un *before advice*, lo que implica un conocimiento por su parte de detalles de la implementación subyacente.

```
public class MapperPasajePaqueteDelFilterAlPipe : ActionMapper,
IPasajePaqueteDelFilterAlPipe {

    private IPaquete mPaquete;
    private IFilter mFilter;

    public MapperPasajePaqueteDelFilterAlPipe(IJoinPoint aJoinpoint) : base(aJoinpoint)
    {}

    public virtual IFilter Filter {
```

```
        get {
            // **Generado Automáticamente**
            if (mFilter == null) {
                if (this.Sender == null)
                {
                    // **Agregado Manualmente**
                    throw new ApplicationException("El sender no es un objeto");
                }
                mFilter = EntityMapper.GetMapper<IFilter>(new object[] { this.Sender });
            }
            return mFilter;
            }
        }

    public virtual SemanticWorld.IPaquete Paquete {
        get {
            // **Generado Automáticamente**
            if (mPaquete == null) {
                mPaquete = EntityMapper.GetMapper<IPaquete>(new object[] {
                                                            this.GetParameterValue(0) });
            }
            return mPaquete;
        }
    }
}
```

4.2.8.2 Escenario 2

```
[Entity(entityInterface="IFilter", mapper="MapperFilter")]
public abstract class Filter {

   [Property(entityInterface="IFilter", property="Nombre")]
    public string name;

    public Filter(string name)
    {
        this.name = name;
    }

    [method: Action1(actionInterface="IPasajePaqueteDelPipeAlFilter",
     mapper="MapperPasajePaqueteDelPipeAlFilter")]
    [method: Action2(actionInterface="IPasajePaqueteDelFilterAlPipe",
     mapper="MapperPasajePaqueteDelFilterAlPipe")]
    [method: Role1(actionInterface = "IPasajePaqueteDelFilterAlPipe", role="Filter",
     roleElement=RoleElement.Receiver)]
    [return: Role2(actionInterface="IPasajePaqueteDelFilterAlPipe", role="Paquete")]
    public abstract string ProcessStream
    (
        [Role(actionInterface="IPasajePaqueteDelPipeAlFilter", role="Paquete")]
        [Entity(entityInterface="IPaquete", mapper="MapperPaquete")]
        string stream
    );
}

public class PipeAndFilter {

    private Filter [] pipeLine;

    public void Add(Filter filter) {
        pipeLine.Add(filter);
    }

    public string Process(string stream) {
        Stream s = stream;
        foreach(Filter f in pipeLine)
            s = f.ProcessStream(s);

        return s;
    }
}

public class Filter1 : Filter {

    ProcessStream(string stream) {
        ...
```

```
    }
}

public class Filter2 : Filter {

    ProcessStream(string stream) {
        ...
    }
}
```

```
public class MapperPaquete : EntityMapper, IPaquete
{
    string dato;

    public string Dato {
        get {
            return this.dato;
        }
    }

    public override object[] Mappee
    {
        set { this.dato = (String)value[0]; }
    }

    public override bool IsMapperFor(object[] o)
    {
        String dato = o[0] as String;
        return (dato != null);
    }
}
```

```
public class MapperFilter : EntityMapper, IFilter
{

    private Filter mFilter;

    public virtual string Nombre {
        get {
            // **Generado Automáticamente**
            return mFilter.name;
        }
    }

    public virtual IPipe Entrada {
        get {
            // **Agregado Manualmente**
            return null;
        }
    }

    public virtual IPipe Salida {
        get {
            // **Agregado Manualmente**
            return null;
        }
    }

    public override object[] Mappee
    {
         // **Generado Automáticamente**
        set { this.mFilter = (pipeAndFilter.Filter)value[0]; }
    }

    public override bool IsMapperFor(object[] mappee)
    {
        // **Generado Automáticamente**
        pipeAndFilter.Filter filter = mappee[0] as pipeAndFilter.Filter;
        return (filter != null);
    }
}
```

En este mapeo puede verse que el filter no cuenta con los pipes de entrada y salida. Esto se debe a que no existen elementos en el código que implementen pipes, sino que

programáticamente se "introduce" el resultado de un filter en el siguiente. Se trata de un diseño simple en el que el código base se encuentra demasiado alejado en estructura de la descripción conceptual del dominio. Esto, junto a la falta de información que no sea la presente específicamente en el contexto de ejecución, impide que al abstraerse a la capa conceptual se permita una navegabilidad a través de los elementos de la arquitectura.

```
public class MapperPasajePaqueteDelPipeAlFilter : ActionMapper,
IPasajePaqueteDelPipeAlFilter
{
    IPipe mPipe;
    IPaquete mPaquete;

    public MapperPasajePaqueteDelPipeAlFilter(IJoinPoint aJoinpoint) :
            base(aJoinpoint) {
    }

    public virtual IPipe Pipe {
        get {
            // **Agregado Manualmente**
            if (mPipe == null)
            {
                mPipe = EntityMapper.GetMapper<IPipe>(new object[] { null,
                            EntityMapper.GetMapper<IFilter>(new object[]
{this.Receiver})});
            }
            return mPipe;
        }
    }

    public virtual IPaquete Paquete {
        get {
            // **Generado Automáticamente**
            if (mPaquete == null)
            {
                mPaquete = EntityMapper.GetMapper<IPaquete>(new object[] {
                                                        this.GetParameterValue(0) });
            }
            return mPaquete;
        }
    }
}
```

```
public class MapperPasajePaqueteDelFilterAlPipe : ActionMapper,
IPasajePaqueteDelFilterAlPipe
{
    IFilter mFilter;
    IPaquete mPaquete;

    public MapperPasajePaqueteDelFilterAlPipe(IJoinPoint aJoinpoint) :
            base(aJoinpoint) {
    }

    public virtual IFilter Filter {
        get {
            // **Generado Automáticamente**
            if (mFilter == null) {
                mFilter = EntityMapper.GetMapper<IFilter>(new object[] { this.Receiver });
            }
            return mFilter;
        }
    }

    public virtual IPaquete Paquete {
        get {
            // **Generado Automáticamente**
            if (mPaquete == null)
            {
                mPaquete = EntityMapper.GetMapper<IPaquete>(new object[] { this.ReturnValue
});
            }
            return mPaquete;
        }
    }
```

```
}
```

En este mapper se ve nuevamente la sensibilidad con respecto al uso del advice. Es necesario utilizar un *after advice* ya que la información requerida es obtenida como resultado de la ejecución del método asociado.

```
// **Agregado Manualmente**
public class MapperPipe : EntityMapper, IPipe
{
    IFilter entrada;
    IFilter salida;

    public IFilter Entrada {
        get {
            return this.entrada;
        }
    }

    public IFilter Salida
    {
        get {
            return this.salida;
        }
    }

    public override object[] Mappee
    {
        set {
            this.entrada = (IFilter)value[0];
            this.salida = (IFilter)value[1];
        }
    }

    public override bool IsMapperFor(object[] mappee)
    {
        IFilter entrada = mappee[0] as IFilter;
        IFilter salida = mappee[1] as IFilter;
        return (entrada != null && salida != null);
    }
}
```

4.2.9 Problemas encontrados en el caso de estudio

Nos interesaba mostrar también con este caso de estudio algunos problemas latentes que requieren mayor maduración y podrían ser contemplados en un trabajo futuro. Estos problemas se agrupan en 3 categorías:

1. Dificultad para acceder a la información desde los mappers: algunas estructuras de datos son modificadas al ser consultadas, lo cual altera el estado del sistema de maneras no contempladas en el código base si la consulta es realizada desde un mapper. Ejemplos de estas estructuras son: archivos que avanzan el puntero al leer un dato, pilas, etc. Este tipo de problemas no son propios de Setpoint, sino que otros enfoques, como el de AspectJ, también lo comparten. En ciertos casos, alternativas para evitar estos efectos sacrifican obliviousnes del lado del código base, como pudo verse en el escenario 1.

2. Necesidad de conservación del estado más allá del contexto: el estado del sistema es obtenido por los aspectos mediante una representación semántica del contexto. Este estado es limitado justamente al contexto del join point, no abarca la totalidad del sistema y es efímero en el sentido de que existe solo durante la ejecución del join point. Ante ciertas situaciones puede ser útil poder mantener el estado global del

sistema y poder consultarlo desde los aspectos. Esto podría requerir la utilización de mecanismos que permitan definir, actualizar y consultar este estado global.

3. Sensibilidad de mappers al tipo de advice (after, before): La forma en que se definen los mappers a partir del contexto se puede ver afectada por el momento en que se utilizan estos mappers. Si se trata de un before advice la forma en que se accede a la información puede ser notablemente diferente a si se trata de un after advice, debido a los cambios operados durante la ejecución del método implicado. Inclusive pueden darse situaciones más extremas: en alguna de estos dos advices la información necesaria para generar el contexto semántico puede no estar disponible.

5 Conclusiones

En esta tesis presentamos una alternativa para atacar el problema del advice fragility, basándonos en que la necesidad de desacoplar el código de aspectos del código base es crucial para facilitar la evolución de software. Una nueva versión de Setpoint fue desarrollada para lograr este propósito, en la cual se introdujeron nuevos formalismos para el modelado del dominio así como también mecanismos de mapeo sofisticado entre el mundo conceptual y el mundo del código base.

A continuación se enumeran las conclusiones resultantes del trabajo realizado:

- Se propuso una nueva representación del dominio, el cual es ahora modelado con objetos (específicamente, mediante interfaces). Este cambio elimina los problemas de expresividad y usabilidad que existían con OWL, los cuales obstruían el objetivo de acceder al contexto en forma semántica. El uso de interfaces permite representar fácilmente el nuevo concepto de acciones y entidades y provee un contrato semántico, el cual debe ser respetado por los mappers que implementen las interfaces. Además, las interfaces son directamente integrables al código de los mappers y de los aspectos, por ser parte del sistema de tipos de los lenguajes orientados a objetos. Esto facilita el mecanismo para vincular modelo y código.

- Los casos de estudio presentados en el informe ejemplifican la forma en que la solución propuesta logra desacoplar el mundo semántico del código base, ya que una única definición de la representación del dominio, de los pointcuts y de los advices es reutilizada por los distintos escenarios propuestos.

- La ampliación de LENDL fue necesaria para lograr el acceso al contexto semántico, de forma de poder identificar los elementos del contexto de ejecución que fueran requeridos. La nueva expresividad en la definición de pointcuts y advices permitió exportar roles de acciones en los pointcuts (derivados finalmente a los aspectos) en forma equivalente a la exportación "sintáctica" de elementos del contexto realizada por lenguajes como AspectJ.

- Al igual que en la solución original, nuestra propuesta sigue preservando las propiedades de obliviousness y quantification. Sin embargo, al incorporar el acceso semántico del contexto se encontraron inconvenientes que surgen por la sensibilidad de los mappers al tipo de advice (before, after) utilizado en la invocación del aspecto. Ejemplos de estas situaciones fueron examinadas en el caso de estudio sobre pipes and filters. Estos muestran ciertas dificultades para abstraerse del todo

del código base por parte del programador de aspectos (no alcanzando así un obliviousness completo). Será necesario buscar mecanismos que permitan mayor transparencia, tal vez reificando conceptos concernientes al momento de utilización de los mappers, que permitan definir a los mappers en función de ellos.

- Análisis de otras formas de representación del dominio: En este trabajo se cambió la representación de dominios, pasando de una representación mediante ontologías definidas en OWL a representaciones descriptas mediante orientación a objetos, más específicamente por medio de interfaces. A pesar del cambio en la representación en esta tesis se siguió en la línea de la tesis anterior en el sentido de utilizar un lenguaje general para describir todo dominio. Sería útil comparar la definición de dominios como se viene realizando hasta ahora contra una definición realizada a través de lenguajes de uso específico, especialmente diseñados para trabajar con dominios particulares.

- El acceso al contexto se limita a elementos que aparecen explícitamente en el contexto de ejecución del join point. Sería interesante poder predicar desde un punto de vista (arquitectura, dominio del problema, etc.) que permita contar con una vista completa del sistema y no solo la accesible desde el contexto de ejecución particular desde el que se quiere predicar. Esto podría requerir definir mecanismos para mantener el mapeo del código base a las vistas de manera permanente, en el sentido de que la información que se pueda obtener sobre las vistas sea accesible en todo momento desde cualquier join point y vaya actualizándose a medida que el sistema evoluciona en su ejecución. La idea, en definitiva, es ampliar el concepto de vista, brindando la complejidad necesaria que permita representar una ejecución abstracta del dominio de discurso en cuestión, corriendo en forma paralela y sincronizada a la ejecución del código base. Esta ejecución abstracta estaría definida en términos de estados y secuencias de acciones. En principio se puede modelar la ejecución mediante objetos, como se plantea en esta tesis, aunque otros formalismos pueden investigarse como medios de representación. Los aspectos estarían vinculados a esta ejecución de forma de poder observar su desarrollo y actuar en las situaciones pertinentes. La coordinación con la ejecución concreta se desarrollaría acoplándose al flujo de ejecución, como se hace actualmente. La sincronización entre ejecución abstracta y concreta no necesariamente debe ser en un sentido. Se pueden analizar mecanismos y aserciones que definan cómo y cuándo los aspectos pueden modificar el flujo y/o estado en la ejecución concreta al operar sobre la ejecución abstracta.

Anexo I: Preweaver 2.0 (implementado en Phoenix)

Qué es Phoenix?

Phoenix es un framework de Microsoft que permite construir compiladores y una amplia gama de herramientas para análisis de programas, optimización y testing. Fue diseñado para soportar técnicas avanzadas de compilación y análisis de programas. Provee una amplia variedad de bloques de construcción, implementados alrededor de una representación intermedia conocida por todos los bloques.

El framework de Phoenix es una infraestructura ampliada para compiladores y herramientas de programación. Phoenix soporta diversos lenguajes, y es fácilmente adaptable a distintos tipos de arquitecturas.

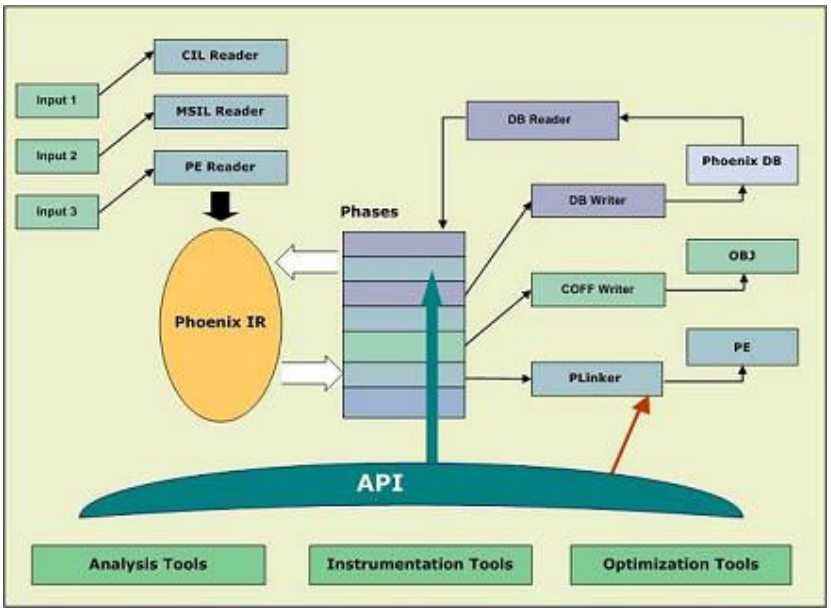


Figura 13 - Visión conceptual de la plataforma de Phoenix.

Todos los componentes de Phoenix interactúan utilizando una *Representación Intermedia* (IR), un *sistema de tipos* y un *sistema de símbolos* en común. Los componentes de entrada/salida manejan *C Intermediate Language* (CIL), archivos del tipo *Microsoft Portable Executable* (PE), *Microsoft Common Object File Format* (COFF), *Microsoft Intermediate Language* (MSIL), ensambladores, etc.

Fases

Phoenix organiza su funcionamiento en etapas de ejecución denominadas *fases*. Las fases son almacenadas como listas ordenadas y son ejecutadas en forma secuencial. Ciertas fases especiales, denominadas *hooks inter-fase,* son utilizadas para poder efectuar procesamientos previos y posteriores a la ejecución de una fase determinada.

La herramienta a ser implementada es responsable de crear la lista inicial de fases. La infraestructura, máquinas destinatarias, Plug-ins, etc. pueden alterar la lista de fases cuando sea necesario.

Las fases y los hooks se encuentran asociados a exactamente un *contenedor de fases*. El objeto contenedor sirve para varios propósitos. Contiene punteros a la lista de fases, a la lista de hooks pre-fase y a la de hooks post-fase. También tiene un puntero a la locación de memoria asociada al tiempo de vida (*lifetime*) del contenedor, de las fases, y de los hooks asociados a éste. Todas las fases y los hooks deben estar alocados desde la memoria asociada al lifetime del contenedor.
La herramienta a desarrollar debe crear un contenedor único para cada lista de fases y hooks que ésta desee ejecutar. Múltiples contenedores no pueden contener la misma fase; un único objeto de fase debe ser creado para cada una.

Representación Intermedia

El framework de Phoenix utiliza una –singular y fuertemente tipada- Representación Intermedia (IR) lineal para representar el *stream* de instrucciones de una función como una serie de flujos de datos de operaciones. Phoenix almacena la IR de una función como una lista de instrucciones doblemente enlazada. Cada instrucción posee un operador, una lista de operadores origen y una lista de operadores destino.

IR representa una función en múltiples niveles de abstracción, desde un muy alto nivel, independiente de la máquina (HIR), hasta un muy bajo nivel, dependiente de la máquina (LIR). IR representa explícitamente todo el flujo de control y de datos correspondiente al stream de una instrucción.

Jerarquía de unidades

La infraestructura de Phoenix crea unidades de compilación a medida que procesa las fases que el cliente decide ejecutar. Las unidades de compilación representan código con un conjunto de estructuras de datos interrelacionadas. Estas estructuras proveen una representación robusta que soporta distintos niveles de análisis.

La siguiente tabla provee definiciones de las unidades en la jerarquía de unidades IR.

Unidad IR	Descripción
FuncUnit	Encapsula la información requerida durante la compilación de una determinada función ó método.
DataUnit	Representa una colección de datos relacionados, como el conjunto de variables inicializadas o el resultado de la codificación de una FuncUnit.
ModuleUnit	Rpresenta una colección de funciones
PEModuleUnit	Representa un archivo compilado, que es una imagen PE, como un EXE o una DLL.
AssemblyUnit	Representa una unidad de compilación ensamblada de .NET
ProgramUnit	Representa una unidad de compilación ejecutable, un .exe o .dll
GlobalUnit	Representa la unidad de compilación más englobadora de todas.

Símbolos

Los símbolos de Phoenix denotan entidades como etiquetas, tipos, nombres de funciones, direcciones, entidades de metadatos, importaciones/exportaciones de módulos, y nombres para áreas de almacenamiento. Los símbolos son agrupados dentro de tablas y son clasificados mediante mapeos. Colectivamente, los símbolos proveen representación estructural de la relación entre entidades de programas –por ejemplo, mostrando que una función tiene determinados argumentos y variables locales en particular. Los símbolos son también un mecanismo para referenciar éstas entidades con la IR de Phoenix.

Implementación del Preweaver con Phoenix

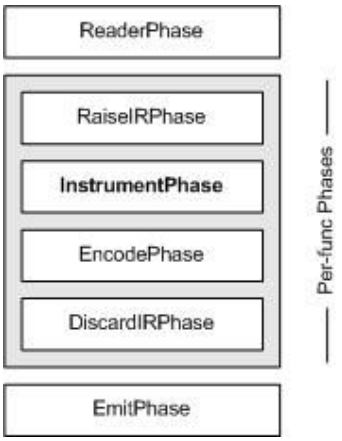
Las capacidades de Phoenix, en particular aquellas orientadas a trabajar con instrumentación de código a diferentes niveles de abstracción, resultaron apropiadas para el proceso de *preweaving*. En los siguientes párrafos explicaremos el uso de Phoenix para implementar el proceso de preweaving, que consiste básicamente en reemplazar cada instrucción de llamada a un método por la instanciación del join point y la posterior llamada al weaver.

Creación de un nuevo PEModuleUnit

El primer paso consiste en crear el módulo para almacenar la imagen del archivo a instrumentar. Esta imagen es almacenada en la forma de la Representación Intermedia de Phoenix (IR) y es luego utilizada para inyectar el código necesario con las llamadas al weaver.

Fases de ejecución del preweaver

Como fue explicado anteriormente, Phoenix organiza su funcionamiento en fases, almacenadas en forma de una lista ordenada. Dichas fases son ejecutadas secuencialmente, en el orden en que fueron presentadas en la lista. Debido a que cada herramienta es responsable de crear su lista inicial de fases, el siguiente paso consiste en definir las fases de ejecución del preweaver. La estructura de las fases del preweaver es presentada a continuación:



ReaderPhase. Esta fase es llevada a cabo por el componente de entrada/salida conocido como *PE Reader*, el cual se encarga de leer la imagen PE del programa ejecutable a instrumentar. Esta fase creará las DataUnits y FuncUnits de dicha imagen y las agregará al PEModuleUnit creado previamente. Las FuncUnits permanecerán en un estado

"comprimido" hasta el momento de ser utilizadas, momento en el cual serán expandidas a la representación IR.

La construcción de la representación IR de todo el programa, y de una sola vez, requiere una gran cantidad de memoria, difícilmente disponible en las computadoras actuales. Por ello Phoenix cuenta con una metodología de procesamiento que es llevada a cabo mediante *fases de función*. Este conjunto de fases procesa individualmente cada una de las unidades de función (*FuncUnits*) del *PEModuleUnit* de manera secuencial. El PE Reader expandirá entonces la representación IR de una función determinada recién en el momento en que ésta necesite ser procesada por las fases de función, y la misma será descartada una vez finalizado su procesamiento. De esta forma es posible lograr el procesamiento de todo el programa sin agotar la memoria del sistema.

A continuación se enumeran las fases de función que componen el preweaver:

- ***RaiseIRPhase.*** Es la primera fase del procesamiento por funciones. Esta fase crea la representación IR de la función a ser procesada. La construcción de la IR es tarea del PE Reader.
- ***InstrumentPhase.*** Es la fase encargada de inyectar las llamadas al weaver dentro de la función. Trabaja sobre la representación IR creada en la fase anterior
- ***EncodePhase.*** Esta fase codifica nuevamente -en forma binaria- las instrucciones de la función, a partir de su representación IR. Es la contraparte de la RaiseIRPhase.
- ***DicsardIRPhase.*** Es la última fase del procesamiento por funciones. Descarta la representación IR de la función en curso, liberando así la memoria utilizada.

EmitPhase. Una vez finalizado el procesamiento de todas las funciones del módulo, debe generarse el archivo ejecutable resultante de la instrumentación. Esta tarea específica es llevada a cabo por el componente de entrada/salida conocido como *PE Writer*. El PE Writer toma el *PEModuleUnit* y genera un archivo con formato *Microsoft Portable Executable* (PE).

Símbolos de Setpoint.dll

Con el propósito de inyectar las llamadas al weaver dentro del código a ser aspectualizado, el preweaver hace uso de los símbolos de Phoenix para referenciar ciertos elementos que se encuentran en Setpoint.dll -el assembly que representa el corazón de Setpoint- durante el proceso de instrumentación. Estos elementos son los siguientes:

- La clase que representa el *sender* y el *receiver* del join point
- La clase que representa el *mensaje* del join point
- La clase que representa el *join point* propiamente dicho
- El método de *weaving*

Proceso de instrumentación

El proceso de instrumentación es llevado a cabo durante la fase *IntrumentPhase*. En dicha fase se utiliza las estructuras de datos de Phoenix *Phx.IR.Instr* para iterar por todas las instrucciones de la función, reemplazando las instrucciones del tipo CALLVIRT, CALL y NEWOBJ por el código necesario para crear un join point y derivar al weaver la ejecución de dicha instrucción. Para lograr esto, Phoenix provee los elementos necesarios para

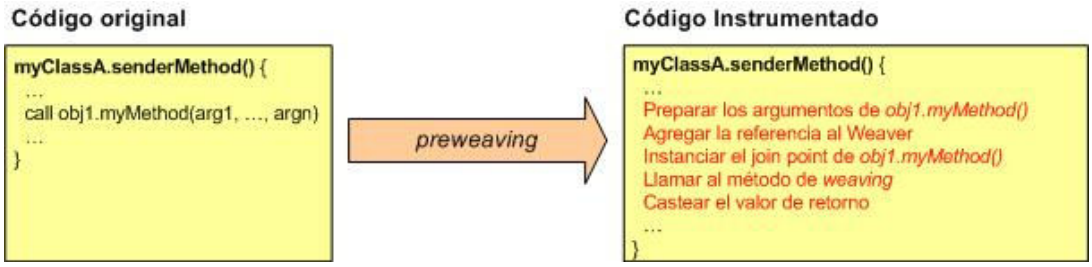
determinar información relevante como el método a ejecutarse, sus parámetros, los valores de retorno, el tipo de llamada (de clase ó de instancia), etc.

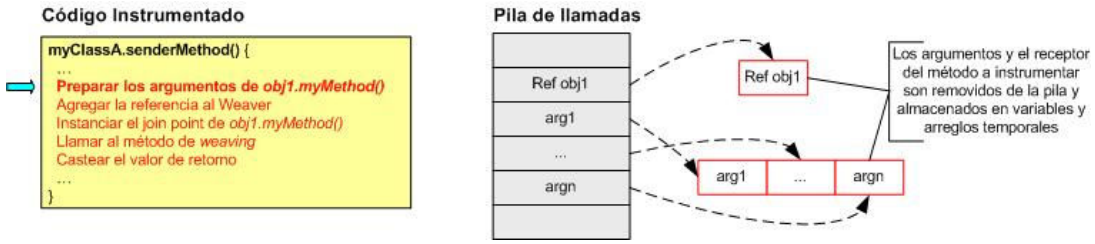
En los siguientes párrafos se describen los pasos del procedimiento de instrumentación, resaltando los elementos utilizados del framework de Phoenix.

```
Para cada ( Phx.IR.Instr instr dentro de la función a instrumentar)
      Si instr está entre
                  (Phx.Targets.Archs.MSIL.Opcode.callvirt,
                   Phx.Targets.Archs.MSIL.Opcode.call,
                   Phx.Targets.Archs.MSIL.Opcode.newobj)
            Entonces
            Inyectar las siguientes instrucciones:
            1. Preparar los argumentos de la llamada al método
            2. Agregar la referencia al weaver
            3. Instanciar el join point
            4. Agregar la llamada al método de weaving
            5. Castear el valor de retorno del método de weaving
      Fin Si
Fin Para
```

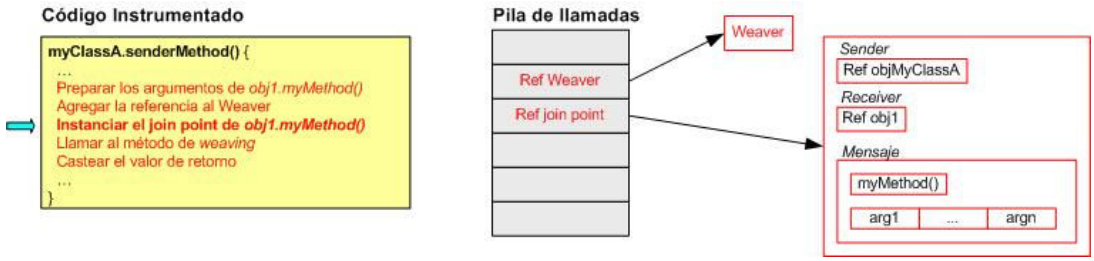
- *Preparar los argumentos de la llamada al método a interceptar.* Accediendo a los atributos de la clase *Phx.Types.FuncType*, una abstracción de Phoenix para representar los tipos de los métodos, se obtiene la cantidad de argumentos, los tipos de los argumentos, etc. Luego se utilizan las instrucciones correspondientes para tomar de la pila los valores de los argumentos, para volcarlos temporalmente en un *arreglo de argumentos*. Este arreglo es luego consultado en el siguiente paso.
- *Agregar la referencia al weaver.* El preweaver inyecta luego la instrucción MSIL *ldsfld* para agregar en la pila una referencia al objeto que representa al weaver, ya que al momento de efectuar la llamada al método de weaving es necesario que la referencia a este objeto se encuentre en el tope de la pila.
- *Instanciar el join point.* En este momento se inyecta la instrucción MSIL *newobj* para la creación de los objetos que representan el sender, receiver, el mensaje y el mismo join point de la llamada al método a ser interceptada por el weaver. Aquí es cuando los símbolos de setpoint.dll previamente mencionados (junto con el arreglo de argumentos) entran en juego, identificando todos estos elementos.
- *Agregar la llamada al método de weaving.* La llamada interceptada es reemplazada por la llamada al método de *weaving*. Para lograr esto, la instrucción MSIL *call* es inyectada en el lugar de la llamada original, la cual es removida de la lista de instrucciones.
- *Castear el valor de retorno del método de weaving.* Debido a la necesidad del weaver de mantener una generalización respecto al tipado del valor de retorno del método interceptado, su valor debe tratado con el tipo *Object* al ser encapsulado dentro del join point. Es por ello que luego de la ejecución del weaver, el valor retornado por el método interceptado (el cual en dicho momento se encuentra en la pila) debe ser casteado al tipo que poseía originalmente. La instrucción MSIL *cast* (ó *unbox*, para el caso de tipos primitivos) es utilizada con este propósito.

El siguiente es un ejemplo gráfico del proceso de instrumentación, mostrando la forma en la que llamada a un método es reemplazada por la llamada al weaver, contemplando el estado de la pila durante la ejecución:


Código original
myClassA.senderMethod() {
...
call obj1.myMethod(arg1, ..., argn)
...
}
preweaving
Código Instrumentado
myClassA.senderMethod() {
...
Preparar los argumentos de obj1.myMethod()
Agregar la referencia al Weaver
Instanciar el join point de obj1.myMethod()
Llamar al método de weaving
Castear el valor de retorno
...
}


Código Instrumentado
myClassA.senderMethod() {
...
Preparar los argumentos de obj1.myMethod()
Agregar la referencia al Weaver
Instanciar el join point de obj1.myMethod()
Llamar al método de weaving
Castear el valor de retorno
...
}
Pila de llamadas
Ref obj1
arg1
...
argn
Ref obj1
arg1 ... argn
Los argumentos y el receptor del método a instrumentar son removidos de la pila y almacenados en variables y arreglos temporales


Código Instrumentado
myClassA.senderMethod() {
...
Preparar los argumentos de obj1.myMethod()
Agregar la referencia al Weaver
Instanciar el join point de obj1.myMethod()
Llamar al método de weaving
Castear el valor de retorno
...
}
Pila de llamadas
Ref Weaver
Weaver
Ref obj1
arg1 ... argn


Código Instrumentado
myClassA.senderMethod() {
...
Preparar los argumentos de obj1.myMethod()
Agregar la referencia al Weaver
Instanciar el join point de obj1.myMethod()
Llamar al método de weaving
Castear el valor de retorno
...
}
Pila de llamadas
Ref Weaver
Ref join point
Weaver
Sender
Ref objMyClassA
Receiver
Ref obj1
Mensaje
myMethod()
arg1 ... argn

Anexo II: Gramática LENDL 2.0

El lenguaje LENDL permite definir los aspectos en forma declarativa y casi total, a excepción de la definición del código a ejecutar, que se expresa en lenguajes de programación .NET. A continuación se muestra la gramática de la nueva versión, desarrollada principalmente para incorporar expresividad sobre el contexto semántico. Esta gramática está definida mediante EBNF[TUCKER] para mayor claridad, aunque luego se utilizó una versión transformada de la misma para la implementación, debido a que esta primera definición más natural es recursiva a izquierda e impedía el procesamiento mediante la herramienta CodeWorker[CODEWORKER], que utiliza un algoritmo de parsing descendente.

LENDL → { *pointcut* | *advice* | *aspect* } *
pointcut → "pointcut" *identifier* "(" *parameters* ") :" *pointcut_statement* ";"
pointcut_statement → { *pointcut_statement* " and " *pointcut_statement*
| *pointcut_statement* " or " *pointcut_statement*
| "not" "(" *pointcut_statement* ")"
| "(" *pointcut_statement* ")"
| *cflow_predicate*
| *cflowbelow_predicate*
| *predicate* }
cflow_predicate → "cflow" "(" *identifier* ")"
cflowbelow_predicate → "cflowbelow" "(" *identifier* ")"
predicate → { *sender_statement* | *receiver_statement* | *message_statement* } *
sender_statement → "sender" "is" *type*
receiver_statement → "receiver" "is" *type*
message_statement → "message" "is" identifier { " && " *role_expression* } *
role_expression → { *role_expression* " and " *role_expression*
| *role_expression* " or " *role_expression*
| "not" "(" *role_expression* ")"
| "("*role_expression* ")"
| *terminal_expression* }
terminal_expression → "message." *identifier* " is " *identifier*
advice → "advice " *identifier* ":" *identifier* "{" { *trigger* } * "}"
trigger → "trigger " *identifier* "(" *arguments*")" *when* "{" *identifier* "(" *arguments* ")" {", " *identifier* "("*arguments* ")" }* "} ;"
when → { "before" | "after" }
aspect → "aspect " *identifier* { builtby *identifier* }$_{opt}$ "{" { *event* } * "}"
event → "event " *identifier* "(" *parameters* ") ";"
parameters → { *parameter* { "," *parameter* } * } *
parameter → *type identifier*
arguments → { *identifier* { "," *identifier* } * } *
type → { 'A'..'Z' } { 'a'..'z' | 'A'..'Z' | '_' | '0'..'9'}*
identifier → { 'a'..'z' | 'A'..'Z' | '_'} { 'a'..'z' | 'A'..'Z' | '_' | '0'..'9'}*

Bibliografía

[ASPECTJ] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm y W. Griswold. An Overview of AspectJ, ECOOP 2001

[CAZZOLA04] W. Cazzola et al. Evolving Pointcut Definition to Get Software Evolution. RAMSE' 04ECOOP' 04 Workshop on Reflection, AOP, and MetaData for Software Evolution. 2004.

[KELLENS06] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the Evolution of Aspect Oriented Software with Modelbased Pointcuts. To be published in ECOOP 2006.

[NAGY05] I. Nagy et al. Utilizing Design Information in Aspect Oriented Programming. Proc. of International Conference NetObjectDays, NODe2005. 2005.

[FILMAN00] Robert Filman and Daniel Friedman: Aspect-Oriented Programming is Quantification and Obliviousness.OOPSLA 2000.

[PARADOX] Tom Tourwé, Johan Brichau y Kris Gybels. On the existence of the AOSD-Evolution Paradox, Workshop on Software-engineering Properties of Languages for Aspect Technologies, AOSD 2003

[FRAGILITY] Christian Koppen and Maximilian Stoerzer. Pcdiff: Attacking the fragile pointcut problem. EIWAS 2004

[SETPOINT] Rubén Altman y Alan Cyment. Setpoint: Un enfoque semántico para la resolución de pointcuts en AOP. Tesis de Licenciatura FCEyN, UBA - Noviembre 2004.

[OWL] http://www.w3.org/TR/owl-features/

[DL] http://dl.kr.org/

[CLIFTON03] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral sub typing analogy. Technical Report TR0301a, Iowa State University. 2003.

[XPI] W.G. Griswold et al. Modular Software Design with Crosscutting Interfaces. IEEE Software, vol. 23, no. 1, pp. 5160, Jan/Feb. 2006.

[CODEWORKER] http://codeworker.free.fr/

[TUCKER] Allen Tucker y Robert Noonan. Programming Languages: Principles and paradigms, Ed. Mc.Graw Hill, 1ra edición, 2002.