Tesis de Licenciatura

“SetPoint: Un enfoque semántico para la resolución de pointcuts en AOP”

Alumnos : Rubén Altman - Alan Cyment

Director : Nicolás Kicillof

Fecha : Noviembre / 2004

Departamento de Computación

Facultad de Ciencias Exactas

Universidad de Buenos Aires

Abstract

Existe numerosa evidencia que muestra que la orientación a objetos no logra modularizar satisfactoriamente la existencia de *crosscutting concerns*: tracing, performance y persistencia representan ejemplos canónicos. AOP plantea su reificación en unidades llamadas *aspectos*. Dos nuevos conceptos que introduce esta disciplina son las nociones de *joinpoints* y *pointcuts*: los primeros son puntos en la ejecución de un programa; los segundos, conjuntos de éstos que permitirán luego especificar dónde corresponde aplicar un determinado aspecto. La gran mayoría de las implementaciones actuales obliga a definirlos en función de elementos sintácticos, lo que conlleva una serie de problemas de magnitud tal que hacen dudar sobre la utilidad real de las tecnologías AOP. La solución propuesta pasa entonces por explicitar los modelos semánticos antes ocultos detrás de convenciones sintácticas. Consideramos el uso de *anotaciones semánticas* estructuradas a través de *ontologías* para asignar significado a porciones del código, y la consecuente definición de los *pointcuts* a través de predicados sobre esos modelos. El objetivo de la presente tesis es entonces la construcción y experimentación con un framework que implemente el concepto de pointcut semántico, bautizado por nosotros como *setpoint*.

Agradecimientos

Antes de empezar con la tesis en si, y aunque el paso del tiempo y sucesivas entregas de los premios Martín Fierro hayan permeado el contenido emotivo de la frase, queremos agradecer a todos los que de una u otra manera hicieron posible que pudiéramos iniciar, continuar y concluir nuestros estudios universitarios con la entrega de este trabajo.

A nuestros padres, hermanos y hermanas, que nos acompañaron desde un primer momento tanto desde el país como desde el extranjero, permaneciendo constantemente a nuestro lado aunque el tiempo que les dedicáramos no haya sido el que realmente se merecieran.

A Caro y a Elvi, que pese a sufrir esto mismo y verse tantas veces conviviendo con parciales, finales y TPs, se mostraron cada día más interesadas en aprender de qué se trataba eso de la programación orientada a aspectos y qué la diferenciaba de los objetos.

A Nico., quien no sólo aceptó ser director de esta tesis, sino que a lo largo de los poco menos de dos años en los que trabajamos estuvo constantemente al lado nuestro, ayudando, empujando, alentando, enseñando y criticando, de acuerdo a lo que necesitáramos a cada momento.

A Noury Bouraqadi, Masashi Umezawa, Bruno Cabral, Diane Corney, Jeen Broekstra, Arjohn Kampman, Jeroen Frijters, Cedric Lemaire y al resto comunidad Open Source, sin quienes habría sido sencillamente imposible llevar a cabo la implementación.

A Sherman y a Mimí, sin cuya compañía se hubiera hecho muy difícil soportar noches a base de Speed con whisky para...¿Salir de parranda? ¿Arrasar en la noche porteña? Mmno exactamente.... pero terminar TPs a tiempo también paga.

A Antonio y José, quienes no sólo fueron muy importantes en nuestro desarrollo profesional, sino que nos brindaron su ayuda y se mostraron siempre bien predispuestos e interesados en colaborar con lo que necesitáramos.

A toda la cátedra de POO de la UBA, que nos permitió no sólo desarrollar el prototipo, sino también enriquecer el trabajo mediante constantes e interesantes discusiones.

A todos aquellos demás amigos, compañeros, profesores, conocidos, que nos ayudaron a que pudiéramos llegar a este momento: Víctor B., Diego G., todos nuestos amigos de las cursadas, y a tantos otros que nos tendieron en todo momento una mano, y que injustamente no estamos nombrando.

Índice

Introducción

Problemática enfrentada

La evolución de los lenguajes de programación ha estado marcada por una meta concreta: la invención de modelos que logren abstraer la complejidad inherente al problema a resolver. De los distintos paradigmas que han surgido, cada uno con sus fortalezas y debilidades, seguramente uno se distingue por su creciente utilización en cantidad y diversidad de proyectos: OO u Orientación a Objetos.

El paradigma de objetos, en su infinidad de variantes (más puristas, lenguajes imperativos con extensiones de objetos, etc.), han permitido modelar con gran facilidad muchísimos problemas complejos, pero tienen también sus falencias. Existen distintas propuestas de extensión de la técnica, denominadas de manera genérica POP ó Post-Object Programming / Programación Post-Objetos. Entre estos nuevos enfoques figura con gran empuje la Programación Orientada a Aspectos, o simplemente AOP, por sus siglas en inglés.

¿Qué objetivo persigue la aparición de los aspectos? Una meta primordial del arquitecto de software es aislar las *incumbencias* o *concerns* [PARNAS]. Es decir, construir unidades con cierto nivel de autonomía y encapsulamiento, que logren individualizar una dimensión distintiva del sistema que estamos modelando. Se ha visto que OO no logra capturar de manera feliz la existencia de *crosscutting concerns* o *incumbencias transversales*[1], es decir, aquellas que afectan a más de una clase u objeto [KIC97]. Tracing, performance y persistencia representan ejemplos canónicos de esto último. AOP plantea entonces la reificación de los *crosscutting concerns* en unidades llamadas *aspectos*. De esta manera se logra, afirman sus defensores, una verdadera *Separation of Concerns*, aun ante la existencia de *concerns* transversales.

Dos conceptos fundamentales que introduce la nueva disciplina son las nociones de *joinpoints* y *pointcuts*: los primeros son puntos en la ejecución de un programa; los segundos, conjuntos de éstos que permitirán luego especificar dónde corresponde aplicar un determinado aspecto. La gran mayoría de las herramientas existentes en la actualidad permite definir dichos conjuntos bien por enumeración o mediante predicados sobre la sintaxis del código. Los *joinpoints* son entonces caracterizados en función de elementos sintácticos. Deben suponerse *convenciones de notación,* que en definitiva no hacen más que dejar implícitos elementos semánticos comunes a todos los integrantes del conjunto. El uso de esta técnica conlleva una serie de problemas de magnitud tal que hacen dudar sobre la utilidad real de las tecnologías AOP [PARADOX]. Las herramientas más recientes permiten valerse también de anotaciones hechas en el código fuente, pero no hay en principio ninguna consideración acerca de cómo deberían utilizarse o como se necesitaría estructurarlas para que resultaran realmente útiles.

Nuestra visión se suma a las que plantean la explicitación de aquello que antes estaba oculto mediante convenciones de notación: el elemento semántico. Definimos entonces el concepto de **anotación semántica***:* una marca que permite asignar una característica particular a una porción de código. A fin de dotarlas de una carga de significado importante, consideramos pertinente el uso de ontologías [FALBO1] para la representación

[1] A lo largo del texto utilizaremos las expresiones *crosscutting concerns* e *incumbencias transversales* indistintamente.

de los modelos de conocimiento sobre los que basar esas anotaciones. Los *pointcuts* pasan a ser ahora predicados sobre esos modelos.

Objetivo

El objetivo de la presente tesis es aportar un grano de arena en torno a la discusión planteada sobre la real utilidad de las herramientas orientadas a aspectos, a través de la construcción de un framework que implemente el concepto de pointcut semántico, bautizado por nosotros como SetPoint. Esto no podría ser logrado si se lo plantea como un elemento aislado: está íntimamente ligado a otras cuestiones relacionadas con la plataforma AOP sobre la que se lo desarrolle. A lo largo del texto se explican y justifican los lineamientos generales considerados para los frameworks elaborados. Se construyeron dos versiones, una en Smalltalk y la otra en .NET, y se utilizaron luego para modificar las características no funcionales de sendas implementaciones del conocido juego del Senku [SENKU], desarrolladas originalmente sin estos nuevos requerimientos en mente. De cada uno de estos experimentos se extraen conclusiones sobre el uso de una plataforma AOP basada en pointcuts semánticos.

Organización de la tesis

El documento está organizado en una serie de capítulos que conforman el cuerpo principal de la tesis, seguidos de un conjunto de anexos destinados a dar un mayor nivel detalle sobre algunos temas específicos. Estos tienen explicaciones que pueden resultar de ayuda para el lector, pero que desviarían la atención hacia cuestiones accesorias de incluirse como parte del cuerpo principal. A continuación se describe brevemente el objetivo de cada capítulo:

1. Aspect Oriented Programming: Introducción a los conceptos principales y objetivos de la disciplina.
2. Algunas Herramientas: Descripción de las herramientas más relevantes del área.
3. Rumbo a SetPoint: Revisión y crítica de los pointcuts, discusión de temas relacionados y esbozo de una solución a través de pointcuts semánticos.
4. Volviendo Computable al Saber: Introducción a las web semántica, ontologías y modelos de representación de conocimiento.
5. AOP + Semántica - SetPoint!: Lineamientos sobre los que se construyeron las distintas versiones de SetPoint, y explicación de los experimentos realizados.
6. SetPoint v0.1 – El prototipo en Smalltalk: Arquitectura de la versión en Smalltalk, detalle del experimento efectuado y conclusiones.
7. "La base está" – SetPoint v1.0: Arquitectura de la versión .NET, y explicación de cómo se utilizó para aplicar aspectos sobre el Senku.
8. Conclusiones: ¡Adivinen cuál es el contenido de este capítulo!
9. Anexo I – El framework .NET: Introducción a las características más importantes de la plataforma, subrayando los elementos relevantes para la tesis.

10. Anexo II – Notación de los diagramas utilizados: Descripción de la notación utilizada para describir los diseños de detalle
11. Anexo III – Diseño del Senku: Diseño de detalle del Senku en sus versiones Smalltalk y .NET
12. Anexo IV – Diseño de detalle del framework: Diseño de detalle del framework desarrollado en el prototipo (a lo largo de la tesis se detallan sus diferencias con la versión final en .NET).
13. Anexo V – Ontología del Common Type System: Transcripción de dicha ontología.
14. Anexo VI – LENDL: Gramática del lenguaje LENDL.

Aspect Oriented Programming

Un médico, un ingeniero y un programador están charlando sobre cuál de sus profesiones es la más antigua. Empieza el medico:

- Bueno. La Biblia dice que Dios creó a Eva de una costilla de Adán. Esto obviamente requiere cirugía, y por lo tanto la medicina es la profesión más antigua.

El ingeniero replica:

- Es cierto, pero antes de eso La Biblia dice que Dios separó el orden del caos, y esa fue obviamente una obra de ingeniería.

El programador se echa para atrás en la silla y dice, sonriendo tranquilamente porque sabe que ha ganado la mano:

- Si, pero... ¿cómo creen que Dios creó el caos?

Introducción

El pan nuestro de cada día

Desarrollar software implica modelar de alguna forma los dominios que afecten al problema a resolver. No podemos realizar este análisis de cualquier manera: en algún momento debe ser posible ingresar el modelo que obtuvimos en una computadora, de forma tal que ésta pueda interpretar qué estamos queriendo decir.

Los ambientes de programación nos permiten realizar esta tarea. Cuando se nos exige escribir el modelo utilizando directivas similares a las que entiende el procesador decimos que estamos trabajando con un lenguaje de bajo nivel. Cuando por el contrario podemos usar expresiones o modelos de un mayor nivel de abstracción hablamos de lenguajes de alto nivel. A lo largo de la historia de la computación, se ha ido evolucionando en torno a reducir la brecha existente entre los lenguajes de programación y el idioma o los modelos utilizados cotidianamente por el hombre. Los diferentes paradigmas de programación dieron marcos para pensar las soluciones de forma cada vez mas independiente del código máquina, permitiendo concentrarse en la modelización y no en cómo escribir instrucciones adecuadas para un procesador, dejando dicha tarea para compiladores o interpretes especializados. Cada paradigma introdujo nuevas abstracciones: procedimientos, funciones, objetos, predicados. La industria informática está hoy dominada por la Orientación a Objetos (OOP), que define una pieza de software como un conjunto de objetos que colaboran entre sí para la resolución del problema atacado. Cada uno tiene una responsabilidad puntual, y se comunica con el resto a través del intercambio de mensajes [LIU].

Esta forma de encarar el desarrollo de software resultó sumamente beneficiosa y representó un salto de calidad en relación al antiguo rey, la programación imperativa. Los objetos permitían *encapsular datos y procedimientos* en cajas negras destinadas para tal fin, paso fundamental en pos de alcanzar el objetivo atribuido a dos publicaciones míticas de la ingeniería del software: *separation of concerns*, o separación de incumbencias[2] [PARNAS][DIJK]. Al modelar un problema existen numerosas dimensiones a tomar en cuenta, y el plantear cada una de éstas de forma independiente nos permite mejorar la calidad del software construido.

[2] A lo largo del texto utilizaremos las expresiones *separation of concerns* y *separación de incumbencias* indistintamente.

Escenas de la vida real

Supongamos ahora el siguiente ejemplo, trabajado dentro del paradigma de orientación a objetos. Modelamos como parte de nuestro dominio una clase JugadorDeFutbol, una Técnico y una Arbitro. Cualquiera de las dos primeras puede enviar un mensaje a la tercera insultándola, aunque ésta luego haga valer su autoridad mostrando una tarjeta amarilla. Ahora bien, Joseph Blatter, al contratarnos para hacer el sistema, nos sorprendió señalándonos que necesitaba persistir un registro histórico de estos mensajes. Debemos escribir entonces el código para tal fin. La forma más rápida probablemente sea el insertar los llamados directamente en los métodos de las clases JugadorDeFutbol y Técnico. El código resultante en la clase JugadorDeFutbol será por ejemplo el siguiente[3]:

```
JugadorDeFutbol>>insultarA: unArbitro
        unArbitro oir: self insulto.
        self log: self insulto.
```

Línea de código agregada para el logging

Mr. Blatter nos exige, unos meses después de comenzado el proyecto, agregar este mismo comportamiento también a los dirigentes, quienes de aquí en más no podrán seguir gritando impunemente desde la tribuna. Procedemos del mismo modo... Debemos modificar necesariamente el código de la clase Dirigente para lograr este objetivo[4]. Unos días después nos damos cuenta de que debemos agregar un parámetro en el mensaje de logueo en la base de datos. Tenemos que modificar entonces al jugador, al técnico y al dirigente...

Code Tangling y Code Scattering

El caso anterior nos da una pista de los principales problemas que apuntan a solucionar las nuevas técnicas conocidas como *Post Object Oriented*, entre ellas el Desarrollo de Software Orientado a Aspectos (AOSD). Hay dos *concerns* independientes que estamos combinando en las mismas abstracciones: el problema funcional y el problema del logueo. Cuando diseñamos la solución, modelamos naturalmente una jerarquía de clases orientada a representar el dominio funcional en cuestión, en este caso el fútbol. Existen sin embargo una serie de incumbencias transversales a dicho modelo, como lo es el logging, que obligan a que las clases dedicadas a resolver cuestiones funcionales tengan que preocuparse también por cómo almacenar un mensaje en una base de datos. Estos son definidos por [KIC97] como *crosscutting concerns*. Los ejemplos canónicos son requerimientos no funcionales típicos como seguridad, transaccionabilidad, performance y logging.

El ejemplo anterior puede generalizarse a los paradigmas lógico, imperativo y funcional. El elemento distintivo es que todos permiten una única descomposición del problema, generalmente la funcional[5]. La

[3] El código de ejemplo está escrito en Smalltalk

[4] Ojalá fuese tan fácil modificar a nuestra clase dirigente!

[5] Algunos autores argumentan en contra de esta "tiranía de la descomposición dominante" [HYPERJ], señalándola como causa principal de que aún no se haya podido alcanzar una real separación de incumbencias.

implementación de crosscutting concerns provoca entonces dos características no deseables en el código fuente [KIC97]:

- Code Scattering: El código correspondiente a una incumbencia, el logging en el ejemplo, no es encapsulado en un único módulo, sino que aparece disperso por el programa.
- Code Tangling: Dentro de un mismo módulo encontramos código correspondiente a distintos concerns. En el ejemplo encontramos código que atañe al comportamiento funcional y código que corresponde al logging.

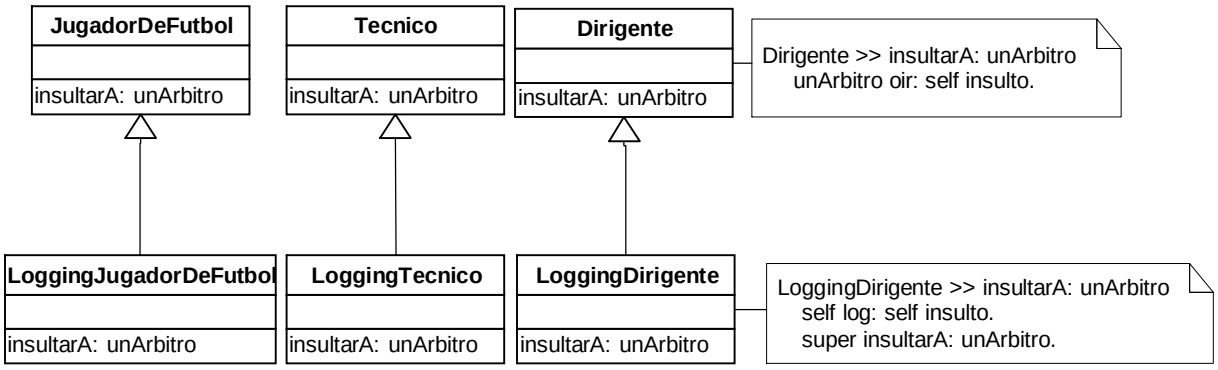
Estas dos propiedades originan código más complejo y difícil de mantener, con todas las desventajas que esto acarrea.

Profundizando los ejemplos

En el apartado anterior planteábamos a partir de un ejemplo simple los problemas de code tangling y scattering. Puede argumentarse que es un caso modelado ad-hoc, que no presenta una solución bien diseñada al problema propuesto. A continuación plantearemos entonces muy brevemente vías de mejorar esta implementación, basadas en el análisis hecho en [DEDECKER], para confirmar la existencia del problema mas allá del ejemplo.

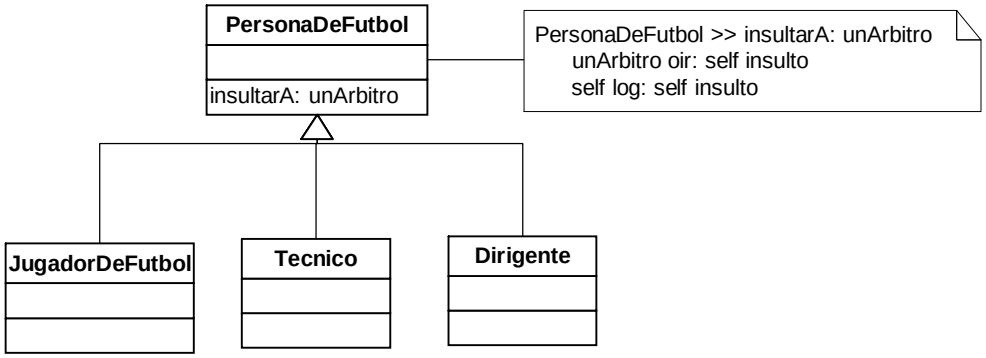
Especialización

Una primera posibilidad es especializar las subclases JugadorDeFutbol, Tecnico y Dirigente en subclases que agreguen el comportamiento de logging:



Este diseño no resuelve el problema del code scattering. Por otro lado, el problema de la explosión de clases [GAMMA] surge si se necesita incluir más concerns.

Otra posibilidad es generar una superclase común:

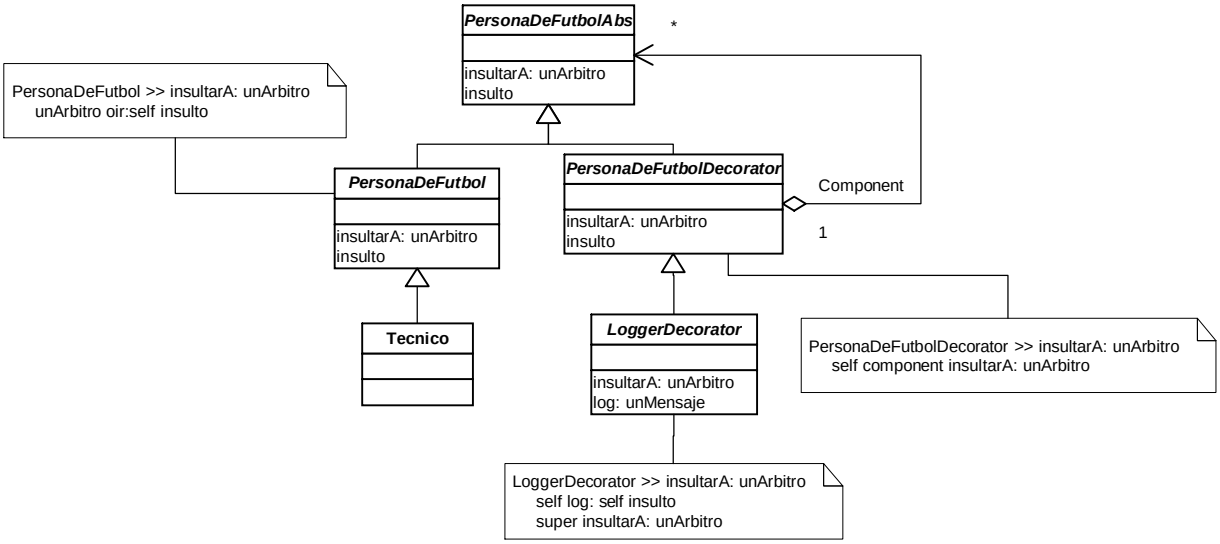


Nuevamente, el problema del code tangling permanece sin resolver. Por su parte, si tuviéramos que loguear más objetos además de las instancias de estas "personas de fútbol", reaparecería también el scattering.

Patrones de diseño

1. Decorador

El modelado del problema siguiendo la estructura de decoradores propuesta en [GAMMA] es similar a la siguiente[6]:



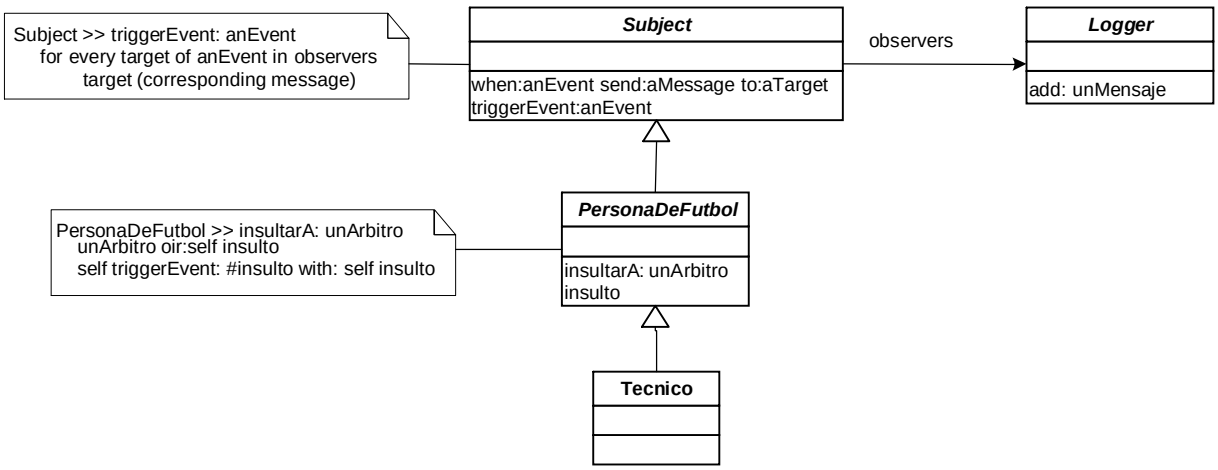
La clase PersonaDeFutbolAbs representa el protocolo que respeta toda persona de fútbol. Puede implementarse una jerarquía de decoradores que lo satisfaga y que agregue funcionalidad al método original, para luego delegar su ejecución al mismo. Para ello deben componerse los objetos correspondientes.

[6] No es nuestra intención dar el detalle del diseño de la solución, sino brindar simplemente un esbozo que permita criticarla. Para un mayor detalle acerca de sus ventajas y desventajas, referirse a [GAMMA]

Estamos mejorando el problema del tangling en relación al modelo de especialización recién planteado, al encapsular un concern en un decorador particular, pero luego será el proceso de construcción del objeto el que deberá ser conciente de la necesidad de aplicación de este patrón. Por otro lado, se debe especializar un decorador para cada jerarquía en donde quiera aplicarse el concern, reapareciendo así el problema del scattering y la explosión de clases. Esto último podría evitarse en algunos ambientes a través de mecanismos generales de reenvío de mensajes, como ser el forwarding mediante el doesNotUnderstand en Smalltalk, pero esto acarrea otra cantidad de desventajas [ALPERT].

2. Observer

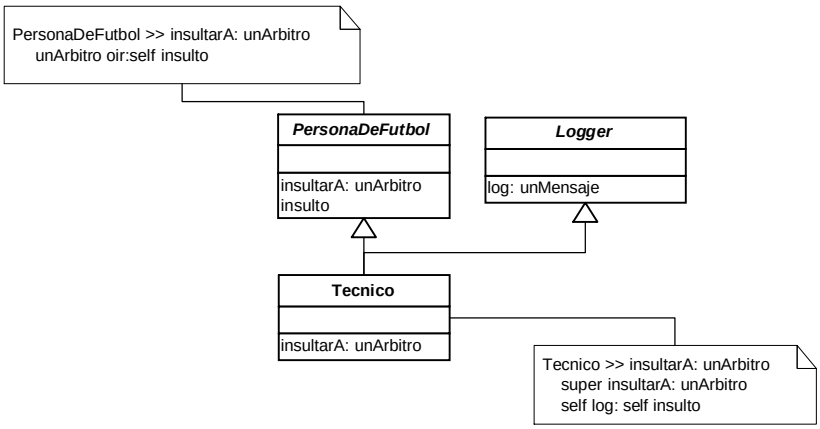
El boceto de la solución siguiendo el patrón (adaptado a una versión simil smalltalk) es el siguiente:



En este caso, la notificación del evento a loguear, pese a ser más desacoplada del código de logging que una llamada explícita, permanece inserta en el código. Por otro lado, si se quiere que un concern afecte a eventos no publicados antes, deberá modificarse el código funcional. Por último, encontramos una nueva forma de scattering cuando el objeto que ejecutará el logging necesita subscribirse a los eventos publicados por un objeto.

Mixins

Algunos ambientes brindan la posibilidad de herencia múltiple: una clase puede tener más de una superclase. Una posibilidad para alcanzar un mejor *separation of concerns* entre el dominio funcional y otros ortogonales es, entonces, hacer que cualquier clase perteneciente a dicho dominio herede de otra que corresponda a una perspectiva distinta. La clase Técnico presentada a continuación es el reflejo de esta idea:



Pese a alcanzarse una buena modularización de los distintos *concerns*, el envío de los mensajes continúa mezclado con el código funcional. Si se necesita implementar otros requerimientos no funcionales además del logging, el problema de acrecienta. El *tangling* y el *scattering* aún sobreviven. La herencia multiple acarrea por otro lado una serie de problemas que exceden su aplicación en este escenario.

Metaprogramación

Algunos ambientes actuales permiten *metaprogramar*, es decir, modificar código a nivel metaclases para alterar el comportamiento del programa. El uso de un framework de metaprogramación sin ninguna extensión que posibilite un real *separation of concerns* lleva a los mismos problemas de scattering y tangling que los mencionados antes. En vez de distribuir el código de logging en las clases, lo haremos a nivel metaclase.

Más adelante veremos sin embargo como el uso de un ambiente que brinde estas características puede extenderse para alcanzar las propiedades buscadas.

Bueno, pero...¿qué es AOP?

AOP puede enorgullecerse hoy en día de poseer herramientas, papers, conferencias, defensores y enemigos. Pero paradójicamente no existe consenso en cuánto a qué es exactamente lo que queremos decir con la bendita sigla. AOSD.net, el portal reconocido por la comunidad, incluye AOP dentro de un conjunto de "*técnicas que permiten modularizar los aspectos transversales de un sistema*". La definición es lo suficientemente amplia para dar lugar a diversas interpretaciones. ¿Qué requisitos debe cumplir una herramienta para ser considerada orientada a aspectos? En los apartados siguientes intentaremos resumir los distintos elementos que componen la técnica en cuestión, adoptando como marco de referencia el planteo surgido en [ASPECTJ], por ser el enfoque más difundido al momento de comenzar con este trabajo. Intentaremos enriquecerlo con aportes provenientes de otras vertientes.

¿Hay una dimensión más dimensión que las otras?

Según la definición anterior, el objetivo que se tiene entre manos es lograr la modularización de los así llamados *crosscutting concerns.* Este planteo lleva a pensar en la existencia de una dimensión base del problema y otras que la cruzan, alterando su comportamiento. En este marco, el movimiento AOP, desde la perspectiva antes mencionada, propone su unidad de modularización: el *aspecto*, a través del cual se encapsularán dichos concerns transversales. Existe consenso general sobre la asociación de aspectos a propiedades no funcionales: logging, seguridad, performance, persistencia, transaccionabilidad, resultan ejemplos canónicos de esta tecnología; hay otros enfoques, sin embargo, que plantean ciertos elementos de la descomposición funcional como pasibles de ser modularizados en aspectos [DHONDT]; se aprovecha en este caso la nueva unidad de modularización para resolver problemas de diseño.

Tal vez suene algo arbitraria la asimetría planteada entre la dimensión funcional y los aspectos, y de hecho... lo es. Incluso antes de la aparición de AOP se considera que un sistema puede ser mirado desde distintas perspectivas (ver [KRUCHTEN] o [PIE], por ejemplo). Algunos autores vieron en AOP la posibilidad de plasmar esto en una implementación, desarrollando una visión simétrica de las distintas dimensiones.

No importa si se prefiere el enfoque simétrico o asimétrico de los aspectos, ambos persiguen un mismo objetivo en pos de una real separación de incumbencias: el programador de una dimensión del problema debe ser **oblivious** (olvidadizo/indiferente) con respecto a la injerencia de las demás; mientras menos distracciones tenga sobre la perspectiva que lo atañe, mejor [FILMAN].

Modelo de joinpoints

Hablamos de la aplicación de aspectos sobre código base, pero... ¿cuándo es que se aplican? Cada plataforma de aspectos define ciertos puntos en la ejecución de un programa en los que pueden entremezclarse aspectos y código base. Estos puntos se denominan **joinpoints**. Algunos ejemplos son las llamadas a métodos, el acceso a atributos de un objeto o el lanzamiento de una excepción.

Weaving

Más allá de los diferentes aspectos que afecten a un programa, el sistema que queremos obtener es uno solo: hay que definir una relación entre los aspectos y el código base. Será necesario establecer algún tipo de “pegamento” entre ellos. En la programación tradicional será el simple código: se codificarán los envíos de mensaje necesarios, cual lo hecho en el ejemplo. Pero en el caso de la orientación a aspectos, el objetivo será desacoplar al máximo al código base de los aspectos. El proceso por el cual estos se combinan se denomina *weaving* (tejido) y lo realizará, idealmente de manera invisible para el programador, un *weaver* (¡qué otro nombre podía tener!). En general, se organiza a los mismos en dos grandes grupos, definidos según el momento en el que introducen los mecanismos para decidir sobre la aplicación de aspectos:

- Estáticos: Compiladores y preprocesadores que generan un único ejecutable con el código de los aspectos.

- Dinámicos: Se valen de distintos eventos que suceden durante la ejecución de un programa para incorporar los mecanismos que decidan la aplicación de aspectos. Consisten en general de frameworks que aprovechan capacidades de metaprogramación del entorno.

Pointcuts y Advices

El modelo anterior permanece aún incompleto: ¿cómo decimos en qué joinpoint aplicar qué aspecto? Se denomina **quantification** a la manera que tenemos de predicar sobre los joinpoints para definir su pertenencia o no a un conjunto, para poder definir si corresponde aplicar algún aspecto. Se denomina *pointcut* a un conjunto definido de joinpoints. Un ejemplo podría ser:

```
Pointcut mensajesAPersonas =
{
        todos los joinpoints que correspondan a llamadas a métodos de instancias de la clase
        Persona o sus subclases
}
```

Las características de cuantificación que brinda la plataforma son un elemento fundamental para su evaluación como herramienta AOP; cuanto mayor poder expresivo, más desacoplados serán los aspectos del código base. Un predicado puede hablar sobre condiciones estáticas (el nombre de un método) o sobre condiciones dinámicas (que un mensaje haya sido invocado por una instancia de la clase Perro).

Se denomina *advice* a la entidad que indica qué aspecto corresponde aplicar en todos los joinpoints de un pointcut.

Elementos de caracterización

A partir de los conceptos introducidos en el apartado anterior, y basándonos fuertemente en las ideas que aparecen en [FILMAN][7], identificamos los siguientes elementos para caracterizar las diferentes herramientas AOP:

- **Modelo de descomposición del dominio:** ¿Son tratadas las diferentes dimensiones del problema de forma simétrica o asimétrica? ¿Existe una descomposición dominante sobre el resto?
- ***Obliviousness*:** ¿Puede el programador del código base ignorar por completo los aspectos que afectarán su código? Si no, ¿con qué profundidad debería conocerlos?
- ***Quantification*:** ¿Cómo se definen los conjuntos de *joinpoints* sobre los que se aplicará un aspecto?¿Están dentro del código del aspecto en sí?¿Es posible definirlos por comprensión? ¿O por

[7] El paper, además de titularse *AOP is Quantification and Obliviousness*, define AOP como: “the desire to make quantified statements about the behavior of a program, and to have these quantifications hold over programs written by oblivious programmers”…

extensión? En este caso, ¿se realizan anotaciones en el código mismo, se elabora una lista o se utiliza una herramienta propia del entorno de desarrollo para detallarlos?

Además de las características de base recién mencionadas, otras que podríamos denominar implementativas hacen a la usabilidad del framework AOP utilizado:

- **Modelo de *joinpoints*:** ¿En qué puntos de la ejecución de nuestro programa podemos aplicar aspectos? ¿Podemos indicar condiciones dinámicas sobre esta definición (por ejemplo, "antes del método A, sólo cuando es llamado por el método B")?
- **Contexto:** ¿Qué conocimiento tiene un aspecto del entorno en el que se ejecuta? ¿Exigirá algún tipo de contrato?¿Se ejecuta de forma totalmente aislada? ¿Se extiende el lenguaje con construcciones destinadas especialmente para este fin?
- ***Introduction***: ¿Es posible modificar la estructura interna de alguna clase a la cual afecta el aspecto? ¿Puedo agregar por ejemplo un método a una clase?
- ***Weaving*:** ¿Cómo se combinará el código de los aspectos y el de la dimensión base? ¿El proceso de *weaving* se hace en tiempo de compilación, en *runtime* o en alguna instancia intermedia? ¿Se inyectará código, se crearán *wrappers*, se usará algún mecanismo de intercepción...?
- **Relaciones entre aspectos:** ¿Es posible definir el orden de ejecución de diferentes aspectos que se aplican sobre un mismo *joinpoint*? ¿Existen relaciones de exclusión mutua, o algún otro tipo de restricción?
- **IDE:** ¿Qué tan integrado está el *framework* de aspectos con el entorno de programación estándar? ¿Qué posibilidades tiene el programador de conocer el impacto de los aspectos sobre su código? ¿Qué tan sencillo resulta refactorizar una aplicación para el uso de aspectos?

Resumen

AOP se presenta como un conjunto de tecnologías orientadas a mejorar la modularización de los diferentes dominios que aparecen entremezclados en la solución a un problema de software. Hay opiniones encontradas acerca de su peso relativo: mientras que algunos consideran la existencia de una dimensión base y de aspectos que la afectan, otros plantean su simetría. Las diferentes dimensiones se combinan a través de un proceso llamado *weaving* en los denominados *joinpoints*, puntos bien definidos en la ejecución de un programa. Se llama *pointcut* a un conjunto bien definido de joinpoints, y *advice* a la relación entre un pointcut y un aspecto a aplicar. La capacidad de expresión de los predicados que definen dichos conjuntos es conocida como *quantification*, y resulta un elemento fundamental en AOP, así también como el nivel de ignorancia (*obliviousness*) que pueda mantener el programador de una dimensión en relación a la existencia de las otras.

Algunas Herramientas

Introducción

Habiendo superado sus primeros años de vida, el movimiento AOP dio lugar a la proliferación de una gran cantidad de herramientas que implementan las ideas de *aspectos* y *separation of concerns*. En el site oficial de la disciplina[8] puede encontrarse un listado exhaustivo tanto de las plataformas comerciales como de las experimentales aparecidas al día de la fecha. El objetivo de este capítulo es describir las que a nosotros nos resultaron más significativas, ya sea por su importancia dentro de la disciplina o por su relevancia para la presente tesis. El análisis de las mismas estará por ello orientado a nuestra área de interés, que es la definición de pointcuts.

AspectJ: la cuna de la civilización

Fuentes: [ASPECTJ], [ASPECTJ2], [ASPECTJ3], [ASPECTJ4]

Desarrollada en los laboratorios del Xerox Palo Alto Research Center por un grupo liderado por Gregor Kiczales, esta herramienta es sin dudas la plataforma AOP más difundida en la actualidad. Cuenta con una activa comunidad de usuarios, tanto en el ámbito académico como en el empresarial. Su simbiosis con la disciplina es tal que cuesta distinguir entre el marco teórico general y los conceptos introducidos y apañados por la herramienta. Por citar un ejemplo más que cercano, en el capítulo anterior explicamos los conceptos de AOP basándonos en su enfoque.

En [ASPECTJ] se definen los conceptos anteriormente introducidos[9]:

- Joinpoint: Punto bien definido en la ejecución de un programa.
- Pointcut: Conjunto de joinpoints.
- Advice: Construcción similar a un método que permite definir comportamiento adicional a ejecutar eventualmente ante un joinpoint[10].
- Aspect: Unidad para la modularización de incumbencias transversales, compuesta de pointcuts, advices y construcciones Java tradicionales[11].

[8] Ver http://www.aosd.net/technology/index.php

[9] En la sección anterior nos basamos en el enfoque de AspectJ pero adaptamos en algunos casos la versión original a un enfoque más genérico. En este apartado transcribimos las definiciones dadas por Kiczales et al en el texto citado.

[10] En nuestro enfoque habíamos definido al advice como la relación entre el pointcut y el aspecto a aplicar, y no la implementación del comportamiento a aplicar en sí.

[11] En la definición dada por nosotros, un aspecto equivale a una incumbencia transversal, y no a la construcción del lenguaje propuesta en este caso

Se introdujeron palabras clave en el lenguaje para permitir la definición de estos nuevos conceptos.

AspectJ plantea un enfoque asimétrico de las dimensiones de un sistema de software: se cuenta con un código base, que puede verse afectado por la aplicación de *aspectos*. Según sus creadores, en principio uno tiende a pensar que éstos se corresponderán siempre con las propiedades sistémicas de la aplicación, pero a medida que avanza en la comprensión de la tecnología descubre que hay elementos de la jerarquía funcional que también deben ser modularizados en aspectos: patrones de diseño y personalizaciones de procesos son algunos ejemplos [ASPECTJ4].

El weaver consiste básicamente de un compilador Java modificado que genera el bytecode correspondiente a partir de código fuente, entremezclando código base y aspectos en los joinpoints en donde eventualmente se podría modificar el comportamiento de la aplicación. En las versiones más recientes se permite también incoporar aspectos a programas base en donde no se tenga el código fuente, trabajándose directamente sobre su bytecode.

Además de permitir modificar el comportamiento de un programa en los joinpoints, AspectJ permite agregar operaciones a clases e interfaces: este concepto fue bautizado por los autores como *crosscutting estático*.

Modelo de joinpoints y *pointcut designators*

En la siguiente tabla se enumeran todos los tipos de joinpoints definidos por AspectJ:

Nombre
Method Call
Method Execution
Constructor Call
Constructor Execution
Static initializer execution
Object pre-initialization
Object initialization
Field reference
Field Set
Handler Execution
Advice Execution

Los pointcuts son definidos a través de una nueva construcción denominada *pointcut designator.* Un posible ejemplo aparece en [ASPECTJ2]:

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

Llamadas a una instancia de Point

Llamadas a los métodos setX o setY con la signatura especificada

Identificador del pointcut

El pointcut designator "setter" define el conjunto de joinpoints formado por las llamadas a los métodos "void setX(int)" y "void setY(int)" de instancias de la clase "Point"

Existe una cantidad de primitivas destinada a identificar el tipo de joinpoint que deseamos incluir en el conjunto (*call* y *target*, en el ejemplo). Luego pueden combinarse para formar expresiones más complejas, en este caso mediante los operadores de disyunción y conjunción. También es posible predicar sobre condiciones dinámicas de la aplicación: contamos para ello con keywords como *cflow* o *if*, que permiten predicar sobre el flujo de ejecución o sobre expresiones booleanas relacionados con elementos del contexto, respectivamente.

Wildcarding

AspectJ admite el uso de *wildcards* en la definición de pointcuts. Suponiendo que se quisiera por ejemplo interceptar todas las llamadas a métodos de la clase Point cuyo nombre comience con “set”, sin importar ni su tipo de retorno ni sus parámetros, podría generalizarse el ejemplo anterior sin necesidad de enumerar cada método (mejorando las capacidad de *quantification* del framework) de la siguiente manera:

```
pointcut setter(): target(Point) &&
                   call(* set*(..));
```

Para cualquier argumento

Todos los métodos cuyo nombre empiece con *set*

Advices y Aspectos

El advice es en AspectJ la construcción sintáctica que define el comportamiento a insertar antes, después o en vez de la ejecución de un joinpoint. En el siguiente ejemplo se determina mostrar un mensaje en pantalla antes de la ejecución de los joinpoints integrantes del pointcut *move*:

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

AspectJ brinda además una serie de mecanismos que posibilitan el acceso y la eventual manipulación del contexto: acceso a los argumentos y al valor de retorno de métodos mediante la definición de parámetros en pointcuts y advices, al joinpoint en ejecución por mecanismos de *reflection* y posibilidad de decidir acerca de la ejecución del joinpoint mediante la palabra clave *proceed*.

El aspecto es, finalmente, la unidad de modularización que agrupa pointcuts y advices. El siguiente ejemplo, ligeramente modificado, aparece en [ASPECTJ]:

```
aspect MoveTracking {
      static boolean flag = false;
      pointcut moves():
                  target(Point) &&
                  (call(void moveX(int)) ||
                   call(void moveY(int)));
      after(): moves(){
            flag = true;
      }
 }
```

HyperJ: mirando el mundo con otro cristal

Fuentes: [HYPERJ], [HYPERJ1], [HYPERJ2], [HYPERJ3]

Los ideólogos de HyperJ adhieren a una visión simétrica de AOP[12], según presentamos en el capítulo anterior. El objetivo es permitir modularizar el software de acuerdo a los diferentes enfoques que puedan utilizarse para descomponerlo (*dimensions of concern*). A veces conviene referirse a la división en clases, otra a los patrones de diseño utilizados, otra a los distintos *features* funcionales que se incluyan. Ninguna descomposición es mejor o peor que la otra. Cada una tiene características que la hacen útil en algún momento del ciclo de vida. Lo importante es que todas puedan coexistir, sabiendo que es difícil que todas sean realmente ortogonales e independientes: probablemente en algunos casos se solapen y en otros deban interactuar para obtener como resultado el sistema finalmente buscado.

1,2,3... al hiperespacio!

Ok, nos convencieron... ¡Derroquemos a la descomposición dominante! ¡Los distintos concerns deben ser libres e iguales! ¿Cómo hacemos? Los autores de la herramienta Hyper/J tienen su propuesta: el enfoque de los hiperespacios.

Los distintos concerns son organizados en *hyperslices*: conjunto de módulos convencionales escritos mediante cualquier formalismo, que incluyen únicamente aquellas unidades relacionadas con un *concern* específico. Una unidad es una construcción sintáctica cualquiera de un lenguaje de desarrollo, como puede un método, una clase o una interfaz. Cada unidad puede aparecer en más de un hyperslice. La condición para que cada tajada esté correctamente formada es que sea *declarativamente completa*: debe definir todos aquellos elementos que utilice, aunque más no sea su interfaz. Esta condición hace que ningún concern dependa de una implementación particular de otro.

La composición de hyperslices dará como resultado el sistema buscado. Ahora bien, ¿cómo hacemos para integrarlos? Contamos para ello con los *hipermódulos*: conjunto de hyperslices unidos por una regla de composición. Pueden verse también como slices compuestos, ya que debe ser posible utilizarlos indistintamente.

De vuelta a la tierra

Hyper/J, plataforma desarrollada como prueba de concepto de las ideas anteriores, fue implementada a fines de los noventa en los laboratorios de IBM. Como su nombre lo indica, está montada sobre Java. Los pasos necesarios para la generación de un programa son los siguientes:

[12] El nombre con el que se conoce su enfoque es MSDOC, *Multidimensional Separation of Concerns*.

- Especificación de las unidades a utilizar: Soporta únicamente el manejo de Java class files como unidades. Puede especificarse archivo por archivo o pueden utilizarse wildcards.
- Asociación de unidades a concerns: Deberán crearse los respectivos hyperslices, asociando a ellos las unidades correspondientes.
- Integración de concerns: Definición de hipermódulos, especificando los hyperslices incluidos y la regla de composición. Estas se especifican basándose en los nombres de las unidades, ya sea mediante la referencia completa o a través del uso de wildcards. Se dan distintas palabras clave para especificar como debe combinarse una unidad cuando aparece en más de un hyperslice.

Java Aspect Components: JAC!

Fuentes: [JAC1], [JAC2]

JAC se presenta como alternativa orientada a aspectos ante los application servers disponibles en el mercado: su ambición es ir más allá del proyecto de investigación para ser utilizado en aplicaciones comerciales. Quizá por ello el paquete incluye no sólo las extensiones necesarias para constituir un entorno AOP, sino también implementaciones de los aspectos más comunes: autenticación, load-balancing, sincronización, GUI, persistencia, etc... Es importante señalar que, a diferencia de AspectJ, se considera que los aspectos corresponden exclusivamente a las propiedades no funcionales del sistema.

Los autores destacan como aporte más importante a la disciplina la manera en que se trata la composición de aspectos cuando coincide la aplicación de varios sobre un mismo joinpoint. A nuestro parecer, el elemento distintivo de JAC es la visión dinámica con la que fue construido.

Visión general

JAC es un entorno AOP dinámico. Esto significa que el weaving de los aspectos con el código base se realiza en runtime. Se aprovecha un framework de metaprogramación de Java para realizar esa tarea. No es necesario por tanto el código fuente, ya que se trabajará directamente sobre bytecode.

Una aplicación JAC consiste de cuatro componentes principales:

- Programa base: Funcionalidad de la aplicación. No hay necesidad que los objetos de negocios implementen interfaces o sigan reglas para su construcción, tal como lo exige J2EE. Los distintos tipos que lo conforman son modificados en load-time para que puedan aplicarse aspectos sobre ellos.
- Aspectos: Implementación de los *crosscutting concerns.* Se programan en Java, especializando clases base provistas por el framework.
- Weaver: Responsable de aplicar en cada joinpoint los aspectos correspondientes
- Composition aspect: Define las reglas a seguir cuando varios aspectos coinciden sobre un mismo joinpoint

A modo de ejemplo…

En la construcción de aspectos se especifica cómo se alterará al programa base. Puede ser de tres formas diferentes:

1. Agregando anotaciones sobre el metamodelo de la aplicación
2. Incorporando *listeners* sobre algún evento del sistema
3. Definiendo pointcuts que especifiquen la aplicación de wrappers sobre distintos objetos. Dichos wrappers pueden alterar el comportamiento de un método (*wrapping methods*), extender su interfaz (*role methods*) o manejar las excepciones que eventualmente dispare (*exception handlers*).

En el siguiente código, por ejemplo, se agrega el atributo *synchronized* al método “m” de la clase especificada por el objeto AClass, se define comportamiento en la salida del sistema y se especifica la aplicación del wrapper “wrappingMethod” sobre todos los métodos de todos los objetos:

```
public class MyAspect extends AspectComponent {
      public MyAspect() {
            ClassItem cl = ClassRepository.get().getClass(AClass.class);
            cl.getMethod("m():void").setAttribute("synchronized", new
      Boolean(true));
            pointcut("ALL", "ALL", "ALL", MyWrapper.class, "wrappingMethod", true);
      }

      public void onExit() {
       [...]
      }
}
```

Pointcuts

Los tres primeros parámetros del método *pointcut* (definido en la clase AspectComponent) especifican el conjunto de joinpoints en dónde corresponde aplicar el aspecto. El primero corresponde al identificador del objeto, el segundo a la clase, y el tercero a la signatura del método. Como se muestra en el ejemplo, no necesariamente deben indicarse los nombres exactos, sino que existen ciertas palabras clave que permiten realizar referencias más genéricas, y que pueden a su vez combinarse para formar expresiones más complejas. Se puede predicar sobre relaciones de herencia entre las clases; se permite indicar también si se desea que los métodos sean exclusivamente accesors, getters o setters de fields, colecciones, referencias o tipos primitivos, o si se hace referencia a constructores o métodos estáticos. En todos los casos existe también la posibilidad de utilizar expresiones regulares.

El siguiente parámetro del método declara el wrapper a aplicar (TracingWrapper en el ejemplo). Se especifican por último un eventual manejador de excepciones que se desee utilizar (ninguno en este caso, representado a través del valor null) y un parámetro que indica si el wrapper se comportará como un singleton ([GAMMA]) o si se instanciará un objeto distinto por joinpoint.

Es posible definir pointcuts tanto en el constructor del aspecto como a través de un archivo de configuración destinado para tal fin. A continuación se muestra un ejemplo de esto último:

```
public class MyTracingAC extends AspectComponent
{
	public void addTrace(String cExpr,String oExpr,String mExpr)
	{
		pointcut(cExpr, oExpr, mExpr, TracingWrapper, null, false);
	}
	public class TracingWrapper extends Wrapper
	{
		public Object invoke(MethodInvocation mi)
		{
			System.out.println("Method  "  +  method()+"  is  called  on  "+
		wrappee());
			return proceed(mi);
		}
	}
}
```

```
		//myApp.jac
```

```
applicationName: myApp
launchingClass: myMainClass
aspects: MyTracing trace.acc true
```

Archivo de configuración de la aplicación.
Se especifican los aspectos a aplicar y sus respectivos archivos de configuración.

```
		// trace.acc
```

```
addTrace ALL ALL "CONSTRUCTORS && FIELDSETTERS";
```

#trace.acc
Se define un pointcut para agregar el aspecto trace a los constructores y setters de fields de todos los objetos

Y no queríamos dejar de mencionar a....

Caesar

El proyecto centra su atención en brindar los mecanismos necesarios para alcanzar una mejor separación de los conceptos que titulan *aspect implementation* y *aspect binding:* el primero corresponde a la implementación del aspecto en sí (el código para manejar transacciones, por ejemplo), y el segundo a *cómo se relaciona éste con el programa base*. Supongamos que implementamos una funcionalidad genérica de venta por internet. Este dominio será transversal a otros sobre los que querremos aplicarlo: nuestro software de manejo de librerías y el de manejo de florerías, por ejemplo. Entonces, ¿cómo implementamos esta relación? Los autores señalan que en general un simple mapeo no basta, sino que hacen falta *construcciones más complejas que hagan las veces de traductores entre ambos mundos*. Proponen entonces una serie de

construcciones sintácticas para implementar el concepto que denominaron *Aspect Collaboration Interfaces* [CAESAR1][CAESAR2].

JBOSS AOP

JBOSS es un application server de Java ampliamente difundido a la fecha. La incorporación de su propio framework AOP contribuye con la difusión de la nueva discipina en la industria, facilitando su uso en aplicaciones comerciales. El enfoque es similar al de JAC, aunque vale la pena destacar que los pointcuts pueden predicar sobre anotaciones hechas sobre el código [JBOSS].

Middleware

COM+[COM] y J2EE [EJB] son los dos ejemplos de *middleware* más difundidos en la actualidad. Ambos permiten agregar determinadas propiedades sistémicas sobre componentes desarrollados en esas plataformas. No es posible, sin embargo, incorporar nuevas o modificar el comportamiento de las existentes, y los mecanismos de cuantificación con los que cuentan son en extremo limitados. Carecen por ello de las características necesarias para ser considerados como verdaderas herramientas AOP, aunque sin dudas resultan significativas en la búsqueda de una mejor SOC en aplicaciones industriales.

Varios

Otras implementaciones influyentes son Composition Filters[CF], Adaptive Methods[AM] y AspectWerkz [AWERKZ]. Además de las herramientas de propósito general a las que nos referimos hasta el momento, existen algunas destinadas a resolver problemáticas particulares, como por ejemplo Iguana[IGUANA] (entorno de programación de sistemas distribuidos).

Rumbo a SetPoint

Introducción

En [FILMAN] se identifican como conceptos fundamentales de la programación orientada a aspectos los denominados *quantification* y *obliviousness*. Si el programador del código base debe estar atento a requerimientos de los aspectos, o si quienes desarrollan los aspectos deben conocer la forma en que está implementado el programa base, no se consigue alcanzar una real *separation of concerns*. Por otro lado, se deberá contar con buenos mecanismos para definir por comprensión los conjuntos de joinpoints en donde se aplicará cada aspecto: lo contrario equivale a conocer la estructura interna del programa. ¿Consiguen las herramientas actuales cumplir estos cometidos? La pregunta intentará ser contestada a lo largo del capítulo.

Pointcuts

Taxonomía

Las herramientas presentadas en el capítulo anterior ejemplifican las posibles formas de definir pointcuts, y nos permiten elaborar una pequeña taxonomía sobre las mismas:

1. Por enumeración: Deben indicarse uno por uno los puntos en donde vaya a aplicarse algún aspecto. Los ejemplos más difundidos de esta técnica son las ya mencionadas plataformas de middleware COM+ y J2EE.

2. Por comprensión sintáctica: Se definen los pointcuts a la manera de AspectJ especificando las condiciones que debe cumplir un joinpoint para pertenecer al conjunto. Para ello se definen expresiones regulares sobre el nombre de métodos, constructores, clases o campos. También pueden especificarse condiciones sobre los tipos de los parámetros de los métodos.

3. Por comprensión sintáctica + anotaciones: Se hacen anotaciones sobre el código, que luego pueden ser utilizadas junto con las expresiones regulares para la definición de los pointcuts: puede decirse, por ejemplo, que pertenezcan al pointcut A todos los métodos cuyo nombre comience con “myMethod*” y que tengan la anotación “annotation1”. JBOSS y AspectWerkz son dos de las herramientas que permiten este tipo de definiciones.

Crítica al estado actual: “AOSD Evolution Paradox”

Esta claro que el primero de los grupos no cumple con el objetivo de *quantification*: los joinpoints en donde aplicar un aspecto deben indicarse uno a uno, por lo que no hay ninguna forma de predicado. Ya sea el

programador del código base o aquel que vaya a determinar la aplicación de aspectos deben conocer a fondo las otras dimensiones del problema, violándose así la condición de *obliviousness* buscada.

¿Qué pasa con el segundo grupo? Los pointcuts se basan en la adopción de convenciones sintácticas sobre el código base. Se puede decir que todos los métodos cuyos nombres comiencen con “set*“ necesitan persistirse en una base de datos, o que todas las llamadas desde instancias de clases cuyo nombre sea “*user*” deban ser registradas. Existe la posibilidad de elaborar predicados para determinar en dónde aplicar aspectos: se cuenta con mecanismos de cuantificación. Sin embargo, la necesidad de asumir dichas convenciones plantea serios inconvenientes, bautizados por Tourwé, Brichau y Gybels como “AOSD-Evolution Paradox” [PARADOX]. Los autores sostienen que el nivel de acoplamiento que terminan teniendo los aspectos y el código base debido a esa manera de definir los pointcuts logra el efecto contrario al buscado: la *SOC* obtenida es ficticia, y el mantenimiento de la aplicación termina siendo tanto o más difícil que antes. En su paper señalan además los siguientes inconvenientes:

- *Se puede estar trabajando con otro que requiera también de sus propias convenciones de notación, lo que seguramente originará conflictos.*

- *No es posible garantizar que todos los joinpoints que integren un pointcut cumplan con la regla requerida. Está claro que se abre también la posibilidad para la aparición de falsos negativos.* Creemos que tal como se lo señala el problema es relativo: siempre existirá la posibilidad de error en la definición de pointcuts. Sin embargo, consideramos que el carácter arbitrario de las convenciones sintácticas aumenta la probabilidad de que ocurran equivocaciones de esta índole.

- *Es bien sabido que los programadores no siempre pueden cumplir con estas convenciones, ya sea por la presión de los deadlines, por la complejidad de la aplicación o por falta de documentación y soporte.* Nuevamente, creemos que el tema pasa por la arbitrariedad de las convenciones sintácticas, y por la consecuente dificultad en su aprendizaje y aplicación: en cualquier ambiente de trabajo el programador deberá seguir reglas, caso contrario el programa no funcionará según lo esperado.

Encontramos además un cuarto inconveniente:

- Si corresponde aplicar varios aspectos sobre un mismo joinpoint, se deberán seguir una cantidad de convenciones sintácticas sobre un mismo identificador, lo cual podría llevar a nuevos conflictos.

En [SHERLIS] se comenta nuevamente la problemática de la definición de pointcuts basados en el conocimiento de reglas sintácticas y en la estructura del programa, señalándolo otra vez como causa de acoplamiento entre el código base y los aspectos. Coincidimos con estas críticas: creemos que esta forma de definir pointcuts no permite alcanzar una real *separation of concerns*.

La posibilidad de utilizar anotaciones sobre el código para la definición de pointcuts presenta un cambio radical con respecto a la adopción de convenciones sintácticas, tema sobre el cual nos explayaremos en los apartados siguientes.

Camino a la meca

¿Por qué decidimos en algún momento aplicar un aspecto X cualquiera sobre "todos los métodos que coincidan con el patrón "draw*? Cuando desarrollamos nuestra aplicación, decidimos titular a todos los métodos que realizaran tareas de dibujo con un prefijo que los identificara conjunto. Dicha semántica permanecerá entonces implícita bajo la sintaxis del nombre, originando los problemas descritos en apartado anterior. Una solución intuitiva es, entonces, reificar estos significados, hacer que formen parte del programa para luego poder predicar directamente sobre ellos. En [ASOC] aparece el término *semantic joinpoints*: permitir que los pointcuts se definan no sólo basados en sintaxis, sino también en contenido semántico. La posibilidad de predicar sobre anotaciones tal como lo propone el tercero de los grupos puede ser vista como una oportunidad para desarrollar la idea anterior. Quedan sin embargo muchos interrogantes por responder antes de que esto pueda ser visto como una solución definitiva: ¿Qué significa más exactamente *explicitar significado*? ¿Qué anotaciones deben hacerse? ¿Qué estructura deben tomar? En las secciones siguientes nos extenderemos sobre algunas cuestiones que creemos relevantes a la hora de buscar respuestas.

Paréntesis: algunas consideraciones necesarias

Antes de adentrarnos en el tema de las anotaciones, creemos que vale la pena detenernos sobre algunas cuestiones generales acerca de la orientación a aspectos.

AOP y paradigmas de programación

La programación orientada a objetos es sin dudas el paradigma de programación más difundido actualmente en la industria del software. Su aparición trajo sensibles mejoras en cuanto a encapsulamiento y reuso de código. La continua maduración de la disciplina introdujo incluso nuevas variantes para el reaprovechamiento de trabajo ya realizado, con frameworks y patrones de diseño erigiendose como abanderados. Con el tiempo comienza a prestarse atención también a sus falencias: no hay una buena modularización de las dimensiones transversales a la descomposición funcional.

Surgen una cantidad de técnicas y trabajos tendientes a atacar este problema, y conseguir un nuevo avance en la lucha por la separación de incumbencias. AOP aparece entre ellos. Las ideas centrales que hacen a esta disciplina, sin embargo, no están asociados únicamente con OOP, sino que pueden aplicarse también a los paradigmas imperativo, lógico y funcional. Ya en [KIC97], uno de los papers fundacionales de la disciplina, se hace referencia a ellos agrupándolos bajo el nombre de *generalized-procedure languages*. Pese a ello, se vislumbra una fuerte inclinación hacia los objetos: los primeros ambientes de experimentación se desarrollan sobre Java, Smalltalk o similares; los primeros joinpoints corresponden a las llamadas a métodos; las características de *introduction* que aparecen en AspectJ permiten modificar clases existentes.

En el presente trabajo nos limitaremos a la aplicación de aspectos sobre plataformas orientadas a objetos. No intentaremos extendernos hacia otros paradigmas: creemos que el trabajo que resta por hacer sobre modelos orientados a objetos es aún lo suficientemente importante.

Las diferentes dimensiones de un sistema

Algunos autores plantean el uso de aspectos para la modularización de dominios transversales, mientras que otros consideran su utilización también para el modelado de elementos de un mismo dominio. Consideramos al primer enfoque como el más natural y práctico de ambos. Tendemos a pensar que el segundo complica el código y no aporta mayores beneficios; la orientación a objetos plantea en general buenas soluciones para modelos que abarquen un único dominio. Esta opinión se basa tanto en nuestra experiencia y juicio personales como en la falta de datos o pruebas en la comunidad en general: no hallamos ejemplos ni métricas concluyentes que justifiquen lo segundo y nos convenzan de ello.

Los aspectos modelan entonces dominios independientes. No importa si son finalmente entremezclados los unos con los otros, cada uno deberá estar en condiciones de funcionar por si solo. Esto implica una condición aun más fuerte que la *completitud declarativa* de HyperJ: cada uno debe ser un componente de software utilizable en si mismo.

¿Qué dominios encontramos que cumplan con dichas características? Por un lado, el modelo funcional y, por el otro, los ejemplos que se dan habitualmente cuando se habla de aspectos: distribución, seguridad, persistencia, y otras propiedades sistémicas. Consecuentemente, creemos que las tecnologías orientadas a aspectos tienen uso principalmente como application servers y middleware, mejorando sustancialmente la potencialidad de las herramientas actuales. Algunas plataformas (JBoss y JAC, por citar dos ejemplos) parecen avanzar en esta dirección.

Habiendo llegado a la conclusión anterior: ¿Es conveniente implementar esta cantidad restringida de dominios mediante un lenguaje de propósito general, o será mejor aprovechar las capacidades de lenguajes específicos diseñados ad-hoc para su representación [DSL]? Siguiendo la tendencia actual de la comunidad, en el presente trabajo nos inclinaremos por la primera opción.

Representaciones y trazabilidad

Documentación y modelos de casos de uso, especificaciones de requerimientos, lineamientos de arquitectura, diseños de detalle, entre otros, son creados y actualizados constantemente a lo largo del proceso de desarrollo de software. Cada uno presenta una visión distinta del sistema a construir, enfocada desde las necesidades de un stakeholder en particular.

Es común en publicaciones sobre ingeniería del software hablar de la importancia de la trazabilidad[13] entre estos artefactos para conseguir una construcción y posterior mantenimiento exitoso de un sistema. Creemos que las plataformas orientadas a aspectos podrían sacar especial provecho de relaciones explícitas

[13] **Definición**: “degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another” [IEEE90]

entre el código fuente y otras perspectivas del sistema. Es menester para esto que se tenga acceso a dichas relaciones: imaginamos una línea de código que nos permita decidir si la ejecución actual del método "setAmount" corresponde al caso de uso "Vendiendo mercadería en Stock". En la sección siguiente enfatizaremos esta idea situándola en un contexto AOP.

Dejando volar la imaginación: setpoints!

Dijimos que en nuestra opinión los aspectos sirven eminentemente como plataforma de middleware para la aplicación de requerimientos no funcionales sobre una aplicación. En general, el arquitecto o los diseñadores responsables de dichas cuestiones se referirán al sistema en un nivel de abstracción mayor al código fuente: hablarán en términos de capas, componentes, patrones u otras construcciones acordes a los métodos utilizados, y muy probablemente no detallarán cada uno de los métodos, clases o atributos, dejando esto para quienes se ocupen específicamente del dominio del problema, sabiendo que se ajustarán a la estructura y a los lineamientos por ellos determinados.

Creemos que la forma más cómoda de decidir sobre la aplicación de aspectos será entonces predicar en ese mismo nivel,:

- "Aplicar un mecanismo de caché sobre cada mensaje enviado a objetos de la `[capa de datos]` manejado por un método que cuente con `[transparencia referencial]` "

- "Encriptar el envío de todos los mensajes cuyos argumentos sean `[datos sensibles]` en el contexto de una `[transacción comercial]` "

Las anotaciones que figuran entre llaves corresponden a las anotaciones sobre el programa que planteábamos en la sección inicial. Es información plasmada directamente sobre el código, que explicita propiedades del mismo relacionándolo a la vez con otras visiones: hay referencias a un modelo de arquitectura (capa de datos), a una caracterización de la información (datos sensibles) y a ciertos aspectos funcionales (transacción comercial). Nótese que no estamos haciendo marcas del tipo *aquí se debe aplicar el aspecto A*, como en el caso del grupo de pointcuts que denominamos *por enumeración*, sino que estamos aprovechando el haber explicitado información sobre el código para tener elementos semánticos sobre los cuáles predicar, y poder alcanzar así un menor nivel de acoplamiento entre las distintas dimensiones del sistema.

Los ejemplos anteriores ilustran lo que imaginamos como pointcut en un framework AOP. Representan nuestra visión de los *semantic joinpoints* de los que se hablaba en [ASOC]. Los bautizaremos **setpoints**: *semantic pointcuts.*[14]

[14] **Nota de los autores**: Sí, esta era la verdadera razón... mal que nos pese, no era en honor a Coria, Gaudio, Nalbandian, o al resto de los legionarios.

Cómo anotar el código

En el apartado anterior vimos dos ejemplos de *setpoints* escritos en lenguaje coloquial. Aparecen entre corchetes los conceptos semánticos que queremos explicitar. Ahora bien... ¿Cómo está modelada dicha semántica? ¿Cómo se define cada concepto? ¿Están relacionados entre ellos?

Procuramos definir una forma de representación de conocimiento que nos permita expresar conceptos que luego relacionaremos con la aplicación, para lo cual deben ser obviamente pasibles de ser representados por una computadora. Debe existir también la capacidad de realizar consultas sobre los mismos: se deben poder elaborar predicados complejos, para satisfacer las características de cuantificación necesarias en un entorno AOP. La web semántica [SciAm], otra área de las ciencias de la computación, se enfrentó con desafíos similares, consiguiendo importantes avances basados en la filosofía y en la inteligencia artificial: el uso de ontologías permitió encontrar algunas respuestas [W3C]. En el presente trabajo exploraremos la aplicación de parte de estos resultados en un framework orientado a aspectos.

El enfoque tomado no es el único que podría adoptarse para experimentación sobre el modelado de la semántica. Decidimos seguirlo por estar en la actualidad lo suficientemente difundido como para creer que su utilización es factible y conveniente. Otros posibles caminos que podríamos haber seguido son los siguientes:

- Representar el modelo semántico utilizando modelos de objetos.

- Utilizar lenguajes particulares para la definición de cada modelo semántico de interés. Esto no daría un marco generalizable a cualquier modelo, pero dado que en nuestra opinión los aspectos son útiles en una cantidad relativamente limitada de cuestiones, podría llegar a servir. Podría usarse, por ejemplo, un ADL para la visión de arquitectura y una taxonomía sencilla para clasificar la importancia de la información manejada. La opción es similar a la que plantea el uso de lenguajes específicos de dominio [DSL] frente a lenguajes de programación de propósito general.

Resumen

En este capítulo se presentó el problema de la definición de pointcuts basados en convenciones sintácticas, bautizado como AOSD-Evolution Paradox. Luego se esbozó una posible solución: los setpoints, pointcuts definidos en base a la semántica de la aplicación. Es menester contar con un modelo de representación semántica para cumplir esta tarea. En esta tesis se explorará el enfoque utilizado por el movimiento de la web semántica en la resolución de problemas similares, basado fuertemente en el concepto de ontologías. En el capítulo siguiente se explicará dicho enfoque.

Volviendo computable el saber

Introducción

Los pointcuts semánticos podrán ser implementados únicamente en la medida en que encontremos una forma de estructurar conocimiento sobre aquellos universos relacionados al programa sobre los que nos interese predicar. Siguiendo con los ejemplos de setpoints planteados en el capítulo anterior, necesitaremos algún modelo que nos permita definir los conceptos de *capa de datos* e *información sensible*, relacionándolos con otros términos de su mismo dominio. El objetivo de definir cualquier término nos plantea entonces un importante desafío: ¿cómo hacer un modelo lo suficientemente descriptivo, y a la vez comprensible por una máquina? La denominada web semántica enfrenta problemas similares. El camino trazado en la búsqueda de una solución se basa en otra área explorada asiduamente por las ciencias de la computación en la última década: las ontologías. En el presente capítulo se introducen dichas ideas, utilizadas luego como base para la construcción de la herramienta orientada a aspectos *SetPoint*.

WWW: La nueva generación

Es viernes a la noche y aún faltan un par de horas para comenzar a disfrutar de la movida porteña. Quizá como mecanismo inconsciente para no caer en la tentación de dormir una siesta que nos ocupe hasta la mañana siguiente, nuestra curiosidad nos inunda con el deseo irrefrenable de conocer los films policiales protagonizados por el gran actor argentino Carlos Calvo. Ingresamos a Google, tipeamos su nombre y... 119.000 coincidencias en 0,08 segundos. Leemos los resultados: el primer link corresponde al jurista, historiador y diplomático uruguayo Carlos Calvo, nacido en Montevideo en 1822. No nos sirve. El segundo, a Juan Carlos Calvo Prieto, director del Programa Infantil Phonak desde Mayo de 2002. Tampoco. El tercero es una nueva cita a nuestro ex-vecino charrúa. Seguimos participando. Cuarto link... ¡Ahora sí! Datos personales y filmografía completa. Ingresamos. Nos sorprendemos unos instantes con su fecha de nacimiento (21/02/1953), y luego sí, nos concentramos en sus actuaciones en “Comodines”, “Adiós, Roberto” y “Ritmo, amor y primavera”. Las sinopsis dan cuenta de que únicamente el primero pertenece al género policial, por lo que la anotamos en nuestra lista de próximos DVDs a alquilar.[15]

La información que se nos fue presentando en las sucesivas páginas pudo ser interpretada correctamente por... nosotros. ¿Quién más hubiera podido hacer lo propio llegando a la misma conclusión? Pues bien, el objetivo de la así llamada web semántica es que el contenido de las páginas que habitan la world wide web pueda ser comprendido también por las máquinas. Un argumento falaz podría decir que la búsqueda anterior podría haber obtenido mucho mejores resultados si se hubiera utilizado un criterio de búsqueda más refinado. Esto es parcialmente cierto. Se podrían haber obtenido mejores resultados, pero nunca se hubiera podido llegar

[15] Los datos fueron obtenidos el 06/09/2004

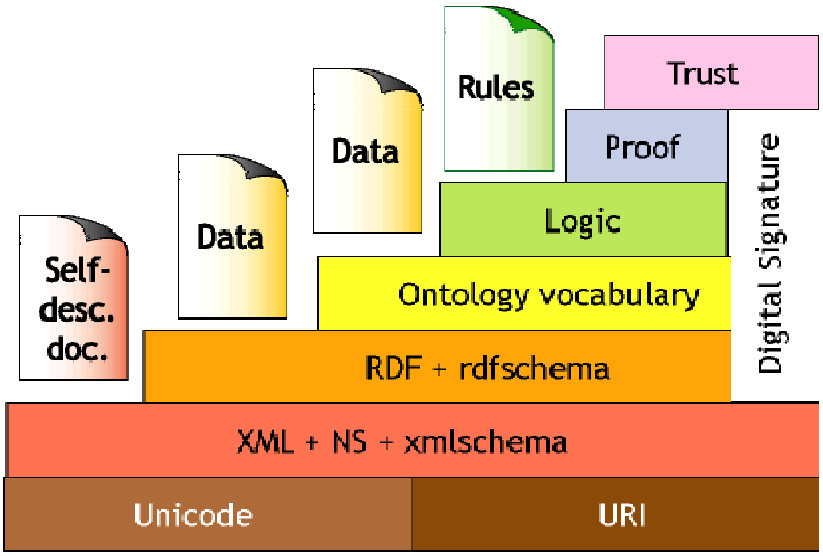
a una conclusión tan terminante: no hay forma de que a través de la sinopsis la computadora pueda interpretar que se trataba de un policial, una comedia o un drama.

En [SciAm] se explica el problema planteado en los párrafos anteriores y el camino de la solución:

"El contenido de la web está hoy en día pensado mayormente para ser leído por seres humanos, no por programas que lo puedan manipular siendo conscientes de su significado. Las computadoras pueden parsear páginas web para dar formato y ejecutar procesos rutinarios (aquí hay un link, aquí un header), pero, en general, no existe una manera confiable de procesar su semántica (...). La web semántica proveerá una estructura para el significado de las páginas, creando un entorno en donde agentes de software puedan llevar adelante tareas complejas, navegando por sí solos y desarrollando tareas que antes realizaba el usuario".

Arquitectura de la web semántica

La arquitectura propuesta para estructurar la web semántica consta de una serie de capas que se van apoyando sucesivamente sobre las anteriores para agregar paulatinamente expresividad al modelo. Se resume en el siguiente gráfico, presentado en el año 2000 por Tim Berners-Lee[16]:



Todo se basa en Unicode, formato universal de representación de símbolos, y en el concepto de URIs, identificadores universales de recursos. Son una expresión más general de las más conocidas URL: no sólo permiten referirse a elementos que tengan una ubicación física en la web, sino también a todo concepto que pueda ser identificado. Siguiendo con el ejemplo del apartado anterior, el actor Carlos Calvo tendría asignado un URI que lo identificaría entre todos los individuos, distinguiéndolo incluso del Carlos Calvo uruguayo, que tendría también su propio identificador. La real utilidad de estos identificadores se dará una vez que sean universalmente conocidos y concensuados. Por sobre ellos aparece la tecnología XML, que da la posibilidad de

[16] El gráfico pertence a una presentación publicada por la W3C que se encuentra en http://www.w3.org/2000/Talks/1206-xml2k-tbl/

estructurar documentos mediante tags. El formato permite a plataformas disímiles interoperar fácilmente, existiendo en la actualidad numerosas herramientas que permiten parsear y realizar operaciones relativamente complejas sobre este tipo de documentos.

Luego encontramos a RDF y RDF-Schema: vocabularios que permiten definir un concepto cualquiera, especificando sus tipos y propiedades. Sobre ellos se posa la capa de ontologías, representada actualmente a través del estándar OWL (Ontology Web Language): ampliamos el modelo permitiendo definir relaciones y restricciones entre ellos. La capa lógica desarrolla cierta inteligencia. A partir de los elementos recién definidos, se deducen mediante la aplicación de reglas de inferencia enunciados no explicitados. El componente de firmas digitales representa la necesidad de mecanismos de seguridad.

No todas las capas tienen actualmente un mismo nivel de desarrollo. La tecnología XML ya ha sido ampliamente adoptada por el mercado, asignándosele usos varios. RDF y RDF-Schema tienen ya sus recomendaciones del W3C consortium y cuentan con una cantidad de herramientas que permiten la definición de documentos en el formato especificado. OWL se ubica en un estrato inmediatamente superior. La aprobación del estándar es mucho más reciente, y tanto su concepción teórica como herramientas prácticas continúan en pleno desarrollo. La investigación sobre las capas superiores es aún incipiente.

Resource Description Framework (RDF)

Conceptos básicos

Según una definición provista por la W3C, RDF es un ..."lenguaje para describir información sobre recursos de la WWW (...). Generalizando la definición de recurso web, se puede decir que RDF permite estructurar información sobre cualquier elemento que pueda ser identificado en la red"[17]... De cada uno de estos elementos se asegura primero su unicidad como concepto, garantizándose que pueda ser identificado de una manera no ambigua. Podremos luego alcanzar un modelado más amplio y ambicioso a través de la definición de relaciones con otros conceptos. Se plantea con este fin la existencia de los dos únicos building blocks que ofrece el modelo: los recursos y los statements realizados sobre ellos, que pueden a su vez ser considerados también recursos.

Los recursos tienen entonces como única propiedad intrínseca su identidad, dada por una URI (Unified Resource Identifier). Los statements continúan el enfoque minimalista y se definen siempre como triplas del tipo <sujeto, predicado, objeto>[18], donde cada uno de sus elementos debe a su vez ser un recurso (recordemos que los statements podían a su vez ser considerados recursos, dado que es posible asignarles una URI). Como toda decisión en la vida, esta manera de representar las relaciones trae beneficios y perjuicios: el hecho que el predicado sea un recurso está reificando la relación entre los recursos sujeto y objeto. Esta manera de representar relaciones, sin embargo, parece obligarnos a convivir en un mundo de relaciones definidas únicamente por extensión. Es cierto, pero no hay que olvidar que el mundo que plantea RDF es en extremo

[17] En http://www.w3.org/TR/rdf-primer/#intro, introducción a RDF recomendada por la W3C

[18] Los componentes de las triplas son también referidos en ocasiones como <recurso, propiedad, valor>

simple y no es útil per se, sino como plataforma sobre la que es posible definir modelos semánticos más ricos, y por ende más útiles.

Representaciones de RDF

Supongamos que queremos representar en RDF el siguiente predicado[19]:

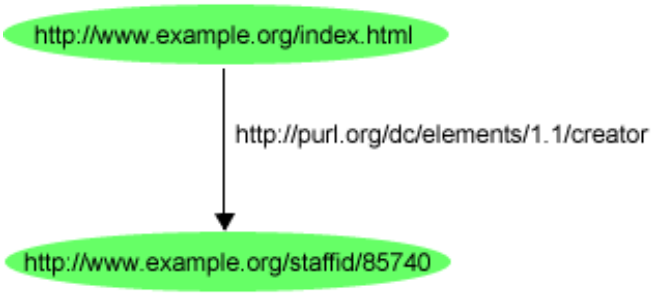
> **http://www.example.org/index.htm** tiene como **creador** a **John Smith**

Analizando el *statement* desde la perspectiva RDF, lo descomponemos en los siguientes elementos:

- *Sujeto*: http://www.example.org/index.htm
- *Predicado*: http://purl.org/d/elements/1.1/creator (URI del recurso *creador*)
- *Objeto*: http://www.example.org/staffID/85740 (URI del recurso John Smith)

Una primer forma de representación es a través de una tripla formada por las tres partes constituyentes.

Una segunda posibilidad es graficando la relación a través de un grafo, en donde se dibujan sujeto y objeto como nodos unidos por un arco que representa al predicado:



Por último, el enfoque que se utilizará como medio de comunicación entre máquinas consiste en un formato XML definido para la representación de recursos RDF, llamado, casualmente, XML/RDF:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
```

[19] El ejemplo se encuentra también en el artículo anterior

```
  <rdf:Descriptionrdf:about="http://www.example.org/index.html">
      <dc:creator rdf:resource=”http://www.example.org/staffid/85740” / >
  </rdf:Description>
</rdf:RDF>
```

No pretendemos con esta breve introducción dar una descripción completa del modelo. Hay algunos conceptos (literales, blank nodes) incluidos en la especificación que creemos se apartan ligeramente del enfoque minimalista que tiene en esencia RDF, y consideramos no aportan valor agregado a nuestro trabajo. Por la misma razón, evitamos también profundizar aquí en la especificación formal del lenguaje. La información completa sobre estos estándares puede encontrarse en [W3CRDF].

RDF-Schema

RDF-Schema extiende RDF permitiendo la definición de vocabularios. Se dan herramientas que posibilitan la especificación de clases, de manera equivalente a un sistema de tipos en un lenguaje de programación. Esto restringe la estructura que puede tomar un elemento de un dominio determinado: mi Ford Falcon es un automóvil, y como tal debe tener necesariamente una patente. La clase auto es subclase de la clase Vehículo. Ergo, sirve para movilizarse.

El objetivo anterior se alcanza simplemente estableciendo convenciones sobre la semántica de determinados términos. Se define por ejemplo que el término SubclassOf determinará una relación de especialización entre dos conceptos. Al no ser más que recursos RDF con significado bien definido, pueden ser utilizados en la elaboración de cualquier statement válido. Cualquier definición de RDF-Schema conforma por lo tanto un conjunto de recursos compatibles con RDF.

Entre los términos declarados encontramos Class, Datatype, Property, domain, range, subClassOf, subPropertyOf, Container, Bag, List, subject, predicate y object. La lista completa con sus respectivas explicaciones aparece en http://www.w3.org/TR/rdf-schema/.

El siguiente extracto de RDF-Schema define, por ejemplo, un vocabulario en donde se define una taxonomía de vehículos (el formato es XML/RDF):

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://example.org/schemas/vehicles">

<rdfs:Class rdf:ID="MotorVehicle"/>

<rdfs:Class rdf:ID="PassengerVehicle">
  <rdfs:subClassOfrdf:resource="#MotorVehicle"/>
```

```
</rdfs:Class>

<rdfs:Class rdf:ID="Truck">
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Van">
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="MiniVan">
  <rdfs:subClassOf rdf:resource="#Van"/>
  <rdfs:subClassOf rdf:resource="#PassengerVehicle"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Person"/>

<rdfs:Datatype rdf:about="&xsd;integer"/>

<rdf:Property rdf:ID="registeredTo">
  <rdfs:domain rdf:resource="#MotorVehicle"/>
  <rdfs:range rdf:resource="#Person"/>
</rdf:Property>

<rdf:Property rdf:ID="rearSeatLegRoom">
  <rdfs:domain rdf:resource="#PassengerVehicle"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>

<rdf:Property rdf:ID="driver">
  <rdfs:domain rdf:resource="#MotorVehicle"/>
</rdf:Property>

<rdf:Property rdf:ID="primaryDriver">
  <rdfs:subPropertyOf rdf:resource="#driver"/>
</rdf:Property>

</rdf:RDF>
```

Puede luego también definirse una instancia de vehículo basada en el vocabulario anterior:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:ex="http://example.org/schemas/vehicles#"
            xml:base="http://example.org/things">
```

```
  <ex:PassengerVehicle rdf:ID="johnSmithsCar">
       <ex:registeredTo rdf:resource="http://www.example.org/staffid/85740"/>
       <ex:rearSeatLegRoom
           rdf:datatype="&xsd;integer">127</ex:rearSeatLegRoom>
       <ex:primaryDriver rdf:resource="http://www.example.org/staffid/85740"/>
  </ex:PassengerVehicle>
</rdf:RDF>
```

RDF y RDF-Schema conforman un fundamental primer eslabón para la conformación de la web semántica; sin embargo, su poder de expresión es limitado. En la sección siguiente se introduce el enfoque seguido para aumentar dicha capacidad.

Ontologías

Según el diccionario de la Real Academia Española, la ontología es la "parte de la metafísica que trata del ser en general y de sus propiedades trascendentales". ¿Qué tiene que ver esta definición un tanto esotérica con la informática? A principios de la década del '90, distintas áreas de las ciencias de la computación, entre ellas y con especial ímpetu la inteligencia artificial, comienzan a explorar medios para compartir conocimiento entre agentes de software. Llaman entonces *ontologías formales* a los modelos planteados para cumplir este cometido. En [GRUBER], se define al término como una "especificación explícita de una conceptualización", donde entiende a esto último como a "los objetos, conceptos y otras entidades pertenecientes a algún área de interés, y a las relaciones sostenidas entre ellos"[20].

¿En qué consiste, entonces, una ontología, en el sentido que le damos nosotros a la palabra? En un párrafo de [FALBO1] se da respuesta muy clara a este último punto, valiéndose de citas a [USCHOLD] y [FALBO2]: ..."'Una ontología puede adoptar una cantidad de formas diferentes, pero necesariamente debe incluir un vocabulario de términos y alguna especificación de su significado. Esto incluye sus definiciones y algún tipo de indicación que muestre cómo estos conceptos están interrelacionados, lo que impone una estructura al dominio y restringe las posibilidades de interpretación de dichos términos'. Una ontología consiste, entonces, de un conjunto de conceptos y relaciones, conjuntamente con sus significados, definiciones y propiedades, expresadas en la forma de axiomas. Una ontología no es sólo una jerarquía de términos, sino una teoría axiomática completa sobre un dominio...".

Ontology Web Language (OWL)

OWL es, como su nombre lo indica, un lenguaje de representación de ontologías ideado para la web semántica, aprobado en febrero del 2004 como recomendación del W3C. Está inspirado en un desarrollo previo

[20] También en [GRUBER], citando a M. R. Genesereth y N. J. Nilsson,, *Logical Foundations of Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann Publishers, 1987.

llamado DAML+OIL, un lenguaje de markup para agentes creado por la DARPA[21]. Montado sobre RDF y RDF-Schema, extiende su vocabulario y enriquece su semántica, dando un marco más poderoso que los anteriores para la descripción de modelos. Permite, entre otras cosas, especificar cardinalidad en las relaciones, definir clases enumeradas y determinar simetría y transitividad. Este mayor poder de expresión acarrea una consecuencia importante: no es decidible (no puede ser dictaminada ya la veracidad o falsedad de todo enunciado); la aplicación de las reglas de inferencia no se transforma necesariamente en un algoritmo computable. Es por ello que se definieron las siguientes variantes del lenguaje, de acuerdo a las necesidades que se tengan:

- OWL Lite: Provee las funcionalidades básicas para la definición de clases y algunos recursos muy limitados para especificar restricciones. Únicamente permite restringir cardinalidades a 0 o 1 elementos, por ejemplo.
- OWL DL: Su nombre proviene de la disciplina denominada *Description Logics*, precursora de la ontología en la búsqueda de modelos formales de representación de conocimiento. Brinda el máximo poder expresivo posible sin perder decidibilidad ni computabilidad. Se puede responder en un tiempo finito si un predicado es verdadero o falso.
- OWL Full: Brinda el máximo poder expresivo posible, a costa de no poder siempre dar una respuesta ante consultas hechas al modelo. El vocabulario definido es el mismo que en la versión DL, sólo que puede combinarse libremente con statements RDF-Schema, pudiéndose entremezclar irrestrictamente clases, propiedades, individuos y valores. La posibilidad que un recurso pueda ser a la vez clase e instancia, conjunto e individuo, permite pensar, pese a la ausencia aún de demostraciones formales, que este lenguaje ingresa dentro del campo de la lógica de segundo orden, y por tanto no es decidible.

Los primeros dos sublenguajes imponen restricciones sobre el uso de las capacidades de metamodelado de RDF-Schema. Ergo, cualquier modelo definido en OWL Lite/DL forma un esquema RDF válido, pero no necesariamente a la inversa. La variante Full, por el contrario, no impone restricción alguna. A continuación presentamos a modo de ejemplo el extracto de una ontología definida en OWL DL:

```
<owl:Class rdf:ID=”SweetFruit”>
    <rdfs:subClassOf rdf:resource=”#EdibleThing” />
</owl:Class>

<owl:Class rdf:ID=”NonSweetFruit”>
    <rdfs:subClassOf rdf:resource=”#EdibleThing” />
    <owl:disjointWith rdf:resource=”#SweetFruit” />
</owl:Class>
```

[21] Defense Advanced Research Project Agency, agencia del Departamento de Defensa de los EEUU encargada del desarrollo de proyectos de investigación

```
<owl:Class rdf:ID="Fruit">
    <owl:unionOf rdf:parseType="#Collection" />
        <owl:Class rdf:about="#SweetFruit" />
        <owl:Class rdf:about="#NonSweetFruit" />
    </owl:unionOf>
</owl:Class>
```

Se puede apreciar únicamente la definición de clases; en un modelo completo, también se declaran sus individuos, en un concepto análogo al de instancias en la orientación a objetos. Se da forma de esta manera a la teoría axiomática de la que se hablaba en la definición de ontologías, sólo que expresándola en XML en vez de cláusulas de Horn u otro tipo de enunciados lógicos.

La documentación relacionada con la recomendación de la W3C puede encontrarse en http://www.w3c.org/2004/owl/.

Consultando ontologías

Un modelo RDF u OWL no terminaría de ser totalmente útil si no existiera alguna forma de realizar consultas sobre él para detectar la veracidad o falsedad de afirmaciones dentro del universo en cuestión. Diversas propuestas han sido ya planteadas; el desarrollo en el área, sin embargo, continúa aún evolucionando, tanto a nivel práctico como a nivel teórico, y está lejos de considerarse un tema cerrado.

En general, pueden encontrarse soluciones a la problemática anterior a tres niveles diferentes [SESAME]:

- Nivel Sintáctico: Puede aprovecharse el formato XML subyacente en los modelos para utilizar algún método de consulta basado en dicha tecnología, como ser XPath. Estos se sustentan en condiciones de navegabilidad sobre el árbol formado por el documento XML y en la aplicación de patrones o expresiones regulares sobre las etiquetas de sus nodos. El gran problema de este enfoque es, justamente, que un modelo RDF va más allá de su representación: es un conjunto de statements relacionados, que puede ser representado tanto en XML como a través de un grafo u otros formalismos. En caso de bajarse a este nivel de abstracción, se perdería la riqueza del modelo. Se pasaría a depender por ejemplo de cómo se haya decidido ordenar los nodos en la estructura del XML, arrojando un mismo query resultados distintos sobre modelos equivalente.

- Nivel Estructural: Consiste en ver a todo documento RDF como un conjunto de triplas. Se puede, por lo tanto, hacer cualquier consulta que involucre la existencia de una combinación <sujeto, predicado, objeto> particular: "`select ?x from... where (type ?x famousWriter)`". Pese a tratarse de un enfoque sensiblemente mejor al anterior, no se aprovecha la semántica definida para términos RDF-Schema o OWL. Las relaciones transitivas pasarán inadvertidas, por citar un único ejemplo.

- Nivel Semántico: Permiten explotar los modelos semánticos, realizando inferencias y deducciones lógicas en base a los hechos definidos y a las características de sus relaciones. En general, la arquitectura de las herramientas que permiten este tipo de consulta incluye dos componentes fundamentales: el *lenguaje de consultas* en sí mismo, cuya sintaxis define la forma en la que el usuario del modelo especifica la información que pretende averiguar, y un *motor de inferencias,* que es quien tiene la inteligencia necesaria para deducir enunciados no explicitados a partir de la semántica de los términos. No todos los motores satisfacen los modelos completos de RDF u OWL. Pueden interpretarlos en su totalidad, en parte, o ser independientes de sus reglas, planteando las suyas propias, aunque esto definitivamente vaya en contra de las pretensiones de conocimiento universal de la web semántica.

  Los lenguajes integrantes de esta familia son relativamente similares, rastreándose sus orígenes en las formas de consulta de modelos orientados a objetos; se proveen en este caso queries simples para la obtención de individuos y medios para navegar el grafo conformado por las triplas definidas en los modelos consultados. Entre aquellos que vale la pena mencionar encontramos a RQL [RQL], inspirado en OQL, primera propuesta basada en RDF; RDQL [RDQL] y SeRQL [SeRQL], versiones mejoradas del anterior ideadas para ser utilizadas desde las APIs Jena [JENA] y Sesame [SESAME], respectivamente; y por último OWL-QL [OWL-QL], reciente propuesta de la universidad de Stanford para la ejecución de queries sobre OWL, que incluye además un protocolo de interacción de agentes de software en la WWW.

Herramientas y desarrollos relacionados

Numerosas herramientas han sido ya desarrolladas para el uso y manejo de ontologías. En [DENNY] se identifican y categorizan más de 50 editores; en [ONTOWEB] se elabora un framework de evaluación que se aplica sobre 11 productos de desarrollo, 4 de integración, 5 de evaluación, 6 para la realización de anotaciones y 14 para el almacenamiento y consulta. En dichas fuentes aparece una reseña exhaustiva de todos ellos, comparando sus capacidades. Al igual que lo ocurrido con los lenguajes de consulta, la mayoría de las herramientas de uso masivo tiende a incorporar RDF y OWL, pese a haber sido concebida quizá para el uso de otros formatos.

Dos grupos necesitaron nuestra especial atención para la elaboración del presente trabajo: los editores y los motores para uso desde plataformas de desarrollo. Entre los primeros encontramos OntoEdit [ONTOEDIT] y Protégé [PROTEGE], entornos visuales que permiten la definición de ontologías de manera gráfica, comprendiendo tanto esquemas como individuos. Sus arquitecturas extensibles a través de plugins les han permitido mantenerse a tono con los requerimientos de la comunidad de usuarios. Protégé, por ejemplo, ha incorporado recientemente componentes que le permiten el manejo de modelos OWL. El segundo de los grupos permite el uso de bases de conocimiento desde una plataforma de programación. Es justo destacar que los desarrollos más importantes fueron realizados para Java. Jena [JENA] y Sesame [SESAME] son dos bibliotecas

que encontramos particularmente útiles. Proveen de funciones para el manejo y consulta de ontologías, organizadas tanto en modelos RDF-S como en OWL. Ambas pueden integrarse con motores de inferencia que permiten tanto la interpretación de los modelos anteriores como la incorporación de nuevas reglas.

Resumen

En el presente capítulo se introdujo la web semántica, que tiene entre sus metas encontrar una manera universal de representación de conocimiento, necesidad similar a la que encontramos para la definición de setpoints. Se introdujeron RDF, RDF-Schema y OWL, lenguajes planteados para tal fin. Se explicó el concepto más general de ontologías, que busca el mismo objetivo sin tener en principio dicha extensión de la WWW en mente. Se hizo por último una breve reseña de algunas herramientas que implementan las teorizaciones anteriores.

AOP + Semántica: SetPoint!

Introducción

Explicamos en los capítulos anteriores las raíces y los conceptos básicos de la disciplina conocida como Aspect Oriented Programming, enfatizando su nacimiento como un paso más en la búsqueda de mejores técnicas de separación de incumbencias. Describimos luego las herramientas más significativas que implementan dichas ideas, desembocando en el análisis de la "AOSD-Evolution Paradox". Concluimos en la necesidad de los así llamados pointcuts semánticos para lograr que AOP traspase la barrera de lo experimental y se convierta en una disciplina con aplicaciones prácticas. Se plantearon por último modelos de representación de conocimiento propuestos para problemáticas similares.

En este marco se desarrolló la presente tesis, que tiene como objetivo aportar un grano de arena en torno a la discusión planteada sobre la real utilidad de las herramientas orientadas a aspectos, a través de la construcción de un framework que implemente el concepto de pointcut semántico. Esto último no puede ser logrado si se lo plantea como un punto aislado: está íntimamente ligado a otras cuestiones relacionadas con la plataforma AOP sobre la que se lo implemente. En el transcurso de este capítulo se intentan explicar estas consideraciones, los lineamientos generales seguidos en la elaboración del framework propuesto y los experimentos luego realizados sobre él.

Rememorando antiguas disertaciones: sobre paradigmas y dominios

En el capítulo que bautizamos "Rumbo a SetPoint", adelantamos ya nuestro punto de vista acerca de la relación entre AOP y los distintos paradigmas de programación: si bien es cierto que los conceptos de la disciplina podrían ser aplicados sobre cualquier paradigma, nuestro enfoque estará sesgado a la orientación a objetos.

Otro concepto determinante en nuestra concepción de AOP es que se trata de entremezclar dominios ortogonales, no de buscar nuevas unidades de modularización para elementos de un mismo dominio que extiendan el paradigma de objetos.

La nostalgia ataca de nuevo: Acerca de las perspectivas

También en "Rumbo a SetPoint" comentábamos que creemos natural definir los pointcuts en base a perspectivas del programa diferentes al código en sí. Necesitábamos para ello dos elementos fundamentales: primero, una forma de modelización de semántica que nos permita representar cualquier eventual perspectiva; segundo, mecanismos para relacionar el código con conceptos de los universos definidos. Aunque no hayamos aún explicado en detalle la respuesta a ambas preguntas, sí dimos los lineamientos de nuestras soluciones: RDF/OWL y anotaciones en el código, respectivamente. A continuación profundizaremos sobre estas decisiones.

RDF/OWL como modelos semánticos

El surgimiento de las ontologías como mecanismo formal de representación de conocimiento y la posterior gestación y uso de RDF y OWL como lenguajes para el desarrollo de la web semántica, nos dieron una base sobre la cual apoyarnos a la hora de concebir formas de representación de las diferentes perspectivas. La adopción de los mencionados formatos como estándares, la cantidad de trabajos publicados sobre ellos, y, sobre todo, su aplicación práctica en cientos de ontologías, nos dan elementos suficientes para considerar que puede ser un primer approach posible en vistas del objetivo buscado.

¿Será también el mejor? Difícil de contestar antes de tener experiencia práctica en la materia. Potencialmente, cualquier dominio puede ser representado mediante el enfoque minimalista de estos lenguajes: basta con describirlo estructuralmente a través de la identificación de sus conceptos y relaciones. Esta última oración deja entrever sin embargo un potencial problema: ¿bastará con las definiciones estructurales, o será necesario en algún momento reflejar la dinámica de estos universos para poder predicar sobre condiciones de esta índole en la definición de setpoints? Imaginemos, por ejemplo, un sistema de control de ferrocarriles diseñado en base a una máquina de estados finitos (FSM) temporizada. Si transcurridos 2 minutos de la llegada a una estación el tren no se ha aún retirado, debemos registrar la situación en un log. Necesitamos predicar sobre esta condición de la FSM. Debimos, por tanto, definirla mediante RDF/OWL. En principio, esto no sería posible… El poder expresivo de los modelos anteriores podría ser mayor si permitieran definir axiomas que provean nuevas capacidades de inferencia: “Tìo(X, Y) **ó** (∃ Z) / Padre(Z, Y) ∧ Hermano(X, Z) “ o “Mover(A, Lugar) **è** OcupanteDe(LugarDe(A)) = Vacio y OcupanteDe(Lugar) = A”. Hay cuestiones sin embargo, en este caso las relacionadas con constraints temporales, que no podremos reflejar. Surge entonces la duda: ¿Es mejor dirigirse hacia un modelo universal de representación de conocimiento, o será conveniente limitar la cantidad de perspectivas sobre las que predicar, especificando el lenguaje de representación de cada una de acuerdo a sus propias necesidades?

Creemos necesario evaluar los pros y contras de ambas opciones en experiencias prácticas, en donde se apliquen las ideas de ambas, y se evalúe cuál resulta finalmente más apropiada para cada circunstancia. ¿El caso anterior de los trenes representa un escenario forzado para el ejemplo o será una situación que se repetirá asiduamente en el ejercicio práctico? No queremos dejar de tener en claro que una travesía de mil millas empieza con un pequeño paso: nuestra tesis proveerá un framework concebido en base a la primera de las vertientes. Pretendemos verificar que sea apropiado al menos para una cantidad de circunstancias, validando su utilidad práctica al emplearlo sobre una aplicación testigo. Es tema para próximas investigaciones desarrollar ambientes que permitan trabajar sobre lenguajes de representación específicos, para poder así realizar luego comparación en base a elementos empíricos.

Anotaciones en el código y nacimiento de los *program elements*

Centramos hasta aquí la atención en los modelos de representación de las diferentes perspectivas a través de las que se mira un programa, concluyendo que para el presente trabajo utilizaremos RDF y OWL. Hay

un detalle todavía faltante: ¿Cómo relacionaremos estas vistas con el código de la aplicación, definido formalmente a través del lenguaje de programación? Para que esta relación no se convierta en algo inútil, y pueda ser usada finalmente por el weaver, debemos contar con algún mecanismo que nos permita consultar qué conceptos están relacionados con cada construcción del programa. Estamos generando *metadatos* sobre el código: información adicional sobre su contenido, similar a la *metainformación* que se genera en las páginas de la nueva versión de la web al anotarlas semánticamente. Las dos plataformas para desarrollos industriales más importantes hoy en día (.NET y J2EE) proveen o proveerán en el corto plazo[22] mecanismos para realizar lo propio: las anotaciones. En ambos se trata de asociar clases, métodos, propiedades y demás construcciones con objetos que se limitan prácticamente a representar conceptos, aunque pueden ser luego utilizados por otros frameworks como señales de aplicación de otras propiedades (recordar la descripción de las anotaciones como mecanismo primitivo AOP en ciertas plataformas de middleware).

En el presente trabajo proveeremos mecanismos similares. Las características específicas dependerán de la plataforma con la que se trabaje (describiremos este punto más adelante en este mismo capítulo). Hay algunas cuestiones sin embargo que son dignas de mención:

- Para que puedan anotarse las construcciones del programa y luego consultarse los metadatos asociados, es necesario que la plataforma cuente con un framework de reflection que facilite estas tareas, reificando clases, métodos, atributos, y demás construcciones.

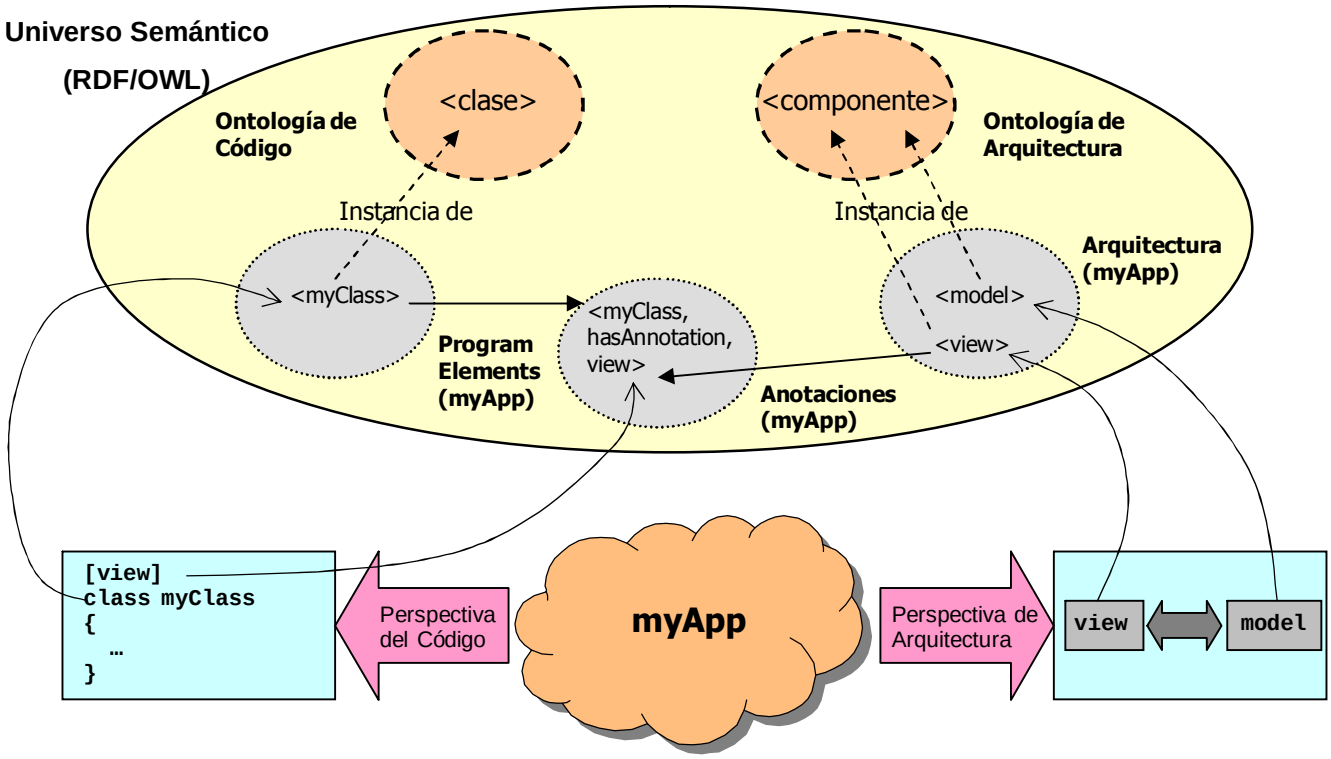
- Toda anotación debe estar asociada a un recurso RDF y tener un significado concreto: estamos hablando de verdaderas *anotaciones semánticas,* término acuñado por la literatura relacionada con la web semántica.

- Los predicados para verificar la inclusión de un joinpoint en un *setpoint* se realizarán sobre los modelos semánticos, trabajándose así en el nivel de abstracción pretendido. Ahora bien, los elementos del programa anotados con recursos RDF estaban descritos mediante otro formalismo, el lenguaje de programación: para poder predicar sobre ellos es necesario incluirlos de alguna manera en el universo anterior, expresarlos como recursos. La técnica denominada *logic metaprogramming* enfoca objetivos similares, intentando representar al programa como hechos lógicos que puedan ser luego referidos desde enunciados *à la* Prolog. [TYRUBA] y [SOUL] son entornos que implementan estas ideas para Java y Smalltalk, respectivamente; ambos han sido incluso utilizados para desarrollar entornos orientados a aspectos basados en la definición de pointcuts declarativos definidos en base a la estructura del programa (ver por ejemplo [DEDECKER]). Vale destacar también el trabajo de [WELTY] resaltando la importancia de mantener una base de conocimiento sobre el código. Nos inspiramos en ellos para representar el código fuente en RDF; definiremos recursos para identificar cada una de las construcciones del programa; declararemos para cada clase, atributo y método su correspondiente

[22] Las anotaciones para plataformas Java están aprobadas como recomendación. Existen hoy en día versiones beta de su implementación.

recurso RDF, al que denominaremos *program element*; indicaremos al motor de inferencia la semántica necesaria para cada una de las construcciones. Debe saber comprender, por ejemplo, que las anotaciones hechas sobre una clase valen también para sus métodos.

Resumiendo los ítems anteriores, podemos decir que para relacionar el código con otras perspectivas realizaremos anotaciones semánticas sobre el metamodelo que tendrán su contraparte en las ontologías definidas. Un último comentario que vale la pena realizar, por más que suene obvio en un principio: las anotaciones deben estar bien hechas. Su asociación al *program element* debe tener sentido, el significado debe ser el correcto. Caso contrario, los pointcuts carecerán de sentido. Esta tarea no será entonces trivial, habrá que prestar especial atención y cuidado. Creemos que dicho esfuerzo será recompensado con creces, tanto en la aplicación de aspectos como en las mejores características de trazabilidad que podrán llevarse adelante entre los distintos artefactos de un proyecto. Podemos encontrar un ejemplo de "malas anotaciones" en muchas de las páginas de la actual web semántica: el afán de atraer internautas hace que los sitios estén asociados con conceptos que no tienen ninguna relación con ellos, desvirtuandose así los modelos de conocimiento representados.



¿Y para el weaving, cómo andamos?

Nos concentramos hasta aquí en los conceptos que atañen al modelado semántico, parte esencial desde ya de SetPoint, el framework AOP propuesto. Todavía no hemos prestado atención, sin embargo, a la otra parte del problema: ¿Cómo haremos para entremezclar el código de los distintos dominios? ¿Cómo se combinarán aspectos y código base?

La primera pregunta que surge es cuándo haremos el weaving: la compilación y la ejecución aparecen como los momentos factibles. En [BOLLERT] se analizan las ventajas y desventajas de los así denominados weavers estáticos y dinámicos. Según él, los primeros son más eficientes en cuanto a temas de performance, mientras que los segundos son más flexibles, brindando posibilidades otrora inalcanzables. Si descubrimos un bug en producción, por ejemplo, podremos habilitar opciones de tracing a través de la definición de nuevos pointcuts y advices sin necesidad de recompilar y reinstalar la aplicación[23]. Decidimos priorizar este segundo beneficio. La performance es indudablemente una cuestión fundamental para pretender que AOP se convierta en una tecnología utilizable; sin embargo, creemos que el mejor rendimiento de los weavers estáticos es una cuestión relativa; consideramos que invirtiendo el tiempo apropiado en desarrollo será posible alcanzar niveles lo suficientemente buenos en los entornos dinámicos. Como bien lo dijo el bueno de Tony Hoare, “Premature optimization is the root of all evil”.

De la importancia de pararse sobre OOP (pero en dos patas)

Una segunda cuestión a dilucidar es el modelo de joinpoints que se adoptará. Dentro del paradigma de orientación a objetos, un programa no es más que un “c*onjunto de objetos que colaboran entre sí enviándose mensajes”*. El único momento que puede identificarse en la ejecución de una aplicación es, entonces, el envío y la recepción de dichos mensajes. Esta directiva se cumple relativamente bien en plataformas OOP *puras* (Smalltalk, SELF)[24], pero en ambientes híbridos esta visión minimalista se ve complicada por nuevas construcciones: condicionales, loops, manejadores de excepciones, constructores, etc…No todo se realiza mediante el envío de mensajes a un objeto. Para crear una instancia, por ejemplo, se usa una instrucción especial del lenguaje, *new* en el caso de C#, C++ y Java. Comenzaremos nuestro análisis desde la visión más pura:

Los joinpoints posibles serán el momento de envío y recepción de mensajes

[23] También en plataformas de weaving estático cabe la posibilidad de definir pointcuts a partir de condiciones dinámicas. Se podría argumentar que es factible la aplicación potencial de todo aspecto sobre cada joinpoint, basándolos en alguna condición dinámica que surja de opciones de configuración. De esta manera, sin embargo, no se está haciendo otra cosa que un weaving dinámico implementado de una manera engorrosa, limitado igualmente en el caso que se tuvieran que aplicar aspectos no considerados en un primer momento.

[24] Es posible también identificar momentos que se refieran a la acción de las máquinas virtuales correspondientes, pero no estamos limitando en este caso a una visión lo más “pura“ posible.

Luego, según la plataforma para la que sea desarrollado el framework identificaremos cuándo están representados estos momentos. Creemos indispensable que se respeten ciertas normas de programación para que esta definición sea realmente útil: debemos poder predicar sobre los joinpoints de manera en que se pueda declarar los pointcuts con cierta coherencia; de nada sirve que cada método esté anotado semánticamente si luego él mismo realiza todas las tareas que tengan que ver con la facturación, el control de stock, el envío de avisos y el mantenimiento de libros contables; será imposible de esa forma definir en dónde aplicar un aspecto. La separación de incumbencias debe comenzar antes, factorizando correctamente las responsabilidades dentro de cada dominio.

Llegado el momento, desde ya, haremos también un balance entre el enfoque minimalista y las capacidades de la plataforma: mecanismos como el *doesNotUnderstand* [LIU] para el envío de excepciones entran sin duda en las definiciones anteriores, y pueden ser incorporados en los predicados de los pointcuts sin necesidad de extensión alguna. En entornos híbridos deberá contarse además con métodos que permitan la aplicación de aspectos sobre el control de excepciones, y será de suma utilidad que la plataforma AOP provea de una forma de hacer esto, pese a que atente contra la simpleza de la visión minimal.

Reificando a la una 1, a las 2, a las 3…

Ya hablamos del weaving dinámico y del modelo de joinpoints…. ¿pero cómo instrumentaremos estas ideas? Daremos una respuesta con aroma a tautología: ¡A través de un **weaver**! Si, pero… ¿Qué es el weaver? En un mundo donde todo son objetos, no tenemos muchas opciones: un objeto. Su responsabilidad será, continuando con las frases redundantes, implementar el weaving entre aspectos y código base: durante la ejecución de un programa se determinará, para cada joinpoint, qué aspectos corresponde ser aplicados, y se dictaminará de alguna manera su eventual ejecución junto con la del mensaje en curso.

Más allá de aclarar algunos conceptos, la afirmación anterior dispara nuevos interrogantes. El primero se refiere al tan repetido término **joinpoint** o *punto bien definido durante la ejecución de un programa*. ¿Cómo se dará cuenta el weaver sobre cuál estamos parados? La respuesta no varía de la dada en el párrafo anterior: el mismo debe estar representado por un objeto, cuya misión es, justamente, representar un envío de mensajes en particular. Debe conocer por tanto emisor, destinatario, selector y argumentos del mismo, y estar capacitado para ejecutarse y devolver eventualmente un valor de retorno. Cada uno de éstos será a su tiempo enviado al weaver para que administre su ejecución, relacionándolo con los aspectos correspondientes.

Ahora bien, ¿quién determina qué aspectos aplicar sobre cada joinpoint? Bueno, primero tenemos que determinar a qué **pointcuts**[25] pertenece, para luego poder relacionar a partir de los **advices**[26] qué aspectos

[25] A partir de aquí usamos indistintamente los términos pointcut o setpoint, asumiendo que la definición será siempre semántica.

[26] Recuerdse que en AspectJ un advice contiene no sólo la relación pointcut-aspecto, sino también el *crosscutting code* que corresponde ejecutar. En nuestra definición incluimos este último dentro del aspecto. No discutimos aquí cómo se conoce el protocolo de cada aspecto, tema desarrollado principalmente en la sección sobre la versión de SetPoint .NET, en donde se introducen el concepto de trigger.

corresponde aplicar…. ¡y para ello estuvimos hablando buena parte de la tesis sobre modelos semánticos! Implementaremos entonces las colaboraciones que representen estos modelos.

Supongamos ahora que, en un momento dado, debemos aplicar los aspectos A y B sobre un joinpoint. ¿Qué hacemos? ¿Cuál va primero? ¿Qué pasa si uno de los aspectos corresponde a la encripción de una información pero por alguna razón la tarea falla y no puede completarse? ¡Mamita! ¡Debemos detener la ejecución del joinpoint, o la privacidad de nuestros datos correrá peligro! ¿¡Reseteamos!? No seremos muy originales en la manera de encarar una solución a este problema: simplemente crearemos un objeto cuya responsabilidad sea la coordinación de la ejecución de los diferentes aspectos, tomando en cuenta esas y otras cuestiones. Lo llamaremos *política* (de aplicación de aspectos).

Llegamos finalmente a la pregunta del millón… Hablamos de ejecutar los diferentes aspectos… Supongamos que conseguimos recién un maravilloso framework de auditoría, y quiero comenzar a utilizarlo. ¿Cómo hace el weaver para conocer su protocolo? ¿Cómo se puede interactuar con él? Denominaremos aspectos a los objetos que hacen las veces de adaptadores a la usanza de [GAMMA]: ajustarán las interfaces de cada dominio ortogonal adaptándola a las necesidades del weaver[27].

Seguimos en esencia una misma directriz para la resolución de los temas que se fueron presentando: *reificar* cada concepto identificado, convertir en objeto las ideas y los términos que pertenecen al universo del weaving. Integrando estas nuevas abstracciones al ambiente logramos crear un entorno que cumple con las características buscadas, presentando a la vez un diseño lo suficientemente flexible para permitir cambios eventuales no contemplados en un primer momento. En las secciones siguientes se presentará el detalle de la implementación de las ideas introducidas en este apartado, clarificándose así la propuesta.

De una buena vez a los bifes: SetPoint

Los lineamientos básicos del framework AOP están ya definidos. *El diablo está en los detalles*, se suele decir, y es por ello que la implementación de este entorno permitirá profundizar y validar las cuestiones planteadas. La tarea aislada de implementar el ambiente no será suficiente: para poder cerrar un primer balance, será necesario el desarrollo de al menos una aplicación que lo utilice; creemos indispensable testear el modelo conceptual contra un escenario práctico.

En una primera etapa implementamos SetPoint sobre Squeak [SQUEAK1], ambiente Open Source de Smalltalk, extendido mediante el framework de metaclases MetaclassTalk [MTALK]. Sobre un juego de Senku [SENKU] desarrollado por nosotros con anterioridad[28], totalmente ajeno a cualquier futuro requerimiento de ampliación, instrumentamos aprovechando la nueva plataforma los siguientes requerimientos:

[27] Notese entonces que a nivel implementación estaríamos usando al objeto aspecto para representar dicho adaptador, y no la dimensión en su totalidad. Creemos sin embargo apropiada esta denominación ya que dicho objeto es el que actúa de puerta de entrada a la misma.

[28] En los anexos se incluye el diseño de detalle del Senku; fue desarrollado como parte de la cursada de la materia Programación Orientada a Objetos de la Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.

1. *Tracing*: Debe mantenerse un registro de los eventos que se hayan producido durante la ejecución del programa y hayan sido comunicados desde componentes del modelo a componentes de la vista.

2. *Performance*: Debe optimizarse el cálculo de las posibles jugadas mediante el caching de los movimientos que pueden realizarse desde un casillero (para analizar un posible movimiento, hay que considerar dos factores: las posibles casillas destino y la existencia o no de fichas sobre ellas).

3. *Distribución*: Los objetos del modelo y los de la vista deberán funcionar en procesos separados.

El objetivo de esta primera etapa fue validar la factibilidad de implementación de un weaver a través de la reificación de los conceptos de la disciplina. Se consideró por consiguiente a Smalltalk como un ambiente idóneo para llevar adelante esta tarea: su enfoque "puro" nos mantendría dentro de la visión minimal del paradigma, abstrayéndonos de eventuales dificultadas originadas en plataformas híbridos. Se planteó un modelo de representación semántica simple: la meta no estaba puesta en profundizar sobre estos conceptos, sino en hacer un primer acercamiento. El producto desarrollado y las conclusiones del trabajo se describen en el capítulo siguiente.

El énfasis puesto en la real usabilidad de la tecnología AOP basada en semántica nos llevó a pensar que una segunda versión debería estar orientada a un ambiente que pudiera ser alcanzado por una comunidad de desarrolladores más amplia. Nos inclinamos por .NET, como consecuencia de esta y otras dos razones fundamentales: por un lado, el soporte nativo de metadatos a través de anotaciones nos brindaría las condiciones necesarias para relacionar un programa con un modelo semántico de una manera natural, y, por el otro, la condición de plataforma multilenguaje nos permitiría explorar dificultades no enfrentadas en el caso anterior. Siempre en la búsqueda del mismo objetivo, fueron mejoradas las capacidades de inferencia del modelo semántico y se aumentó el poder expresivo en la definición de pointcuts. Se incorporaron también mejoras aprendidas en la experiencia anterior.

El experimento realizado fue equivalente al desarrollado en Smalltalk: se programó el Senku[29] y se aplicaron luego los aspectos de tracing, performance y distribución. El detalle de los trabajos ejecutados y las mejoras en la capacidad de inferencia y predicado son explicados en los capítulos subsiguientes.

Resumen

Durante el capítulo se discutieron algunos conceptos generales sobre los que se basará SetPoint: será una plataforma orientada a aspectos cuyo modelo semántico se apoyará fuertemente en RDF/OWL, pensada para trabajar por sobre OOP; la construcción del framework se hará reificando los conceptos de la disciplina. Se explicaron luego las plataformas y los experimentos que se llevaron a cabo. En los siguientes capítulos damos el detalle de estas tareas y sus respectivas conclusiones.

[29] En el mismo anexo en donde se muestra el diseño de detalle se explican las diferencias entre la versión Smalltalk y la .NET

SetPoint v0.1: El prototipo en Smalltalk

Introducción

En el capítulo anterior describíamos las diferentes etapas desarrolladas a lo largo de la tesis, dando los lineamientos generales de la herramienta propuesta. Es el turno ahora de comenzar con el detalle de nuestra primera versión: un prototipo de SetPoint en Smalltalk, implementado con el objetivo de validar la factibilidad de un weaver basado en la reificación de los conceptos de AOP. Se aprovechó también como primer acercamiento a la idea de pointcut semántico; pese a no profundizarse en el tema, se validó su utilidad en una aplicación práctica.

Los requerimientos para la plataforma y el experimento propuesto se explican en el capítulo previo: aplicar características de tracing, performance y distribución a un Senku implementado con anterioridad[30], ajeno originalmente a dichas necesidades. Damos primero una breve reseña de Squeak/MetaclassTalk, plataforma utilizada para el desarrollo. Continuamos detallando la arquitectura del framework AOP propuesto, narrando en los apartados posteriores la forma en que fue utilizado para resolver el problema planteado. Consideramos importante repetir que esta versión consistió en un primer acercamiento al problema, y, por lo tanto, el modelo planteado será ligeramente diferente a la versión definitiva. Finalizamos el capítulo con las conclusiones obtenidas a partir de la experiencia anterior.

La plataforma de desarrollo: Squeak y MetaclassTalk

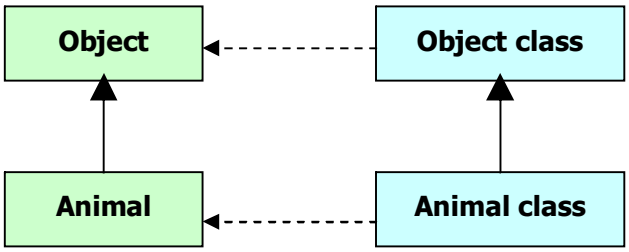
Una breve definición de Squeak nos dice que es una …"implementación abierta y altamente portable de Smalltalk-80, cuya máquina virtual está escrita enteramente en Smalltalk, haciéndolo fácil de debuggear, analizar y modificar"… [SQUEAK2]. Squeak puede ser descargado gratuitamente desde su homepage (http://www.squeak.org), y la imagen que uno adquiere incluye la implementación de la máquina virtual y las clases del ambiente, de forma tal de permanecer accesibles para su análisis y modificación. Agrega a estas cuestiones niveles de performance que, si bien no llegan a igualar los de los Smalltalks comerciales, redundan en niveles aceptables. Este conjunto de características lo convierte en un ambiente que ha gozado del visto bueno de la comunidad académica y de desarrolladores independientes desde su gestación: claramente resulta idóneo para la exploración de nuevas áreas de conocimiento.

MetaclassTalk [MTALK] está compuesto de una imagen de Squeak modificada que enriquece el manejo de metaclases en el entorno. Reifica valiéndose de ellas una serie de mecanismos (envío y recepción de mensajes, method lookup, acceso a variables de instancia, alocación de memoria) que facilitan la exploración de nuevos paradigmas, conceptos y extensiones del lenguaje.

En Smalltalk, todo es un objeto, y todo objeto es instancia de una clase. Ergo, toda clase es también un objeto, y como tal debe ser instancia de una clase. Se conoce a estas "clases de clases" como *metaclases*. En

[30] Descrito en el tercer Anexo

la versión estándar, se construye automáticamente una jerarquía paralela a la que puede accederse enviando el mensaje *class* al objeto adecuado[31]:



Con MetaclassTalk las metaclases pasan a estar también en primer plano. Se agrega un mensaje que permite especificar que objeto hará las veces de metaclase para una clase; aparece la denominada StandardClass, que incluye los métodos que manejan las funciones antes descritas, entre ellas el envío de mensajes a través de los selectores `send:selector from:sender to:receiver arguments:args superSend:superFlag originClass: orginCl` y `receive: selector from: sender to: receiver arguments: args superSend: superFlag originClass: original`; se modifica también el System Browser para que aparezca el detalle de la metaclase en el panel correspondiente, entre otros tantos cambios introducidos al ambiente.

MetaclassTalk fue utilizado con antelación a Setpoint para la implementación de ambientes experimentales AOP: se basaron bien en la asignación de metaclases que "aspectualicen" el comportamiento de sus instancias (ej. creando una metaclase TraceClass, especialización de StandardClass, que ante cada recepción de mensaje emita un mensaje en el Transcript) o bien en el uso de cadenas de "metaobjetos" que cumpliesen una tarea equivalente [MTALK-AOP]. Aparecen en estas soluciones, más allá de cuestiones menores, dos diferencias importantes con nuestra propuesta:

- No se cuenta con buenos mecanismos de cuantificación. La asignación de dichos metaobjetos a una clase equivale a enumerar uno a uno los joinpoints sobre los que corresponde aplicar un aspecto.

- Los aspectos se implementan como entidades pertenecientes a otro nivel de abstracción (metaobjetos, en este caso), y no como dimensiones paralelas.

Pese a estas diferencias, hay también una cantidad de aristas en común. Entre ellas, la que más nos interesa destacar es que en ambos casos se trata de ambientes orientados a objetos modificados que constituyen verdaderos frameworks *aspect-aware*: no existió la necesidad de introducir nuevas construcciones al lenguaje ni de hacer cambios al paradigma para incorporar el uso de aspectos.

[31] Recordemos que este modelo de Metaclases es el aplicado por Smalltalk, y no debe ser compartido necesariamente por otras herramientas

Arquitectura de SetPoint

Como se deja entrever a lo largo del trabajo, identificamos dos áreas bien diferenciadas al momento de plantear el framework buscado: el weaving y el modelado semántico. Estos dominios no permanecen aislados en el universo: durante el proceso de weaving, desearemos hacer consultas sobre la semántica de la aplicación en curso para poder determinar qué aspectos corresponde aplicar. Designaremos a los advices como objetos responsables de la traducción entre ambos mundos.

En el cuarto anexo de la tesis se incluyen los diagramas que describen el diseño de detalle del framework planteado; se recomienda su lectura para una mejor comprensión de los conceptos que se plantearán a lo largo de los apartados siguientes.

Reificación de los joinpoints

Según la concepción de AOP ya explicada, la recepción y el envío de mensajes están reificados a través de joinpoints. Se aprovechan los respectivos métodos de las metaclases para llevar esto a cabo. Se define AspectualizedClass como ancestro de todas las metaclases de los objetos sobre los que se desee aplicar aspectos; se redefinió el comportamiento de la recepción y el envío de mensajes para instanciar el joinpoint correspondiente y delegar su ejecución al weaver.

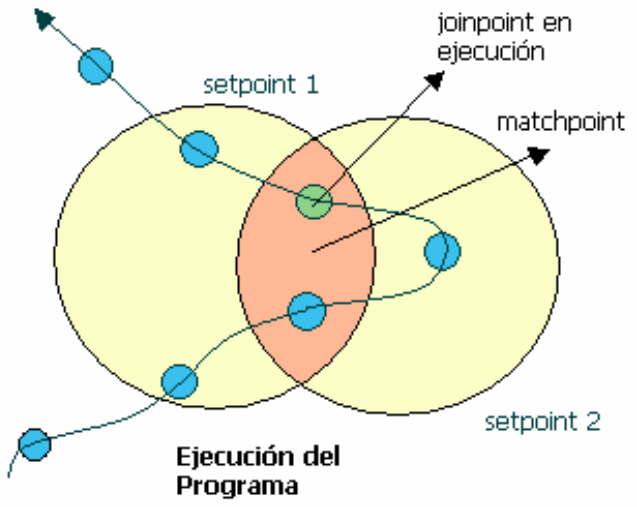
La implementación inicial no brinda en principio el grado de obliviousness buscado: es indispensable especificar la mencionada AspectualizedClass como ancestro de las metaclases de los objetos afectados. Podría implementarse en futuras versiones algún mecanismo automático para ejecutar esta tarea.

Weaving

Lo primero que debe hacer el weaver al interceptar la ejecución de un joinpoint es verificar qué aspectos corresponde aplicar. Ello se decide evaluando la semántica del mismo en función de los setpoints definidos. Se produce el choque de civilizaciones antes mencionado: el universo semántico se encuentra con el del weaving. Tenemos, por un lado, un conjunto de recursos RDF, que incluye ontologías y *program elements*; encontramos por el otro al universo de *setpoints (pointcuts semánticos)*; el joinpoint en curso será evaluado en este contexto.

No profundizaremos por el momento en la manera en la que se se decide la pertenencia a un setpoint: basta saber que el responsable es el mencionado universo de *setpoints*[32]. Podría ocurrir que el joinpoint en cuestión pertenezca a más de un pointcut semántico; llamamos por ello *MatchPoint* al objeto que representa esta posible intersección. No es ésta una abstracción relevante dentro del mundillo de la orientación a aspectos; preferimos para esta primera versión sin embargo reificar el concepto, y verificar su utilidad en la práctica.

[32] En esta implementación dicho universo de setpoints incluye también a los advices; en la implementación en .NET se intenta clarificar la terminología utilizada



Ya determinamos qué aspectos aplicar... resta definir cómo. Un objeto titulado *PolicyBroker* es el encargado de decirlo: conoce bajo qué política corresponde aplicar el conjunto de aspectos que conforman un matchpoint. ¿Tienen que mantener un orden determinado? ¿Qué pasa si alguno falla? ¿Son excluyentes? En esta implementación sólo existe la denominada *ChainPolicy*, que indica que los aspectos deben ser aplicados en secuencia, sin ningún orden dado. Cada aspecto define código a ejecutar antes y después de un joinpoint. Supongamos que corresponde aplicar los aspectos 1, 2 y 3. Primero ejecutamos el código previo de 1, luego el de 2 y luego el de 3, para terminar ejecutando el código del joinpoint en sí. Luego ejecutaremos el código posterior de 3, el de 2 y el de 1, sucesivamente.

Algunas cuestiones que vale la pena remarcar:

- Para la ejecución final del joinpoint definimos un aspecto llamado “functional aspect”, que representa la adaptación que se hace del dominio para que pueda convivir con el weaving. Lo único que hace en la práctica es decirle al joinpoint que se ejecute. Realizamos esto para que la política interactúe siempre con aspectos; nos pareció que resultaría más claro y permitiría manejar la ejecución de forma uniforme.
- No siempre se van a ejecutar ni todos los aspectos ni el código del joinpoint. Antes de continuar con la ejecución de la cadena, se le consulta al aspecto recién llamado. Esto fue necesario, por ejemplo, para el desarrollo del aspecto de cache: si el mensaje ya había sido cacheado, no corresponde que se continúe con la ejecución, sino que se devuelve el valor almacenado. En caso contrario, sí se permite seguir.
- Los aspectos pueden modificar el valor de retorno de un joinpoint. Lo mismo sucede con los argumentos que van siendo enviados a los aspectos que continúan la cadena. Encontramos un ejemplo de lo primero en el aspecto de cache: si vamos a cortar la cadena, tenemos que setear como valor de retorno el dato que almacenamos

No profundizamos en la presente tesis en el tema de las relaciones entre aspectos. Creemos, sin embargo, que el haber identificado y convertido en objeto a las políticas de aplicación brinda un excelente marco para permitir futuros avances en la materia.

El universo semántico

El modelo semántico utilizado en este prototipo se basa en RDF: la clase Resource permite instanciar objetos que imitan, justamente, a los recursos, ya sea los básicos o las triplas <sujeto,predicado,objeto>. No se modelaron ni RDF-Schema ni OWL. Se adoptó por lo tanto un modelo ad-hoc para poder implementar el concepto de universo semántico descrito en el capítulo anterior. Se representa cada perspectiva mediante una instancia de la clase Ontology, que no es más que un contenedor de recursos. Los program elements son representados también a través de instancias de la clase Resource. Las anotaciones, por último, son nuevas triplas formadas por un recurso de cada uno de los conjuntos anteriores, relacionados mediante un predicado universalmente conocido, representado obviamente por medio de una instancia de Resource.

Volvamos por un instante al problema del weaving: teníamos un universo formado por todos los advices definidos en el entorno; al evaluarse un joinpoint, cada uno deberá determinar si corresponde o no su aplicación evaluando para ello cada uno de sus pointcuts semánticos. Estos no son más que predicados lógicos: se utilizó en esta primera versión la lógica proposicional como modelo de consulta, reificada a través de clases que incluyen And, Or y Not como únicas operaciones. Cada proposición necesita una valuación para poder definir su valor de verdad: se utiliza como tal al universo semántico definido previamente. Nada de esto serviría, sin embargo, si no se tuviera acceso también a la información del contexto de ejecución que contiene el joinpoint: este es el momento principal en donde el advice funciona como traductor entre los dos mundos.

Profundicemos ahora en cómo se utiliza la valuación anterior: un predicado es bien una composición de predicados o bien un predicado atómico; los primeros se evalúan mediante la disyunción, conjunción o negación de los valores de verdad de sus partes. ¿Cómo se evalúan estas últimas? El responsable de la tarea es el universo semántico. La veracidad de una tripla depende de las reglas de inferencia definidas en dicho mundo. En este caso implementamos únicamente lo que denominamos “universo simple”, en donde la verdad es equivalente a la existencia explícita del predicado: no se realiza inferencia alguna, no se tiene noción de la semántica de las relaciones. Este modelo nos permitió, a pesar de su simpleza, cumplir satisfactoriamente con los objetivos buscados. El diseño permite, a su vez, realizar eventuales extensiones definiendo universos que implementen la semántica de RDF-Schema, OWL, o cualquier otro vocabulario.

Aplicando aspectos sobre el Senku

El único paso que nos resta es describir cómo fue utilizado el framework para satisfacer los requerimientos enunciados. Necesitamos seguir para ello los siguientes pasos:

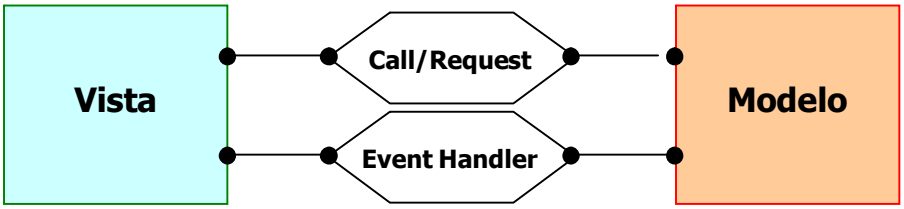
1. Definir las ontologías a utilizar.
2. Anotar semánticamente el código.

3. Programar las dimensiones ortogonales y los respectivos aspectos que les permitan interactuar con el weaver.
4. Crear las metaclases explícitas de cada objeto del Senku.
5. Instanciar e incorporar al ambiente los advices y sus correspondientes pointcuts.
6. ¡Jugar al Senku!

Como última aclaración, vale decir que el énfasis del trabajo estuvo puesto sobre el ambiente construido, y no se le prestó mayor importancia a las herramientas con las que contara el usuario para su utilización, más allá de facilitarse sí lo mínimo indispensable. Es por ello que tuvimos que escribir directamente código Smalltalk para la definición de diversas cuestiones, como recursos, advices y pointcuts.

Ontologías utilizadas

Identificamos dos perspectivas diferentes que deberíamos modelar para la implementación de los aspectos requeridos. No las exploramos con rigurosidad, sino que nos limitamos a definir los conceptos que fueran necesarios para cumplir nuestra tarea. El primero surge de una visión del programa que denominamos "arquitectura" (inspirada en la arquitectura conceptual propuesta en [HOFMEISTER]), utilizada para estructurar luego el diseño de detalle. Se pensó en un esquema de tipo MVC, que podríamos representar según el siguiente gráfico:



Declaramos entonces la ontología de arquitectura, que incluye los recursos que representarán los conceptos de vista, event handler y modelo. Luego encontramos que, para poder escribir predicados para la aplicación de aspectos como la encriptación, el caching o la compresión, sería útil una ontología que predique sobre ciertas características de la información manejada. Denominamos a este modelo "ontología de la información". Se presenta a continuación el código utilizado para la definición de los modelos anteriores:

```
| ontoArq ontoInf |
ontoArq := Ontology name: 'Arquitectura'.
ontoArq addResource: (BasicResource description: 'Model' identifier: #model).
ontoArq addResource: (BasicResource description: 'View' identifier: #view).
ontoArq addResource: (BasicResource description: 'EventHandler' identifier: #eventHandler).

ontoInf := Ontology name: 'Información'.
ontoInf addResource: (BasicResource description: 'non referentially transparent' identifier:
#nonReferentiallyTransparent).
```

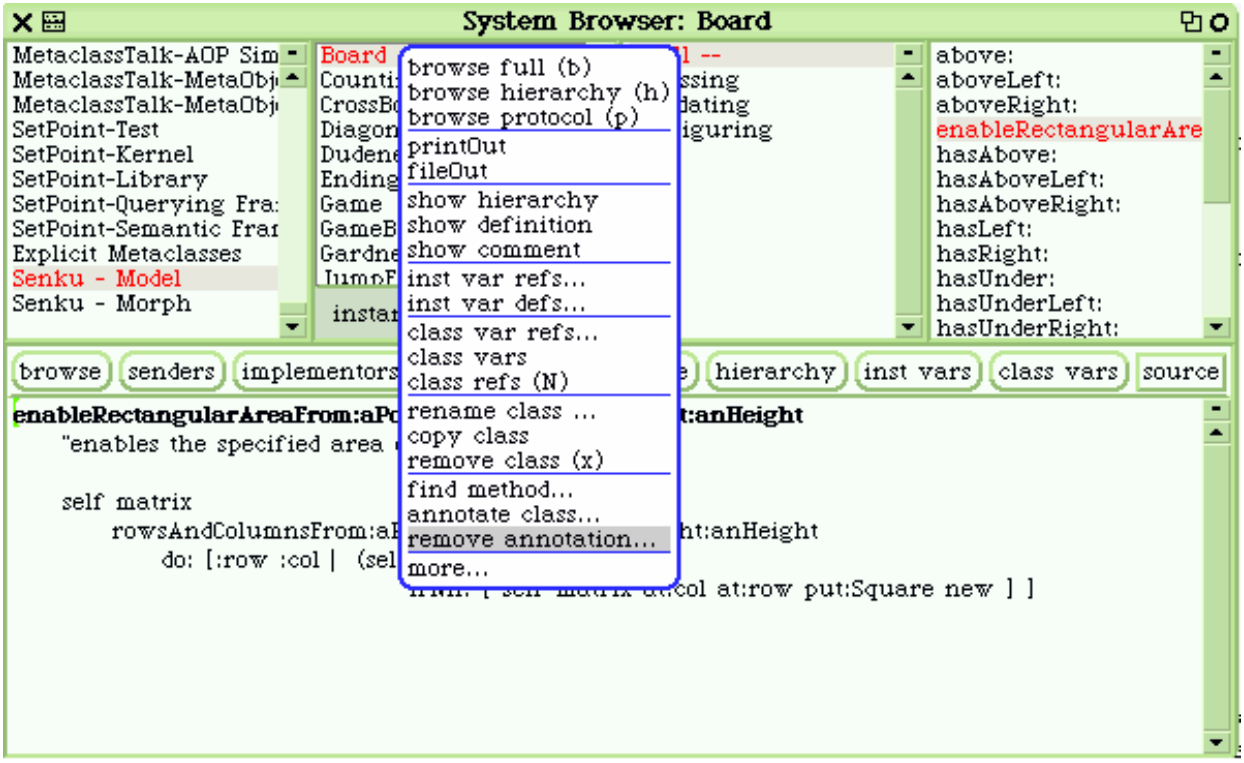
```
ontoInf addResource: (BasicResource description: 'referentially transparent' identifier:
#referentiallyTransparent).


Ontologies at: ontoArq name put: ontoArq.
Ontologies at: ontoInf name put: ontoInf.
```

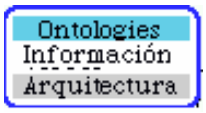
Anotaciones realizadas

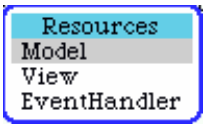
Las clases del Senku habían sido organizadas en dos grupos: *Senku – Model* y *Senku – Morph*. Todas las del primer grupo fueron asociadas al recurso *Model* de la ontología de Arquitectura, mientras que sobre las del segundo se hizo lo propio con *View.* Por otro lado, se relacionaron también los métodos de la familia triggerEvent: de la clase Object con el recurso *EventHandler* de la ontología "arquitectura". Quedaron de esta forma relacionados el código fuente y la perspectiva arquitectónica, según el objetivo de relacionar las diferentes visiones de un sistema planteado desde los primeros capítulos.

Para poder llevar a cabo esta tarea se aprovecharon los menús contextuales agregados al browser. Al desplegarse las opciones posibles sobre cada clase o método, surge ahora la posibilidad de anotarlos o borrar las anotaciones ya hechas:



Cuando se selecciona la opción para anotar un elemento del programa, se eligen sucesivamente la ontología sobre la que se quiere trabajar y el recurso con el que se relacionará la construcción en cuestión:





La segunda perspectiva con la que se deseaba relacionar el programa es la de la información. Nos limitamos en este caso a señalar los métodos pertinentes de la clase Board como referencialmente transparentes (above, left, hasLeft, hasUnder, etc…).

Definición de las dimensiones ortogonales

Necesitamos entremezclar tres dimensiones diferentes sobre el juego del Senku: tracing, caching y distribución. Lo primero que tenemos que hacer, entonces, es implementarlas y definir luego los aspectos que permitan integrarlas con el mundo del weaving. Dado que se trata de una prueba de concepto del framework AOP, las dos primeras se modelarían de una forma muy sencilla, a fin de no invertir demasiado esfuerzo en cuestiones que no hagan al foco del trabajo. Bajo estas premisas nacen las clases TranscriptLoggingAspect y SimpleCacheAspect.

Con la distribución el tema se puso peliagudo: no cabía la posibilidad de una “implementación sencilla”; deben enfrentarse en cualquier caso problemáticas complejas en sí mismas: definición de protocolos de comunicación, serialización de objetos, mantenimiento de la identidad en imágenes separadas, manejo de parámetros por valor y referencia. ¡Es una tarea digna para un experto, y no para nosotros, meros conocedores del universo del Senku! Ampliamos por lo tanto la imagen de Squeak/MetaclassTalk con el paquete SOAPOpera [SOPERA], que incorpora estas capacidades de manejo remoto de objetos, y definimos luego el aspecto (“DistributionAspect”) que hace las veces de adaptador y permite a este ambiente de trabajo interactuar con el weaver.

Creación de las metaclases

Para poder aplicar aspectos sobre el Senku, debemos explicitar las metaclases de cada una de las clases que lo conforman, especializando AspectualizedClass y asignando estos nuevos objetos como metaclases de las instancias correspondientes. Creamos un script para automatizar esta función: debería, por un lado, tomar en cuenta la existencia de métodos de clase, que deberían pasar a ser métodos de instancia en las metaclases respectivas; por el otro, habría que mantener la jerarquía de superclases correspondiente, a riesgo de tener problemas con los métodos anteriores.

No hubo mayores inconvenientes en ejecutar esta tarea sobre las clases del modelo; surgieron sin embargo problemas de compatibilidad entre los paquetes Morph (usado para la realización de los gráficos) y MetaclassTalk, que nos impidieron realizar lo propio sobre los objetos de la vista. Esto modificó nuestros planes de definición de setpoints y, consecuentemente, de aplicación de aspectos.

Declaración de pointcuts y advices

El último elemento que resta para que el weaver aplique correctamente los aspectos sobre el Senku es la declaración de los predicados que determinan los pointcuts afectados, y su posterior relación con los dominios ortogonales en cuestión a través de los advices.

Empecemos por el *caché*: ¿Dónde lo querremos aplicar? Habíamos dicho que es posible optimizar la performance de una partida evitando el recálculo de los valores que permanezcan inalterados durante su desarrollo. Este es el caso de la gran mayoría de los métodos del tablero: cada vez que se le pregunta si un casillero tiene una fila superior, por ejemplo, vuelve a efectuar las colaboraciones necesarias para averiguarlo. Quien aplique los aspectos, sin embargo, no tiene por qué conocer esta característica de la implementación; no tendría por qué estar al tanto siquiera de la existencia de una clase que representa al tablero. Sabe simplemente que los métodos que tengan la propiedad de ser *referencialmente transparentes* (es decir, que el valor de retorno dependa exclusivamente de los argumentos recibidos, independientemente de condiciones del contexto), son pasibles de ser optimizados. La declaración del setpoint se lleva a cabo construyendo la composición de objetos correspondiente. La declaración efectuada es la siguiente:

```
| nonVolatileSetPoint setPointCol advice |
nonVolatileSetPoint := SetPointStatement
	subject: #method
	predicate: (BasicResource description:'belongsTo' identifier: #belongsTo)
	object: (BasicResource description: 'non referentially transparent' identifier:
#nonReferentiallyTransparent).
setPointCol := OrderedCollection new.
setPointCol add: nonVolatileSetPoint.
advice := Advice on: setPointCol apply: SimpleCacheAspect.
SetPointUniverse instance advices add: advice.
```

Notese que como sujeto pueden utilizarse identificadores que representan propiedades del joinpoint (*#method* en el ejemplo anterior), de forma tal de acceder al contexto. Vale aclarar también que el advice se construye a partir de una colección de setpoints, interpretada como una disyunción, y no a través de un único predicado.

El segundo setpoint efectuará el logging de todos los mensajes que vayan de algún objeto del modelo hacia la vista mediante el mecanismo de eventos. Los inconvenientes antes mencionados nos quitaron la posibilidad de predicar sobre objetos de la vista; salvaremos esta situación aprovechando que únicamente en este caso se atraviesa el mecanismo de eventos, y definiremos el pointcut en base a ello:

```
| andSetPoint modelSetPoint eventSetPoint setPointCol advice |
modelSetPoint := SetPointStatement
	subject: #receiverClass
	predicate: (BasicResource description:'belongsTo' identifier: #belongsTo)
	object: (BasicResource description:'model' identifier: #model) .
```

```
eventSetPoint := SetPointStatement
        subject: #method
        predicate: (BasicResource description:'belongsTo' identifier: #belongsTo)
        object: (BasicResource description:'event' identifier: #eventHandler) .
andSetPoint := AndOperator with: modelSetPoint with: eventSetPoint.
setPointCol := OrderedCollection new.
setPointCol add: andSetPoint.
advice := Advice on: setPointCol apply: TranscriptLoggingAspect.
SetPointUniverse instance advices add: advice.
```

Nos resta únicamente incorporar la distribución. Nuevamente, el problema surgido de la incompatibilidad de los paquetes Morphic y MetaclassTalk nos llevó a redefinir el requerimiento original: los factories de movimientos serán ahora quienes deberán funcionar sobre una imagen, mientras que la vista y el resto del modelo lo harán sobre otra. Nos valimos de anotaciones ad-hoc para representar esta situación:

```
| notSetPoint andSetPoint recSetPoint sndSetPoint setPointCol advice |
recSetPoint := SetPointStatement
        subject: #receiverClass
        predicate: (BasicResource description:'belongsTo' identifier: #belongsTo)
        object: (BasicResource description:'Servidor' identifier: #Servidor) .
sndSetPoint := SetPointStatement
        subject: #senderClass
        predicate: (BasicResource description:'belongsTo' identifier: #belongsTo)
        object: (BasicResource description:'Servidor' identifier: #Servidor) .
notSetPoint  := NotOperator with:sndSetPoint.
andSetPoint := AndOperator with: recSetPoint with:notSetPoint.
setPointCol := OrderedCollection new.
setPointCol add: andSetPoint.
advice := Advice on: setPointCol apply: DistributionAspect.
SetPointUniverse instance advices add: advice.
```

Conclusiones

El desarrollo del framework orientado a aspectos en Smalltalk redundó en un importante acercamiento al mundo de los weavers, a la vez que nos permitió dar los primeros pasos en cuanto a la definición de pointcuts semánticos. Intentaremos resumir a continuación los elementos positivos y negativos más salientes que detectamos durante este proceso, a fin de tenerlos en cuenta para la segunda iteración del proyecto, la implementación del framework AOP en .NET:

- La implementación del weaver a través de la reificación de los conceptos de la disciplina nos permitió desarrollar un primer modelo del mismo, con el que se validó la factibilidad de dichas ideas. El trabajar sobre un ambiente con un enfoque minimalista como Squeak/MetaclassTalk, nos facilitó el trabajo.

- Es cierto también que, pese a su pureza teórica, la condición de versión beta de MetaclassTalk nos complicó en cuestiones de índole: el inconveniente con el paquete Morph es el más claro, llevándonos incluso a redefinir el experimento.

- Pese a permitir un mayor nivel de abstracción que si se hubiese predicado directamente sobre el código, la definición de los pointcuts semánticos no resultó del todo satisfactoria:
    - El ejemplo de distribución necesitó implementarse mediante anotaciones ad-hoc
    - El caso del tracing también se vio afectado por los problemas en el paquete Morph. Se utilizó además conocimiento sobre la estructura del programa: se sabía que el manejo de eventos estaba implementado a través de un método, por lo que pudo indicarse esta condición en el pointcut. Si el event handler hubiese sido representado a través de un objeto, habría sido necesario reescribir el setpoint. Ergo, éste no consiguió abstraerse completamente de la implementación.
    - Quizá el que consiguió mejores resultados fue el caché: la condición de transparencia referencial lleva una connotación semántica independiente del uso que se le vaya luego a dar; la explicitación de esta condición pudo ser luego aprovechada para mejorar la performance de la aplicación a través de la aplicación de aspectos.

"La base está": SetPoint v1.0

Introducción

Nos dedicaremos en este capítulo a presentar la implementación del framework de aspectos SetPoint en su versión .NET[33]. ¿Por qué decidimos no continuar en Squeak y migrar hacia esta plataforma? La razón principal es la necesidad de utilizar el ambiente AOP semántico en aplicaciones extra-laboratorio: la potencial llegada a una comunidad de usuarios masiva permitirá recibir feedback y validar la factibilidad de aplicación de los pointcuts semánticos. Existen además otros motivos que llevaron a que nos inclinásemos por este ambiente, en detrimento de otros que también cuentan con comunidades de usuarios numerosas (Java principalmente). Por un lado, la existencia de atributos[34] nos permite implementar de forma nativa y sin mayores dificultades la idea de anotaciones semánticas. Por el otro, el hecho de ser una plataforma *multilenguaje* nos enfrentará con una dificultad adicional, ausente en otros entornos. Vale mencionar que se incorporaron también, más allá del cambio de plataforma, una cantidad de mejoras, destacándose las derivadas del enriquecimiento del modelo semántico.

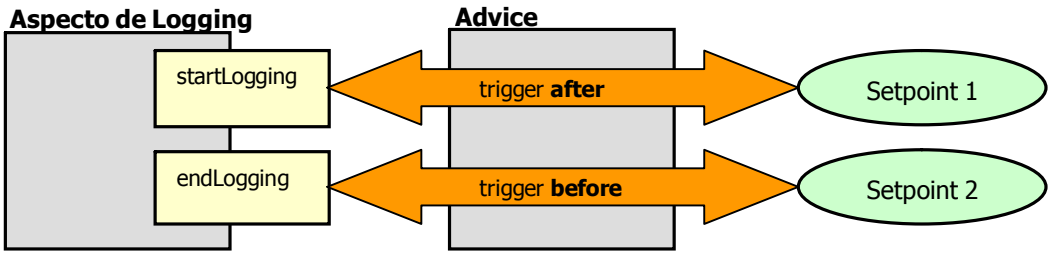
Lecciones aprendidas

Las principales falencias de la implementación original fueron encontradas en la extrema sencillez del modelo semántico y de reificación de advices y aspectos: el poder expresivo resultó insuficiente para poder predicar sobre otras perspectivas del sistema abstrayéndose de la estructura del programa subyacente. El caso del manejador de eventos descrito en las conclusiones del capítulo anterior representa el ejemplo más concluyente. Creemos posible relacionar este problema con la poca flexibilidad que se dio a la definición de la interfaz de los aspectos: todos debían respetar el protocolo de *before*, *after* y *cancel*, no pudiéndose especificar momentos temporalmente separados que les sean significativos. Volvamos al ejemplo del event handler: supongamos que deseamos loguear la hora en la que se publica un evento y la hora en que el mismo es notificado a sus suscriptores. Dependiendo de cómo esté implementado, estas tareas pueden realizarse en el contexto de un único joinpoint, o no. Pero el framework no provee formas de explicitar esto último. Es cierto que podría salvarse esta situación mediante la declaración de dos aspectos diferentes, en donde cada uno se hiciera responsable por una parte del proceso e interactúe luego con el otro para alcanzar el objetivo final, resultando en dos adaptadores diferentes para un mismo dominio. Volveríamos en este caso sin embargo a dejar implícita parte del modelo, con los potenciales problemas que esto trae aparejado. ¿Por qué no apuntar a una solución que los evite? Decidimos agregar un nuevo elemento al modelo para resolver esta cuestión: los *triggers,* mensajes que indican al aspecto la ocurrencia de un evento para él relevante. Un advice agrupará los

[33] En el primer anexo incluimos una descripción del framework .NET orientada a destacar las características que resultaron relevantes para la presente tesis

[34] Ver el anexo I para mayor detalle sobre los atributos

triggers que relacionan cada evento de un aspecto con los correspondientes pointcuts que lo disparan. En el ejemplo anterior, se relacionarán por ejemplo la publicación de un evento con el mensaje de inicio de logueo y la notificación con el de fin. Vale la pena destacar que, tal como antes lo definían los aspectos, cada trigger puede indicar la ejecución de código antes o después del correspondiente joinpoint, a la vez que puede decidir, ante determinadas circunstancias, que no deben continuar ejecutándose otros aspectos o el código base de ese joinpoint.



Habíamos aclarado ya en la implementación anterior que no era el objetivo en aquel momento profundizar sobre el modelo semántico utilizado; nos limitamos por ello a meros recursos RDF. Intentaremos avanzar en este caso un paso más, permitiendo el uso de RDF-Schema y OWL para la representación de ontologías, y reemplazando la lógica proposicional por LENDL, un lenguaje específico de dominio creado para facilitar la escritura de pointcuts, triggers y advices. Mejoraremos también el modelo a través de la aplicación de reglas de inferencia. Permitiremos a través de LENDL la definición de nuevas reglas.

Precursores .NET

SetPoint no es el primer framework orientado a aspectos desarrollado para .NET. Un puñado de implementaciones ha sido ya realizado; de cada una de ellas pudimos extraer ideas que nos resultaron de suma utilidad a la hora de tomar decisiones sobre las que construir nuestro ambiente. Vale aclarar que estamos hablando de cuestiones eminentemente técnicas: la base teórica fue edificada en función de los conceptos y herramientas descritas en las secciones correspondientes de la tesis.

El primer approach

Fuente: [Shukla]

Microsoft publica en este artículo de la revista MSDN Magazine la descripción de dos frameworks AOP implementados, uno para la tecnología COM y otro para .NET. En ambos se intenta seguir idénticos lineamientos adaptados a la tecnología en cuestión. Se dan las bases para desarrollar entornos que permitan extender las funcionalidades provistas por dichas plataformas de middleware, pero no se presta atención a dos conceptos esenciales de la programación orientada a aspectos: *quantification* y *obliviousness*.

En el caso de COM, se aprovecha la posibilidad de modificar el proceso de creación de objetos que brinda la función GetObject. A través de un objeto denominado *moniker,* se aprovecha cada instanciación para anteponer proxies que implementen los aspectos requeridos. Las relaciones componente-aspecto se enumeran en un archivo XML, sin ninguna posibilidad de predicar de una manera más compleja. Esto podría llegar a

mejorarse, aunque la granularidad de las relaciones impone restricciones desde un comienzo. La limitación más significativa no está sin embargo aquí, sino en la necesidad de especificar el *moniker* antes mencionado en cada llamada al método GetObject. El cliente no puede ser de esta forma ajeno a la aplicación de aspectos.

La versión .NET consiste en aprovechar la posibilidad de crear contextos de ejecución. Para ello, cada clase sobre la que se desee aplicar aspectos debe especializar a la clase ContextBoundObject, manifestándose la misma falta de obliviousness que en el caso anterior. Por otro lado, los aspectos a aplicar se indican a través de atributos especificados uno a uno sobre los objetos afectados, por lo que tampoco las características de cuantificación son las deseables. A diferencia del caso anterior, la no-reificación del weaver (representado antes a través del moniker) impide vislumbrar posibles mejoras.

AspectC#

Fuente: [AspectC#]

Inspirado fuertemente en AspectJ, esta herramienta ejecuta el weaving entremezclando el código fuente de los aspectos y del programa base a partir de los lineamientos dados por un archivo XML denominado *Aspect Deployment Descriptor*. Un parser construye en un principio los árboles de sintaxis abstractos (ASTs) correspondientes a los primeros, valiéndose de la funcionalidad provista por el namespace System.CodeDOM. Vale destacar que sólo es soportado un subconjunto del lenguaje C#, ya que existen construcciones aún no contempladas en dichas clases del framework. Luego el weaver los combina según lo especificado en el XML anterior, y se compila finalmente el resultado mediante las funciones provistas por el namespace System.Compiler. El modelo de joinpoints consiste de un subconjunto del de AspectJ. En el *Aspect Deployment Descriptor* debe especificarse por enumeración qué aspecto afecta a cada joinpoint: valen entonces las mismas críticas a la definición de pointcuts que las hechas en el capítulo pertinente de la tesis.

Cross-Language Aspect Weaving (CLAW)

Fuente: [CLAW1], [CLAW2]

CLAW es quizá la primera implementación ideada para servir en un ambiente multilenguaje, trabajando directamente a nivel IL. Su desarrollo no superó la versión preliminar, a causa del escepticismo que despertó en su autor la programación orientada a aspectos. El weaver aprovecha la posibilidad de interceptar ciertos eventos del framework mediante el Profiling API para anteponerse a cada compilación JIT e incluir las llamadas a los aspectos que deban afectar al joinpoint en curso. Define esto en base a dos archivos de configuración, uno que determina qué clase implementa qué aspecto y otro que define los diferentes advices. Los mecanismos para la definición de pointcuts continúan siendo precarios.

Loom .NET

Fuente: [LOOM1], [LOOM2]

En la versión estática, un aspecto corresponde a un *template* que especifica cómo generar el proxy de las clases a aspectualizar a partir de su correspondiente IL; mediante una sintaxis dada se define, por ejemplo, insertar un mensaje de log antes de cada método. Los antiguos clientes deben ocuparse de instanciar los

correspondientes proxies. Se determina cuáles generar seleccionando las relaciones aspecto-clase desde una consola de administración. Resulta interesante en esta implementación la idea de trabajar directamente sobre assemblies ya compilados, aunque los mecanismos de definición de aspectos y, nuevamente, los de cuantificación, resulten insuficientes.

En la versión dinámica, llamada Rapier-Loom, se definen aspectos especializando la clase Loom.Aspect. Los pointcuts se escriben mediante atributos sobre los respectivos aspectos. Puede predicarse sobre condiciones del nombre de los tipos afectados (es posible el uso de wildcards) y sobre sus atributos. Sin embargo, para que el weaving tenga lugar los potenciales objetos afectados deben instanciarse a través de la función Loom.Weaver.CreateInstance, afectando la condición de oblivious de sus clientes.

Otros

AOP# [AOP#] realiza el weaving valiéndose del profiling API. Implementa parcialmente la idea de conector esbozada en Caesar mediante un archivo XML en donde se definen los mapeos de métodos y parámetros. CAMEO [CAMEO] y AopDotNetAddIn [ADDINN] presentan un enfoque similar al de AspectJ, realizando el denominado weaving estático sobre código fuente. El primero modificó para ello el compilador C# de SSCLI [ROTOR]; sus puntos más destacables son el permitir el agregado de atributos a tipos preexistentes y el aislar en un XML la definición del orden de aplicación de los aspectos que coincidan sobre un mismo joinpoint. El segundo realiza lo propio a partir de un add-in que extiende la herramienta de desarrollo Visual Studio. Las definiciones de advices y pointcuts se leen también de un archivo XML. Otros proyectos de frameworks orientados a aspectos, algunos ya abandonados, son Aspect# [Aspect#], AOP.NET [AOP.NET], Weave.Net [Weave.NET], Aspect.NET [Aspect.NET] y AspectDNG [AspectDNG].

Esbozo de la solución propuesta

En el prototipo se validó la factibilidad de implementar un weaver a través de la reificación de los conceptos de la disciplina; se intentará seguir en la medida de lo posible dicho modelo de weaving, mejorado con el agregado de triggers. Todo debe comenzar por tanto por la incorporación de joinpoints al ambiente y la posterior delegación de su ejecución al weaver. Podríamos pensar en modificar los compiladores de la plataforma para que introduzcan el código correspondiente. Nuestra intención es, sin embargo, proveer de un framework AOP multilenguaje: esa opción no se presenta entonces como una alternativa valida. Para ser satisfecha deberían modificarse el/los compiladores de cada lenguaje soportado, lo cual, por un lado, no sería escalable, y, por el otro, no sería necesariamente factible por cuestiones de licenciamiento. La situación se complicaría aún más en la medida en que surjan nuevos lenguajes. Sería necesario por otro lado contar siempre con el código fuente del programa sobre el que se quieran aplicar aspectos, lo que restringiría las posibles aplicaciones del framework.

Descartamos también toda alternativa que implique que el programador del código base deba tener en cuenta requerimientos especiales para la utilización del ambiente de aspectos (cabe aclarar que no

consideramos la escritura de las anotaciones semánticas dentro de este grupo: su alcance se extiende más allá del framework propuesto; no es un requerimiento propio de AOP, sino que es una buena práctica que luego puede ser explotada por SetPoint). Una posibilidad que nos queda es hacer uso del Profiling API para agregar, justo antes de la compilación JIT de las clases, el código para llamar al weaver, a la manera de CLAW o JAC. Ciertas características insalvables de performance y seguridad originadas en su uso convierten a este enfoque en impracticable dentro de un ambiente de producción, lo que atenta contra nuestra intención de utilizar SetPoint en aplicaciones extra-laboratorio. Este enfoque resulta además muy poco práctico para el desarrollo: en el momento del weaving sólo se tendría un bloque de instrucciones IL sin referencias a otros elementos del programa; no aparecen reificados por ejemplo métodos referenciados a través de tokens, lo que dificulta su manipulación. Las funcionalidades aún no desarrolladas al momento de diseñar SetPoint y la escasa difusión que tiene en ambientes productivos nos llevaron a rechazar también la posibilidad de modificar SSCLI para agregar el manejo de aspectos directamente en el runtime. Optamos entonces por instrumentar el código sobre assemblies ya compilados, modificando las porciones de IL correspondientes. Este approach sirve para aplicaciones programadas utilizando cualquier lenguaje[35], a la vez que evita la necesidad contar con su código fuente.

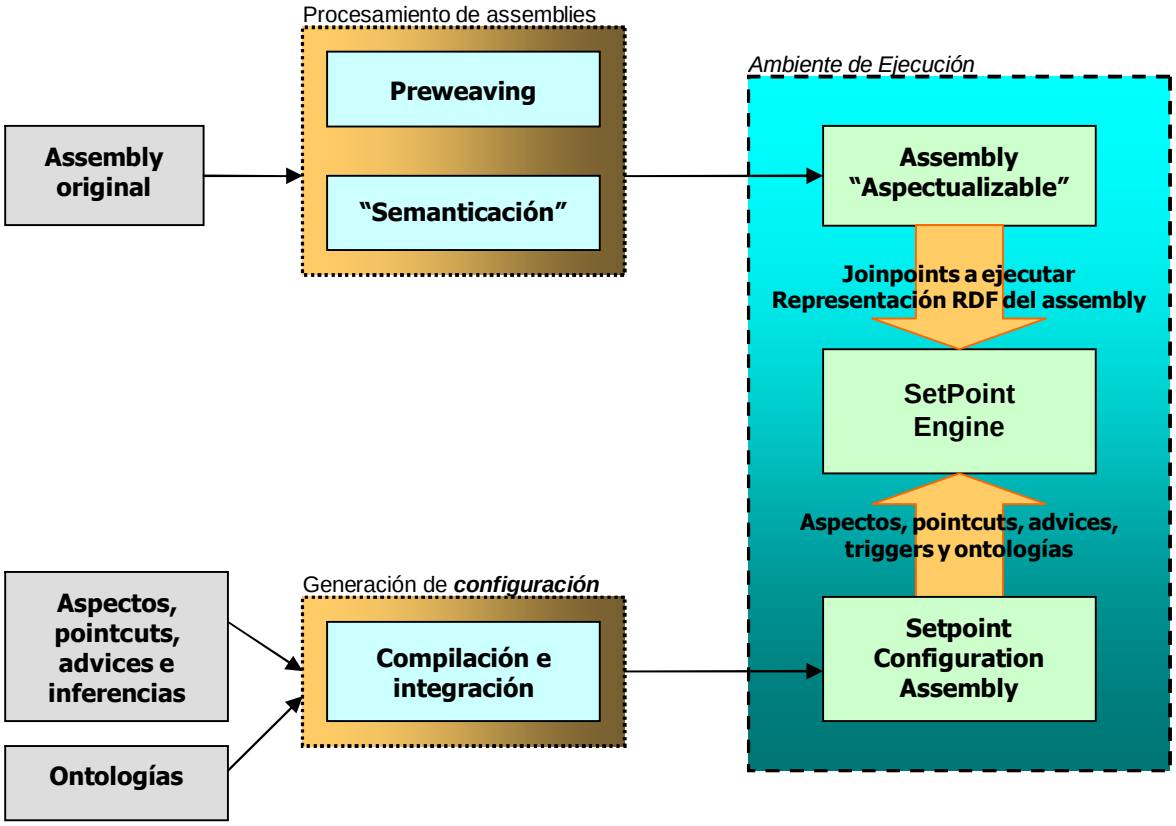
Otra de las tareas que debemos realizar, llevada adelante en la versión Smalltalk al escribir las anotaciones, es la creación de los *program elements*, los recursos RDF que representan al programa. Ambas tareas deben concretarse de forma previa a la ejecución de la aplicación: se definieron para ello dos procesos, denominados *preweaving* y *semanticación*. Su output consiste en un assembly funcionalmente idéntico al original, solo que al correr va construyendo los distintos joinpoints y delegando su ejecución al weaver (que integra un componente denominado *Setpoint Engine*). Incorpora también su representación RDF como recurso embebido.

Los últimos elementos que nos faltarían para completar todas las patas del ambiente orientado a aspectos al que nos venimos refiriendo son las declaraciones de aspectos, advices y pointcuts y las definiciones de los diferentes modelos semánticos, que incluyen tanto a las ontologías como a las reglas de inferencia definidas por el usuario. No importa en este momento la manera de definirlos: diremos por ahora que son incluidos en un assembly que debe acompañar la ejecución de una aplicación bajo el entorno SetPoint. Recordemos que las anotaciones semánticas se hacían valiéndose de atributos; estaban por tanto incluidas en la representación RDF del assembly original, y no aquí, en donde sí podríamos encontrar la ontología que defina los conceptos utilizados en las anotaciones[36].

[35] El trabajar sobre IL trae a la luz el cuestionamiento sobre la calidad de plataforma multilenguaje. Se parte de un modelo de joinpoints basado en el paradigma orientado a objetos, pero el código fuente podría estar escrito originalmente en imperativo o funcional… ¿Dependerá entonces del compilador la forma en que se traduzcan las potenciales anotaciones realizadas? ¿Cómo funcionará en la práctica esto? ¿Estaremos trabajando en realidad siempre con un modelo válido sólo para IL y para lenguajes similares? No es el objetivo de la tesis responder estas preguntas; pretendimos simplemente implementar un framework aplicable potencialmente a aplicaciones desarrolladas con cualquier compilador .NET.

[36] En la implementación presentada no es estrictamente necesario incluir la ontología que defina los conceptos utilizados para hacer las anotaciones: no se chequea dicha consistencia. En caso que se utilicen en las anotaciones conceptos no definidos en ontologías, se

En el gráfico siguiente intentamos esquematizar la descripción anterior. En los sucesivos apartados se describirá en detalle cada uno de los componentes que integran la solución propuesta.



Procesamiento de assemblies

Tiene como objetivo permitir que un assembly cualquiera pueda funcionar bajo el framework orientado a aspectos. Deben para ello ejecutarse dos tareas bien diferenciadas:

- *Preweaving*: Instanciar en cada envío de mensaje el correspondiente joinpoint y delegar su posterior ejecución al weaver.
- *Semanticación*: Crear la representación RDF del assembly e incluirla como recurso embebido.

Será necesario en ambos casos contar con la capacidad de acceder e interpretar el assembly. No hay mecanismos nativos[37] qué permitan instrumentar código IL. El namespace System.Reflection no tiene las

consideran no relacionados con otros elementos. En caso que pertenezcan a algún modelo de conocimiento, se pueden aprovechar entonces sí las capacidades de inferencia sobre los mismos.

[37] Al menos durante el período en que se realizó este trabajo. Próximas versiones del framework (mediante la incorporación del proyecto que actualmente se conoce como Phoenix) sí proveerían de esta funcionalidad.

capacidades necesarias, y el uso combinado de las herramientas ILASM e ILDASM implicaría un trabajo extra alejado de los objetivos de la presente tesis.

Una cantidad de proyectos se dedicaron específicamente a desarrollar herramientas que satisfagan la necesidad de leer assemblies y permitir su manipulación de forma programática, a través de un modelo de objetos que abstraiga su estructura interna. Realizamos pruebas con RAIL [RAIL], CLIFileReader [CLIFR], ILReader [ILREADER], la versión beta del namespace System.Reflection en Whidbey [WHIDBEY] y PERWAPI [PERWAPI], siendo este último el que encontramos finalmente más adecuado para nuestras necesidades, tanto funcionales como de estabilidad.

Esta biblioteca está formada por un conjunto de clases que permiten leer un archivo ejecutable en .NET (PE File), manipularlo y volver luego a escribirlo en disco con relativa facilidad. A continuación mostramos a modo de ejemplo un extracto de código que permite llevar adelante parte de las tareas anteriores: abrir un assembly, recorrer sucesivamente sus tipos, métodos y el correspondiente código IL para finalmente reescribirlo:

```
PEFile semanticatedFile = PEFile.ReadPEFile(assemblyFileName);
CILInstruction instruction = null;
foreach (ClassDef type in semanticatedFile.GetClasses())
{
      foreach(MethodDef method in type.GetMethods())
      {
            if (method.code ==null) // It's an abstract method
                  loop;
             while ((instruction = method.code.GetNextInstruction() as
                  CILInstruction) != null)
                  Console.WriteLine(instrucion.ToString());
      }
}
semanticatedFile.WritePEFile(true);
```

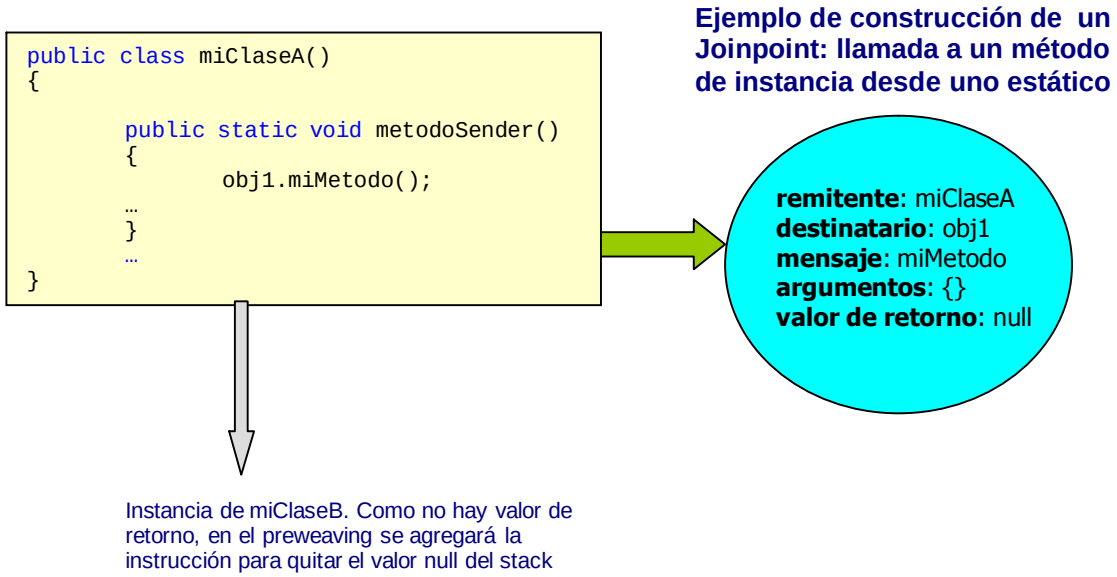
Preweaving

No contamos en .NET con un método que reifique la mecánica de envío de mensajes; podemos identificar sin embargo cada envío a partir de las instrucciones IL destinadas a hacer lo propio. Alcanzaremos el objetivo de delegarlos reemplazando dichas instrucciones por la instanciación del correspondiente joinpoint y la posterior llamada al weaver. Nótese que reificaremos entonces sólo uno de los dos joinpoints planteados en un comienzo: el envío de mensajes, y no ya la recepción. Veremos a través de la práctica si esto resulta suficiente.

Identificamos tres instrucciones que deberían ser afectadas por esta operación: CALLVIRT, usada comúnmente para las llamadas a métodos, NEWOBJ, utilizada para la creación de nuevas instancias (tarea que en Smalltalk se realizaba enviando un mensaje a una clase, ciudadana de primer nivel en ese ambiente, por lo que no era necesario diferenciarla), y CALL, que sirve también para la invocación de métodos, pero hace referencia a una implementación específica. Dejamos para versiones posteriores el manejo de la instrucción CALLI, que invoca un método a través del correspondiente puntero a función; no creemos que presente grandes

impedimentos para la usabilidad del framework, ya que no es generada a menudo, al menos por el compilador C#.

Fue necesario también tomar en cuenta cierta heterogeneidad en la forma en la que operan las instrucciones anteriores: los argumentos no son los mismos al hacer una llamada a un método estático o a uno de instancia; tampoco es lo mismo hacer la llamada desde métodos de uno u otro tipo. No se trata de la misma manera a las funciones que tienen un valor de retorno y a las que no lo tienen: el manejo de la pila es distinto. La reificación de los joinpoints necesitó prestar especial atención a estas diferencias. Se debería lograr que su construcción las homogeneizara, para poder luego continuar trabajando de manera uniforme. Todos necesitan sus respectivos remitentes, destinatarios, mensajes, argumentos y valores de retorno. Seguimos para ello los siguientes lineamientos: cuando se tratase de métodos estáticos (incluimos a los constructores dentro de este grupo), el tipo correspondiente haría las veces de sender o receiver; si no importase el valor de retorno, se lo setearía en null y se lo quitaría luego de la pila al devolver el weaver el control.



Habiendo creado el joinpoint, es necesario llamar al correspondiente método del weaver, representado a través de una clase incluida dentro del assembly Setpoint.dll. Ahora bien, el programa original no conocía necesariamente la existencia de este último assembly... es necesario, por lo tanto, que el preweaving agregue la correspondiente referencia siempre que hiciese falta.

Semanticación

Este proceso consiste básicamente en iterar los tipos definidos en el assembly y crear los *program elements* correspondientes: clases, métodos, anotaciones, relaciones de herencia, etc...(en el siguiente apartado se amplia sobre la forma en la que se realizan las anotaciones). Esta representación es luego incluida dentro del assembly. Fue necesario para poder hacer esto reproducir conceptos del Common Type System[38] en

[38] Ver el anexo I para mayor detalle

una ontología que permitiese la generación de dichas triplas RDF: para la definición de relaciones del estilo <Identificador1, esUna, Clase>, debe estar definido en algún lado el concepto “Clase”[39]. La misma es incluida dentro de SetPoint Engine. Cabe destacar que no se pretendió imitar el CTS en su totalidad, sino sólo aquellos elementos que fuesen necesarios para esta implementación.

Vale la pena mencionar por último que existen reglas de inferencia relativas a las anotaciones que contribuyen a la usabilidad del framework:

- Una clase debe heredar las anotaciones hechas en su superclase y en las interfaces que implemente.
- Los métodos deben estar relacionados con las anotaciones hechas (o inferidas) sobre las clases que los definen.
- Cualquier *program element* anotado con un concepto “B” que especialice un concepto “A”, debe estar relacionado también con el concepto “A”.

Generación de configuración

Luego de ser retocados por el proceso anterior, nuestros assemblies estarán listos para recurrir a cada instante al weaver, quien decidirá sobre la aplicación de aspectos en los distintos joinpoints. Pero... ¿en qué basará sus determinaciones? En la versión Smalltalk debíamos preocuparnos por escribir el código que instanciara y construyera los objetos que hicieran las veces de advices, pointcuts y aspectos, necesitando luego también incluirlos en sendas colecciones para que permanecieran accesibles desde el framework... En este caso necesitamos también cumplir una tarea equivalente. Alguien debe declarar finalmente cuándo y dónde aplicar qué aspectos, y esto es decididamente algo de lo que el weaver no se dará cuenta por sí solo.

Empecemos por el final. Los elementos antes mencionados formarán parte del denominado *setpoint configuration assembly*. Para que el weaver pueda distinguirlos y utilizarlos se debe implementar la interfaz denominada *IConfiguration*, definida en *setpoint.dll*, integrante del Setpoint Engine. Una posibilidad es, entonces, crearla junto con todo el resto de los objetos desde código C# (o cualquier otro lenguaje soportado por .NET), compilar la respectiva DLL respetando la convención de nombres establecida, y luego distribuirla junto con el assembly original. Podemos pensar también en facilitar la escritura de las definiciones de advices, pointcuts y aspectos, proveyendo de un lenguaje específico de dominio que se ajuste a las necesidades puntuales de cada caso. Una tercera mirada nos dice que tanto una como otra forma pueden resultar convenientes de acuerdo al caso. Si el elemento a describir tiene unas pocas características variables, será más cómodo emplear un lenguaje ideado especialmente para describirlas. Si un advice estuviera definido sólo por su nombre, el de un aspecto y el de un pointcut, este sería indudablemente el caso; Si por el contrario se requieren loops, condicionales, y otras estructuras complejas (ahora imaginemos el caso de un aspecto que necesita, para interactuar con cierto dominio, realizar ciertos cálculos complejos), ¿para qué reinventar la

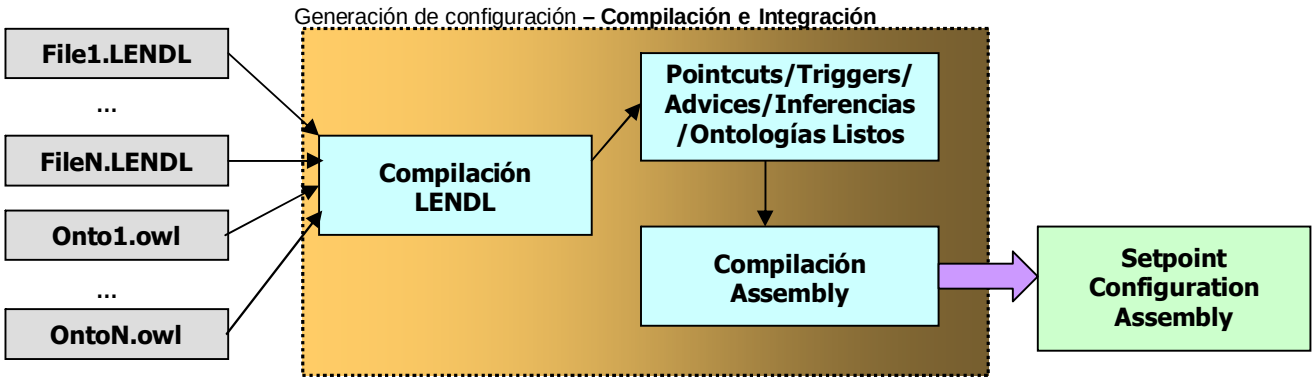
[39] En los anexos se transcribe el detalle de dicha ontología

rueda? Usemos en ese caso un lenguaje de propósito general ya existente. La idea es, entonces, dar ambas posibilidades: luego todo será finalmente mezclado e incluido en el mencionado assembly de configuración.

Es importante destacar también que podrían querer reutilizarse algunas definiciones en diferentes aplicaciones: un mismo pointcut puede por ejemplo querer aprovecharse tanto en el Senku como en un sistema de control de stock. Lo ideal sería obviamente que la plataforma no fuerce la reescritura, sino que permita aprovechar el trabajo ya realizado.

No nombramos tampoco en el presente apartado a las ontologías: antes también eran incorporadas como variables globales; ahora deben formar parte del nuevo assembly para que puedan ser consultadas por intermedio de los predicados semánticos. Son quizá el principal ejemplo de definiciones a compartir por distintos programas, que no deseamos especificar una y otra vez. Tampoco mencionamos a las reglas de inferencia definidas por el usuario. Como es de esperar, también deberán ser incluidas en el nuevo assembly.

Los puntos anteriores nos llevaron a definir el siguiente esquema: el archivo de configuración debe compilarse con cualquier lenguaje compatible con .NET, respetando siempre las convenciones establecidas en relación a interfaces a implementar. Creamos además un lenguaje llamado LENDL[40] que facilita la escritura de pointcuts y advices, y un traductor de éste a C#, que permite generar el esqueleto del programa que luego será compilado para la generación del assembly. Los archivos que conforman los programas LENDL pueden reutilizarse así en diferentes aplicaciones, tomándose como base para la generación del programa de configuración. Habitualmente entonces la creación del archivo de configuración puede esquematizarse según el siguiente gráfico:



No es indispensable seguir este esquema; es posible crear directamente el assembly de configuración con cualquier lenguaje de .NET.

[40] En el anexo 6 se describe cómo fue implementado el lenguaje. Cualquier parecido con el nombre del gran Iván es mera coincidencia. Está claro que se trata de un acrónimo por "Little aspEct & poiNtcut Definition Language".

Ontologías

Permitiremos a partir de esta versión la definición de modelos semánticos a través de OWL: el motor de consultas podrá inferir relaciones a partir de los significados de los conceptos preestablecidos. El proceso de generación de inputs aceptará como entrada archivos en formato XML/RDF, que pueden ser generados en la forma en la que el usuario considere más conveniente. En nuestro caso, decidimos valernos del entorno Prótegé [PROTEGE].

Para realizar las anotaciones, se definió en setpoint.dll (SetPoint Engine) un atributo que indica que un program element está relacionado con un determinado concepto. Recibe como parámetro el identificador unívoco del concepto a asociar al objeto de metadatos. Supongamos que pretendemos que las instancias de Form1 sean identificadas como objetos pertenecientes al componente “vista”. En la declaración de la clase debemos agregar:

```
[setPoint.programAnnotation.ProgramAnnotation(“semantics://perspectives/architecture#view”)]
public class Form1: System.Windows.Form { … }
```

Una vez escritas las anotaciones, serán incorporadas a la representación RDF del assembly en cuestión a través del proceso de semanticación, según lo descrito en el capítulo anterior.

Reglas de inferencia

Como parte de las ampliaciones al modelo semántico soportado por el framework, esta nueva versión permitirá al usuario agregar sus propias reglas de inferencia por sobre el modelo de conocimiento utilizado. Tomemos como ejemplo el caso de la regla mencionada anteriormente sobre la transitividad en la especialización de anotaciones: cualquier *program element* anotado con un concepto “B” que especialice un concepto “A”, debe estar relacionado también con el concepto “A”. El lenguaje LENDL permite escribir este predicado lógico de la manera siguiente:

```
rule annotationTransitivity{
   infer B [semantics://programElements/objectOriented/CTS#hasAnnotation] X
   when A [semantics://programElements/objectOriented/CTS#hasProgramElement] B and
       A [semantics://programElements/objectOriented/CTS#hasAnnotation] X;
}
```

El framework ya cuenta con algunas reglas incorporadas; ésta es una de ellas, y no hace falta que cada assembly de configuración la incorpore.

Alias

Cuando escribimos una URI, tenemos que refirnos al namespace al que pertenece el concepto en cuestión. En el ejemplo anterior, necesitamos repetir todo el tiempo “semantics://programElements/objectOriented/CTS”. Un bajón. LENDL brinda la posibilidad de introducir *alias* para los namespaces más utilizados, de forma tal que no sea necesaria la escritura del namespace en su

totalidad, sino que podamos referirnos a él a través de una abreviatura. El ejemplo anterior, por ejemplo, es equivalente al siguiente extracto de código LENDL:

```
declare CTS alias semantics://programElements/objectOriented/CTS;
rule annotationTransitivity{
   infer B [CTS#hasAnnotation] X
   when A [CTS#hasProgramElement] B and
       A [CTS#hasAnnotation] X;
}
```

El lenguaje cuenta con 4 alias incoporados por defecto al ambiente: cts, rdf, rdfs y owl. El primero hace referencia al alias ya descrito, mientras que los últimos corresponden a los correspondientes namespaces definidos por el W3C consortium.

Pointcuts

Tenemos ya una manera de definir modelos de conocimiento y relacionarlos con el código fuente. Nos resta ahora encontrar la forma de predicar sobre ellos. ¿Implementaremos nuevamente nuestro propio modelo de predicados e inferencias, o aprovecharemos en este caso alguna API ya desarrollada? Nos inclinamos por la segunda opción: Sesame [SESAME] fue la biblioteca elegida. Usaremos internamente SeRQL [SeRQL] como lenguaje de consultas: dicha biblioteca lo soporta nativamente, permitiendo la realización de queries sobre modelos RDF y OWL e infiriendo reglas sobre la semántica definida por RDF-Schema (vale la pena aclarar que aún no son soportadas inferencias a partir de conceptos OWL).

LENDL permitirá predicar sobre la existencia de relaciones, donde se considera que existen no sólo aquellas tuplas explicitas, como ocurría en el prototipo, sino también las inferidas a través de reglas. Recordemos que no serviría de nada hablar únicamente sobre características de la ontología, sino que el meollo de la cuestión consiste en relacionarla con el contexto de ejecución, escenificado a través del joinpoint.

```
pointcut MyPointcut {
      sender is [semantics://anOntology#concept1];
      receiver is [semantics://anOntology#concept2];
}
```

Nótese que se indica el nombre del pointcut, y luego nos limitamos a especificar con qué conceptos están relacionados los objetos del contexto. Utilizamos para ello las palabras clave *sender*, *receiver* y *message*, que hacen referencia a los *program elements* relativos a las correspondientes propiedades del joinpoint que se evaluará al consultar el predicado. La palabra clave *is* equivale al concepto *hasAnnotation* introducido por el proceso de semanticación a partir de las anotaciones, y entre corchetes figura el concepto relacionado. Se verifica, en resumen, la existencia de la tripla <*program element*, *hasAnnotation*, *concepto*>, ya sea por haber sido explicitada a través de anotaciones o deducible mediante la aplicación de reglas de inferencia. Es posible también reemplazar la palabra clave *is* por la URI de cualquier otro concepto, que hará luego las veces de predicado en la tripla consultada, como en el ejemplo siguiente:

```
pointcut MyOtherPointcut {
      message [semantics://anOntology#aRelation] [semantics://anOntology#concept1];
}
```

Esta sintaxis es diferente a la de SeRQL, pero el modelo que se obtiene al parsearla corresponde sin embargo a un subconjunto de aquél. Creemos que este lenguaje resulta más claro para comenzar la exploración de los pointcuts semánticos. Eventualmente permitiremos también la escritura de predicados directamente en SeRQL. Vale la pena mencionar también algunas funcionalidades no incluidas dentro de esta versión que consideramos deben ser exploradas en un futuro cercano:

- Se permite hablar únicamente del joinpoint en curso, aislándolo de todo contexto de ejecución. Se debería poder hacer referencia al menos a todos los joinpoints que permanecieran dentro del call stack. Esto permitiría escribir enunciados tales como "todos los mensajes que impliquen `[movimientos de stock]` originados por un transacción de `[venta]`".

- Cuando escribimos "**sender** `is [producto manufacturado]`", por citar un ejemplo, estamos definiendo que el objeto que envía el mensaje está relacionado con el concepto [`producto manufacturado`]. Al exisitir esta relación y estar definido este último concepto en una ontología formal, seguramente definimos otras características que se deberían cumplir. Supongamos por ejemplo que todo objeto anotado como `[producto manufacturado]` debiera tener un [`costo de fabricación`]. Sería útil que en el pointcut podamos referirnos a esta última característica. Para ello debimos relacionarla previamente con la porción de código correspondiente… ¿Será necesariamente uno sólo? ¿Bastará con una simple anotación, o tendremos quizá que ejecutar alguna transformación previa? ¿Cómo se garantizará la coherencia en las definiciones (es decir, marcar una clase como *producto manufacturado* implica que también debo señalar su *costo*)? Denominamos a este área *contratos semánticos*. La sintaxis de LENDL deberá soportar este concepto. El ejemplo presentado en el párrafo, por ejemplo, podría expresarse de una manera similar a la siguiente:

  ```
  pointcut SemanticContractPointcutExample {
        sender is [semantics://manufacture];
        then sender.cost > 100;
  }
  ```

  Paralelamante a esta extensión a LENDL, deberán darse los mecanismos para relacionar estos nuevos conceptos con los *program elements* que correspondan, ya sea mediante anotaciones directas (el costo podría estar representado, por ejemplo, por algún parámetro del método que esté siendo ejecutado) o mediante anotaciones que denoten algún procesamiento.

- Más allá de la recomendación de trabajar siempre en base a semántica, es posible que el framework quiera utilizarse sobre sistemas ya desarrollados en los que vayan incorporándose paulatinamente las

anotaciones, y se necesite en un principio predicar sobre la sintaxis. Deberían enriquecerse las herramientas que permiten esto. También por esta razón debería poderse hablarse sobre los parámetros de los métodos, aún cuando carezcan de un modelo semántico que les otorgue real significado.

Aspectos

En esta nueva versión, es posible definir distintos eventos relevantes para un aspecto, y en cada uno puede decirse qué código ejecutar antes o después del joinpoint en cuestión. Cada aspecto tiene su propio protocolo, que no se limita ya al after y al before; los advices deben relacionar cada uno de los nuevos mensajes con los correspondientes pointcuts. A continuación mostramos cómo se define un aspecto en LENDL:

```
aspect MyAspect builtby DefaultSingletonAspectFactory {
            event oneRelevantEvent;
            event anotherRelevantEvent;
      }
```

Se especifican sólo algunos pocos datos: el nombre del aspecto, su protocolo (cada uno de los eventos), y su factory. En rigor de verdad, esta definición sirve exclusivamente para la posterior validación de los advices, y no contribuye a la programación de los aspectos en sí. En próximas versiones podría ampliarse el traductor de LENDL para que genere el código C# con el esqueleto del aspecto correspondiente: la clase MyAspect, en este caso, que debe implementar la interfaz IAspect (únicamente incluye por ahora el método “mustCancelExecution”, que indica si debe continuarse con la ejecución del joinpoint) y contiene los métodos públicos, oneRelevantEvent y anotherRelevantEvent en este caso, que reciben como argumento un joinpoint. Estos últimos serán quienes permitan finalmente al weaver interactuar con la dimensión ortogonal en cuestión. El programador debería luego escribir sólo el código de estos métodos.

El hecho de que las clases no sean verdaderos ciudadanos de primera en .NET trajo problemas a la hora de definir la interfaz IAspect: no fue posible incluir métodos estáticos abstractos. Necesitamos por ello reemplazar el antiguo método instanceFor: aJoinpoint, que decide sobre la instanciación del aspecto, por la creación de un factory [GAMMA] de aspectos. La cláusula opcional “builtBy” indica la clase que hará las veces del mencionado factory. En caso de no especificarse, se asumirá para la instanciación de los aspectos el comportamiento de un Singleton [GAMMA].

Advices

El último de los elementos que nos hace falta mencionar son los advices. Serán ahora más específicos: asociarán cada evento de un aspecto con los correspondientes pointcuts. Llamamos *triggers* a estas relaciones de menor granularidad.

```
advice myAdvice : myAspect {
      trigger oneRelevantEvent after miPointcut1;
      trigger anotherRelevantEvent before miPointcut2;
}
```

Se declaran primero el nombre del advice y el aspecto al cuál nos estamos refiriendo. A partir de él pueden deducirse los correspondientes eventos que necesitan relacionarse: *oneRelevantEvent* y *anotherRelevantEvent,* en este caso. Deben indicarse asimismo los pointcuts relacionados, y si serán disparados antes o después de la ejecución del joinpoint en cuestión.

Setpoint Engine

Tenemos ya todos los elementos para comprender cómo se comporta SetPoint durante la ejecución de un programa. El weaver es el encargado de la coordinación de las partes, ayudándose en cada caso de los objetos correspondientes. Comenzará su misión cuando un assembly aspectualizable realice la primera llamada a un método. Se aprovecha este momento para configurar y cargar en memoria todos los elementos necesarios para la toma de decisiones sobre la aplicación de aspectos: advices, pointcuts, aspect factories, ontologías, recursos y reglas de inferencia. Habrá dos situaciones en la que esta configuración podrá ser revisada:

- Cuando se haga referencia a un nuevo assembly aspectualizable, deben incorporarse sus *program elements* al universo semántico definido originalmente.
- Si se desea modificar la configuración original definiendo un nuevo archivo de configuración, habrá que reemplazar todos los modelos levantados en memoria. .Net no permite el denominado *HotSwap*[41], por lo que en principio sería necesario detener la aplicación para producir el reemplazo. Esta situación será evitada en próximas versiones mediante algún mecanismo de carga y descarga dinámica de assemblies.

El esquema seguido al ejecutarse cada joinpoint es cuasi idéntico al del prototipo: se determina a qué pointcuts pertenece el joinpoint en curso; luego se verifica con qué triggers están éstos asociados; se obtiene finalmente la política de aplicación de aspectos y se ordena luego a ésta que proceda[42]. Los aspectos serán instanciados a través del factory correspondiente.

La diferencia principal radica en la manera en que se maneja el denominado universo semántico, conformado por el conjunto de *program elements*, anotaciones, reglas de inferencia y ontologías definidas. Tal como dijimos anteriormente, decidimos aprovechar en este caso APIs para manejo de modelos de conocimiento que ya hubieran sido desarrolladas. La implementación de un sistema que respete las inferencias de los modelos utilizados y que permita además la realización de consultas, no resulta una tarea para nada trivial. Los

[41] La definición aparece en http://java.sun.com/products/hotspot/docs/whitepaper/Java_HSpot_WP_v1.4_802_3.html#hot

[42] Vale la pena aunque más no sea comentar una dificultad surgida de la aparición de los triggers: ¿Cómo debe continuar la ejecución de la combinación aspectos/joinpoint si un aspecto decidió que la misma no debía continuar? No nos es tan fácil en este nuevo esquema relacionar eventos de after y de before: anteriormente un aspecto establecía una relación uno a uno entre estos eventos; en la nueva versión hay libertad para definir cualquier cardinalidad. La decisión fue que la chainPolicy ejecutara sólo los triggers de after que pertenecieran a advices que contuvieran triggers de before que ya hubieran sido ejecutados.

avances más importantes en esta área fueron desarrollados por la comunidad Java. Decidimos valernos entonces de las capacidades de IKVM [IKVM], un sistema que permite la interacción entre ambas plataformas mediante la implementación de una máquina virtual de Java para .NET, de sus bibliotecas base y de un conversor de código. Pudimos de esta forma aprovechar Sesame [SESAME] para la representación del universo semántico. Implementamos luego algunas modificaciones para adaptarlo a nuestras necesidades: incorporamos un motor de inferencia para permitir el agregado de nuevas reglas en un modelo de conocimiento residente en memoria, que fueron luego agregados a la versión estándar.

Cuando decíamos, entonces, que las ontologías y los recursos fueron cargados, nos referíamos a que fueron incluidos en una base de conocimiento de Sesame (almacenada también en memoria). En ese momento el nuevo motor de inferencias calculará las nuevas relaciones existentes en el modelo a partir de las reglas predefinidas en el framework y las nuevas incorporadas por los usuarios. Entre las predefinidas encontramos las correspondientes al modelo RDF-Schema y la transitividad de las anotaciones semánticas, tanto a nivel de especialización de conceptos (como se explicaba en el apartado sobre reglas de inferencia) como a nivel de los *program elements* (si un assembly está anotado con un concepto X, también lo estarán sus modulos; si un módulo está anotado, también lo estarán sus clases e interfaces; y así sucesivamente). Falta incorporar a estas inferencias la transitividad de anotaciones a partir de relaciones de herencia entre clases y a través de la implementación de interfaces, reglas que serán incorporadas en las versiones siguientes de SetPoint. Luego, el objeto que representa al universo semántico se valdrá de esas APIs para la ejecución de las consultas.

El componente denominado Setpoint Engine consiste, por lo tanto, no sólo del Setpoint.dll, que contiene las definiciones de las clases e interfaces que representan aspectos, pointcuts, advices, triggers y configuraciones, sino también de las DLLs que representan la máquina virtual de java para .NET, sus bibliotecas base y la implementación de Sesame.

Aplicando aspectos sobre el Senku

Los pasos a seguir para la utilización del framework descrito sobre el Senku son prácticamente idénticos a los seguidos en la versión Smalltalk, con ligeras modificaciones:

1. Definir las ontologías a utilizar.
2. Anotar semánticamente el código.
3. Programar las dimensiones ortogonales necesarias, en el caso que no estuvieran ya desarrolladas.
4. Escribir el código LENDL.
5. Generar el código C# con la definición de la configuración.
6. ¡Jugar al Senku (intentando esta vez sí ganar aunque más no sea una partida)!

Ontologías utilizadas

Se mantuvieron las perspectivas de arquitectura y de la información utilizadas en el prototipo. El modelo creado en ese momento se constituía de conceptos aislados; no habían podido ser definidas relaciones entre ellos. En esta nueva versión es factible definir modelos de conocimiento OWL. Para verificar el correcto funcionamiento y la potencialidad del motor de inferencias, se implementó una ligera modificación en la

ontología de la información: se crearon dos subtipos del concepto *referentiallyTransparent*: “constant” y “byArguments”. Luego, el código será anotado utilizando estos dos últimos conceptos, pero el pointcut seguirá siendo definido en base al primero de los términos. Aún en la simpleza del ejemplo, puede vislumbrase el potencial de la aplicación de inferencias sobre los modelos de conocimiento.

La ontología de arquitectura seguirá por lo pronto siendo definida en base a conceptos aislados. Se aprovechó además la ontología definida para el CTS, con el fin de predicar sobre el modelo subyacente. Por último, se definió también un modelo denominado funcional: es discutible si esto era estrictamente necesario o si se debió a las limitaciones actuales de LENDL. En cualquier caso, era una necesidad de la implementación para poder referirse al comienzo de una partida.

Anotaciones realizadas

Las anotaciones fueron muy similares a las aplicadas en la versión anterior, con las salvedades extraídas de las facilidades que provee esta nueva implementación y de los cambios en la definición de ontologías. En relación a la perspectiva de arquitectura, vale decir que la vista y el modelo fueron implementadas en assemblies diferentes. Se anotaron por tanto exclusivamente estos últimos, mediante la incorporación de los siguientes atributos en los correspondientes archivos de información del assembly:

```
[assembly: ProgramAnnotation("semantics://perspectives/architecture#model")]
[assembly: ProgramAnnotation("semantics://perspectives/architecture#view")]
```

Necesitó incorporarse a cada uno la referencia a Setpoint.dll, y agregar en cada archivo la cláusula `using setPoint.programAnnotation;` para que se reconociese dicho namespace.

Los métodos de la clase *Board* que antes habían sido anotados con el concepto *referentiallyTransparent* ahora lo fueron con *`semantics://perspectives/information#byArguments`*. El siguiente extracto muestra uno de ellos, el denominado *above*:

```
[ProgramAnnotation("semantics://perspectives/information#byArguments")]
public Square above(Square aSquare)
{
      …
}
```

Se anotó también el método *build* de la clase *GameBuilder* para señalar el comienzo de una nueva partida. Vale aclarar por último que aprovechamos también para la elaboración de los predicados las triplas relativas al CTS introducidas automáticamente por el proceso de semanticación. En los apartados siguientes detallaremos su uso.

Definición de las dimensiones ortogonales

Nuevamente se siguió la premisa del experimiento anterior: los aspectos de caching y logging se desarrollaron de forma sencilla, aprovechando las funcionalidades provista por las clases Hashtable y EventLog, respectivamente, mientras que para el caso de la distribución se aprovecharon las facilidades de .NET Remoting, framework de distribución nativo del ambiente. La utilización del mismo impide que el desarrollador

del código base sea totalmente ajeno a la implementación del aspecto: para que una clase pueda ser instanciada remotamente, debe haber especializado a la clase MarshalByRefObject. El programador del código funcional necesitó contemplar este tema: caso contrario, el aspecto no podrá aplicarse.

Cabe aclarar por último los eventos de relevancia que fueron definidos para cada uno de los aspectos:

- Logging: *startLogging* y *endLogging*. El primer mensaje inicializa y dispara un timer, mientras que el segundo registra el intervalo de tiempo transcurrido en el event log del sistema operativo.
- Caching: *lookUp, afterLookUp* y *flush*. El primero busca la existencia de un mensaje en el caché, el segundo realiza luego la operatoria correspondiente en caso de un hit o un miss, y el tercero se ocupa de vacíar oportunamente la memoria caché.
- Distribución: *createRemoteObject*. Crea un objeto en un proceso remoto.

Código LENDL

Los pointcuts, advices y aspectos fueron definidos en base a la suma de los conceptos descritos en la primera versión y los agregados en los apartados anteriores:

```
declare architecture alias semantics://perspectives/architecture;

/***********************************/
/* Logging Aspect                  */
/***********************************/
pointcut ViewToModelMessages{
      sender is [architecture#view];
      receiver is [architecture#model];
}

aspect LoggingAspect{
      event startLogging;
      event endLogging;
}

advice LogEvents : LoggingAspect{
      trigger startLogging before {ViewToModelMessages};
      trigger endLogging after {ViewToModelMessages};
}

/***********************************/
/* Caching Aspect                  */
/***********************************/
pointcut ReferentiallyTransparentMessages{
      message is [semantics://perspectives/information#referentiallyTransparent];
}

pointcut GameBeginningMessages{
      message is [semantics://perspectives/functional#startGame];
}

aspect CachingAspect{
```

```
      event lookUp;
      event flush;
      event afterLookUp;
}

advice CacheMessages : CachingAspect{
      trigger lookUp before {ReferentiallyTransparentMessages};
      trigger afterLookUp after {ReferentiallyTransparentMessages};
      trigger flush after {GameBeginningMessages};
}

/**********************************/
/* Distribution Aspect            */
/**********************************/
pointcut ModelObjectsCreationMessages{
      sender is [architecture#view];
      receiver is [architecture#model];
      message  [rdf#type] [cts#Constructor];
      receiver [cts#isDelegate] [cts#false];
}

aspect DistributionAspect{
      event createRemoteObject;
}

advice DistributeComponents : DistributionAspect{
      trigger createRemoteObject before {ModelObjectsCreationMessages};
}
```

Resumen

Setpoint, en su versión .NET, intenta ser lo más semejante posible a su antepasado en Smalltalk, aunque presenta cambios importantes debidos tanto a diferencias en los ambientes como a mejoras introducidas. Para que una aplicación pueda trabajar bajo el framework debe pasar a través de los procesos de *preweaving* y *semanticación*, que introducen las llamadas al weaver y crean la representación RDF del assembly, respectivamente, y de generación de inputs, que genera el archivo que incluye la configuración de pointcuts, advices, y demás elementos necesarios para determinar la aplicación de aspectos. Para la implementación del universo semántico se aprovecha la biblioteca Sesame, que incorpora la capacidad de realizar inferencias sobre modelos RDF-Schema. Se crea también el lenguaje LENDL para facilitar la definición de advices y pointcuts, basados esta vez en SeRQL, que da también la posibilidad de definir nuevas reglas de inferencia. Se introduce el concepto de trigger, que permite a un aspecto estar asociado a eventos que le permitan interactuar con más de un joinpoint. Se muestra por último cómo se aprovechó todo esto para satisfacer los requerimientos establecidos para el Senku.

Conclusiones

- El objetivo principal que habíamos planteado pudo ser satisfecho: la construcción de un framework (en rigor de verdad, se crearon dos) que implementara el concepto de *pointcut semántico*. En la primera versión se detectaron una cantidad de fallas en la aplicación de setpoints no originadas en el plano conceptual, sino en la necesidad de mejorar y profundizar el modelo implementado. Habiéndose superado los problemas originados en bugs de la plataforma y habiéndose ampliado el modelo semántico para la segunda versión, la utilización de *setpoints* resultó más satisfactoria.

- Es menester destacar que, pese a que los pointcuts utilizados permitieron que nos abstrayéramos del código fuente en sí, no conseguimos independizarnos completamente de la plataforma: para el caso de la distribución, por ejemplo, necesitamos predicar sobre la ontología del CTS.

- Creemos que la idea de *pointcuts semánticos* desarrollada representa un buen punto de partida en la búsqueda de plataformas orientadas a aspectos realmente utilizables. Sin embargo, vale la pena resaltar que se trata únicamente de un buen punto de partida. Aún resta profundizar sobre una cantidad de conceptos teóricos y cuestiones eminentemente prácticas, entre las que destacamos:
    - Contratos semánticos: ¿Cómo podrán los pointcuts referirse a propiedades que un joinpoint debería incluir a partir de su contenido semántico? ¿Cómo se puede garantizar que al realizarse cada anotación se respeten esas características que deberían darse de acuerdo al significado? El caso de la manufactura y su costo, presentado en el capítulo anterior, resulta un claro ejemplo de esta necesidad.
    - Modelado de Perspectivas: Esta prueba de concepto se realizó sobre un modelo genérico de definición de ontologías. Se detectaron algunas falencias: no se pudo por ejemplo predicar sobre los suscriptores a un evento particular del event handler. Podría haberse optado también por el enfoque de modelos ad-hoc, de acuerdo a perspectivas particulares del sistema; sería útil desarrollar pruebas sobre alguna plataforma que implemente esta idea para poder comparar los resultados.
    - LENDL: Será necesario continuar mejorando el lenguaje, extendiéndolo de acuerdo a las necesidades ya mencionadas: posibilidad de predicar sobre el callstack, contratos semánticos, mayor riqueza en los predicados sintácticos y capacidad de predicar sobre argumentos de métodos.
    - Performance: El overhead introducido por la intercepción de cada envío de mensajes y la ejecución de las consultas semánticas correspondientes debería repercutir el mínimo indispensable sobre la performance de una aplicación. Se debe trabajar sobre mecanismos que reduzcan al máximo esta sobrecarga de trabajo, para que la plataforma AOP sea comparable a otras utilizadas en la industria. También debe optimizarse la arquitectura del framework, intentando evitar la complejidad

(y consecuente sobrecarga) introducida por la diversidad de entornos utilizados: el uso de Sesame a través de IKVM aparece como el principal componente a evaluar.

- IDE y herramientas de desarrollo: La interfaz de aplicación de aspectos no está integrada con el entorno utilizado por el programador. Sería bueno incluirla a través de add-ins o extensiones para Visual Studio u otros entornos de programación. Es necesario también desarrollar browsers que ayuden a entender a qué joinpoints afecta un pointcut, herramientas que faciliten la detección de crosscutting concerns, debuggers, etc...

- Para que los *setpoints* tengan el efecto buscado, el programa en ejecución debe estar correctamente anotado. Esto requiere o bien que el programador lleve adelante esta tarea o bien la creación de un rol destinado especialmente para tal fin. Creemos que para la correcta aplicación de aspectos algún tipo de trabajo extra será necesario, ya sea el antes mencionado o el respeto hacia convenciones de notación, en las herramientas criticadas. En la plataforma propuesta estas tareas extras no invalidan el *obliviousness* hacia las incumbencias transversales, y su potencial aplicación va más allá de la utilización desde un framework AOP.

- Vale la pena repetir que la correcta separación de incumbencias no puede ni debe comenzar por los aspectos, sino que es necesario un diseño cuidado del programa, asignando claramente a cada objeto su responsabilidad, para recién luego avanzar un nuevo escalón y modularizar los concerns transversales.

- El intento de desarrollar un ambiente multilenguaje quedó relativamente trunco. El basarse en IL no garantiza la neutralidad de la plataforma. IL mismo propone sus propias abstracciones, que no necesariamente coincidirán con las del lenguaje con el que se escriba el código fuente. El Senku desarrollado fue implementado usando C#. Será necesario entonces experimentar con programas hechos con otros lenguajes.

- Las distintas características de las plataformas sobre las que trabajamos presentaron sus respectivas ventajas y desventajas a la hora de la comparación. Resumiendo, podemos decir que el enfoque minimalista de Squeak nos facilitó la modificación del ambiente, a la vez que su calidad de experimental nos complicó a la hora de intentar utilizarlo sobre una aplicación real, mientras que con .NET nos sucedió lo contrario: la cantidad de construcciones y detalles a tomar en cuenta dificultó el desarrollo del motor de weaving, pero una vez contempladas estas cuestiones no tuvimos mayores inconvenientes en su uso.

Anexo I: El framework .NET [43]

Introducción

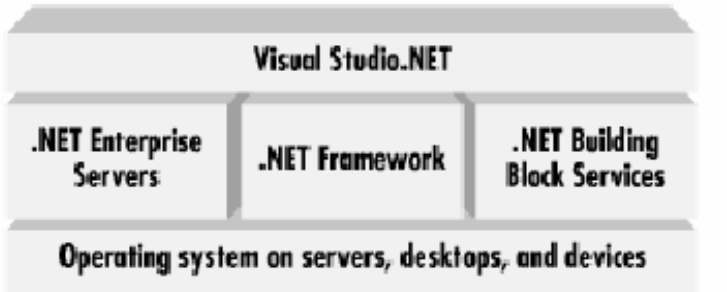
El objetivo del capítulo es introducir la plataforma .NET, utilizada como entorno de desarrollo para el framework de aspectos concebido en el presente trabajo. *Putting your finger on exactly what .NET is and isn't at any point in time is hard*[44], dice un artículo publicado en el sitio de Microsoft, y eso fue exactamente lo que encontramos al escribir esta sección. Abundan las definiciones no siempre equivalentes de un mismo término. En esos casos, tomamos como última palabra la bibliografía publicada por dicha empresa. No pretendemos dar en este capítulo una visión crítica de la plataforma, sino simplemente destacar las características que resultaron trascendentes para nuestra tesis.

La plataforma Microsoft .NET

Microsoft anuncia en julio del 2000 el lanzamiento de la plataforma .NET. Los principales objetivos eran:

- Mejorar la interoperabilidad entre sistemas
- Mejorar el soporte para dispositivos móviles, que comenzaban a ser productos de consumo masivo.
- Facilitar el desarrollo de aplicaciones para internet y de servicios web
- Aumentar la productividad de las tareas de desarrollo y deploy.

La plataforma consta de 5 componentes fundamentales[45]:



- Visual Studio .NET: Herramienta de desarrollo concebida por Microsoft desde el inicio del proyecto. Con el tiempo fueron naciendo otras herramientas de terceras partes, como Web Matrix o Sharp Developer.

[43] **Fuentes**: [NET][NET1][NET2][NET3][NET4][NET5][ECMA335][MONO][ROTOR]

[44] **Fuente**: DR GUI .NET 1.1 #0, Introduction to .NET, Hello Word, and a Quick Look on the .NET Runtime, 24/04/2003

[45] **Gráfico**: .NET Framework Essentials, Thuan Thai y Hoang Lam, O'Reilly [NET]

- .NET Enterprise Servers: Familia de servidores que permiten ejecutar, operar y administrar aplicaciones. Forman parte de este grupo Windows 2003 Server, MSSQL Server y BizTalk Server, entre otros.
- Operating Systems: Conjunto de sistemas operativos integrados con la plataforma. Se incluyen Windows XP, Windows CE y Windows ME.
- .NET Building Block Services: Servicios web provistos por Microsoft.
- .NET Framework: Conjunto de servicios destinados a la ejecución y construcción de aplicaciones.

.NET Framework

La arquitectura de .NET framework está representada por el siguiente esquema[46]:

Los componentes de la capa superior manejan aquello que tenga que ver con interfaces de un sistema hacia el exterior: páginas web, aplicaciones clásicas para windows, web services publicados, etc... Una segunda capa colabora en las cuestiones relacionadas con la persistencia de datos, a través de ADO.NET, y en el manejo de información en formato XML.

Luego aparecen las llamadas “Base Framework Classes”, que proveen una serie de servicios que pueden utilizarse desde cualquier aplicación. Por último encontramos al CLR (Common Language Runtime), cuya función es administrar la ejecución de aplicaciones.

Common Language Runtime

El CLR es el motor que administra la ejecución de aplicaciones bajo el .NET Framework. Brinda servicios como garbage collection, seguridad, control de tipos, reflection y manejo de excepciones. Se llama *managed code* al código que corre bajo su control. A continuación presentamos la descomposición del runtime en sus diferentes componentes[47]:

[46] **Gráfico**: http://msdn.microsoft.com/netframework/technologyinfo/overview/ [NET3]

[47] **Gráfico**: Curso .NET paso a paso: introducción al desarrollo en .NET (http://msdn.microsoft.com.ar/) [NET5]

El CLR es la implementación comercial del conjunto de especificaciones agrupadas bajo el nombre de *Common Language Infraestructure* (CLI), estandarizadas según la especificación ECMA-335. Cada programa que quiera ser ejecutado por esta máquina virtual debe ajustarse a dicha especificación.

Otra implementación destacable del CLI es la conocida como *Rotor* o *SSCLI* (Shared Source CLI). Como su nombre lo indica, presenta una versión del estándar bajo licenciamiento de tipo shared source. Presenta funcionalidades más reducidas que el CLR, y el código fuente está organizado para que resulte más legible.

Vale la pena mencionar por último al proyecto open source MONO, auspiciado por Novell, que implementa el framework para Linux.

Common Language Infraestructure

El objetivo es brindar de un entorno para el cual cualquier compilador pueda generar código. Que se pueda, por ejemplo, programar una clase en C# y luego utilizarla desde una aplicación desarrollada en Visual Basic .NET, ya sea enviándole un mensaje o designando a la una como superclase de la otra.

El estándar provee la especificación que debe seguir el código a ser ejecutado, la que debe seguir el entorno de ejecución (llamado *Virtual Execution System*) y el formato de los archivos ejecutables. En los apartados subsiguientes introduciremos los componentes principales.

Common Type System (CTS)

Sistema de tipos manejado. Se definen tipos y operaciones comunes. Algunos (integer, floating point, strings) son soportados nativamente por la plataforma, mientras que se dan los lineamientos para las entidades que puede definir el usuario.

Common Language Specification (CLS)

Conjunto de reglas a seguir por todos los miembros públicos definidos por lenguajes compatibles con CLI. El objetivo es facilitar la interoperabilidad: es por ello que no es necesario que los elementos no visibles hacia el exterior de una unidad de compilación sigan dichas condiciones.

Formato de archivos

Se define como *assembly* a la unidad mínima de deploy que puede definirse bajo CLI. Representa una unidad cohesiva de tipos. Se dice que es auto-descriptivo porque contiene toda la información acerca de los tipos implementados; se denomina *metadata* a dicha información.

A grandes rasgos, los elementos que aparecen en un assemby son:

- Encabezado: Marca del archivo como ejecutable bajo CLI, contiene el índice a las secciones principales y la dirección de inicio, en caso de que se trate de un ejecutable.
- *Manifest*: Identificación del assembly, archivos que lo componen, tipos y miembros definidos, referencias hacia tipos definidos en otros assemblies y dependencias.
- Código: Implementación de los tipos definidos. Se encuentran definidos en un lenguaje llamado *IL*, descrito más adelante.
- Recursos: Texto, imágenes, o cualquier otra información el assembly necesite.

La existencia de estas secciones hace que un assembly sea auto-definido. Es decir, que contenga toda la información necesaria y no necesite de otros elementos para poder ser ejecutado (siempre y cuando no se expliciten dependencias).

Metadata

Define el formato binario en el que se almacena la información de tipos, por ejemplo en el manifiesto de un assembly. La existencia de metadatos posibilita entre otras cosas la existencia de un framework de reflection. Vamos a mencionar aquí solo dos características de relevancia para este proyecto: los *atributos* y los *tokens*.

Atributos

Cada objeto de metadatos (la definición de una clase, un método, una interfaz, etc...), puede tener asociados *atributos,* que permiten al programador especificar propiedades sobre él. Estos atributos son análogos a las definidos por keywords como *public* u *override*, sólo que, además de poder utilizarse los ya definidos en el framework, se pueden definir los propios. Los mismos pueden luego ser consultados mediante las características de reflection mencionadas antes.

Ejemplificando con código C#[48]:

```
[MyAttributes.IsPrintingMethod(“screen”)]
public override doSomething()
{
      Console.WriteLn(“imprimo”);
      …
}
```

[48] No se sigue la representación de CLI, sino una de un lenguaje de alto nivel compatible, pero para los fines del ejemplo resulta más claro

Si en algún momento queremos consultar la definición del método doSomething, veremos que tiene asociado el atributo IsPrintingMethod. Para definir un atributo, se debe crear una subclase de la clase System.Attributte, integrante de las librerías base de CLI (ver más adelante). Pueden especificarse también parámetros para su construcción, que luego también podrán ser consultados.

Metadata Tokens

La información de metadatos es almacenada en una sección especial de los archivos compatibles con CLI. Es organizada en diferentes tablas: existe una que identifica las clases, otra los métodos, otras las interfaces.... Cada una de las filas de estas tablas se conoce unívocamente través de su *token,* identificador de 4 bytes compuesto por un primer byte que determina la tabla y los siguientes 3 que señalan el número de registro dentro de la misma.

Virtual Execution System

Entorno para la ejecución de programas que respeta las especificaciones comentadas en los apartados anteriores. Provee mecanismos para garantizar la seguridad, el manejo de excepciones, la administración de memoria, etc... El código a ejecutar debe estar escrito en un lenguaje llamado MSIL (Microsoft Intermediate Language), especie de assembler diseñado para que pueda servir como objetivo de una cantidad de lenguajes de programación de alto nivel. Las instrucciones de IL que necesitan hacer referencia a objetos de metadatos se refieren a ellos a través de su token.

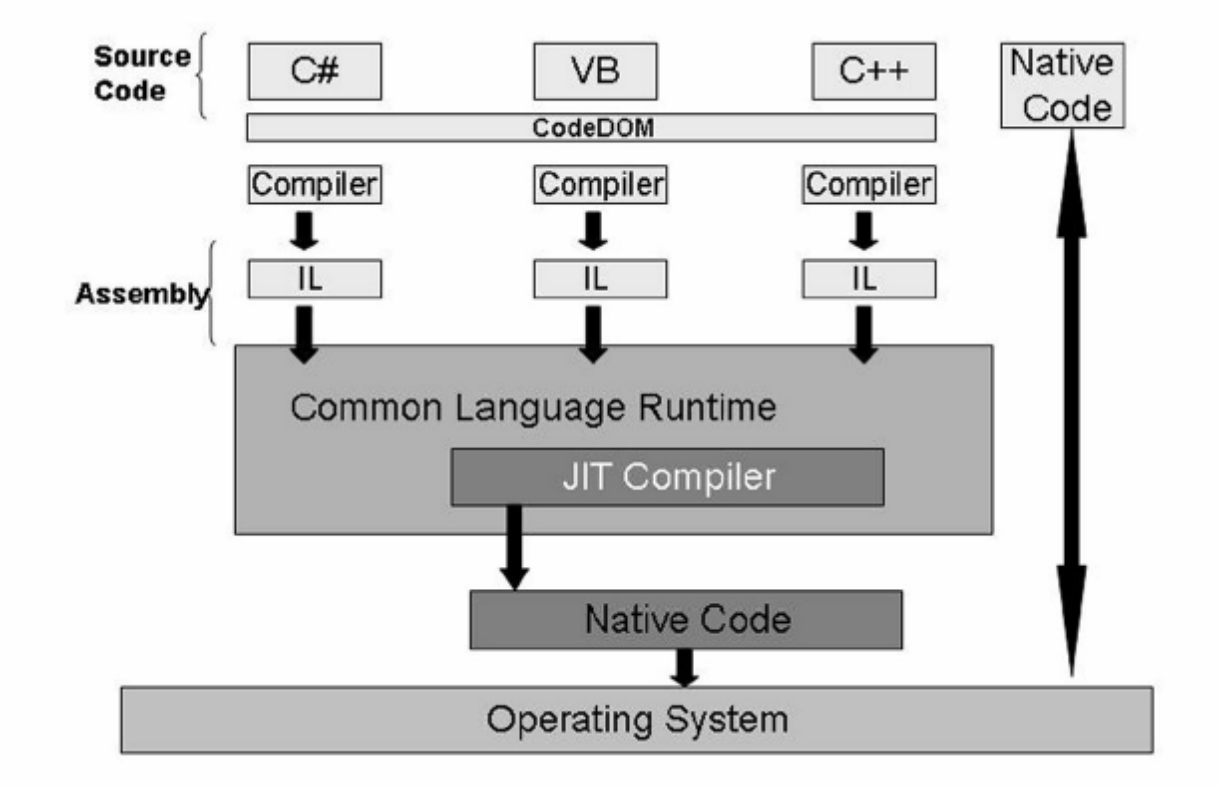
Bibliotecas CLI

Conjunto de bibliotecas que provee el framework. Entre las que nombraremos durante el presente trabajo, vale la pena destacar las siguientes:

- System.Reflection: Posibilita leer metadatos de un archive, así también como invocar y crear tipos dinámicamente.
- System.Reflection.Emit: Genera metadata y MSIL valiendose de los objetos provistos por el namespace anterior, y eventualmente permite volcarlos en un archivo CLI compatible.
- System.CodeDOM: Permite representar el árbol de sintaxis abstracto (AST) de un programa.
- System.CodeDOM.Compiler: Genera código IL a partir de AST anterior.

Entonces... ¿cómo se ejecuta finalmente una aplicación?

Habiendo introducido todos los conceptos mencionados en los apartados anteriores, vale la pena detenerse un instante más para aclarar qué pasará finalmente cuando se ejecute una aplicación. Nos valdremos para eso de un gráfico que aparece en [ASPECTC#]:



El programador elegirá un lenguaje cualquiera para escribir su programa .NET. Luego, el compilador, probablemente utilizando los objetos de CodeDOM provistos por el framework, generará los assemblies que contengan el código IL correspondiente. Ahora sí tenemos archivos listos para ser ejecutados por el virtual execution system, implementado en el gráfico a través del CLR. El componente de Just In Time Compiling irá traduciendo las instrucciones a código máquina en la medida en que esto sea necesario, para que puedan ser luego efectivamente ejecutadas. CLR utilizará su componente Cargador de Clases (*class loader*) para levantar los diferentes tipos del assembly antes que sean traducidos a código nativo.

Resumen

El objetivo de este anexo es introducir la plataforma .NET. Se explicó brevemente su historia y objetivos, destacándose la intención de que sea un ambiente *multilenguaje*: uno puede escribir un programa en cualquier lenguaje compatible. Una característica importante de los assemblies es que son autodescriptivos: contienen toda la información necesaria para su ejecución, incluyendo sus *metadatos*. Cada objeto de metadatos puede ser anotado por el usuario mediante *atributos*, que luego pueden ser consultados utilizando un framework de reflection.

Anexo II: Notación de los diagramas utilizados

Introducción[49]

Utilizamos tres tipos de gráfico para el modelado de los diseños ideados: los diagramas de objetos, los de secuencia y los de clases. Los dos primeros ilustran escenarios puntuales (el fin especifico de cada uno se detallará en el apartado correspondiente); el último muestra las clases definidas y sus respectivos protocolos. Un modelo es representado por un conjunto de diagramas cuya composición ilustra la forma en que funcionará finalmente la implementación. No se muestra cada pequeña situación que pudiera llegar a ocurrir; sí se intentan dar (¡o, al menos, eso creemos!) los elementos necesarios y suficientes para poder inducir cómo funcionará la aplicación ante cada eventualidad. Con la excepción de los diagramas de clase, nos referiremos siempre a escenarios puntuales: creemos que esta forma de expresión permite una mejor comprensión del problema, evitando generalizaciones que compliquen los desarrollos pero carezcan de utilidad práctica, garantizando a la vez que se resuelva cada pormenor de esa cuestión particular. Como ya dijimos en algún momento de la tesis, el diablo está en los detalles. A continuación procederemos con su descripción.

Diagrama de objetos

Muestra la estructura estática de un modelo en un intervalo acotado de tiempo, restringido por lo general a una cantidad de colaboraciones determinadas. Grafica los objetos intervinientes, representados a través de “cajitas”, y las relaciones entre ellos (flechas). Si un objeto A está relacionado con otro B, entonces podrá eventualmente enviarle un mensaje. La otra forma en que se puede dar esto último es en el caso que el segundo haya sido enviado como argumento en algún mensaje anterior.

No es indispensable graficar cada objeto y cada relación que exista en el intervalo señalado, sino sólo aquellas necesarias para comunicar la idea que se está queriendo expresar. Dado que se trata de representar situaciones puntuales, cada “cajita” debe indicar una instancia determinada.

Existen dos maneras para representar relaciones con colecciones, utilizadas de acuerdo a la importancia con que se desee mostrar los elementos que la componen: la primera es mediante una doble fecha (como aparece en el ejemplo), y la segunda es detallando la relación con cada uno de los elementos que la componen. Vale la pena señalar, por último, que es posible indicar también sobre las flechas el nombre con el que se “conoce” la relación, si esto ayuda en la interpretación del diagrama.

[49] La notación y los conceptos utilizados son los definidos por la cátedra de Orientación a Objetos de la FCEN, UBA. Es similar a un subconjunto simplificado y reducido de los correspondientes diagramas UML.

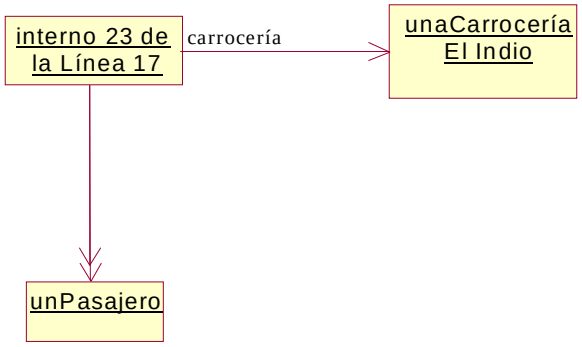


Diagrama de secuencia

Representa las colaboraciones efectuadas por un conjunto de objetos para resolver determinada tarea. Al igual que en el caso anterior, se resuelve una situación puntual. En consecuencia, si debiese mostrarse una repetición o un condicional se deberá graficar la secuencia efectuada en el escenario que se esté describiendo: en el primer caso, cada una de las iteraciones realizadas; en el segundo, las acciones ejecutadas a partir de la decisión tomada. Si se necesitase clarificar más la situación, puede escribirse una nota aclaratoria o dibujarse un nuevo diagrama representando el escenario en donde se tome la decisión contraria.

Los mensajes se escriben en notación Smalltalk. No es necesario mostrar cada valor de retorno, sino solamente aquellos que resulten relevantes; se indican anteponiendo en símbolo "^". Nuevamente, al tratarse de escenarios puntuales, no se escribe en los diagramas la forma genérica de la signatura de los mensajes (ej: `sumar: unNumero con:otroNumero`), sino que se detallan los argumentos con los que se esté trabajando (`sumar: 2 con: 5`). Las colecciones son señaladas por el símbolo numeral "#" seguido de una lista de elementos encerrada entre paréntesis (#(1 2 3) por ejemplo corresponde a la colección conformada por los objetos 1, 2 y 3).

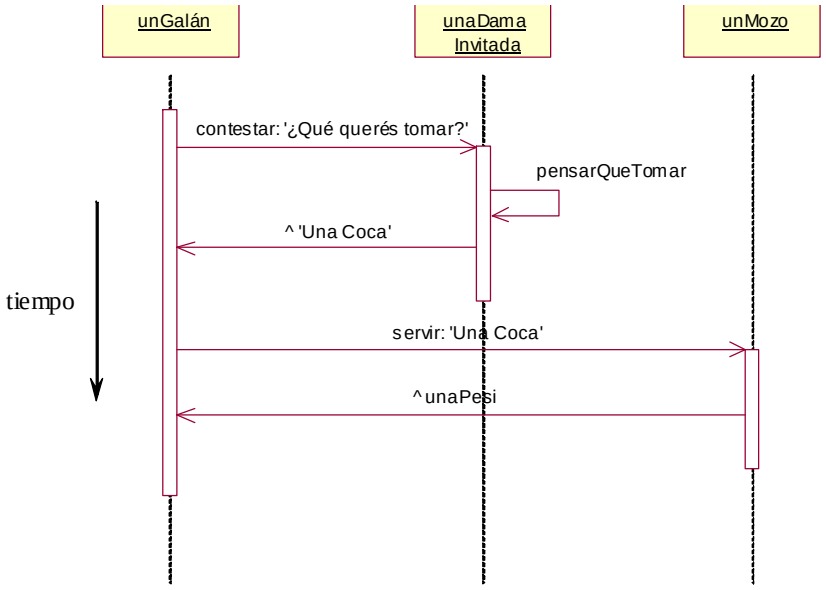
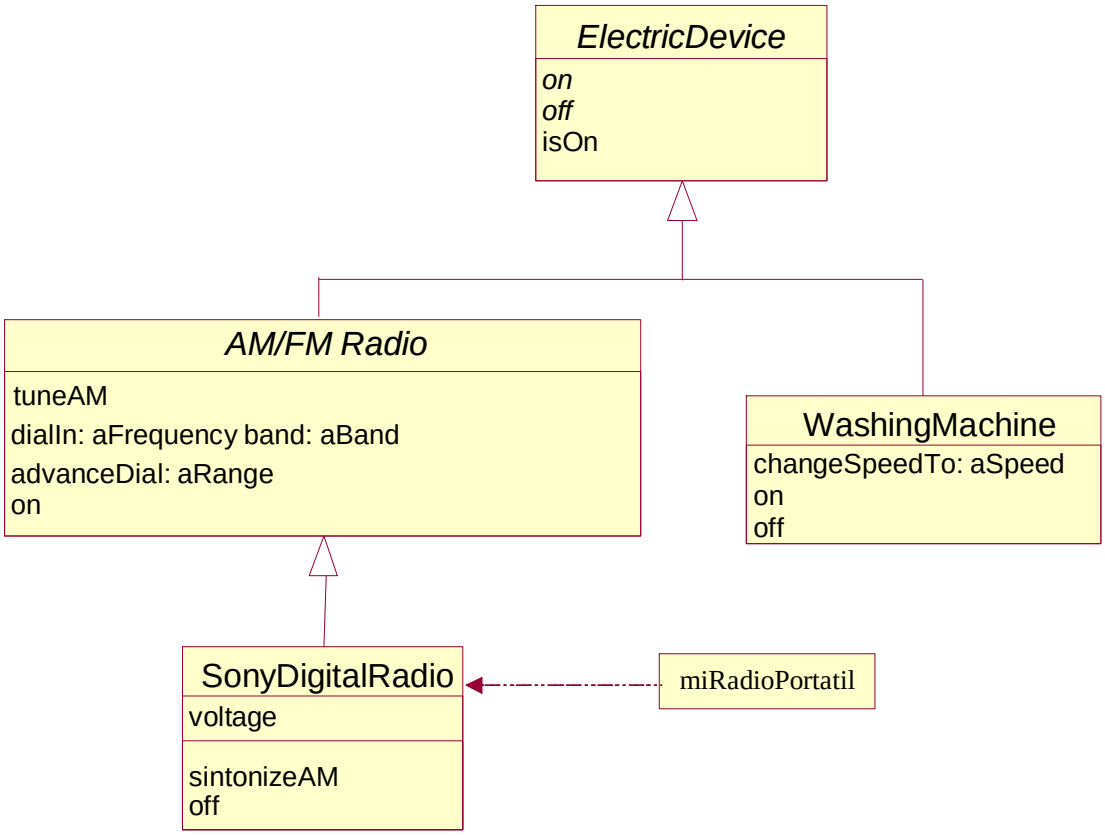


Diagrama de clases

Descripción de las clases intervinientes en un modelo y el vínculo de herencia existente entre ellas. A diferencia de UML, no se incluyen relaciones entre instancias (agregaciones, composiciones, asociaciones), por considerarse que corresponden a otro nivel de abstracción y no contribuyen a la comprensión del diagrama, asumiéndose que en caso de ser necesarias estarán explicitadas en los diagramas de objetos correspondientes. Se incluye tanto el protocolo que poseen las instancias de las clases como el que poseen ellas mismas, representadas por objetos de primera clase en ambientes Smalltalk o a través de constructores y métodos estáticos en plataformas híbridas, .NET en el caso de la presente tesis.

Cada cajita representa una clase. Se puede escribir el nombre en itálica o escribir a su lado una “A” para indicar que se trata de una abstracta. Debajo del nombre, separados por una raya, aparecen los métodos de clase y, por último, los de instancia. De figurar una única separación, se asume que se trata de métodos de instancia. Que un mensaje aparezca tanto en una subclase como en su superclase indica que ha sido redefinido. Si su nombre es especificado en itálica, se trata de un método abstracto.

Pueden figurar también comentarios o, excepcionalmente, el detalle de la implementación de un método puntual o un ejemplo de instancia de esa clase, ambos relacionados con la misma a través de una flecha punteada.


ElectricDevice
on
off
isOn
AM/FM Radio
tuneAM
dialIn: aFrequency band: aBand
advanceDial: aRange
on
WashingMachine
changeSpeedTo: aSpeed
on
off
SonyDigitalRadio
voltage
sintonizeAM
off
miRadioPortatil

Anexo III: Diseño del Senku[50]

Introducción

El programa desarrollado respeta la siguiente consigna:

Implementar el **Senku** (alias *Lame game*), y las siguientes variantes:

a) En un tablero de 5x5 colocar nueve fichas en las casillas centrales. El desafío es, como en el Senku, dejar una sola ficha, en la casilla central, en la menor cantidad posible de movimientos (que son como en el Senku, agregando el salto en diagonal). Fue propuesto por Sam Lloyd y hay una solución en 8 movimientos.

b) La variante Dudeney: Se trata de contar todos los saltos seguidos de una misma ficha como un solo movimiento. Hay una solución en 4 movimientos.

c) La variante Gardner: Aquí los movimientos pueden ser saltos, como antes, o pasos. Un paso es el desplazamiento de una ficha a una casilla adyacente, en cualquiera de las ocho direcciones, que esté vacía. Hay una solución en sólo 3 movimientos. Al igual que en la variante Dudeney, los saltos seguidos de una misma ficha cuenta como un solo movimiento.

En las variantes, la cantidad de movimientos está indicada sólo como desafío y aliciente, no es necesario que programen el juego para que encuentre la solución (aunque, si les sobra algo de tiempo, no es difícil y pueden consultarnos). Lo que sí es necesario es que su interfase permita elegir la variante, o sea, que todas las variantes estén incluidas en un solo "programa", y no que nos den una aplicación por variante.

La arquitectura de la solución presenta un esquema similar al clásico Model-View-Controller. Identificamos dos grandes grupos de objetos: aquellos que representan la interfaz mediante la cual el usuario interactuará con el juego, que agruparemos genéricamente bajo el título de *Morphs* o *vistas*, y aquellos que representan su modelo, sin preocuparse por nada que tenga que ver con la responsabilidad anterior, a los que (¡vaya coincidencia!) denominaremos *model*. Este esquema nos permite separar claramente la lógica de la presentación, permitiendo eventualmente agregar nuevas vistas al juego o modificar las ya existentes sin impactar en el modelo que está funcionando.

Mediante el esquema anterior, estamos intentando desacoplar el modelo de sus posibles representaciones. Un tema a tratar entonces es la forma en la que se comunicarán ambos componentes arquitectónicos. Para ello usaremos el concepto de *observer* [GAMMA], en donde habrá una cantidad no definida a priori de componentes de las vistas suscriptos a los eventos significativos del modelo, y de esta forma irán conociendo cuándo actualizar su representación. Las vistas, por el contrario, sí enviarán los mensajes que

[50] Este capítulo se apoya fuertemente en [SENKU]

correspondan a objetos del modelo al recibir feedback del usuario. De esta forma, los objetos de las vistas tienen conocimiento del modelo, pero no así a la inversa.

Se presentan a continuación los diagramas y explicaciones correspondientes a la versión original en Squeak. Se incluye al final un apartado en donde se detallan las diferencias con respecto a la implementación en .NET.

Model

Descripción breve

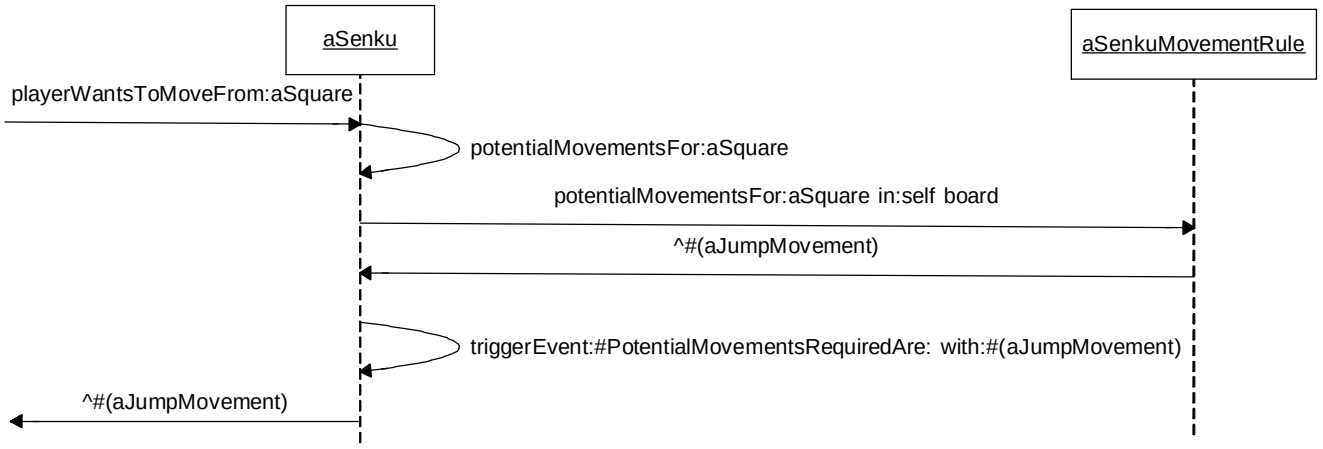
La pregunta directriz que consideramos al explorar el dominio fue, ineludiblemente, cómo se desarrollaría una partida. Esto representó un inicio para luego adentrarnos y preguntarnos el detalle: ¿Qué elementos tienen en común el Senku, el Dudeney, el Gardner y el Lloyd? ¿Cómo se construyen? ¿Cuándo se considera que una partida fue ganada (o perdida!)? ¿Cómo se mueve una ficha? Encontrar una respuesta a estas preguntas representó un proceso iterativo, en el que el planteo de la solución a cada problema requirió la revisión de lo anteriormente resuelto.

En esta sección se detalla la solución que encontramos a las preguntas anteriores. Se comienza por diagramas de secuencia que describen cómo se realiza un movimiento, y luego se evoluciona hacia la forma de conteo y finalización de la partida. A continuación se detalla la creación de la misma, a partir de diagramas de objetos que ejemplifican su estructura “estática”. Encontramos por último al diagrama de clases utilizadas.

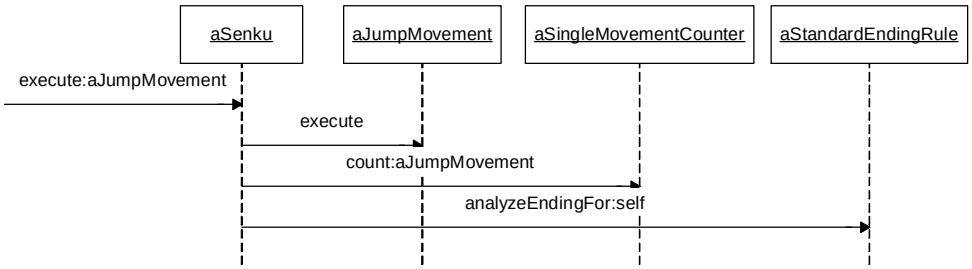
Diseño

Secuencia: Pidiendo movimientos válidos a partir de una casilla de un Senku

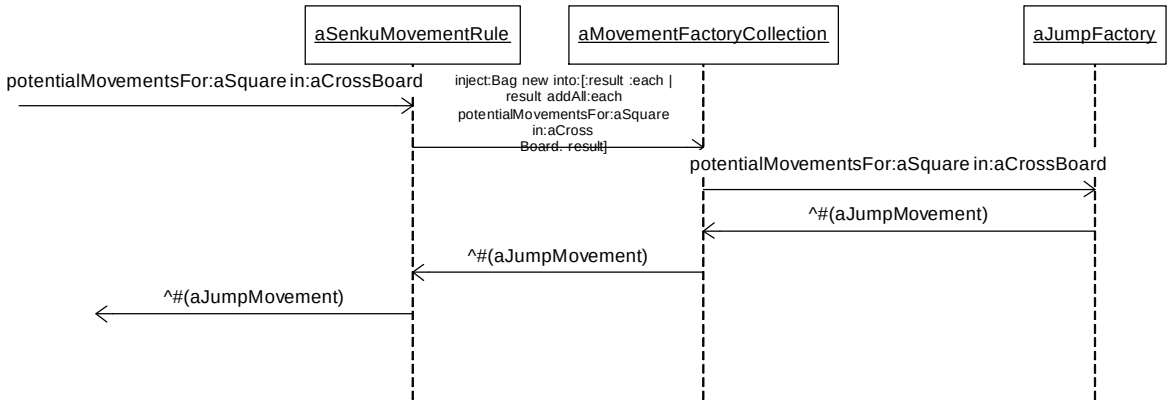
Cuándo un jugador desee mover una ficha desde una casilla, se verifica qué movimientos podrían llevarse a cabo. Si hubiese más de uno posible, será indispensable que él decida qué hacer; caso contrario, podrá ejecutarse automáticamente, pedirse una confirmación, o cualquier otra opción que pueda imaginar el diseñador de turno: los pasos a seguir dependerán finalmente de la interfaz. En el ejemplo ilustrado, existe un único movimiento posible, que es el que se responde como resultado (nótese que la respuesta es igualmente una colección, en este caso de un único elemento, ya que potencialmente podrían ser varios).


aSenku
aSenkuMovementRule
playerWantsToMoveFrom:aSquare
potentialMovementsFor:aSquare
potentialMovementsFor:aSquare in:self board
^#(aJumpMovement)
triggerEvent:#PotentialMovementsRequiredAre: with:#(aJumpMovement)
^#(aJumpMovement)

Secuencia: Ejecutando un movimiento válido en un Senku

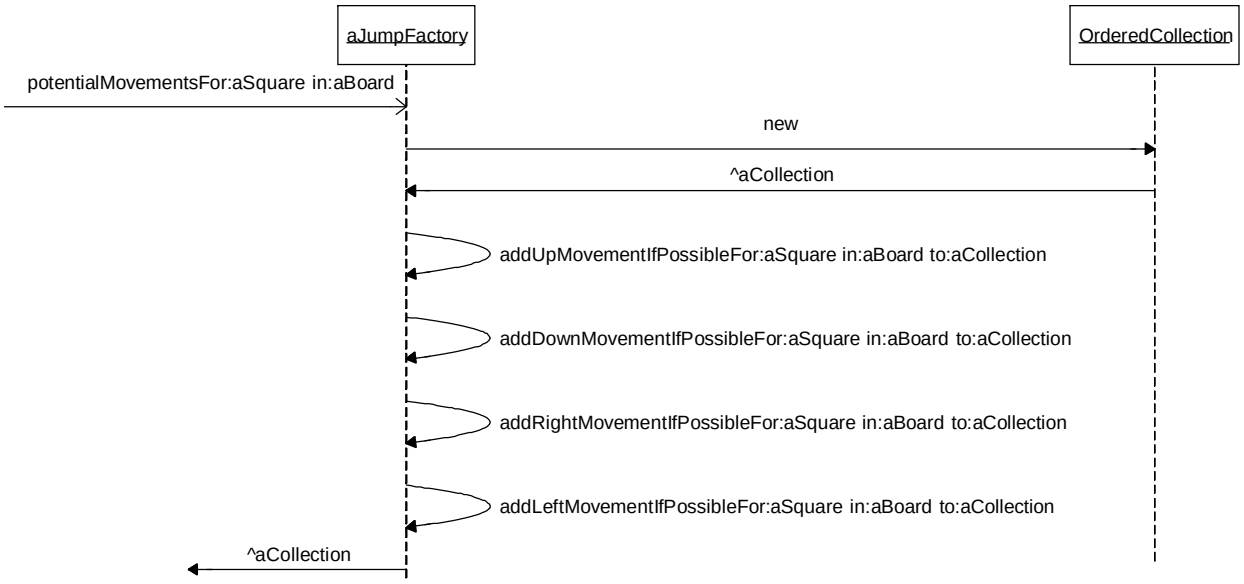


Secuencia: Generando movimientos válidos en un Senku, desde una casilla con un único movimiento posible

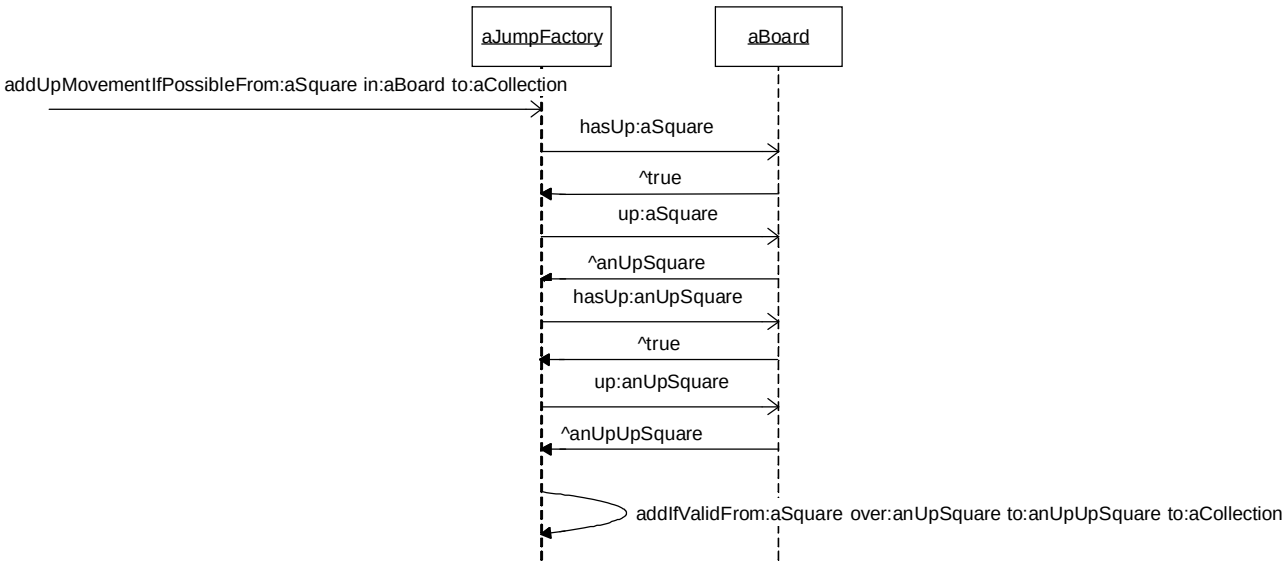


Secuencia: Generando movimientos válidos en un Gardner, desde una casilla con dos movimientos posibles

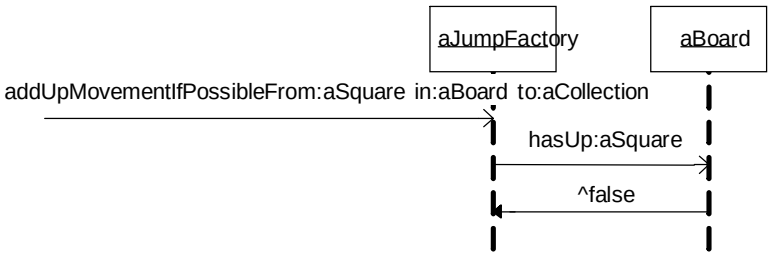
Secuencia: Detalle de la generación de los movimientos del Senku



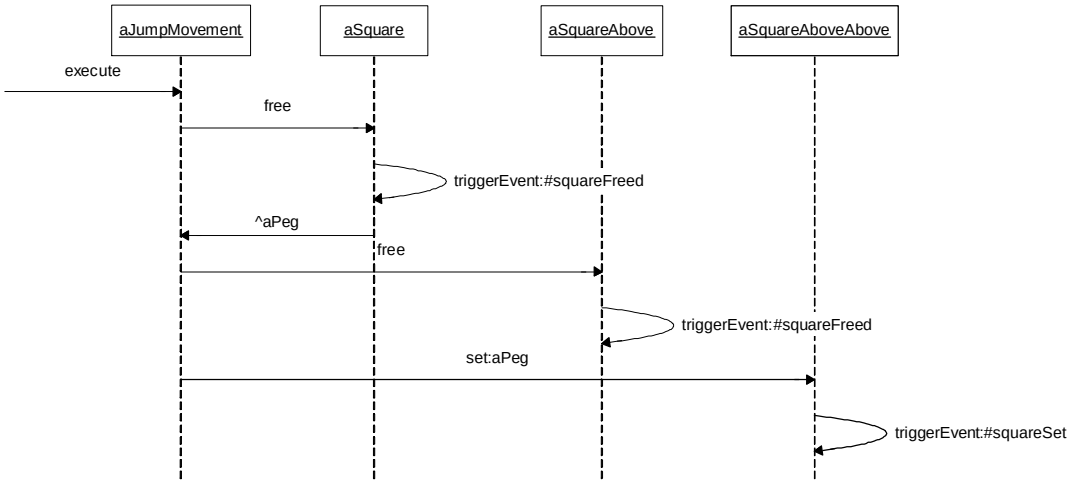
Secuencia: Detalle del addUpMovementIfPossible cuando hay casillas superiores



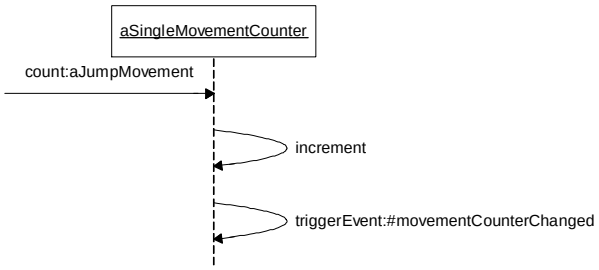
Secuencia: Detalle del addUpMovementIfPossible cuando no existe la casilla inmediatamente superior



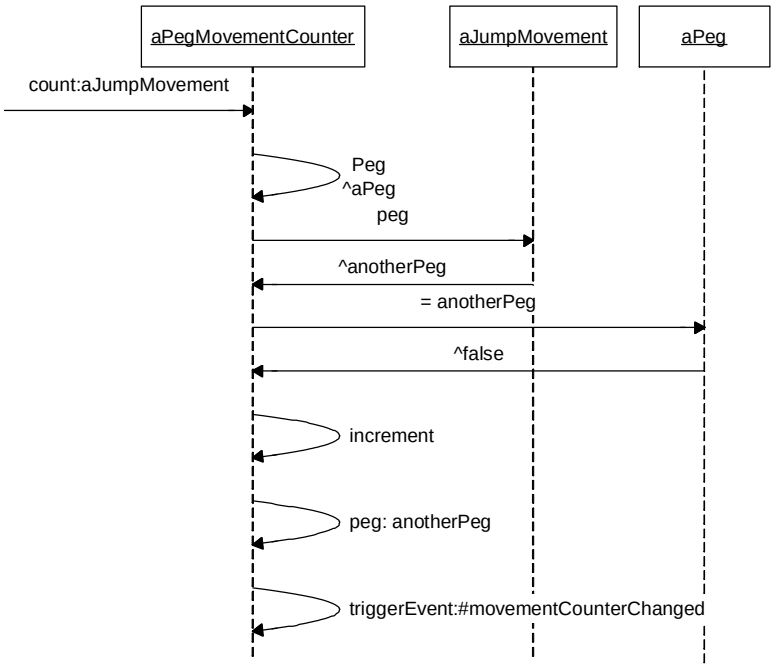
Secuencia: Detalle de la ejecución un movimiento



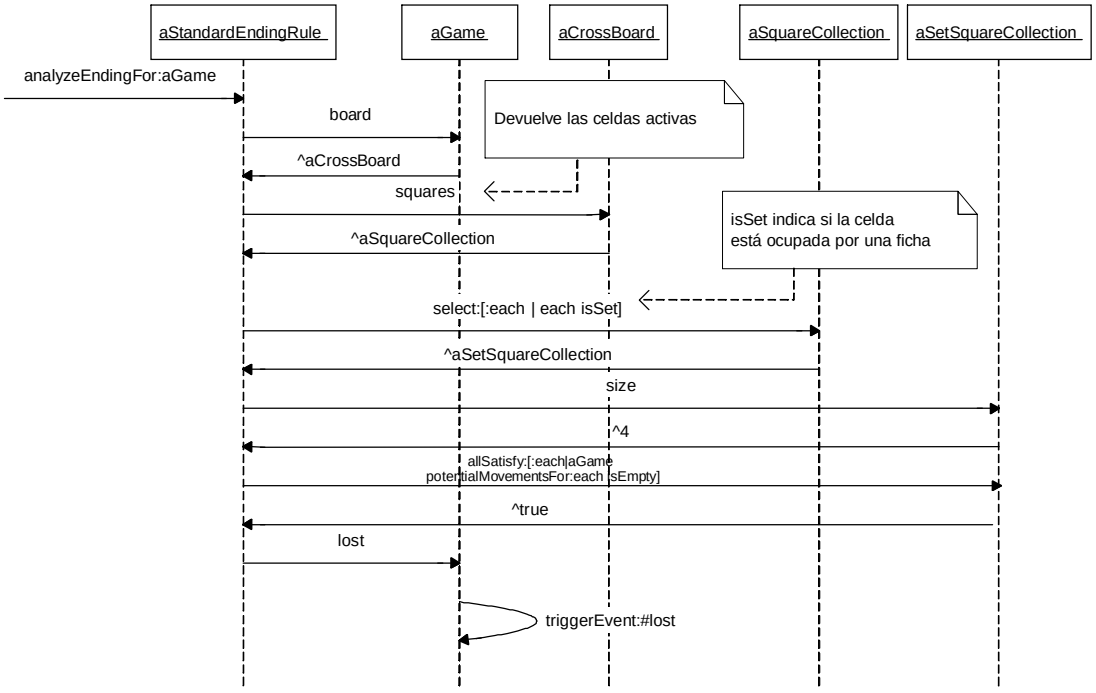
Secuencia: Detalle del conteo de movimientos con un contador simple (de los que usa el Senku)



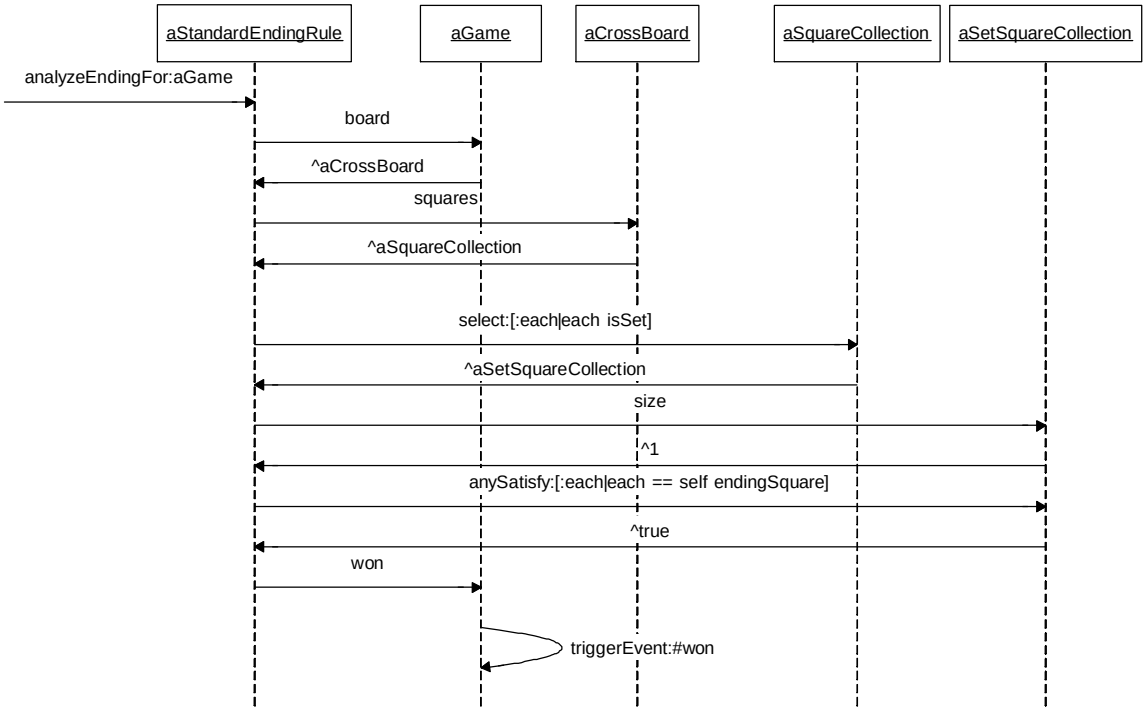
Secuencia: Detalle del conteo de movimientos con un contador tipo variante Gardner



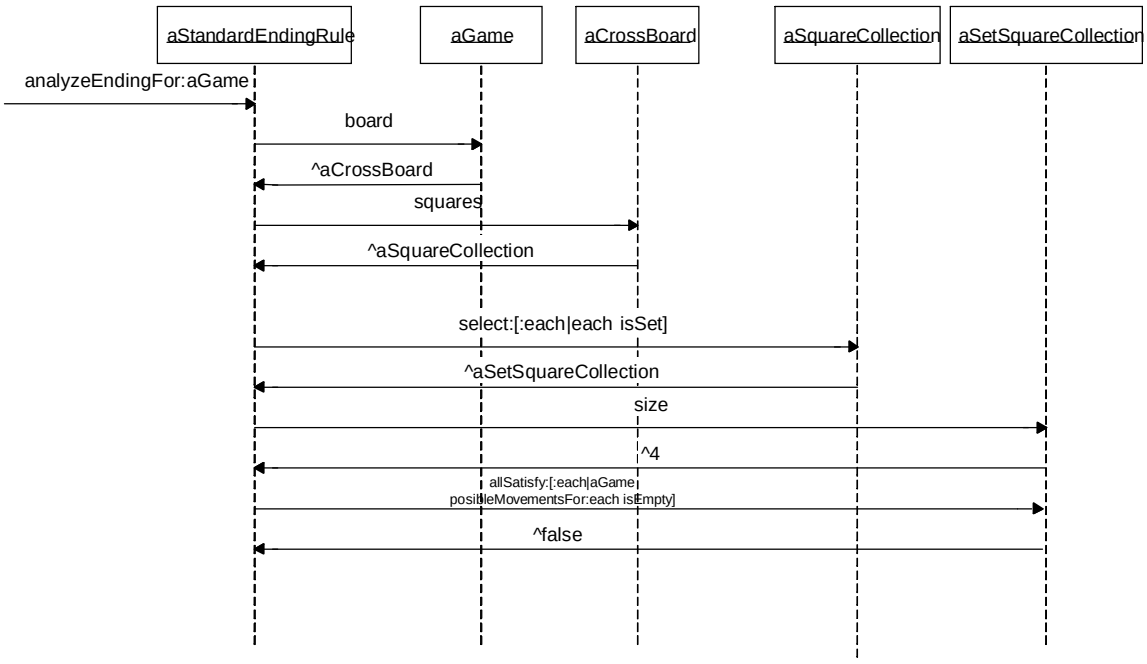
Secuencia: Detalle del análisis de finalización de una partida perdida



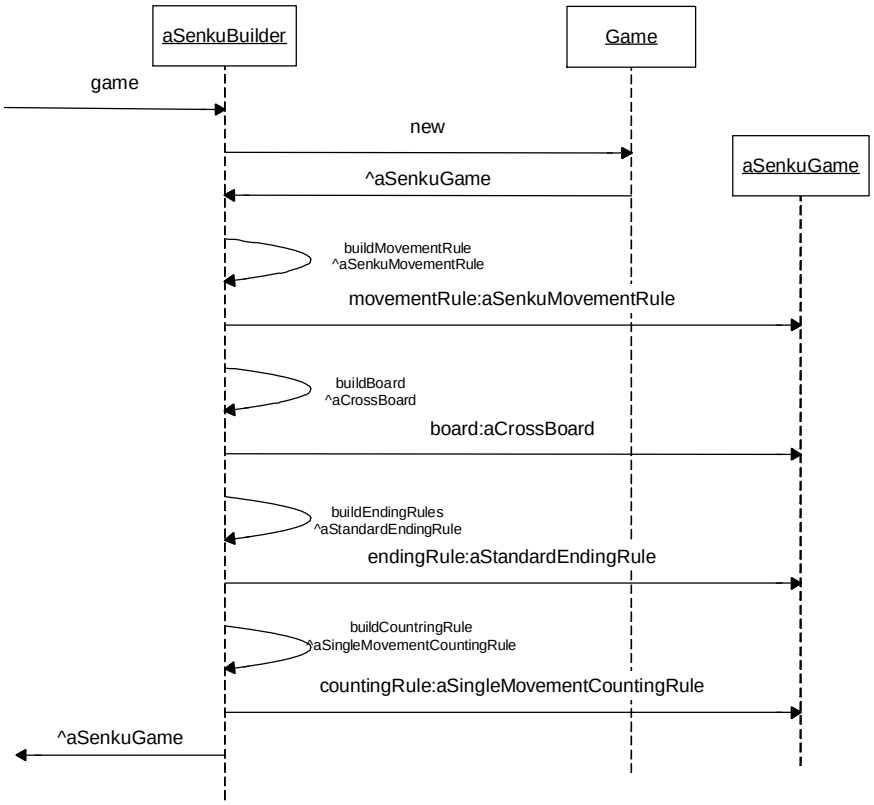
Secuencia: Detalle del análisis de finalización de una partida ganada



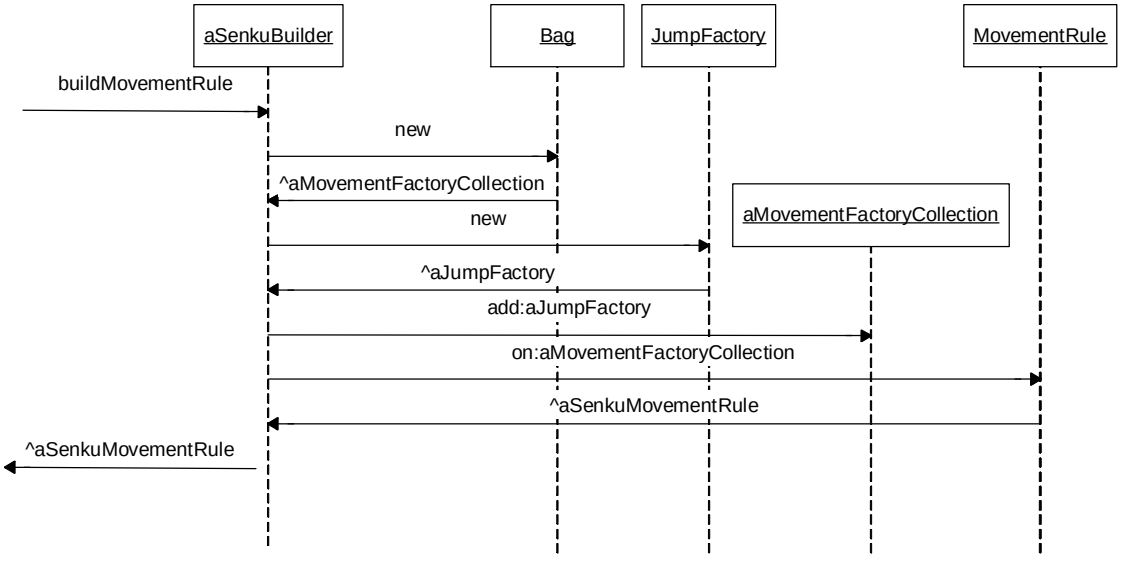
Secuencia: Detalle del análisis de finalización de una partida aún no terminada



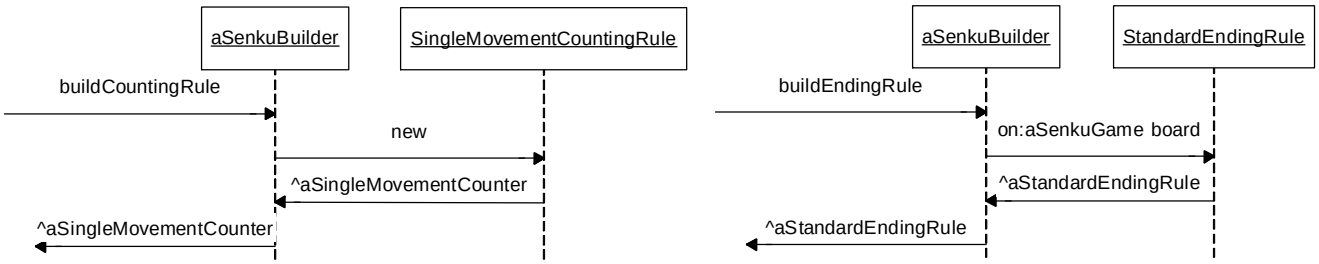
Secuencia: Creación de un Senku



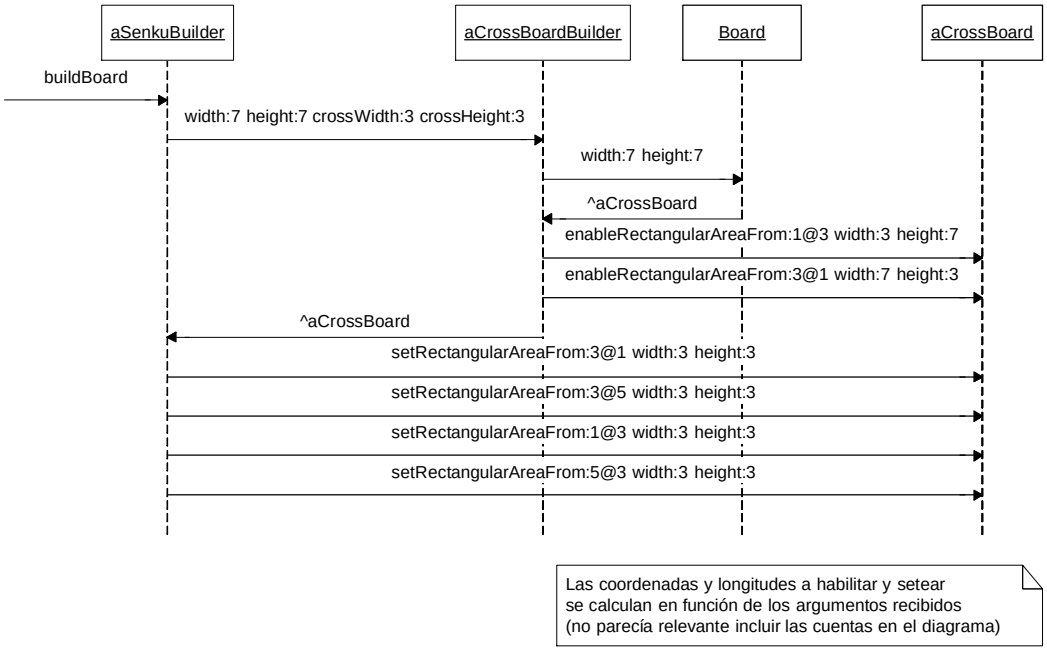
Secuencia: Detalle de creación de reglas de movimiento en un Senku

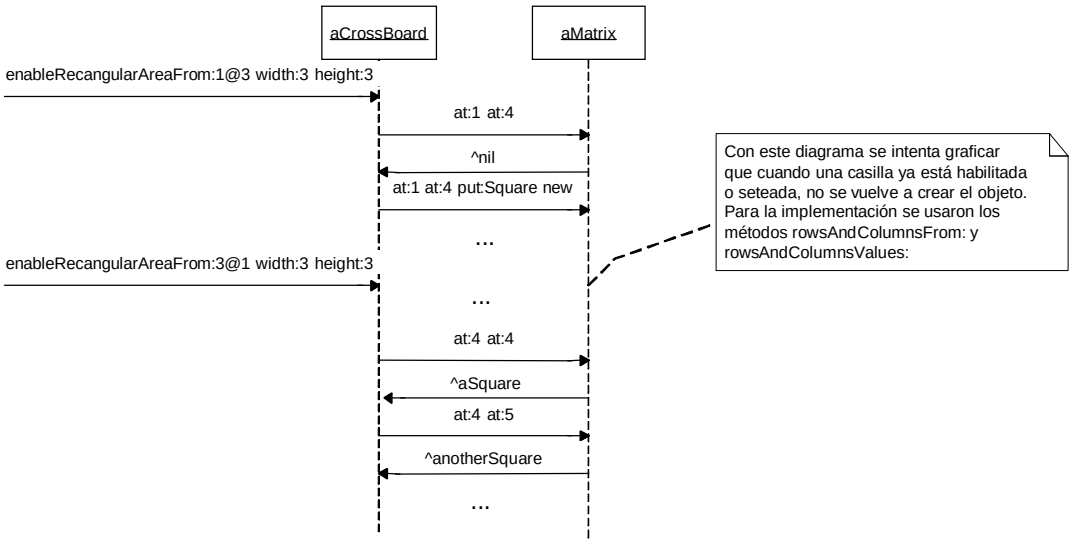


Secuencia: Detalle de creación de reglas de conteo y fin de partida de un Senku

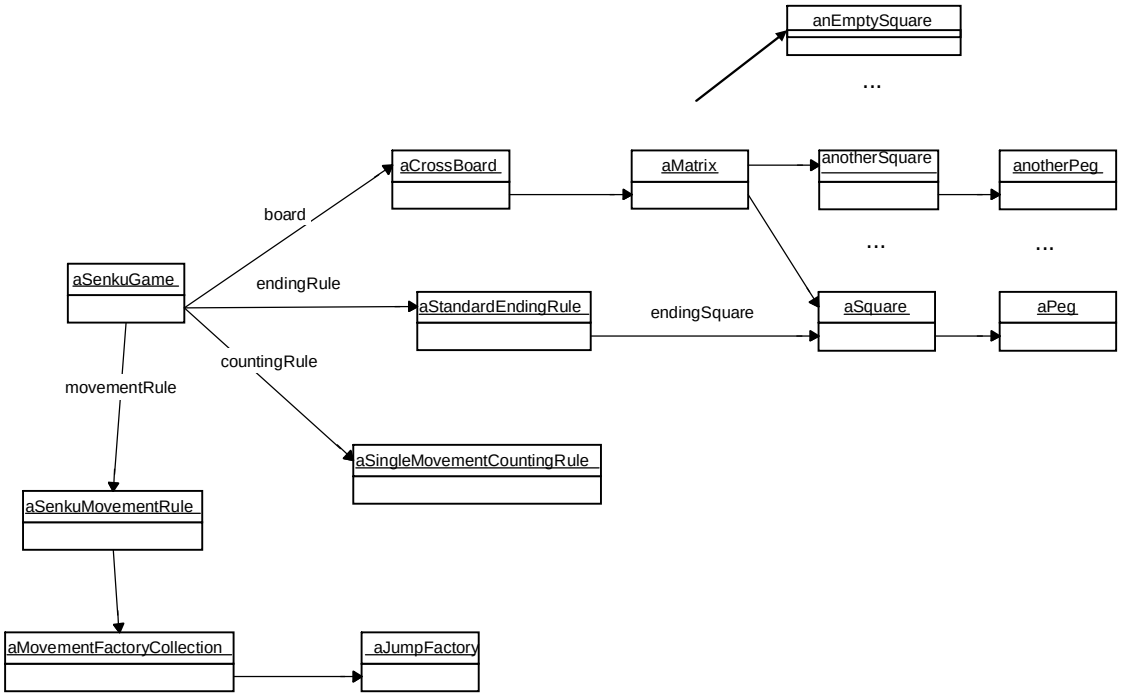


Secuencia: Detalle de creación del tablero en un Senku

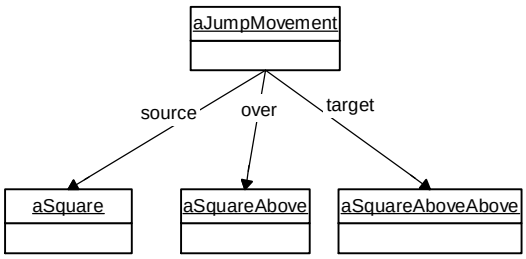




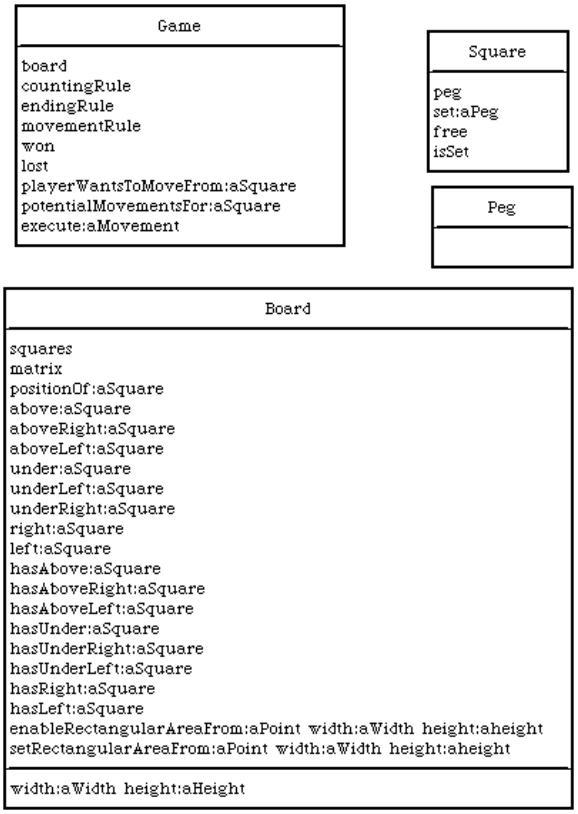
Objetos: Senku ya creado



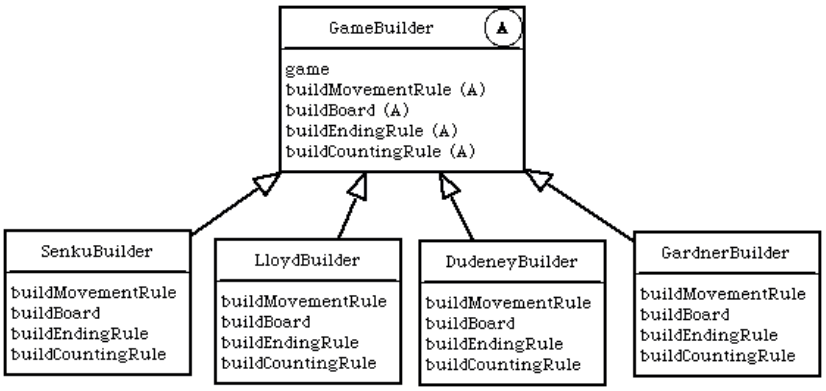
Objetos: Detalle de un Jump Movement

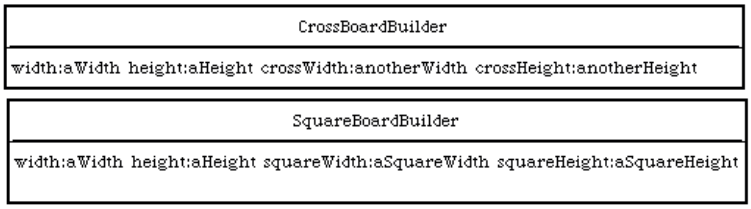


Clases de estructura estática

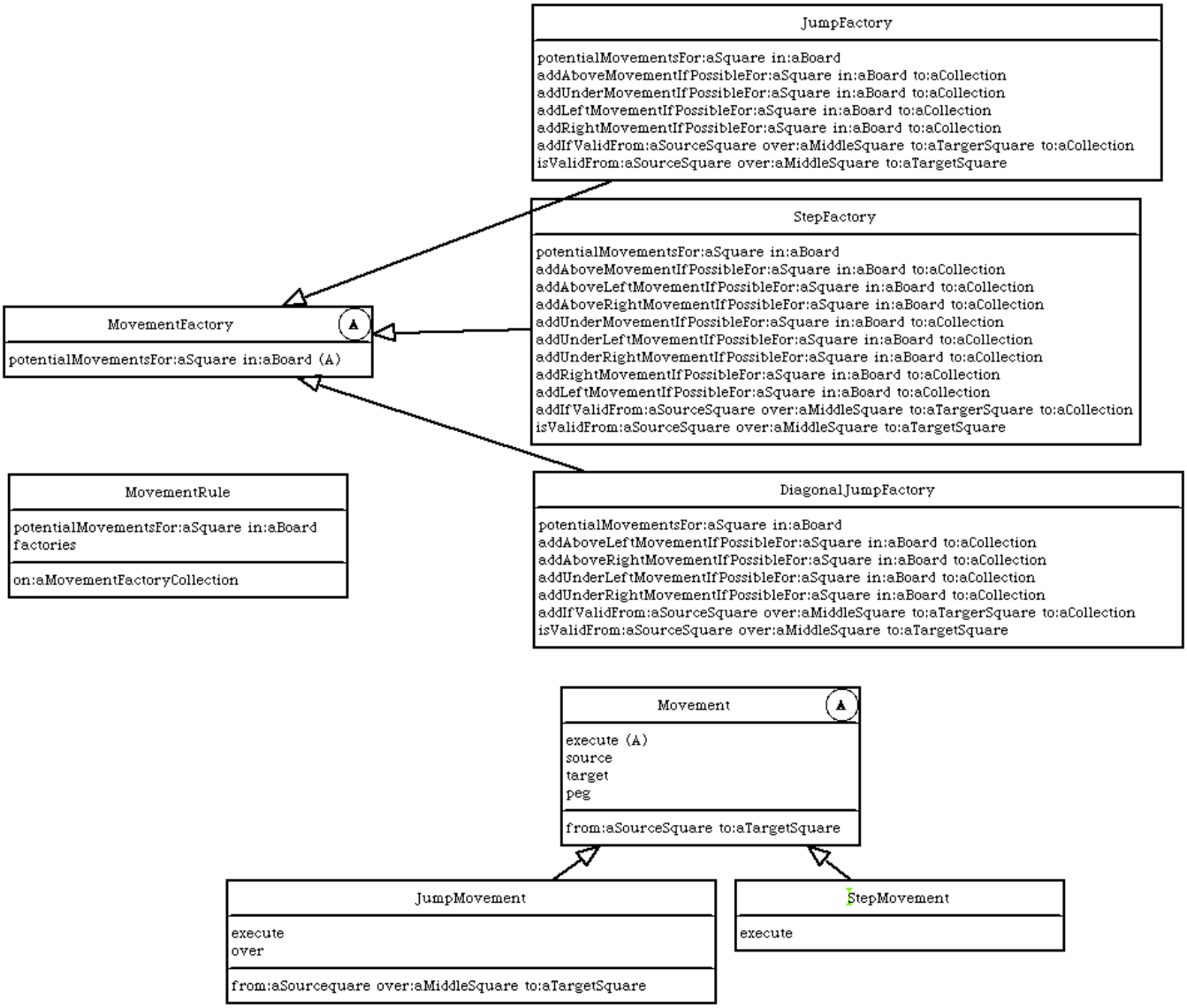


Clases de Builders

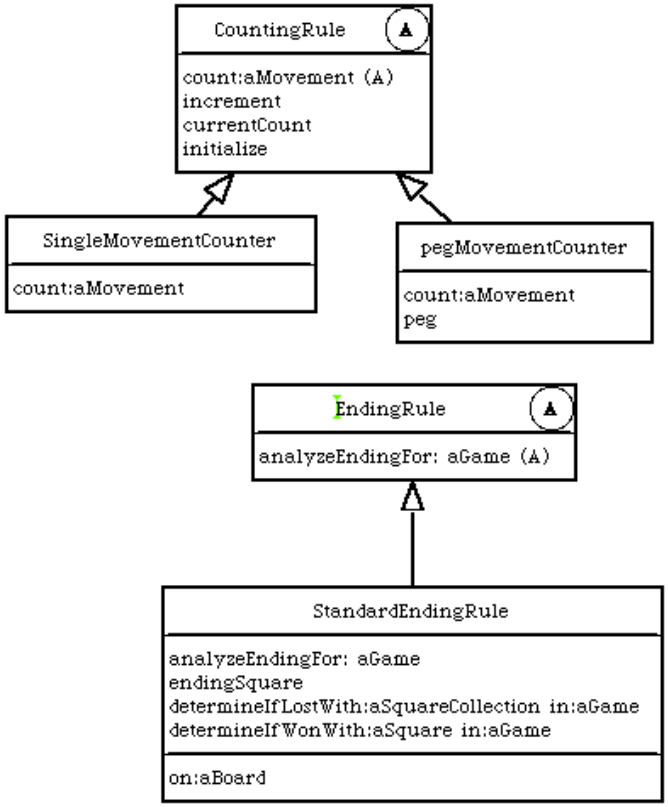




Clases de estructura dinámica


JumpFactory
potentialMovementsFor:aSquare in:aBoard
addAboveMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addUnderMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addLeftMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addRightMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addIfValidFrom:aSourceSquare over:aMiddleSquare to:aTargerSquare to:aCollection
isValidFrom:aSourceSquare over:aMiddleSquare to:aTargetSquare
StepFactory
potentialMovementsFor:aSquare in:aBoard
addAboveMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addAboveLeftMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addAboveRightMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addUnderMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addUnderLeftMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addUnderRightMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addRightMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addLeftMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addIfValidFrom:aSourceSquare over:aMiddleSquare to:aTargerSquare to:aCollection
isValidFrom:aSourceSquare over:aMiddleSquare to:aTargetSquare
MovementFactory
A
potentialMovementsFor:aSquare in:aBoard (A)
MovementRule
potentialMovementsFor:aSquare in:aBoard
factories
on:aMovementFactoryCollection
DiagonalJumpFactory
potentialMovementsFor:aSquare in:aBoard
addAboveLeftMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addAboveRightMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addUnderLeftMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addUnderRightMovementIfPossibleFor:aSquare in:aBoard to:aCollection
addIfValidFrom:aSourceSquare over:aMiddleSquare to:aTargerSquare to:aCollection
isValidFrom:aSourceSquare over:aMiddleSquare to:aTargetSquare
Movement
A
execute (A)
source
target
peg
from:aSourceSquare to:aTargetSquare
JumpMovement
execute
over
from:aSourcequare over:aMiddleSquare to:aTargetSquare
StepMovement
execute

Clases de Reglas



Morph

Descripción breve

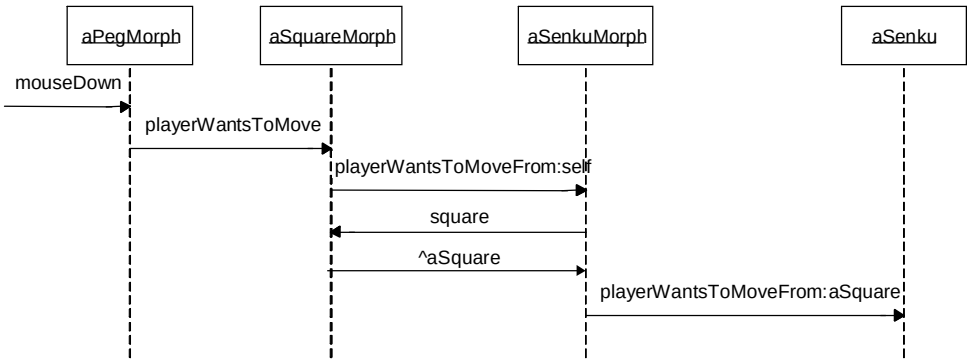
En la presente sección se describe el diseño de la interfaz implementada para el Senku. Siguiendo con los lineamientos planteados en la introducción, definimos una posible vista del modelo; podrían eventualmente desarrollarse la cantidad y variedad que se desee. Cada una deberá obviamente corresponderse de alguna forma con el modelo subyacente. En nuestra implementación, supusimos que lo primero que haría el usuario cuando quiera jugar será seleccionar la variante. Hecho esto, comenzará la partida, que continuará hasta que gane o pierda. En cualquier momento podrá iniciar un nuevo juego. Para ejecutar una jugada, seleccionará una ficha. Si hay un único movimiento posible, éste se ejecutará automáticamente. De haber más, se elegirá entre los elementos de este conjunto.

Antes de entrar de lleno en el diseño, valen algunas aclaraciones:

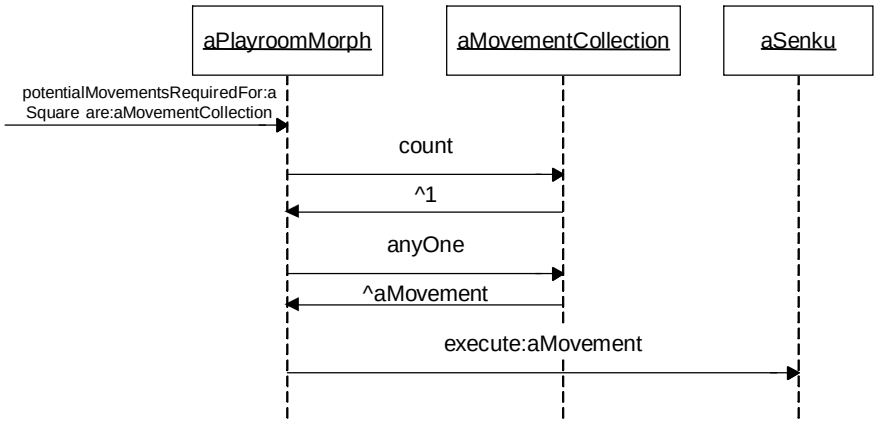
- Se incluyeron en las colaboraciones y en el protocolo de las clases sólo aquel que consideramos relevante para comprender el funcionamiento del sistema. No entramos en detalles que consideramos meramente estéticos.
- En el diagrama de clases no se incluye ninguna relación de especialización con las clases del paquete Morphic [MORPHIC], porque consideramos que no aporta a la comprensión del diseño, pero vale destacar que en él nos basamos para la implementación de la parte gráfica.

Diseño

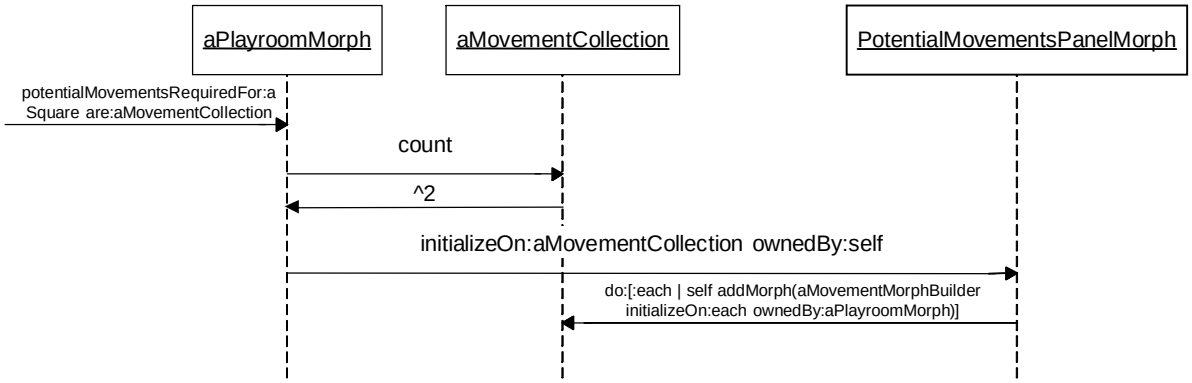
Secuencia: Un jugador solicita mover una ficha en un Senku



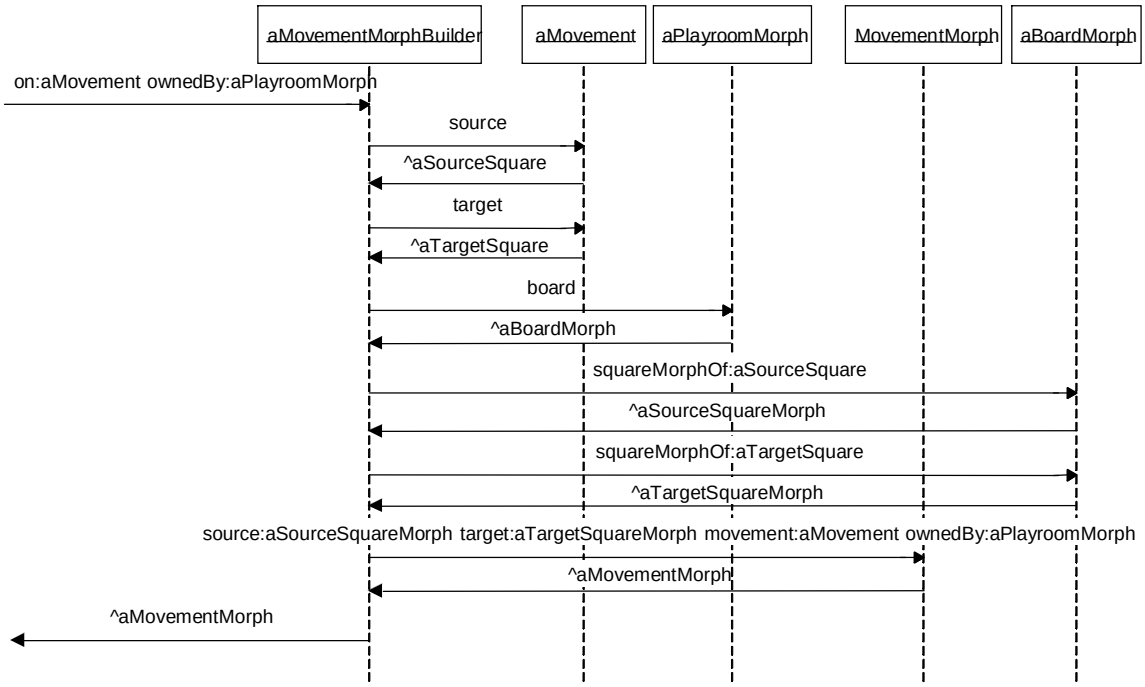
Secuencia: El playroomMorph es avisado de la solicitud de movimientos posibles, y hay uno solo



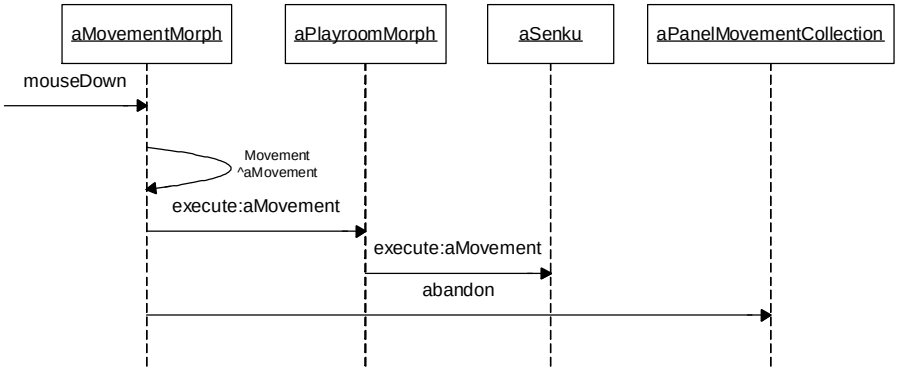
Secuencia: El playroomMorph es avisado de la solicitud de movimientos posibles, y hay más de uno


aPlayroomMorph
aMovementCollection
PotentialMovementsPanelMorph
potentialMovementsRequiredFor:a Square are:aMovementCollection
count
^2
initializeOn:aMovementCollection ownedBy:self
do:[:each | self addMorph(aMovementMorphBuilder initializeOn:each ownedBy:aPlayroomMorph)]

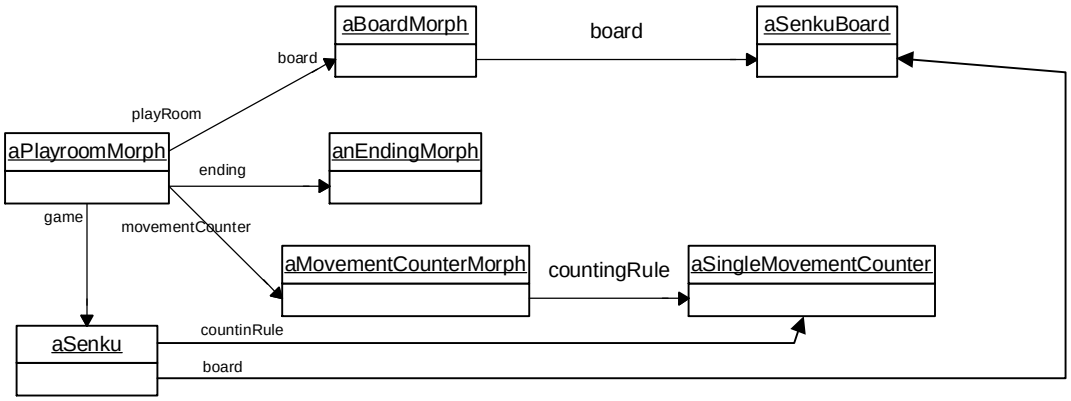
Secuencia: Construcción de un MovementMorph



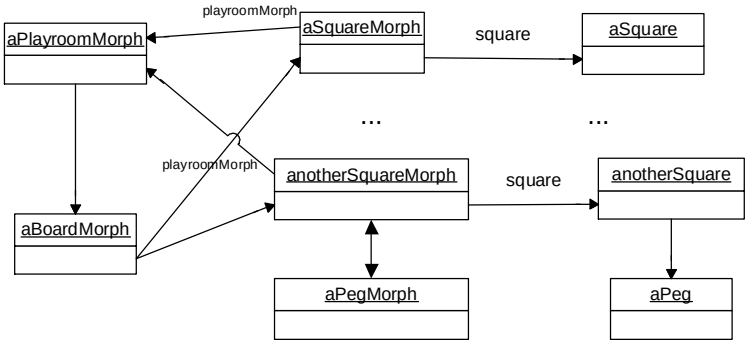
Secuencia: Ejecución de un movementMorph



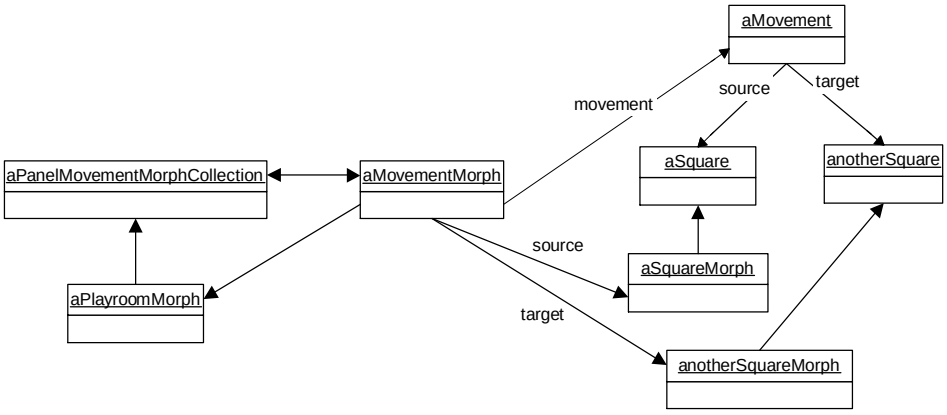
Objetos: Playroom y su contraparte del modelo



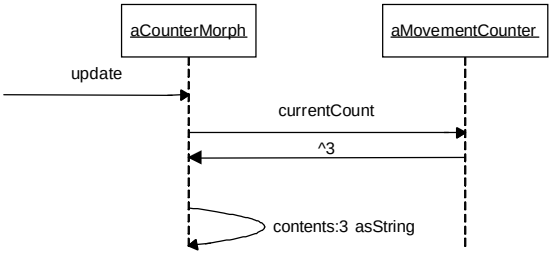
Objetos: Detalle del tablero y sus colaboradores. Se muestran una casilla ocupada y una vacía.



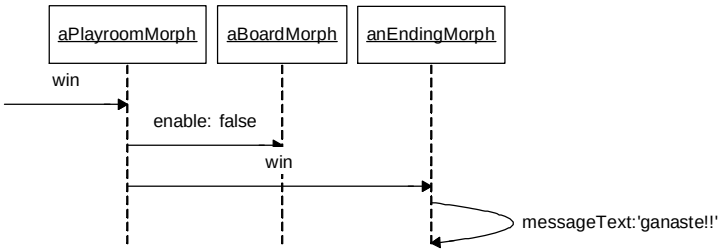
Objetos: Detalle de un movementMorph



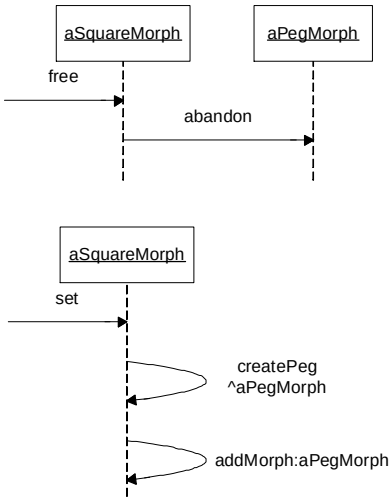
Secuencia: Se produce el evento de actualización del contador de movimientos (llegó a tres!)



Secuencia: Se produce el evento que indica que se ganó la partida

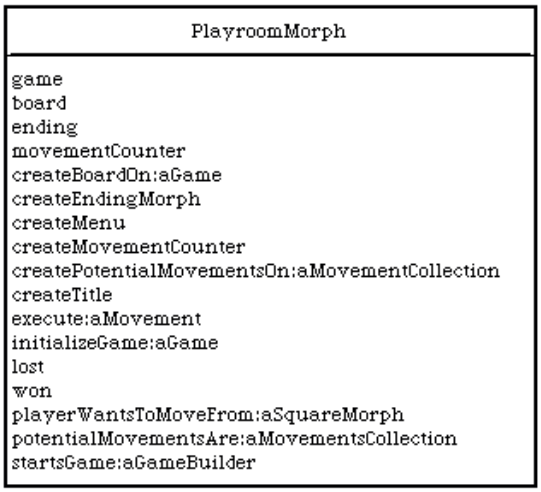


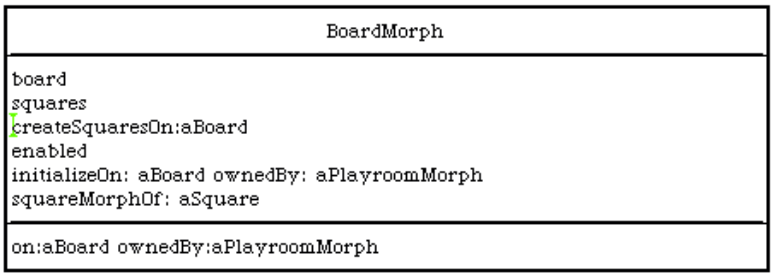
Secuencia: Se producen los eventos de quita y puesta de pegs sobre un square

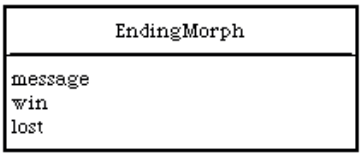


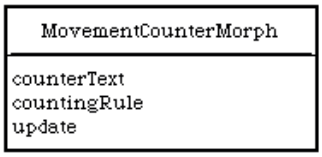
A diferencia del model, al ejecutarse un movimiento no interesa que la ficha que se saca del casillero de origen sea la misma que la que se coloca en el destino. Esto representaría mejor la realidad, a la vez que sería más flexible para juegos con esquemas de fichas más complejos. Sin embargo, para esta implementación en particular esto no aportaba ninguna ventaja, así que decidimos hacer que cada vez que se ejecuta un moviemiento se crean y destruyen nuevas fichas.

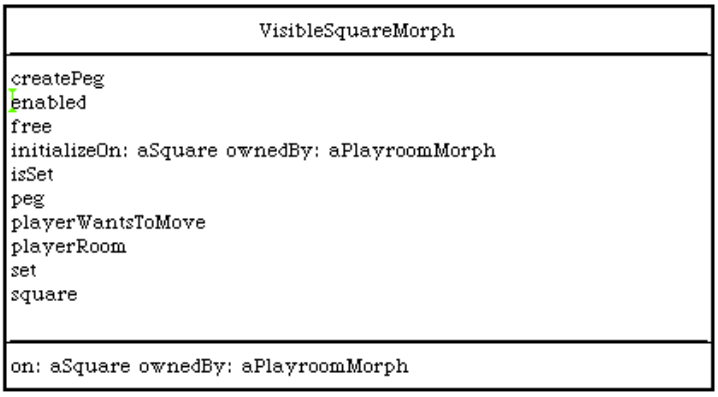
Clases de elementos del juego

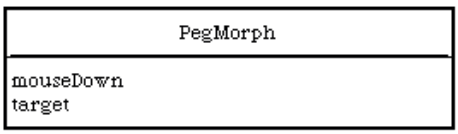




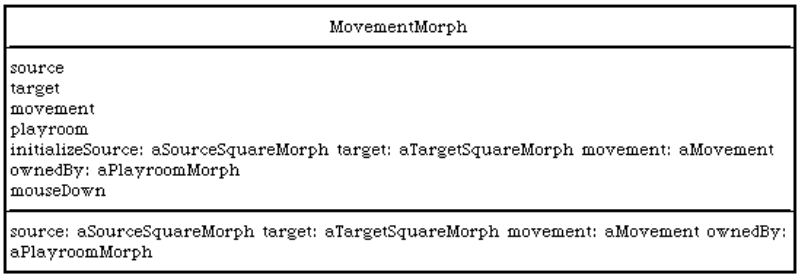


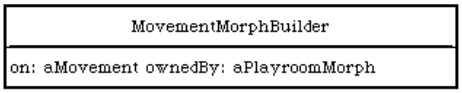


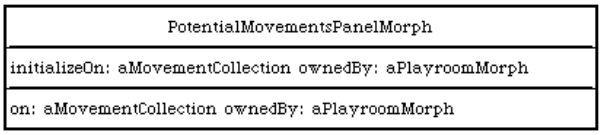




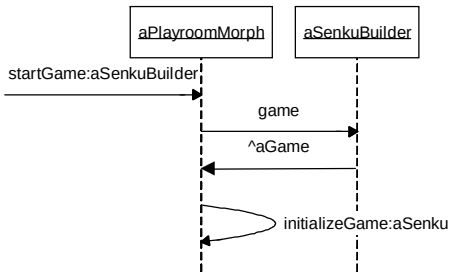
Clases morph de los movimientos



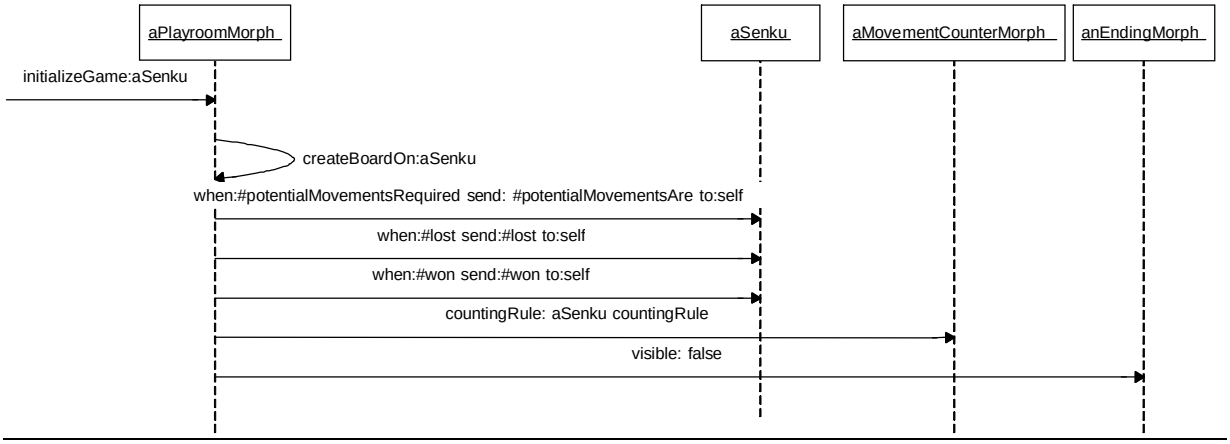




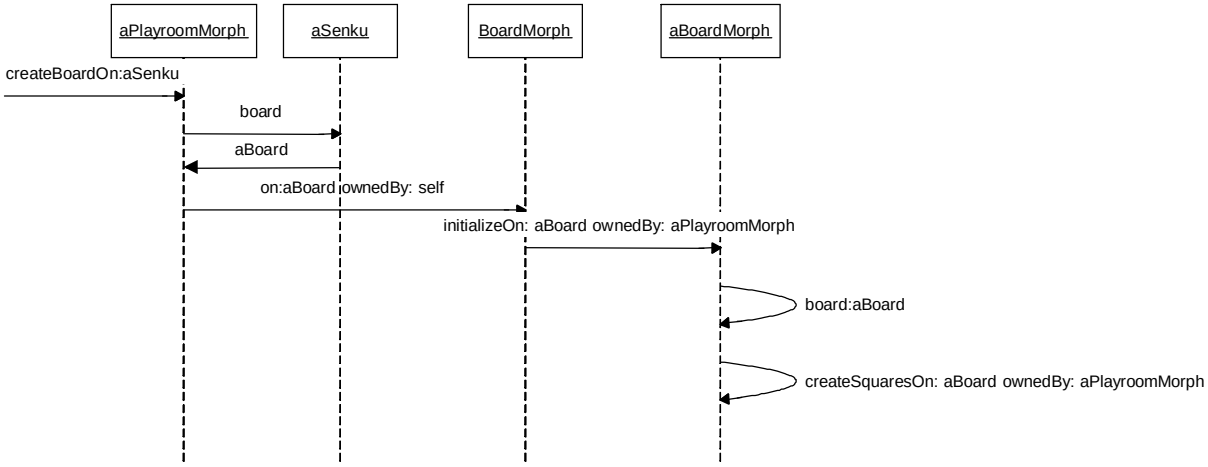
Secuencia: Se selecciona un nuevo Senku



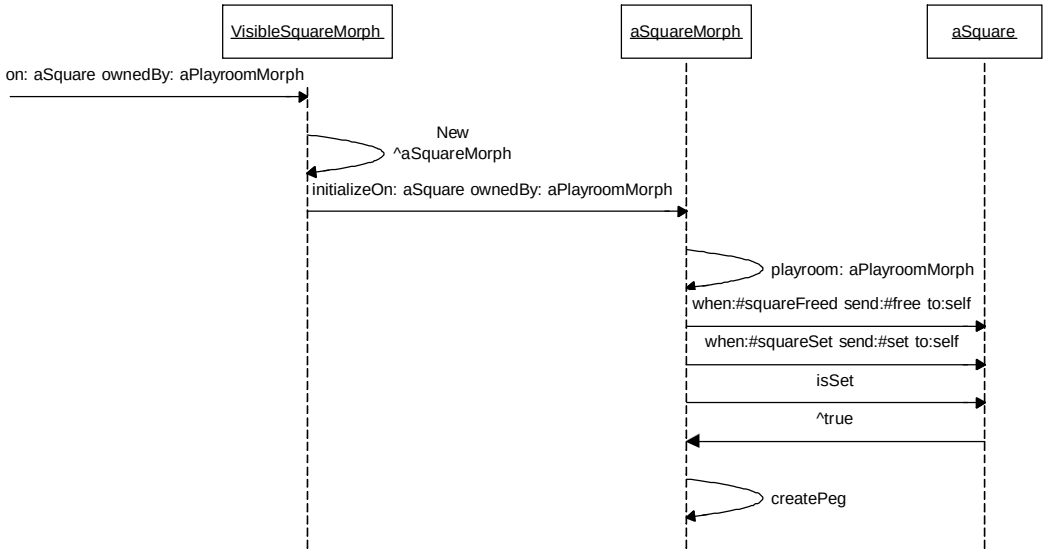
Secuencia: Inicialización del playroom para jugar al Senku



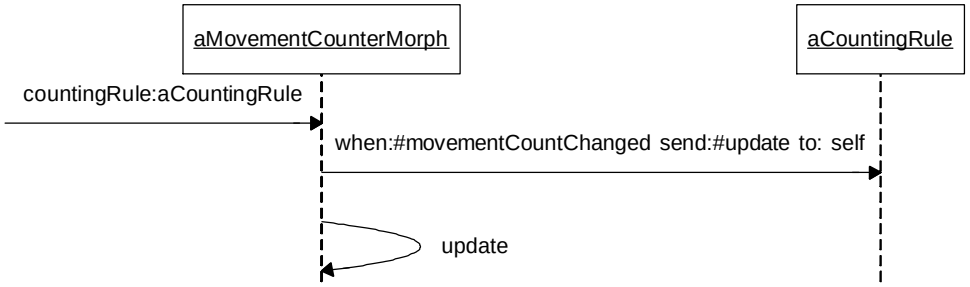
Secuencia: Creación del boardMorph para el Senku



Secuencia: Creación de un squareMorph a partir de un square lleno



Secuencia: Inicialización del movementCounterMorph para el Senku



Diferencias con .NET

El Senku para la plataforma .NET (“dot Senku”) se programó en C#, intentándose seguir con el diseño original planteado para Squeak. Más allá de la traducción efectuada entre las construcciones de ambos lenguajes, se pudo seguir con relativa fidelidad el modelo de la primera versión. Las principales modificaciones que hubo que realizar fueron:

- El mecanismo que provee .NET para el manejo de eventos se basa en la definición de *delegates,* que son asignados luego a *events* de un objeto, a sabiendas que cuando éste ocurra se ejecutarán las funciones delegadas recién mencionadas. La suscripción y desuscripción se efectúa mediante los operadores “+” y “-“, respectivamente. Esto vino a reemplazar los anteriores mensajes triggerEvent: y when:send:to:
- Smalltalk provee una serie de iteradores internos a través de los mensajes do:, select: e inject:into:, entre otros, que resultan sumamente prácticos para el manejo de colecciones. En la versión utilizada de C# no hay instrucciones equivalentes; hubo que convertirlas a iteradores externos utilizando las instrucciones for, foreach o while, dependiendo de cuál resultara más práctica para cada caso.
- Los elementos visuales de Morph no tienen su contraparte en .NET; son similares a los controles de usuario, pero tienen diferencias en cuanto a prestaciones y modo de uso. Por citar sólo un par de ejemplos, dibujar un formulario resulta decididamente más sencillo en .NET; pero luego, cuando se quiere centrar una línea, deben realizarse manualmente los cálculos apropiados, cuando en Morph basta con especificar el parámetro adecuado en el mensaje de creación. Pese a esta y otras diferencias, ninguna resultó sustancial, y pudo seguirse un esquema similar.

Anexo IV: Diseño de detalle del framework

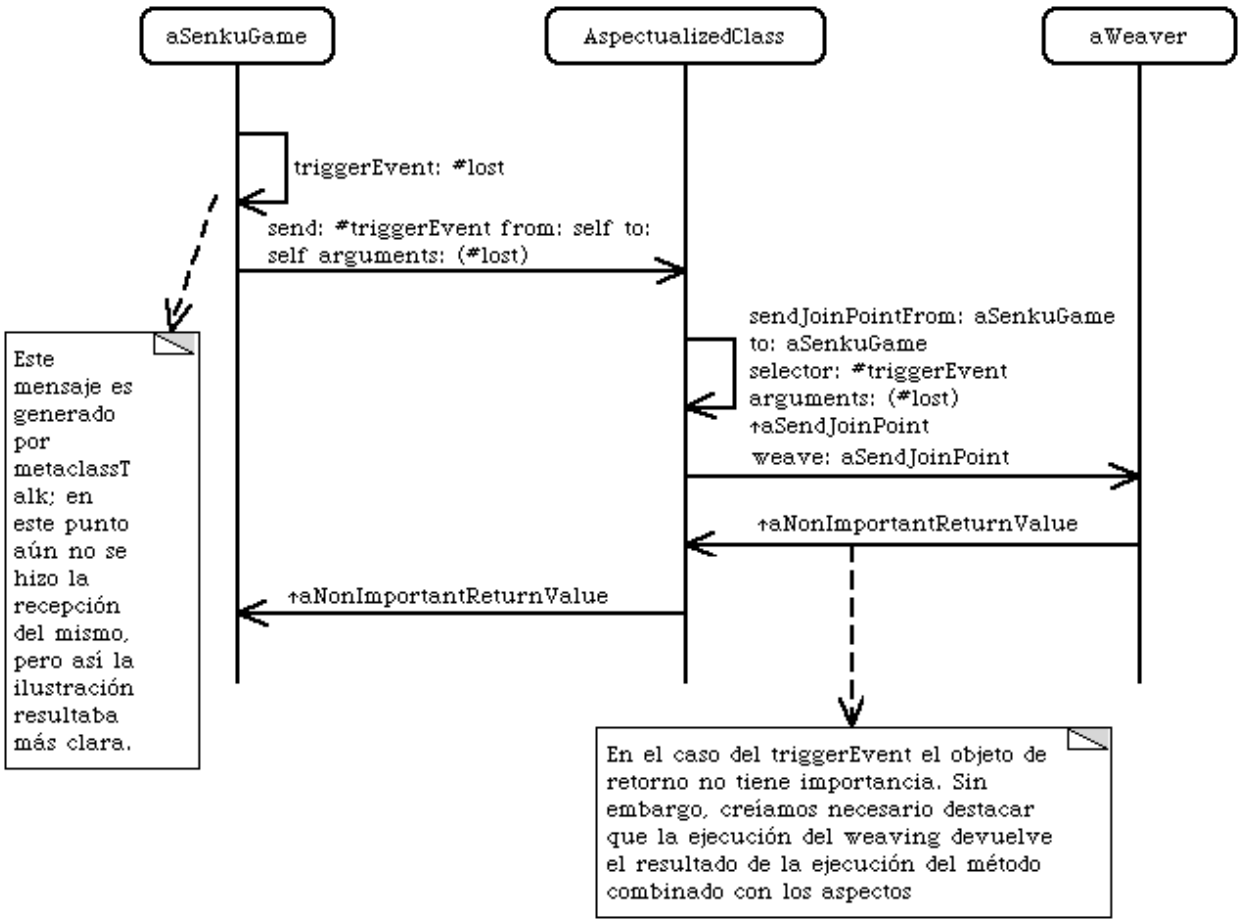
Introducción

En el presente capítulo se complementa la explicación dada de la versión Smalltalk de SetPoint, describiendo el diseño de detalle de la implementación realizada. Se recomienda su lectura conjunta para una mejor comprensión. Vale aclarar que el diseño se intentó mantener lo más fielmente posible en la implementación .NET, aunque la experiencia adquirida sobre el dominio y las diferencias entre las plataformas conllevaron una cantidad de cambios, descritos en la sección correspondiente.

Weaving

Reificación de los joinpoints

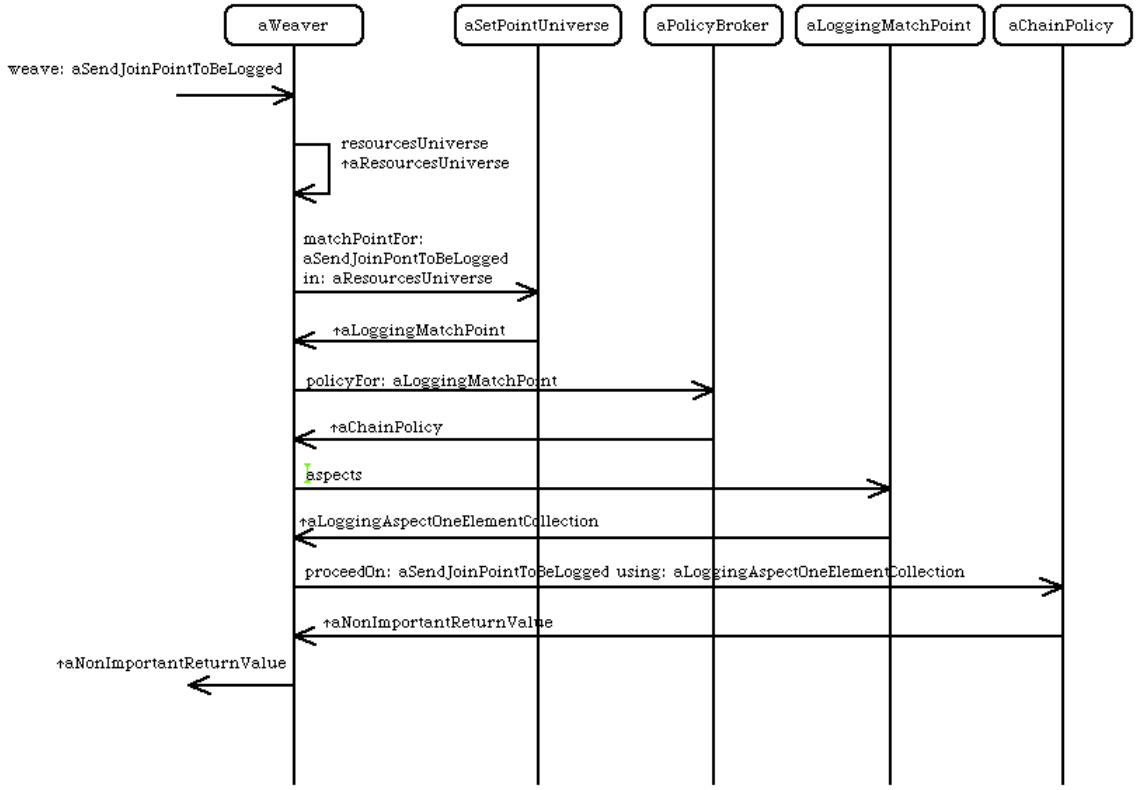
Ejemplificando esta situación sobre un caso concreto, ¿qué ocurre, por ejemplo, cuando el objeto aSenkuGame, perteneciente al conjunto que llamamos *objetos del modelo* del juego de Senku, notifica a sus observadores que se perdió la partida? El siguiente diagrama de secuencia describe el escenario planteado:



El algoritmo del weaver

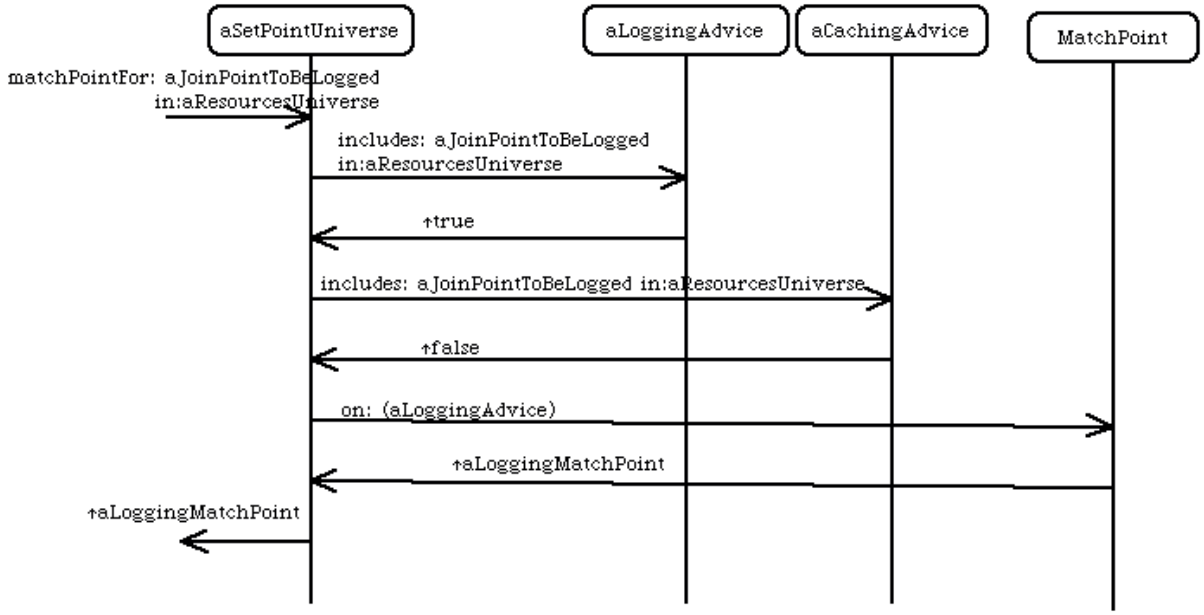
Sigamos con el ejemplo. AspectualizedClass había instanciado al joinPoint correspondiente al envío del mensaje #triggerEvent, que según los requerimientos que tenemos debe ser logueado. Llamamos al objeto recién creado aJoinPointToBeLogged. Estamos pecando de futurólogos... el weaver es quien va a aplicar los aspectos que correspondan al joinpoint, pero, hasta ahora, no está definido cuáles van a ser. Mediante el mensaje “MatchPointFor:”, el universo de setpoints devuelve una instancia de MatchPoint que representa los advices que incluyen al joinpoint que está siendo ejecutado. En el ejemplo que estamos desarrollando, incluye exclusivamente al de logueo. El policy broker define luego que la política de ejecución que corresponde es la denominada c*hainPolicy.*

Los universos de recursos, Setpoints y el policy broker son, actualmente, conocidos globalmente. Una posible mejora en la implementación es hacer que formen parte del contexto, de manera que cada workspace tenga los suyos propios, a la vez que exista un conjunto de elementos compartidos.



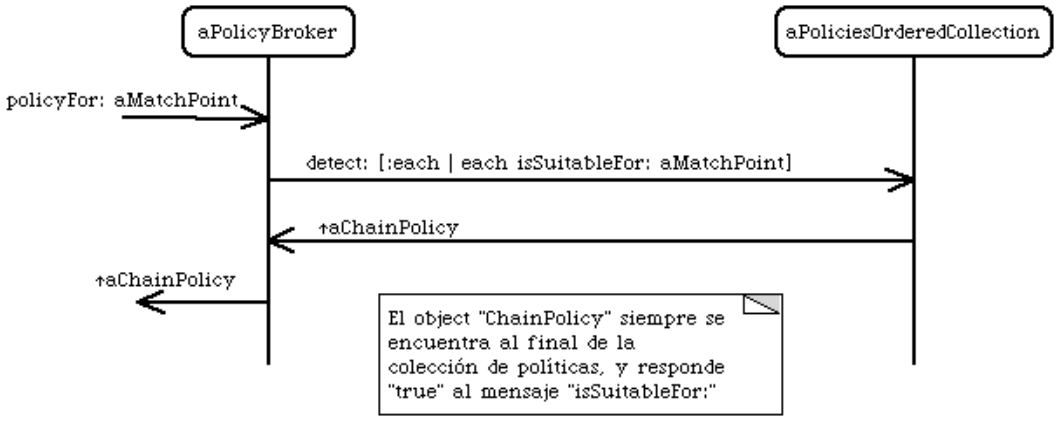
Detalle de la obtención del matchpoint

Dijimos que el matchpoint va a incluir al advice de logueo. Eso lo decide consultando a cada uno de los advices si incluye o no al joinpoint en cuestión. En el diagrama siguiente se describe la secuencia correspondiente.



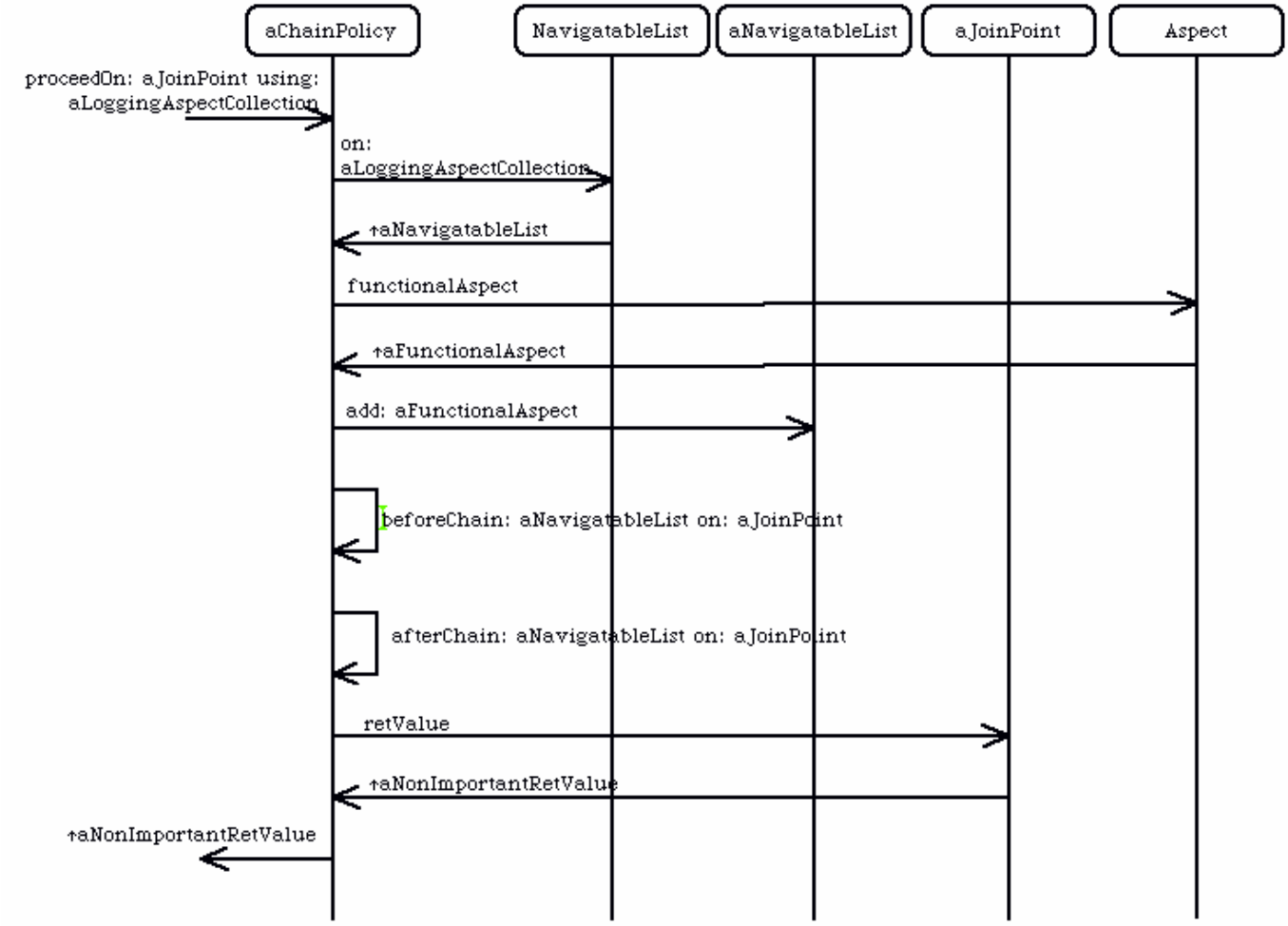
Obtención de la política apropiada para el matchpoint de siempre

El broker de políticas de este prototipo responde siempre con una instancia de *ChainPolicy*, que dice que se deben aplicar los aspectos secuencialmente antes y después del joinpoint. La implementación que realizamos no es sin embargo tan sencilla como responder siempre con un objeto de estos: la idea es que se pueda, si así se lo desea, definir distintas políticas de aplicación de aspectos. Para esto se cuenta con una colección ordenada, organizadas según el nivel de especificidad de cada política. Se devuelve entonces el primer elemento de esta colección que aplique a la combinación de aspectos y el joinpoint dados (resumida por ahora en el matchpoint).



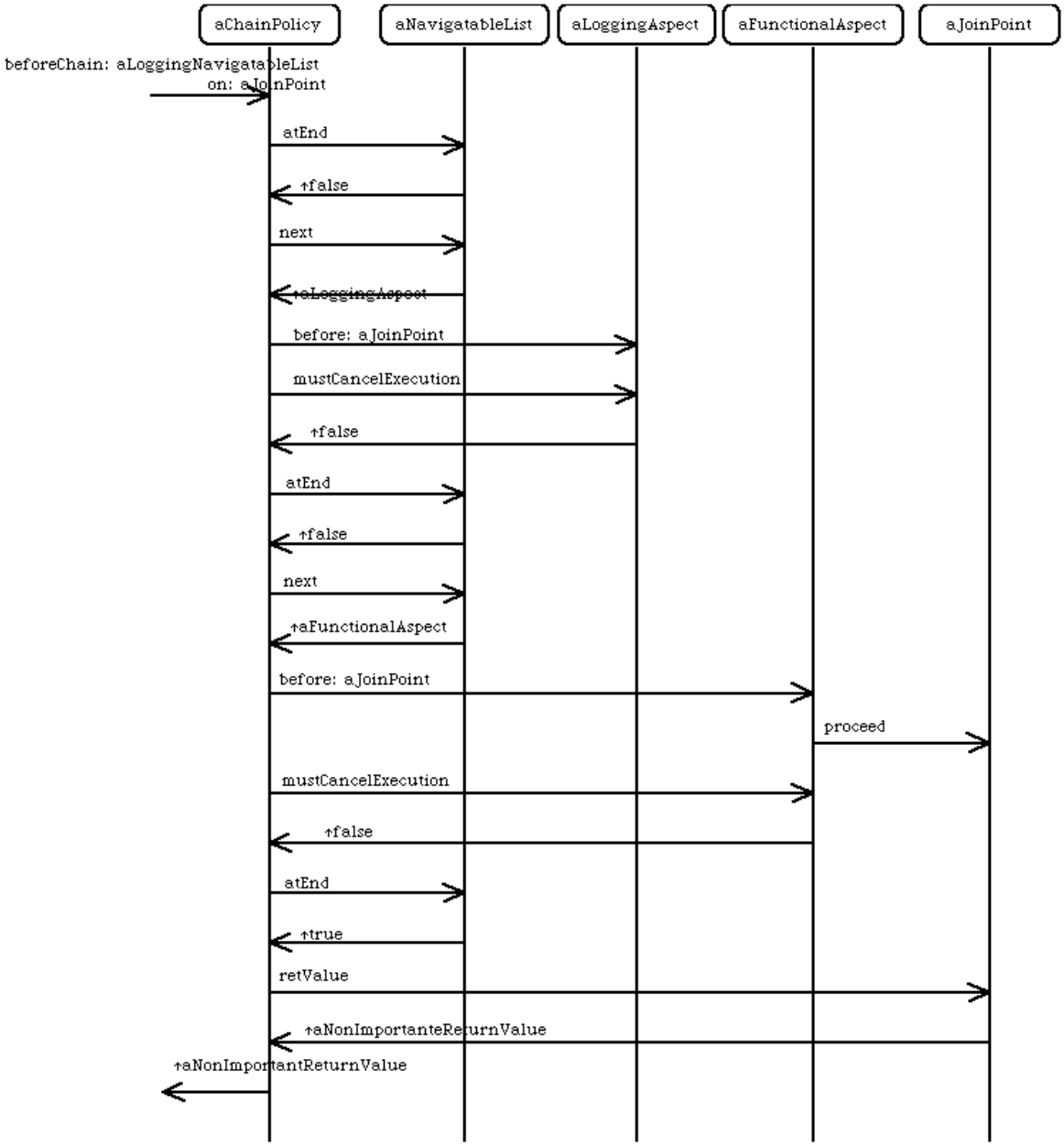
Ejecución del joinpoint a loguear, a través de la chain policy

Hablamos varias veces sobre la política en cadena y de ejecutar secuencialmente los aspectos. En la siguiente secuencia se muestra de una manera un poco más formal que el texto escrito la idea a la que nos referíamos.



Ejecución de aspectos previa a la ejecución del joinpoint

En este diagrama se detalla lo explicado en el anterior para la cadena de *before* del envío del mensaje #triggerEvent (objeto “aJoinPoint”) que queremos loguear.



Estructura de los objetos usados en el ejemplo para realizar el weaving

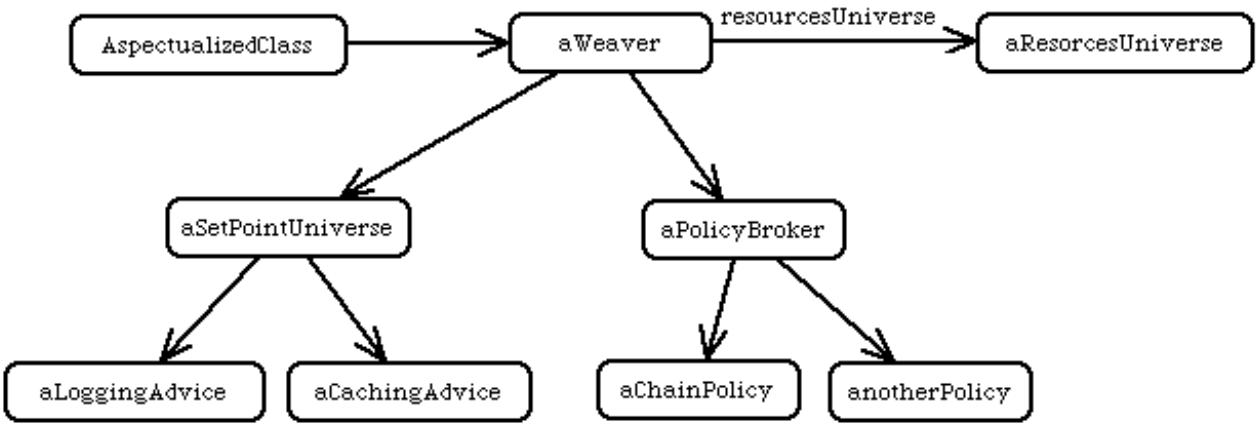
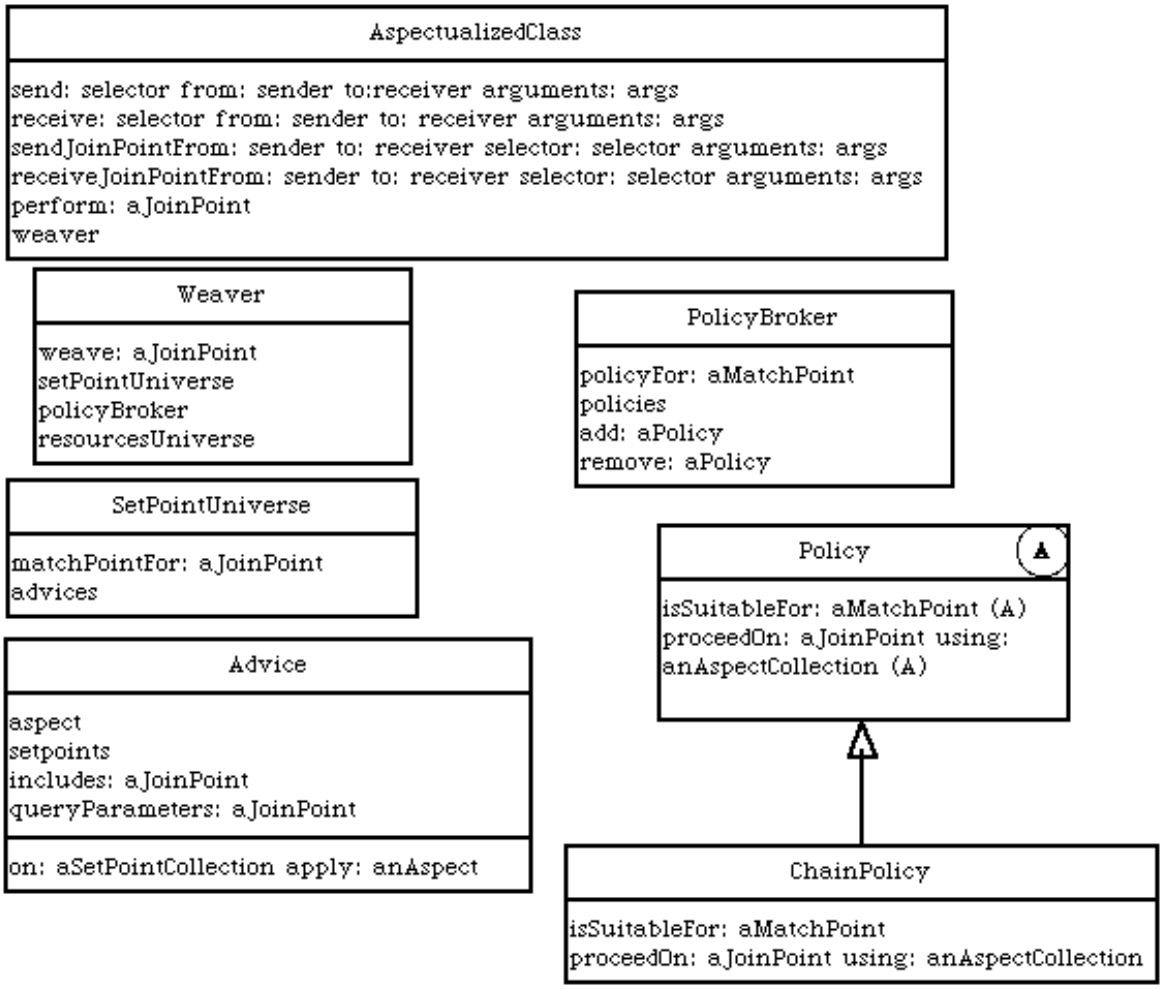
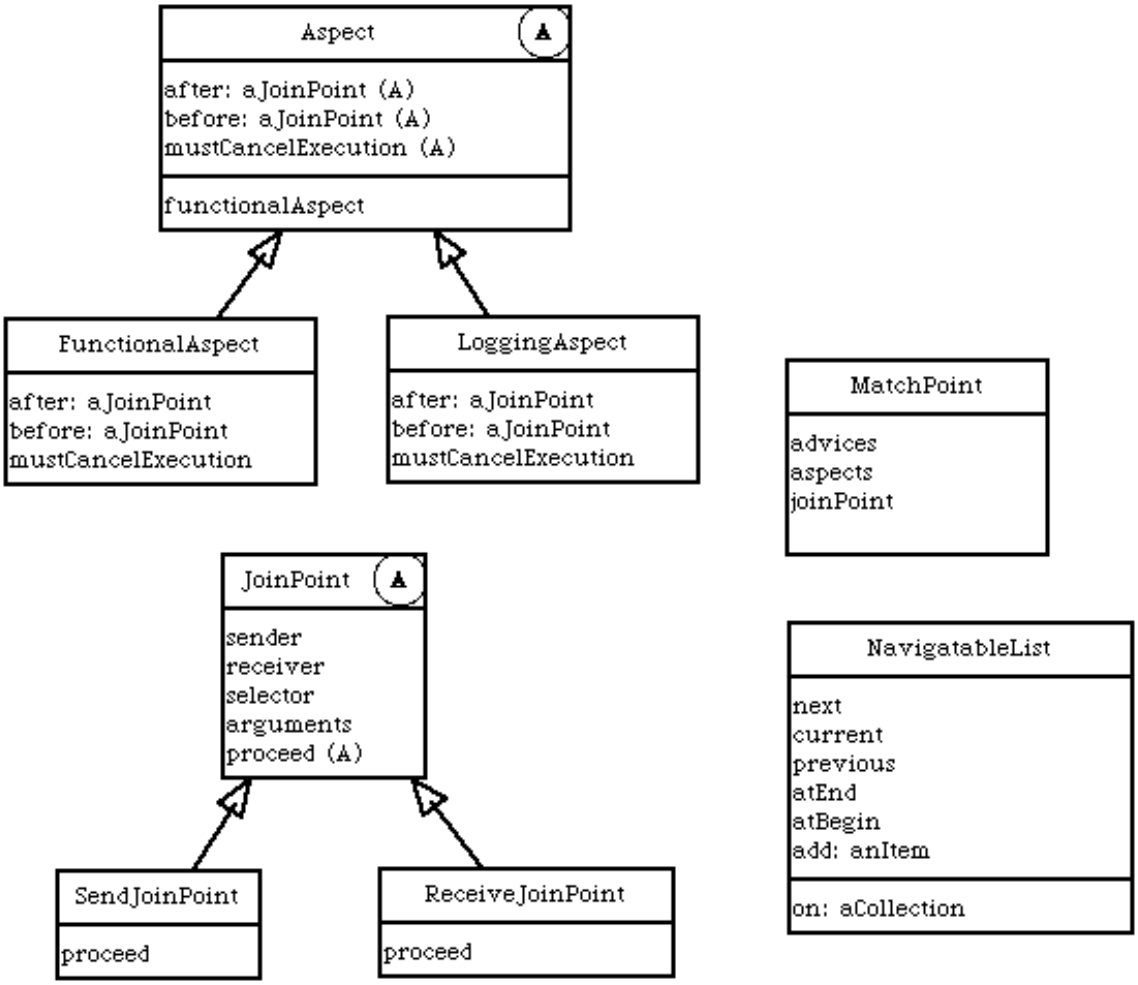


Diagrama de Clases

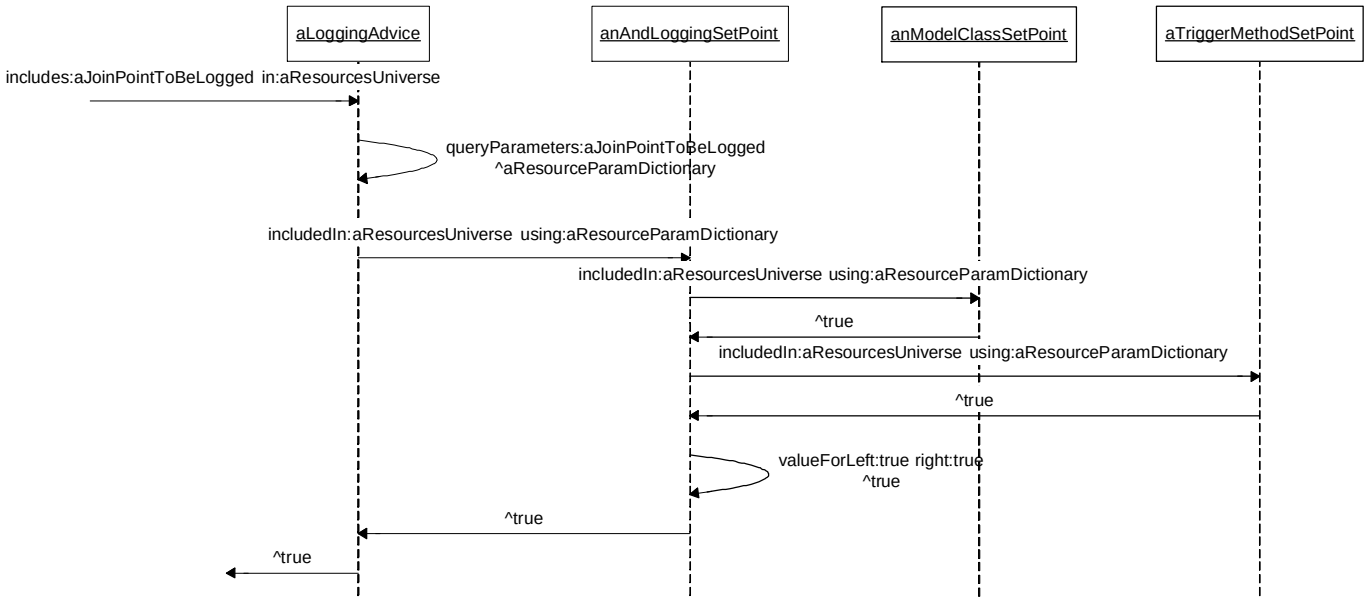




Modelo semántico

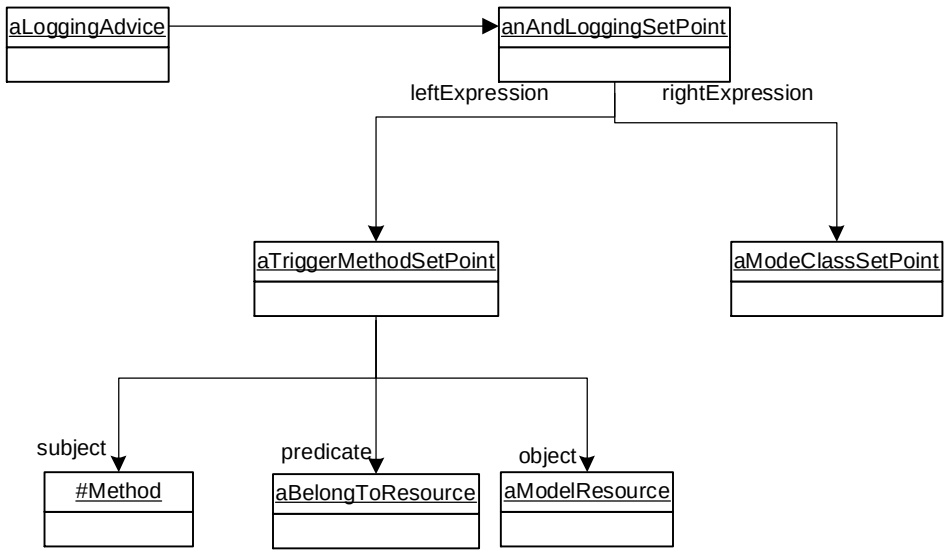
Consulta de aplicabilidad del advice de logueo sobre el joinpoint a loguear

Recordemos el ejemplo desarrollado: estamos consultando los aspectos a aplicar sobre el joinpoint definido en el envío del mensaje #triggerEvent por parte de una clase del modelo. El objeto setpoint que responde a la consulta planteada es el llamado anAndLoggingSetPoint, que evalúa si el método llamado corresponde al envío de eventos (aTriggerMethodSetPoint), y si el objeto que lo envía pertenece al modelo del Senku (anModelClassSetPoint)



Objetos que interactúan para resolver la aplicabilidad del advice anterior

Veamos más en detalle las relaciones de conocimiento estructurales entre el advice y los setpoints. El advice conoce un setpoint, que resuelve si el joinpoint en cuestión cumple con el enunciado “un objeto del modelo que esté enviando un mensaje que asociemos al componente de eventos”. Para esto, realiza la conjunción de dos nuevos setpoints: uno decide si el método corresponde al manejo de eventos y el otro si el objeto pertenece al modelo. Mediante este tipo de composiciones se van formando los predicados de la lógica proposicional. Se evalúa la veracidad de los statements individuales en el universo de recursos, y luego se efectúa la operación correspondiente (conjunción en este caso).



Detalle de la ejecución del triggerMethodSetPoint

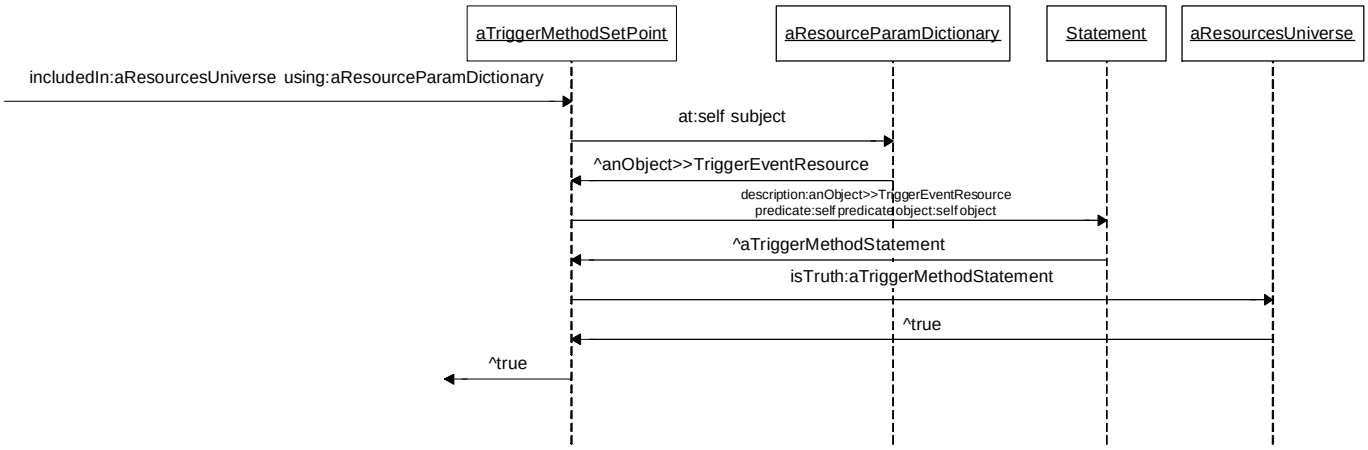
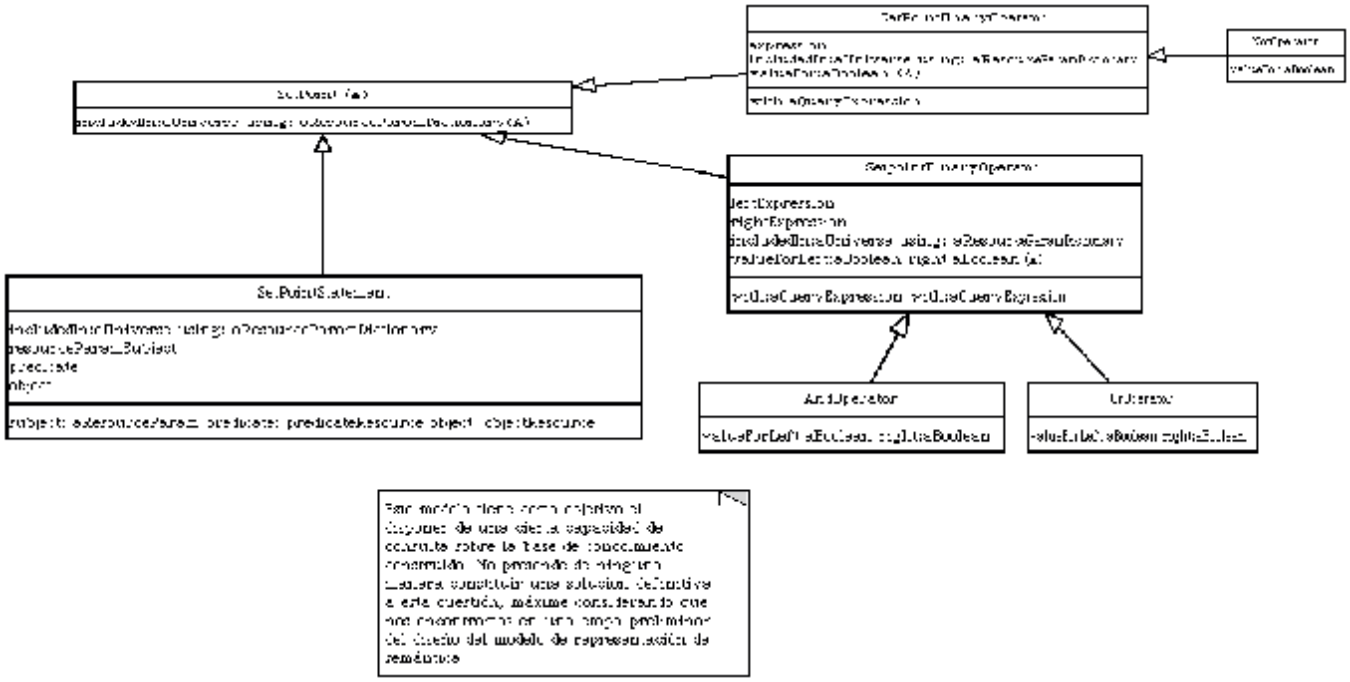
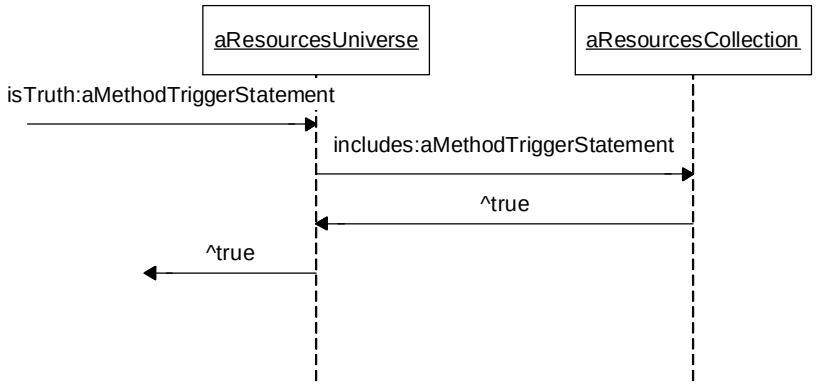


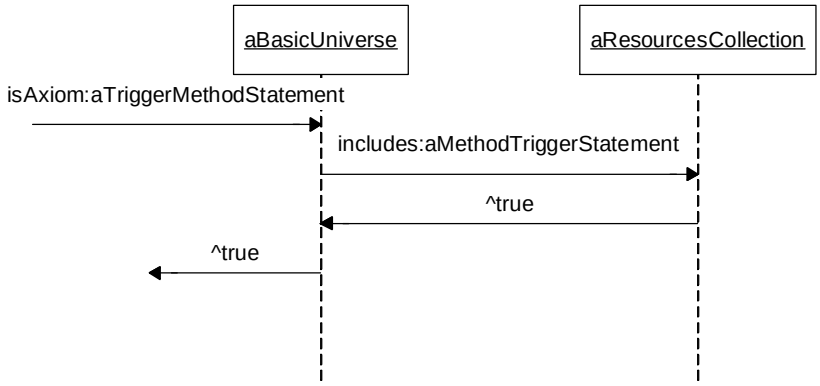
Diagrama de clases de los setpoints



Chequeo de valor de verdad de un statement en el universo de recursos



Verificación de un axioma en un universo simple



Agregado de un axioma en un universo simple

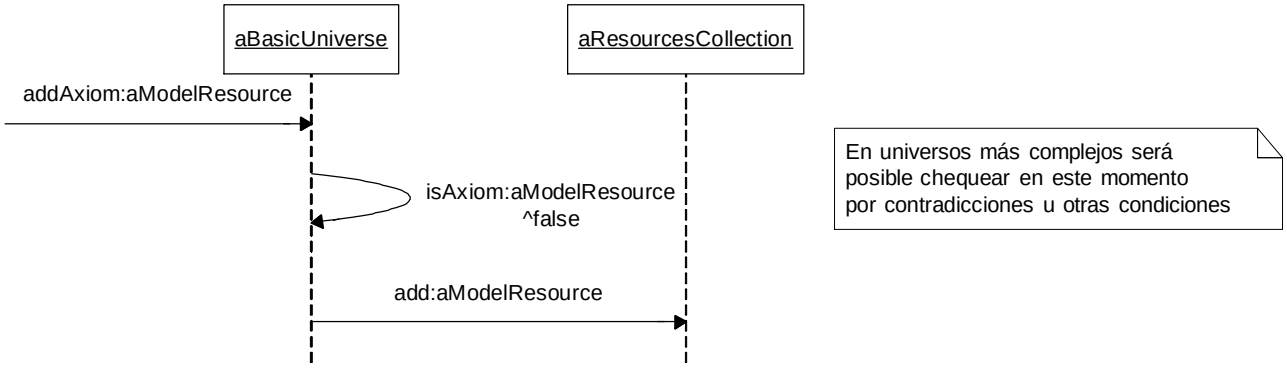


Diagrama de objetos del universo de recursos usado en los ejemplos

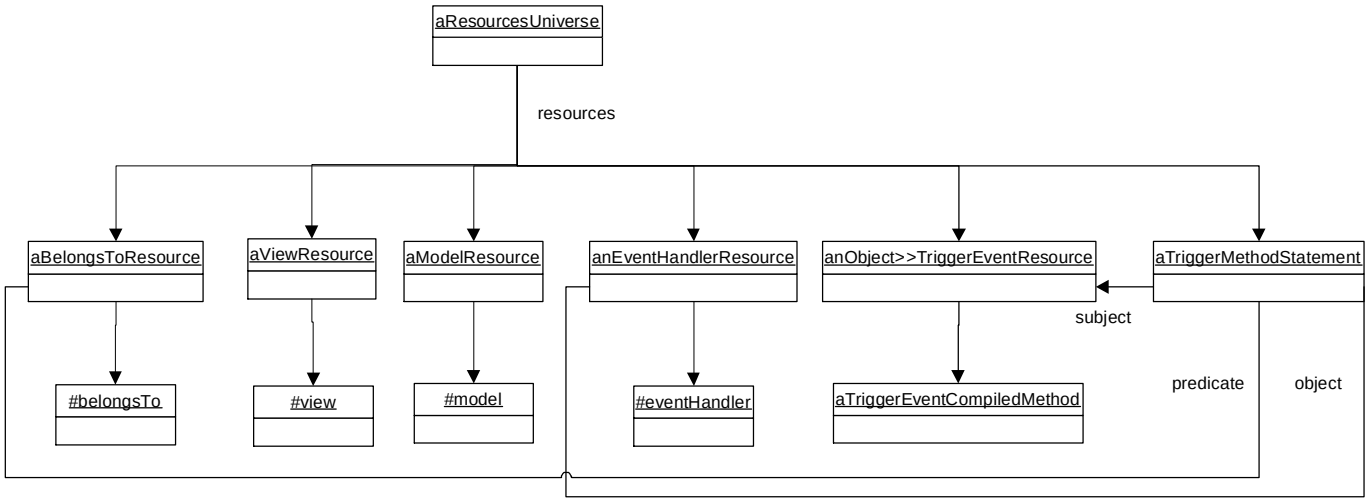
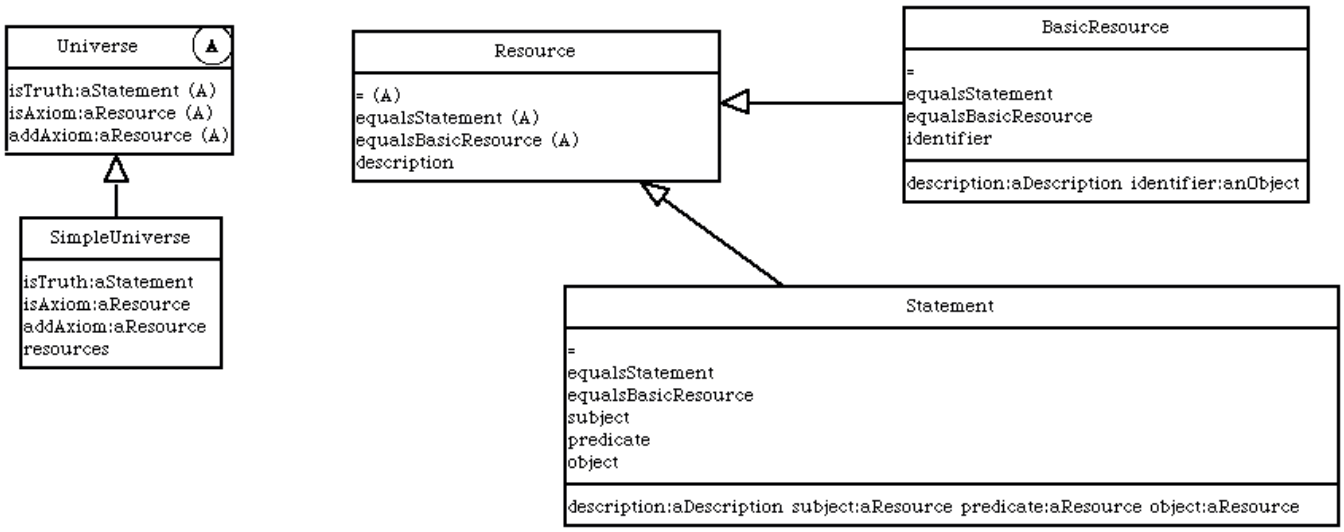
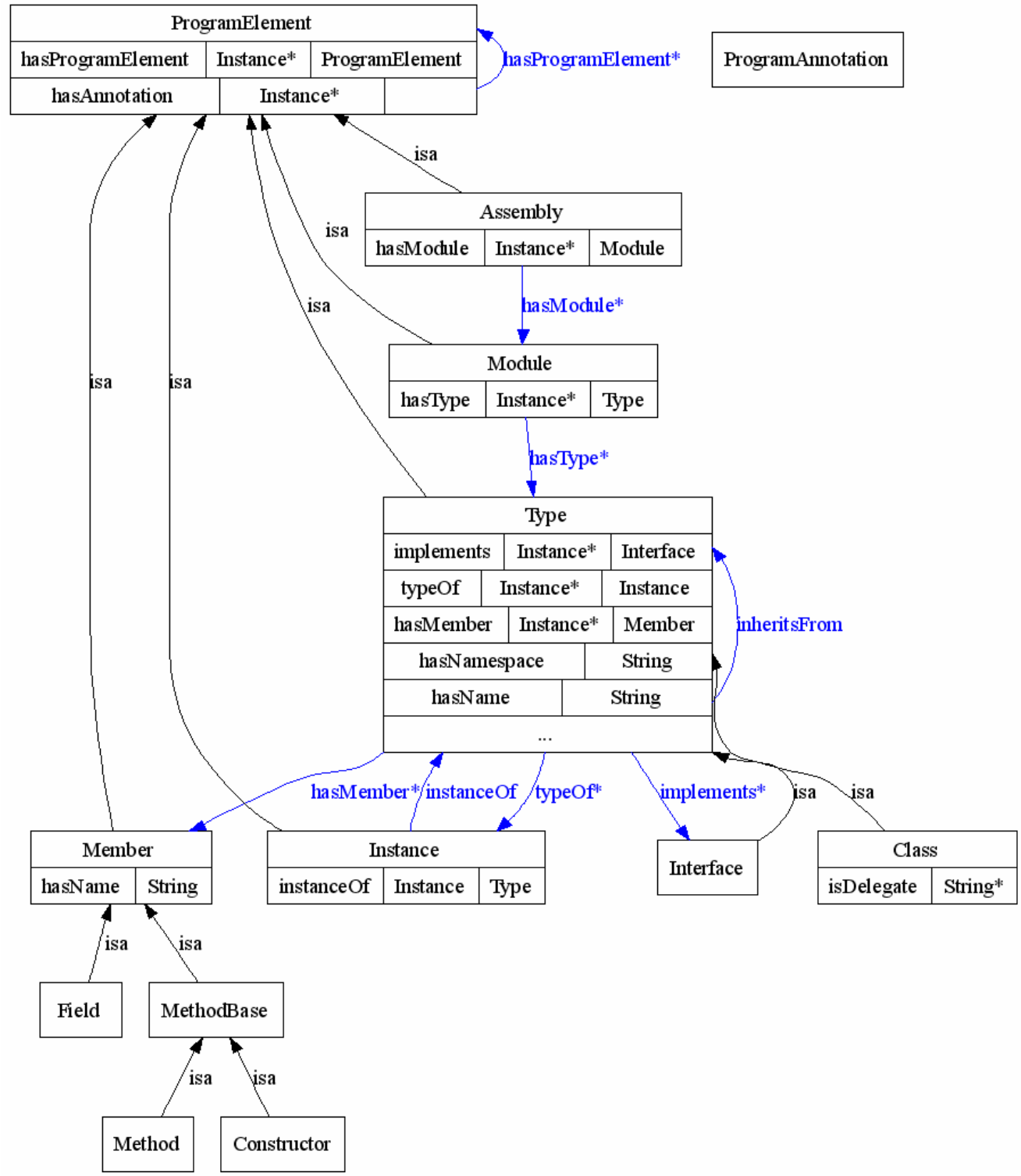


Diagrama de clases



Anexo V: Ontología del Common Type System

Representación gráfica[51]



[51] Realizada con el plugin Ontoviz, de Protegé [PROTEGE]

Representación XML

```
<?xml version="1.0"?>
<rdf:RDFxmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns="semantics://programElements/objectOriented/CTS#"
      xml:base="semantics://programElements/objectOriented/CTS">
<owl:Ontology rdf:about="" />
<owl:Class rdf:ID="ProgramAnnotation" />
<owl:Class rdf:ID="Assembly">
      <rdfs:subClassOf>
            <owl:Class rdf:ID="ProgramElement" />
      </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Method">
      <rdfs:subClassOf>
            <owl:Class rdf:ID="MethodBase" />
      </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Instance">
      <rdfs:subClassOf rdf:resource="#ProgramElement" />
</owl:Class>
<owl:Class rdf:ID="Member">
      <rdfs:subClassOf rdf:resource="#ProgramElement" />
</owl:Class>
<owl:Class rdf:about="#MethodBase">
      <rdfs:subClassOf rdf:resource="#Member" />
</owl:Class>
<owl:Class rdf:ID="Module">
      <rdfs:subClassOf rdf:resource="#ProgramElement" />
</owl:Class>
<owl:Class rdf:ID="Type">
      <rdfs:subClassOf rdf:resource="#ProgramElement" />
</owl:Class>
<owl:Class rdf:ID="Constructor">
      <rdfs:subClassOf rdf:resource="#MethodBase" />
</owl:Class>
<owl:Class rdf:ID="Field">
      <rdfs:subClassOf rdf:resource="#Member" />
</owl:Class>
<owl:Class rdf:ID="Interface">
      <rdfs:subClassOf rdf:resource="#Type" />
</owl:Class>
<owl:Class rdf:ID="Class">
      <rdfs:subClassOf rdf:resource="#Type" />
</owl:Class>
<owl:ObjectProperty rdf:ID="hasModule">
      <rdfs:subPropertyOf>
            <owl:TransitiveProperty rdf:ID="hasProgramElement" />
      </rdfs:subPropertyOf>
      <rdfs:domain rdf:resource="#Assembly" />
      <rdfs:range rdf:resource="#Module" />
```

```
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="inheritsFrom">
        <rdfs:range rdf:resource="#Type" />
        <rdfs:domain rdf:resource="#Type" />
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="typeOf">
        <owl:inverseOf>
                <owl:FunctionalProperty rdf:ID="instanceOf" />
        </owl:inverseOf>
        <rdfs:domain rdf:resource="#Type" />
        <rdfs:range rdf:resource="#Instance" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasAnnotation">
        <rdfs:domain rdf:resource="#ProgramElement" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasType">
        <rdfs:range rdf:resource="#Type" />
        <rdfs:subPropertyOf>
                <owl:TransitiveProperty rdf:about="#hasProgramElement" />
        </rdfs:subPropertyOf>
        <rdfs:domain rdf:resource="#Module" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="implements">
        <rdfs:range rdf:resource="#Interface" />
        <rdfs:domain rdf:resource="#Type" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasMember">
        <rdfs:range rdf:resource="#Member" />
        <rdfs:subPropertyOf>
                <owl:TransitiveProperty rdf:about="#hasProgramElement" />
        </rdfs:subPropertyOf>
        <rdfs:domain rdf:resource="#Type" />
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="isDelegate">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
        <rdfs:domain rdf:resource="#Class" />
</owl:DatatypeProperty>
<owl:TransitiveProperty rdf:about="#hasProgramElement">
        <rdfs:domain rdf:resource="#ProgramElement" />
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty" />
        <rdfs:range rdf:resource="#ProgramElement" />
</owl:TransitiveProperty>
<owl:FunctionalProperty rdf:about="#instanceOf">
        <rdfs:range rdf:resource="#Type" />
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty" />
        <rdfs:domain rdf:resource="#Instance" />
        <owl:inverseOf rdf:resource="#typeOf" />
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasName">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty" />
        <rdfs:domain>
```

```
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="#Member" />
                    <owl:Class rdf:about="#Type" />
                </owl:unionOf>
            </owl:Class>
        </rdfs:domain>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasNamespace">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty" />
        <rdfs:domain rdf:resource="#Type" />
</owl:FunctionalProperty>
</rdf:RDF>
```

Anexo 6: LENDL[52]

El lenguaje LENDL permite al programador de aspectos definir fácilmente el esqueleto del código fuente C# que permite crear el assembly de configuración para el framework de aspectos SetPoint en su versión .NET. Una explicación informal de su uso aparece en el capítulo *La base está: Setpoint v1".* La herramienta CodeWorker [CODEWORKER] fue utilizada para su desarrollo. En el primer apartado se detalla la sintaxis, escrita en EBNF [TUCKER]. En el segundo se transcribe el input utilizado para su generación mediante la herramienta anterior.

Sintaxis[53]

LENDL **à** { *alias* }* { *pointcut* | *advice* | *aspect* | *rule*} *

alias **à** "declare " *identifier* " alias " *realUri* ";"

pointcut **à** "pointcut {" *pointcut_statement* "}"

pointcut_statement **à** *variableKeyword uri uri* ";"

variableKeyword **à** { "sender" | "receiver" | "message" }

advice **à** "advice " *identifier* ":" *identifier* "{" { *trigger* } * "}"

trigger **à** "trigger " *identifier when* "{" *identifier* {", " *identifier* }* "} ;"

when **à** { "before" | "after" }

aspect **à** "aspect " *identifier* { builtby *identifier* }$_{opt}$ "{" { *event* } * "}"

event **à** "event " *identifier* ";"

uri **à** { "[" *identifier* { "://" *uri_atom{* "/" *uri_atom }* }$_{opt}$ "#" *uri_atom*"]" | "is" }

uri_atom **à** { 'a'..'z' | 'A'..'Z' | "." | "_" }+

identifier **à** { 'a'..'z' | 'A'..'Z' | '_'} { 'a'..'z' | 'A'..'Z' | '_' | '0'..'9'}*

alpha **à** { 'a'..'z' | 'A'..'Z' }+

Input para CodeWorker

```
LENDL    ::= #continue #ignore(C++) LENDL_body #empty;
LENDL_body ::= =>{
      insert this.aliases["rdf"] = "http://www.w3.org/1999/02/22-rdf-syntax-ns";
      insert this.aliases["rdfs"] = "http://www.w3.org/2000/01/rdf-schema";
      insert this.aliases["owl"] = "http://www.w3.org/2002/07/owl";
      insert this.aliases["cts"] = "semantics://programElements/objectOriented/CTS";
      }
[ALIAS]*[POINTCUT|ADVICE|ASPECT|RULE]*;
//////////////////////////// ALIAS ////////////////////////////////////////
ALIAS ::= "declare " #readIdentifier:sAlias #continue "alias " REAL_URI:sAliasUri #continue
";"
```

[52] No será Coria, pero hizo sus méritos

[53] No se incluyen tokens que tengan que ver con el manejo de los espacios en el texto, porque consideramos que no aportan a la clarificación del lenguaje. Dicho nivel de detalle puede encontrarse en el input para CodeWorker.

```
      =>{ insert this.aliases[sAlias] = sAliasUri; };

//////////////////////////// POINTCUT ////////////////////////////////////////
POINTCUT ::= "pointcut" #continue #readIdentifier:sPointCutName =>{
             if this.pointcuts.findElement(sPointCutName) error("pointcut '" + sPointCutName
      + "' is already defined");
             insert this.pointcuts[sPointCutName].name = sPointCutName;
             insert this.pointcuts[sPointCutName].SeRQLFROM = "";
      }
      '{'
              [pointcutStatement(this.pointcuts[sPointCutName])]*
             => set this.pointcuts[sPointCutName].SeRQLFROM =
             leftString(this.pointcuts[sPointCutName].SeRQLFROM,
             sub(lengthString(this.pointcuts[sPointCutName].SeRQLFROM),1));
      '}';

function formatUri(originalToken : value) {
    local sResult;
    switch(originalToken){
       case "is" :
             set sResult =
             "<!semantics://programElements/objectOriented/CTS#hasAnnotation>";
             break;
       default :
             local i = lengthString(originalToken);
             decrement(i); decrement(i);
             local uri;
             set uri =  midString(originalToken, 1, i);
             if  findString(uri, ":") = -1 {
                   local alias = midString(uri, 0, findString(uri, "#"));
                   if !this.aliases.findElement(alias)
                         error("alias '" + alias+ "' is not defined");
                   set uri = this.aliases[alias] + rightString(uri, sub(lengthString(uri),
                   findString(uri, "#")));
             }
             set sResult = "<!" + uri + ">";
    }
    return sResult;
}

function getPathElement(aToken : value, aTokenPosition : value) {
    local sResult;
    if mod(aTokenPosition,2)=0
       set sResult = aToken;
    else
       set sResult = '{' + aToken + '}';
    return sResult;
}


pointcutStatement(aPointcut : node) ::= variableKeyword(aPointcut, 1):aSubject #continue
messageKeyword(aPointcut, 2):aPredicate #continue pointcutRDFResource(aPointcut, 3):anObject
';'
      => trim(aPointcut.SeRQLFROM);
      => set aPointcut.SeRQLFROM += ',';;

variableKeyword(aPointcut : node, tokenPosition : value)  ::=
["sender"|"receiver"|"message"]:aToken
      => set aPointcut.SeRQLFROM += getPathElement(aToken, tokenPosition) + ' ';;

messageKeyword(aPointcut : node, tokenPosition : value)  ::= URI:anUri
      => set aPointcut.SeRQLFROM += getPathElement(formatUri(anUri), tokenPosition) + ' ';;
```

```
pointcutRDFResource(aPointcut : node, tokenPosition : value) ::= URI:anUri
      => set aPointcut.SeRQLFROM += getPathElement(formatUri(anUri), tokenPosition) + ' ';;


//////////////////////////// ADVICE ////////////////////////////////////////
ADVICE ::= "advice" #readIdentifier:sAdviceName ":" #readIdentifier:sAspectName
      =>{
            if  this.advices.findElement(sAdviceName)
                  error("advice '" + sAdviceName + "' is already defined");
            insert this.advices[sAdviceName].name = sAdviceName;
            insert  this.advices[sAdviceName].triggers;
            insert  this.advices[sAdviceName].aspectName = sAspectName;
      }
      '{' #continue
             [trigger(this.advices[sAdviceName])]*
      '}';

trigger(anAdvice : node) ::=  "trigger" => pushItem anAdvice.triggers;=> localref aTrigger =
anAdvice.triggers#back;=>insert  aTrigger.pointCutNames;
      #continue #readIdentifier:aTrigger.eventName  WHEN:aTrigger.when
      '{' #readIdentifier:sPointCutName => pushItem aTrigger.pointCutNames = sPointCutName;
      [',' #readIdentifier:sPointCutName => pushItem aTrigger.pointCutNames =
      sPointCutName;]* '}' ';';

WHEN ::= ["after"|"before"];

//////////////////////////// ASPECT ////////////////////////////////////////
ASPECT ::= "aspect" #readIdentifier:sAspectName #continue ["builtby"
#readIdentifier:sFactoryName]?
      =>{
            if  this.aspects.findElement(sAspectName)
                  error("aspect '" + sAspectName + "' is already defined");
            insert this.aspects[sAspectName].name = sAspectName;
            insert this.aspects[sAspectName].factoryName = sFactoryName;
            insert  this.aspects[sAspectName].events;
      }
      '{' #continue
             [event(this.aspects[sAspectName])]*
      '}';

event(anAspect : node) ::=  "event" => pushItem anAspect.events;=> localref anEvent =
anAspect.events#back;
      #continue #readIdentifier:anEvent.eventName ';';

//////////////////////////// RULES ////////////////////////////////////////
RULE ::= "rule" #readIdentifier:sRuleName
      =>{
            insert this.rules[sRuleName].name = sRuleName;
            insert this.rules[sRuleName].consequence = "";
            insert  this.rules[sRuleName].premises;
      }
      '{' #continue
             [inference(this.rules[sRuleName])]*
      '}';

inference(aRule : node) ::=  "infer" #continue consequence(aRule) #continue
                              "when" #continue premise(aRule, "") #continue
                              ["and" #continue premise(aRule, ", ")]* #continue ';';


consequence(aRule : node) ::= uriOrVariable(aRule.consequence, 1, ""):aSubject #continue
uriOrVariable(aRule.consequence, 2, " "):aPredicate #continue
uriOrVariable(aRule.consequence, 3, " ");
```

```
premise(aRule : node, beginChar:value) ::= uriOrVariable(aRule.premises, 1, beginChar)
#continue uriOrVariable(aRule.premises, 2, " ") #continue uriOrVariable(aRule.premises, 3, "
");

uriOrVariable(resourceToFill:node, position:value, beginChar:value) ::=
[ALPHA|URI:anUri]:aResource
	=> {
		if (anUri != "")
			set aResource = formatUri(anUri);
		resourceToFill += beginChar + getPathElement(aResource, position);
	};

//////////////////////////// MISC ////////////////////////////////////
URI ::= #!ignore ['[' #continue ID[":/"URI_ATOM['/'URI_ATOM]*]?'#'URI_ATOM:anUri #continue
']'|"is"];
REAL_URI ::= #!ignore ALPHA":/"URI_ATOM['/'URI_ATOM]*;
URI_ATOM ::= #!ignore ['a'..'z'|'A'..'Z'|'.'|'_']+;
ALPHA ::= #!ignore ['a'..'z'|'A'..'Z']+;
ID ::= #!ignore ['a'..'z'|'A'..'Z'|'_']['a'..'z'|'A'..'Z'|'_'|'0'..'9']*;
```

Bibliografía

1. [AM] K. Lieberherr, D. Orleans y J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods, Communications of the ACM, vol. 44(10), 2001
2. [ALPERT] Sherman Alpert, Kyle Brown y Bobby Woolf. The Design Patterns Smalltalk Companion, Adison-Wesley, 1998
3. [ADDIN] http://www.geocities.com/m_mesalem/aop.html
4. [AOP#] Mario Schüpany, Christa Schwanninger y Egon Wuchner, Aspect-Oriented Programming for .NET. En First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, AOSD 2002.
5. [AOP.NET] http://sourceforge.net/projects/aopnet/
6. [ASOC] Eds. P. Tarr, M. D'Hondt, L. Bergmans, y C.V. Lopes. Requirements on, and Challenge Problems For, Advanced Separation of Concerns. En Proceedings of Workshop on Aspects and Dimensions of Concern, ECOOP 2000
7. [ASPECT.NET] Vladimir Safonov, Aspect.NET - a new approach to aspect oriented programming .NET Developers Journal, Abril de 2003.
8. [ASPECT#] http://aspectsharp.sourceforge.net/
9. [ASPECTC#] Howard Kim, AspectC#: An AOSD implementation for C# (tesis de licenciatura), Department of Computer Science, Trinity College Dublin, Septiembre de 2002
10. [ASPECTDNG] http://aspectdng.sourceforge.net
11. [ASPECTJ] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm y W. Griswold. An Overview of AspectJ, ECOOP 2001
12. [ASPECTJ2] "The aspectJ Team". The AspectJ Programming Guide, Xerox-Corporation, 2002-2003
13. [ASPECTJ3] http://eclipse.org/aspectj/
14. [ASPECTJ4] Interview With Gregor Kiczales, The Server Side, Julio de 2003
15. [AWERKZ] http://aspectwerkz.codehaus.org/
16. [BOLLERT] K. B ollert. On weaving aspects. En Proceedings of the European Conference on Object-Oriented Programming '99 Workshop on Aspect-Oriented Programming, 1999.
17. [CAESAR1] Mira Mezini y Klaus Ostermann, Integrating Independent Components with On-Demand Remodularization, Proceedings Of OOSPLA 2002
18. [CAESAR2] Mira Mezini y Klaus Ostermann, Conquering Aspects with Caesar, Proc. International Conference on Aspect-Oriented Software Development (AOSD '03), Boston, USA
19. [CAMEO] M. Devi Prasad and B.D. Chaudhary. AOP support in C#. En Proceedings of the 2nd AOSD Workshop on Aspects, Components and Patterns on Infrastructure Software (held together with AOSD 2003), Marzo de 2003.
20. [CF] L. Bergmans y M. Aksit. Composing crosscutting concerns using composition filters, Communications of the ACM, vol. 44(10), 2001

21. [CLAW1] http://www.iunknown.com/000092.html
22. [CLAW2] http://www.iunknown.com/000216.html
23. [CLIFR] http://dotnet.di.unipi.it/MultipleContentView.aspx?code=103
24. [CODEWORKER] http://codeworker.free.fr/
25. [COM] D. Shukla, S. Fell y C. Sells. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse, MSDN Magazine, 2002
26. [DEDECKER] Jessie Dedecker, Dynamic Aspect Composition using Logic Metaprogramming (tesis de licenciatura), Vrieje Universiteit Brussel, Bélgica, 2002
27. [DENNY] Michael Denny, Ontology tools survey, revisited, XML.com, O'Reilly, Julio de 2004 (http://www.xml.com/pub/a/2004/07/14/onto.html)
28. [DIJK] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976
29. [DHONDT] Maja D'Hont y Theo D'Hont. Is Domain Knowledge an Aspect?, ECOOP Workshops 1999
30. [DSL] Niels Christaensen, Domain specific language in software development, and the relation to partial evaluation. PhD. Thesis, University of Copenhagen, 2003,
31. [ECMA335] Standard ECMA 335: Common Language Infraestructure (CLI), 2da edición, Diciembre de 2002. Se puede encontrar en http://www.ecma-international.org/publications/standards/Ecma-335.htm
32. [EJB] Jung Pil Choi. Aspect-Oriented Programming with Enterprise JavaBeans, EDOC 2000
33. [FALBO1] Ricardo Falbo, Giancarlo Guizzardi, Katia Duarta y Ana Candida Natali, Developing Software for and with reuse: An Ontological Approach. ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA-02), Foz do Iguacu, Brazil, 2002.
34. [FALBO2] Ricardo Falbo, C. Menezes y A.R.C. Rocha, A systematic approach for building ontologies. En Proceedings of the IBERAMIA '98, 1998
35. [FILMAN] R. Filman y D. Friedman. Aspect-oriented programming is quantification and obliviousness. Proc. Workshop on Advanced Separation of Concerns, OOPSLA 2000
36. [GAMMA] Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Adison-Wesley, 1994
37. [GRUBER] Thomas Gruber, Toward Principles for the Design of Ontologies Used for Knowledge Sharing. International Workshop on Formal Ontology, Marzo de 1993
38. [HOFMEISTER] C. Hofmeister et al., Applied Software Architecture, capítulo 4. Addison Wesley Professional, 1999.
39. [HYPERJ] Harold Ossher y Peri Tarr. Hyper/J: multi-dimensional separation of concerns for Java, ICSE 2000.
40. [HYPERJ1] Harold Ossher y Peri Tarr Multi-dimensional separation of concerns and the Hyperspace approach. Software Architectures and Component Technology (M. Aksit, ed.), 293--323, Kluwer, 2002.
41. [HYPERJ2] Peri Tarr, Harold Ossher, William Harrison y Stanley Sutton Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns, International Conference on Software Engineering Proceedings, 1999, pp. 107-119.

42. [HYPERJ3] Peri Tarr, Harold Ossher. Hyper/J user and installation manual. IBM Research, 2000
43. [IEEE90] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* New York, 1990.
44. [IGUANA] D. Lafferty y V. Cahill. Real World Evaluation of Aspect-Oriented Programming with Iguana, ECOOP 2000
45. [IKVM] http://www.ikvm.net/
46. [ILREADER] http://www.aisto.com/roeder/dotnet/
47. [JAC1] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gerard Florin, JAC: A Flexible Solution for Aspect-Oriented Programming in Java, Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, 2001
48. [JAC2] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Laurent Martelli. JAC: An Aspect-Based distribution dynamic framework, publicado en el site oficial de JAC, http:// jac.objectweb.org/, 2002
49. [JBOSS] http://www.jboss.org/products/aop
50. [JENA] http://jena.sourceforge.net/
51. [KIC97] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Videira Lopes, C. Maeda y A. Mendhekar. Aspect-Oriented Programming, ECOOP 1997
52. [KRUCHTEN] Philippe Kruchten, The 4 + 1 View Model Of Architecture, IEEE Software, Volumen 12, Número 6, Noviembre de 1995
53. [LIU] Chamond Liu. Smalltalk, Objects, and Design, toExcel, iUniverse.com, Inc., 1996.
54. [LOOM1] Wolfgang Schult, Peter Tröger y Andreas Polze, Loom.NET - an Aspect Weaving Tool, en Workshop on Aspect-Oriented Programming, ECOOP 2003.
55. [LOOM2] http://www.rapier-loom.net
56. [MTALK] http://csl.ensm-douai.fr/MetaclassTalk
57. [MTALK-AOP] Noury Bouraqadi y Thomas Ledoux, Aspect-Oriented Programming using Reflection, Technical Report 2002-10-3, Ecole des Mines de Douai, Francia.
58. [MONO] http://www.go-mono.com/
59. [MORPHIC] http://minnow.cc.gatech.edu/squeak/30
60. [NET] Thuan L. Thai y Hoang Lam, .NET Framework Essentials, O'Reilly, Junio de 2001
61. [NET1] Extending Metadata Using Attributes, .NET Framework Developer's Guide, MSDN Library, Microsoft. Se puede encontrar en http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconextendingmetadatausingattributes.asp
62. [NET2] .NET Framework Fundamentals, .NET Framework Developer's center, MSDN Library, Microsoft. Se puede encontrar en http://msdn.microsoft.com/netframework/programming/fundamentals/default.aspx
63. [NET3] .NET Framework Techonology Overview, .NET Framework Developer's center, MSDN Library, Microsoft. Se puede encontrar en http://msdn.microsoft.com/netframework/technologyinfo/overview/
64. [NET4] The Common Language Runtime, .NET Framework Developer's center, MSDN Library, Microsoft. Se puede encontrar en http://msdn.microsoft.com/netframework/programming/clr/default.aspx

65. [NET5] Curso .NET paso a paso: introducción al desarrollo en .NET, Curso Desarrollador 5 estrellas, MSDN Latinamérica
66. [ONTOEDIT] http://www.ontoprise.de/documents/tutorial_ontoedit.pdf
67. [ONTOWEB] OntoWeb Consortium, coordinado por Asunción Gomez Perez, OntoWeb: A survey of ontology tools. OntoWeb, Mayo de 2002
68. [OWL-QL] R. Fikes, P. Hayes y I. Horrocks, OWL-QL: A Language for Deductive Query Answering on the Semantic Web. Knowledge Systems Laboratory, Stanford University, Stanford, CA, 2003
69. [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
70. [PARNAS] D. Parnas. On the criteria to be used in decomposing systems into modules, Communications of the ACM, vol. 15(2), 1972
71. [PERWAPI] http://www.citi.qut.edu.au/research/plas/projects/cp_files/pefile.jsp
72. [PIE] Ira Goldstein y Daniel Bobrow, An Experimental Description-Based Programming Environment: Four Reports, Palo Alto Research Center, Xerox Corporation, 1981
73. [PROTEGEE] http://protegee.stanford.edu/
74. [RAIL] http://rail.dei.uc.pt/
75. [RDQL] http://www.hpl.hp.com/semweb/doc/tutorial/RDQL/index.html
76. [ROTOR] http://www.microsoft.com/downloads/details.aspx?FamilyId=3A1C93FA-7462-47D0-8E56-8DD34C6292F0&displaylang=en
77. [RQL] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, y M. Scholl. RQL: A declarative query language for RDF. The Eleventh International World Wide Web Conference (WWW'02), Mayo de 2002.
78. [SciAm] Tim Berners-Lee, James Hendler y Ora Lassila, The Semantic Web, Scientific American, Mayo de 2001
79. [SENKU] Rubén Altman y Alan Cyment, Lame Game, trabajo práctico presentado a la cátedra de POO de la Facultad de Ciencias Exactas de la Universidad de Buenos Aires, Julio de 2003
80. [SeRQL] http://www.openrdf.org/doc/users/ch05.html#d0e807.
81. [SESAME] J. Broekstra, A. Kampman y F. van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. En Semantics for the WWW. MIT Press, 2001.
82. [SHERLIS] Elissa Newman y William Scherlis, Toward Query-based Constraints, Software-engineering Properties of Languages for Aspect Technologies (SPLAT) 2003, en AOSD 2003, Boston, EEUU, Marzo de 2003
83. [SHUKLA] Dharma Shukla, Simon Fell y Chris Sells, Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. MSDN Magazine, Microsoft, Marzo de 2002.
84. [SOPERA] http://www.mars.dti.ne.jp/~umejava/smalltalk/soapOpera/
85. [SOUL] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. En Proceedings of TOOLS USA '98. IEEE Computer Society Press, agosto de 1998
86. [SQUEAK1] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace y Alan Key, Back to the future: The story of squeak, a practical smalltalk written in itself. En OOSPLA '97 Conference Proceedings, ACM, Atlanta, 1997.
87. [SQUEAK2] http://www.squeak.org

88. [TAN] Andrew S. Tanenbaum, Structured Computer Organization, Ed. Prentice Hall, 4ta edición, 1999, pags. 2-4.
89. [TUCKER] Allen Tucker y Robery Noonan. Programming Languages: Principles and paradigms, Ed. Mc. Graw Hill, 1ra edición, 2002.
90. [TYRUBA] Kris De Volder. Aspect Oriented Logic Meta Programming. En Proceedings of MetaLevel Architectures and Reflection, Second International Conference, Reflection 1999
91. [USCHOLD] M. Uschold, Knowledge level modelling: concepts and terminology. Knowledge engineering review, vol. 13, no. 1, 1998
92. [W3C] Marja-Riitta Koivunen and Eric Miller, W3C Semantic Web Activity. En Proceedings of the Semantic WebKick-off Seminar in Finland, Mayo de 2002.
93. [W3C-RDF] http://www.w3c.org/RDF/
94. [WEAVE.NET] Donald Lafferty y Vinny Cahill, Language Independent Aspect Oriented Programming. En Proceedings of OOPSLA 2003, California, USA.
95. [WELTY] Christopher Welty, An integrated representation for software maintenance and discovery. PhD. Thesis, Rensselaer Polythecnical Institute, 1995.
96. [WHIDBEY] http://lab.msdn.microsoft.com/vs2005/