Facultad Católica de Química e Ingeniería
"Fray Rogelio Bacon"
anexa a la
Pontificia Universidad Católica Argentina

Cátedra de Seminario de Sistemas

"Programación Orientada a Aspectos – POA"

Alonso, Alfredo
Gastaldo, Ma. Celeste
Santamaría, Martín

Rosario, Octubre 2008.

Abstract

La ingeniería del software se encuentra en permanente evolución logrando así mejorar la calidad de los productos de software. Con este avance se fueron introduciendo conceptos que llevaron a una programación de más alto nivel como: la noción de tipos, bloques estructurados, agrupamientos de instrucciones a través de procedimientos y funciones como una forma primitiva de abstracción, unidades, módulos, tipos de datos abstractos, herencia, encapsulamiento. Con técnicas de abstracción de alto nivel conocidas hasta el momento, se logra un diseño y una implementación que satisface la funcionalidad básica, y con una calidad aceptable. Sin embargo, existen conceptos que no pueden encapsularse dentro de una unidad funcional.

Uno de los últimos avances es la Programación Orientada a Aspectos (POA) la cual aspira a soportar la separación de incumbencias de un sistema. Esto implica separar la funcionalidad básica del sistema y los aspectos, a través de mecanismos que permitan abstraerlos y componerlos para formar todo el sistema.

En la presente tesis se desarrollaran las características principales de la Programación Orientada a Aspectos y el grado de adopción de este paradigma en la actualidad. Asimismo se desarrollarán ejemplos asociados a este paradigma, en diversos lenguajes de programación.

En el capítulo 1, se hace un repaso de la historia del paradigma de programación orientada a aspectos, presentando la definición de aspecto como una nueva unidad funcional.

En el capítulo 2 se describen los elementos fundamentales de la Programación Orientada a Aspectos.

En el capítulo 3 se describen los tipos de lenguajes de programación orientados a aspectos, abarcando en profundidad tres lenguajes de programación: AspectJ, AspeCt C y PHPAspect.

En el capítulo 4 se realiza una aplicación práctica en cada uno de los lenguajes de programación seleccionados en el capítulo 3.

En el capítulo 5 se exponen las conclusiones en base a lo redactado en los capítulos anteriores.

Indice de Contenidos

Capítulo 1. Introducción.

1.1 Historia de la POA……………………………………………………………5

1.2 Definición de Aspecto…………………………………………………………..6

Capítulo 2. Fundamentos de la Programación Orientada a Aspectos.

2.1 Características más importantes de la POA y su definición……………………11

2.2 Conceptos Fundamentales introducidos por la POA…………………………..12

2.3 Estructura general de la implementación………………………………………13

2.4 Desarrollo en la Programación Orientada a Aspectos………………………….14

2.5 Tejido Estático versus Tejido Dinámico………………………………………..17

2.6 Guías de Diseño…………………………………………………………………..18

2.7 Ventajas y Desventajas de la POA……………………………………………..19

2.8 Extendiendo UML para soportar Aspectos…………………………………….20

2.9 Métricas de Software para Aspectos…………………………………………...23

2.9.1 Métrica LDCF (Líneas De Código Fuente)……………………………...23

2.9.2 Métricas de Chidamber y Kemerer………………………………………24

2.10 Conclusiones…………………………………………………………………….27

Capítulo 3. Lenguajes de Aspectos.

3.1 Lenguajes de Propósito Específico…………………………………………….29

3.2 Lenguajes de Propósitos Generales…………………………………………….30

3.2.1 AspectJ…………………………………………………………………...34

3.2.2 AspeCt C…………………………………………………………………45

3.2.3 Aspectos con PHP………………………………………………………..55

Capítulo 4. Aplicación Práctica.

4.1 Ejemplos Generales en AspeCt C………………………………………………61

4.2 Carrito básico en PHPAspect…………………………………………………...64

4.3 Solucionando un Registro de Operaciones con AspectJ………………………..68

Capítulo 5. Conclusiones………………………………………………………………79

Anexos

AspectJ: Sintaxis………………………………………………………………………80

AspeCt C……………………………………………………………………………..102

PHPAspect: Generalidades…………………………………………………………...112

aoPHP: Sintaxis……………………………………………………………………….120

Indice de Figuras y Tablas………………………………………………………………….121
Referencias………………………………………………………………………………..123

Capítulo 1. Introducción

En este capítulo se realiza una breve descripción de cómo fue el surgimiento de la Programación Orientada a Aspectos, se presenta una nueva definición de aspecto como una nueva unidad funcional.

1.1 Historia

La definición y terminología de la Programación Orientada a Aspectos (POA) fue introducida por Gregor Kickzales y su grupo en el cual colaboraban Cristina Lopes y Kerl J. Lieberherr. Estos colaboradores eran integrantes del grupo Demeter[KL,1996] el cual había estado utilizando ideas orientadas a aspectos antes de que se acuñara el término.

El grupo Demeter centraba su trabajo en la programación adaptativa (PA), la cual es una instancia previa a la POA. La programación adaptativa se introdujo en 1991, ésta constituyo un gran avance en la ingeniería del software, la misma se basa en autómatas finitos y en una teoría formal de lenguajes para expresar y procesar conjuntos de caminos de un grafo arquitectónico, por ejemplo los diagramas de clase en UML.

Se relaciona a la PA con la POA mediante la Ley de Demeter conocida de forma abreviada como: "Solo conversa con tus amigos inmediatos"[KL,1988], se puede decir que es una regla de estilo para el diseño de sistemas orientados a objetos. Produce una clara separación de las incumbencias de comportamiento e incumbencias de funcionalidad básica.

La separación de incumbencias fue un tema de interés de varios integrantes del grupo como Cristina Lopes y Walter Hürsch en 1995, los cuales presentaron un reporte técnico sobre separación de incumbencias[WH,1995], incluyendo técnicas como filtros composicionales y programación adaptativa para tratar con los conceptos que se entremezclan. Este reporte presentó el tema de la separación de incumbencias y su implementación proponiéndose como un tema importante a resolver.

La primera definición del concepto de aspecto fue publicada en 1995 por el grupo Demeter y la misma decía: *"Un aspecto es una unidad que se define en términos de información*

[KL,1996] Karl Lieberherr, *Adaptative Object-Oriented Software. The Demeter Method*, College of Computer Science, Northeastern University Boston, 1996
[KL,1988] K. Lieberherr, I. Holland, A. Riel, *Object-Oriented Programming:An Objective Sense of Style*, College of Computer Science Northeastern University, 1988.
[WH,1995] Walter L. Hürsch and Cristina Videira Lopes, *Separation of Concerns*, in College of Computer Science, Northeastern University Boston, February 24 1995

parcial de otras unidades". La definición de aspecto ha evolucionado con el tiempo hasta llegar a la definitiva creada por Gregor Kickzales, la cual es citada más adelante.

1.2 Definición de Aspecto

Según la Real Academia Española se define[RA,2001]: aspecto (Del Lat. aspectus).

1. m. Apariencia de las personas y los objetos a la vista. El aspecto venerable de un anciano. El aspecto del campo, del mar.
2. m. Elemento, faceta o matiz de algo. Los aspectos más interesantes de una obra.
3. m. Particular situación de un edificio respecto al Oriente, Poniente, Norte o Mediodía.
4. m. Astr. Fases y situación respectiva de dos astros con relación a las casas celestes que ocupan.
5. m. Gram. En ciertas lenguas, categoría gramatical que expresa el desarrollo interno de la acción verbal, según se conciba esta como durativa, perfecta o terminada, reiterativa, puntual, etc.

De estas definiciones, la más aproximada al concepto de Aspecto como será tratada en este trabajo es la número 2.

Una de las primeras definiciones que aparecieron del concepto de Aspecto fue publicada en 1996[KL,1996], y se describía de la siguiente manera: *"Un Aspecto es una unidad que se define en términos de información parcial de otras unidades"*.

Gregor Kickzales, creador del paradigma de Programación Orientada a Aspectos (POA), define a un Aspecto como: *"una unidad modular que se disemina por la estructura de otras unidades funcionales. Los Aspectos existen tanto en la etapa de diseño como en la de implementación. Un Aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un Aspecto de implementación es una unidad modular del programa que aparece en otras unidades modulares del programa"*.

[RA,2001] Diccionario de la Real Academia Española - Vigésima segunda edición - http://www.rae.es/. Consultado en Agosto 2008.

[KL,1996] Karl Lieberherr, *Adaptative Object-Oriented Software. The Demeter Method*, College of Computer Science, Northeastern University Boston, 1996.

De manera más informal podemos decir que los Aspectos son la unidad básica de la POA, y pueden definirse como los elementos que se diseminan por todo el código, que son difíciles de describir localmente con respecto a otros componentes y que se van a implementar de forma modular y separada del resto del sistema. El ejemplo más común y simple de un aspecto es el registro de sucesos dentro del sistema, ya que necesariamente afecta a todas las partes del sistema que generan un suceso.

Se puede diferenciar entre un Componente y un Aspecto viendo al primero como aquella propiedad que se puede encapsular claramente en un procedimiento, mientras que un aspecto no se puede encapsular en un procedimiento con los lenguajes tradicionales[GK,1997].

Los Aspectos no suelen ser unidades de descomposición funcional del sistema, sino propiedades que afectan al rendimiento o la semántica de los componentes. Algunos ejemplos de Aspectos son: los patrones de acceso a memoria, la sincronización de procesos concurrentes, el manejo de errores, el registro de sucesos, etc.

La idea central que persigue la POA es permitir que un programa sea construido describiendo cada incumbencia separadamente.

El soporte para este nuevo paradigma se logra a través de una clase especial de lenguajes, llamados Lenguajes Orientados a Aspectos (LOA), los cuales brindan mecanismos para capturar aquellos elementos que se diseminan por todo el sistema. A estos elementos se les da el nombre de Aspectos.

A modo de ejemplo[GO,2004]: Supongamos que estamos desarrollando una aplicación Web donde un determinado código captura los valores ingresados a un formulario HTML, los asigna a un objeto, procesa la información y luego retorna la respuesta al usuario. En una primera instancia el código desarrollado sería muy simple y pequeño. Sin embargo, la cantidad de código comienza a crecer debido a la implementación de requerimientos secundarios, los aspectos, como el procedimiento de inicio de sesión, el manejo de excepciones y la seguridad. Los requerimientos secundarios afectan al código generado inicialmente dado que éste no debería tener que administrar sesiones, excepciones o seguridad para poder cumplir su requerimiento principal, aceptar la entrada del usuario, procesarla y devolver el resultado.

[GK,1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, *Aspect-Oriented Programming*, Xerox Palo Alto Research Center, 1997.

[GO,2004] Graham O'Regan, *ONJava.com - Introduction to Aspect-Oriented Programming*, O'Reilly. Enero 2004.

En la Figura 1, se muestra en forma gráfica una comparación de la apariencia que presentan un Programa Tradicional y un Programa Orientado a Aspectos.

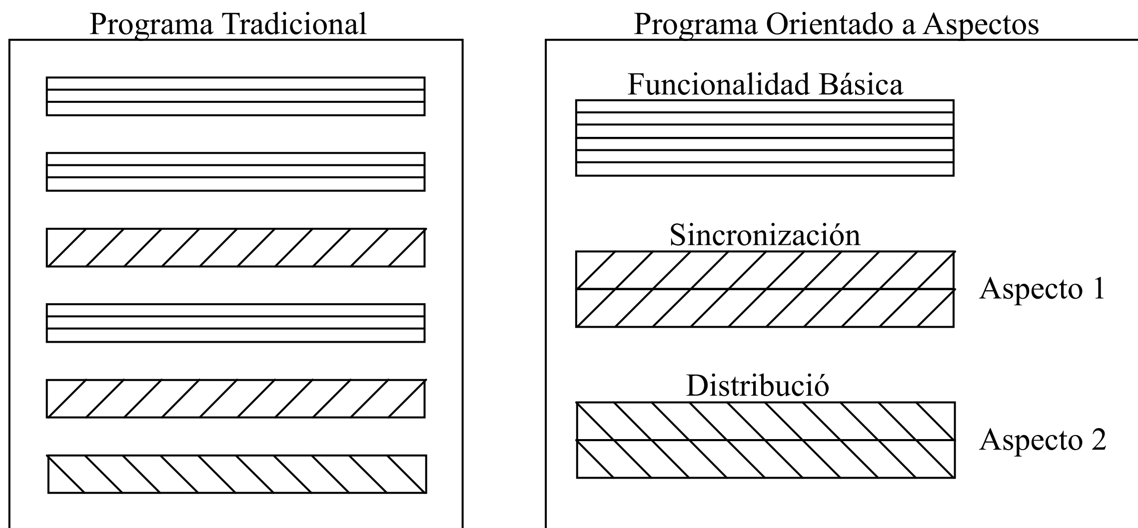


Figura 1: Programa Tradicional vs. Programa Orientado a Aspectos

La POA permite modificar dinámicamente el modelo estático incluyendo el código requerido para cumplir los requerimientos secundarios sin tener que modificar el modelo estático original.

Un segundo ejemplo[MV,2004] ayudará más a comprender la noción de aspecto. Imaginemos una aplicación tradicional que trabaja concurrentemente sobre datos compartidos. Los datos compartidos pueden estar encapsulados en un objeto Dato (una instancia de la clase Datos). En la aplicación, existen múltiples objetos de diferentes clases trabajando en un solo objeto Dato, en donde sólo uno de estos objetos puede tener acceso al dato compartido a la vez. Para lograr el comportamiento deseado, se tiene que establecer algún tipo de bloqueo. Esto significa, que cuando uno de dichos objetos quiere acceder al dato, el objeto Dato debe ser bloqueado, y luego desbloqueado cuando se ha terminado de utilizarlo. El enfoque tradicional implica introducir una clase base abstracta, heredada por todas las clases trabajadoras que necesiten establecer los bloqueos al objeto Dato. Ésta clase define al método `bloqueo()` y al método `desbloqueo()` los que deben ser llamados antes y después de que el trabajo sea realizado. Este enfoque tiene los siguientes inconvenientes:

- Cada método que trabaje con los datos debe ocuparse de las acciones bloquear y desbloquear. El código es mezclado con declaraciones correspondientes al bloqueo.

[MV,2004] Markus Voelter, *Aspect-Oriented Programming in Java*.

- En un mundo de herencia simple, no es siempre posible permitir a las clases heredar desde una clase base común, el único vínculo de herencia ya puede estar establecido con otra clase.

- Se compromete la reusabilidad: las clases trabajadoras pueden ser reutilizadas en otro contexto donde no es necesario realizar el bloqueo o se utiliza otra técnica para lograrlo. Al poner el código de bloqueo en las clases trabajadoras, las clases se ven atadas a la perspectiva utilizada en esa aplicación específica.

El concepto de bloqueo en el ejemplo anterior puede ser descrito con las siguientes propiedades:

- No es el trabajo primario de las clases trabajadoras.

- El esquema de bloqueo es independiente del trabajo principal de la clase trabajadora.

- El bloqueo entrelaza el sistema y probablemente varios métodos de esas clases sean afectadas por los procedimientos de bloqueo y desbloqueo.

Para manejar este tipo de problemas, la programación orientada a aspectos propone un enfoque alternativo: Definir una nueva construcción para que se haga cargo de las incumbencias transversales del sistema. Esta construcción se llama aspecto.

El nuevo aspecto "bloquear" tendría las siguientes responsabilidades:

- Proveer las características necesarias para bloquear y desbloquear objetos a las clases que tienen que ser bloqueadas y desbloqueadas (en nuestro ejemplo a la clase Datos).

- Asegurar que todos los métodos que modifiquen al objeto `Dato` llamen a `bloqueo()` antes de realizar su trabajo y a `desbloqueo()` luego de haberlo terminado (en nuestro trabajo, las clases trabajadoras).

No existe la **necesidad** de utilizar aspectos para resolver problemas ya que cualquier problema puede ser resuelto sin su uso. Pero ellos proveen un nuevo y más alto nivel de abstracción que logra simplificar el proceso de diseño y de entendimiento del sistema. A medida que los sistemas crecen en tamaño, el entendimiento del mismo se torna más complicado, por lo que los aspectos se convierten en una herramienta clave para el entendimiento.

Se puede decir que los aspectos quiebran el encapsulamiento de los objetos pero de forma

controlada. Los aspectos tienen acceso a la parte privada de los objetos a los cuales están asociados, pero no comprometen el encapsulamiento entre objetos de diferentes clases.

Capítulo 2. Fundamentos de la Programación Orientada a Aspectos.

En este capítulo se describen los elementos fundamentales de la Programación Orientada a Aspectos. En primer lugar se definen sus características más importantes, conceptos fundamentales, la estructura general en su implementación, se muestra como es el desarrollo de la POA, tipos de tejidos utilizados para generar código, ventajas y desventajas de su uso y métricas de software asociadas a la POA .

2.1 Características más importantes de la POA y su definición[RH,2005]

Las características relevadas de la POA se pueden definir de la siguiente forma:

1. **Punto de unión o enlace (join point)**: Representa un "momento" en la ejecución de un programa, por ejemplo, una llamada a un método, o un constructor, o el acceso a un miembro de una clase en particular.

2. **Intersección o punto de corte (pointcut)**: Declaración de un conjunto de puntos de unión, por ejemplo, "llamadas a los métodos que empiecen por set".

3. **Guía/Consejo/Orientación (advice)**: Comportamiento que queremos invocar cuando se alcance un determinado punto de intersección. Es el código que se ejecuta en un punto de unión en particular. Existen varios tipos de *advices*, por ejemplo el *before*, que se ejecuta antes de un punto de unión o el *after* que se ejecuta después.

4. **Aspecto (aspect)**: es la combinación de *advices* y puntos de corte. Esta combinación es el resultado de la definición lógica de lo que debe tener la aplicación y cuando debe ejecutarse.

5. **Objetivo (target)**: Un objeto con un flujo de ejecución que se modifica por algún proceso POA que es llamado como un *target object* a veces llamado también *advised object*.

6. **Introducción (introduction)**: Comportamiento que añadimos a un objeto en tiempo de ejecución para que implemente una interfaz adicional aparte de la suya original.

7. **Weaving (tejido)**: es el proceso de insertar aspectos en el código de la aplicación en el punto apropiado. Para las soluciones POA a tiempo de compilación, se hace en la

[RH,2005] Rob Harrop y Jan Machacek, *Pro Spring*, Apress, 2005.

compilación usualmente como un paso extra en el proceso de construcción. Y en las soluciones POA en ejecución el proceso de tejido se ejecuta dinámicamente en el tiempo de ejecución. El momento en que se realice el tejido define el tipo de aspecto.

Por ejemplo, se tiene una clase objetivo que se desea modificar. Para ello se establece un punto de intersección para la clase objetivo y se agrega un *advice*, para insertar una nueva función. De esta manera, se crea un aspecto, el cual insertará su nuevo código con el proceso de tejido.

La forma de crear un aspecto, ya sea de forma dinámica (tiempo de ejecución) o estática (tiempo de compilación), es definido por la herramienta que se utilice en POA. Por ejemplo, AspectJ crea a los aspectos en tiempo de compilación, mientras que Spring lo hace en tiempo de ejecución.

2.2 Conceptos fundamentales introducidos por la POA[FA,2003]

Para poder comenzar a programar en el paradigma de Aspectos se necesita:

- **Lenguaje base**: Un lenguaje para definir la funcionalidad básica. Suele ser un lenguaje de propósito general. Por ejemplo, C++ o Java (por citar algunos de ellos).
- **Lenguaje orientado a aspectos**: Uno o varios lenguajes de aspectos (COOL, AspectJ, etc.). El lenguaje de aspectos define la forma de los aspectos – como por ejemplo, los aspectos de AspectJ se programan de forma muy parecida a las clases.
- **Un tejedor**: El tejedor se encarga de combinar los lenguajes. El proceso de mezcla puede hacerse en tiempo de ejecución o en tiempo de compilación.

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software para encapsular aquellas incumbencias que atraviesan todo el código. De la misma manera, estos lenguajes deben soportar la separación de incumbencias como la sincronización, la distribución, el manejo de errores, la optimización de memoria y la seguridad, entre otras. Está claro que debe existir una interacción entre los componentes y los aspectos. Para que los componentes y los aspectos se puedan mezclar, deben existir algunos puntos en común que se denominan "puntos de enlace".

[FA,2003] Fernando Asteasuain, Bernando E. Contrares, Elsa Estévez, Pablo R. Fillotrani, *Programación Orientada a Aspectos: Metodología y Evaluación, Departamento de Ciencias e Ingeniería de la computación*, Universidad Nacional del Sur, Año 2003.

Los **puntos de enlace**, son una clase especial de interfaz entre los aspectos y los módulos del lenguaje de componentes. Son los lugares del código en los que este se puede aumentar con comportamientos adicionales. Estos comportamientos se especifican en los aspectos.

El encargado de realizar este proceso de mezclar las dos partes (los módulos del lenguaje de componentes y los aspectos) es el tejedor (weaver).

El tejedor se encarga de mezclar los diferentes mecanismos de abstracción y composición que aparecen en los lenguajes de aspectos y componentes gracias a los puntos de enlaces.

2.3 Estructura general de la Implementación

La estructura de una implementación basada en aspectos es análoga a la estructura de una implementación basada en LPG (Lenguajes de Programación General).

La estructura de LPG se define de la siguiente manera:

- un lenguaje.
- un compilador o intérprete para ese lenguaje.
- Un programa escrito en ese lenguaje que implemente la aplicación.

La Figura 2 muestra el funcionamiento de la estructura LPG:

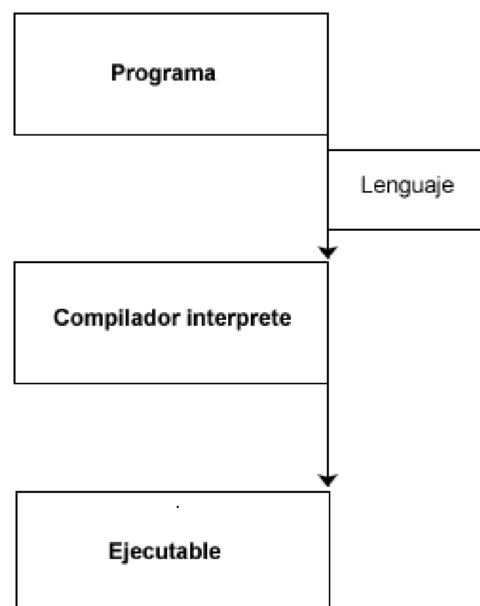


Figura 2: Estructura de un Lenguaje de Programación General

La estructura para una implementación basada en POA consiste en:

- El lenguaje base o componente para programar la funcionalidad básica.
- Uno o más lenguajes de aspectos para especificar los aspectos.
- Un tejedor de aspectos para la combinación de los lenguajes.
- El programa escrito en el lenguaje componente que implementa los componentes.
- Uno o más programas de aspectos que implementan los aspectos.

En la Figura 3 se muestra gráficamente la estructura de una implementación en los Lenguajes de Aspectos.

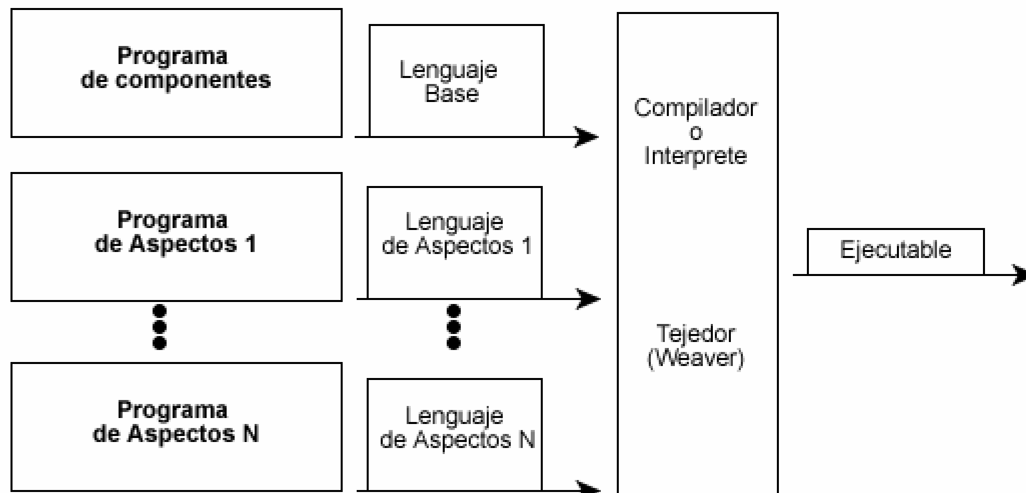


Figura 3: Estructura de un Lenguaje Orientado a Aspectos

Se puede destacar al comparar ambas figuras que la función que realizaba el compilador se encuentra ahora incluida en las funciones del tejedor.

2.4 Desarrollo en la programación orientada a aspectos

Realizar un diseño basado en el paradigma de la programación orientada a aspecto no es sencillo, ya que se debe tener bien en claro previo al desarrollo qué se debe incluir en el lenguaje base, qué se debe incluir dentro de los lenguajes de aspectos y qué debe compartirse entre ambos lenguajes.

El encargado de proveer la funcionalidad básica es el lenguaje de componentes y

también es el encargado de que los lenguajes componentes no interfieran con los aspectos. Los lenguajes de aspectos tienen que proveer los medios para implementar los aspectos deseados de manera intuitiva, natural y concisa.

Básicamente, el desarrollo de una aplicación basado en aspectos consiste en tres grandes pasos.

1. **Descomposición de aspectos (Aspectual Decomposition)**: descomponer los requerimientos para distinguir aquellos que son componentes de los que son aspectos.
2. **Implementación de incumbencias (Concern Implementation)**: implementar cada requerimiento por separado.
3. **Recomposición (Aspectual Recomposition)**: dar las reglas de recomposición que permitan combinar el sistema completo.

John Lamping[JL,1999] propone una visión diferente. Mantiene que la decisión sobre qué conceptos son base y cuáles son manejados por aspectos es irrelevante ya que no afectará a la estructura del programa. Dice que la clave está en definir cuáles serán los ítems de la funcionalidad básica y cómo obtener una clara separación de responsabilidades entre los conceptos. La primera parte se hereda de la programación no orientada a aspectos y tiene la misma importancia dentro de la POA ya que la misma mantiene la funcionalidad básica. La segunda parte es inherente a la programación orientada a aspectos.

Pueden existir varios aspectos para contribuir a la implementación de un mismo ítem de la funcionalidad básica y un solo aspecto puede contribuir a la implementación de varios ítems. Una buena separación de responsabilidades entre las incumbencias es lo que hace esto posible, porque el comportamiento introducido por los diferentes aspectos se enfoca en diferentes temas en cada caso, evitando gran parte de los conflictos.

Como conclusión, Lamping dice que el trabajo del programador que utiliza POA es definir en forma precisa los ítems de la funcionalidad básica y obtener una buena separación de responsabilidades entre las incumbencias. Luego, los aspectos le permitirán al programador separar las incumbencias en el código.

En nuestra opinión, Lamping no da una visión completamente diferente, sino que

[JL,1999] John Lamping, *"The role of the base in aspect oriented programming"*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP) workshops, 1999.

concluye llegando a los tres primeros pasos de la programación orientada a aspectos de una manera un tanto más compleja pero que en esencia no demuestra nada diferente.

Un informe de la Universidad de Virginia, afirma que muchos de los principios centrales de la programación orientada a objetos son ignorados. Menciona por ejemplo el ocultamiento de la información, debido a que los aspectos tienen la habilidad de violar estos principios. Para esto se propone una filosofía diferente de diseño orientada a aspectos, que consiste en cuatro pasos[TH,1999].

1. **Un objeto es algo**: un objeto existe por sí mismo, es una entidad.

2. **Un aspecto no es algo, es algo sobre algo**: Un objeto se escribe para encapsular un concepto entrecruzado. Por definición un aspecto entrecruza diferentes componentes, los cuales en la POO son llamados objetos. Si un aspecto no está asociado en este contexto. Luego, para que un aspecto tenga sentido debe estar asociado a una o más clases; no es una unidad funcional por sí mismo.

3. **Los objetos no dependen de los aspectos**: Un aspecto no debe cambiar las interfaces de las clases asociadas a él. Sólo debe aumentar la implementación de dichas interfaces. Al afectar solamente la implementación de las clases y no sus interfaces, la encapsulación no se rompe. Las clases mantienen su condición original de cajas negras, aún cuando puede modificarse el interior de las cajas.

4. **Los aspectos no tienen control sobre los objetos**: Esto significa que el ocultamiento de información puede ser violado en cierta forma por los aspectos porque éstos conocen detalles de un objeto que están ocultos al resto de los objetos. Sin embargo, no deben manipular la representación interna del objeto más allá de lo que sean capaz de manipular el resto de los objetos. A los aspectos se les permite tener una visión especial, pero debería limitarse a manipular objetos de la misma forma que los demás objetos lo hacen, a través de la interfaz.

Esta filosofía planteada, en cambio, permite a los aspectos hacer su trabajo de automatización y abstracción, respetando los principios de ocultamiento de información e interfaz[MB,1998].

[TH,1999] Timothy Highley, Michael Lack, Perry Myers, *Aspect-Oriented Pragramming: A Critical Analysis of a new programming paradigm*, University of Virginia, Departament of computer Science, Technical Report CS-99-29, Mayo 1999.

[MB,1998] MacLennan B, *Principles of programming languages*, 3º Edicion, John Wiley & Sons, 1998.

Esta nueva filosofía nos parece muy cercana a lo que se pretende con la programación orientada a aspectos, se debería aplicar y conocer durante todo el desarrollo en la programación orientado a aspectos. Sería muy apropiado llevar esta política a formar parte de las reglas del lenguaje orientado a aspectos. Es decir, que no sea una opción de programación, sino que verdaderamente sea forzado por el lenguaje a trabajar de esta forma. El mismo problema que antes existía entre las distintas componentes de un sistema, está presente ahora entre objetos y aspectos. Es decir, los aspectos no deben meterse en la representación interna del objeto, por lo tanto se necesita una solución similar: el lenguaje debería proveer una interfaz para que aspectos y objetos se comuniquen respetando esa restricción.

Creemos que una de las principales desventajas de los lenguajes orientados a aspectos actuales. Uno de los principales desafíos está en construir lenguajes orientados a aspectos capaces de brindar mecanismos lingüísticos suficientemente poderosos para respetar por completo todos los principios de diseño.

2.5 Tejido estático versus Tejido dinámico

La primera decisión que hay que tomar al implementar un sistema orientado a aspectos es relativa a las dos formas de entrelazar el lenguaje componente con el lenguaje de aspectos. Dichas formas son el entrelazado o tejido estático y el entrelazado o tejido dinámico.

El entrelazado estático implica modificar el código fuente escrito en el lenguaje base, insertando sentencias en los puntos de enlace. Es decir, que el código de aspectos se introduce en el código fuente. Un ejemplo de este tipo de tejedor, es el tejedor de aspectos de AspectJ.

El entrelazado dinámico requiere que los aspectos existan y estén presentes de forma explícita tanto en tiempo de compilación como en tiempo de ejecución. Para conseguir esto, tanto los aspectos como las estructuras entrelazadas se deben modelar como objetos y deben mantenerse en el ejecutable. Un tejedor dinámico será capaz añadir, adaptar y remover aspectos de forma dinámica durante la ejecución. Como ejemplo, el tejedor dinámico AOP/ST utiliza la herencia para añadir el código específico del aspecto a las clases, evitando modificar así el código fuente de las clases al entrelazar los aspectos.

El tejido estático evita que el nivel de abstracción introducido por la POA derive en un impacto negativo en la eficiencia de la aplicación, ya que todo el trabajo se realiza en tiempo de compilación, y no existe sobrecarga en ejecución. Si bien esto es deseable, el costo es una menor flexibilidad: los aspectos quedan fijos, no pueden ser modificados en tiempo de ejecución, ni

existe la posibilidad de agregar o remover nuevos aspectos. Otra ventaja que surge es la mayor seguridad que se obtiene efectuando controles en compilación, evitando que surjan errores catastróficos o fatales en ejecución. Podemos agregar también que los tejedores estáticos resultan más fáciles de implementar y consumen menor cantidad de recursos.

El tejido dinámico significa que el proceso de composición se realiza en tiempo de ejecución, y de esta forma decrementa la eficiencia de la aplicación.

El postergar la composición brinda mayor flexibilidad y libertad al programador, ya que cuenta con la posibilidad de modificar un aspecto según información generada en ejecución, como también introducir o remover dinámicamente aspectos. La característica dinámica de los aspectos pone en riesgo la seguridad de la aplicación, ya que se puede dinámicamente remover comportamiento de un aspecto que quizás luego se requiera, o más grave aún, qué sucede si un aspecto en su totalidad es removido, y luego se hace mención al comportamiento de ese aspecto de otra manera.

El tener que llevar mayor información en ejecución, y tener que considerar más detalles, hace que la implementación de los tejedores dinámicos sea más compleja. Es importante notar que la naturaleza dinámica hace referencia a características o propiedades de un aspecto, y no al aspecto en sí mismo.

2.6 Guías de Diseño

Tomando como base la guía de diseño para la implementación de sistemas abiertos, podemos enunciar las siguientes pautas para aquellos tejedores que soporten tanto aspectos dinámicos como estáticos[GK,1997]:

- Los aspectos dinámicos deben separarse claramente de los aspectos estáticos, como por ejemplo, a través de un constructor o palabra reservada que indique la naturaleza del aspecto. Los aspectos estáticos están precedidos por la palabra reservada *"static"*, y los dinámicos por la palabra reservada *"dynamic"*. La especificación de la naturaleza del aspecto debe ser opcional[PK,2000].

[GK,1997] Gregor Kickzales, John Lamping, Cristina Lopes, Chris Maeda, Anurag Mendhekar, Gail Murphy, *"Open Implementation Design guidelines"*, Proceedings of the 19th International Conference on Sofware Engineering (Boston, MA), Mayo de 1997.

[PK,2000] Peter Kenens, Sam Michiels, Frank Matthijs, Bert Robben, Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, *"An AOP Case with Static and Dynamic Aspects"*, Departament of Computer Science, K.U.Leuven, Bélgica.

- El alcance de la influencia de dicha especificación tiene que ser controlado de una forma natural y con suficiente granularidad. Esta pauta ayuda al programador a entender y razonar sobre su programa.

- El conocimiento que tiene el cliente sobre los detalles internos de implementación debe ser mínima. Por ejemplo, el cliente podría elegir diferentes niveles de seguridad para el aspecto Seguridad-Edificio (alto, mediano o bajo) en ejecución, sin conocer cómo está implementada cada alternativa.

2.7 Ventajas y Desventajas de la POA

Algunas de las principales **ventajas**[RL,2002] del paradigma de Programación Orientado a Aspectos son:

- Ayuda a superar los problemas causados por el:

  - **Código Mezclado (Code Tangling)**: Se presenta cuando en un mismo módulo de un sistema de software conviven simultáneamente más de un requerimiento. Esto hace que en el modelo existan elementos de implementación de más de un requerimiento.

  - **Código Diseminado (Code Scattering)**: Como los requerimientos están esparcidos sobre varios módulos, la implementación resultante también queda diseminada sobre esos módulos.

- **Implementación modularizada**: POA logra separar cada concepto con mínimo acoplamiento, resultando en implementaciones modularizadas aún en la presencia de conceptos que se entrecruzan. Esto lleva a un código más limpio, menos duplicado, más fácil de entender y de mantener.

- **Mayor posibilidad de evolución**: La separación de conceptos permite agregar nuevos aspectos, modificar y / o remover aspectos existentes fácilmente.

- **Mejor diseño**: Permite retrasar las decisiones de diseño sobre requerimientos actuales o que surjan en el futuro, ya que se los puede implementar separadamente, e incluirlos automáticamente en el sistema.

[RL,2002] Ramnivas Laddad, *"I want my AOP"*, JavaWorld, Enero 2002

- **Mayor reusabilidad**: Al ser implementados separadamente, tienen mayores probabilidades de ser reusados en otros sistemas con requerimientos similares.

- Al separar la funcionalidad básica de los aspectos, se aplica con mayor intensidad el principio de dividir y conquistar.

- **Permite N-dimensiones**: se tiene la posibilidad de implementar el sistema con las dimensiones que sean necesarias, no una única dimensión "sobrecargada".

- Hace hincapié en el principio de mínimo acoplamiento y máxima cohesión.

Las **desventajas** más evidentes se encuentran en los Lenguajes Orientados a Aspectos (LOA)[GC,1999], las cuáles surgen del hecho de que la POA está en su desarrollo temprano:

- Posibles choques entre el código funcional (expresado en el lenguaje base) y el código de aspectos (expresados en los lenguajes de aspectos). Por lo general estos choques nacen de la necesidad de violar el encapsulamiento para implementar los diferentes aspectos.

- Posibles choques entre los aspectos. El ejemplo clásico es tener dos aspectos que trabajan perfectamente por separado pero al aplicarlos conjuntamente resultan en un comportamiento anormal.

- Posibles choques entre el código de aspectos y los mecanismos del lenguaje.

2.8 Extendiendo UML para soportar Aspectos

La propuesta que se realiza en[JS, 99] implica la extensión del metamodelo de UML para incluir a los Aspectos en la fase de diseño. Para ello se agregan nuevos elementos al metamodelo para el Aspecto ("Clase tejida") y se reutiliza un elemento ya existente para la relación clase-aspecto.

[GC,1999] Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga, *"Language Support for Evolvable Software: An Initial Assessment of Aspect-Oriented Programming"*, Proceedings of the International Workshop on the Principles of Software Evolution, IWPSE99, Julio 1999.

[JS, 1999] Junichi Suzuki, Yoshikazu Yamamoto. Extending UML with Aspect: Aspect Support in the Design Phase. 3er Aspect-Oriented Programming (AOP) Workshop at ECOOP'99.

Aspecto

El aspecto se añade como un nuevo constructor derivado del elemento `Clasificador` (Figura 4) que describe características estructurales y de comportamiento [OMG, 07].

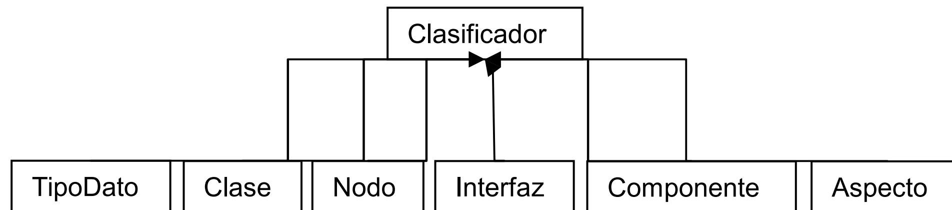


Figura 4: Aspecto como un elemento del metamodelo UML derivado de Clasificador

Un aspecto puede tener atributos, operaciones y relaciones. Los atributos son utilizados por su conjunto de definiciones tejidas. Las operaciones se consideran como sus declaraciones tejidas. Las relaciones incluyen la generalización, la asociación y la dependencia.

El aspecto se representa como un rectángulo de clase con el estereotipo `<<aspecto>>` (Figura 5). En la sección de operaciones se muestran las declaraciones tejidas. Cada tejido se muestra como una operación con el estereotipo `<<tejido>>`. Los atributos se muestran en la sección de atributos.

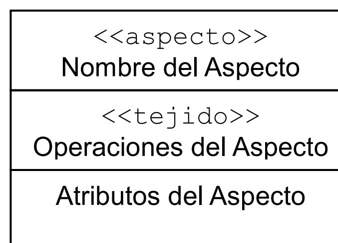


Figura 5: Representación de un aspecto en UML extendido

Relación Aspecto-Clase

En el metamodelo de UML se definen tres relaciones básicas, que se derivan del elemento Relación del metamodelo: Asociación, Generalización y Dependencia [OMG, 07].

La relación entre un aspecto y su clase es un tipo de relación de dependencia. La relación de dependencia modela aquellos casos en que la implementación o el funcionamiento de un elemento necesita la presencia de otro u otros elementos.

En el metamodelo UML del elemento Dependencia se derivan otros cuatro que son: Abstracción, Ligadura, Permiso y Uso (Figura 6). La relación aspecto-clase se clasifica como

una relación de dependencia del tipo abstracción.

Una relación de dependencia de abstracción relaciona dos elementos que se refieren al mismo concepto pero desde diferentes puntos de vista, o aplicando diferentes niveles de abstracción. El metamodelo UML define también cuatro estereotipos[OMG,2007] para la dependencia de abstracción: derivación, realización, refinamiento y traza.

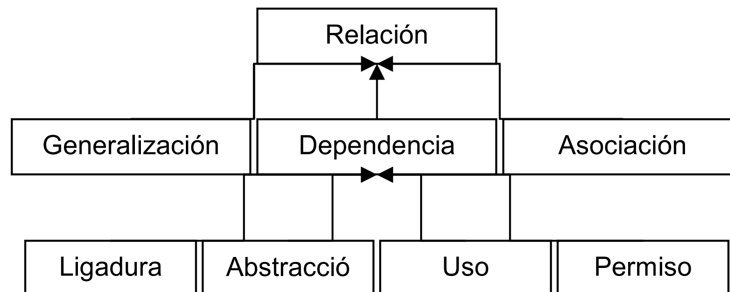


Figura 6: tipos de relaciones del metamodelo de UML

Con el estereotipo `<<realiza>>` se especifica una relación de realización entre un elemento o elementos del modelo de especificación y un elemento o elementos del modelo que lo implementa. El elemento del modelo de implementación ha de soportar la declaración del elemento del modelo de especificación.

La relación aspecto-clase se recoge en la dependencia de abstracción con el estereotipo realización (`<<realiza>>`). Esta relación se representa en la Figura 7.

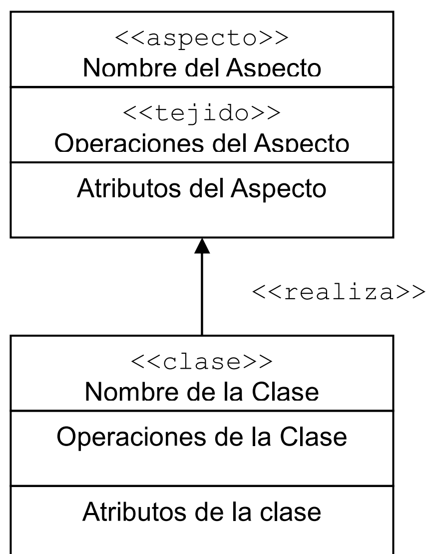


Figura 7: Notación de la relación aspecto-clase

[OMG,2007] OMG. Unified Modeling Language Specification v2.1.2, 2007.

Clase Tejida

Al utilizar un tejedor de aspectos, el código de las clases y los aspectos se mezclan y se genera como resultado una clase tejida. La estructura de la clase tejida depende del tejedor de aspectos y del lenguaje de programación que se haya utilizado. Por ejemplo, AspectJ reemplaza la clase original por la clase tejida, mientras que AOP/ST genera una clase tejida que se deriva de la clase original. La solución que proponen[JS,1999] es introducir el estereotipo `<<clase tejida>>` en el elemento Clase para representar una clase tejida. Se recomienda que en la clase tejida se especifique con una etiqueta la clase y el aspecto que se utilizaron para generarla. En la Figura 8 se representa la estructura de las clases tejidas, utilizando tanto el tejedor AspectJ como el tejedor AOP/ST.

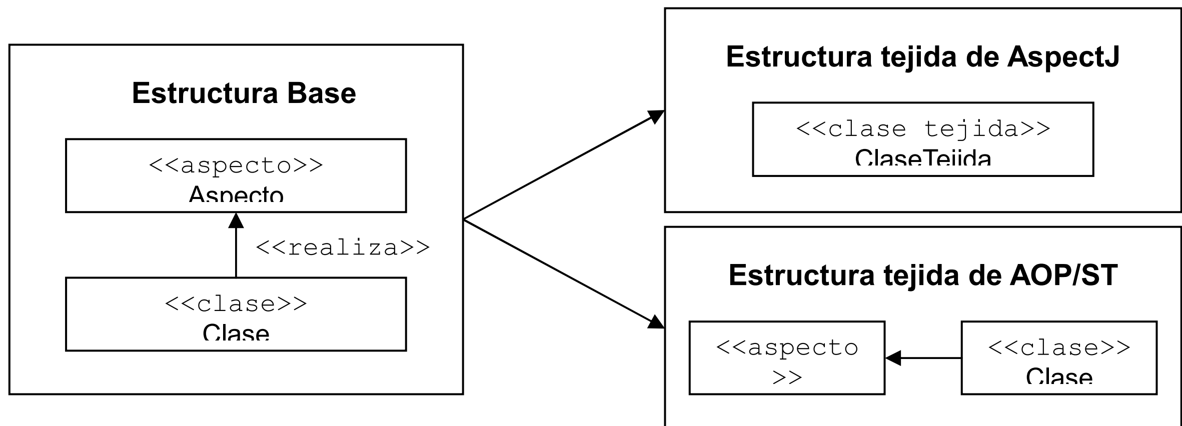


Figura 8: Estructura de las clases tejidas

2.9 Métricas de Software

Las métricas de software intentan medir la complejidad del software para predecir el costo total del proyecto y la evaluación de la calidad y la efectividad del diseño. Estas tienen múltiples aplicaciones en las distintas tareas de la ingeniería de software tales como pruebas, refactoring, gerenciamiento y mantenimiento.

2.9.1 Métrica LDCF (Líneas De Código Fuente)

LDCF (SLOC por sus siglas en inglés) es una métrica de software utilizado para medir el tamaño de un programa de software mediante el conteo del número de líneas en el texto del

[JS,1999] Junichi Suzuki, Yoshikazu Yamamoto. Extending UML with Aspect: Aspect Support in the Design Phase. 3er Aspect-Oriented Programming (AOP) Workshop at ECOOP'99.

código fuente del programa. LDCF es usado típicamente para predecir la cantidad de esfuerzo que será requerido para desarrollar un programa, así como estimar la productividad de la programación o esfuerzo una vez que el software ha sido producido.

Existen dos tipos de medidas de LDCF: LDCF físicas y LDCF lógicas.

En las LDCF físicas se cuentan las líneas de texto del código fuente incluyendo los comentarios. Las líneas en blanco también son incluidas a menos que sean mayores al 25% del total para una sección. En ese caso las líneas que superan el 25% no son contadas para el total de LDCF. Una contrapartida que posee LDCF físicas, es que sensitiva al formato del texto.

Las LDCF lógicas tienen en cuenta el número de "declaraciones" en el código fuente. Hay que tener en cuenta que la definición de una "declaración" depende del lenguaje utilizado.

Los aspectos pueden ayudar a reducir la cantidad de líneas por código. Al reunir el código entrecruzado a lo ancho de las clases en una nueva unidad modular, el aspecto, la cantidad de líneas de código disminuye, siendo más fácil su mantenimiento y comprensión.

2.9.2 Métricas de Chidamber y Kemerer[CH,1993]

Las métricas utilizadas para la evaluación de sistemas orientados a objetos son aplicables a los sistemas orientados a aspectos, como es el caso de las seis métricas C&K definidas por Chidamber Kemerer.

Veremos como estas métricas se ven afectadas al aplicarse a sistemas Orientados a Aspectos.

1. **Peso de los Métodos por Clase (WMC)**

PMC (WMC por sus siglas en inglés, Weighted Methods per Class) cuenta la cantidad de métodos que componen una clase.

En lo posible hay que intentar de mantener el PMC bajo. Las clases con muchos métodos tienen mayores probabilidades de estar muy atadas a una aplicación, limitando su posibilidad de reuso. Esto también afecta a las clases derivadas, dado que heredan los métodos de las clases base.

[CH,1993] Shyam R. Chidamber, Chris F. Kemerer. *A Metrics suite for Object Oriented design*. M.I.T. Sloan School of Management E53-315. 1993

Los aspectos combinan el entrecruzamiento de funcionalidades en unidades modulares y encapsuladas. Sin el diseño orientado a aspectos, entrecruzamiento de funcionalidades estaría desparramado en la clase.

2. Profundidad del Árbol de Herencia (DIT)

Cuanto más profunda sea una clase en la jerarquía, tendrá más métodos heredados, haciéndola más compleja. Una gran profundidad indica una gran complejidad en el diseño, aunque por otro lado, promueven el reuso debido a la herencia de métodos.

Una gran profundidad en el árbol de herencia suele tener mayores errores, aunque no necesariamente en las clases más profundas. Glasberg[GD,2000] indica las clases más propensas a contener errores son las que están en la mitad del árbol. Las que están en los extremos suelen ser las más consultadas y usadas, por lo que suelen tener menos errores que las clases menos vistas.

Se recomiendan hasta 5 clases de profundidad.

Las subclases que deban ser definidas con el propósito de extender su funcionalidad para realizar acciones que podrían ser realizadas por un aspecto, no existirían en sistemas desarrollados con POA, lo que ayuda a reducir la profundidad del árbol de herencia de una clase.

3. Número de Hijos (NOC)

NOC mide la amplitud de la jerarquía de árbol, mientras que DIT mide la profundidad. NOC cuenta la cantidad de clases hijas inmediatas de una clase que se derivan por medios de la herencia.

Un NOC alto indica un alto reuso de la clase, dado que la herencia es una forma de reuso. También puede indicar una mala abstracción en la clase padre. Si una clase tiene demasiados hijos, puede indicar un mal uso de creación de subclases. Una clase con muchos hijos también requiere más testeo.

Dado que la competencia entrecruzada se da más a lo ancho que en lo profundo de un árbol de clases, POA, al igual que en el caso anterior, puede ayudar a reducir el código sin perder su reusabilidad.

[GD,2000] Daniela Glasberg, Khaled El Emam, Walcelio Melo, Nazim Madhavji: Validating Object-Oriented Design Metrics on a CommercialJava Application. 2000.

4. Acoplamiento Entre Objetos (CBO)

Dos clases están acopladas cuando los métodos declarados en una clase utilizan métodos o variables de instancia definidas en otra clase. Los usos de relaciones pueden ser en cualquier sentido: ambas relaciones, usan y son usadas, son tomadas en cuenta, pero sólo una vez.

Múltiples accesos a la misma clase son contadas como un único acceso. Sólo las llamadas a los métodos y variables son contadas. Otro tipo de referencias, como el uso de constantes, llamadas a declaraciones de API, manejo de eventos, uso de tipos definidos por el usuario, e instanciación de objetos son ignoradas. Si una llamada a un método es polimórfico, todas las clases a las cuales la llamada puede ir son incluidas en la cuenta de acoplamiento.

Un CBO alto es indeseable. Un acoplamiento excesivo entre objetos es perjudicial para el diseño modular y evita el reuso. Cuanto más independiente es una clase, es más fácil usarla en otra aplicación. En orden para mejorar la modularidad y promover el encapsulamiento, el acoplamiento ente objetos se debe dar lo menos posible. Cuanto más grande sea el nivel de acoplamiento, más grande va a ser la sensibilidad a los cambios en otras partes del diseño, por lo que el mantenimiento resulta más dificultoso.

Un CBO alto es propenso a fallos, por lo que es necesario realizar pruebas rigurosas. Según [HS,2003] un CBO mayor a 14 es demasiado alto.

En el caso de POA, la presencia de aspectos disminuye el acoplamiento entre clases, aunque aumenta el acoplamiento entre clases y aspectos. Esto se debe a que los aspectos son nuevas entidades de las cuales las clases dependen.

5. Respuesta por Clase (RPC)

RPC es el número de métodos locales de una clase más el número de métodos llamados por dichos métodos locales.

RFC incrementa ante la presencia de aspectos. Esto se debe a que el número de entidades con las que se comunica una clase se incrementa, y las clases tienen que comunicarse con los aspectos. El punto a favor en este caso es que los aspectos pueden diseñarse de manera tal que encapsule la lógica y los objetos con los cuales la clase se comunica en forma modular.

6. Métrica de falta de cohesión (LCOM)

La sexta métrica en C&K es la falta de cohesión de métodos. Esta métrica ha recibido grandes críticas y ha originado varias alternativas.

Las métricas de cohesión miden cuan bien los métodos de una clase están relacionados unos con otros. Una clase cohesiva es la que realiza una sola función. Una clase no cohesiva es la que realiza dos o más funciones no relacionadas. Una clase cohesiva debería ser reestructurada en dos o tres clases más pequeñas.

La asunción detrás de la métrica de cohesión es que los métodos están relacionados si trabajan sobre el mismo nivel de variables de clase.

Una alta cohesión es deseable ya que promueve el encapsulamiento. Como contrapartida, una clase muy cohesiva tiene alto nivel de acoplamiento con los métodos de la clase.

Una baja cohesión indica un diseño inapropiado y complicado. Además suelen haber más errores en clases con baja cohesión.

La métrica de cohesión de C&K se calcula tomando cada para de métodos en una clase. Si acceden a distintas variables de instancias, se aumenta P en uno. Si comparte por lo menos un valor, se aumenta Q en uno.

$LCOM = P-Q$ si $P>Q$, caso contrario

$LCOM = 0$

Un valor alto en LCOM indica que la clase estaría intentando cumplir con más de un objetivo, posee más de una función.

Aplicando POA, el nivel de cohesión decae, dado la orientación a aspectos ayuda a extraer los procedimientos y variables que se utilizan para realizar otras funciones (registro de operaciones, sincronización, etc.) que no son naturales para las clases que las implementan.

2.10 Conclusiones.

En este capítulo se han introducido los conceptos básicos de la Programación Orientada a Aspectos, sus características fundamentales, la estructura general de la implementación de POA, el desarrollo en la POA y las métricas de software posibles a utilizar.

A continuación, en el Capítulo 3, se presenta el estado del arte de los lenguajes de aspectos, profundizando en algunos lenguajes.

En el Capítulo 4, se aplicarán los conceptos antes vistos en tres lenguajes de aspectos.

Capítulo 3. Lenguajes de Aspectos

Existen dos enfoques principales en el diseño de los lenguajes de aspectos: *los lenguajes de aspectos de propósito específico y los lenguajes de aspectos de propósito general*.

3.1 Lenguajes de Propósito Específico

Esta clase de lenguajes[SM,2005] son diseñados para trabajar sobre determinado tipo de aspectos pero no pueden tratar con aquellos aspectos para los que no fueron diseñados. Normalmente tienen un nivel de abstracción mayor que el del lenguaje base e imponen restricciones en la utilización del lenguaje base para evitar que los aspectos se programen en ambos lenguajes, lo que podría dar lugar a conflictos.

Estos lenguajes normalmente imponen restricciones en la utilización del lenguaje base. Esto se hace para garantizar que los conceptos del dominio del aspecto se programen utilizando el lenguaje diseñado para este fin y evitar así interferencias entre ambos. Se quiere evitar que los aspectos se programen en ambos lenguajes lo cual podría conducir a un conflicto.

Algunos lenguajes orientados a aspectos de dominio específico son:

- AML[JL,1997] es una extensión de Matlab para procesamiento de matrices dispersas.
- RG[AM,1997] que es un lenguaje para el procesamiento de imágenes.
- COOL[CL,1997] es un lenguaje para sincronización de hilos concurrentes.
- RIDL (Remote Interaction and Data transfers Language) es un lenguaje para transferencia de datos remotos.

En esta tesis, no trataremos los lenguajes de aspectos de Propósito Específico.

[SM,2005] Salvador Manzanares Guillén, *"Programación Orientada a Aspectos, una experiencia práctica con AspectJ"*, Facultad de Informática de la Universidad de Murcia, Junio 2005.
[JL,1997] J. Irwin, J.-M.Loingtier, J. R. Gilbert, and G. Kiczales, *"Aspect-oriented programming of sparse matrix code"*, Lecture Notes in Computer Science, 1997.
[AM,1997] Mendhekar A., Kiczales G. and Lamping J, *"RG: A Case-Study for Aspect-Oriented Programming"*, Xerox PARC, 1997.
[CL,1997] Cristina Lopes, *"A Language Framework for DistributedProgramming"*, Northeastern University, Noviembre 1997.

3.2 Lenguajes de Propósitos Generales

Los lenguajes de aspectos de propósito general pueden ser utilizados con cualquier clase de aspecto, no solamente con aspectos específicos (distribución, coordinación, manejo de errores, etc.), por lo que no pueden imponer restricciones en el lenguaje base. Soportan la definición separada de los aspectos proporcionando unidades de aspectos. Por lo general tienen el mismo nivel de abstracción que el lenguaje base y también el mismo conjunto de instrucciones, ya que debería ser posible expresar cualquier código en las unidades de aspectos. Por ejemplo AspectJ, que utiliza Java como base, y las instrucciones de los aspectos también se escriben en Java.

Se puede llegar a decir que los lenguajes de aspectos de propósito general no pueden cubrir completamente las necesidades, ya que permiten la separación del código, pero no garantizan la separación de conceptos, es decir, que la unidad de aspecto se utilice únicamente para programar el aspecto. La separación de conceptos es uno de los objetivos principales de la POA.

Estos lenguajes proporcionan un ambiente más adecuado para el desarrollo de aspectos, ya que es una única herramienta capaz de describir todos los aspectos que se necesiten, sin tener en cuenta la aplicación, por ejemplo al pasar de trabajar en un contexto de sincronización a uno de manejo de excepciones.

Desde el punto de vista empresarial, siempre les será más fácil a los programadores el aprender un lenguaje de propósito general, que el tener que estudiar varios lenguajes distintos de propósito específico, uno para tratar cada uno de los aspectos del sistema.

Uno de los primeros lenguajes de propósito general fue AspectJ, creado por el grupo Xerox PARC, dirigido por Gregor Kickzales. La semántica introducida por este lenguaje (puntos de enlace, puntos de corte, avisos) ha servido de base para el resto de los lenguajes de propósito general. En la Tabla 1 figuran otros[W,2008] lenguajes de programación de propósito general. En la primera columna se indican los nombres de los lenguajes, agrupados según el lenguaje base sobre el cuál se implementaron. Los lenguajes listados en esta tabla fueron analizados y en la columna Observaciones, se presentan algunos comentarios generales.

[W,2008] Wikipedia, Aspect-oriented programming, http://en.wikipedia.org/wiki/Aspect-oriented_programming, 2008, Consultado en Agosto 2008

Tabla 1: Lenguajes de Programación Orientados a Aspectos de Propósitos General

Lenguaje	Observaciones
C/C++	
AspectC	Es una simple extensión de C orientada a Aspectos, destinada a apoyar a sistemas operativos y sistemas integrados de programación. Actualmente se esta utilizando a diario en un proyecto de kernel(University of British Columbia) - http://www.cs.ubc.ca/labs/spl/projects/aspectc.html
AcpectC++	EL proyecto AspectC++ amplía el enfoque de AspectJ a C/C++, es un conjunto de extensiones del lenguaje C++ para facilitar la programación orientada a aspectos - http://www.aspectc.org/
AspectCt-oriented C	Presenta un compilador C orientado a Aspectos, que realiza la traducción de código escrito en AspectCt-oriented C a ANSI-C. Es un desarrollo de la Universidad de Toronto - http://research.msrg.utoronto.ca/ACC
FeatureC++	Es una extensión de C++ que permite programar de manera modular, el cual utiliza la POA para resolver problemas de modularidad transversal(crosscutting) - http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/
XWeaver	Es una herramienta para aplicaciones C/C++ y Java. Puede ser usado como una herramienta en la línea de comando o usando el plug-in AXDT para Eclipse http://www.xweaver.org/index.html
C# / VB.Net	
AspectDNG	Es un weaver de aspectos para multilenguajes en .Net, esta implementado en C#. http://sourceforge.net/projects/aspectdng/
Aspect#	Es un Framework de POA para el CLI de .Net y Mono, se basa en un Proxy-Dinámico y permite dentro del lenguaje la declaración y configuración de aspectos y es compatible con AopAlliance. Esta en el proyecto Castle desde el 2005 - http://www.castleproject.org/aspectsharp/
LOOM.NET	http://www.rapier-loom.net/
PostSharp	http://www.postsharp.org/
Wicca	http://www1.cs.columbia.edu/~eaddy/wicca/
Cocoa	
AspectCocoa	http://www.cocoadev.com/index.pl
Cold Fusion	
ColdSpring	Es un Framework para componentes de Cold Fusion (CFCs) que utiliza la "inversión de controles" la cual provee ventajas sobre los enfoques tradicionales de modelos de aplicación. ColdSpring presenta el primer Framework con POA para CFCs. http://www.coldspringframework.org/
Common Lisp	
AspectL	Es una librería que provee la extensión orientada a aspectos para Common Lisp/CLOS - http://common-lisp.net/project/closer/aspectl.html

Delphi	
Infra Aspect	Es un conjunto de Framework desarrollados inicialmente en Delphi para facilitar la POO, pero también toma características encontradas en desarrollos en Java y .Net, incorporando también aspectos. - http://code.google.com/p/infra/
Java	
AspectJ	http://eclipse.org/aspectj/
Azuki	Es un Framework de Java destinado a reducir el desarrollo, organización y gastos de mantenimiento de sistemas de software planteando dos etapas iniciales en su desarrollo: *Independencia en la creación de componentes y *Una aplicación de ensamble del tejedor - http://www.azuki-framework.org/
CaesarJ	Es un compilador basado en Java, presenta mejoras en cuanto a la modularidad y al desarrollo de componentes reutilizables, la herramienta esta disponible para usarla con el plugin de Eclipse - http://www.caesarj.org/
JACT	http://jac.objectweb.org/
Jastadd	http://jastadd.org/
Javassist	Es una librería de Java que permite adicionar a sus clases los métodos pertenecientes a la POA(before/after/around) Es un desarrollo del Instituto de tecnología de Tokio - http://www.csg.is.titech.ac.jp/~chiba/javassist/
Jboss AOP	http://www.jboss.org/jbossaop/
LogicAJ	http://roots.iai.uni-bonn.de/research/logicaj/
Object Teams	http://www.objectteams.org/
Prose	Son extensiones de servicios programables que permite a un programa en Java modificarse en tiempo de ejecución, teniendo características técnicas de la POA - http://prose.ethz.ch/
GluonJ	http://www.csg.is.titech.ac.jp/projects/gluonj/
Steamloom	http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp
JavaScript	
Ajaxpect	Es una librería de POA para AJAX con Java Script. El proyecto se inicio en el 2003 y recibió modificaciones- http://code.google.com/p/ajaxpect/
jQuery AOP	Es un pequeño plugin que adhiere POA a jQuery, contiene los advice de POA para aplicar a cualquier objeto, permite definir intersecciones (pointcut). http://plugins.jquery.com/project/AOP
Aspectes	Es un Framework que aplica POA a JavaScript v1.5, esta estandarizado por EcmaScript Specification 262 http://aspectes.tigris.org/
AspectJS	http://www.dodeca.co.uk/aspectjs/
Lua	
AspectLua	Es una extensión de Lua con POA, este toma conceptos de

	AspectJ, permite la creación de aspectos para permitir la modularización transversal en objetos escritos puramente en Lua - http://luaforge.net/projects/aspectlua/
Perl	
The Aspect Module	Es un módulo de PERL que sigue la terminología del proyecto AspectJ http://search.cpan.org/~eilara/Aspect-0.12/lib/Aspect.pm
PHP	
PHPAspect	Es una extensión de PHP que implementa POA, se puede utilizar a partir de la versión PHP5. El compilador de PHPAspects teje los aspectos implementado el crosscutting de conceptos dentro del código fuente. El proceso de tejido es estático (antes de la ejecución del código fuente) y sobre la base del análisis de Lex&Yacc para generar XML parse trees. XSLT se utiliza para transformar el código fuente en los árboles - http://phpaspect.org/
Aspect-Oriented PHP	Es una extensión para PHP que permite el uso de POA en middleware. Es una herramienta muy poderosa cuando se utiliza correctamente en desarrollos Web. Fue desarrollada en 2005 y continúa en desarrollo. - http://www.aophp.net/
AOP API for PHP	http://www.phpclasses.org/browse/package/2633.html
Transparent AOP PHP	Es un paquete para descargar que permite implementar POA de forma transparente. http://www.phpclasses.org/browse/package/3215.html
Python	
Lightweight Python AOP	Es una librería `aspects.py`, proporciona los medios para interceptar llamadas a funciones, la librería permite la aplicación de advices (wraps) para llamar a los puntos de unión de métodos y funciones. Está disponible a partir de Python 2.1 (incluida Jython)- http://www.cs.tut.fi/~ask/aspects/aspects.html
Logilab's aspect module	Es un módulo de Python que soporta POA, por ahora solo contiene un set de aspectos listos para usar, permite crear de manera fácil aspectos, estas posibilidades todavía son limitadas pero están en constante desarrollo. http://www.logilab.org/2738
Ruby	
AspectR	http://aspectr.sourceforge.net/
Aquarium	Es un Framework que implementa POA en Ruby, cuyos objetivos son: 1) Un poderoso lenguaje para especificar dónde aplicar los aspectos, 2) Gestión de los aspectos concurrentes, 3) Añadir y eliminar los aspectos dinámicamente, 4) fácil de usar DSL, 5) Asesorar a las clases de Java con JRuby. http://rubyforge.org/projects/aquarium/
Squeak Samalltalk	
Metalclass Talk	Es una metaclase de Smalltalk que permite programar con protocolos de meta-objetos para controlar la estructura de los objetos. - http://csl.ensm-douai.fr/MetaclassTalk
XML	
AspectXML	http://www.aspectxml.org/

A partir de este análisis, se decidió realizar pequeñas aplicaciones en los lenguajes AspectJ, AspectC y PHPAspect, para discutir con mayor profundidad las dificultades, ventajas y el estado de evolución de cada lenguaje en la implementación de la programación orientada a aspectos. La elección de estos lenguajes se debió a la gran popularidad que poseen sus lenguajes base[1], Java, C/C++ y PHP; lo que creemos que facilitaría la adopción del paradigma de POA.

A continuación se describirán los lenguajes seleccionados para evaluar su grado de adopción del paradigma de POA.

3.2.1 AspectJ

AspectJ es un lenguaje orientado a aspectos de propósito general, cuya primera versión fue lanzada en 1998 por el equipo conformado por Gregor Kickzales, Ron Bodkin, Bill Griswold, Erik Hilsdale, Jim Hugunin, Wes Isberg y Mik Kersten; creado en los laboratorios de XEROX Parc. En el año 2002 el proyecto fue integrado a la fundación Eclipse y desde entonces se le ha dado soporte dentro del entorno de desarrollo Eclipse. La versión actual de AspectJ es la 1.6.1, con fecha de lanzamiento 3 de Julio de 2008[1].

AspectJ consiste en una extensión compatible de Java para facilitar el uso de aspectos, por lo que cualquier programa válido para Java, lo será también para AspectJ. Implementar AspectJ como una extensión de Java hace que el lenguaje sea fácil de aprender, puesto que sólo añade unos pocos términos nuevos al lenguaje base (Java). Además no pierde las características ventajosas de Java, como la independencia de la plataforma, que han hecho de este lenguaje orientado a objetos, el más usado en la actualidad. De la misma forma que Java, se trata de un lenguaje multipropósito, es decir, no está ligado a un tipo de dominio en particular.

Por compatible se entiende:

- **Compatibilidad base**: todos los programas válidos de Java deben ser programas válidos de AspectJ.
- **Compatibilidad de plataforma**: todos los programas válidos de AspectJ deben correr sobre la máquina virtual estándar de Java, sin ninguna modificación adicional.
- **Compatibilidad de programación**: la programación en AspectJ debe ser una extensión natural de la programación en Java.

1 En el Anexo 1 se encuentra la calificación de los lenguajes de programación más populares según la encuesta de TIOBE.

- **Compatibilidad para el programador**: para el programador, AspectJ es una extensión natural del lenguaje Java.

AspectJ extiende Java para soportar el manejo de aspectos agregando a la semántica de Java las siguientes entidades:

- **Puntos de unión o enlace (join points)**: son puntos bien definidos en la ejecución de un programa, entre ellos podemos citar llamadas a métodos y accesos a atributos. Representan los lugares donde los aspectos añaden su comportamiento.

- **Intersección o punto de corte (pointcuts)**: agrupan *join points* y permiten exponer el contexto en ejecución de dichos puntos. Existen *pointcuts* primitivos y también definidos por el usuario. Por ejemplo, mediante un punto de corte podemos agrupar las llamadas a todos los métodos de cierta clase y exponer su contexto (argumentos, objeto invocador, objeto receptor).

- **Guías/Consejos/Orientaciones (advices)**: especifican el código que se ejecutará en los *join points* que satisfacen cierto *pointcut*, pudiendo acceder al contexto de dichos *join points*.

Por lo tanto los *pointcuts* indican **dónde** y los *advices* **qué** hacer. Un ejemplo de *crosscutting* dinámico podría ser añadir cierto comportamiento tras la ejecución de ciertos métodos o sustituir la ejecución normal de un método por una ejecución alternativa.

- **Declaraciones inter-tipo (inter-type declarations)**: permiten añadir miembros (campos, métodos o constructores) a clases, interfaces o aspectos de una aplicación.

- **Declaraciones de parentesco (declare parents)**: permiten especificar que ciertas clases implementan una interfaz o extienden nuevas clases.

- **Declaraciones en tiempo de compilación (compile-time declarations)**: permiten añadir advertencias o errores en tiempo de compilación para notificar ciertas situaciones que deseamos advertir o evitar, por ejemplo la utilización de cierto método en desuso.

- **Declaraciones de precedencia (declare precedence)**: permiten especificar relaciones de precedencia entre aspectos.

- **Excepciones suavizadas (softening exceptions)**: permiten ignorar el sistema de chequeo

1 En el Anexo 2 se encuentra una tabla con las fechas de lanzamiento de las distintas versiones de AspectJ.

de excepciones de Java, silenciando las excepciones que tienen lugar en determinados puntos de enlace, encapsulándolas en excepciones no comprobadas y relanzándolas.

En AspectJ, para expresar la funcionalidad de los aspectos se utiliza Java estándar, mientras que para especificar las reglas de entretejido (*weaving rules*) el lenguaje proporciona una serie de constructores.

La implementación de las reglas de entretejido se suele llamar *crosscutting concerns* o competencia transversal, puesto que especifican la forma en que los aspectos "atraviesan" las clases de la aplicación principal. AspectJ soporta tanto la implementación transversal estática como dinámica de competencias transversales:

- **Implementación transversal dinámica**: permite definir comportamiento adicional que se ejecutará en puntos específicos dentro del programa, modificando o añadiendo un nuevo comportamiento. Es el tipo de entrecruzado más usado en AspectJ. Los tipos de constructores dinámicos son los *join points*, *pointcuts* y *advices*.

- **Implementación transversal estática**: permite modificar la estructura estática del programa, agregando superclases o implementaciones de interfaces. La función principal del crosscutting estático es dar soporte a la implementación del crosscutting dinámico. Los constructores son las: i*nter-type declarations, declare parents, compile-time declarations, declare precedence* y las *softening exceptions*.

Todos los constructores citados anteriormente (puntos de corte, avisos, y declaraciones) se encapsulan en una entidad llamada *aspect* o aspecto. Un aspecto es la unidad básica de AspectJ, al igual que una clase lo es de un lenguaje orientado a objetos. Un aspecto también puede contener atributos, métodos y clases anidadas como una clase Java normal.

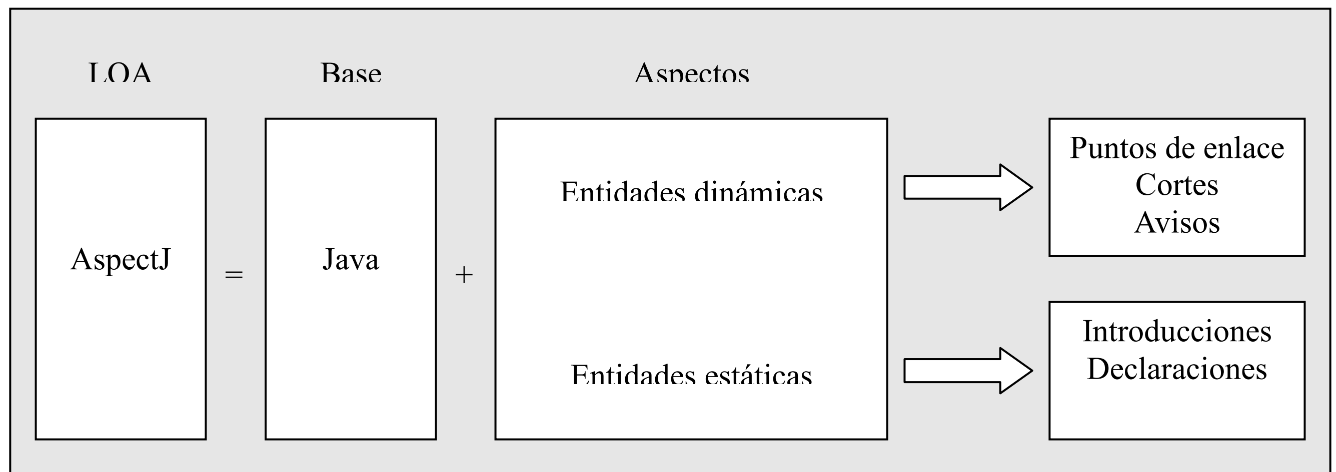


Figura 9: Composición de AspectJ

Modelo transversal dinámico

Un elemento crítico en el diseño de un lenguaje orientado a aspectos es el modelo de puntos de unión. El modelo de puntos de unión provee un marco común de referencia que hace posible definir la estructura dinámica de la competencia transversal.

Punto de Unión (join point)

Los puntos de enlace son puntos bien definidos en la ejecución de un programa, entre ellos podemos citar llamadas a métodos y accesos a atributos. No hay que confundirlos con posiciones en el código de un programa sino que son puntos sobre el flujo de ejecución de un programa. Representan los lugares donde los aspectos añaden su comportamiento.

AspectJ soporta los siguientes tipos de puntos de enlace:

- **Llamada a un método**: un punto de enlace llamada a método ocurre cuando un método es invocado por un objeto. En el caso de los métodos estáticos no existe objeto invocador.
- **Llamada a un constructor**: un punto de enlace llamada a constructor ocurre cuando un constructor es invocado durante la creación de un nuevo objeto.
- **Ejecución de un método**: un punto de enlace ejecución de método engloba la ejecución del cuerpo de un método.
- **Ejecución de un constructor**: un punto de enlace ejecución de constructor abarca la ejecución del cuerpo de un constructor durante la creación de un nuevo objeto.
- **Lectura de un atributo**: un punto de enlace lectura de atributo ocurre cuando se lee un

atributo de un objeto dentro de una expresión.

- **Escritura de atributo**: un punto de enlace escritura de atributo ocurre cuando se asigna un valor a un atributo de un objeto.

- **Ejecución de un manejador de excepciones**: un punto de enlace ejecución de manejador ocurre cuando un manejador es ejecutado.

- **Iniciación de clase**: un punto de enlace iniciación de clase ocurre cuando se ejecuta el inicializador estático de una clase específica, en el momento en que el cargador de clases carga dicha clase.

- **Iniciación de objeto**: un punto de enlace iniciación de objeto ocurre cuando un objeto es creado. Comprende desde el retorno del constructor padre hasta el final del primer constructor llamado.

- **Iniciación de un objeto**: un punto de enlace pre-iniciación de un objeto ocurre antes de que el código de iniciación para una clase particular se ejecute. Comprende desde el primer constructor llamado hasta el comienzo del constructor padre. Es raramente usado y suele abarcar las instrucciones que representan los argumentos del constructor padre.

- **Ejecución de un aviso**: un punto de enlace ejecución de aviso, comprende la ejecución del cuerpo de un aviso.

Punto de Corte (pointcut)

Los puntos de corte identifican o capturan una serie de puntos de unión y permiten exponer su contexto a los avisos (*advice*). Forman parte de la definición de un aspecto, permitiéndonos especificar los puntos de enlace donde se aplicará dicho aspecto.

Su estructura que no se asemeja a ninguno de los miembros tradicionales de una clase. Básicamente, se trata de un identificador seguido de parámetros y la definición de una expresión que indicará cuales son los puntos de enlace a capturar.

Usando las construcciones ofrecidas por el lenguaje, podremos crear puntos de corte desde muy generales, que identifican a un gran número de puntos de enlace, a muy específicos, que identifican a un único punto de enlace.

Existen puntos de corte anónimos y con nombre. Un punto de corte anónimo se define en

el lugar donde se usa (en un aviso o en otro punto de corte) y no puede ser referenciado en otras partes del código, a diferencia de los puntos de corte con nombre que si se pueden reutilizar.

Se pueden definir puntos de corte abstractos (`abstract`) dentro de un aspecto, de forma que sean sus aspectos derivados los encargados de implementarlos. Su definición es similar a la de un método abstracto sólo declara la signatura del punto de corte, comprendida por el identificador y sus parámetros. Al igual que en un método abstracto, su objetivo es declarar la existencia del elemento para que pueda ser referenciado dentro del aspecto, dejando abierta la posibilidad de definirlo en aspectos derivados.

Si bien los puntos de corte se pueden sobrescribir en aspectos derivados, al igual que un método, no existe la noción de sobrecarga (*overload*). Eso quiere decir que no se puede declarar más de un punto de corte con el mismo identificador en un mismo aspecto o aspectos derivados, que posean una lista de parámetros distinta, ya sea en cantidad o en tipos.

Tanto en los puntos de corte anónimos como en los puntos de corte con nombre, se pueden combinar las expresiones que identifican los puntos de enlace mediante los operadores lógicos `!` (*not*), `||` (*or*) y `&&` (*and*) . Dichas expresiones están formadas principalmente por elementos llamados puntos de corte primitivos. Hay varios tipos de puntos de corte primitivos y cada uno define un conjunto específico de puntos de unión prefijado dentro del universo de los puntos de unión posibles en un programa.

Puntos de corte primitivos

AspectJ proporciona una serie de descriptores de puntos de corte que nos permiten identificar grupos de puntos de enlace que cumplen diferentes criterios. Estos descriptores se clasifican en diferentes grupos:

- **Basados en las categorías de puntos de unión**: capturan los puntos de enlace según la categoría a la que pertenecen: llamada a método (`call`), ejecución de método (`execute`), lectura de atributo (`get`), etc.
- **Basados en el flujo de control**: capturan puntos de enlace de cualquier categoría siempre y cuando ocurran en el contexto de otro punto de corte. Estos descriptores son `cflow` y `cflowbelow`.
- **Basados en la localización de código**: capturan puntos de enlace de cualquier categoría que se localizan en ciertos fragmentos de código, por ejemplo, dentro de una clase o

dentro del cuerpo de un método. Estos descriptores son `within` y `withincode`.

- **Basados en los objetos en tiempo de ejecución**: capturan los puntos de enlace cuyo objeto actual (`this`) u objeto destino (`target`) son de un cierto tipo. Además de capturar los puntos de enlace asociados con los objetos referenciados, permite exponer el contexto de los puntos de enlace.

- **Basados en los argumentos del punto de enlace**: capturan los puntos de enlace cuyos argumentos son de un cierto tipo mediante el descriptor args. También puede ser usados para exponer el contexto.

- **Basados en condiciones**: capturan puntos de enlace basándose en alguna condición usando el descriptor if (expresionBooleana).

Aviso (Advice)

Los puntos de corte sólo capturan puntos de enlace. Para implementar un comportamiento de corte transversal (*crosscutting behaivor*), se utilizan Avisos. Un aviso une a un punto de corte (que captura puntos de enlace) con un bloque de código a ejecutar (en cada uno de los correspondientes puntos de enlace).

Modelo Estático de Puntos de Unión

Declaraciones inter-tipos (Inter-type declarations)

Las declaraciones inter-tipos en AspectJ son declaraciones que cortan transversalmente las clases y sus herencias. Pueden declarar miembros que cortan transversalmente a múltiples clases, o cambiar la relación de herencia entre clases. A diferencia de los Avisos, que funcionan de manera dinámica, las declaraciones entre-tipos funcionan de manera estática, en tiempo de compilación.

Aspectos

Los aspectos integran a los puntos de corte, avisos y declaraciones inter-tipo en una unidad modular de implementación transversal. Su definición es muy similar a la de una clase y también pueden ser instanciados, pero AspectJ controla la forma en la cual sucede, por lo que no se puede instanciar un aspecto de forma explícita. Por defecto, cada aspecto es un *singleton*, por lo que una instancia del aspecto siempre es creada.

Los aspectos, al igual que una clase, pueden implementar interfaces y/o extender otra clase. Además, pueden extender de otros aspectos, manteniendo la misma semántica que en el caso de las clases, es decir, se heredan los elementos públicos y protegidos del aspecto o clase base. Como restricciones, los aspectos no pueden implementar ni la interfaz "Serializable" ni la interfaz "Clonable". La semántica de estas interfaces no tiene sentido para el caso de un aspecto. Si bien un aspecto puede extender tanto de una clase como de un aspecto, lo contrario no es cierto en AspectJ, una clase sólo puede extender otra clase, ya que los aspectos tienen un grupo de elementos que incluyen a los de una clase. Es decir, un aspecto es una clase con elementos adicionales.

Por último, un aspecto debe poseer un cuerpo en donde se definen los elementos integrantes del aspecto. Estos elementos pueden ser, al igual que en una clase: atributos, métodos, clases internas o interfaces internas, teniendo en cuenta que un aspecto no puede definir constructores. En el cuerpo de un aspecto también se pueden definir los siguientes elementos: punto de corte, aviso, declaraciones inter-tipo y otras declaraciones.

Un ejemplo: La gestión de una cola circular[LD,1996]

Lea D, en su libro "Concurrent Programming in Java: design principles and patterns" expone el ejemplo de la gestión de una cola circular en Java y luego utilizando AspectJ:

En primer lugar, se implementará la cola circular en Java sin el aspecto de sincronización. Luego, se le añadirán las líneas de código necesarias para gestionar este aspecto, y se podrá apreciar claramente cómo estas se diseminan por todo el programa, quedando un código poco claro. Después el mismo ejemplo se implementará con Aspect J.

A continuación se muestra el código Java de una lista circular, en principio, sin restricciones de sincronización. La clase `ColaCircular` se representa con un `array` de elementos. Los elementos se van añadiendo en la posición `ptrCola` (por atrás), y se van borrando en la posición apuntada por `ptrCabeza` (la parte de delante). Estos dos índices se ponen a cero al alcanzar la capacidad máxima del `array`. Esta iniciación se consigue utilizando el resto obtenido al dividir la posición a tratar entre el número total de elementos del `array`. Esta clase solamente contiene un constructor y dos métodos: `Insertar` para introducir elementos en la cola, y `Extraer`, para sacarlos.

[LD,1996] Lea D, *Concurrent Programming in Java: design principles and patterns*, Addison-Wesley, 1996.

```
public class ColaCircular
 {
 private Object[] array;
 private int ptrCola = 0, ptrCabeza = 0;
 private int eltosRellenos = 0;
    public ColaCircular (int capacidad)
     {
           array = new Object [capacidad];
     }
    public void Insertar (Object o)
     {
           array[ptrCola] = o;
           ptrCola = (ptrCola + 1) % array.length;
           eltosRellenos++;
     }
    public Object Extraer ()
     {
      Object obj = array[ptrCabeza];
           array[ptrCabeza] = null;
           ptrCabeza = (ptrCabeza + 1) % array.length;
           eltosRellenos--;
       return obj;
     }
 }
```

El problema del código presentado es que si varios hilos solicitan ejecutar métodos del objeto `ColaCircular`, éstos no están sincronizados, pudiéndose dar el caso de que la cola haya cubierto su capacidad, y se sigan haciendo peticiones de inserción de elementos, y al contrario, es decir, que la cola esté vacía y se hagan peticiones de extracción de elementos. En estos casos, el hilo que hace la solicitud debería esperar a que se extraiga algún elemento, en el primer caso, o a que se inserte alguno en el segundo. Es decir, que se necesita código adicional para eliminar estas inconsistencias.

Java permite coordinar las acciones de hilos de ejecución utilizando *métodos e instrucciones sincronizadas*. A los objetos a los que se debe coordinar el acceso se le incluyen métodos sincronizados. Estos métodos se declaran con la palabra reservada `synchronized`.

Un objeto solamente puede invocar a un método sincronizado al mismo tiempo, lo que impide que los métodos sincronizados en hilos de ejecución entren en conflicto.

Todas las clases y objetos tienen asociados un *monitor*, que se utiliza para controlar la forma en que se permite que los métodos sincronizados accedan a la clase u objeto. Cuando se invoca un método sincronizado para un objeto determinado, no puede invocarse ningún otro método de forma automática. Un monitor se libera automáticamente cuando el método finaliza su ejecución y regresa. También se puede liberar cuando un método sincronizado ejecuta ciertos métodos, como `wait()`. El subproceso asociado con el método sincronizado se convierte en no ejecutable hasta que se satisface la situación de espera y ningún otro método ha adquirido el

monitor del objeto.

A continuación se presenta la versión sincronizada del ejemplo anterior, en donde se incluye la exclusión mutua y condiciones con guardas.

```
public class ColaCircular
 {
 private Object[] array;
 private int ptrCola = 0, ptrCabeza = 0;
 private int eltosRellenos = 0;
    public ColaCircular (int capacidad)
     {
          array = new Object [capacidad];
     }
    public synchronized void Insertar (Object o) {
       while (eltosRellenos == array.length){
            try {
               wait ();
            }catch (InterruptedException e) {}
       }
       array[ptrCola] = o;
       ptrCola = (ptrCola + 1) % array.length;
       eltosRellenos++;
       notifyAll();
     }
    public synchronized Object Extraer () {
     while (eltosRellenos == 0){
           try {
              wait ();
           }catch (InterruptedException e) {}
     }
     Object obj = array[ptrCabeza];
     array[ptrCabeza] = null;
     ptrCabeza = (ptrCabeza + 1) % array.length;
     eltosRellenos--;
     notifyAll();
     return   obj;
     }
 }
```

Como se puede observar los métodos `Insertar` y `Extraer` se han regado con el código correspondiente a la sincronización de los hilos. Además el código añadido se esparce por todo el método, no está localizado en una sola parte del código.

A continuación se presenta el código del aspecto que se encargaría de la sincronización de la clase `ColaCircular`:

```
aspect ColaCirSincro{
 private int eltosRellenos = 0;

 pointcut insertar(ColaCircular c):
    instanceof (c) && receptions(void Insertar(Object));

 pointcut extraer(ColaCircular c):
    instanceof (c) && receptions (Object Extraer());

 before(ColaCircular c):insertar(c) {
          antesInsertar(c);}
```

```
protected synchronized void antesInsertar (ColaCircular c){
  while (eltosRellenos == c.getCapacidad()) {
    try { wait(); } catch (InterruptedException ex) {};
  }
}

after(ColaCircular c):insertar(c) { despuesInsertar();}

protected synchronized void despuesInsertar (){
     eltosRellenos++;
     notifyAll();
}

before(ColaCircular c):extraer(c) {antesExtraer();}

protected synchronized void antesExtraer (){
  while (eltosRellenos == 0) {
    try { wait(); } catch (InterruptedException ex) {};
  }
after(ColaCircular c):extraer(c) {
                despuesExtraer();}

protected synchronized void despuesExtraer (){
    eltosRellenos--;
     notifyAll();
}
```

El código de la clase asociada al aspecto anterior es el siguiente:

```
public class ColaCircular
 {
 private Object[] array;

 private int ptrCola = 0, ptrCabeza = 0;

    public ColaCircular (int capacidad)
     {
           array = new Object [capacidad];
     }

    public void Insertar (Object o)
     {
           array[ptrCola] = o;
           ptrCola = (ptrCola + 1) % array.length;
     }

    public Object Extraer ()
     {
      Object obj = array[ptrCabeza];
           array[ptrCabeza] = null;
           ptrCabeza = (ptrCabeza + 1) % array.length;
       return obj;
     }

   public int getCapacidad(){
       return capacidad;
     }
 }
```

La idea que se ha seguido es bastante simple. Se crean dos cortes, uno para capturar el caso de que se produzca una llamada al método `Insertar`, y otro para el método `Extraer`.

Con cada corte se definen dos avisos, uno de tipo `before` y otro de tipo `after`. En el de tipo `before` se comprueba si se dan las condiciones necesarias para que se ejecute el método. En el caso de la inserción, la comprobación de que el buffer no esté lleno, y en el caso de la extracción que no esté vacío. Si alguna de estas condiciones no se cumple, el hilo que ejecuta el objeto se pone en espera hasta que se cumpla la condición.

En los avisos de tipo `after` se incrementa o decrementa el número de elementos del buffer, dependiendo si estamos en la inserción o la extracción, y se indica a los demás hilos que pueden proseguir. Al dispararse el evento asociado a los cortes se ejecuta el código definido en el bloque asociado al aviso.

3.2.2 AspeCt C[SR,2008]

AspeCt C es un simple lenguaje de aspectos de propósito general que extiende a C, es un subconjunto de AspectJ sin ningún soporte para la programación orientada a objetos o módulos explícitos.

El código de aspectos, conocido como aviso, interactúa con la funcionalidad básica en los límites de una llamada a una función, y puede ejecutarse antes, después, o durante dicha llamada. Los elementos centrales del lenguaje tienen como objeto señalar llamadas de funciones particulares, acceder a los parámetros de dichas llamadas, y adherir avisos a ellas.

Estructura General de programas orientados a aspectos en C

Una estructura general de programas orientados a aspectos en C consta de dos partes, comúnmente conocida como el *núcleo* o *base* de programa y el *aspecto* del programa.

El programa principal está escrito en C clásico y el programa orientado a aspectos, en el lenguaje orientado a aspectos C.

La clave para distinguir al programa núcleo o base y al orientado a aspectos reside en la utilización de el sufijo `.mc` para el programa principal y el sufijo `.ac` para el programa orientado a aspectos.

El compilador de la orientación a aspectos C está diseñado para la conversión código a código. El compilador de la programación orientada a aspectos lee código C como entrada, procesa los datos y produce los datos en C como salida.

[SR,2008] Aspect Oriented C - http://research.msrg.utoronto.ca/ACC/WebHome, Consultado Agosto 2008.

El paradigma orientado a aspectos en C espera `.mc` y `.ac` y éste genera archivos ANSI-C compatibles con archivos de C.

Estos archivos generados pueden ser compilados por cualquier compilador C como por ejemplo gcc, cc. Para generar archivos objetos y su ejecutable, existe el compilador xlc.

Ejemplo 1: El aspecto orientado a C "Hola Mundo" Programa

El "Hola Mundo" en la orientación a aspectos en C se basa en el siguiente programa principal (hello.mc):

```
Int main() {
   printf("Hello ");
}
```

y el aspecto del programa (world.ac) es:

```
after(): execution(int main()) {
    printf(" World from !AspeCt-Oriented C ! \n");
}
```

Después de compilar y correr el ejecutable, la salida es:

```
Hello World from !AspeCt-oriented C !
```

El programa principal imprime la primera parte del mensaje. El programa orientado a aspectos define un *advice*, que especifica:

- **Cuándo ejecutar**: after de la llamada a la función, cuyo prototipo es "int main ()"
- **Qué ejecutar**: imprimir la segunda parte del mensaje

El compilador de aspectos C procesa el *advice declaration* (declaración de aviso) el archivo de aspectos y del programa básico desde al programa básico y genera código en C. El mismo contiene información de los dos archivos.

Este momento del proceso se refiere a la compilación de aspectos. Este es el aviso específico en el archivo de aspecto que es tejido dentro del programa básico para dar como resultado un programa que refleja la intención de ambos.

Ejemplo 2: Un aspecto reutilizable para la comprobación de asignación de memoria

Es práctica común, comprobar el valor de retorno después de asignación de memoria para garantizar que el valor de retorno no sea nulo.

Este código se ve a menudo como sigue:

```
x = (int *)malloc(sizeof(int) * 4); <--- dynamic memory allocation
    if (x == NULL) {
    // rountina para manejar el caso cuando no hay asignación de
   // memoria
   }
  // rountina para el manejo de caso normal
```

Este control debe llevarse a cabo después de cada llamada a `malloc()`. La llamada a `malloc()` es un medio muy utilizado para la asignación dinámica de memoria. El fragmento anterior de código está casi idénticamente disperso en todo el sistema.

Por otra parte, se aplican controles similares para otras llamadas de asignaciones de memoria, como `calloc()` y `realloc()`. Este es un ejemplo de un aspecto.

La implementación concerniente a este chequeo de asignación de memoria está dispersa en todo el programa.

Mientras que esto es un importante chequeo que definitivamente debe llevarse a cabo sin embargo es necesario aclarar que el código innecesario no debe distraernos de la lógica del programa principal. Esto hace que sea difícil concentrarse en la esencia del programa y entender la lógica. Por otra parte, la actualización del código concerniente al chequeo significaría que muchas líneas de código estarían dispersas por todo el programa básico podrían tener que cambiar.

Tales cambios generalizados son sin duda posibles pero resultan tediosos llevarlos a cabo manualmente.

Las técnicas convencionales de descomposición (por ejemplo, orientación a objetos y la programación imperativa) no pueden ser consideradas de forma aisladas ya que afectan transversalmente a todo el sistema.

Solución con POA

El chequeo de la asignación de memoria es un aspecto típico. Su función es complementaria a la lógica central del programa y su uso atraviesa todo el sistema (*crosscuts*).

Para una mejor modularización del sistema, un mejor mantenimiento, un aumento en la legibilidad de código, los diseñadores de sistemas podrán decidir aislar el foco de preocupación. Esta decisión tiene que ser reflejada cuidadosamente. En este caso, por ejemplo, el control es totalmente necesario. A efectos ilustrativos, se muestra la forma de representar el chequeo de la

asignación de memoria con aspecto orientado a C (AspeCt-oriented C). El chequeo lógico sería vería reflejado en un el siguiente archivo de aspecto, como sigue:

```
after(void * s):
   (call($ malloc(...)) || call($ calloc(...)) || call($realloc(...)))&&
result(s) {
           char * result = (char *)(s);
           if (result == NULL) {
             // rutina para manejar el caso en que la locación de
             // memoria falla
           }
         }
```

Ahora, el núcleo del programa es el siguiente:

```
  ...
  int *x ;
  x = (int *)malloc(sizeof(int) * 4);   <--- dynamic memory allocation
     /* routine for handling the normal case */
     ...
```

El advice incorporan las siguientes informaciones:

- **Cuándo ejecutar**: (*after*) después de cada llamada a funciones cuyo nombre sea "`malloc`", "`calloc`", o "`realloc`" y cuyo retorno es de tipo "`void *`"

- **Qué ejecutar**: el chequeo del valor de "`s`" . Si es nulo, el error es invocado. El valor de "`s`" se verifica a través del contexto expuesto en función de la programación orientada al aspecto C, quien pasa el valor de retorno de la función que llama a "`s`"

Beneficios

- **La programación en aspectos es reutilizable:** Se puede aplicar a cualquier programa en C usando la memoria por encima de la asignación y el mecanismo de comprobación.

- **El sistema es más fácil de entender**: El programa principal y la comprobación de la asignación de memoria no se enreda más.

- **La asignación total de memoria se modulariza en un único archivo**: Esto le da a los programadores una visión de cómo trabaja.

- **El sistema es más flexible**: La comprobación de la asignación de memoria se puede excluir de modo simple en la etapa de la compilación de aspectos. Tenga en cuenta, en este caso particular, que es poco probable que un diseñador opte por excluir el control, sino más bien utilizar el Aspecto para sustituir diferentes controles, seguimientos o la depuración de la lógica.

- **El código es menos propenso a errores**: El código reside en un archivo, donde es más fácil y rápido encontrar los errores. En contraste, códigos similares suelen ser copiados y pegados lo cual es a menudo fuente de errores.

- **El código es robusto**: Una nueva dinámica de asignación de memoria es también la funcionalidad que abarca el control de aspecto, sin requerir precauciones especiales.

Ejemplo 3: Un aspecto reutilizable para perfiles memoria

Problema

Los desarrolladores están a menudo interesados en la dinámica del uso de la memoria de un programa. Del mismo modo, el número de llamadas a `malloc()`, `calloc()` o `realloc()` puede ser de interés.

Tradicionalmente, este problema se resuelva mediante la adición de macros, contra la vinculación de diferentes bibliotecas de asignación de memoria, o instrumentando el código.

Si bien estas son todas las soluciones viables, cada uno viene acompañado de un conjunto de ventajas y desventajas.

Una solución que añade más discretas llamadas de perfiles y elimina o desactiva de nuevo cuando el código entra en producción es un objetivo deseable

Solución con POA

La dificultad inherente a la memoria correctamente modularizada de perfiles es el factor que atraviesa (*crosscuts*) todo el sistema.

Memoria de perfiles es un clásico ejemplo de un aspecto en esta situación

La raíz de la aplicación de perfiles de la memoria como aspecto puede tejerse en un programa ya existente:

```
  #include <stdlib.h>
      size_t totalMemoryAllocated; <-- use global variables to account
for                                  <--  profiling information
      int totalAllocationFuncCalled;
      int totalFreeFuncCalled;
      void   initProfiler()
      { <-- AspectC file can contain regular C functions
            totalMemoryAllocated = 0;
            totalAllocationFuncCalled = 0;
            totalFreeFuncCalled = 0;
      }
```

```
      void   printProfiler() {
            printf("total       memory       allocated       =       %d
bytes\n",totalMemoryAllocated );
            printf("total memory allocation function called = %d \n",
                                          totalAllocationFuncCalled);
                  printf("total memory free function called = %d\n",
                                                totalFreeFuncCalled);
         }
      before(): execution(int main())                               <--
advice 1
      {
             initProfiler();
         }
      after(): execution(int main())                                <--
advice 2
      {
            printProfiler();
      }

      before(size_t s): call($ malloc(...)) && args(s)              <--
advice 3
      {
            totalMemoryAllocated += s;
            totalAllocationFuncCalled ++;
      }

      before(size_t n, size_t s): call($ calloc(...)) && args(n, s) <--
advice 4
      {
            totalMemoryAllocated += n * s;
            totalAllocationFuncCalled ++;
      }

      before(size_t s): call($ realloc(...)) && args(void *, s)     <--
advice 5
      {
            totalMemoryAllocated += s;
            totalAllocationFuncCalled ++;
      }

      before() : call(void free(void *))                            <--
advice 6
      {
            totalFreeFuncCalled++;
      }
```

Explicaciones

- *advices 1 y 2:* son los responsables del inicio del profiler y de los resultados de salida. Típicamente, un programa en C comienza y finaliza su vida con la ejecución de la función cuerpo principal. Por lo tanto, se teje el advice 1 y 2 en la ejecución del `main()`.
- *advices 3, 4, 5 y 6:* ellos actualización el profile de información siempre que se llama a una función de interés. Usando la característica del contexto expuesta en la programación

orientada a aspectos C, la información asignada sobre la cantidad de memoria se pasa como parámetros en el *advice* utilizando `args()` en el *pointcut* declaración.

Beneficios

Los beneficios son similares a los antes manifestados para el control con aspectos de asignación de memoria.

Se usan dos mecanismos de coincidencia

1. reglas de simple carácter (*simple character matching*)
2. reglas de carácter con comodines (*wildcard character matching*)

Matching Rules

Son las reglas que nos indican qué interceptar para aplicar los aspectos.

- **Reglas de simple carácter (simple character matching)**:

Si no hay caracteres comodines (es decir, caracteres "$", o "...") utilizados en la declaración del pointcut, la orientación a aspectos en C distingue entre mayúsculas y minúsculas para la comparación de cadenas para reglamentar contra las funciones de prototipo.

Por ejemplo:

1. El pointcut "`call (int foo(int))`" recoge cualquier llamada a una función que tenga el nombre "`foo`", se acepta un "`int`" parámetro, y devuelve un "`int`".

```
int foo (int);
```

2. El pointcut "`args (int, char)`" recoge cualquier llamada o ejecución de una función que puede tener cualquier nombre, se acepta un "`int`" y un "`char`" como parámetros, y retornando cualquier valor (incluido el "vacío", sin valor.) Entre otros, se hallan las siguientes funciones con reglas prototipos:

```
void foo(int, char);
int foo2(int, char);
char * foo3(int, char);
double x2(int, char);
```

- **Reglas de carácter con comodines (wildcard character matching)**:

Los comodines incluyen "`$`", "`...`" y "`$`" Representan una simple cadena de longitud

arbitraria, incluida la cadena vacía. "..." representan una lista única de cualquier longitud, incluyendo la lista vacía. El uso de comodines aumenta las capacidades de las reglas orientadas hacia el aspecto C.

Por ejemplo:

1. El pointcut "`call(i$t f$oo(in$))`" recoge cualquier llamada a una función con un nombre que comienza con "`f`" y termina con "`oo`", aceptando un tipo de parámetro que comienzan con "`in`" y retorna un tipo comenzando con "`i`" y termina con "`t`". Entre otros, se hallan las siguientes funciones con reglas prototipo:

```
it foo(in);
int foo(int);
int f1oo(in);
int f2oo(inxy);
```

2. El pointcut "`args (int, ..., char))`" recoge cualquier llamada o ejecución de una función de puede tener cualquier nombre, aceptando una lista de parámetros que comiencen con "`int`" y termina con "`char`", y para retornar cualquier valor (incluyendo vacío.) Entre otros, se hallan las siguientes funciones con reglas prototipos:

```
void foo(int, char);
void foo1(int, char);
int foo2(int, char *, char);
char * foo3(int, struct A * , char);
double x2(int, int, int, char , double, char);
```

Pointcut Matching

Para aplicar el código *advices* en un *join point*, el *pointcut* asociado debe coincidir con el join point. El Aspecto orientado a C lleva a cabo la adecuación de la función prototipo en el programa contra el *pointcut* que se especifican dentro del código de aspecto. Las reglas del *pointcut* se basan en esta información.

Por ejemplo, supongamos que dos declaraciones de *advice:*

```
/* advice 1 */
 before() : call (void foo1()) {...}

/* advice 2 */
before() : call (void foo2()) {...}
```

y el programa básico es el siguiente:

```
/* both foo1 and foo2 are defined in other files */
/* only foo1's prototype is given here */
```

```
void foo1();

int main() {
    foo1();
    foo2();
}
```

En este ejemplo, sólo la llamada a `foo1()` es combinada por el `advice1` y la llamada a `foo2()` no se corresponde porque su prototipo es desconocida para el aspecto orientado a compilador de C.

Usando el compilador de aspecto orientado a C: ACC

Si no hay archivos a "incluir" los archivos, utilice acc de la siguiente manera:

```
> acc hello.mc workday <-- acc generará hello.c y world.c

> gcc hello.c world.c
```

Si hay archivos a "incluir" los archivos, el archivo de entrada al acc, deben ser preprocesados. Utilice el CAC de la siguiente manera:

```
> cp hello.mc hello_temp1.c      <-- copiar el archivo original para tener el sufijo ".c", porque gcc no reconoce ".mc" y ".ac".

> cp world.ac world_temp1.c

> gcc -E hello_temp1.c > hello_temp2.mc <-- pre procesa archivos, y se guardan a ellos para ser: ".mc" y ".ac" files.

> gcc -E world_temp1.c > world_temp2.ac

> acc hello_temp2.mc world_temp2.ac <-- acc generará hello_temp2.c y world_temp2.c

> gcc hello_temp2.c world_temp2.c
```

Nota: gcc-E invoca solo el preprocesador. Es como cpp

Un típico Makefile (Crear archivo)

Un ejemplo típico involucrando a Makefile con acc es el siguiente:

```
a.out: hello.o world.o <-- a.out esta compuesto por dos
                           módulos: hello.o y world.o

  gcc hello.o world.o
```

```
hello.o: hello.mc world.ac  <-- hello.o depende de hello.mc

                                y world.ac

   acc hello.mc world.ac <-- acc genera hello.c y world.c

   gcc -c hello.c        <-- gcc genera hello.o desde hello.c

world.o: world.ac        <-- world.o depende de world.ac

   acc world.ac          <-- acc genera world.c

       gcc -c world.c       <-- gcc genera world.o from world.o
```

Verificando la semántica y depurando la información

El actual aspecto orientado a la versión 0,2 C tiene las siguientes limitaciones. Como el aspecto orientado a C madura, estas limitaciones van a desaparecer. Por consiguiente, se recomienda a la práctica cuidadosa de codificación, de doble control y triple comprobación de su código.

- Comenzando con el acc V 0.4 la semántica del control de aspecto es compatible.
- Comenzando con el acc V 0.5 se genera información deputada. Esto sirve para que el código principal puede verse depurado, cuando se produce la intensificación del código de aspecto, por ejemplo. Esto también sirve al compilador ANSI-C al señalar el número de línea en el archivo fuente original cuando se informa acerca de los errores.

Para ayudar al desarrollador a depurar se recomienda que el código fuente generado en C se pase a través de las plataformas Unix a menudo disponible en el guión programa, que imprime bien la entrada del archivo fuente en C. Esto ayuda a comprender la lógica de aspectos weaver y a identificar los errores.

El aspecto orientado a compilador de C

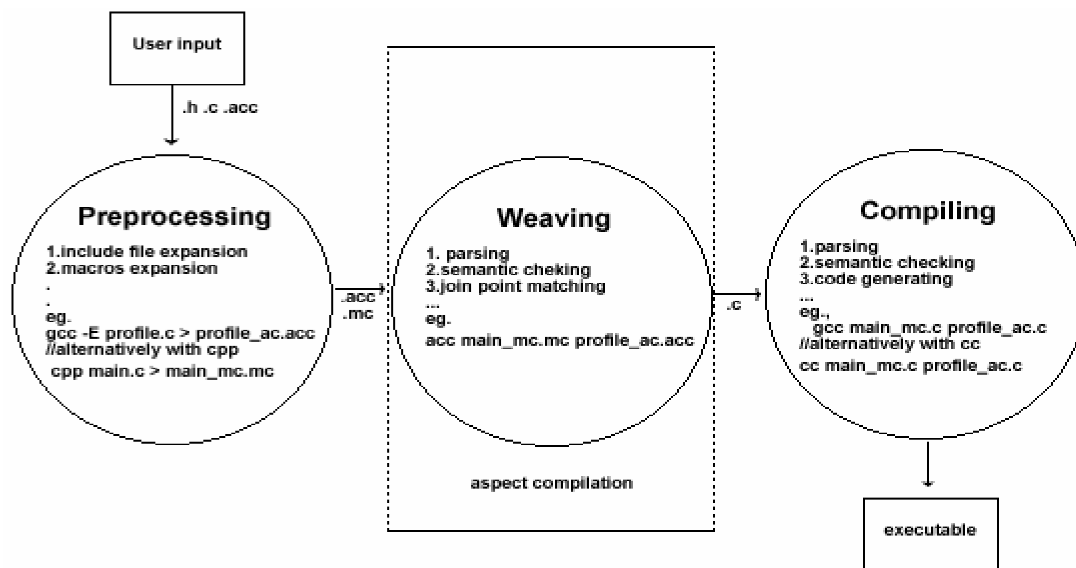


Figura 10: Proceso de Compilación en AspeCt C

El aspecto orientado a compilador de C es una fuente a fuente traductor. Como aportación que los procesos de aspecto orientado a C y C ficheros fuente y produce ANSI-C compatible con los archivos como de salida. Los archivos de salida pueden ser cumplidos por un ANSI-C compatible con el compilador, como gcc.

La siguiente tabla describe los archivos tanto de entrada como de salida para un programa básico y uno orientado a aspectos; junto con los sufijos generados para cada archivo.

Tabla 2: Archivos generados por el compilador orientado a aspectos

Descripción	Archivo de entrada	Archivo generado
programa básico	. MC por ejemplo, hello.mc	. c por ejemplo, hello.c
aspecto del programa	. AC por ejemplo, world.ac	. c por ejemplo, world.c

3.2.3 Aspectos con PHP

PHP es un lenguaje de programación utilizado especialmente para el desarrollo Web. Tiene código dinámico y soporta la programación procedural como la orientada a objetos. Para la POO ofrece modelos de clases similares a las de C# o Java.

Aquí se intenta dar una visión general de las diferentes opciones de la aplicación de POA para PHP y revisar las actuales implementaciones.

1. **Preprocesador**: un preprocesador se puede utilizar para realizar transformaciones de código fuente tejiendo estáticamente aspectos en el código del programa base. El resultado de este tejido es código PHP que puede ser aplicado en un entorno estándar de PHP.

Implementaciones existentes

PHPAspect[1]

Es una extensión del lenguaje PHP que permite incorporar la programación orientada a aspectos (POA), inspirado en AspectJ. Este proyecto fue creado por William Candillon y Gilles Vanwormhoudt junto con Google Summer of Code 2006.

PHPAspect proporciona un compilador, escrito en PHP, que realiza las transformaciones del código fuente utilizando el tejido estático (antes de la ejecución del código fuente) y basado en el análisis de Lex and Yacc para generar los árboles de XML. Como se muestra en la Figura 11, se utiliza XSLT para realizar la transformación de código fuente en árboles de XML.

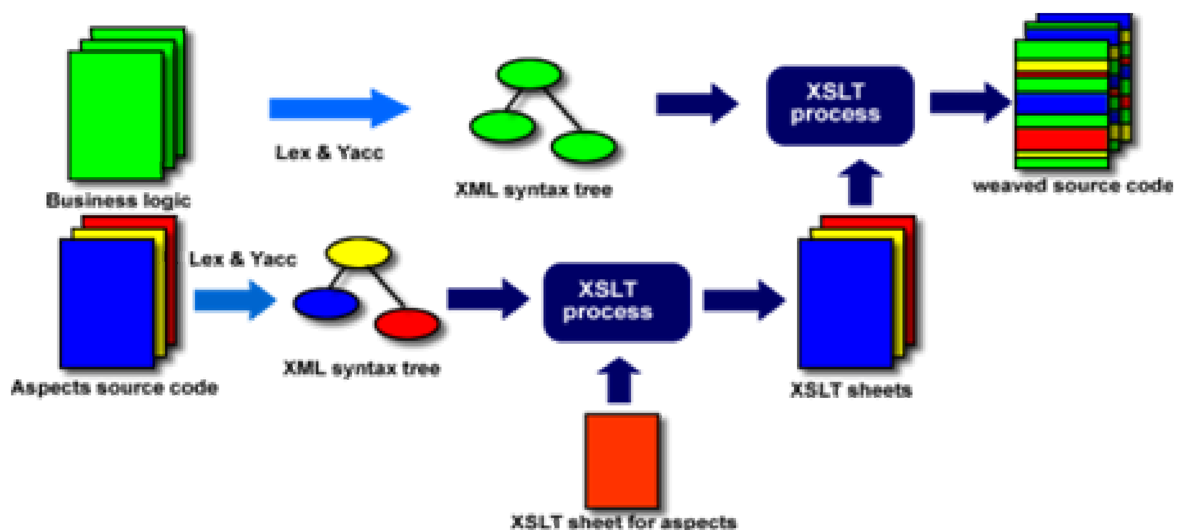


Figura 11: Cadena de tejido de PHPAspect[F,11]

El tejedor de código PHP puede ser ejecutado en cualquier versión de PHP5. PHPAspect contiene todos los joinpoints tradicionales.

Con el uso de XSLT y XPath el autor de PHPAspect pretende lograr la independencia en

1 PHPAspect: http://www.phpaspect.org/, consultado en Septiembre 2008.
[F,11] Figura extraída de http://www.phpaspect.org/, Septiembre 2008.

el léxico y la sintaxis del análisis de la versión de PHP y una mayor flexibilidad en relación con el lenguaje de aspectos.

Aspect-Oriented PHP ó aoPHP[1]

Es un añadido para PHP que permite el uso de la programación orientada a aspectos en *middleware*.

aoPHP fue escrito por John Stamey y Bryan Saunders de la Universidad de Carolina Coastal (Conway, SC) y Matthew Cameron de SOI, Inc (Charlotte, NC). El proyecto se inició en marzo de 2004, fue impulsado por la conferencia celebrada en Wofford College, en noviembre de 2004. La fecha de lanzamiento oficial de aoPHP 1.0 fue el 28 de diciembre de 2004.

aoPHP fue originalmente escrito y desarrollado en Java 1.5 (se basa 1,0 y 2,0). Se basó en un script de PHP y en el módulo `mod_rewrite` de Apache para reordenar adecuadamente las llamadas entrantes a los scripts de PHP. Las llamadas son redirigidas al tejedor Java aoPHP donde los aspectos se tejen en el servidor. El código resultante se pasa a PHP para ser ejecutado.

La versión original de aoPHP contiene los 3 tipos básicos de *advice* (`before`, `after`, and `around`).

Las versiones actuales de aoPHP (construida la 3.0 y se continua) trabaja en forma muy parecida a la versión original. Todavía se basa en un script PHP que llama al tejedor. Sin embargo, el tejedor está escrito en C++ utilizando plenamente expresiones regulares. La combinación de ambos, y la eliminación de la JVM, proporciona una gran mejoría en el rendimiento, la velocidad y la exactitud del analizador y del tejedor.

Las futuras versiones de aoPHP serán escritas como un módulo de Apache como para eliminar la necesidad del script PHP y el uso de mod_rewrite. Esto también ayuda a simplificar la instalación y el proceso en general.

Evaluación: ya que PHP es un intérprete, un lenguaje dinámico, el tener un tejido estático tiene sus ventajas y desventajas. Por una parte, el tejido se realiza una sola vez antes de la ejecución y no provoca ningún impacto en el rendimiento del tiempo de ejecución. Por otro lado, el tejido estático impone límites a los modelos de join point y a los pointcut del lenguaje.

2. PHP Intérprete extensible a Aspectos

1 AoPHP, http://www.aophp.net/, consultado en Septiembre 2008

Que PHP sea extensible es una de las razones por las cuales se convirtió en el favorito para el desarrollo web.

El Motor Zend: puede ampliar la compilación y ejecución del Interprete PHP usando el lenguaje de programación C. Una extensión puede ser implementada como la carga dinámica de un plug-in o cambiando el interprete del código fuente.

Implementaciones Existentes

Prototipo aspectPHP[1] es una reimplementación de aoPHP a C. Esta se encuentra disponible sólo como un parche para PHP 4.3.10.

3. Metaprogramación

Algunas extensiones del lenguaje se pueden implementar en la programación en PHP mediante las capacidades de la metaprogramación. Esto incluye:

- Una API reflexiva para contemplar en tiempo de ejecución clases, métodos, etc.
- Interceptor de métodos basado en el doesNotUnderstan de Smalltalk.
- Modificaciones en los bytes del código a través de la extensión de un Runkit.

Implementaciones Existentes

Librería POA para PHP[2]: es una biblioteca que admite un modelo rudimentario de *join point* y requiere de cambios manuales de código, fallando a dos características de los sistemas orientados a aspectos: abstracción y cuantificación[RF,2000].

La Figura 12 muestra la forma de declarar "aspectos" y "advices" mediante la librería POA para PHP: se invoca a `aMethod()` en `Aclass()`, luego se invoca el *advice before* y por último el *advice after*.

1 AspectPHP, http://www.cs.toronto.edu/~yijun/aspectPHP, consultado en Septiembre 2008
2 http://phpclasses.org/aopinphp
[RF,2000] Robert E. Filman and Daniel P. Friedman, *"Aspect-Oriented Programming is Quantification and Obliviousness"*, Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis.

```
<?php
require_once 'AClass.php';
require_once 'aop.lib.php';

$aspect   = new Aspect;
$pointCut = $aspect->pointcut('call AClass::aMethod');
$pointCut->_before('... before advice code ...');
$pointCut->_after('... after advice code ...');
$pointCut->destroy();

$object = new AClass($aspect);
?>

<?php class AClass {
    private $aspect;

    public function __construct($aspect) {
        $this->aspect = $aspect;
    }

    public function aMethod() {
        Advice::_before($this->aspect);
        // ... base code ...
        Advice::_after($this->aspect);
    }
} }?>
```

Figura 12: Declaración de Aspectos usando la librería AOP para PHP y clase base usando aspectos. El "aspecto" es solo una librería que es invocada en el código fuente.

El ejemplo muestra que el código base es modificado ampliamente para usar aspectos. Primero el constructor de la clase base tiene que ser modificado para aceptar el objeto `$aspect` y almacenarlo en una variable de instancia[1]. Luego se llama explícitamente a los objetos de los métodos `_before()` y `_after()` estos deben ser insertados en cada método de clase. Esto no tiene diferencia en la invocación de cualquier otra librería, por lo tanto no se justifica el apoyo de la misma.

Evaluación: no se puede desestimar el enfoque de la Metaprogramación por el único intento de llevar Aspectos en PHP con características de metaniveles. Esto genera potencial para aprovechar el dinamismo de PHP y la compatibilidad con diferentes lenguajes.

Evaluando todas las visiones de la programación orientada a aspectos para PHP sólo tres proporcionan realmente la funcionalidad de los aspectos. Dos de ellos se basan en el tejido estático, pero se encuentran por debajo de las características dinámicas del lenguaje PHP.

1 `_constructor()` es el nombre del método constructor en PHP.

Además, se basan en AspectJ, el cual no es compatible con algunos nuevos conceptos de aplicaciones web. El otro intento evaluado tiene una concepción anticuada.

Capítulo 4. Aplicación Práctica

En este capitulo se realizarán diferentes implementaciones de POA en los lenguajes previamente seleccionados, AspeCt C, PHPAspect y AspectJ.

En AspeCt C demostramos la aplicación de POA en un lenguaje procedural, no orientado a objetos, como es C y la implementación de sus características más importantes, join point, pointcut, advice.

En PHPAspect se realiza una aplicación básica orientada a objetos y su equivalente en aspectos, cuyo objetivo es demostrar una particularidad en la declaración de aspectos, la cual se detallará a continuación.

Finalmente en AspectJ se realiza una aplicación, similar a la de PHPAspect, que intenta reflejar, de manera simple, como POA maneja la separación de incumbencias.

4.1 Ejemplos Generales en AspectC.

Introducción

En nuestros ejemplos, vamos a intentar explicar de manera completa y sencilla el funcionamiento de la Programación Orientada a Aspectos en C.

La explicación se basa en tres ejemplos principales en donde se usan diferentes tipos de advices(`before() - after() - around()`) como así también las diferentes declaraciones de variables y el alcance de cada una de ellas en cada uno de los casos. Cabe aclarar que en todos los casos estamos usando dos archivos, uno con extensión .acc que corresponde al aspecto y otro archivo .mc que corresponde al lenguaje C. Una vez que están los dos módulos armados, se procede a crear los archivos de aspectos por medio del weaver (acc) y por ultimo, creamos el ejecutable (gcc).

Ejemplo 1

En nuestra primer aplicación, vamos a mostrar la utilización del advice `before()` y del `after()` de forma sencilla. El módulo world.mc corresponde al código escrito en C mientras que hello.acc corresponde al de aspecto también escrito en C pero hace una diferenciación obligada en la extensión para que el tejedor reconozca cada archivo al momento de realizar el entretejido.

Dentro del world.mc encontramos:

```
int main() {
      printf("world");
}
```

El programa principal comienza mostrando un mensaje por pantalla, el mismo en este caso es "world" pero cabe aclarar que no es la primer frase que muestra en pantalla ya que existe un advice (`before()`) que se ejecutará antes.

El módulo de aspecto: hello.acc es:

```
before(): execution(int main()) {
      printf("Hello ");
}
after(): execution(int main()) {
      printf(" from ACC ! \n");
}
```

En este caso, encontramos dos advices, uno es el `before()` y se ejecuta antes de la ejecución del `main()` y una vez que se ejecuta la instrucción que se encuentra dentro del mismo, devuelve la ejecución al `main()` con el mensaje "Hello" y se le agrega "World".

Por último, el advice `after(),` agrega "from ACC !" ya que esa instrucción se ejecuta después que haya terminado de ejecutar el `main()`.

Ejemplo 2

El siguiente ejemplo intenta demostrar el manejo de parámetros en las funciones y el uso de uno de los advices más potentes; `around()`. El mismo se procesa al momento de ejecución de otra instrucción, ni antes ni después, sino que durante la ejecución.

El programa principal comienza con dos parámetros, un entero y otro array tipo char.

El primer resultado que la aplicación muestra es "before call incr" y luego llama a la función incr con el parámetro 99. Al momento de recibir la función la orden de proceso, el aspecto toma el foco y muestra el mensaje que contiene el mismo con los parámetros recibidos. A su vez el aspecto retorna con el parámetro multiplicado por dos y el `main()` se encarga de mostrar el valor de la nueva variable.

Archivo main.mc

```
int incr(int x){
  int q;
```

```
  printf("inside incr, x = %d\n", x);
  return ++x;
}
int main(int argc, char *argv[]){
  int p;
  printf("before call incr \n");
  p = incr(99);
  printf("after call incr p = %d \n", p);
  return 0;
}
```

El módulo de aspecto around.acc es:

```
Int around(int x): execution(int incr(int )) && args(x){
   printf("around incr function call, parameter = %d \n", x);
   return x * 2 ;
}
```

Ejemplo 3

En el siguiente ejemplo, el programa principal main realiza la llamada a diferentes funciones, las mismas son: `foo()` , `foo1()` , `foo2()` y nuevamente realiza la llamada a la función anterior `foo()`

La primer función `foo()` y la primer instrucción muestra por pantalla "in foo". La siguiente instrucción es interrumpida por el aspecto mediante el advice `before()` y luego por el `after()` mostrando el mensaje correspondiente en cada caso. Una vez terminado eso, muestra el valor de la variable inicializada al comienzo del módulo. Por último, dentro de la misma función, muestra un mensaje informando la finalización del la primer función.

En la función foo1, a la variable utilizada se le pasa el valor 1. La siguiente línea muestra un mensaje "in foo1, local a shadows global a"

En la siguiente línea el contenido de la variable a es 1 y como podemos ver no se ejecutan los aspectos ya que esta definida a nivel local.

En la función `foo2()`, muestra el mensaje "in foo2, assign a be 2" y le asigna 2 a la variable global a. Por último, muestra el mensaje "end of foo2"

La ultima función que se repite en la ejecución es la `foo()`, la cual vuelve a mostrar los datos ya vistos con la diferencia que la variable global cambio de valor ya que foo2 realizó el cambio a la misma.

Archivo main.mc:

```
Int a = 99;
void foo2() {
   printf("in foo2, assign a be 2\n");
   a = 2;
   printf("end of foo2\n");
}
void foo1() {
   int a = 1;
   printf("in foo1, local a shadows global a \n");
   printf("a = %d\n", a);
   printf("end of foo1\n");
}

void foo() {
   printf("in foo\n");
   printf("a = %d\n" , a);
   printf("end of foo\n");
}
int main() {
   foo();
   foo1();
   foo2();
   foo();
   return 0;
}
```

Aspecto

```
before(): get(int a) {
      printf("aspect: before get global a\n");
}
after(): get(int a) {
      printf("aspect: after get global a\n");
}
```

4.2 Carrito básico en PHPAspect.

Introducción

A continuación presentamos la adaptación de un ejemplo de código en PHPAspect

publicado en la página oficial del lenguaje[4], para demostrar una particularidad en la declaración de aspectos.

Implementaremos una versión simple de un carrito de compras, en el cual, al momento de utilizar aspectos, se muestra la implementación de dos advices distintos, mediante la declaración de dos puntos de corte que se aplican, sobre un mismo procedimiento.

Parte 1: Un carrito simple

El sistema consta de las siguientes clases: Order y Catalog.

Descripción de las clases

- Order
  La clase Order representa al carrito de compras, el cual contiene los ítems a comprar. Posee los métodos addItem, para agregar los ítems al carrito, y el método getAmount, para calcular el monto total de compra por ítem.
- Catalog
  La clase Catalog, contiene para cada ítem su precio asociado. Es una clase abstracta

Código Fuente

- Clase Order:

```
<?php

class Order{

      private $items = array();

      private $amount = 0;

      public function addItem($reference, $quantity){

            $this->amount+=$quantity*Catalog::getPrice($reference);
            $this->items[]=array($reference,
$quantity,Catalog::getPrice($reference),$this->amount);

      }

      public function getAmount(){

            return $this->amount;

      }
```

[4] http://www.phpaspect.org/documentation/joinpoints.html

```
}

?>
```

- Clase Catalog:

```
<?php

class Catalog{

      private static $priceList=array('Item1'=>9.31, 'Item2'=>8.46,
'Item3'=>8.70);

      public static function getPrice($reference){

           return self::$priceList[$reference];

      }

}

?>
```

- Programa Principal:

```
<?php

$myOrder = new Order();

$myOrder->addItem('Item1', 2);

$myOrder->addItem('Item2', 2);

$myOrder->addItem('Item1', 6);

while(list($k,$v)=each($myOrder)){

      if(is_array($v)){

            while(list($k1,$v1)=each($v)){

                  $acumulado= $acumulado + ($v1[1]*$v1[2]);

                  echo $v1[1] . ' ' .$v1[0] . ' agregados al
carrito. Monto Total agregado: '. $acumulado . ' Pesos<br>';

            }

      }

}

?>
```

Resultado de la ejecución del sistema:

```
2 Item1 agregados al carrito. Monto Total agregado: 18.62 Pesos
2 Item2 agregados al carrito. Monto Total agregado: 35.54 Pesos
6 Item1 agregados al carrito. Monto Total agregado: 91.4 Pesos
```

Parte 2: Utilización de Aspectos

El objetivo de la Parte 2 es la de demostrar como se puede implementar dos advices distintos, mediante la declaración de dos puntos de corte, logAddItem y logTotalAmount, que se aplican sobre el procedimiento addItem de la clase Order.

Código Fuente

La única modificación que se realiza es en el programa principal.

- Programa principal

```
<?php

$myOrder = new Order();

$myOrder->addItem('Item1', 2);

$myOrder->addItem('Item2', 2);

$myOrder->addItem('Item1', 6);

?>
```

La funcionalidad que se elimina del programa principal, será realizada por el aspecto.

- Archivo de Aspectos

```
<?php

aspect TraceOrder{

  pointcut logAddItem:exec(public Order::addItem(2));

  pointcut logTotalAmount:call(Order->addItem(2));

  after($quantity, $reference): logAddItem{

      printf("%d %s agregados al carrito.", $quantity, $reference);

      //printf(date("d-m-Y H:i:s"));

  }

  after(): logTotalAmount{
```

```
      $fecha = date("d-m-Y H:i:s");

      printf(" Monto Total agregado: %f   Pesos <b>--Fecha de la
compra:</b> %s <br>",

      $thisJoinPoint->getTarget()->getAmount(), $fecha);

   }

}

?>
```

Como se puede observar, se definen dos pointcuts para el procedimiento addItem:

- pointcut logAddItem:exec(public Order::addItem(2));
- pointcut logTotalAmount:call(Order->addItem(2));

La diferencia que existe entre estos dos pointcuts es el contexto: 'call' accede al conexto de llamada, mientras que 'exec' al contexto de ejecución. Como regla, si se quiere capturar un joint point que corra con una signatura en particular, se utiliza 'call'. Caso contrario, 'exec'. No importa el orden en el cual se declaren, primero se ejecuta 'exec', que no depende de la sigantura, y luego 'call'.

Los advices implementados se ejecutan luego de la ejecución del método addItem. El primero muestra la cantidad de ítems cargados en el carrito, y el segundo, calcula el monto total para cada ítem.

Resultado de la ejecución del sistema:

```
2 Item1 agregados al carrito. Monto Total agregado: 18.620000 Pesos
--Fecha de la compra: 28-09-2008 19:58:55
2 Item2 agregados al carrito. Monto Total agregado: 35.540000 Pesos
--Fecha de la compra: 28-09-2008 19:58:55
6 Item1 agregados al carrito. Monto Total agregado: 91.400000 Pesos
--Fecha de la compra: 28-09-2008 19:58:55
```

4.3 Solucionando un Registro de Operaciones con AspectJ.

Introducción

A continuación presentamos la adaptación de un ejemplo de código en AspectJ publicado en [RL,2003][1] para demostrar de manera práctica los conceptos y beneficios de utilizar la

[1] [RL,2003] Ramnivas Laddad, *AspectJ in Action*, Manning Publications, 2003.

programación orientada a aspectos.

Implementaremos una versión simple de un carrito de compras, en el cual al agregarle opciones de registro de las operaciones realizadas, comenzamos a generar código entremezclado. Finalmente se presenta una solución utilizando POA.

El registro de operaciones es una de las técnicas más comunes utilizadas para comprender el comportamiento de un sistema. En su forma más simple, el registro de operaciones imprime cuáles fueron las operaciones realizadas por el sistema. Por ejemplo, en un sistema bancario se podrían registrar todas las operaciones de una cuenta con información como ser la naturaleza de la transacción, el número de transacción, número de cuenta, fecha y monto de la misma. Mediante la revisión de un registro, el desarrollador puede descubrir un comportamiento no deseado del sistema y corregirlo.

Por qué utilizar POA para el registro de las operaciones

Los mecanismos actuales implementan el registro de operaciones junto con la lógica principal de las operaciones, la cual es mezclada con las declaraciones de registro. Además, cambiar el modo de registrar las operaciones, por lo general, requiere modificar varios módulos. Dado que el registro de operaciones es una competencia entrecruzada (un requerimiento, elemento de diseño o implementación que se esparce entre múltiples módulos), la programación orientada a aspectos puede ayudar a modularizarla. Con POA se logra implementar los mecanismos de registro de forma independiente a la lógica principal. POA simplifica las tareas de registro de operaciones modularizando su implementación y eliminando la necesidad de cambiar varios archivos fuente cuando cambian los requerimientos. AOP no sólo resume código, sino que establece un control centralizado, consistente y eficiente.

A pesar de que ya existen varios buenos instrumentos de registro de operaciones, aún se deben escribir las declaraciones de registro en cada lugar que las necesitemos. En las próximas secciones evaluaremos el funcionamiento de un sistema de registro de operaciones tradicional y otro basado en POA.

Presentación del ejemplo: Un Carrito de compras

Como primera parte del ejemplo, presentamos un carrito de compra sin implementar el registro de operaciones. Luego agregaremos el registro de operaciones convencional y finalmente el registro de operaciones aplicando POA.

Parte 1: Un carrito sin registro de operaciones

El sistema consta de los siguientes objetos: Item, ShoppingCart, Inventory y

ShoppingCartOperator. A continuación se presentan los diagramas UML de cada una de las clases:

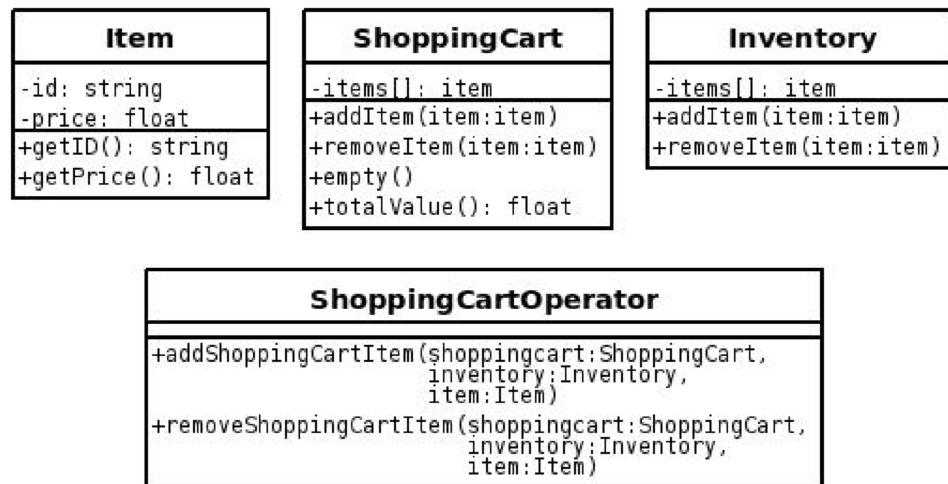


Figura 13: Clases ejemplo AspectJ

Descripción de las Clases

- Clase Item

  Representa a los diversos artículos que se pueden encontrar en un supermercado. Al momento de crear un ítem se especifica su id y precio (price)

- Clase Inventory

  Representa al inventario o stock del supermercado. Posee los métodos addItem y removeItem para agregar y sacar Items del stock.

- Clase ShoppingCart

  Representa al carrito de compras. En él se pueden agregar o sacar ítems, así como vaciarlo y calcular el precio total de los ítems que contiene.

- Clase ShoppingCartOperator

  Representa al comprador. Se encarga de sacar ítems del inventario del supermercado y agregarlos al carrito. También posee la función inversa, sacar los ítems del carrito y devolverlos al stock.

- Clase Test

  Finalmente, existe una clase llamada Test (no se encuentra representada en la imagen), que cumple el rol de nuestra aplicación principal. En ella haremos lo siguiente:

  - Crear tres ítems.
  - Crear el inventario.
  - Agregar los ítems al inventario.

- Crear un carrito de compras.
- Sacar los ítems del inventario y agregarlos al carrito de compras.

Código fuente

- Clase Item

```
public class Item {
    private String id;
    private float price;

    public Item(String id, float price) {
          this.id = id;
          this.price = price;
    }

    public String getID() {
          return this.id;
    }

    public float getPrice() {
          return this.price;
    }

    public String toString() {
          return "Item: " + this.id;
    }
}
```

- Clase Inventory

```
public class Inventory {
    private List items = new Vector();

    public void addItem(Item item) {
          this.items.add(item);
    }
    public void removeItem(Item item) {
          this.items.remove(item);
    }
}
```

- Clase ShoppingCart

```
public class ShoppingCart {
    private List items = new Vector();

    public void addItem(Item item) {
          this.items.add(item);
    }

    public void removeItem(Item item) {
          this.items.remove(item);
    }

    public void empty() {
          this.items.clear();
    }
```

```
      public float totalValue() {
            //unimplemented... free!
            return 0;
      }
}
```

- Clase ShoppingCartOperator

```
   public class ShoppingCartOperator {

            public static void addShoppingCartItem (
                        ShoppingCart shoppingcart,
                        Inventory inventory,
                        Item item) {

                  inventory.removeItem(item);
                  shoppingcart.addItem(item);
            }

            public static void removeShoppingCartItem (
                        ShoppingCart shoppingcart,
                        Inventory inventory,
                        Item item) {

                  shoppingcart.removeItem(item);
                  inventory.addItem(item);
            }
}
```

- Clase Test

```
   public class Test {

      public static void main(String[] args) {
            Inventory inventory = new Inventory();

            Item item1 = new Item("1", 30);
            Item item2 = new Item("2", 31);
            Item item3 = new Item("3", 32);

            inventory.addItem(item1);
            inventory.addItem(item2);
            inventory.addItem(item3);

            ShoppingCart shoppingcart = new ShoppingCart();

            ShoppingCartOperator.addShoppingCartItem(shoppingcart,
inventory, item1);
            ShoppingCartOperator.addShoppingCartItem(shoppingcart,
inventory, item2);

            System.out.println("gracias por su compra");
      }

}
```

Resultado de la ejecución del sistema

En este caso, la salida es una línea de texto por la consola que dice:

```
Gracias por su compra
```

Parte 2: Un carrito con un sistema convencional de Registro de Operaciones

A continuación se presenta el mismo ejemplo pero agregando una interfaz la cual le adicionará a cada clase presentada con anterioridad la capacidad de realizar el registro de sus operaciones.

A diferencia del primer diagrama, ahora hay una interfaz que extiende las capacidades de las clases Item, Inventory y ShoppingCart, para que puedan realizar las operaciones de registro de operaciones. Tanto la clase ShoppingCartOperator como Test, quedan intactas.

Código Fuente

En el código fuente, se marca en negrita los cambios realizados a las clases. Omitiremos las clases que no han sufrido modificación alguna.

- Interfaz Logger

```
public interface Logger {
    public void printLog(String object, String method);
}
```

- Clase Item

```
public class Item implements Logger{
    private String id;
    private float price;

    public Item(String id, float price) {
          this.id = id;
          this.price = price;
          this.printLog("Item", "Item");
    }

    public String getID() {
          return this.id;
    }

    public float getPrice() {
          return this.price;
    }
```

```
      public String toString() {
            return "Item: " + this.id;
      }

      public void printLog(String object, String method){
            Date now = new Date();
            System.out.println(now + " " + object + " " + method);
      }
}
```

- Clase Inventory

```
   public class Inventory implements Logger{
      private List items = new Vector();

      public void addItem(Item item) {
            this.items.add(item);
            this.printLog("Inventory", "addItem");
      }
      public void removeItem(Item item) {
            this.items.remove(item);
            this.printLog("Inventory", "removeItem");
      }

      public void printLog(String object, String method){
            Date now = new Date();
            System.out.println(now + " " + object + " " + method);
      }
}
```

- Clase ShoppingCart

```
   public class ShoppingCart implements Logger{
      private List items = new Vector();

      public void addItem(Item item) {
            this.items.add(item);
            this.printLog("ShoppingCart", "addItem");
      }

      public void removeItem(Item item) {
            this.items.remove(item);
            this.printLog("ShoppingCart", "removeItem");
      }

      public void empty() {
            this.items.clear();
      }

      public float totalValue() {
            //unimplemented... free!
            return 0;
      }

      public void printLog(String object, String method){
            Date now = new Date();
            System.out.println(now + " " + object + " " + method);
      }
}
```

Resultado de la ejecución del sistema

La salida es la siguiente:

```
 Mon Sep 08 20:08:54 ART 2008 Item Item
 Mon Sep 08 20:08:54 ART 2008 Item Item
 Mon Sep 08 20:08:54 ART 2008 Item Item
 Mon Sep 08 20:08:54 ART 2008 Inventory addItem
 Mon Sep 08 20:08:54 ART 2008 Inventory addItem
 Mon Sep 08 20:08:54 ART 2008 Inventory addItem
 Mon Sep 08 20:08:54 ART 2008 Inventory removeItem
 Mon Sep 08 20:08:54 ART 2008 ShoppingCart addItem
 Mon Sep 08 20:08:54 ART 2008 Inventory removeItem
 Mon Sep 08 20:08:54 ART 2008 ShoppingCart addItem
Gracias por su compra
```

En este caso, se observa la salida de la ejecución de los métodos creados mediante la interfaz Logging. Para cada método que registra sus operaciones, se imprime la fecha, el objeto y su operación.

Al final hay una línea de texto que indica el fin de la ejecución del sistema.

Inconvenientes

Si bien este sistema es muy pequeño, ya es factible notar que la introducción de la interfaz implementada para realizar el registro de operaciones agrega código que no compete a cada una de las clases afectadas, diseminándose así por todo el sistema y tornándolas menos comprensibles.

En un sistema de tamaño medio, y mucho más en uno grande, la cantidad de situaciones como la que acabamos de presentar sería mucho mayor. Mantener, corregir y entender el código se tornaría complicado y tedioso.

Parte 3: Un carrito de compras utilizando Aspectos para el registro de operaciones.

Con el uso de Aspectos vamos a mejorar la calidad del código generado en la segunda parte del ejemplo. Agregando un aspecto al sistema lograremos que el código agregado por la interfaz se concentre de forma modular y sin agregar una línea de código a las clases base.

Código Fuente

Como dijimos anteriormente, las clases base, Item, Inventory, ShoppingCart, ShoppingCartOperator y Test poseen el mismo código que en la primera parte del ejemplo, por lo que serán omitidas.

- Aspecto Logger

```
package aspecto;
import org.aspectj.lang.*;
import java.util.Date;

public aspect Logger {

  pointcut logMethods():
    call(* *.*(..)) && !within(Logger);

    //call(void Inventory.addItem(..));

  after(): logMethods() {
    Signature sig = thisJoinPointStaticPart.getSignature();
    Date now = new Date();
    System.out.println(now + " "
sig.getDeclaringType().getSimpleName()
                      + " " + sig.getName());

  }

}
```

Resultado de la ejecución del sistema

La salida es la siguiente:

```
Mon Sep 08 20:09:54 ART 2008 Inventory addItem
Mon Sep 08 20:09:54 ART 2008 List add
Mon Sep 08 20:09:54 ART 2008 Inventory addItem
Mon Sep 08 20:09:54 ART 2008 List add
Mon Sep 08 20:09:54 ART 2008 Inventory addItem
Mon Sep 08 20:09:54 ART 2008 List add
Mon Sep 08 20:09:54 ART 2008 ShoppingCartOperator
addShoppingCartItem
Mon Sep 08 20:09:54 ART 2008 Inventory removeItem
Mon Sep 08 20:09:54 ART 2008 List remove
Mon Sep 08 20:09:54 ART 2008 ShoppingCart addItem
Mon Sep 08 20:09:54 ART 2008 List add
Mon Sep 08 20:09:54 ART 2008 ShoppingCartOperator
addShoppingCartItem
Mon Sep 08 20:09:54 ART 2008 Inventory removeItem
Mon Sep 08 20:09:54 ART 2008 List remove
Mon Sep 08 20:09:54 ART 2008 ShoppingCart addItem
Mon Sep 08 20:09:54 ART 2008 List add
Gracias por su compra
```

De forma similar a la parte 2, en esta salida también se imprime la fecha y hora de la llamada a los métodos.

Cabe destacar que en la salida de la parte 3 no figura la invocación a los constructores de la clase Item, dado que la forma en la que se declaró al aspecto no lo involucra. Además, pueden verse la salida de otros métodos, que antes no habían sido declarados en la parte 2.

Solución de los inconvenientes anteriores

El código original de cada una de las clases no se vio afectada por el aspecto, quedando sin ese código extra necesario que no forma parte de la competencia de las clases. Ahora, mantener, corregir y entender el código es más simple, dado que se encuentra reunido en una nueva unidad modular, el aspecto.

Aplicación de las Métricas LDCF en el ejemplo anterior

En esta sección se comparan las tres partes que forman al ejemplo práctico del Carrito de compras aplicando la métrica de LDCF.

Para implementar la métrica, se tomó como estándar el conteo de líneas lógicas de código fuente. Se omite en el conteo las líneas con blancos, los comentarios y las que sólo contienen las llaves ({}) de apertura y cierre de métodos y clases.

Tabla 3: Comparación de LDCF para cada una de las clases en las tres partes de nuestro ejemplo

Clases	Parte 1 (Simple)	Parte 2 (Interfaz)	Parte 3 (Aspecto)
Item	12	16	12
Inventory	6	11	6
ShoppingCart	10	15	10
ShoppingCartOperator	7	7	7
Test	12	12	12
Interfaz Logger	-	2	-
Aspecto Logging	-	-	7
Total	**47**	**63**	**54**
	100%	**136%**	**114%**

Como se puede observar en la Tabla 3, la Parte 2 es la que contiene la mayor cantidad de líneas de código, con un total de 63. La Parte 3 contiene un 14,28% menos de código, con 54 líneas.

Además, cada clase de la Parte 3, es idéntica a las de la Parte 1; mientras que en la Parte 2, las clases afectadas por las funciones de logging tienen agregadas líneas de código específicas para realizar la función de registro. La clase Item contiene un 33% más de LDCF, Inventrory un 83% y ShoppingCart un 50%.

Si bien el Aspecto es un tres veces y media más grande que la interfaz, el aspecto evita tener que agregar líneas de código a cada una de las demás clases.

Conclusión

A medida que se generen más clases en el sistema y se necesite mantener un registro de las operaciones, en la Parte 2 se irán agregando más LDCF, mientras que en la Parte 3 no hace falta agregar ni una línea más de código fuente.

Un uso correcto de la POA permite reducir la cantidad de LDCF de sistemas complejos orientados a objetos, permitiendo tener código más limpio, concreto e interpretable. Las incumbencias ajenas a las responsabilidades de las clases se agrupan en una nueva unidad modular llamada Aspecto, en donde es más fácil controlarlas.

Capítulo 5. Conclusiones

Luego del desarrollo de este trabajo tanto en su parte teórica como en la práctica, pudimos experimentar los inconvenientes y virtudes que presenta la implementación de este paradigma.

Como mencionamos en el capítulo 2, la programación orientada a aspectos posee grandes ventajas, así como inconvenientes si no es utilizada con cuidado.

Hemos encontrado algunas dificultades a la hora de encontrar información detallada para muchos lenguajes de programación base que incorporan el uso de aspectos. Para el caso de los lenguajes seleccionados, en especial PHPAspect y AspeCt C, surgieron inconvenientes a la hora de ponerlos en funcionamiento, no así en AspectJ que es el lenguaje 'oficial' de POA.

Estamos convencidos de que queda mucho por hacer e investigar, como por ejemplo obtener métricas formales que sirvan para cuantificar la utilidad de la POA; analizar cómo aplicar aspectos en las otras etapas del ciclo de vida del software: en el análisis, diseño, testing y documentación. Asimismo son pocas las herramientas existentes que respeten sus principios de diseño.

Si bien el paradigma ya tiene varios años de existencia, no posee la difusión que le corresponde, dado que en gran parte no ha sido incluido en las currículas académicas de las universidades.

Anexos

AspectJ

1. Fechas de lanzamiento de las distintas versiones de AspectJ

Tabla 4: Fechas de lanzamiento de las versiones de AspectJ y su tamaño aproximado

Versión	Fecha de lanzamiento	Tamaño aprox. en MB
1.6.1	03/07/2008	11
1.6.0	23/04/2008	11
1.5.4	20/12/2007	11
1.5.3	22/11/2006	10
1.5.2	30/06/2006	10
1.5.1a	10/04/2004	10
1.5.0	20/12/2005	10
1.2.1	05/11/2004	7
1.2.0	25/05/2004	6
1.1.1	22/09/2003	6
1.1.0	06/06/2003	6
1.0.6	Fecha no disponible	2

Fuente: Página oficial de descargas de AspectJ http://www.eclipse.org/aspectj/download.php, consultada en Septiembre 2008.

2. Popularidad de los lenguajes de programación según TIOBE

Tabla 5: Top 10 de popularidad de los lenguajes de programación para el mes de Julio de 2008 medido en porcentaje.

Posición	Lenguaje de programación	Popularidad en %
1	Java	21,345
2	C	15,945
3	C++	10,693
4	(Visual) Basic	10,447
5	PHP	9,525
6	Perl	5,131
7	Python	4,973
8	C#	4,000
9	JavaScript	2,757
10	Ruby	2,735

Fuente: TIOBE http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, consultada en Septiembre 2008.

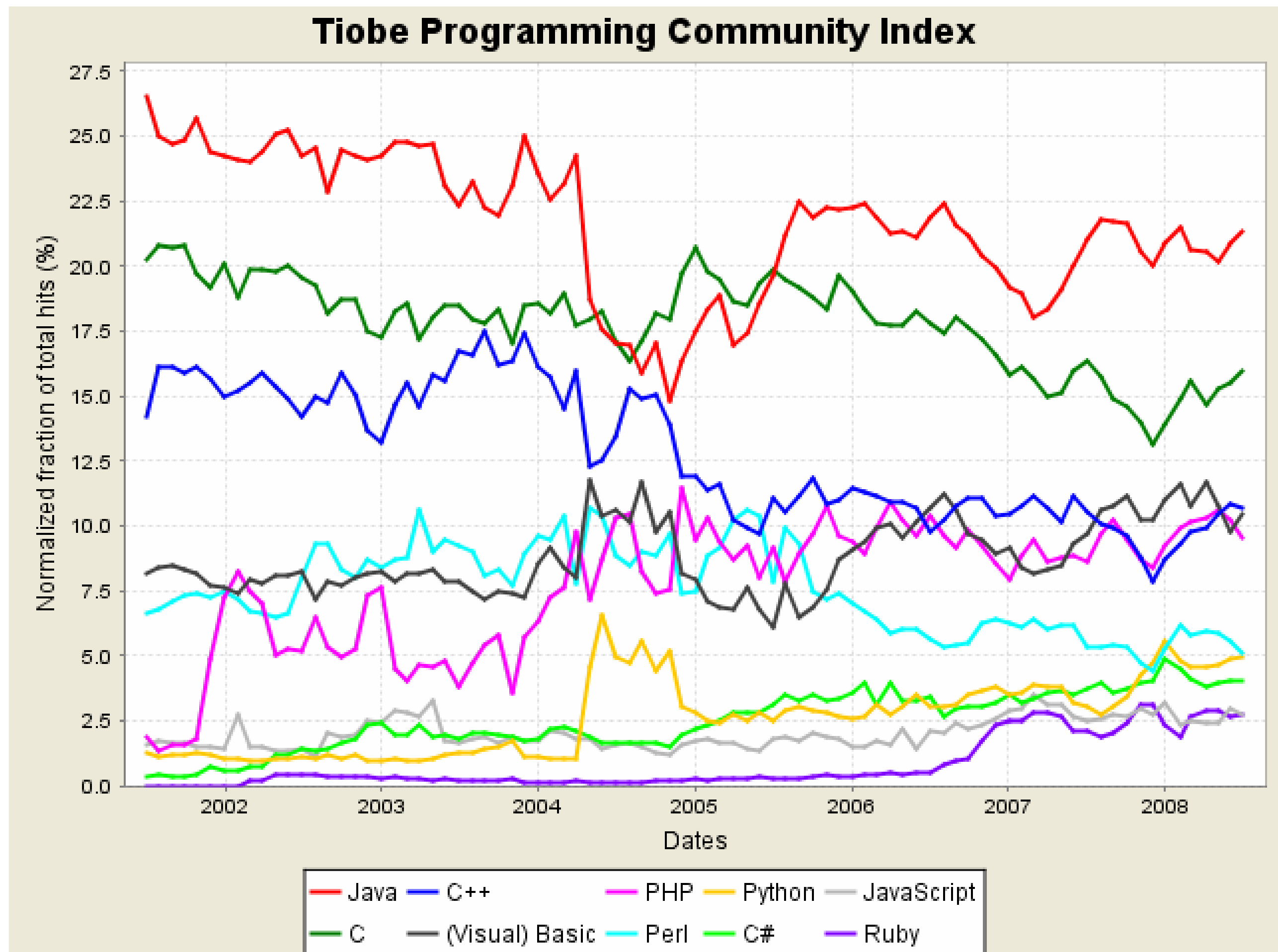


Figura14: Evolución de popularidad de los 10 primeros lenguajes de programación más populares desde mediados del año 2001 hasta Julio de 2008. **Fuente:** TIOBE

3. Sintaxis específica de AspectJ

A continuación se presentan ejemplos de la sintaxis utilizada en AspectJ

3.1 Puntos de Unión (join point)

- **Llamada a un método**

```
punto.setX(10);
```

- **Llamada a un constructor**

```
Punto p = new Punto(5,5)
```

- **Ejecución de un método**

```
public void setX(int x) {
    this.x = x;
}
```

- **Ejecución de un constructor**

```
public Punto(int x, int y){
    super();
    this.x = x;
    this.y = y;
}
```

- **Lectura de un atributo**

```
public int getX() {
    return x ;
}
```

- **Escritura de atributo**

```
public void setX(int i) {
    x = i;
}
```

- **Ejecución de un manejador de excepciones**

```
try{
    ......................
}catch(IOException e){
    e.printStackTrace();
}
```

- **Iniciación de clase**

```
public class Main {
    ........................
    static {
        try {
            System.loadLibrary("lib");
        }
        catch(UnsatisfiedLinkError e)
        {
        }
    }
    ........................
}
```

- **Iniciación de objeto.**

```
public Punto(int x, int y) {
    super();
    this.x = x;
    this.y = y;
}
```

Si el constructor anterior es llamado, la llamada al constructor padre (`super()`) no forma parte del punto de enlace.

- **Pre-iniciación de un objeto**

```
public CuentaAhorros(int numCuenta, string titular, int saldoMinimo){
    super(titular, Cuenta.generarId() );
    this.saldoMinimo=saldoMinimo;
}
```

Si el constructor anterior es llamado, el punto de enlace sólo abarca la llamada al método `Cuenta.generarId()`.

- **Ejecución de un aviso**

```
after(Punto p): cambioPosicionPunto(p) {
    Logger.writeLog("...");
}
```

3.2 Sintaxis y ejemplos de los Puntos de corte

- **`call(<PatronFirma>)`**: incluye los *join points* en donde se realice una llamada a métodos que coincidan con la firma pasada como parámetro. Esta firma puede incluir comodines en distintos puntos, para poder agrupar varios métodos relacionados con una sola expresión. La expresión <PatronFirma> puede ser una patrón de método o de constructor y la sintaxis formal es:

```
<PatronFirma> := PatronMetodo | PatronConstructor

<PatronMetodo> :=
      [ <PatronModificadoresMetodo> ] <PatronTipo>
      [ <PatronTipo> . ] <PatronIdentificador> ( <PatronTipo> |
      .. , ... )
      [ throws <PatronThrow> ]

<PatronConstructor> :=
      [ <PatronModificadoresConstructor> ] [ <PatronTipo> . ]
      new ( <PatronTipo> | .. , ... )
      [ throws <PatronThrow> ]
```

Como se puede ver en la sintaxis formal, la diferencia entre ambos es que para el caso de un constructor, no hace falta el tipo de retorno y en vez de un identificador, se utiliza la palabra new.

Las expresiones <PatronModificadoresMetodo> y <PatronModificadores- Constructor> son simplemente una enumeración de los modificadores de Java que se pretenden incluir o el modificador precedido por el signo "!" si se lo quiere excluir. Los modificadores posibles para un constructor son public, protected y private. Para el caso de un método se agregan los modificadores static, final y synchronized.

La expresión <PatronIdentificador> es una secuencia de caracteres válidos para un identificador en Java con la posibilidad de utilizar el carácter comodín "*" para reemplazar un grupo de caracteres.

La expresión <PatronThrow> es una lista separadas por coma de <PatronTipo> en donde cada elemento puede ir precedido del signo "!" en caso de querer excluirlo.

Por último, la expresión <PatronTipo> conceptualmente representa un patrón que agrupa clases o interfaces. Este patrón es utilizado en la expresiones <PatronMétodo> y <PatronConstructor> para reemplazar el tipo de retorno del método, el calificador (a qué tipo pertenece el método) y a los ti pos de cada uno de los parámetros. A partir de la versión 1.5 de AspectJ se introdujo compatibilidad con Java 1.5 y es por eso que se aceptan anotaciones dentro de este patrón. La sintaxis es:

```
<PatronTipo> := <PatronTipoSimple> |
```

```
                 ! <PatronTipo> |
                 ( [ <PatronAnotacion> ] <PatronTipo> )
                 <PatronTipo> '&&' <PatronTipo> |
                 <PatronTipo> '||' <PatronTipo>

<PatronTipoSimple> := <PatronNombreTipo> [ + ] [ '[]' ... ]
```

Esta definición recursiva, muestra que el <PatronTipo> consiste de expresiones simples de tipos que pueden ser combinadas con operadores booleanos y a los cuales se les puede indicar un patrón de anotación. El <PatronTipoSimple> consta de una expresión <PatronNombreTipo> que consta de una cadena de caracteres para especificar un identificador de tipo, opcionalmente con su paquete y puede contener dos tipos de comodines, "*" y "...".

El comodín "*" reemplaza una cadena de caracteres que no incluyan el carácter"..." mientras que el comodín "..." reemplaza un grupo de términos separados por el carácter ".". Si bien se pueden agrupar distintos tipos de esta manera (basados en patrones de nombre o de paquete), el signo "+" permite hacer coincidir además, a todos los tipos derivados de las coincidencias originales, es decir clases o interfaces hijas según sea el caso. En caso de tratarse de vectores se deben agregar la cantidad correspondientes de pares de corchetes según la dimensión.

La expresión <PatronAnotacion> permite reducir el grupo de coincidencias basándose en las anotaciones que posee el elemento. La sintaxis es:

```
<PatronAnotación> := [ ! ] @ <PatronNombreTipo> | (
         <ExpresionTipo> ) ...
```

Es una lista de términos encabezados por el signo "@". La semántica en el caso de los términos de la lista es la conjunción, es decir una coincidencia se dará cuando se cumplan las condiciones de todos los términos. Opcional mente se puede preceder a cada término por el signo "!" para negarlo. Cada término puede ser simple o complejo. En el primer caso consta de un identificador de tipo. En el segundo caso, la expresión debe ir encerrada entre paréntesis y consta de una expresión booleana de identificadores del primer tipo con la posibilidad de combinarlos con los operadores "||", "&&" o "!".

- **`get(<PatronAtributo>)`**: representa los *join points* en donde se efectúa una lectura de los atributos que coinciden con la expresión <PatronAtributo>. Dicha expresión tiene la siguiente sintaxis:

```
<PatronAtributo> :=
         [ <PatronAnotacion> ] [ <PatronModificadoresAtributo> ]
         <PatronTipo> [ <PatronTipo> ] . <PatronIdentificador>
```

- La expresión consta de un patrón de anotación opcional seguido de un <PatronModificadoresAtributo> que tiene la misma sintaxis que en los casos anteriores y

las posibilidades son equivalentes a los modificadores permitidos para un atributo en Java, es decir, public, private, protected, static, transient y final.

- La expresión se completa con un término <PatronTipo>, explicado anteriormente, que indica el tipo de los atributos a capturar y otra expresión opcional del mismo tipo que especifica a qué clase pertenecen los atributos a capturar. Por último se especifica el patrón que aplicará al identificador del atributo.

3.2.1 Puntos de corte anónimos

La sintaxis de un punto de corte anónimo es la siguiente:

```
<pointcut> ::= { [!] designator [ && | || ] };
designator ::= designator_identifier(<signature>)
```

Un punto de corte anónimo consta de una o más expresiones (designator) que identifican una serie de puntos de enlace. Cada una de estas expresiones esta formada por un descriptor de punto de corte y una signatura. El descriptor de punto de corte especifica el tipo de punto de corte y la signatura indica el lugar donde se aplica.

El siguiente ejemplo muestra un punto de corte anónimo asociado a un aviso de tipo after que consta de dos expresiones combinadas mediante el operador lógico AND (&&). La primera expresión utiliza el descriptor de punto de corte call con su correspondiente signatura de método, para identificar las llamadas a los métodos set del la clase Punto y la segunda usa el descriptor target para exponer el contexto del punto de enlace.

```
after(Punto p): call( * Punto.set*(int) ) && target(p) {
    Logger.writeLog("Cambio posición Punto: "+p.toString());
}
```

3.2.2 Puntos de corte con nombre

La sintaxis de un punto de corte con nombre es la siguiente:

```
<pointcut> ::= <access_type> [abstract] pointcut <pointcut_name>
({<parameters>}) : {[!] designator [ && | || ]};

designator ::= designator_identifier(<signature>)
```

Un punto de corte con nombre consta del modificador de acceso (access_type), la palabra reservada pointcut seguida del nombre del punto de corte y de sus parámetros, que permiten exponer el contexto de los puntos de enlace a los avisos (*advice*). Los modificadores posibles son los mismos que para el caso de un aspecto: public, private, default y final. Los parámetros tienen la misma sintaxis que en el caso de los métodos de Java, es decir, una lista separada por coma de parámetros los cuales se componen del tipo (clase o interfaz) y el identificador del parámetro. Por último, vienen las expresiones que identifican los puntos de enlace precedidas del carácter

dos puntos ":". Estas expresiones son idénticas a las utilizadas en los puntos de corte anónimos. El siguiente ejemplo muestra un punto de corte con nombre equivalente al punto de corte anónimo anterior.

```
pointcut cambioPosicionPunto(Punto p):
    call ( * Punto.set*(int) ) && target (p);
```

3.2.3 Comodines y operadores lógicos

A la hora de expresar los puntos de enlace se pueden usar una serie de comodines para identificar puntos de enlace que tienen características comunes. El significado de estos comodines dependerá del contexto en el que aparezcan:

*: el asterisco en algunos contextos significa cualquier número de caracteres excepto el punto y en otros representa cualquier tipo (clase, interfaz, tipo primitivo o aspecto).

.. : el carácter dos puntos representa cualquier número de caracteres, incluido el punto. Cuando se usa para indicar los parámetros de un método, significa que el método puede tener un número y tipo de parámetros arbitrario.

+: el operador suma representa una clase y todos sus descendientes, tanto directos como indirectos.

Las expresiones que identifican los puntos de enlace se pueden combinar mediante el uso de los siguientes operadores lógicos:

exp1 | | exp2: operador lógico OR. La expresión compuesta se cumple cuando se satisface al menos una de las dos expresiones, exp1 o exp2.

exp1 && exp2: operador lógico AND. La expresión compuesta se cumple cuando se cumple exp1 y exp2.

!exp: operador de negación. La expresión compuesta se cumple cuando no se satisface exp.

La precedencia de estos operadores lógicos es la misma que en Java. Es recomendable el uso de paréntesis para hacer más legible las expresiones o para modificar la precedencia por defecto de los operadores.

3.2.4 Puntos de Cortes primitivos

3.2.4.1 Basados en las categorías de puntos de enlace

Este tipo de puntos de corte identifica puntos de enlace que pertenecen a cierta categoría. Existe un descriptor para cada categoría de puntos de enlace.

Tabla 6: Sintaxis básica de los puntos de corte según la categoría de los puntos de unión.

Descriptor	Categoría punto de enlace
`call(methodSignature)`	Llamada a método
`execution(methodSignature)`	Ejecución de método
`call(constructorSignature)`	Llamada a constructor
`execute(constructorSignature)`	Ejecución de constructor
`get(fieldSignature)`	Lectura de atributo
`set(fieldSignature)`	Asignación de atributo
`handler(typeSignature)`	Ejecución de manejador
`staticinitialization(typeSignature)`	Inicialización de clase
`initialization(constructorSignature)`	Inicialización de objeto
`preinitialization(constructorSignature)`	Pre-inicialización de objeto
`adviceexecution()`	Ejecución de aviso

- **`call (methodSignature)`**: incluye los puntos de unión en donde se realice una llamada a métodos que coincidan con la firma pasada como parámetro. Esta firma puede incluir comodines en distintos puntos, para poder agrupar varios métodos relacionados con una sola expresión.

- **`execution(methodSignature)`**: similar al punto de corte primitivo del tipo call, con la diferencia de que el punto de unión se encuentra inmediatamente después de haber realizado la llamada e inmediatamente antes de iniciar la ejecución del método.

- **`get(fieldSignature)`**: representa los puntos de unión en donde se efectúa una lectura de los atributos que coinciden con la expresión *fieldSignature*.

- **`set(fieldSignature)`**: similar al punto de corte primitivo del tipo get, con la diferencia que se seleccionan los puntos de unión de escritura en vez de lectura, de los atributos que coincidan con el patrón especificado.

- **`handler(typeSignature)`**: representa los puntos de unión en donde se captura una excepción del tipo indicado por la expresión *typeSignature*. Esta captura está siempre especificada por una cláusula catch en Java.

- **`adviceexecution()`**: representa los puntos de enlace donde se inicia la ejecución de un aviso (*advice*). No requiere argumentos.

- **`within(typeSignature)`**: representa el conjunto de todos los puntos de unión que tienen como elemento en la cadena de llamadas al tipo especificado en la expresión pasada como argumento.

Tabla 7: Sintaxis básica de los puntos de corte.

Punto de corte	Puntos de enlace identificados
`call(void Punto.set*(..))`	Llamadas a los métodos de la clase `Punto`, cuyo nombre empieza por `set`, devuelven `void` y tienen un número y de parámetros arbitrario.
`execute(public Elemento.new(...))`	Ejecución de todos los constructores de la clase `Elemento`.
`get(private * Linea , *)`	Lecturas de los atributos privados de la clase `Linea`.
`set(* Punto.*)`	Escritura de cualquier atributo de la clase `Punto`.
`handler(IIOException)`	Ejecución de los manejadores de la excepción Exception `IOException`.
`initialization(Elemento.new(..))`	Inicialización de los objetos de la clase `Elemento`.
`staticinitialization(Logger)`	Inicialización estática de la clase `Logger`

3.2.4.2 Basados en flujo de control

- **`low (<PointcutExpression>)`**: representa los *join points* que están dentro del flujo de ejecución de los join points capturados por la expresión pasada como parámetro, incluyendo a estos últimos. Dentro del flujo de ejecución significa que los *join points* figuran dentro de la cadena de llamada.

- **`Cflowbelow (<PointcutExpression>)`**: similar al caso anterior, con la diferencia que los *join points* representados por la expresión pasada como parámetro, son excluidos del conjunto.

- **`Staticinitialization (<PatronTipo>)`**: captura los *join points* del inicio de la ejecución de un inicializador estático para los tipos que coinciden con el patrón pasado como parámetro.

- **`Initialization (<PatronFirma>)`**: captura los *join points* del inicio de la ejecución de un constructor desde que retorna la ejecución del constructor padre (super) hasta el retorno del primer constructor, para los constructores que coinciden con el patrón pasado como parámetro.

- **`Preinitialization (<PatronFirma>)`**: a diferencia del anterior, los *join points* capturados son los de la ejecución desde el inicio del constructor hasta la llamada del constructor padre (super).

Para poder enlazar los parámetros que se le proveen al *pointcut* con la expresión que lo definirá, se cuenta con un proceso denominado "Exposición de Contexto". En dicho proceso,

ciertos operadores llevan a cabo la tarea de relacionar los parámetros del *pointcut* con objetos definidos según el tipo de operador. Estos operadores son:

- **`this (<Tipo>|< PointcutVar>)`**: representa el conjunto de *join points* en donde el objeto que se está ejecutando es del tipo indicado por el parámetro. En el caso de pasar como parámetro un identificador de variable del *point cut*, se asigna a dicha variable el objeto en ejecución al momento de alcanzar el *join point*.
- **`Target (<Tipo>|< PointcutVar>)`**: similar al caso anterior con la diferencia de que en vez de referirse al objeto que se está ejecutando, se refiere al objeto destino del *join point*. Por ejemplo, en un *join point* del tipo "call" corresponderá al objeto sobre el cual se efectúa la llamada.
- **`args( <Tipo> | <PointcutVar> , ... )`**: en este caso, se trata de referenciar los argumentos del *pointcut*. A diferencia de los otros dos casos, recibe una lista de argumentos que pueden ser tipos o identificadores. Semánticamente representa los *join points* cuyos tipos parámetros coinciden con los tipos (o los tipos de los identificadores) pasados como parámetro. En el caso de proveerse identificadores como parámetro, a dicha variable se le asigna el valor que posee el argumento correspondiente en el *join point* al momento de ser capturado, por ejemplo al ejecutar un *advice*.

3.3 Ejemplos de puntos de corte

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

El punto de corte captura cada punto de enlace que es invocado por un método con una firma void Point.setX (int) – o sea, el método setX del objeto Point con un único parámetro int.

Un punto de corte puede ser compuesto por los operadores AND (&&), OR (||) y NOT (!). Por ejemplo:

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

Los puntos de corte pueden identificar puntos de enlace de diferentes tipos. Por ejemplo:

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

AspectJ permite a los programadores definir la forma de nombrar a los puntos de corte. Un punto de corte se puede formar por una parte izquierda y una parte derecha, separadas ambas por dos puntos. En la parte izquierda se define el nombre del corte y el contexto del corte. La parte derecha define los eventos del corte. De esta forma permite declarar un nuevo punto de corte:

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

A los descriptores de eventos de la parte derecha de la definición del corte se les llama designadores. Un designador puede ser:

- Un método.
- Un constructor.
- Un manejador de excepciones.

También se pueden utilizar caracteres comodines en la descripción de los eventos. Por ejemplo:

```
call(void Figure.make*(..))
```

El punto de corte captura cada punto de enlace que es invocado por los métodos definidos en el objeto Figura cuyo nombre comience con make y sin importar el tipo de parámetro que posean.

4. Sintaxis de un Aviso

```
<Advice> := [ strictfp ] <AdviceSpec> [ throws <ListaDeTipos> ] :
        <PointcutExpression> { <CuerpoAdvice> }

<AdviceSpec> :=
        before ( <Argumentos> ) |
        after  ( <Argumentos> ) |
        after  ( <Argumentos> ) returning [ ( <Argumento> ) ]
        after  ( <Argumentos> ) throwing [ ( <Argumento> ) ]
        <Tipo> around ( <Argumentos> )
```

AspectJ posee diferentes tipos de Avisos, tal como lo indica la Tabla 8

Tabla 8: Tipo de Avisos utilizados en AspecJ

Tipo de Aviso	Momento de ejecución
Before	Justo antes de que lo hagan las acciones asociadas con los eventos del punto de corte (antes de que el método actual se ejecute pero luego de que los argumentos del método invocado son evaluados).
After	Justo después de que lo hayan hecho las acciones asociadas con los eventos del punto de corte. Dado que Java permite dejar un punto de enlace de forma normal o por medio de una excepción, existen tres tipos de avisos after: `after`, `after returning` y `after throwing`.
Catch	Cuando durante la ejecución de las acciones asociadas con los eventos definidos en el punto de corte se ha elevado una excepción del tipo definido en la propia cláusula `catch`
Finally	Justo después de la ejecución de las acciones asociadas con los eventos del punto de corte, incluso aunque se haya producido una excepción durante la misma.

Around	Atrapan la ejecución de los métodos designados por el evento.

Ejemplos la declaración de un Avisos en AspectJ:

Before

```
before(): move() {
    System.out.println("me voy a mover");
}
```

After

```
after() returning: move() {
    System.out.println("ya me moví");
}
```

El ejemplo completo de un Aviso sería la unión del punto de corte y del aviso mismo:

```
pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))              ||
    call(void Point.setY(int))              ||
    call(void Line.setP1(Point))            ||
    call(void Line.setP2(Point));
before(): move() {
    System.out.println("me voy a mover");
}
after() returning: move() {
    System.out.println("ya me moví");
}
```

5. Sintaxis de declaraciones inter-tipo

Existen varias estructuras sintácticas para este tipo de declaraciones, según sea el tipo de miembro que se quiera agregar. Estas son:

```
1) abstract [ <Modificadores> ] <Tipo> <Tipo> . <Id> ( <Parametros> )
        [ throws <ListaDeTipos> ] ;

2) [ <Modificadores> ] <Tipo> <Tipo> . <Id> ( <Parametros> )
        [ throws <ListaDeTipos> ] { <Cuerpo> }

3) [ <Modificadores> ] <Tipo> . new ( <Parametros> )
        [ throws <ListaDeTipos> ] { <Cuerpo> }

4) [ <Modificadores> ] <Tipo> <Tipo> . <Id> [ = <Expresion> ] ;
```

El primer y el segundo caso corresponden a declaraciones de introducción de métodos abstracto y concreto, respectivamente. En estos casos la sintaxis es casi idéntica a la declaración de un método en Java. La única diferencia es que el nombre o identificador del método debe ir precedido del nombre de la clase en donde se desea introducir.

El tercer caso corresponde a la introducción de un constructor. La declaración cuenta con la particularidad de que en vez de requerir un identificador, como en el caso de un método, se debe utilizar la palabra reservada new, para indicar que se trata de un constructor. Al igual que el identificador de un método, la palabra debe ir precedida de la clase en donde se quiere introducir

el constructor.

El cuarto caso representa la introducción de un atributo. Sigue la forma de una declaración de atributos en Java, con la diferencia de anteponer el tipo al que se quiere introducir, antes del identificador del atributo.

Ejemplos

Consideremos el problema de expresar una capacidad compartida por varias clases existentes que ya son partes de una herencia de clases. En Java, crearíamos una interfaz que captura este nueva capacidad, y luego la agregaríamos a cada clase afectada un método que implemente esta interfaz.

AspectJ puede expresar el problema en un solo lugar, usando declaraciones entre-tipos. El aspecto declara los métodos y elemento que se necesitan implementar en la nueva capacidad, y lo asocia a los métodos y elementos de las clases ya existentes.

Por ejemplo, supongamos que queremos tener objetos Screen que observen los cambios de los objetos Point, donde Point es una clase ya existente. Podemos implementar esto mediante un aspecto declarando que la clase Point tiene un elemento de instancia, obserservers, que vigilan a los objetos Screen que están observando a los objetos Point.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}
```

El elemento observers es privado, entonces sólo PointObserving lo puede ver. Así observers son agregados o removidos con el método estático addObserver y removeObserver en el aspecto.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

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

Además, podemos definir un punto de corte changes que defina lo que queremos observar, y un aviso *after* defina lo que queremos hacer cuando se observe un cambio.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

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

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

    pointcut changes(Point p): target(p) && call(void
Point.set*(int));

    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }

    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
}
```

En ningún momento es necesario modificar el código de Screen o Point, y todos los cambios necesarios para soportar esta nueva capacidad son locales de éste aspecto.

6. Sintaxis de un aspecto:

```
<Aspecto> :=
        [ privileged ] [ <Modificadores> ] aspect <Identificador>
        [ extends <Tipo> ] [ implements <ListaDeInterfaces> ]
        [ <ClausulaDeInstanciación> ]
        { <Cuerpo> }
```

La palabra reservada "class" que define una clase en Java es reemplazada por "aspect" para el caso de un aspecto en AspectJ. De la misma forma que una clase, un aspecto debe tener un identificador único, que le dará el nombre al aspecto.

Los modificadores permitidos para un aspecto son similares a los que puede preceder a una clase:

- **`public`**: el aspecto es expuesto públicamente y por ende puede ser accedido desde cualquier punto en la jerarquía de paquetes.
- **`default` o package**: el modificador por defecto indica que sólo es posible acceder al aspecto dentro del paquete donde fue definido.
- **`final`**: al igual que el modificador de clases, indica que el aspecto no puede ser extendido.
- **`abstract`**: de la misma forma que en una clase, los aspectos abstractos tienen la característica de contener elementos abstractos, es decir, sólo definidos pero no implementados. En el caso de los aspectos, estos elementos pueden ser atributos, métodos o pointcuts.

La palabra reservada *privileged* es un modificador introducido por AspectJ

exclusivamente para los aspectos e indica que el aspecto tiene el privilegio de acceder a miembros de clases que comúnmente no podría, ya sean privados, protegidos o "default" de otros paquetes.

Como mencionamos anteriormente, un aspecto no puede poseer constructores debido a que no puede ser instanciado explícitamente. En cambio, el aspecto es automáticamente instanciado. El momento y forma de su instanciación depende de la política que se defina en la *<ClausulaDeInstanciación>*. Las distintas políticas disponibles son:

- **`issingleton()`**: política por defecto que indica que existirá sólo una instancia del aspecto.
- **`perthis(<Pointcut>)`**: una instancia es creada y asociada a cada objeto que actúa de iniciador al llegar al punto de ejecución definido por cualquiera de los puntos de enlace que forman el punto de corte que se recibe como parámetro.
- **`pertarget(<Pointcut>)`**: similar al caso anterior, pero con la diferencia que la instancia se asocia al objeto destino del punto de enlace perteneciente al punto de corte pasado como parámetro.
- **`percflow(<Pointcut>)`**: es la abreviación de "per control flow" (por cada control de flujo). El aspecto es instanciado al entrar en el flujo de ejecución definido por el punto de enlace. A diferencia de los dos casos anteriores, puede haber más de una instancia de un aspecto definido para el mismo objeto, si el flujo del programa pasa más de una vez por los puntos de enlace definidos en el punto de corte. En los últimos dos casos sólo puede haber una instancia del mismo aspecto asociada a un objeto.
- **`percflowbelow(<Pointcut>)`**: Variación del caso anterior en donde la instanciación se hace inmediatamente después de llegar a uno de los puntos de enlace del punto de corte.
- **`pertypewithin(<PatronTipo>)`**: A diferencia de los demás tipos de instanciación, pertypewithin recibe como parámetro una expresión del tipo <PatronTipo>. Dicha expresión representa un conjunto de Clases o Interfaces con ciertas características en común y será explicada en detalle más adelante. Al utilizar este tipo de instanciación, una instancia del aspecto es creada por cada Clase o Interface que concuerda con la expresión.

Ejemplo de un aspecto

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

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

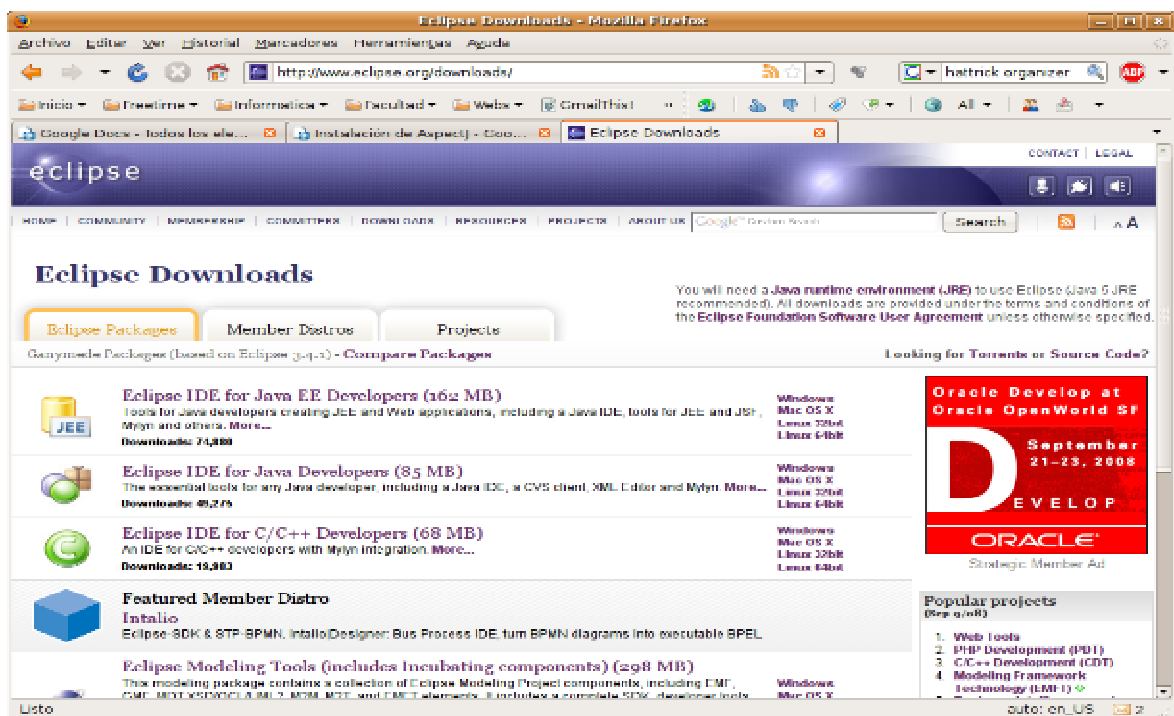
Instalación de AspectJ

1- Conseguir Eclipse

El primer paso para instalar AspectJ es instalar Eclipse.

Eclipse es una comunidad open source cuyos proyectos se centran en la creación de una plataforma de desarrollo abierta formada por frameworks extensibles, herramientas y runtimes para crear, desplegar y gestionar software en todo el ciclo de vida.

Para obtener la última versión de Eclipse, diríjase a la página oficial de descargas de Eclipse: http://www.eclipse.org/downloads/



Al ser un framework extensible, existen distintas versiones de descarga de Eclipse. Las más descargadas son:

- Eclipse IDE for Java EE Developers (163 MB)
- Eclipse IDE for Java Developers (85 MB)

Eclipse es un framework multiplataforma, por lo que existen archivos de instalación para distintos sistemas operativos, como ser Windows, Mac OS X y Linux.

Recuerde que para poder ejecutar Eclipse debe tener instalado previamente Java Runtime Environment (JRE).

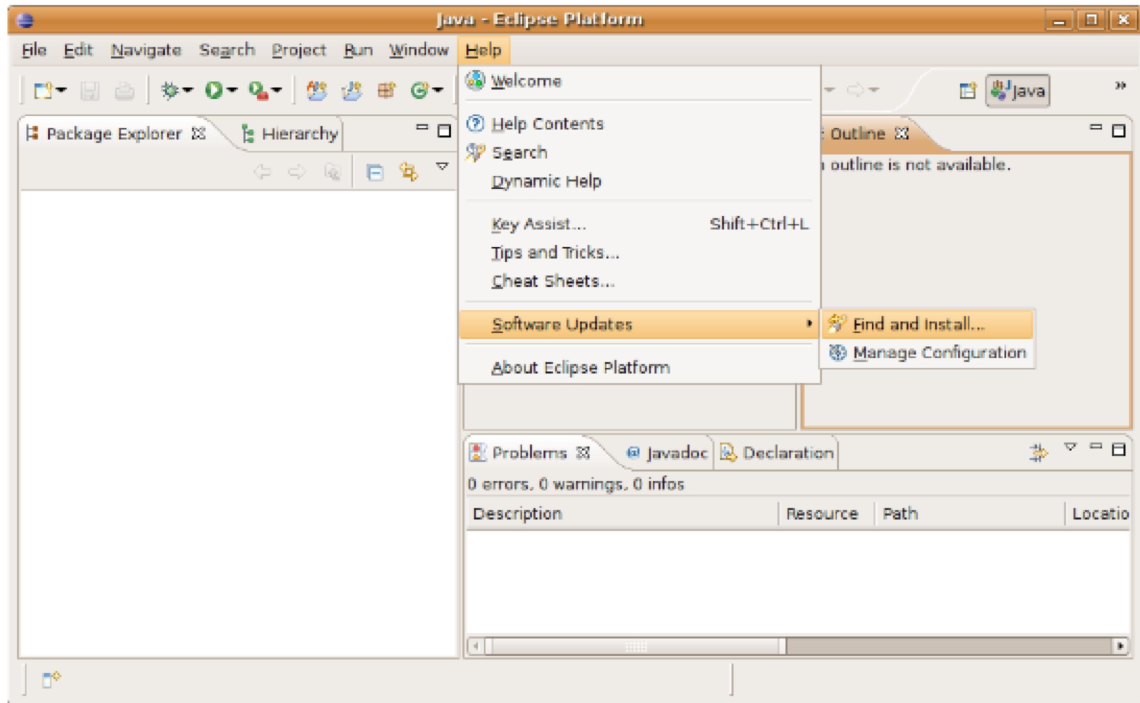
2- Instalar Eclipse bajo Ubuntu Linux

Para instalar Eclipse, se debe ejecutar el siguiente comando:

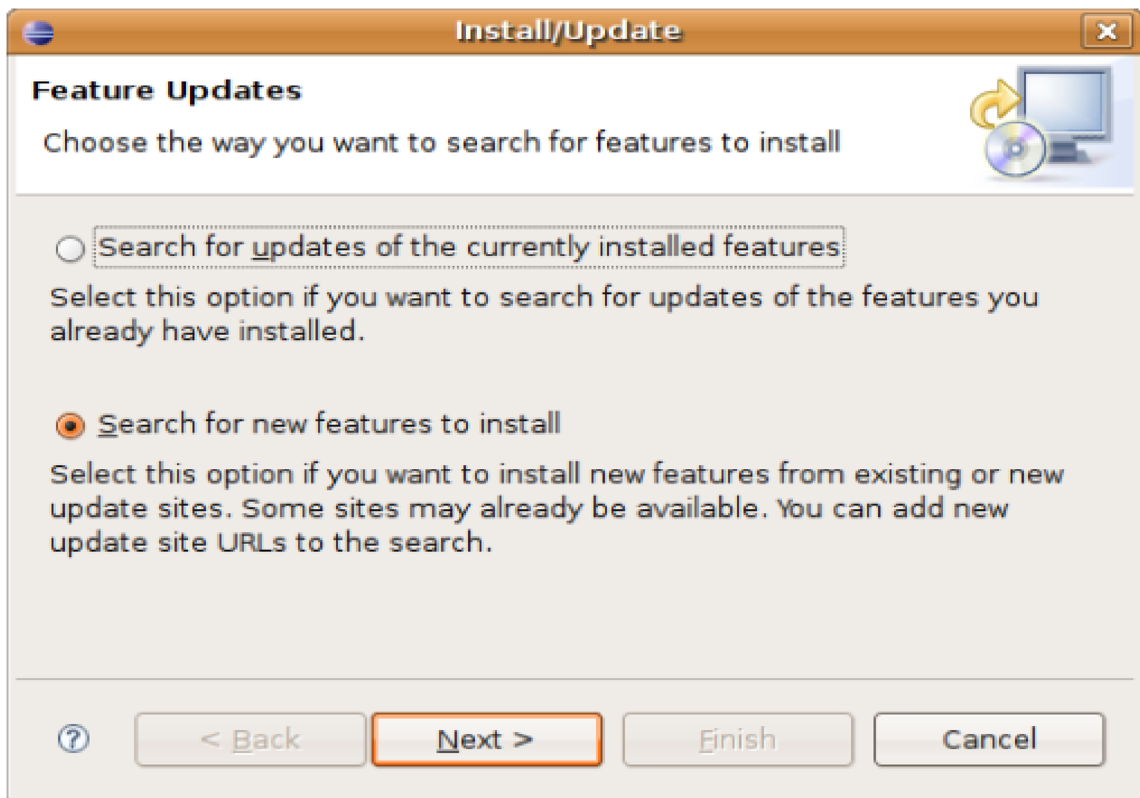
```
sudo apt-get install sun-java6-sdk eclipse
```

3- Instalar AspectJ

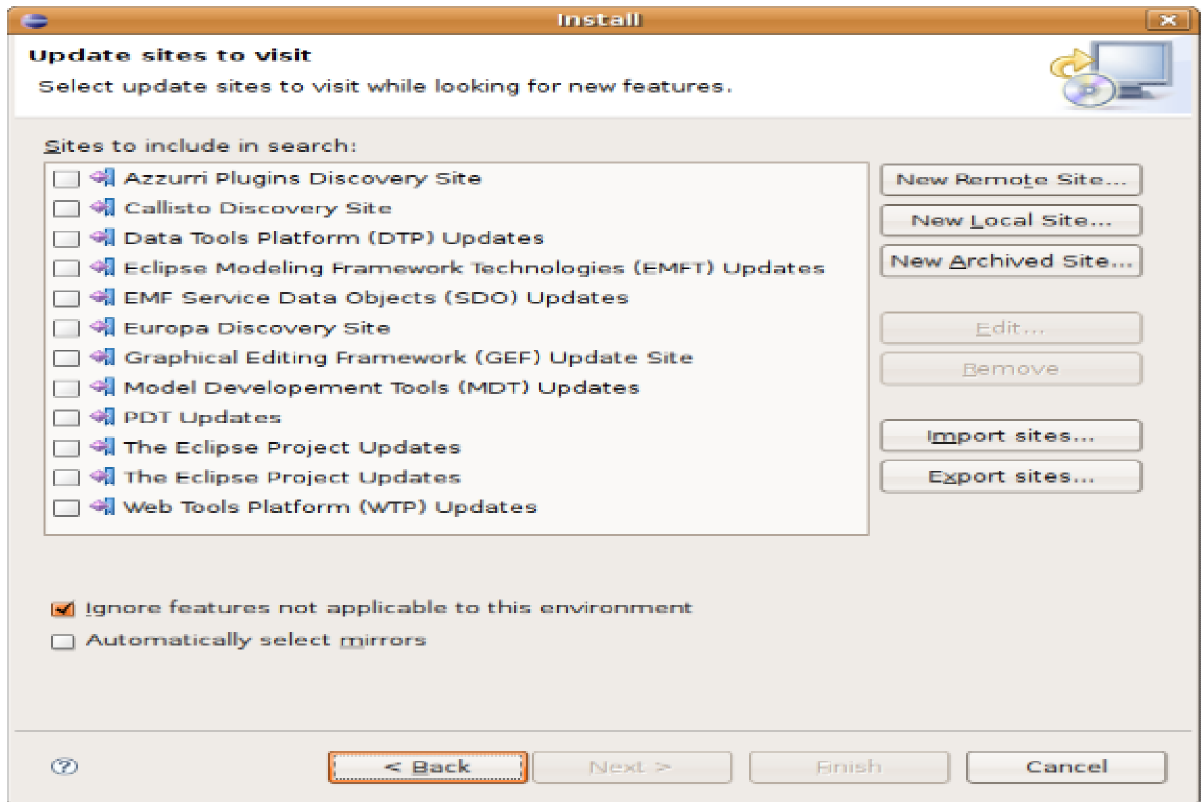
Una vez en Eclipse, para instalar AspectJ, se debe hacer click en el menú 'Help' y luego en 'Software Updates' y finalmente en 'Find and Install…'



En la pantalla 'Software Updates and Add-ons', hacer click en la pestaña 'Available Software' y luego en el botón 'Manage Sites…'

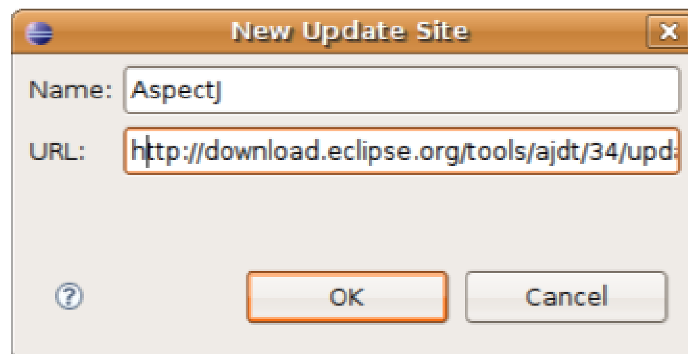


En la pantalla 'Install/Update' seleccionamos la opción 'Search for new features to install' y hacemos click en el botón 'Next >'.



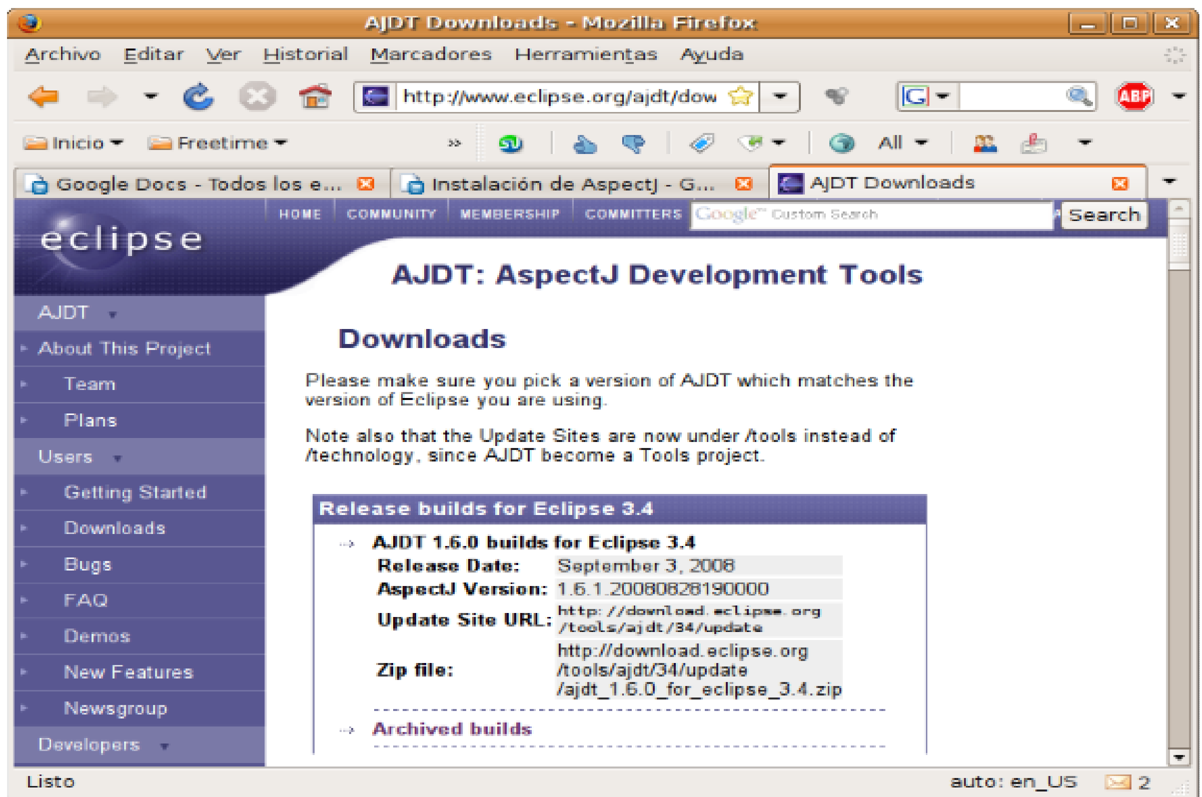
En la pantalla 'Install' hacemos click en el botón 'New Remote Sites...' y como datos agregamos:

- Name: AspectJ
- URL: http://download.eclipse.org/tools/ajdt/34/update

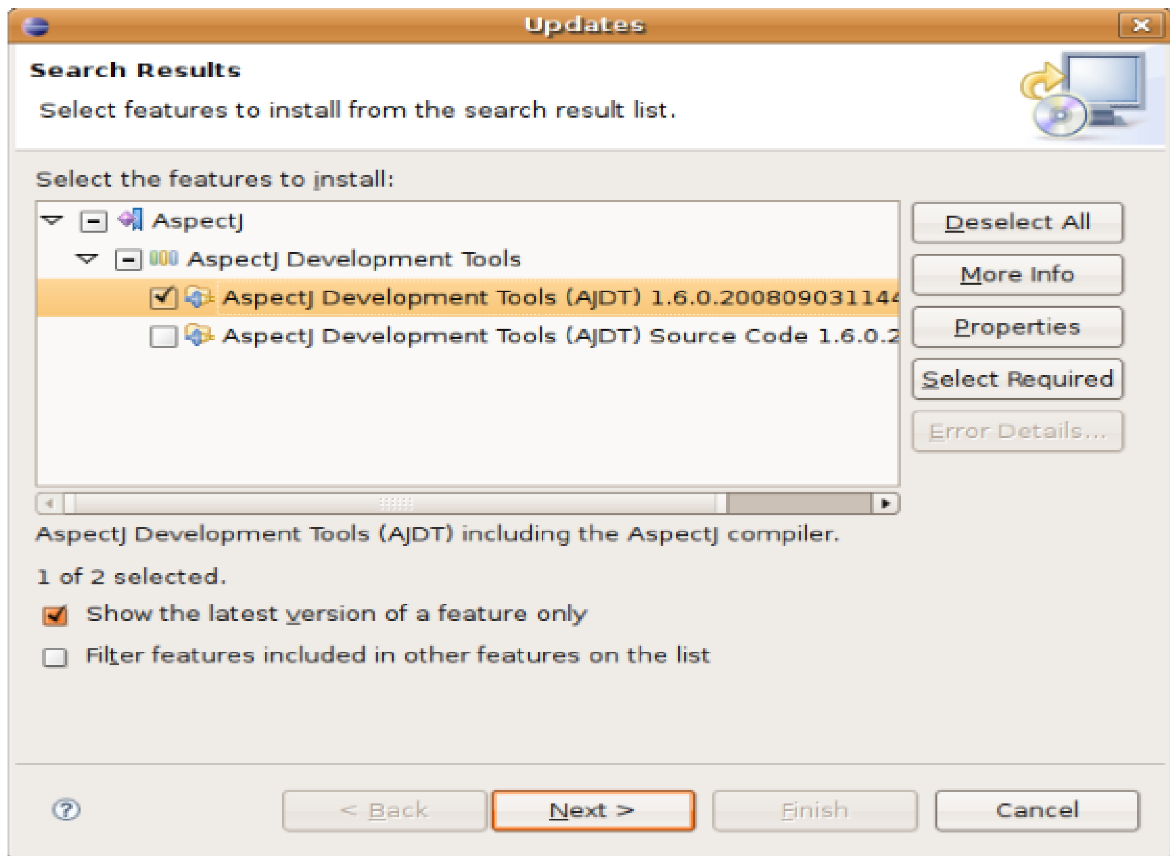


Luego hacemos click en 'OK' y posteriormente en 'Finish'

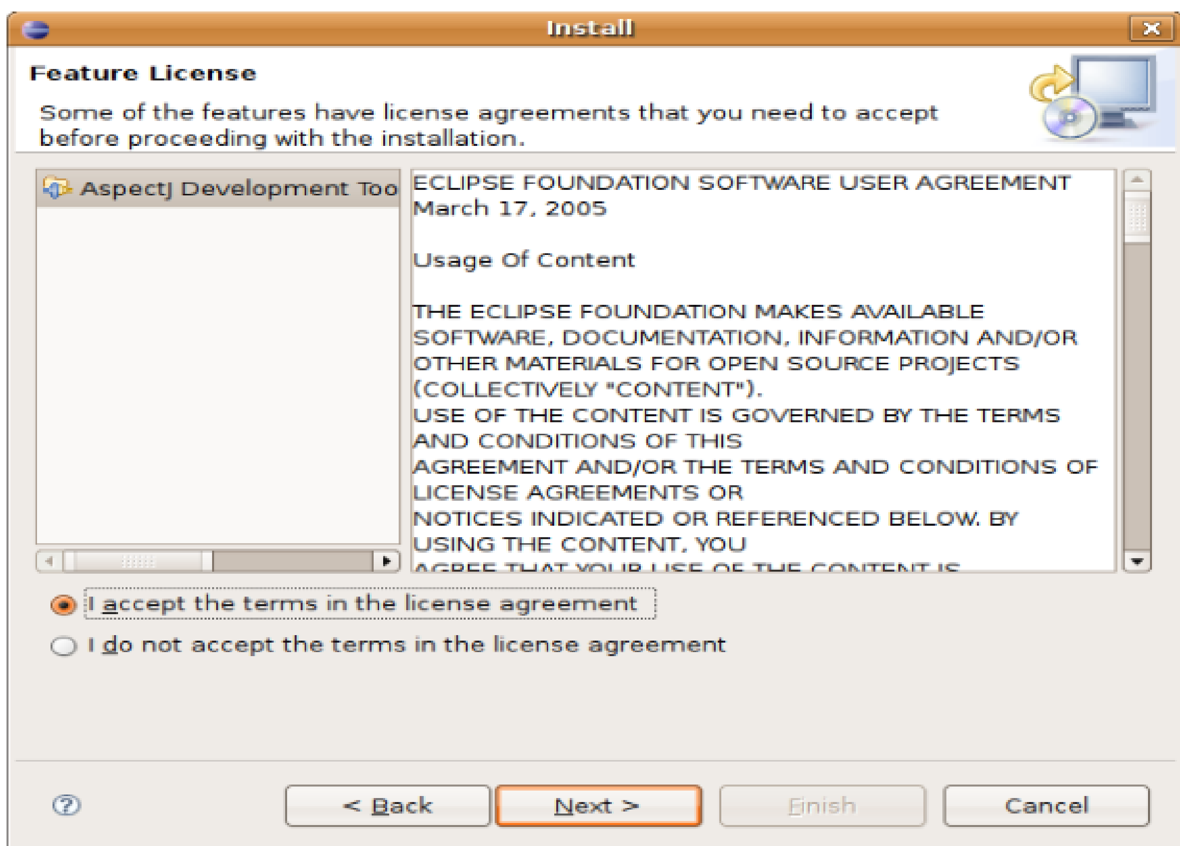
La URL de la última versión de AspectJ se consigue en http://www.eclipse.org/ajdt/downloads/



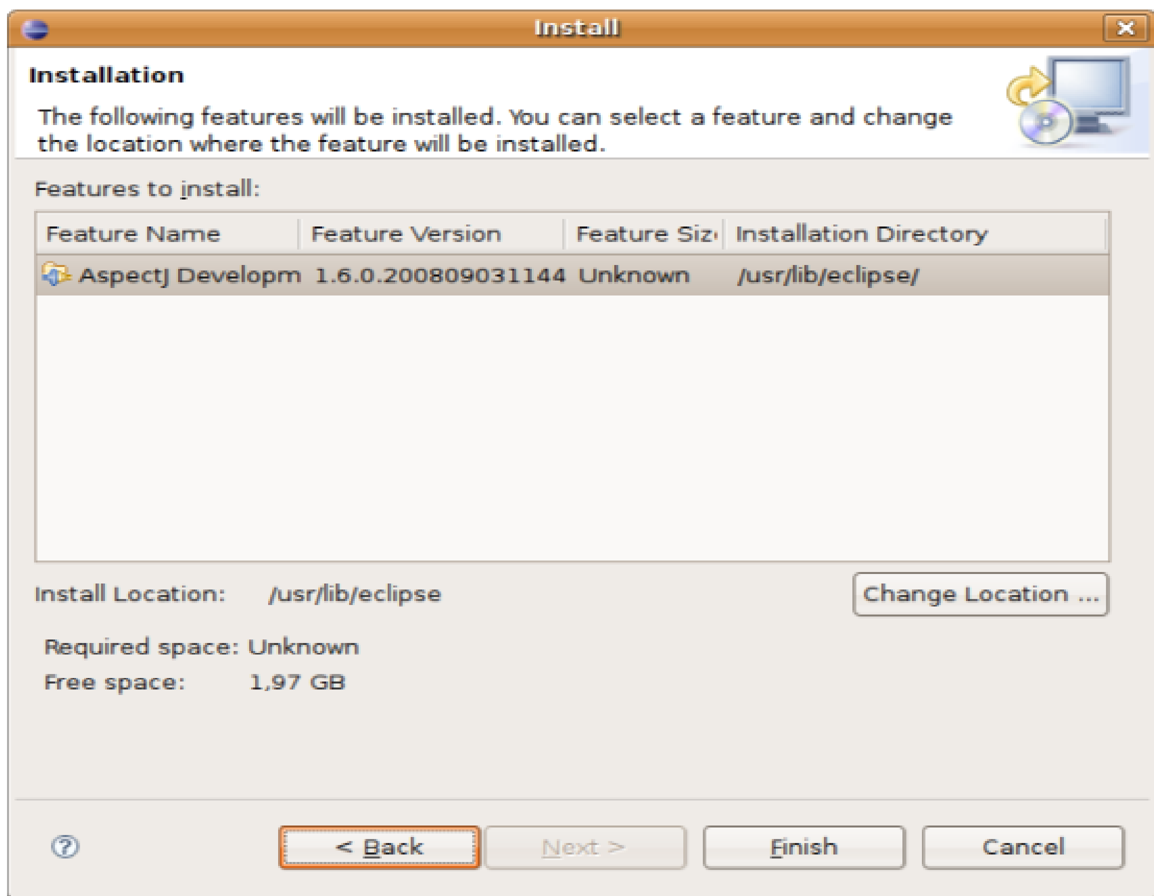
Aparecerá la ventana 'Updates' en donde debemos seleccionar la opción 'AspectJ Development Tools (AJDT) 1.6.x'. Verificamos que la casilla 'Show the latest version of a feature only' se encuentre marcada y hacemos click en 'Next >'



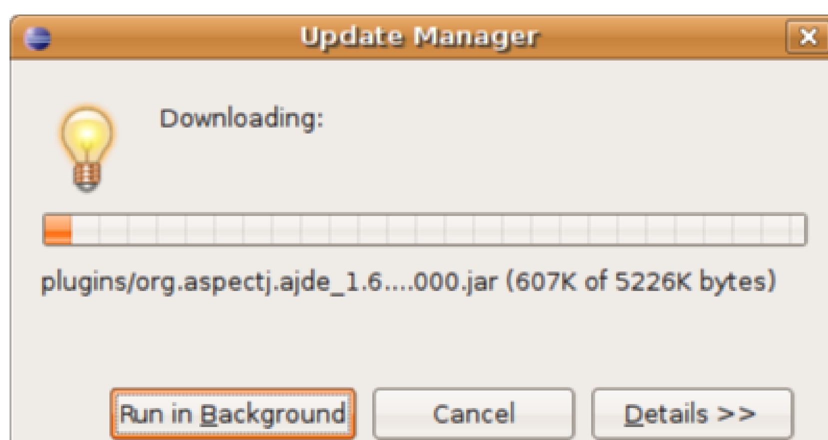
Aceptamos la licencia



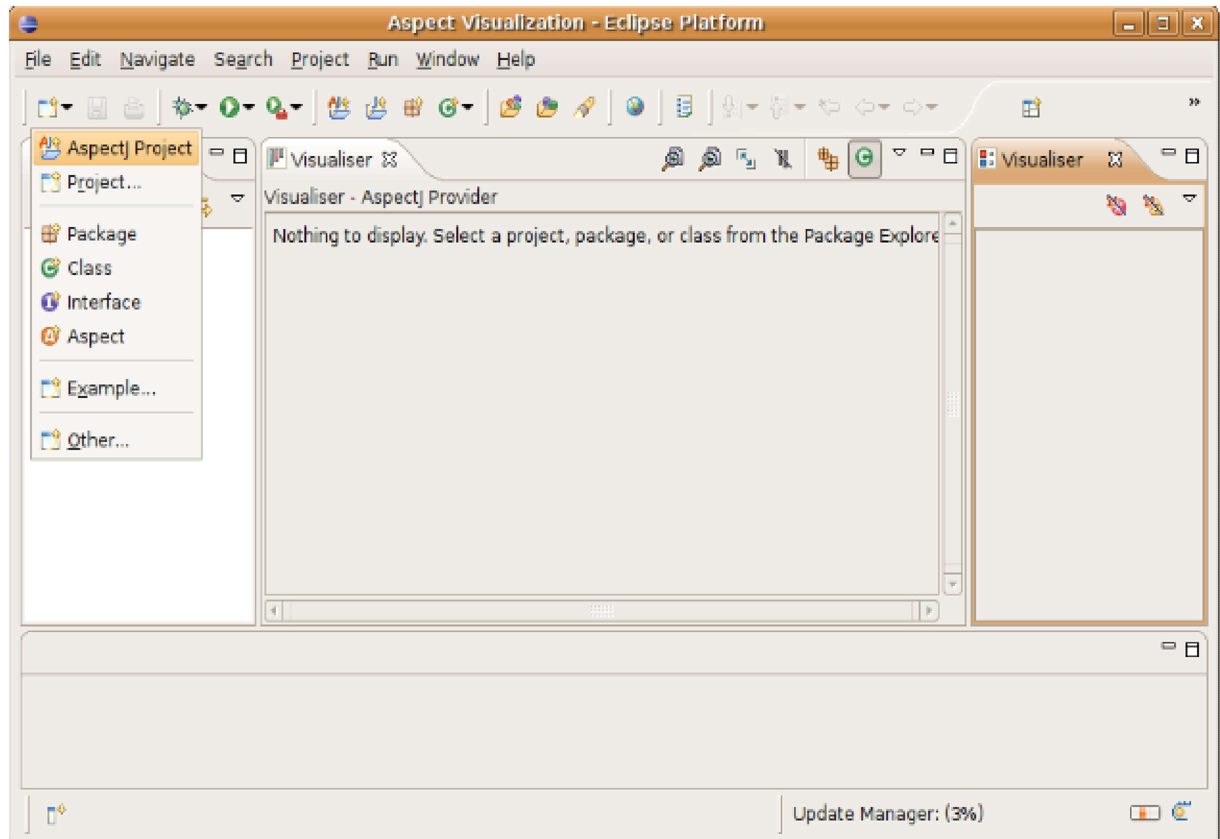
Se llega a la última pantalla:



Hacemos click en 'Finish' y Eclipse comenzará a bajar los archivos necesarios para configurar AspctJ.



Una vez instalados los archivos necesarios y habiendo reiniciado Eclipse, podremos crear un nuevo proyecto de AspectJ.



AspeCt C

Identificadores y palabras claves de la programación orientado a aspecto C

Aspecto orientado a C sigue la nomenclatura de las normas establecidas por la nueva Comunidad para de Desarrollo de software orientado a Aspectos y utiliza la siguiente terminología, identificadores y palabras clave

Para una detallada especificación de lenguaje y la sintaxis del aspecto orientado a lenguaje C, por favor, consulte las últimas aspecto orientado a C Especificación.

(http://www.aspectc.net/ac_manual.pdf)

La siguiente terminología es la utilizada en la presentación de Aspecto Orientado a C:

advice

El código a ejecutar cuando un join point es acompañado por un pointcut definido dentro de la declaración.

Join point

Un punto bien definido en el contexto de ejecución de un programa. La orientación a aspectos C se apoya en la función *join points*.

pointcut

Un lenguaje de extensión que representan uno o más join points.

Se definen las palabras claves para la orientación a aspectos en C.

after

La declaración de tipo advice, representa el código de avisos que debe ser ejecutado **después** del join point(s).

args

La declaración tipo *pointcut*, representa el join points cuyo tipos de argumentos son unidos por el pointcut especificado.

around

La declaración tipo advice, representa el código de avisos que debe ser ejecutado después

del join point(s), que por falta del *proceed()* , son evitados.

before

La declaración tipo advice, que representa que el código de aviso (advice) que debe ser ejecutado antes de unirse join point(s).

call

La declaración tipo *pointcut*, representa un join point de una llamada a una función cuyo prototipo es unido por el pointcut especificado.

cflow

La declaración tipo pointcut, representa a todos los join point debajo del flujo de control especificado en el pointcut.

execution

Una declaración tipo *pointcut*, representa la función del *join point* en ejecución acompañado por el pointcut especificado.

infile

La declaración tipo *pointcut*, representa el *join point* que existe en el archivo especificado.

infunc

La declaración tipo *pointcut*, representa el *join point* que existe en el archivo especificado.

pointcut

El nombre asociado a la definición de un *pointcut*. El nombre puede ser utilizado en las declaraciones del *advice* para referirse al nombre a las declaraciones del *pointcut*

proceed

Usado dentro de una función de aviso "circular", donde se identifica que el punto de unión original debería ser ejecutado

result

Una declaración tipo pointcut, representa el join point cuyo tipo de retorno es comparado por el pointcut especificado

this

Es usado dentro de funciones *advice*. es un puntero a una estructura y permite al código del

advice acceder a la información contextual de los join points. A través de el, el código del advice puede acceder a la función por medio del nombre "this-funcname" y al tipo de join point vía "this-kind"

Aspecto orientado a C-Ejemplos

Los siguientes son ejemplos de aspecto orientado a C en el contexto del OS/161 núcleo del sistema operativo.

Ejemplo: **after**

Después de llamar a *boot()* `función` dentro de `kmain()`, el advice imprime un mensaje.

```
    after(): call($ boot()) && infunc(kmain) {
          kprintf("aspect: after boot call in kmain function\n");
    }
```

Ejemplo: **args**

Descripción: Antes de llamar a una función que toma un puntero a una estructura semáforo, el advice comprueba el puntero valor para asegurarse de que no es nula.

```
before(struct semaphore * x): call($ $(...)) && args(x) {
    if(x == NULL) {
          panic("aspect: call function with null pointer\n");
    }
}
```

Ejemplo: **around**

Descripción: el *advice* sustituye el cuerpo de la función de órgano `null_fsync()` con un mensaje.

```
int around(): execution(int null_fsync(struct vnode *)) {
    kprintf("aspect: skip null_fsync function, do nothing.\n");
    return 0;
}
```

Ejemplo: **before**

Véase el ejemplo de "args".

Ejemplo: **call**

Ver ejemplos de "after" o "args".

Ejemplo: **cflow**

Pronto.

Ejemplo: **ejecución**

Véase el ejemplo de "around".

Ejemplo: **infile**

Descripción: El advice utiliza un mensaje a sustituir el cuerpo de todas las funciones cuyo nombre comience con "`null_`" y que se definen en un archivo con un nombre que comienza con "*device*".

```
 int around(): execution(int null_$(...)) && infile("device$.c") {
           kprintf("aspect:  skip  all  null_  functions  from  device.c
file.\n");
           return 0;
     }
```

Ejemplo: **infunc**

Véase el ejemplo de "after".

Ejemplo: **pointcut**

Descripción: Después de que el nombre de "`CallMalloc`" es asociado con una llamada a pointcut a la función `kmalloc()`, el nombre se puede utilizar siempre que un pointcut pueda ser utilizado para referirse a la definición del nombre del pointcut , como la declaración de un before o de un after advice.

```
 pointcut CallMalloc(): call(void * kmalloc(size_t));
 before(): CallMalloc() {
     kprintf("aspect: before call kmalloc\n");
 }

 after() : CallMalloc() {
     kprintf("aspect: after call kmalloc\n");
 }
```

Ejemplo: **proceed**

Descripción: las llamadas a *kmalloc()* se sustituirá por el *advice* "circular". Sin embargo, el original *kmalloc()* es todavía llamado dentro del advice "circular".

Si la asignación de memoria devuelve un puntero nulo, el kernel sale con un mensaje de error.

```
 void * around(): call(void * kmalloc(...)) {
     char * result;
     result = (char *)proceed();
     if(result == NULL) {
           panic("aspect: out of memory\n");
     }
 }
```

Ejemplo: **result**

Descripción: Después de cada llamada a `kmalloc` (), El advice comprueba el valor devuelto. Si se trata de un puntero nulo, el sistema sale con un mensaje de error. El ejemplo tiene el mismo efecto que el *`proceed()`* del ejemplo anterior.

```
after(void * s): call($ kmalloc(...)) && result(s) {
    char * result = (char *)s;
    if(result == NULL) {
          panic("aspect: out of memory\n");
     }
}
```

Ejemplo: **this**

Descripción: Después de una llamada a *`kmalloc()`*, El *advice* comprueba el valor devuelto. Si se trata de un puntero nulo, el sistema sale con un mensaje de error. Por otra parte, el *advice* imprime en qué función ocurrió que fallo el *`kmalloc()`*

- **Depuración de la programación en aspectos C**

Aspecto orientado a programa en C puede ser depuradas al igual que los programas ordinarios de C, porque el aspecto orientado a compilador de C genera archivos C ordinario.

Para ayudar al desarrollador a depurar se recomienda que el generado fuentes de C se pasan a través de las plataformas Unix a menudo disponible `guión` programa, que imprime bastante de la fuente de entrada C archivo.

A partir de aspecto orientado a CV 0,5 el número de línea y archivo de información de los archivos generados y los archivos de origen se mantienen en sinfonía.

Para versiones anteriores a 0,5 V `astillero` cartografía de la fuente para los archivos generados no es correcta. Al depurar el programa, el desarrollador tiene que paso a través de la fuente C generado archivos. A continuación presentamos una sesión de depuración para ilustrar cómo depurar.

Usando el aspecto orientado a "Hola Mundo" como un programa de ejemplo, podemos ilustrar una depuración de sesiones siguiente.

```
>acc hello.mc world.ac   <-- compila por acc
    >gcc -g hello.c world.c  <-- compila por gcc y crea un ejecutable
                                       depurable
    >gdb a.out <-- comienza el depurador GDB
    (gdb) break main         <-- set a breakpoint
    (gdb) run               <-- run a.out to the breakpoint
```

```
      Starting                                                  program:
/home/mwgong/temp/AspectC/ACC/src/working/example/a.out
      10      printf("Hello ");
      (gdb) list    <-- show source, it is the generated file, not the
original                     hello.mc
      5
      6       int main()  {int retValue_acc;
      7
      8
      9
      10      printf("Hello ");
      11      {
      12      world$1(); <-- inserted by acc
      13      }
      14      return retValue_acc;
      (gdb) next
      12      world$1();
      (gdb) step              <-step into the "world$1" function call
      world$1 () at world.c:6 <-inside world.c, which is generated from
world.ac
      6       printf(" World from AspectC ! \n"); }
      (gdb) next               <-- vuelve a hello.c
      Hello  World from AspectC !
      main () at hello.c:14
      14       return retValue_acc;      <-- este "return" es insertado
por acc
      (gdb) next
      15      }
      (gdb) next
      0x42015704 in __libc_start_main () from /lib/tls/libc.so.6
      (gdb) continue   <-- continua el programa para terminar
      Continuing.

      El programa termina con código 040
      (gdb)
```

Fechas de lanzamiento de las distintas versiones de AspeCt C

Tabla 9: Fechas de lanzamiento de las versiones de AspeCt C y su tamaño aproximado

Versión	Fecha de lanzamiento	Tamaño aprox. en MB
0.8 (Actualización)	15/06/2008	2.1
0.7	01/07/2007	21
0.6	11/05/2007	20
0.5	03/03/2007	19
0.4	20/01/2007	15
0.3	17/10/2006	3.4
0.2	08/09/2006	2.0
0.1	03/07/2006	1.4

Fuente: *Página oficial de descargas de AspeCt C*

http://research.msrg.utoronto.ca/ACC/Features#feature08

Instalación de AspeCt C

Como lo antes ya expuesto, este lenguaje surgió en el mes de septiembre del año 2008 y desde entonces, nuevas actualizaciones están disponibles para descargarlas desde el sitio oficial. (http://www.aspectc.net).

La última versión disponible es la 0.8 correspondiente a Enero del 2008 aunque ya están desarrollando la 0.9 que saldrá en los próximos meses.

Requerimientos para poder compilar y entretejer en AspeCt (Aspecto C)

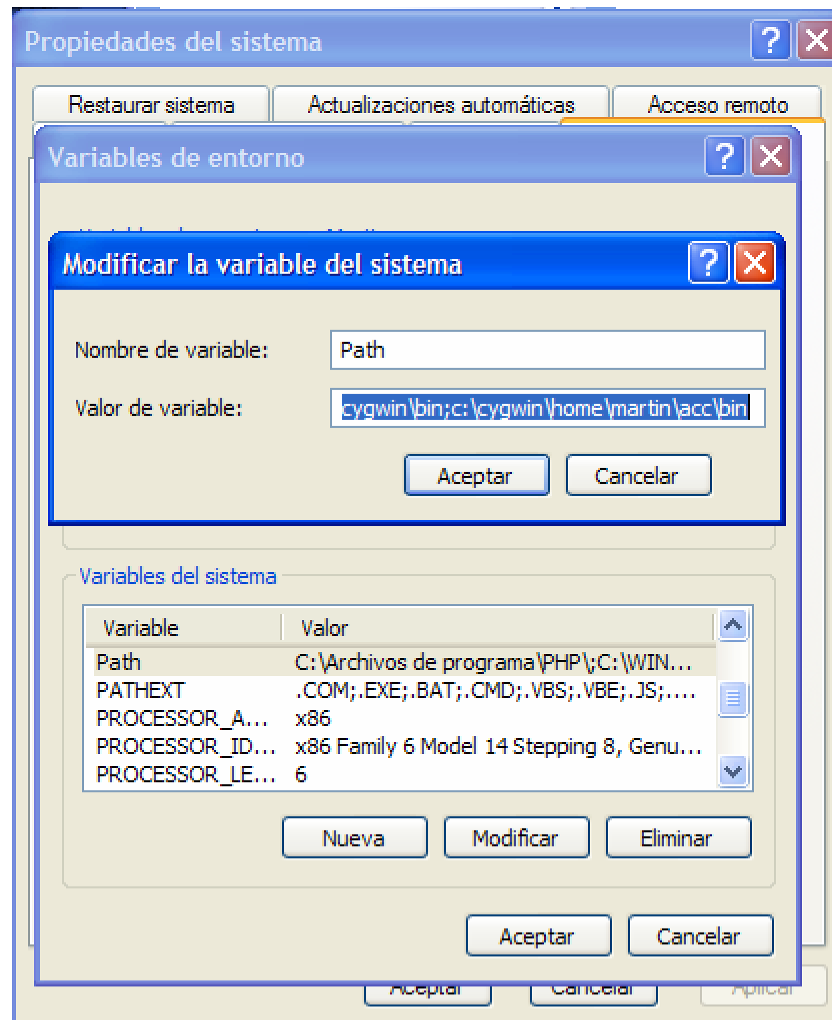
Plataformas:

- Linux
- Mac / OS X
- Solaris
- Windows (con Cygwin)

La plataforma que se uso es Windows (con Cygwin). Para poder utilizarlo, es necesario descargar de la página www.cygwin.com la aplicación.

Para descargar el compilador C, se debe dirigir al sitio: http://research.msrg.utoronto.ca/ACC/Download en donde se pueden encontrar la última versión 0.8 hasta el momento.

Una vez que descargamos esos dos programas, comenzaríamos a la configuración del compilador, para eso, deberíamos agregar a las variables de entorno (PATH) la ruta en donde se encuentra nuestro compilador, tanto el compilador C (gcc) como el de aspectos (acc).



Dentro del sitio oficial se pueden descargar un modelo completo con todos ejemplos disponibles.

Una vez instalado el Cygwin, deberíamos instalar el compilador y junto con el se instalan varios ejemplos.

Pasos para la instalación:

1- Ejecutar Cygwin
2- tar –xvf ACC.tar
3- cd ACC
4- make

Comienza la instalación, al finalizar muestra un resumen del resultado de la instalación.

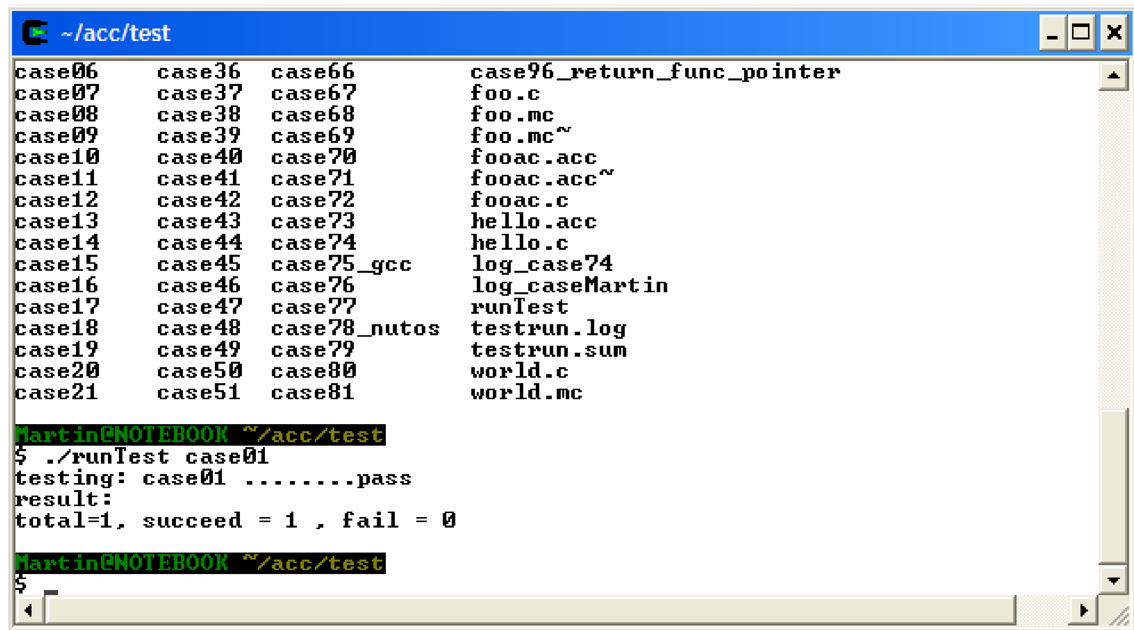
El compilador ACC compiler reside dentro del directorio src/working con el nombre ACC.

Prueba para comprobar la correcta instalación del modulo ACC

cd ACC/test

./runTest case01

El resultado que obtendríamos sería similar al siguiente:


```
case06     case36   case66         case96_return_func_pointer
case07     case37   case67         foo.c
case08     case38   case68         foo.mc
case09     case39   case69         foo.mc~
case10     case40   case70         fooac.acc
case11     case41   case71         fooac.acc~
case12     case42   case72         fooac.c
case13     case43   case73         hello.acc
case14     case44   case74         hello.c
case15     case45   case75_gcc     log_case74
case16     case46   case76         log_caseMartin
case17     case47   case77         runTest
case18     case48   case78_nutos   testrun.log
case19     case49   case79         testrun.sum
case20     case50   case80         world.c
case21     case51   case81         world.mc

Martin@NOTEBOOK ~/acc/test
$ ./runTest case01
testing: case01 ........pass
result:
total=1, succeed = 1 , fail = 0

Martin@NOTEBOOK ~/acc/test
$
```

Una vez comprobado esto, estamos en condiciones de comenzar a programar en AspeCt C.

PHPAspect

Aquí se encuentra la documentación disponible por los desarrolladores de PHPAspect, la cual aporta conceptos y ejemplos simples en la aplicación de aspectos.

Los lectores de este anexo deberán tener conocimiento sobre la POO y conocer PHP 5.

Primeramente se realiza una introducción de la POO con sus falencias.

En la POO por lo general las clases están contaminadas con cuestiones transversales entremezcladas con las funciones básicas (también llamado competencia lógica del negocio).

Estas cuestiones transversales suelen ser diversas: transacciones, seguridad, logging, persistencia, sesiones, etc.

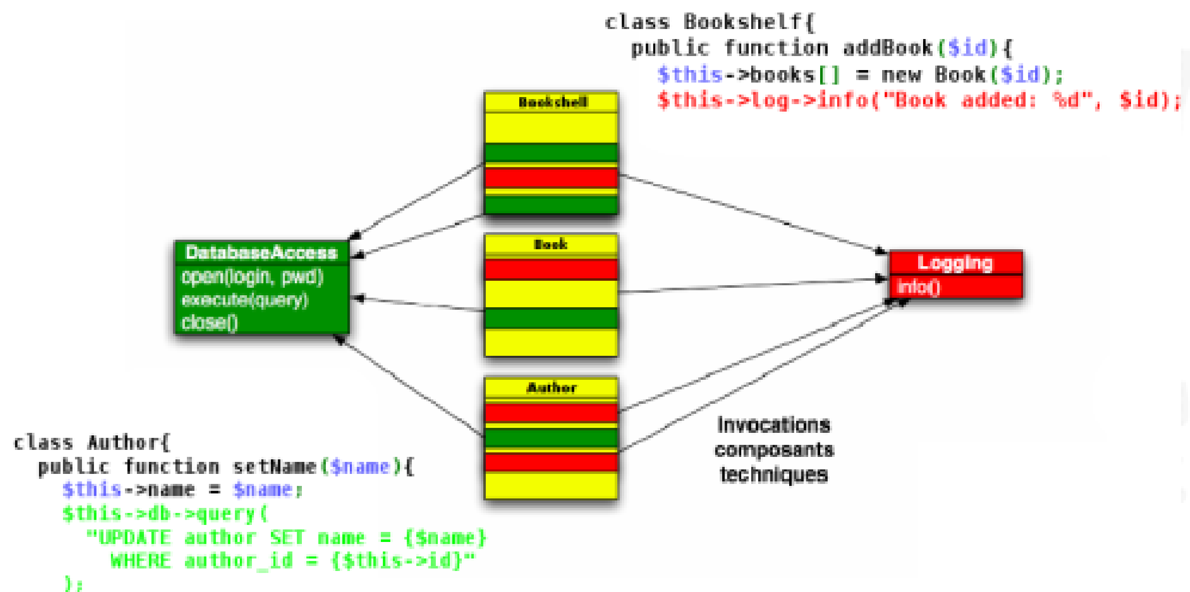


Figura 15: En POO, la lógica del negocio tiene una fuerte dependencia con los módulos técnicos[F,15].

Las falencias tienen como síntomas:

1. Código enredado.
2. Código disperso.
3. Reutilización de código.
4. Evolución del código.

En la ingeniería del software, el paradigma de programación orientada a aspectos intenta ayudar a los programadores en las competencias transversales. POA define los mecanismos para:

[F,15] Figura extraída de http://www.phpaspect.org/, Septiembre 2008.

- Escribir los aspectos como una nueva entidad del software.
- La técnica de tejido se refiere a la lógica del negocio.

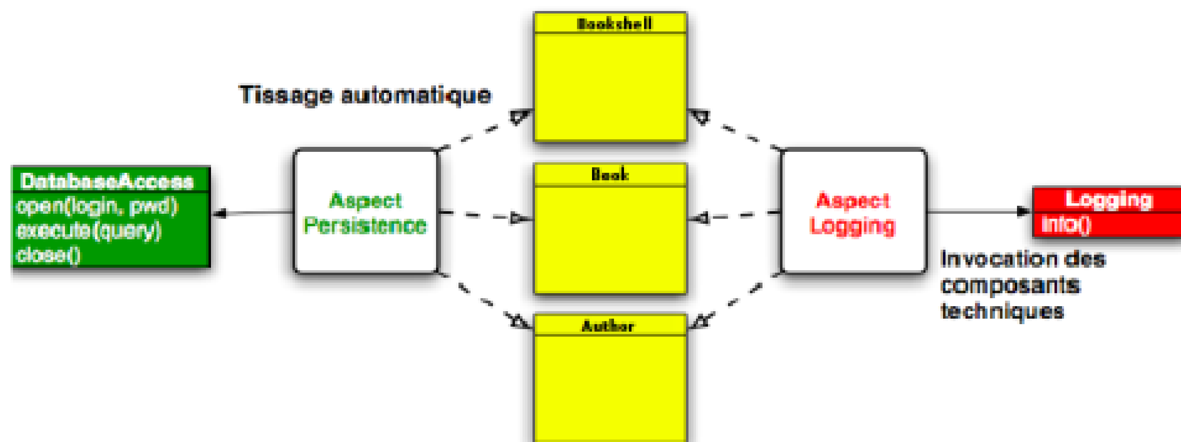


Figura 16: Inversión del control en POA[F,16]

Un aspecto tiene las siguientes partes:

- Joinpoints.
- Pointcuts.
- Advice.
- Atributos.
- Métodos.

La noción de joinpoint proporciona todos los mecanismos para identificar patrones siguiendo el flujo del programa.

[F,16] Figura extraída de http://www.phpaspect.org/, Septiembre 2008.

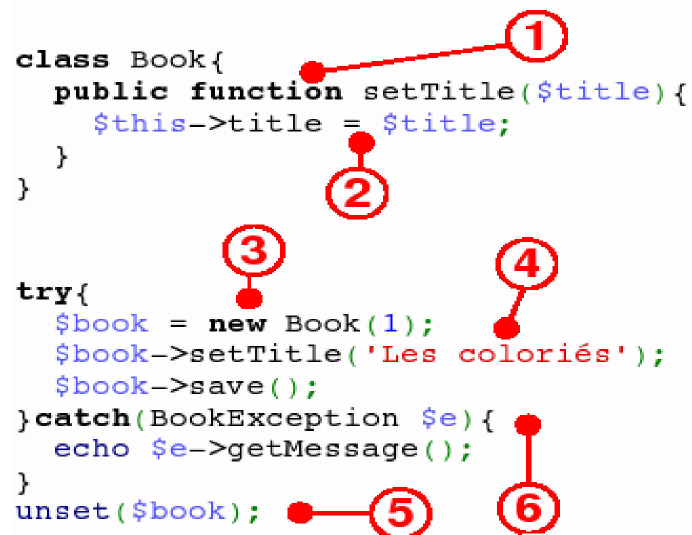


Figura 17: Ejemplo de JoinPoint

Explicación de las marcas de la Figura 17:

(1) Método de ejecución.

(2) Atributo de escritura/lectura.

(3) Objeto a construir.

(4) Llamada al método.

(5) Destrucción del objeto.

(6) Excepción.

Un poincut es una composición de joinpoints.

Los advice representan el código inmerso en los poincuts. Existen tres tipos de advice: Before, Around(instancia), After.

- **Escribiendo Aspectos**

PHPAspect extiende la sintaxis del compilador de PHP para introducir una nueva entidad "aspectos". Todos los aspectos deben ser escritos en un archivo con la extensión xx.aspect.php.

En un archivo aspect.php, todo código fuera de la entidad será ignorado por PHPAspect.

Ejemplo 1: Aspecto con PHPAspect

```
<?php
aspect MyCrosscuttingConcern{
   //Attributes
  //Methods
```

```
    //Poincuts
   //Advices
   //Inter-type declarations
}
//El código fuera del aspecto deberá ignorarse
?>
```

Los aspectos son las primeras entidades durante la ejecución, estos aspectos están representados por los métodos y atributos de las clases. Permite al usuario escribir métodos, atributos y constantes dentro de los aspectos.

Ejemplo 2: Los aspectos son las primeras entidades durante la ejecución.

```
<?php
aspect ObserverPattern{
   private $observers = array();
    public function addObserver(Observer $o){
       $this->observers[] = $o;
   }
}?>
```

1 – Joinpoints(Puntos de Enlace)

La versión actual de PHPAspect cuenta con 3 tipos de joinpoints:

- Ejecución de un Método(exec)
- Llamada a un Método(call)
- Objeto Construcción(new)

También hay 3 buscadores de joinpoints:

- Archivo
- Dentro de(within)
- Este(this)

Ejecución de un método

Este joinpoint identifica las ejecuciones de ciertos métodos en el programa.

La sintaxis de este punto de enlace es:

exec (method_modifiers className:: MethodName (ParametersNumber))

Ejemplo 3: Método de ejecución.

```
<?
```

```
phpaspect BankTransfer (
        / * Identifica todas las ejecuciones del método público "retirar" en la clase  Cuenta tomando
        un parámetro.* /
        poincut retirar: exec (public Cuenta :: retirar (1));
        / * Identifica todas las ejecuciones del método público "deposito" en la clase  Cuenta
        tomando  un parámetro.* /
        poincut deposito: exec (public Cuenta :: deposito (1 ));
}?>
```

Llamada a un método

Este joinpoint identifica las llamadas a ciertos métodos en el programa. La diferencia con el joinpont de ejecución es el contexto: este Joinpoint accede solo al contexto que se llama y no ve la ejecución de ese contexto.

La sintaxis de este punto de enlace es:

call(ClassName->MethodName(ParametersNumber))

Ejemplo 4: Llamada a un método con joinpoint

```
<?php
aspect Chekchout{
   /*Identifica todas las llamadas al método addItem de la clase Cart tomando 2 parámetros.*/
   poincut addItem:call(Cart->AddItem(2));
   /*Identifica todas las llamadas al método checkout de la clase Cart con todos los parámetros.*/
   poincut checkout:call(Cart->checkout(0));
}?>
```

Construcción Joinpoint

Este joinpoint identifica la construcción de un objeto en el programa.

La sintaxis de este punto de enlace es:

new(ClassName(ParametersNumber))

Ejemplo 5: Construcción Joinpoint

```
<?phpaspect Singleton{
 /*Identifica toda la construcción de los objetos Foo con cualquier parámetro.*/
 poincut newFoo:new(Foo(*));
/*Identifica toda la construcción de los objetos Bar con cualquier parámetro.*/
poincut newBar:new(Bar(*));
}?>
```

Archivo Joinpoint

Busca todos los tipos joinpoints en el archivo que tiene como referencia.

La sintaxis de este punto de enlace es:

file(ClassName)

Ejemplo 6: Archivo(File)

```
<?phpaspect Context{
//Busca todas las llamadas a la clase foo() en el archivo Bar.php
pointcut fooInBar:call(*->bar(*)) && file('Bar.php');
}?>
```

Dentro de(within)

Busca todos los tipos de joinpoints dentro de la clase de referencia.

La sintaxis de este punto de enlace es:

within(ClassName)

Ejemplo 7: Within

```
<?phpaspect Context{
 //Busca todas las llamadas a foo() en la clase Bar.
  pointcut fooInBar:call(*->bar(*)) && within(Bar);
}?>
```

2- PointCuts (Puntos de Corte)

Permiten agrupar joinpoints usando : || (or), && (and) and ! (not).

Ejemplo 8: se interceptan todas las construcciones en la instancia Cuenta y todas las ejecuciones de los métodos de la clase Cuenta, excepto el método login.

```
pointcut ensureAuthorization:(new(Account(0)) || exec(public Account::*(*))) && !exec(public
Account::login(2));
```

Ejemplo 9: se muestra la flexibilidad de la sintaxis de un pointcut.

```
<?phpaspect Security{
        pointcut accountConstruction:new(Account(0));
        pointcut accountExecution:exec(public Account::*(*));
        around(): (accountConstruction || accountExecution)
                and !exec(public Account::login(2)){
              //Ensure Authorization...
```

```
        }
}?>
```

3- Advice

Los Advices especifican el código que se ejecutará en los joinpoints que satisfacen a cierto pointcut.

<u>Before Advice</u>

Se ejecuta antes de un JoinPoint.

Ejemplo 10:

```
<?php
        before(): call(HelloWorld::say(0)){
                echo "Before saying Hello World\n";
}?>
```

<u>After Advice</u>

Se ejecuta después de un JoinPoint

Ejemplo 11:

```
<?php
        after(): call(HelloWorld::say(0)){
             echo "After saying Hello World\n";
  }?>
```

<u>Around Advice</u>

Sustituye la ejecución del JoinPoint. En un Around advice, la llamada a la función proceed() representa la ejecución del JoinPoint.

Ejemplo 12:

```
<?php
        around(): call(HelloWorld::say(0)){
                echo "50% chance to say Hello World\n";
                if(rand(0, 1)){
                        proceed();
                }
}?>
```

La sentencia return puede ser usada en el advice Around.

La expresión retornada reemplaza la ejecución del joinpoint.

Ejemplo 13: Sentencia return en el advice Around.

```
<?php
        around(): exec(Cart::getAmount(0)){
         //Give a 20% discount
        $amount = proceed();
         return $amount-0.2*$amount;
 }?>
```

Instalación de PHPAspect

PHPAspect está escrito en PHP5, por lo cual se debe contar con la versión correspondiente. Luego se pasará a instalar las siguientes extensiones y paquetes para el correcto funcionamiento del mismo:

Instalar Parse_Tree

PHPAspect necesita una extensión de pecl llamada Parse_Tree

01. para poder usar pecl, primero hay que instalar php-pear:

```
sudo apt-get install php-pear
```

02. para poder compilar la extension Parse_Tree antes hay que tener instalado el phpize que se encuentra en el paquete php5-dev:

```
sudo apt-get install php5-dev
```

03. instalar la extension Parse_Tree:

```
sudo pecl install -f Parse_Tree
```

#04. editar el archivo php.ini y agregar:

```
extension=parse_tree.so;
sudo nano /etc/php5/cli/php.ini
```

Instalar PHP_Beautifier

```
sudo pear install PHP_Beautifier-0.1.14
```

Instalar Console_Getopt

```
sudo pear install Console-Getopt-1.2.3
```

indica que ya esta instalado

Descargar XSLTPROCESSOR

```
sudo apt-get install php5-xsl
```

Reiniciar

```
sudo /etc/init.d/apache2 restart
```

Descargar phpAspect

```
wget http://phpaspect.googlecode.com/files/phpaspect-0.1.1.tar.gz
```

La instalación explicada anteriormente se realizó en el Sistema Operativo Linux.

Se intento instalar PHPAspect en el Sistema Operativo WindowsXp y no se logró su funcionamiento.

Luego para realizar el tejido del aspecto, osea generar el nuevo archivo con aspecto se debe hacer mediante consola: xx-es la carpeta raíz que contiene el archivo a tejer-

```
cd www/phpaspect
php ./phpaspect.php -b ~/www/xx/src ~/www/xx/src ~/www/xx/bin
```

aoPHP

Las sintaxis utilizadas para programar aspectos son las siguientes:

Pointcut:
joinpoint(signature)
exec(FUNC(PARAMS))
execr(FUNC(PARAMS))
ie: exec(foo($x))
ie: execr(bar($x)) | execr(foo($x))

Named Pointcut:
pointcut Name = PC1;
pointcut Name = PC1 | PC2 | PC3;
ie: pointcut testpc = exec(foo($x));
ie: pointcut testpc = execr(bar($x)) | execr(foo($x));

Advice Definition:
advice(): pointcut{ ... }
ie: before(): execr(bar($x)) | execr(foo($x)) { ... }
ie: after(): testpc { ... } // Using a Named Pointcut

PHP File Call:
ie: <?aophp filename="file1.aophp" debug="off"
ie: <?aophp filename="file1.aophp,file2.aophp" debug="off

La extensión de un archivo que se refiera a aspectos será **xx.aophp**

Indice de Figuras

Figura 1: Programa Tradicional vs Programa Orientado a Aspectos.

Figura 2: Estructura de un lenguaje de programación general.

Figura 3: Estructura de un lenguaje de programación orientado a aspectos.

Figura 4: Aspecto como un elemento del metamodelo UML derivado de Clasificador.

Figura 5: Representación de un aspecto en UML extendido.

Figura 6: tipos de relaciones del metamodelo de UML.

Figura 7: Notación de la relación aspecto-clase.

Figura 8: Estructura de las clases tejidas.

Figura 9: Composición de AspectJ.

Figura 10: Proceso de Compilación en AspectC.

Figura 11: Cadena de tejido de PHPAspect.

Figura 12: Declaración de Aspectos usando la librería AOP para PHP y clase base usando aspectos. El "aspecto" es solo una librería que es invocada en el código fuente.

Figura 13: Clases ejemplo AspectJ.

Figura14: Evolución de popularidad de los 10 primeros lenguajes de programación más populares desde mediados del año 2001 hasta Julio de 2008. Fuente: TIOBE.

Figura 15: En POO, la lógica del negocio tiene una fuerte dependencia con los módulos técnicos.

Figura 16: Inversión del control en POA.

Figura 17: Ejemplo de JoinPoint.

Indice de Tablas

Tabla 1: Lenguajes de Programación Orientados a Aspectos de Propósitos General.

Tabla 2: Archivos generados por el compilador orientado a aspectos.

Tabla 3: Comparación de LDCF para cada una de las clases en las tres partes de nuestro ejemplo.

Tabla 4: Fechas de lanzamiento de las versiones de AspectJ y su tamaño aproximado.

Tabla 5: Top 10 de popularidad de los lenguajes de programación para el mes de Julio de 2008 medido en porcentaje.

Tabla 6: Sintaxis básica de los puntos de corte según la categoría de los puntos de unión.

Tabla 7: Sintaxis básica de los puntos de corte.

Tabla 8: Tipo de Avisos utilizados en AspecJ.

Tabla 9: Fechas de lanzamiento de las versiones de AspeCt C y su tamaño aproximado.

Referencias

[KL,1996] Karl Lieberherr, *Adaptive Object-Oriented Software, The Demeter Method*, PWS Publishing Company, 1996.

[KL,1998] K. Lieberherr, I. IIolland, A. Riel, *Object-Oriented Programming:An Objective Sense of Style*, College of Computer Science Northeastern University, 1988.

[WH,1995] Walter L. Hürsch and Cristina Videira Lopes, Separation of Concerns, in College of Computer Science, Northeastern University Boston, February 24 1995.

[RA,2001] Diccionario de la Real Academia Española - Vigésima segunda edición - http://www.rae.es/, Consultado Agosto 2008.

[GK,1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, *Aspect-Oriented Programming*, Xerox Palo Alto Research Center, 1997.

[GO,2004] Graham O'Regan, ONJava.com - Introduction to Aspect-Oriented Programming, O'Reilly. Enero 2004.

[RH,2005] Rob Harrop y Jan Machacek, *Pro Spring*, Apress, 2005.

[FA,2003] Fernando Asteasuain, Bernando E. Contrares, Elsa Estévez, Pablo R. Fillotrani, *Programación Orientada a Aspectos: Metodología y Evaluación*, Departamento de Ciencias e Ingeniería de la computación, Universidad Nacional del Sur, Año 2003.

[JL,1999] John Lamping, *"The role of the base in aspect oriented programming"*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP) workshops, 1999.

[TH,1999] Timothy Highley, Michael Lack, Perry Myers, *Aspect-Oriented Pragramming: A Critical Analysis of a new pragramming paradigm*, University of Virginia, Departament of computer Science, Technical Report CS-99-29, Mayo 1999.

[MB,1998] MacLennan B, Principles of programming languages, 3º Edicion, John Wiley & Sons, 1998.

[GK,1997] Gregor Kickzales, John Lamping, Cristina Lopes, Chris Maeda, Anurag Mendhekar, Gail Murphy, *"Open Implementation Design guidelines"*, Proceedings of the 19th

International Conference on Sofware Engineering, (Boston, MA), Mayo de 1997.

[PK,2000] Peter Kenens, Sam Michiels, Frank Matthijs, Bert Robben, Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, *"An AOP Case with Static and Dynamic Aspects"*, Departament of Computer Science, K.U.Leuven, Bélgica.

[GC,1999] Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga, "Language Support for Evolvable Software: An Initial Assessment of Aspect-Oriented Programming", Proceedings of the International Workshop on the Principles of Software Evolution, IWPSE99, Julio 1999.

[JL,1997] J. Irwin, J.-M.Loingtier, J. R. Gilbert, and G. Kiczales, *"Aspect-oriented programming of sparse matrix code"*, Lecture Notes in Computer Science, 1997.

[AM,1997] Mendhekar A., Kiczales G. and Lamping J, *"RG: A Case-Study for Aspect-Oriented Programming"*, Xerox PARC, 1997.

[CL.1997] Cristina Lopes, *"A Language Framework for Distributed Programming, Northeastern University"*, Noviembre 1997.

[SM,2005] Salvador Manzanares Guillén, "Programación Orientada a Aspectos, una experiencia práctica con AspectJ", Facultad de Informática de la Universidad de Murcia, Junio 2005.

[W,2008] Wikipedia, Aspect-oriented programming, http://en.wikipedia.org/wiki/Aspect-oriented_programming, 2008, Consultado en Agosto 2008.

[RL,2002] Ramnivas Laddad, *"I want my AOP"*, JavaWorld, Enero 2002.

[LD,1996] Lea D, *Concurrent Programming in Java: design principles and patterns, Addison-Wesley*, 1996.

[SR,2008] Aspect Oriented C - http://research.msrg.utoronto.ca/ACC/WebHome/, Consultado Agosto 2008.

[RF,2000] Robert E. Filman and Daniel P. Friedman, *"Aspect-Oriented Programming is Quantification and Obliviousness"*, Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis.

[JS, 1999] Junichi Suzuki, Yoshikazu Yamamoto. Extending UML with Aspect: Aspect

Support in the Design Phase. 3er Aspect-Oriented Programming (AOP) Workshop at ECOOP'99.

[OMG,2007] OMG. Unified Modeling Language Specification v.2.1.2, 2007.

[CH,1993] Shyam R. Chidamber, Chris F. Kemerer. *A Metrics suite for Object Oriented design*. M.I.T. Sloan School of Management E53-315. 1993.

[GD,2000] Daniela Glasberg, Khaled El Emam, Walcelio Melo, Nazim Madhavji: Validating Object-Oriented Design Metrics on a CommercialJava Application. 2000.

[HS,2003] Houari A. Sahraoui, Robert Godin, Thierry Miceli: Can Metrics Help Bridging the Gap Between the Improvement of OO Design Quality and Its Automation.