PROGRAMACIÓN ORIENTADA A ASPECTOS

Análisis del paradigma

Fernando Asteasuain
Bernardo Ezequiel Contreras

Tesis de Licenciatura

Departamento de Ciencias e Ingeniería de la Computación

UNIVERSIDAD NACIONAL DEL SUR

Octubre de 2002

Este trabajo fue presentado como tesis de final de carrera en el Departamento de Ciencias e Ingeniería de la Computación de la Universidad Nacional del Sur para obtener el título de Licenciado en Ciencias de la Computación.

La dirección del trabajo estuvo a cargo de la Licenciada Elsa Estevez y el Dr. Pablo Fillottrani y fue finalizado en octubre de 2002.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGO

ÍNDICE DE BNF

ÍNDICE DE ILUSTRACIONES Y TABLAS

1. Introducción

La ingeniería del software ciertamente ha evolucionado desde sus comienzos. Al principio, se tenía un código en el que no existía la separación de conceptos; datos y funcionalidad se mezclaban sin una línea que los dividiera claramente. A esta etapa se la conoce como código spaghetti, debido a que se tenía una maraña entre datos y funcionalidad similar a la que se forma al comer un plato de esta comida italiana.

A medida que la ingeniería de software fue creciendo, se fueron introduciendo conceptos que llevaron a una programación de más alto nivel: 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, genericidad, herencia. Como vemos, los progresos más importantes se han obtenido aplicando tres principios, los cuales están estrechamente relacionados entre sí: abstracción, encapsulamiento, y modularidad. Esto último consiste en la noción de descomponer un sistema complejo en subsistemas más fáciles de manejar, siguiendo la antigua técnica de "dividir y conquistar".

Un avance importante lo introdujo la Programación Orientada a Objetos (POO), donde se fuerza el encapsulamiento y la abstracción, a través de una unidad que captura tanto funcionalidad como comportamiento y estructura interna. A esta entidad se la conoce como clase. La clase hace énfasis tanto en los algoritmos como en los datos. La POO está basada en cuatro conceptos [17]:

- Definición de tipos de datos abstractos.
- Herencia
- Encapsulamiento.
- Polimorfismo por inclusión.

La herencia es un mecanismo lingüístico que permite definir un tipo de dato abstracto derivándolo de un tipo de dato abstracto existente. El nuevo tipo definido "hereda" las propiedades del tipo padre [17].

Ya sea a través de la POO o con otras técnicas de abstracción de alto nivel, 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, debido a que atraviesan todo el sistema, o varias parte de él (crosscutting concerns). Algunos de estos conceptos son: sincronización, manejo de memoria, distribución, chequeo de errores, profiling, seguridad o redes, entre otros. Así lo muestran los siguientes ejemplos:

1) Consideremos una aplicación que incluya conceptos de seguridad y sincronización, como por ejemplo, asegurarnos que dos usuarios no intenten acceder al mismo dato al mismo tiempo. Ambos conceptos requieren que los programadores escriban la misma funcionalidad en varias partes de la aplicación. Los programadores se verán forzados a recordar todas estas partes, para que a la hora de efectuar un cambio y / o una actualización puedan hacerlo de manera uniforme a través

de todo el sistema. Tan solo olvidarse de actualizar algunas de estas repeticiones lleva al código a acumular errores [4].

2) Manejo de errores y de fallas: agregar a un sistema simple un buen manejo de errores y de fallas requiere muchos y pequeños cambios y adiciones por todo el sistema debido a los diferentes contextos dinámicos que pueden llevar a una falla, y las diferentes políticas relacionadas con el manejo de una falla [8].

3) En general, los aspectos en un sistema que tengan que ver con el atributo performance, resultan diseminados por todo el sistema [8].

Las técnicas de implementación actuales tienden a implementar los requerimientos usando metodologías de una sola dimensión, forzando los requerimientos a ser expresados en esa única dimensión. Esta dimensión resulta adecuada para la funcionalidad básica y no para los requerimientos restantes, los cuales quedan diseminados a lo largo de la dimensión dominante. Es decir, que mientras el espacio de requerimientos es de n-dimensiones, el espacio de la implementación es de una sola dimensión. Dicha diferencia resulta en un mapeo deficiente de los requerimientos a sus respectivas implementaciones. Algunos síntomas de este problema pueden ser categorizados de la siguiente manera [18]:

1. *Código Mezclado* (Code Tangling): En un mismo módulo de un sistema de software pueden simultáneamente convivir más de un requerimiento. Esta múltiple existencia de requerimientos lleva a la presencia conjunta de elementos de implementación de más de un requerimiento, resultando en un *Código Mezclado*.
2. *Código Diseminado* (Code Scattering): Como los requerimientos están esparcidos sobre varios módulos, la implementación resultante también queda diseminada sobre esos módulos.

Estos síntomas combinados afectan tanto el diseño como el desarrollo de software, de diversas maneras [18]:

- Baja correspondencia: La implementación simultánea de varios conceptos oscurece la correspondencia entre un concepto y su implementación, resultando en un pobre mapeo.
- Menor productividad: La implementación simultánea de múltiples conceptos distrae al desarrollador del concepto principal, por concentrarse también en los conceptos periféricos, disminuyendo la productividad.
- Menor reuso: Al tener en un mismo módulo implementados varios conceptos, resulta en un código poco reusable.
- Baja calidad de código: El *Código Mezclado* produce un código propenso a errores. Además, al tener como objetivo demasiados conceptos al mismo tiempo se corre el riesgo de que algunos de ellos sean subestimados.
- Evolución más dificultosa: Como la implementación no está completamente modularizada los futuros cambios en un requerimiento implican revisar y modificar cada uno de los módulos

donde esté presente ese requerimiento. Esta tarea se vuelve compleja debido a la insuficiente modularización.

Tenemos entonces que las descomposiciones actuales no soportan una completa separación de conceptos, la cual es clave para manejar un software entendible y evolucionable. Podemos afirmar entonces que las técnicas tradicionales no soportan de una manera adecuada la separación de las propiedades de aspectos distintos a la funcionalidad básica, y que esta situación tiene un impacto negativo en la calidad del software.

Como respuesta a este problema nace la Programación Orientada a Aspectos (POA). La POA permite a los programadores escribir, ver y editar un aspecto diseminado por todo el sistema como una entidad por separado, de una manera inteligente, eficiente e intuitiva.

La POA es una nueva metodología de programación que aspira a soportar la separación de las propiedades para los aspectos antes mencionados. Esto implica separar la funcionalidad básica y los aspectos, y los aspectos entre sí, a través de mecanismos que permitan abstraerlos y componerlos para formar todo el sistema.

La POA es un desarrollo que sigue a la POO, y como tal, soporta la descomposición orientada a objetos, además de la procedimental y la funcional. Sin embargo, la programación orientada a aspectos no es una extensión de la POO, ya que puede utilizarse con los diferentes estilos de programación mencionados anteriormente.

Teniendo en cuenta nuevamente la forma en que la ingeniería de software ha crecido, siempre de la mano de nuevas formas de descomposición que implicaron luego nuevas generaciones de sistemas, es válido preguntarnos si también de la mano de la POA nacerá una nueva generación de sistemas de software.

1.1 Reseña histórica[1]

Aún antes de que surgiera el concepto de programación orientada a aspectos, el grupo Demeter[13] había considerado y aplicado ideas similares.

El concepto principal detrás del trabajo del grupo Demeter es la programación adaptativa (PA), la cual puede verse como una instancia temprana de la POA. El término programación adaptativa se introdujo recién en 1991. La programación adaptativa constituye un gran avance dentro de la tecnología de software, basada en el uso de autómatas finitos y una teoría formal de lenguaje para expresar concisamente y procesar eficientemente conjuntos de caminos en un grafo arquitectónico, como por ejemplo los diagramas de clase en UML.

La relación entre la POA y la PA surge de la *Ley de Demeter: "Solo conversa con tus amigos inmediatos"*. Esta ley inventada en 1987 en la Northeastern University y popularizada en libros de Booch, Budd, Coleman, Larman, Page-Jones,

[1] El desarrollo histórico se obtuvo de la página del grupo de Demeter[13], para los interesados en profundizar en la historia.

Rumbaugh, entre otros, es una simple regla de estilo en el diseño de sistemas orientados a objetos.

Para poder escribir código respetando la Ley de Demeter observaron que los conceptos que se entrecruzan entre varias clases deberían y tendrían que ser claramente encapsulados. Esto resultaría en una clara separación de los conceptos de comportamiento y de aquellos conceptos de la funcionalidad básica.

Por lo tanto la separación completa de conceptos fue área de interés de este grupo aún antes de que la POA existiera como tal. En 1995 dos miembros de este grupo, Cristina Lopes, actualmente integrante del grupo Xerox PARC, y Walter Huersch, presentaron un reporte técnico sobre separación de conceptos[13] incluyendo varias técnicas como filtros composicionales y PA para tratar con los conceptos que se entrecruzan. Este reporte identificó el tema general de separación de conceptos y su implementación, y lo propuso como uno de los problemas a resolver más importante en el diseño y desarrollo de software.

La definición formal de PA puede considerarse como una definición inicial y general de la POA: en PA los programas se descomponen en varios bloques constructores de corte (crosscutting building blocks). Inicialmente separaron la representación de los objetos como un bloque constructor por separado. Luego agregaron comportamiento de estructura (structure-shy behavior) y estructuras de clase como bloques constructores de corte.

La primera definición del concepto de aspecto fue publicada en 1995, también por el grupo Demeter, y se describía de la siguiente forma: *Un aspecto es una unidad que se define en términos de información parcial de otras unidades*. La definición de aspectos ha evolucionado desde entonces hasta llegar a la definición actual, que será citada más adelante en este trabajo.

Cristina Lopes y Karl Lieberherr empezaron a trabajar con Gregor Kickzales y su grupo. A Lopes y Kickzales no les gustó el nombre "Programación Adaptativa" e introdujeron un mejor término "Programación Orientada a Aspectos" con su propia definición y terminología. En la literatura se lo considera a Gregor Kickzales como el creador de este nuevo paradigma.

Al crecer la POA como paradigma fue posible definir la PA como un caso de especial de la POA[13], esto es, la PA es igual a la POA con grafos y estrategias transversales. Las estrategias transversales podrían considerarse como expresiones regulares que especifican un recorrido en un grafo.

La POA es un paradigma que recién está naciendo. Se están realizando las primeras experiencias prácticas para mostrar su aplicabilidad, y obtener datos empíricos que estimulen la investigación en el tema. La POA está en su primera etapa, donde constantemente surgen nuevos problemas, nuevas herramientas, nuevos contextos en los cuales es posible aplicar aspectos. Este panorama hace pensar que la programación orientada a aspectos se encuentra en mismo lugar que se encontraba la POO hace veinte años.

2. POA: Consideraciones generales

La idea central que persigue la POA es permitir que un programa sea construido describiendo cada concepto 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 y constructores para capturar aquellos elementos que se diseminan por todo el sistema. A estos elementos se les da el nombre de aspectos. Una definición para tales lenguajes sería: Los LOA son aquellos lenguajes que permiten separar la definición de la funcionalidad pura de la definición de los diferentes aspectos.

Los LOA deben satisfacer varias propiedades deseables [7], entre ellas:

- Cada aspecto debe ser claramente identificable.
- Cada aspecto debe auto-contenerse.
- Los aspectos deben ser fácilmente intercambiables.
- Los aspectos no deben interferir entre ellos.
- Los aspectos no deben interferir con los mecanismos usados para definir y evolucionar la funcionalidad, como la herencia.

2.1 ¿Qué es un aspecto?

Gregor Kickzales y su grupo, en [8], brinda un marco adecuado que facilita y clarifica la definición de un aspecto. Lo que propone es agrupar los lenguajes orientados a objetos, los procedurales y funcionales como lenguajes de procedimiento generalizado (LPG), ya que sus mecanismos claves de abstracción y composición pueden verse como agrupados bajo una misma raíz. Esa raíz tendría la forma de un procedimiento generalizado. Los métodos de diseño que han surgido para los LPG tienden a dividir el sistema en unidades de comportamiento o funciones. A este estilo se lo conoce como descomposición funcional [11-12] (si bien la naturaleza exacta de la descomposición funcional difiere entre los paradigmas, para los propósitos de este trabajo alcanza con agruparlos bajo LPG).

En general, decimos que dos propiedades se entrecruzan si al implementarse deben componerse de manera diferente, y aún deban ser coordinadas. Esto se debe a que tienen un comportamiento en común. Al proveer los LPG solamente un único medio de composición, es el programador quien debe realizar la tarea extra de co-componer manualmente las propiedades que se entrecruzan, lo cual lleva a un oscurecimiento y a un aumento de la complejidad del código.

Gracias a todo este marco, descripto en [8], podemos diferenciar los aspectos de los demás integrantes del sistema: al momento de implementar una propiedad, tenemos que la misma tomará una de las dos siguientes formas:

1. Un componente: si puede encapsularse claramente dentro de un procedimiento generalizado. Un elemento es claramente encapsulado si está bien localizado, es fácilmente accesible y resulta sencillo componerlo.

2. Un aspecto: si no puede encapsularse claramente en un procedimiento generalizado. Los aspectos tienden a ser propiedades que afectan la performance o la semántica de los componentes en forma sistemática (Ejemplo: sincronización, manejo de memoria, distribución, etc.)

A la luz de estos términos, podemos enunciar la meta principal de la POA: brindar un contexto al programador que permita separar claramente componentes y aspectos, separando componentes entre sí, aspectos entre sí, y aspectos de componentes, a través de mecanismos que hagan posible abstraerlos y componerlos para producir el sistema completo. Tenemos entonces una importante y clara diferencia respecto de los LPG, donde todos los esfuerzos se concentran únicamente en los componentes, dejando de lado los aspectos.

Una vez diferenciados los aspectos de los componentes, estamos en condiciones de **definir a un aspecto como un concepto que no es posible encapsularlo claramente, y que resulta diseminado por todo el código.** Los aspectos son la unidad básica de la programación orientada a aspectos. Una definición más formal, y con la que se trabaja actualmente es: Un aspecto es 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. Dicha definición pertenece al creador de la POA, Gregor Kickzales.

2.2 Comparación gráfica

Ahora que ya hemos introducido los principales rasgos de la POA, podemos comparar la forma de una implementación basada en los LPG versus implementación basada en POA.

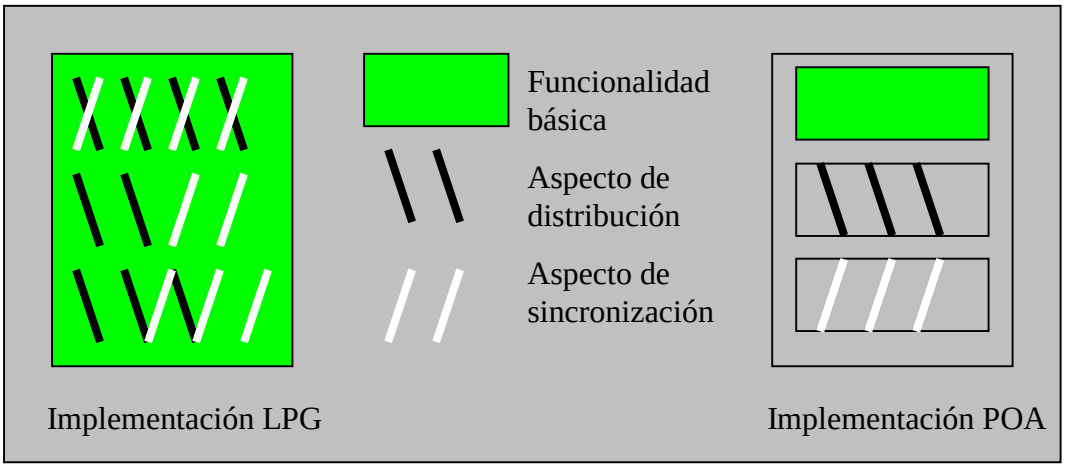


Figura 1: Implementación POA versus LPG.

En la implementación sobre LPG, los aspectos se mezclan con la funcionalidad básica, y entre ellos. En cambio, en la implementación basada en el nuevo paradigma de la POA, la separación es completa, con una notable mejoría en lo que respecta a la modularización.

También podemos comparar las diferentes tecnologías que han sido creadas hasta la actualidad, con el paradigma de aspectos, observando la evolución que han tenido las tecnologías.

Tecnología	*Conceptos Claves*	*Constructores*
Programación estructurada	Constructores de control explícitos	Do, while, y otros iteradores
Programación modular	Ocultamiento de Información	Módulos con interfaces bien definidas
Abstracción de datos	Ocultar la representación del dato	Tipo de dato abstracto
Programación orientada a objetos	Objetos con clasificación y especialización	Clases, objetos, polimorfismo
Programación orientada a aspectos.	"6 C" y "4 S"	Aspectos

Tabla 1: Comparación entre los paradigmas

Las "6 C" , de Mehmet Aksit, hacen referencia a los seis aspectos cuyos nombres en inglés comienzan con la letra 'c' y que son los siguientes:

- Entrecruzado (**C**rosscutting en inglés) : el concepto ya ha sido introducido previamente en este trabajo.
- **C**anónico : brindar una implementación estable para los conceptos.
- **C**omposición: proveer factores de calidad, como adaptabilidad, reusabilidad y extensibilidad.
- **C**lausura: mantener los factores de calidad del diseño en la etapa de implementación.
- **C**omputabilidad: crear software ejecutable.
- **C**ertificabilidad: evaluar y controlar la calidad de los modelos de diseño e implementación.

Las "4 S" hacen referencia a conceptos sobre Separación Exitosa, de Harold Ossher, y corresponden a:

- **S**imultáneo: la coexistencia de diferentes composiciones son importantes.
- Auto-Contenido(**S**elf-Contained): Cada módulo debe declarar sus dependencias, para poderlo entenderlo individualmente.

- **S**imétrico: no debe haber distinción en la forma en que los diferentes módulos encapsulan los conceptos, para obtener una mayor confiabilidad en la composición.
- Espontaneidad (**S**pontaneous): debe ser posible identificar y encapsular nuevos conceptos, y aún nuevas formas de conceptos que se presenten durante el ciclo de vida del software.

2.3 Fundamentos de la POA

Los tres principales requerimientos de la POA son: [19]

- Un lenguaje para definir la funcionalidad básica, conocido como lenguaje base o componente. Podría ser un lenguaje imperativo, o un lenguaje no imperativo (C++, Java, Lisp, ML).
- Uno o varios lenguajes de aspectos, para especificar el comportamiento de los aspectos. (COOL, para sincronización, RIDL, para distribución, AspectJ, de propósito general.)
- Un tejedor de aspectos(Weaver), que se encargará de combinar los lenguajes. Tal proceso se puede retrasar para 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 aquellos conceptos que cruzan todo el código.

A la hora de "tejer" los componentes y los aspectos para formar el sistema final, es claro que se necesita una interacción entre el código de los componentes y el código de los aspectos. También es claro que esta interacción no es la misma interacción que ocurre entre los módulos del lenguaje base, donde la comunicación está basada en declaraciones de tipo y llamadas a procedimientos y funciones. La POA define entonces una nueva forma de interacción, provista a través de los puntos de enlace (join points).

Los puntos de enlace brindan la interfaz entre aspectos y componentes; son lugares dentro del código donde es posible agregar el comportamiento adicional que destaca a la POA. Dicho comportamiento adicional es especificado en los aspectos.

Aún nos falta introducir el encargado principal en el proceso de la POA. Este encargado principal conocido como tejedor debe realizar la parte final y más importante: "tejer" los diferentes mecanismos de abstracción y composición que aparecen tanto en los lenguajes de aspectos como en los lenguajes de componentes, guiado por los puntos de enlace.

2.3.1 Estructura general

La estructura de una implementación basada en aspectos es análoga a la estructura de una implementación basada en los LPG.

Una implementación basada en LPG consiste en: [8]

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

La figura 2 nos permite visualizar gráficamente la estructura anterior.

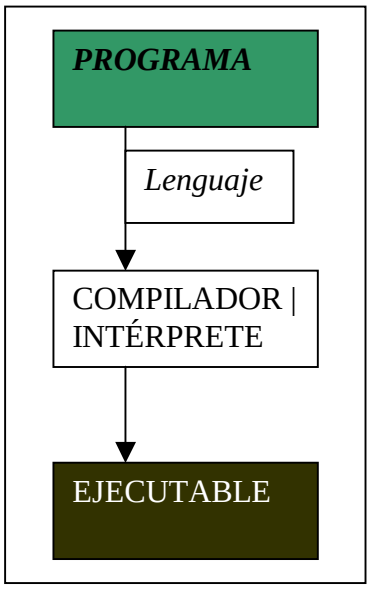


Figura 2: Estructura LPG.

Una implementación basada en POA consiste en: [8]

- 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.

Gráficamente, se tiene una estructura como la siguiente:

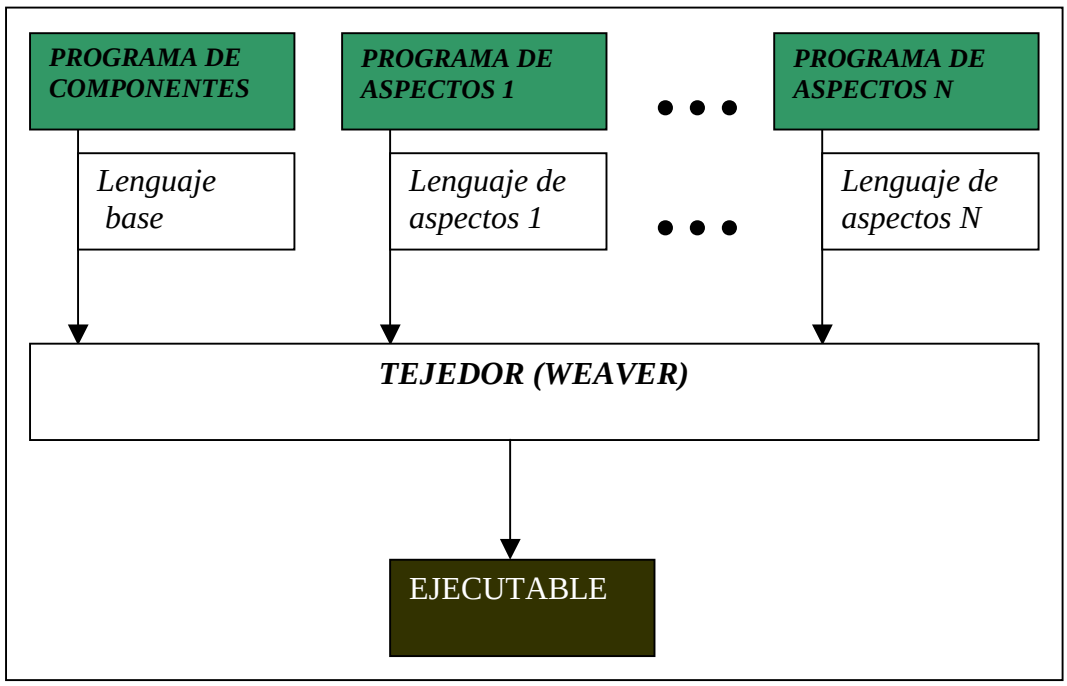


Figura 3: Estructura POA

Cabe notar que la función que realizaba el compilador se encuentra ahora incluida en las funciones del tejedor.

2.3.2 Desarrollo orientado a aspectos

Diseñar un sistema basado en aspectos requiere entender 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 lenguaje componente debe proveer la forma de implementar la funcionalidad básica y asegurar que los programas escritos en ese lenguaje componente no interfieran con los aspectos. Los lenguajes de aspectos tienen que proveer los medios para implementar los aspectos deseados de una manera intuitiva, natural y concisa[8].

En general el desarrollo de una aplicación basada en aspectos consiste de tres pasos [18]:

1- Descomposición de aspectos: es descomponer los requerimientos para distinguir aquellos que son componentes de los que son aspectos.
2- Implementación de requerimientos: implementar cada requerimiento por separado.
3- Recomposición: dar las reglas de recomposición que permitan combinar el sistema completo.

En [14], John Lamping propone una diferente visión del diseño. Asegura que la decisión sobre qué conceptos son base y cuáles deben ser manejados por aspectos es irrelevante, ya que no afectará demasiado a la estructura del programa, sino 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.

Esta parte es fundamental para una descomposición de aspectos exitosa. Diferentes aspectos pueden contribuir a la implementación de un mismo ítem de la funcionalidad básica y un sólo aspecto puede contribuir a la implementación de varios ítems. Una buena separación de responsabilidades entre los conceptos 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.

Lamping concluye que el trabajo del programador que utiliza POA es definir precisamente los ítems de la funcionalidad básica y obtener una buena separación de responsabilidades entre los conceptos. Luego los aspectos le permitirán al programador separar los conceptos en el código.

Creemos que la propuesta de Lamping es solo una manera más compleja de llegar a los tres pasos del desarrollo orientado a aspectos, enumerados al comienzo de esta sección. Estos pasos nos proveen un camino más natural y directo. Tenemos

entonces dos formas de lograr el mismo resultado pero al seguir los tres pasos logramos el resultado de una manera más apegada y coherente a la filosofía orientada a aspectos.

En un reporte técnico de la Universidad de Virginia [15], se establece que muchos de los principios centrales a la POO son ignorados dentro del diseño orientado a aspectos, como por ejemplo el ocultamiento de información, debido a que los aspectos tienen la habilidad de violar estos principios. Para esto se propone una filosofía de diseño orientada a aspectos que consiste de cuatro pasos:

- Un objeto es algo.
- Un aspecto no es algo. Es algo sobre algo.
- Los objetos no dependen de los aspectos.
- Los aspectos representan alguna característica o propiedad de los objetos, pero no tienen control sobre los mismos.

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

Un aspecto no es algo. Es algo sobre algo: un aspecto 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 con ninguna clase, entonces entrecruza cero clases, y por lo tanto no tiene sentido 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.

Los objetos no dependen de los aspectos: Un aspecto no debe cambiar las interfaces de la clases asociadas a él. Solo 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.

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 capaces de manipular el resto de los objetos. A los aspectos se les permite tener esta visión especial, pero debería limitarse a manipular objetos de la misma forma que los demás objetos lo hacen, a través de la interface.

Esta filosofía de diseño permite a los aspectos hacer su trabajo de automatización y abstracción, respetando los principios de ocultamiento de información e interface manifiesta [16].

Creemos que ésta filosofía es una política de diseño totalmente aceptable, y que se debe conocer y aplicar durante el desarrollo orientado a aspectos. Si bien es una excelente política de diseño, no deberíamos quedarnos solamente en esto, sino que deberíamos llevarla más allá, haciéndola formar parte de las reglas del lenguaje orientado a aspectos. Esto es, que no sea una opción, sino que sea forzada por el lenguaje. 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

inmiscuirse en la representación interna del objeto, por lo tanto se necesita una solución similar: el lenguaje debería proveer una interface para que aspectos y objetos se comuniquen respetando esa restricción.

Esta es una de las desventajas de los lenguajes orientados a aspectos actuales. El desafío está en construir lenguajes orientados a aspectos capaces de brindar mecanismos lingüísticos suficientemente poderosos para respetar por completo todos los principios de diseño.

2.4 Tejido estático versus 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.

Otro ejemplo más completo sería el Junction Point Aspect Language (JPAL) [20], basado en los conceptos de protocolos de meta objetos (metaobject protocols MOP) y meta-programación. En JPAL existe una entidad llamada Administrador de Programas de Aspectos el cual puede registrar un nuevo aspecto de una aplicación y puede llamar a métodos de aspectos registrados. Es implementado como una librería dinámica que almacena los aspectos y permite dinámicamente agregar, quitar o modificar aspectos, y mandar mensajes a dichos 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 implica que el proceso de composición se realiza en tiempo de ejecución, decrementando 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. Es decir, una vez que se identificó un aspecto, el mismo se mantendrá como concepto a lo largo de todo el ciclo de vida de la aplicación. Lo que puede variar son las características o niveles de aplicación de ese concepto. Por ejemplo, la comunicación es estática, en el sentido que el concepto de comunicación está siempre presente, pero algunas características de la misma, como la velocidad o el costo, pueden cambiar por distintas razones en ejecución. Es más, quizás algunas propiedades de la comunicación no se conozcan hasta ejecución. Por lo tanto es necesario que el aspecto de comunicación sea dinámico para adaptarse correctamente a los distintos casos de uso. Un ejemplo más detallado de lo anterior se encuentra en [21], un estudio de aspectos dinámicos en la plataforma SMove.

2.4.1 Guías de diseño

Tomando como base la guía de diseño para la implementación de sistemas abiertos, descripta en [22], podemos enunciar las siguientes pautas para aquellos tejedores que soporten tanto aspectos dinámicos como estáticos:

- 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. En SMove[21], 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.
- 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 minimal. 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.

2.5 Lenguajes de aspectos específicos versus de propósito general

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

Los lenguajes de aspectos de dominio específico son capaces de manejar uno o más aspectos, pero no pueden manejar otros aspectos más allá de los aspectos para los cuales fueron diseñados. Tienen un mayor nivel de abstracción que el lenguaje base, y expresan los conceptos de dominio específico en un nivel de representación más alto. Para garantizar que los aspectos del dominio sean escritos en el lenguaje de aspectos, y evitar conflictos de control y dependencia entre el lenguaje base y el lenguaje de aspectos, los lenguajes de dominio específicos imponen restricciones en la utilización del lenguaje base. Como ejemplo de este tipo de lenguaje de aspectos se puede nombrar a COOL (COOrdination Language), de Xerox [23], un lenguaje para sincronización de hilos concurrentes. En COOL, el lenguaje base es una restricción de Java, ya que se han eliminado los métodos wait, notify, y notifyAll, y la palabra reservada synchronized para evitar que el lenguaje base sea capaz de expresar sincronización. Se obtiene un mayor nivel de abstracción que en Java al especificar la sincronización de hilos de manera declarativa. Un segundo ejemplo es RIDL(Remote Interaction and Data transfers Language), para el aspecto de distribución: invocación remota y transferencia de datos.

Los lenguajes de aspectos de propósito general son diseñados para describir cualquier clase de aspecto, no solo específicos, por lo que no pueden imponer restricciones al lenguaje base. El nivel de abstracción del lenguaje base y del lenguaje de aspectos de propósito general es el mismo. Además, tienen el mismo conjunto de instrucciones, ya que debe ser posible expresar cualquier código al programar un aspecto. Un ejemplo es AspectJ, donde Java es el lenguaje base, y las instrucciones de los aspectos también se escriben en Java.

La principal desventaja de los lenguajes de aspectos de propósito general es que no garantizan la separación de conceptos, esto es, que la unidad de aspecto se utilice únicamente para programar el aspecto. Esto último es uno de los objetivos principales de la POA. En comparación, los lenguajes de aspectos de dominio específico fuerzan la separación de conceptos.

Sin embargo, los de propósito general, proveen un ambiente más adecuado para el desarrollo de aspectos. Es una única herramienta capaz de describir todos los aspectos que se necesiten. Al pasar de una aplicación a otra, por ejemplo al pasar de trabajar en un contexto de sincronización a uno de manejo de excepciones, se mantiene la misma herramienta para programar los aspectos. No se debe aprender para cada aplicación todo de nuevo, que es costoso y hasta frustrante, sino que cada vez se obtiene una mayor experiencia, con lo cual programar nuevos aspectos es más sencillo, y permite evolucionar los conceptos sobre el paradigma. Con los específicos, el pasar de una aplicación a otra, significa comenzar de cero nuevamente, otros lenguajes, otras restricciones, que quizás sean similares, pero quizás sean completamente distintas. Si es el caso que se desee utilizar aspectos para una única aplicación, por una única vez, entonces un lenguaje de aspectos de dominio específico hará un mejor papel, con una separación de conceptos más clara.

Pero dada la gran variedad de aplicaciones en los cuales los aspectos pueden ser aplicados y observando las ventajas que trae un lenguaje de propósito general, es preferible esta última aproximación siendo capaz de manejar varios ambientes de trabajo con el mismo lenguaje, reduciendo costos de aprendizaje y desarrollo.

2.6 El rol del lenguaje base

Una de las decisiones relevantes a la hora de diseñar un lenguaje orientado a aspectos tiene que ver con el lenguaje base. Las distintas alternativas son: diseñar un *nuevo* lenguaje base o *adoptar* un lenguaje base existente.

En el caso de adoptar un lenguaje base existente, es preferible que el mismo sea un lenguaje de propósito general, de manera de no restringir la aplicabilidad del lenguaje orientado a aspectos. Una ventaja obvia de esta opción es el menor esfuerzo para diseñar e implementar lenguajes para ambientes orientados a aspectos, dado que nos ahorramos el trabajo que implica el diseño de un nuevo lenguaje base. También, al trabajar con un lenguaje existente nos evitamos los problemas de trabajar con algo que todavía no ha sido probado exhaustivamente. Una última ventaja radica en el hecho que el programador solo debe aprender el lenguaje de aspectos, y no también un nuevo lenguaje para la funcionalidad básica, la cual sigue siendo, aún en la POA, la parte más grande del sistema.

Si en cambio se elige diseñar un nuevo lenguaje base entonces se obtiene una mayor flexibilidad, una mayor libertad, y una manera más natural de expresar aquellas abstracciones que forman parte de la funcionalidad básica. Si bien estas ventajas son importantes, el costo de diseñar un nuevo lenguaje resulta demasiado alto. Sin embargo, existe otra, más oculta, pero de mucha mayor trascendencia, dado que afecta directamente al corazón mismo de la POA, al surgir del enunciado de la meta principal que persigue este nuevo paradigma, la separación completa de conceptos. Una gran parte de los investigadores que han trabajado con aspectos, establecen que no está claro si es posible lograr la separación completa de conceptos utilizando un lenguaje base existente. Si esta posición logra finalmente imponerse, entonces el futuro de la POA indica que el diseño de nuevos lenguajes base será altamente necesario. En [2], K. Mehner y A. Wagner, de la Universidad de Paderborn, plantean una futura investigación con respecto a este último concepto. Estudiarán si en un domino de aplicación ,como por ejemplo el dominio de sincronización de hilos concurrentes, la separación de conceptos puede lograrse utilizando un lenguaje base existente. De no ser así investigarán si el problema ocurrirá con cualquier lenguaje base de propósito general o si se debe construir un nuevo lenguaje base.

Hasta ahora los lenguajes orientados a aspectos específicos y de propósito general han elegido utilizar un lenguaje base existente. Aparte de las ventajas de esta elección esto trae consigo una serie de problemas. En el caso de los lenguajes de dominio específico uno no es completamente libre para diseñar los puntos de enlace sino que se restringe a aquellos que pueden ser identificados en el lenguaje base.

Otro problema surge de la restricción que imponen sobre el lenguaje base. Dicha restricción es bastante difícil de lograr ya que se deben eliminar elementos de un sistema complejo, donde cada uno tiene su lugar y sus responsabilidades, y están coordinados entre sí. Tener en cuenta que ya es una tarea compleja el diseño de un lenguaje de programación, aún es más difícil realizarle cambios a un lenguaje que no fue diseñado para esta evolución.

Los lenguajes de propósito general son más fáciles de implementar sobre la base de un lenguaje de programación existente al no imponer restricciones sobre la base. También tienen el mismo problema de limitar los puntos de enlace a aquellos que pueden ser identificados en el lenguaje base.

2.7 Aspectos en lenguajes procedurales

Las razones para utilizar aspectos dentro de los lenguajes procedurales son similares a aquellas para los lenguajes orientados a objetos. Los distintos procedimientos que forman parte del sistema quizás necesiten compartir datos e incluso otros procedimientos, y también quizás sea deseable agregarles comportamiento. Se podría pensar que la programación orientada a objetos solucionaría estos problemas, pero como hemos discutido antes, existen conceptos que no son capturados correctamente.

Sin utilizar aspectos, los lenguajes procedurales pueden compartir datos y procedimientos a través del pasaje de parámetros o a través de un espacio de nombre común. En los lenguajes estructurados por bloques, el espacio de nombre se particiona en una estructura jerárquica, donde cada bloque tiene visibilidad sobre el(los) bloque(s) que lo contiene(n). Luego, si dos procedimientos *Proc1* y *Proc2* necesitan acceder a una variable *V*, entonces *V* deberá declararse en el mismo bloque que *Proc1* y *Proc2*, o en un bloque que contenga a los procedimientos, así como lo muestra el siguiente código:

```
Procedure ProcGlobal;
  Var V: boolean;
     Procedure Proc1;{
     …
     };
     Procedure Proc2;{
     …
     };
{
 …
};
```

Sin embargo, el mecanismo de alcance de los lenguajes estructurados por bloques sufre de importantes limitaciones. En el análisis de Wulf y Shaw sobre los lenguajes estructurados [34] se describen los cuatro problemas más importantes:

- Efectos colaterales: pueden definirse como la modificación del ambiente no local. Están presentes en la mayoría de los lenguajes, ya que deshabilitar todos los posibles efectos colaterales resulta en un lenguaje muy restrictivo, poco práctico, y de escasa utilidad.
- Acceso indiscriminado: toda entidad declarada en un bloque, será siempre visible a los bloques internos, y no hay manera de evitarlo.
- Vulnerabilidad: puede ser imposible mantener el acceso a una variable, cuando se lo desea. Puede ocurrir cuando un procedimiento *P* declara una variable *V*, y tiene un procedimiento interno *H* que accede a *V*. Ahora si se inserta entre ellos un procedimiento *J*, que declara una variable local *V* (oculta a la variable *V* declarada en *P*) entonces *H* accederá a la variable *V* declarada en *J* y no a la declarada en *P*, que era lo deseado.

- No se permiten definiciones solapadas: Dados tres procedimientos, *P1*, *P2*, y *P3*, que desean compartir dos variables, *V1* y *V2* de la siguiente forma: *P1* y *P2* comparten *V1*, *P2* y *P3* comparten *V2*, *P3* no debe acceder a *V1*, y *P1* no debe acceder a *V2*. Este tipo de relación es imposible de implementar en un lenguaje estructurado por bloques. Gráficamente, se desea modelar el siguiente comportamiento:

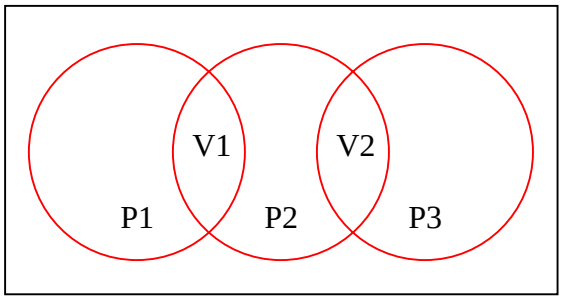


Figura 4: Definición solapada

Por lo tanto, es claro que la estructura de bloques no es un mecanismo apropiado para capturar los conceptos entrecruzados debido a los problemas de programación potenciales que puede ocasionar (acceso indiscriminado y vulnerabilidad), y la imposibilidad de capturar ciertos tipos de conceptos entrecruzados (definiciones solapadas).

Luego, es la intención en [15] que los aspectos se empleen como un mecanismo que extienda las reglas de alcance de los lenguajes procedurales para capturar los conceptos entrecruzados sin la presencia de los problemas anteriormente mencionados. Entonces un aspecto se utiliza como un contenedor para almacenar procedimientos y variables específicas. El acceso a los aspectos es especificado por una palabra clave (*accessed by* <Lista_Procedimientos>) que determina los únicos procedimientos que pueden acceder a él, sin tener en cuenta el nivel de anidamiento de los mismos. De esta forma, los aspectos solucionan el acceso indiscriminado, la vulnerabilidad y permiten definiciones solapadas, al introducir un nuevo espacio de nombre, que solo es accesible por algunos procedimientos. Por ejemplo, el siguiente código resuelve el problema de definición solapada:

```
Aspect A
  accessed by  P1, P2 ;
  { var  V1: boolean;
   }
Aspect B
  accessed by  P2, P3 ;
  { var  V2: boolean;
   }

Procedure P1;{
 // puede acceder a V1 y no  a  V2
 }
Procedure P2;{
// puede acceder a V1 y a V2
}
Procedure P3;{
// puede acceder a V2 y no a V1
 }
```

Finalmente, los aspectos descriptos también satisfacen las propiedades requeridas en el análisis de Wulf y Shaw [34] para una solución alternativa apropiada a las limitaciones de los lenguajes estructurados por bloques:

- No debe extender el alcance de un nombre a los bloques internos: Se especifica explícitamente los procedimientos que pueden acceder a un aspecto.
- El derecho de acceso a un nombre tiene que ser de mutuo acuerdo entre el creador y el que accede: se cumple por la misma razón de la propiedad anterior.
- Los accesos a una estructura y a sus sub-estructuras deben ser independientes: el acceso a una estructura no implica acceder a sus sub-estructuras.

Como conclusión, se observa que los aspectos juegan un rol diferente en los lenguajes procedurales que en los lenguajes orientados a objetos. Esto resulta de la diferencias entre los paradigmas, donde los componentes se describen como objetos en uno y como procedimientos en el otro. Al aplicar aspectos a los lenguajes procedurales, los procedimientos tienen conocimiento y dependen de los aspectos, a diferencia de los objetos, que no tienen conocimiento ni dependen de los aspectos.

2.8 Aproximaciones alternativas de aspectos

La meta-programación es la capacidad que tienen los programas de razonar sobre sí mismos. Un lenguaje permite la meta-programación cuando tiene la capacidad de razonar sobre programas escritos en ese lenguaje. Dentro del paradigma funcional se denominan lenguajes meta funcionales, y dentro del paradigma lógico, lenguajes meta lógicos.

El paradigma declarativo, que incluye el paradigma funcional y el lógico, permite naturalmente la meta-programación. Se puede pensar que los aspectos son a la funcionalidad básica lo que los meta-programas son a los programas:

$$\frac{Meta-programación}{programas} = \frac{Aspectos}{funcionalidad\_básica}$$

Siguiendo este razonamiento surgen dos aproximaciones alternativas para describir aspectos que serán abarcadas a continuación.

2.8.1 Meta-programación lógica de aspectos

Se basa en la utilización de lenguajes meta lógicos para razonar sobre aspectos, unificando el lenguaje para declarar aspectos con el lenguaje para implementar el tejedor.

Los aspectos no son declarados dentro de un lenguaje de aspectos diseñado especialmente para tal efecto, sino que son simplemente aserciones lógicas expresadas en un lenguaje lógico general. Estos hechos lógicos funcionan como "libros" de una librería de reglas lógicas que implementan el tejedor.

La principal ventaja de utilizar hechos lógicos para declarar aspectos en vez de un lenguaje de aspectos diseñados especialmente es que las declaraciones de aspectos se pueden acceder y declarar por reglas lógicas. Esta técnica permite al usuario adaptar o extender el lenguaje de aspectos.

El programa básico es representado indirectamente por un conjunto de proposiciones lógicas. Bajo esta aproximación un tejedor entendería un conjunto de declaraciones para describir aspectos como las que propone Brichau en [31]:

- *introduceMethod(<nombre_de_clase>,{<código_del_método>})*: introduce el método *<código_del_método>* en la clase *<nombre_de_clase>*.
- *introduceVariable(<nombre_de_clase>,{<variable>})*: introduce la variable *<variable>* en la clase *<nombre_de_clase>*.
- *adviceBeforeMethod(<nombre_de_clase>,<nombre_del_método>,{ <código_agregado>})*: agrega *<código_agregado>* antes del método *<nombre_del_método>* en la clase *<nombre_de_clase>*.
- *adviceAfterMethod(<nombre_de_clase>,<nombre_del_método>,{ <código_agregado>})*: agrega *<código_agregado>* después del método *<nombre_del_método>* en la clase *<nombre_de_clase>*.

Dado un módulo de declaraciones el tejedor genera código para todas las declaraciones presentes en dicho módulo. Esto lo logra generando consultas lógicas para recoger todas las declaraciones mencionadas. Por ejemplo: el siguiente módulo implementa un aspecto simple de traza.

```
adviceBeforeMethod(claseA,unmétododeA,
                    {system.out.println(`Entrando a claseA.unmétodoA´);}).
adviceAfterMethod(claseA,unmétododeA,
                   {system.out.println(`Saliendo de claseA.unmétodoA´);}).
adviceBeforeMethod(claseB,unmétododeB,
                    {system.out.println(`Entrando a claseB.unmétodoB´);}).
adviceAfterMethod(claseB,unmétododeB,
                   {system.out.println(`Saliendo de claseB.unmétodoB´);}).
adviceBeforeMethod(claseC,unmétododeC,
                    {system.out.println(`Entrando a claseC.unmétodoC´);}).
adviceAfterMethod(claseC,unmétododeC,
                   {system.out.println(`Saliendo de claseC.unmétodoC´);}).
```

Tomando ventaja del paradigma lógico podemos evitar la duplicación de código del módulo anterior reformulándolo a través de reglas lógicas:

```
adviceBeforeMethod(?clase,?unmétodo,
   {system.out.println(`Entrando a ?clase.?unmétodo´);}):-traza(?clase,?unmétodo).

adviceAfterMethod(?clase,?unmétodo,
   {system.out.println(`Saliendo de ?clase.?unmétodo´);}):-traza(?clase,?unmétodo).

traza(claseA,unmétododeA).
traza(claseB,unmétododeB).
traza(claseC,unmétododeC).
```

Nota: los identificadores de variables comienzan con un signo de interrogación. Por ejemplo: *?clase*.

Las dos primeras declaraciones son reglas lógicas que declaran avisos para cada declaración de la forma *traza(?clase,?unmétodo)*. Significa que las declaraciones de avisos son válidas solo si hay declaraciones de trazas. En este proceso, el motor de inferencia lógico liga las variables *?clase* y *?unmétodo* con las constantes especificadas en las declaraciones de trazas. A través de este proceso de evaluación obtenemos el mismo aspecto que antes. También se observa la separación entre la implementación del aspecto y el uso del aspecto: las dos primeras reglas son implementadas por el programador del aspecto y las últimas tres son introducidas por el usuario del aspecto cuando el aspecto se combina con un programa básico particular. El verdadero código del aspecto se concentra en las dos primeras reglas

mientras que las tres restantes completan los parámetros donde el código de aspecto debe insertarse.

También se pueden componer aspectos de una manera directa, aprovechando una vez más el poder de la lógica. Supongamos que deseamos implementar un nuevo módulo de aspectos a partir de dos aspectos ya definidos: *aspectoA* y *aspectoB*. El nuevo aspecto tendrá el comportamiento de *A* y *B* en aquellos métodos que ambos afecten; el comportamiento de *A* si *B* no afecta ese método y el comportamiento de *B* si *A* no afecta ese método. Contiene las variables y métodos de ambos aspectos.

```
introduceMethod(?clase,?códigométodo):-
        aspectA.introduceMethod(?clase,?códigométodo).
introduceMethod(?clase,?códigométodo):-
        aspectB.introduceMethod(?clase,?códigométodo).
introduceVariable(?clase,?variable):-
        aspectA.introduceVariable(?clase,?variable).
introduceVariable(?clase,?variable):-
        aspectB.introduceVariable(?clase,?variable).

adviceBeforeMethod(?clase,?método,{?codigoA ?codigoB }):-
    aspectA.adviceBeforeMethod(?clase,?método,{?codigoA}) and
    aspectB.adviceBeforeMethod(?clase,?método,{?codigoB}).

adviceBeforeMethod(?clase,?método,{?códigoA }):-
    aspectA.adviceBeforeMethod(?clase,?método,{?códigoA}) and
    not(aspectB.adviceBeforeMethod(?clase,?método,{?códigoB})).

adviceBeforeMethod(?clase,?método,{?códigoB }):-
    not(aspectA.adviceBeforeMethod(?clase,?método,{?códigoA})) and
    aspectB.adviceBeforeMethod(?clase,?método,{?códigoB}).

adviceAfterMethod(?clase,?método,{?códigoA ?códigoB }):-
    aspectA.adviceAfterMethod(?clase,?método,{?códigoA}) and
    aspectB.adviceAfterMethod(?clase,?método,{?códigoB}).

adviceAfterMethod(?clase,?método,{?códigoA }):-
    aspectA.adviceAfterMethod(?clase,?método,{?códigoA}) and
    not(aspectB.adviceAfterMethod(?clase,?método,{?códigoB})).

adviceAfterMethod(?clase,?método,{?códigoB }):-
    not(aspectA.adviceAfterMethod(?clase,?método,{?códigoA})) and
    aspectB.adviceAfterMethod(?clase,?método,{?códigoB}).
```

Nota: se introduce el operador de alcance ".", para referir a las declaraciones definidas en otros módulos. Por ejemplo: *aspectB.declaraciónS* especifica que *declaraciónS* se encuentra definida en el módulo *B*. Además se introduce el orden de

la composición. Esto se refleja en la quinta y sexta regla donde el código de *A* precede al código de *B*.

Una de las herramientas dentro de esta aproximación hacia los aspectos es TyRuBa, un lenguaje de meta-programación lógico para Java. Principalmente fue diseñado para generar código Java a partir de descripciones lógicas de la estructura del programa. Fue implementado como parte de la tesis de postgrado de Kris de Volder sobre meta-programación lógica orientada a tipos. El acrónimo TyRuBa deriva de TypeRuleBase, que significa basado en reglas con tipos. En pocas palabras, es un lenguaje de programación lógico con facilidades para la manipulación de código Java con el propósito de generar código. Para mayor referencia dirigirse a la página de TyRuBa [38].

Las ventajas de esta aproximación provienen del paradigma lógico, porque el programador se concentra en los aspectos y no en cómo estos serán combinados por el tejedor. La combinación es tarea del proceso de evaluación. Los aspectos se pueden componer fácilmente, como se vio anteriormente, reduciendo los conflictos entre ellos.

En [30] se destacan ventajas tanto para el implementador como para el usuario de aspectos: Los usuarios pueden extender el lenguaje de aspectos "on the fly" definiendo nuevos tipos de declaraciones de aspectos a partir de las declaraciones existentes. De igual forma el implementador puede definir declaraciones de aspectos en términos de declaraciones de aspectos de menor nivel.

Cabe destacar que la utilización de un lenguaje lógico maduro, existente y probado con los años, resulta en una importante ventaja sobre un lenguaje de aspectos especialmente diseñado: el lenguaje lógico puede servir de manera uniforme como formalismo para declarar aspectos como hechos lógicos y como meta-lenguaje para la combinación como reglas lógicas.

2.8.2 Meta-programación funcional de aspectos

Se basa en la utilización de lenguajes funcionales de alto orden para describir aspectos.

La meta-programación funcional es una propuesta general y efectiva para especificar el código de aspectos y el tejedor. Utilizando la terminología de aspectos se tiene que los componentes son programas funcionales mientras que los aspectos se implementan como transformación de programas, como se explicará en la sección 5.2 Trabajos relacionados. El proceso de tejer se considera como una composición de programas que combina los componentes y los aspectos aplicando las transformaciones a los componentes modeladas por los aspectos [33].

La razón de utilizar meta-programas funcionales radica en la facilidad de verificar propiedades y la naturalidad de los lenguajes funcionales de alto orden para definir manipulaciones abstractas de programas.

Ambas aproximaciones, lógica y funcional, proveen un ambiente adecuado para estudiar e investigar los cimientos teóricos del paradigma de la programación orientada a aspectos[32].

3. Lenguajes orientados a aspectos

A continuación describiremos informalmente algunos lenguajes orientados a aspectos disponibles actualmente. En la sección 3.9 estudiaremos con mayor profundidad el lenguaje orientado a aspectos AspectJ, ya que hemos seleccionado este lenguaje por ser la herramienta con mayor usabilidad, futuro, popularidad y desarrollo.

3.1 JPAL

La principal característica de esta herramienta es que los puntos de enlace son especificados independientemente del lenguaje base. Estos puntos de enlace independientes reciben el nombre de Junction Point (JP). Por lo tanto la herramienta se llama JPAL que significa Junction Point Aspect Language[20], esto es, Lenguaje de Aspectos basados en JP. Usar programas escritos en JPAL para describir nuevos lenguajes de aspectos facilita para ese lenguaje el desarrollo de tejedores. De hecho, el tejedor JPAL genera un esquema del tejedor de aspectos llamado Esquema del Tejedor. Este esquema tiene un mecanismo que automáticamente conecta el código base con los programas de aspectos en puntos de control llamados acciones.

El código que agrega el Esquema del Tejedor invoca, cuando es alcanzado en ejecución, las acciones correspondientes para permitir la ejecución de los programas de aspectos. Esto permite una vinculación dinámica con los programas de aspectos, lo cual hace posible modificar en tiempos de ejecución los programas de aspectos[20]. Sin embargo, esta solución no es lo suficientemente poderosa como para agregar o reemplazar programas de aspectos en ejecución. Para tal efecto se agrega al Esquema del Tejedor una entidad llamada Administrador de Programas de Aspectos (APA), el cual puede registrar un nuevo aspecto de una aplicación y puede llamar a métodos de aspectos registrados. Es implementado como una librería dinámica que almacena los aspectos y permite dinámicamente agregar, quitar o modificar aspectos, y mandar mensajes a dichos aspectos. La comunicación entre el Administrador y el Esquema del Tejedor se logra a través de un Protocolo de Comunicación entre Procesos que permite registrar aspectos dinámicamente.

La siguiente figura resume la arquitectura JPAL:

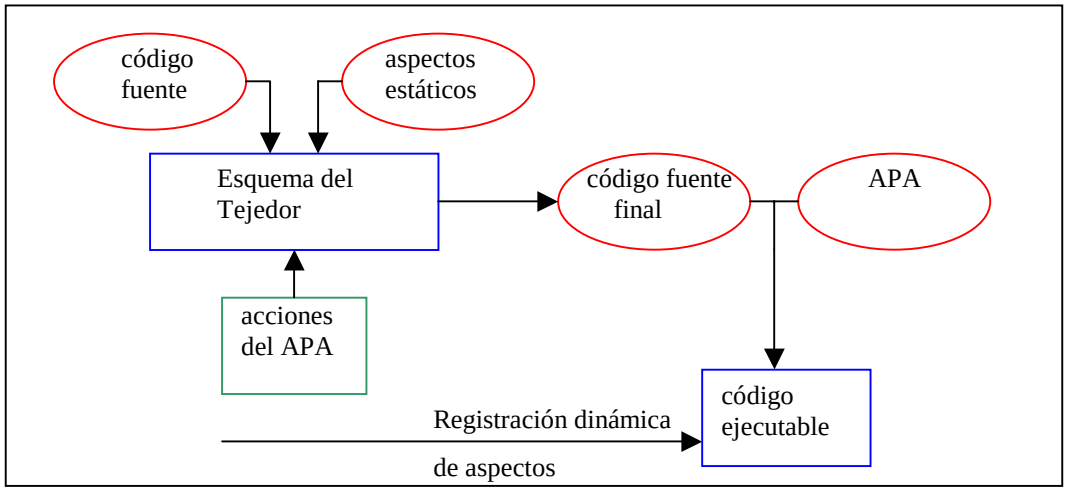


Figura 5: Arquitectura JPAL.

El costo de ejecución de una arquitectura basada en JPAL depende en la implementación del Administrador de Programas de Aspectos que tiene mayor eficiencia en ambientes interpretados[20].

Resumiendo, JPAL , un descendiente de AspectJ, tiene la particularidad de separar los puntos de enlace, que son independientes del lenguaje base, de sus acciones asociadas que dependen de decisiones de implementación. Esta separación permite generar un Esquema de Tejedor para cualquier lenguaje de aspectos. Este esquema ofrece un puente entre el control de la ejecución y la ejecución de la acción. Tomando ventaja de esta redirección se obtiene un modelo de arquitectura para el manejo dinámico de aspectos[20].

Su principal aplicación es para la implementación de sistemas distribuidos.

3.2 D

D [23] es un ambiente de lenguajes de aspectos para la programación distribuida. Se llama ambiente de lenguajes, en vez de solamente lenguaje, porque consiste en realidad de dos lenguajes: COOL, para controlar la sincronización de hilos(threads), y RIDL, para programar la interacción entre componentes remotos. Estos dos lenguajes se diseñaron de manera independiente de un lenguaje componente. Sin embargo establecen un número de condiciones sobre el lenguaje componente.

El diseño de D es semi-independiente del lenguaje componente, ya que D impone requerimientos sobre el lenguaje componente que satisfacen naturalmente los lenguajes orientados a objetos. Luego el lenguaje componente puede ser cualquiera mientras sea orientado a objetos. Por lo tanto, en teoría podría ser implementado con C++, Smalltalk, CLOS, Java o Eiffel. Cristina Lopes, principal diseñadora del lenguaje, implementó D sobre Java llamando al lenguaje resultante DJ.

La sincronización es un ítem que D captura como un aspecto separado. Por lo tanto las clases no pueden tener código para el control de concurrencia. Un programa puede tener varios hilos concurrentes. La estrategia de sincronización por defecto, sin la intervención de COOL, es que no hay estrategia: en la presencia de múltiples hilos todos los métodos de todos los objetos pueden ejecutarse concurrentemente.

La integración remota es el otro ítem que D captura como un aspecto separado. Luego las clases tampoco pueden tener código para la comunicación remota. Un programa sin la intervención de RIDL no es un programa distribuido, la estrategia de comunicación por defecto es que no hay ninguna estrategia de comunicación, entonces un programa es distribuido sólo si utiliza RIDL.

Los módulos de aspectos pueden acceder a los componentes. Este acceso sin embargo está especificado por un protocolo el cual forma parte del concepto de aspecto. Este protocolo puede ser diferente para cada aspecto debido a que diferentes aspectos influyen de maneras diferentes sobre los componentes. En caso de COOL y RIDL los derechos de acceso son idénticos, se les permite inspeccionar al objeto pero no invocar sus métodos o modificar su estado.

Con respecto a la herencia, los módulos de aspectos se relacionan con la herencia de clases a través de simples reglas: Sea *C* una clase y *A* el módulo de aspecto asociado a *C*, se verifican las siguientes propiedades:

- Visibilidad de campos: Los elementos heredados desde ancestros de *C* son visibles a *A*.
- No hay propagación de efectos: *A* no afecta a ninguna superclase de *C*.
- Redefinición de semántica: *A* redefine completamente cualquier módulo de aspecto del mismo tipo definido en una superclase de *C*. No hay ninguna relación entre *A* y los módulos de aspectos de las superclases de *C*.
- Herencia de coordinación: *A* afecta todas las subclases de *C* que no tienen asociado un módulo de aspecto del mismo tipo. *A* no afecta los nuevos métodos definidos en una subclase de *C*. Si un método *m* declarado en *C* es redefinido por una subclase de *C* sin asociarle otro módulo de aspecto del mismo tipo entonces *A* también es el aspecto para ese método.

No es posible para los módulos de aspectos referir a otro módulo de aspecto, como consecuencia no es posible establecer ninguna relación entre los módulos de aspectos, incluyendo a la herencia.

3.2.1 COOL

COOL (COOrdination aspect Language) es un lenguaje de aspectos de dominio específico para tratar con la exclusión mutua de hilos, sincronización, suspensión y reactivación de hilos.

Un programa COOL consiste de un conjunto de módulos coordinadores. Los módulos coordinadores, o directamente coordinadores, se asocian con las clases a través del nombre. Un mismo coordinador puede coordinar a más de una clase. La unidad mínima de sincronización es el método. La declaración de un coordinador describe la estrategia de coordinación. Los coordinadores no son clases: utilizan un lenguaje diferente, no pueden ser instanciados y sirven para un propósito muy específico. Los coordinadores se asocian automáticamente con las instancias de clases que coordinan en tiempo de instanciación, y durante el tiempo de vida de los objetos esta relación tiene un protocolo bien definido.

Los coordinadores tienen conocimiento de las clases que coordinan para definir la mejor estrategia de coordinación posible. Sin embargo, las clases no tienen conocimiento de los aspectos, es decir que dentro de una clase no es posible nombrar a un coordinador.

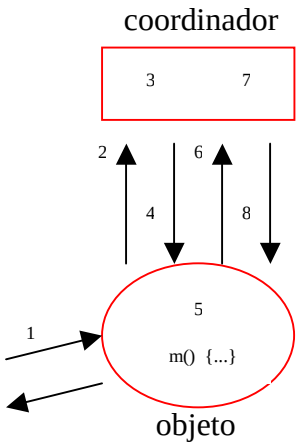
La asociación entre los objetos y los coordinadores es uno-a-uno por defecto y recibe el nombre de coordinación “per object”. Sin embargo, un coordinador también puede asociarse con todos los objetos de una o más clases y recibe el nombre de coordinación “per class”. Las estrategias de sincronización se expresan de forma declarativa.

El cuerpo de un coordinador contiene declaración de variables de condición y ordinarias, un conjunto de métodos auto-exclusivos, varios conjuntos de métodos mutuamente exclusivos y manejadores de métodos. Los métodos nombrados en el cuerpo deben ser métodos válidos de las clases coordinadas.

Un coordinador asociado a una clase *C* tiene la siguiente visibilidad:

- Todos los métodos públicos, protegidos y privados de *C*.
- Todos los métodos no privados de las superclases de *C*.
- Todas las variables privadas, protegidas y públicas de *C*.
- Todas las variables no privadas de las superclases de *C*.

La siguiente figura describe el protocolo de comunicación entre un coordinador y un objeto:



1: Dentro de un hilo *T*, se produce una invocación al método *m* del objeto.
2: El pedido es enviado primero al coordinador del objeto.
3: El coordinador controla los requerimientos de exclusión y las precondiciones para el método *m*. Si alguno de los requerimientos no se satisface, se suspende *T*. Cuando todas las condiciones se cumplen, T obtiene el derecho de ejecutar el método *m*.
4: El pedido se envia al objeto.
5: El hilo *T* ejecuta el método *m*.
6: Al retornar de la invocación del método, el resultado se presenta al coordinador.
7: El coordinador ejecuta sus acciones de salida.
8: La invocación del método finalmente retorna.

Figura 6: Protocolo de comunicación de un coordinador en COOL

3.2.2 RIDL

RIDL (Remote Interaction and Data transfers aspect Language) es un lenguaje de aspectos de dominio específico que maneja la transferencia de datos entre diferentes espacios de ejecución.

Un programa RIDL consiste de un conjunto de módulos de portales. Los módulos de portales o directamente portales se asocian con las clases por el nombre. Un portal es el encargado de manejar la interacción remota y la transferencia de datos de la clase asociada a él, y puede asociarse como máximo a una clase. La unidad mínima de interacción remota es el método.

La declaración de los portales identifica clases cuyas instancias pueden invocarse desde espacios remotos. Dichas instancias se llaman objetos remotos. La declaración de un portal identifica qué métodos de una clase serán exportados sobre

la red. En el portal estos métodos se llaman operaciones remotas. Para cada una de estas operaciones se describe qué objetos remotos esperan y qué datos enviarán a los llamadores.

Los portales no son clases: utilizan un lenguaje diferente, no pueden ser instanciados y sirven para un propósito muy específico. Tampoco son tipos en el sentido estricto de la palabra. Un portal se asocia automáticamente con una instancia de la clase en el momento que una referencia a esa instancia se exporta fuera del espacio donde la instancia fue creada. Durante el tiempo de vida de la instancia esta relación se mantiene mediante un protocolo bien definido.

Un portal asociado a una clase *C* tiene la siguiente visibilidad:

- Todos los métodos públicos, protegidos y privados de *C* a excepción de los métodos estáticos.
- Todos los métodos no privados de las superclases de *C* a excepción de los métodos estáticos.
- Todas las variables privadas, protegidas y públicas de todas las clases de una aplicación D.

Este último punto establece una dependencia explícita entre los portales y la relaciones estructurales completas de las clases. Esta dependencia expone la necesidad de controlar la transferencia de datos entre los distintos espacios de ejecución.

A continuación se describe el protocolo de comunicación entre un portal y un objeto:

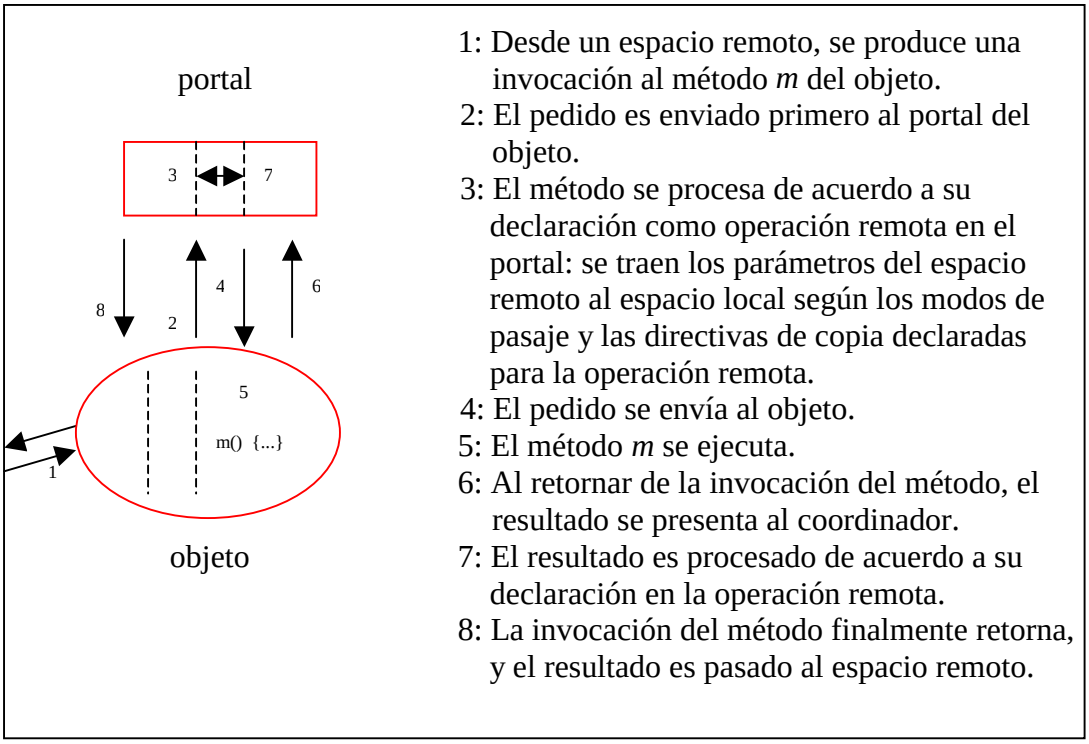


Figura 7: Protocolo de comunicación de un portal en RIDL.

Nota: Tanto el objeto como el portal se dividen con las líneas punteadas para representar los manejadores locales y remotos.

3.3 ASPECTC

AspectC[27] es un simple lenguaje de aspectos de propósito general que extiende 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 objetivo señalar llamadas de funciones particulares, acceder a los parámetros de dichas llamadas, y adherir avisos a ellas.

Los cortes en AspectC tomas las siguientes formas:

- Llamadas a una función: *call(f(arg))*, captura todas las llamadas a la función f con un argumento.
- Durante el flujo de control: *cflow(cualquier corte)*, captura el contexto de ejecución dinámico del corte.
- Referencias a una variable: v*arref(nombre_variable)*, captura las referencias a la variable nombre_variable.
- Todos los cortes se pueden describir utilizando expresiones lógicas, aumentando la expresividad del lenguaje: el operador "y"(&&), el operador "o" (||), y el operador de negación(!). Un ejemplo sería: call(f(arg)) || call(h(arg)), con lo cual se captura las llamadas a la función f o las llamadas a la función g.

Como el lenguaje C es de naturaleza estática, el tejedor de AspectC es estático.

Es interesante contar con esta herramienta de aspectos basada en C, debido a que C es utilizado en numerosas aplicaciones, en especial, aquellas donde la eficiencia es primordial, como por ejemplo, la implementación de sistemas operativos. Al existir también conceptos entrecruzados en la implementación de sistemas operativos, AspectC pasa a ser la herramienta ideal para tal desarrollo. Un trabajo a mencionar sería el proyecto *a-kernel* [27], cuya meta es determinar si la programación orientada a aspectos puede optimizar la modularidad de los sistemas operativos, y reducir así la complejidad y fragilidad asociada con la implementación de los mismos.

3.4 ASPECTS

AspectS[26] extiende el ambiente Squeak/Smalltalk para permitir un sistema de desarrollo orientado a aspectos. Squeak es una implementación abierta y portable de Smalltalk-80 cuya máquina virtual está completamente escrita en Smalltalk, para mayor referencia visitar su página [38]. Principalmente, AspectS está basado en dos proyectos anteriores: AspectJ de Xerox Parc y el MethodWrappers de John Brant,

que es un mecanismo poderoso para agregar comportamiento a un método compilado en Squeak.

AspectS, un lenguaje de aspectos de propósito general, utiliza el modelo de lenguaje de AspectJ y ayuda a descubrir la relación que hay entre los aspectos y los ambientes dinámicos. Soporta programación en un metanivel, manejando el fenómeno de *Código Mezclado* a través de módulos de aspectos relacionados. Está implementado en Squeak sin cambiar la sintaxis, ni la máquina virtual.

En este lenguaje los aspectos se implementan a través de clases y sus instancias actúan como un objeto, respetando el principio de uniformidad. Un aspecto puede contener un conjunto de receptores, enviadores o clases enviadoras. Estos objetos se agregan o se remueven por el cliente y serán usados por el proceso de tejer en ejecución para determinar si el comportamiento debe activarse o no.

En Squeak, la interacción de los objetos está basada en el paradigma de pasaje de mensajes. Luego, en AspectS los puntos de enlace se implementan a través de envíos de mensajes. Los avisos asocian fragmentos de código de un aspecto con cortes y sus respectivos puntos de enlace, como los objetivos del tejedor para ubicar estos fragmentos en el sistema. Para representar esos fragmentos de código se utilizan bloques(instancias de la clase BlockContext).

Los tipos de avisos definibles en AspectS son:

- Antes y después de la invocación a un método (*AsBeforeAfterAdvice*).
- Para manejo de excepciones(*AsHandlerAdvice*).
- Durante la invocación de un método(*AsAroundAdvice*).

Un calificador de avisos (*AsAdviceQualifier*) es usado para controlar la selección del aviso apropiado. Es similar al concepto de designadores de cortes de AspectJ.

Utiliza un tejedor dinámico que transforma el sistema base de acuerdo a lo especificado en los aspectos. El código tejido se basa en el MethodWrapper y la meta-programación. MethodWrapper es un mecanismo que permite introducir código que es ejecutado antes, después o durante la ejecución de un método.

El proceso de tejer sucede cada vez que una instancia de aspectos es instalada. Para revertir los efectos de un aspecto al sistema, el aspecto debe ser desinstalado. A este proceso se lo conoce como “destejer”, del inglés unweaving. El tejido de AspectS es completamente dinámico ya que ocurre en ejecución.

3.5 ASPECTC++

AspectC++[29] es un lenguaje de aspectos de propósito general que extiende el lenguaje C++ para soportar el manejo de aspectos. En este lenguaje los puntos de enlace son puntos en el código componente donde los aspectos pueden interferir. Los puntos de enlaces son capaces de referir a código, tipos, objetos, y flujos de control.

Las expresiones de corte son utilizadas para identificar un conjunto de puntos de enlaces. Se componen a partir de los designadores de corte y un conjunto de operadores algebraicos. La declaración de los avisos es utilizada para especificar código que debe ejecutarse en los puntos de enlace determinados por la expresión de corte.

La información del contexto del punto de enlace puede exponerse mediante cortes con argumentos y expresiones que contienen identificadores en vez de nombres de tipos, todas las veces que se necesite.

Diferentes tipos de aviso pueden ser declarados, permitiendo que el aspecto introduzca comportamiento en diferentes momentos: el aviso después (after advice), el aviso antes (before advice) y el aviso durante (around advice).

Los aspectos en AspectC++ implementan en forma modular los conceptos entrecruzados y son extensiones del concepto de clase en C++. Además de atributos y métodos, los aspectos pueden contener declaraciones de avisos. Los aspectos pueden derivarse de clases y aspectos, pero no es posible derivar una clase de un aspecto.

La arquitectura del compilador de AspectC++ se muestra en la figura 8. Primero el código fuente de AspectC++ es análizado léxica, sintáctica y semánticamente. Luego se ingresa a la etapa de planificación, donde las expresiones de corte son evaluadas y se calculan los conjuntos de puntos de enlace. Un plan para el tejedor es creado conteniendo los puntos de enlace y las operaciones a realizar en estos puntos. El tejedor es ahora responsable de transformar el plan en comandos de manipulación concretos basados en el árbol sintáctico de C++ generado por el analizador PUMA. A continuación el manipulador realiza los cambios necesarios en el código. La salida del manipulador es código fuente en C++, con el código de aspectos “tejido” dentro de él. Este código no contiene constructores del lenguaje AspectC++ y por lo tanto puede ser convertido en código ejecutable usando un compilador tradicional de C++.

Si bien el manejo de aspectos en AspectC++ es similar al manejo que proporciona AspectJ, introduce modificaciones importantes dentro del modelo de puntos de enlace, permitiendo puntos de enlace sobre clases, objetos, y sobre el flujo de control. Esto resulta en un diseño del lenguaje más coherente con el paradigma de aspectos.

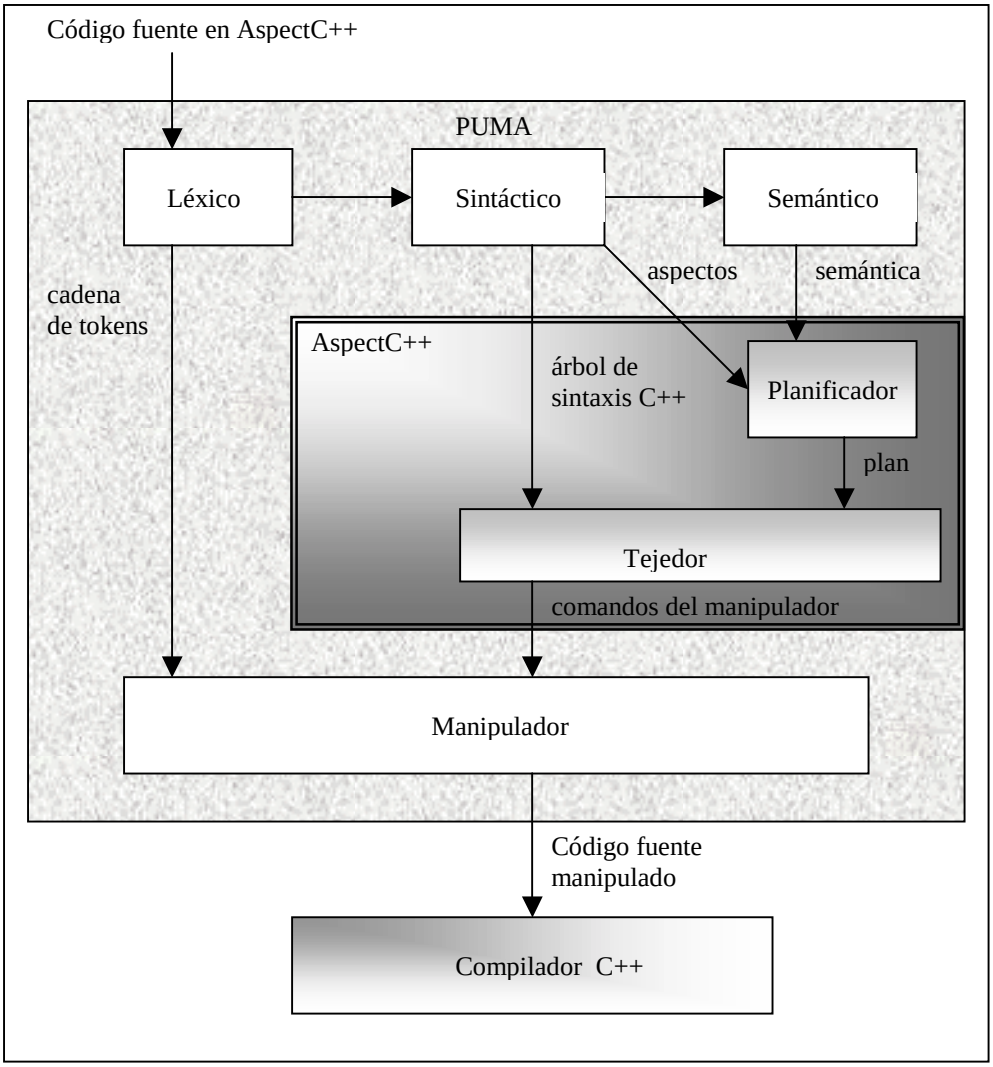


Figura 8: Arquitectura del compilador de AspectC++

3.6 MALAJ

Malaj [24,25] es un sistema que soporta la programación orientada a aspectos. Define constructores lingüísticos separados para cada aspecto de dominio específico, donde el código de los aspectos tiene una visibilidad limitada del código funcional, reduciendo los posibles conflictos con las características lingüísticas tradicionales y también, con el principio de encapsulación.

Malaj es un lenguaje orientado a aspectos de dominio específico, concentrándose en dos aspectos: sincronización y relocación. Puede verse como un sucesor de los lenguajes COOL y RIDL por su filosofía, enfatizando la necesidad de restringir la visibilidad de los aspectos, y reglas claras de composición con los constructores tradicionales. Para cada aspecto, provee un constructor lingüístico distinto, limitando así la visibilidad del aspecto sobre el módulo funcional asociado a él. Esto último se logra al estudiar cuidadosamente la relación entre el código funcional y un aspecto dado.

El lenguaje base de Malaj es una versión restringida de Java, donde se han removido los servicios que proveen los aspectos mencionados. Los servicios removidos son: la palabra clave synchronized, y los métodos wait, notify, and notifyAll.

Para el aspecto de sincronización Malaj provee el constructor *guardian.* Cada guardián es una unidad distinta con su propio nombre, y se asocia con una clase en particular (esto es, "vigila" esa clase) y expresa la sincronización de un conjunto relacionado de métodos de esa clase, es decir, que el guardián de una clase *V* representa básicamente el conjunto de métodos sincronizados de *V*. Los guardianes no pueden acceder a los elementos privados de la clase que vigilan, y el acceso a los atributos públicos y protegidos se limita a un acceso de sólo lectura. El comportamiento adicional en un guardián para un método *m* de la clase asociada se especifica introduciendo código que se ejecutará antes o después de *m*, a través de las cláusulas before y after. Una última característica de los guardianes es que pueden heredarse. Para reducir el problema de anomalía de herencia [10] las cláusulas before y after en una clase pueden referirse a las cláusulas before y after de su clase padre a través de la sentencia **super**.

El aspecto de relocación involucra el movimiento de objetos entre sitios en un ambiente de redes. Este tipo de relación es claramente dinámico. Para este aspecto Malaj provee el constructor *relocator*, que llamaremos relocador. Un relocador será una unidad diferente con su propio nombre, y se asocia con una clase en particular. Las acciones de relocación pueden ejecutarse antes o después de la ejecución de un método. Para modelar este comportamiento, el relocador brinda cláusulas before y after, que permiten la especificación deseada. También la visibilidad del relocator es limitada. Como el constructor guardián, un relocador puede heredarse, y las cláusulas before y after en una clase pueden referirse a las cláusulas before y after de su clase padre a través de la sentencia **super**, y así reducir el impacto de la anomalía de herencia[10].

Como conclusión Malaj provee una solución intermedia entre flexibilidad y poder por un lado, y entendimiento y facilidad de cambio por el otro. No permite describir cualquier aspecto, pero sí captura el comportamiento de dos conceptos relacionados con el código funcional. El próximo paso en Malaj es extenderlo para abarcar otros aspectos.

3.7 HYPERJ

La aproximación por Ossher y Tarr sobre la separación multidimensional de conceptos (MDSOC en inglés) es llamada hyperspaces, y como soporte se construyó la herramienta HyperJ[28] en Java.

Para analizar con mayor profundidad HyperJ es necesario introducir primero cierta terminología relativa a MDSOC :

- Un *espacio de concepto* concentra todas las unidades, es decir todos los constructores sintácticos del lenguaje, en un cuerpo de software, como una librería. Organiza las unidades en ese cuerpo de software

para separar todos los conceptos importantes, describe las interrelaciones entre los conceptos e indica cómo los componentes del software y el resto del sistema pueden construirse a partir de las unidades que especifican los conceptos.

- En HyperJ un *hiperespacio* (hyperspace) es un *espacio de concepto* especialmente estructurado para soportar la múltiple separación de conceptos. Su principal característica es que sus unidades se organizan en una matriz multidimensional donde cada eje representa una dimensión de concepto y cada punto en el eje es un concepto en esa dimensión.
- Los *hiperslices* son bloques constructores; pueden integrarse para formar un bloque constructor más grande y eventualmente un sistema completo.
- Un *hipermódulo* consiste de un conjunto de *hiperslices* y conjunto de reglas de integración, las cuales especifican cómo los *hiperslices* se relacionan entre ellos y cómo deben integrarse.

Una vez introducida la terminología se puede continuar con el análisis de HyperJ. Esta herramienta permite componer un conjunto de modelos separados, donde cada uno encapsula un concepto definiendo e implementando una jerarquía de clases apropiada para ese concepto. Generalmente los modelos se superponen y pueden o no referenciarse entre ellos. Cada modelo debe entenderse por sí solo. Cualquier modelo puede aumentar su comportamiento componiéndose con otro: HyperJ no exige una jerarquía base distinguida y no diferencia entre clases y aspectos, permitiendo así que los *hiperslices* puedan extenderse, adaptarse o integrarse mutuamente cuando se lo necesite. Esto demuestra un mayor nivel de expresividad para la descripción de aspectos en comparación con las herramientas descriptas anteriormente.

Existe una versión prototipo de HyperJ que brinda un marco visual para la creación y modificación de las relaciones de composición, permitiendo un proceso simple de prueba y error para la integración de conceptos en el sistema. El usuario comienza especificando un *hipermódulo* eligiendo un conjunto de conceptos y un conjunto tentativo de reglas de integración. HyperJ crea *hiperslices* válidos para esos conceptos y los compone basándose en las reglas que recibe. Luego el *hiperslice* resultante se muestra en pantalla, si el usuario no está conforme puede en ese momento introducir nuevas reglas o modificar las reglas existentes y así continuar este proceso de refinación hasta obtener el modelo deseado.

3.8 Tabla comparativa de las herramientas

En la siguiente tabla se encuentran resumidas las principales características de las herramientas orientadas a aspectos descriptas en las secciones anteriores:

Leng. OA	Leng. Base	Tejido	Propósito	Características salientes
JPAL	Independiente	Dinámico	General	Meta-Tejedor. Independiente del leng. base. Totalmente dinámico.
D	Cualquier lenguaje OO	Estático	Específico	Usado en la programación distribuida. Provee interacción entre aspectos y herencia.
COOL	Cualquier lenguaje OO	Estático	Específico	Describe la sincronización de hilos concurrentes. Visibilidad limitada del aspecto.
RIDL	Cualquier lenguaje OO	Estático	Específico	Modulariza la interacción remota. Visibilidad limitada del aspecto.
AspectC	C	Estático	General	Usado en la implementación orientada a aspectos de sistemas operativos.
AspectS	Squeak	Dinámico	General	Basado en el paradigma de pasaje de mensajes. Todo aspecto es un objeto.
AspectC++	C++	Estático	General	Aspectos son extensiones del concepto de clase. Descripción más natural de los puntos de enlace.
MALAJ	Java	Dinámico	Específico	Modulariza los aspectos de sincronización y relocación. Su objetivo es eliminar los conflictos entre POA y POO.
HyperJ	Java	Dinámico	General	Basado en "*hyperslices*". Permite la composición de aspectos. Ambiente visual de trabajo.

Tabla 2: Comparación entre herramientas orientadas a aspectos.

Como lo refleja la tabla anterior las tres decisiones de diseño: lenguaje base, tejido y propósito, son totalmente independientes entre sí para los LOA. Esto es, que todas las combinaciones entre las decisiones son teóricamente posibles. Aunque existe la tendencia a implementar el comportamiento del tejedor según la naturaleza del lenguaje base: si el lenguaje base es estático entonces el tejedor es estático y si el lenguaje base es dinámico entonces el tejedor es dinámico.

Hasta donde conocemos nadie ha afrontado la tarea de desarrollar un nuevo lenguaje base. Se han tomado lenguajes bases existentes a los cuales se los restringe en algunos casos, como en los lenguajes de dominio específico, y en otros se los utiliza sin modificaciones.

3.9 AspectJ

AspectJ [35] es un lenguaje orientado a aspectos de propósito general, cuya primera versión fue lanzada en 1998 por el equipo conformado por Gregor Kickzales (líder del proyecto), Ron Bodkin, Bill Griswold, Erik Hilsdale, Jim Hugunin, Wes Isberg y Mik Kersten. La versión que analizamos en este trabajo es la versión actual a la fecha (Junio de 2002) AspectJ 1.0.4; es importante hacer esta aclaración porque continuamente surgen nuevas versiones que mejoran la calidad final. Es una herramienta que está en desarrollo y las nuevas versiones pueden tanto corregir errores de su versión predecesora como modificar el lenguaje.

AspectJ es una extensión compatible de Java para facilitar el uso de aspectos por parte de los programadores de Java. 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.
- Compatibilidad de programación: la programación en AspectJ debe ser una extensión natural de la programación en Java.

Esta última meta fue una de las guías más importante a la hora de tomar decisiones sobre el lenguaje. Es estáticamente tipado y usa el sistema de tipos estático de Java.

Extiende Java para soportar el manejo de aspectos agregando a la semántica de Java cuatro entidades principales. Esto se ve reflejado en la figura 9.

- 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.
- Los *cortes* agrupan puntos de enlace y permiten exponer el contexto en ejecución de dichos puntos. Existen cortes primitivos y también definidos por el usuario.
- Los *avisos* son acciones que se ejecutan en cada punto de enlace incluido en un corte. Los avisos tienen acceso a los valores expuestos por el corte. Las tres entidades anteriores son dinámicas porque permiten definir comportamiento adicional que actuará en tiempo de ejecución.
- Las *introducciones y declaraciones* permiten cambiar la estructura de clases de un programa agregando o extendiendo interfaces y clases con nuevos atributos, constructores o métodos. Esta última entidad es estática porque afecta la signatura estática del programa.

Obtenemos entonces que AspectJ soporta tanto implementación estática como dinámica de conceptos entrecruzados.

Un aspecto es un tipo entrecruzado que encapsula cortes, avisos y las propiedades estáticas. Por tipo se entiende una unidad modular de código con una interface bien definida sobre la cual es posible razonar en tiempo de compilación.

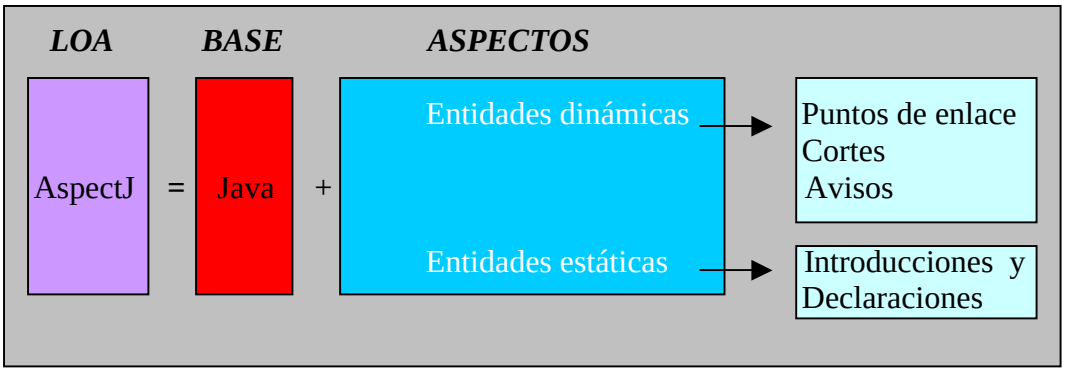


Figura 9: AspectJ es una extensión de Java para el manejo de aspectos.

Existen otras metas en la investigación del paradigma POA que AspectJ no tiene intención de abarcar. No es una traducción purista de los conceptos del paradigma, tampoco es un cálculo formal de aspectos ni tampoco representa un esfuerzo agresivo para explorar las posibilidades de un lenguaje orientado a aspectos. La intención de AspectJ es ser un LOA práctico, que provea un conjunto sólido y maduro de características orientadas a aspectos, compatible con Java para aprovechar su popularidad.

3.9.1 Puntos de enlace

Para entender el concepto de punto de enlace consideremos el siguiente ejemplo, en el cual se define una clase para manejar números complejos:

```
Class NumComplejo{
   private  real parte_imaginaria, parte_real;

NumComplejo(real x, real y){
        this.parte_imaginaria=x;
        this.parte_real=y;
}
void ingresar_parte_imaginaria(real x){ this.parte_imaginaria=x; }
void ingresar_parte_real(real y){ this.parte_real=y; }
real devolver_parte_ imaginaria(){ return parte_imaginaria; }
real devolver_parte_real(){ return parte_real; }
void aumentar_parte_real(real x) {
  real a = devolver_parte_real();
  a= a+x;
  ingresar_parte_real(a);
 }
}
```

Código 1: Clase número complejo.

Entonces lo que establece el código

void ingresar_parte_ imaginaria(real x){ this.parte_imaginaria=x; }

es que cuando el método *ingresar_parte_imaginaria* es invocado con un real como argumento sobre un objeto de tipo *NumComplejo* entonces se ejecuta el cuerpo del método. De igual forma cuando un objeto de tipo *NumComplejo* es instanciado a través de un constructor con dos argumentos de clase *real*, entonces se ejecuta el cuerpo del constructor.

El patrón que surge de esta descripción es que cuando "algo" pasa entonces "algo" se ejecuta. El conjunto de las "cosas que pasan" representan los puntos de enlace.

Los *puntos de enlace* son entonces puntos bien definidos en la ejecución de un programa y AspectJ define los siguientes conceptos:

Llamadas a métodos: Cuando un método es invocado, no incluye llamadas a **super**.
Ejemplo: cc.aumentar_parte_real_primera(5.5);. Cuando el método *aumentar_parte_real_primera* es invocado sobre el objeto *cc*.

Ejecución de un método: Cuando el cuerpo de un método se ejecuta.
Ejemplo: Cuando el cuerpo del método *aumentar_parte_real_primera* es ejecutado.

Llamada a un constructor: Cuando un objeto es creado y un constructor es invocado, sin incluir la llamada al constructor **this** o **super**.
Ejemplo: NumComplejo nc1 = new NumComplejo(3.0,5.3);. El objeto es creado por **new** y luego el constructor *NumComplejo* es invocado.

Ejecución de un inicializador: Cuando los inicializadores no estáticos de una clase se ejecutan.

Ejecución de un constructor: Cuando se ejecuta el código de un constructor, luego de su llamada al constructor **this** o **super**.
Ejemplo: Cuando el código del constructor *NumComplejo* se ejecuta.

Ejecución de un inicializador estático: Cuando se ejecuta el inicializador estático para una clase.

Pre-inicialización de un objeto: Antes que el código de inicialización para una clase particular se ejecute. Comprende el tiempo entre el comienzo de la llamada al primer constructor y el comienzo del constructor de su clase padre. Luego, la ejecución de estos puntos de enlace comprenden los puntos de enlace del código encontrado en las llamadas a constructores **this** y **super**.

Inicialización de un objeto: Cuando el código de inicialización para una clase particular se ejecuta. Comprende el tiempo entre el retorno del constructor de su

clase padre y el retorno de la llamada a su primer constructor. Incluye todos los inicializadores dinámicos y constructores usados para crear el objeto.

Referencia a un atributo: Cuando se referencia a un atributo no final. Un atributo final es un atributo que no cambia su valor una vez inicializado.

Asignación a un atributo: Cuando se realiza la asignación a un atributo.

Ejemplo: numerocomplejo.parte_imaginaria=5.5;. Cuando al atributo *parte_imaginaria* del objeto *numerocomplejo* se le asigna el valor real 5.5.

Ejecución de un manejador: Cuando el manejador de una excepción se ejecuta.

El modelo de punto de enlaces es un elemento crítico en el diseño de cualquier LOA ya que provee la interface entre el código de aspectos y el código de la funcionalidad básica. En AspectJ este modelo puede considerarse como nodos en un grafo de llamadas a objetos en ejecución. Estos nodos incluyen puntos en los cuales un objeto recibe una llamada a un método o puntos en los cuales un atributo de un objeto es referenciado. Los arcos representan el flujo de control entre los nodos. En este modelo el control pasa por cada punto de enlace dos veces, una cuando realiza el cálculo y otra al regresar del mismo.

Como ejemplo del modelo, consideremos la siguiente clase, y la figura 10:

```
Class CoordenadaCompleja {

NumComplejo NumComplejo1,NumComplejo2;
real distancia;
...

void  aumentar_parte_real_primera(real x){
  ...
  real s =  this.distancia;
  …
  NumComplejo1.aumentar_parte_real(x);
  ...
 }

...
}
```

Código 2: Clase coordenada compleja.

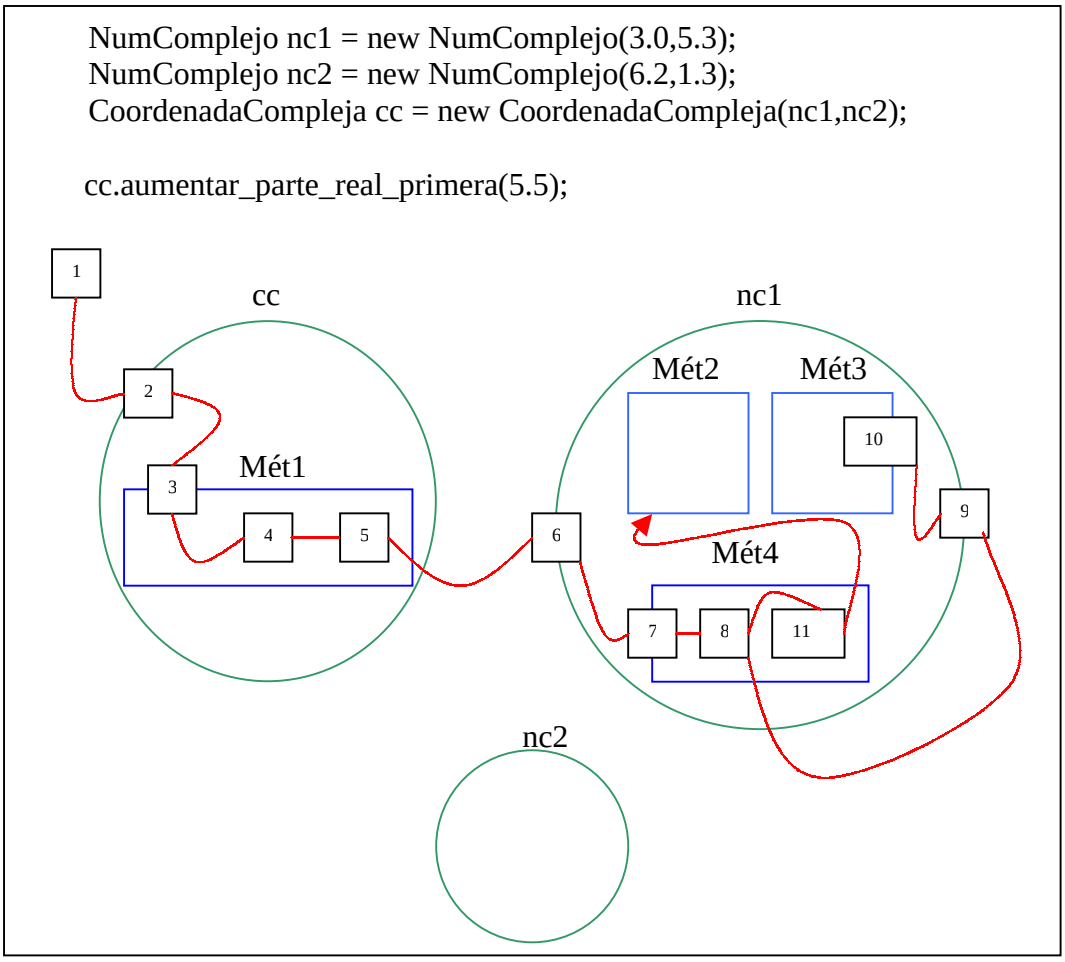


Figura 10: Modelo de puntos de enlace

= Métodos. i = Puntos de Enlace. = Objetos.

= Cálculo.

Mét1 = *aumentar_parte_real_primera*, de la clase *CoordenadaCompleja.*
Mét2 = *ingresar_parte_real*, de la clase *NumComplejo.*
Mét3 = *devolver_parte_real*, de la clase *NumComplejo.*
Mét4 = *aumentar_parte_real*, de la clase *NumComplejo.*

La ejecución de las tres primeras líneas de la figura 10 provoca la creación de los objetos *cc*, *nc1* y *nc2*. La ejecución de la última línea inicia una computación que irá siguiendo los puntos de enlace:

1. Un punto de enlace de llamada a un método, correspondiente al método *aumentar_parte_real_primera* invocado sobre el objeto *cc*.
2. Un punto de enlace de recepción de llamada a un método, en el cual *cc* recibe la llamada *aumentar_parte_real_primera*.
3. Un punto de enlace de ejecución de un método, en el cual el método *aumentar_parte_real_primera* definido en la clase *CoordenadaCompleja* empieza su ejecución.

4. Un punto de enlace de acceso a un atributo, en el cual el atributo *distancia* de *cc* es referenciado.
5. Un punto de enlace de llamada a un método, en el cual el método *aumentar_parte_real* es invocado sobre el objeto *nc1*.
6. Un punto de enlace de recepción de llamada a un método, en el cual *nc1* recibe la llamada *aumentar_parte_real*.
7. Un punto de enlace de ejecución de un método, en el cual el método *aumentar_parte_real* definido en la clase *NumComplejo* empieza su ejecución.
8. Un punto de enlace de llamada a un método, en el cual el método *devolver_parte_real* es invocado sobre el objeto *nc1*.
9. Un punto de enlace de recepción de llamada a un método, en el cual n*c1* recibe la llamada *devolver_parte_real*.
10. Un punto de enlace de ejecución de un método, en el cual el método *devolver_parte_real* definido en la clase *NumComplejo* empieza su ejecución.

El control retorna por los puntos de enlace 10 y 9.

11. Un punto de enlace de llamada a un método, en el cual el método *ingresar_parte_real* es invocado sobre el objeto *nc1*.

... y así siguiendo, hasta que finalmente el control retorna por los puntos de enlace 3, 2 y 1.

3.9.2 Cortes

Un corte es un conjunto de puntos de enlace más datos del contexto de ejecución de dichos puntos. Los cortes principalmente son usados por los avisos y pueden ser compuestos con operadores booleanos para crear otros cortes. AspectJ provee varios designadores de cortes primitivos. El programador luego puede componerlos para definir designadores de cortes anónimos o con nombres. Los cortes no son de alto orden, porque sus argumentos y resultados no pueden ser cortes ni existen designadores de cortes paramétricos.

Un designador de corte captura en ejecución varios puntos de enlace, por ejemplo el designador:

```
call(void NumComplejo.ingresar_parte_real(real))
```

captura todas las llamadas al método *ingresar_parte_real* de la clase *NumComplejo* con un argumento de clase *real*.

Otro ejemplo sería:

```
pointcut acceso():
  get(NumComplejo.parte_real) || get(CoordenadaCompleja.distancia);
```

este designador de corte captura todas las referencias al atributo *parte_real* de la clase *NumComplejo* o todas las referencias al atributo distancia de la clase *CoordenadaCompleja*. Como vemos los designadores de cortes pueden capturar puntos de enlaces de diferentes clases, esto es, entrecruzan las clases.

Cortes primitivos

AspectJ incluye una variedad de designadores de cortes primitivos que identifican puntos de enlace de diferentes formas.

Los designadores de cortes primitivos son:

Call : call(PatróndeMétodo), call(PatróndeConstructor).
Captura todos los puntos de enlace de llamadas a métodos o constructores cuyo encabezamiento coincide con el respectivo patrón.
Ejemplo: call(* NumComplejo.* (..))

captura todos los puntos de enlace de llamadas a cualquier método, que devuelve cualquier tipo y con cualquier número y tipo de argumentos, de la clase *NumComplejo*. Los patrones se refieren a expresiones como NumComplejo.* que permite identificar todos los métodos de la clase *NumComplejo*.

Execution: execution(PatróndeMétodo), execution(PatróndeConstructor).
Captura todos los puntos de enlace de ejecución de métodos o constructores cuyo encabezamiento coincide con el respectivo patrón.
Ejemplo: execution(NumComplejo.new(..))

captura la ejecución de los constructores de la clase *NumComplejo* con cualquier número de argumentos.

Get: get(PatróndeAtributo)
Captura todos los puntos de enlace de referencia a los atributos que coinciden con el patrón correspondiente.
Ejemplo: get(NumComplejo.parte_imaginaria)

captura las referencias al atributo *parte_imaginaria* de la clase *NumComplejo*.

Set: set(PatróndeAtributo)
Captura todos los puntos de enlace de asignación a los atributos que coinciden con el patrón correspondiente.
Ejemplo: set(NumComplejo.parte_*)

captura las asignaciones a los atributos de la clase *NumComplejo* que coinciden con el patrón "parte_*", en este caso coinciden *parte_imaginaria* y *parte_real*.

Initialization: initialization(PatróndeConstructor)
Captura los puntos de enlace de las inicializaciones de los objetos cuyos constructores coinciden con el patrón .

Staticinitialization: staticinitialization(PatróndeClase)

Captura los puntos de enlace de ejecución de un inicializador estático de las clases que coinciden con el patrón.

Handler: handler(PatróndeClase)

Captura los manejadores de excepción de las clases de excepciones que coinciden con el patrón.

This: this(PatróndeClase), this(identificador)

Captura todos los puntos de enlace donde el objeto actual (el objeto ligado a **this**) es una instancia de una clase que coincide con el PatróndeClase, o es instancia de una clase que coincide con la clase asociada al identificador.

Target: target(PatróndeClase), target(identificador)

Captura todos los puntos de enlace donde el objeto objetivo (el objeto sobre el cual se invoca un método o una operación de atributo) es una instancia de una clase que coincide con el PatróndeClase, o es instancia de una clase que coincide con la clase asociada al identificador.

Args: args(PatróndeClase, . . .),args(identificador, . . .)

Captura todos los puntos de enlace donde los argumentos son instancias de una clase que coincide con el PatróndeClase o con la clase del identificador. Si es un PatróndeClase entonces el argumento en esa posición debe ser instancia de una clase que coincida con el PatróndeClase. Si es un identificador entonces el argumento en esa posición debe ser instancia de una clase que coincida con la clase asociada al identificador (o cualquier clase si el identificador está asociado a Object. Este caso especial se verá nuevamente en la sección Exposición de contexto).

Ejemplo: args(*,int)

captura todos los puntos de enlace con dos argumentos, el primero puede ser cualquiera y el segundo debe ser un entero.

Within: within(PatróndeClase)

Captura todos los puntos de enlace donde el código que se está ejecutando está definido en una clase que coincide con el patrón. Incluye la inicialización de la clase, del objeto, puntos de enlaces de ejecución de métodos y constructores, y puntos de enlaces asociados con las sentencias y expresiones de la clase.

Ejemplo: within(NumComplejo)

captura todos los puntos de enlace donde el código que se está ejecutando está definido dentro de la clase *NumComplejo*.

Withincode: withincode(PatróndeMétodo),withincode(PatróndeConstructor)

Captura todos los puntos de enlace donde el código que se está ejecutando está definido en un método o constructor que coincide con el patrón correspondiente. Incluye todos los puntos de enlace de ejecución de métodos y constructores, y puntos de enlaces asociados con las sentencias y expresiones del método o constructor.

Ejemplo: withincode(real devolver_parte_real())

captura todos los puntos de enlace donde el código que se está ejecutando está definido en el método *devolver_parte_real*.

Estos dos últimos cortes primitivos, within y withincode, tratan con la estructura léxica del programa.

Cflow: cflow(Corte)
Captura todos los puntos de enlace en el flujo de control de los puntos de enlace capturados por el corte pasado como parámetro.
Ejemplo: cflow(withincode(real devolver_parte_real()))

captura todos los puntos de enlace que ocurren entre el comienzo y la finalización de los puntos de enlace especificados por withincode(real devolver_parte_real()).

Cflowbelow: cflowbelow(Corte)
Captura todos los puntos de enlace en el flujo de control debajo de los puntos de enlace capturados por el corte pasado como parámetro, sin incluir el punto de enlace inicial del flujo de control.
Ejemplo: cflowbelow(withincode(real devolver_parte_real()))

captura todos los puntos de enlace que ocurren entre el comienzo y la finalización de los puntos de enlace especificados por withincode(real devolver_parte_real()), pero no lo incluye a éste.

If: if(Expresiónbooleana)
Captura todos los puntos de enlace cuando la expresión booleana se satisface.

Cortes definidos por el programador

El programador puede definir nuevos cortes asociándoles un nombre con la declaración **pointcut**.

pointcut nombre(<Parametros_formales>)**:** <Corte>**;**
donde <Parametros_formales> expone el contexto del corte, y
< Corte> puede ser un corte primitivo o definido por el programador.

Ejemplo:
```
/*  Corte1 */
pointcut aumentar():  call(void aumentar*(…)) ;

/*  Corte2 */
 pointcut   aumentarsolocomplejo(): aumentar() && target(NumComplejo);
```

Corte1 captura todas las llamadas a un método que comience con la palabra aumentar. *Corte2* utiliza *Corte1* para capturar aquellas llamadas a métodos que empiecen con aumentar pero que sean invocados sólo sobre un objeto de clase *NumComplejo*.

Un corte nombrado puede ser definido tanto en una clase como en un aspecto y es tratado como un miembro de esa clase o aspecto. Como miembro puede tener modificadores de acceso privado o público (**private** o **public**). No se permite la sobrecarga de cortes definidos por el programador. Por lo tanto será un error de

compilación cuando dos cortes en una misma clase o aspecto tengan asociado el mismo nombre. Esta es una de las diferencias con la declaración de métodos.

El alcance de un corte definido por el programador es la declaración de la clase o aspecto que lo contiene. Es diferente al alcance de otros miembros que sólo se limitan al cuerpo de la clase o aspecto.

Los cortes pueden declararse como abstractos, y son definidos sin especificar su cuerpo. Estos cortes sólo pueden declararse dentro de un aspecto declarado abstracto.

Composición de cortes

Tanto los cortes definidos por el programador como los primitivos pueden componerse entre sí a través de la utilización de operadores algebraicos para crear nuevos cortes. Los operadores algebraicos soportados por AspectJ son: && (“y”), || (“o”), !(“no”) y los paréntesis.

Lo anterior puede modelarse a través de una BNF :

<Corte>::= <Corte_Primitivo>| <Corte_Definido_por_el_programador> |
<Corte> **&&** <Corte> |
<Corte> || <Corte> |
!<Corte> |
(<Corte>**)**

BNF 1: Definición de cortes

donde la semántica asociada a los operadores es la siguiente:

<Corte>$_1$ **&&** <Corte>$_2$: captura todos los puntos de enlace de Corte$_1$ y todos los puntos de enlace de Corte$_2$.

<Corte>$_1$|| <Corte>$_2$: captura todos los puntos de enlace de Corte$_1$ o todos los puntos de enlace de Corte$_2$.

!<Corte>: captura todos los puntos de enlace que no captura Corte.

(<Corte>**)**: captura todos los puntos de enlace que captura Corte.

Esto es, el resultado de componer uno o más cortes también es un corte.

La composición de los cortes resulta uno de los mecanismos más importantes para permitir la expresividad en el manejo de aspectos de AspectJ. La composición es sencilla y simple, a través de operadores bien conocidos. Daría la impresión que esta simplicidad tendría como efecto secundario una pérdida importante en el poder de expresividad; pero esto no es así ya que se pueden obtener expresiones tan complejas como se requiera. La composición es entonces un mecanismo muy poderoso, y aún así mantiene su simplicidad.

Como ejemplo, consideremos la siguiente implementación de una pila para la cual cada vez que un elemento es agregado, se desea mostrar la pila resultante. Para esto, se define un corte que captura todas las llamadas al método apilar:

pointcut al_apilar() : call(void Pila.Apilar(Object));

Si bien con este corte se capturan los puntos de enlace deseados, caemos en un problema de eficiencia porque en el método *Apilar_Multiple*, por cada elemento se invoca al método *Apilar*. Idealmente, en el caso de este método, la pila se debería mostrar una vez que se apilaron todos los elementos, y no cada vez que se apila un elemento.

```
Class Pila {
 private Object [] arreglo ;
 private int ultimo;
 private int capacidadMaxima=100;

 public Pila() {
 arreglo = new Object[capacidadMaxima];
 ultimo=0;
 }
 public void Apilar(Object elem) {
 arreglo[ultimo] = elem;
 ultimo++;
 }
public Object Desapilar(){
Object elem = arreglo[ultimo-1];
ultimo--; return elem;
}
public void Apilar_Multiple(Object[] arregloElem){
 for (int i=0,i<=arregloElem.length,i++){
  this.apilar(arregloElem[i] );
 }
}
public int Dar_Capacidad_Maxima(){ return capacidadMaxima;}
public int Cantidad_Elementos() {return ultimo;}
public boolean Pila_Vacia(){return (ultimo==0);}
public boolean Pila_ Llena(){return (ultimo==capacidadMaxima);}
}
```

Código 3: Definición clase pila.

Para discriminar estos dos casos y solucionar el problema, utilizamos la composición de cortes:

```
pointcut al_apilar_deauno(): al_apilar() &&
                                !withincode(void Pila.Apilar_Multiple(..));

pointcut al_apilar_multiple(): call(void Pila.Apilar_Multiple(..));
```

```
pointcut al_apilar_eficiente(): al_apilar_deauno() && al_apilar_multiple();
```

El corte al_apilar_deauno captura todas las invocaciones al método *Apilar* que no están dentro del código de *Apilar_Multiple*. El corte *al_apilar_multiple* captura todas las invocaciones al método *Apilar_Multiple*. Y el último corte, *al_apilar_eficiente*, captura los puntos de enlace del primer y segundo corte, solucionado así el problema de eficiencia.

Como vemos la composición de cortes de AspectJ nos permite modelar una situación dinámica, en el cual el punto de enlace depende del contexto dinámico de ejecución[36]: en nuestro ejemplo, dentro de todas las invocaciones al método *Apilar* necesitamos descartar las que se realicen desde el código de *Apilar_Multiple*. Para ello debemos analizar dinámicamente cada invocación al método *Apilar* y descartar aquellas que estén en el contexto de *Apilar_Multiple*.

Otro ejemplo más sencillo de composición sería obtener los momentos donde otros objetos requieran el valor de los atributos de la pila, en particular *capacidadMaxima* y *ultimo*. Esto se logra con el siguiente corte:

```
pointcut atributos_leidos(): (call (int Pila.Dar_Capacidad_Maxima(..))  ||
                              call (int Pila. Cantidad_Elementos(..))   ) &&
                              !this(Pila) ;
```

El corte *atributos_leidos* captura todas las llamadas a los métodos *Dar_Capacidad_Maxima* y *Cantidad_Elementos* tal que el llamador no es un objeto de la clase *Pila*.

Exposición de contexto

Los cortes pueden exponer el contexto de ejecución en sus puntos de enlace a través de una interface. Dicha interface consiste de parámetros formales en la declaración de los cortes definidos por el programador, análogo a los parámetros formales en un método.

Una lista de parámetros vacía, como en el ejemplo anterior, significa que cuando los eventos ocurren ningún contexto está disponible, en cambio

```
pointcut atributos_leidos(Pila p): (call (int Pila.Dar_Capacidad_Maxima(..)) ||
                                    call (int Pila. Cantidad_Elementos(..))   ) &&
                                    !this(Pila) && target(p);
```

sería obtener los momentos donde otros objetos requieran el valor de los atributos de la pila y disponer del objeto de clase *Pila* receptor de los mensajes a través del parámetro *p*, que es instanciado por el corte primitivo target. El contexto disponible en este caso es *p*.

En la parte derecha de la declaración de un corte o de un aviso se permite un identificador en lugar de especificar una clase o PatróndeClase. Existen tres cortes primitivos donde esto se permite: this, target y args. Usar un identificador en vez de un PatróndeClase es como si la clase seleccionada fuera de la clase del parámetro

formal, por lo tanto en el ejemplo anterior target(p) captura todos los objetos receptores de clase *Pila*.

Otro ejemplo de exposición de contexto sería:

```
pointcut puntosiguales(Punto p):  target(Punto) && args(p)
                                  && call(boolean igual(Object ));
```

donde suponemos definida la clase *Punto* con un método público *igual* que recibe como parámetro otro objeto de clase *Punto* y retorna un valor booleano estableciendo si las coordenadas de ambos objetos son iguales o no.

El corte tiene un parámetro de clase *Punto*, entonces cuando los eventos descriptos en la parte derecha ocurren un objeto p de clase *Punto* está disponible. Pero en este caso si miramos con atención la parte derecha el objeto disponible no es el objeto receptor del método sino el objeto pasado como parámetro en dicho método. Si queremos acceder a ambos objetos el corte debe ser definido como sigue:

```
pointcut puntosiguales(Punto p1,Punto p2):  target(p1) && args(p2)
                                  && call(boolean igual(Object ));
```

donde *p1* expone el objeto de clase *Punto* receptor del método *igual* y *p2* expone el objeto de clase *Punto* que es pasado como parámetro al método *igual* .

La definición de los parámetros en un corte es flexible. Sin embargo la regla más importante a tener en cuenta es que cuando cada uno de los eventos definidos ocurre todos los parámetros del corte deben ligarse a algún valor. Por consiguiente este corte daría un error de compilación :

```
pointcut enDosPilas(Pila p1, Pila p2) :
            (target(p1) && call(void Pila.Apilar(..))) ||
            (target(p2) && call(Object Pila.Desapilar(..)));
```

El corte anterior captura las llamadas al método *Apilar* en un objeto *p1* de clase *Pila* o las llamadas al método *Desapilar* en un objeto *p2* de clase *Pila*. El problema es que la definición de los parámetros intenta acceder a dos objetos de clase *Pila*. Pero cuando el método *Apilar* es llamado sobre un objeto no existe otro objeto de clase de *Pila* accesible, por lo que el parámetro *p2* quedaría sin un valor ligado y de ahí el error de compilación. La otra situación sería cuando el método *Desapilar* es llamado sobre un objeto y el parámetro *p1* quedaría sin un valor asociado resultando en el mismo error.

Un caso especial de exposición de contexto se da al utilizar la clase Object:

```
pointcut  llamadaspublicas( ): call(public *.*(..)) && args(Object);
```

este corte captura todos los métodos públicos unarios que toman como único argumento subclases de Object pero no los tipos primitivos como **int**. Pero :

```
pointcut  llamadaspublicas(Object o): call(public *.*(..)) && args(o);
```

captura todos los métodos unarios con cualquier clase de argumento, si el argumento fuera de tipo **int** el valor ligado a *o* sería de la clase java.lang.Integer.

Patrones

En esta sección se describen los patrones que permite especificar AspectJ.

Un PatróndeMétodo es una descripción abstracta que permite agrupar uno o más métodos según su encabezamiento.

Un patrón para los métodos sigue el siguiente esquema:

<PatróndeMétodo>::=
[<ModificadoresdeMetodos>]
<PatróndeClase>[<PatróndeClase>**.**]
<PatróndeIdentificador>**(**<PatrondeClase>**,...)**[**throws**<PatróndeExcepción>]

BNF 2: Patrones de métodos

El esquema incluye los modificadores de métodos como **static**, **private** o **public**, luego las clases de los objetos retornados, las clases a las cuales pertenece el método, el nombre del método, que puede ser un patrón, los argumentos, que también pueden ser patrones, y la cláusula throws que corresponda.

La lista de parámetros formales puede utilizar el wildcard ".." para indicar cero o más argumentos. Luego

execution(void m(..,int))

captura los métodos *m* cuyo último parámetro es de tipo entero, incluyendo el caso de que tenga un sólo parámetro entero. Mientras que

execution(void m(..))

captura todos los métodos *m* con cualquier cantidad de argumentos.

Los nombres de métodos pueden contener el wildcard "*" que indica cualquier número de caracteres en el nombre del método. Luego

call(boolean *())

captura todos los métodos que retornan un valor booleano sin tener en cuenta el nombre de los métodos. En cambio

call(boolean dar*())

captura todos los métodos que retornan un valor booleano cuyo nombre empiece con el prefijo "dar". También se puede utilizar este wildcard en el nombre de la clase que contiene al método.

call(*.Apilar(..))

captura todas las llamadas al método *Apilar* definido en cualquier clase. Mientras que

call(Pila.Apilar(..))

captura todas las llamadas al método *Apilar* definido en únicamente en la clase *Pila*. De no estar presente un nombre de clase se asume el wildcard "*" indicando cualquiera de las clases definidas.

Un PatróndeClase es un mecanismo para agrupar un conjunto de clases y usarlo en lugares donde de otra forma se usaría una sola clase. Las reglas para usar los patrones de clase son simples.

Todos los nombres de clases son patrones de clase. Existe un nombre especial "*" que también es un patrón. El "*" captura todas las clases y también los tipos primitivos. Luego

call(void Apilar(*))

captura todas las llamadas a los métodos *Apilar* que tienen un único argumento de cualquier clase.

Los patrones de clase pueden también utilizar los wildcard "*" y "..". El wildcard "*" indica cero o más caracteres a excepción del punto(".". Luego puede ser usado cuando las clases respetan una convención para formar sus nombres.

call(java.awt.*)

captura todas las clases que comiencen con "java.awt." y no tengan más puntos. Esto es captura las clases en el paquete java.awt pero no clases interiores como java.awt.datatransfer o java.awt.peer.

El wildcard ".." indica cualquier secuencia de caracteres que comienza y termina con punto (".") , por lo tanto puede ser utilizado para agrupar todas las clases en cualquier subpaquete o todas las clases internas. Luego

target(java.awt..*)

captura todos los puntos de enlaces donde el objetivo es de una clase que comienza con "java.awt.". Incluye clases definidas en java.awt tanto como clases definidas en java.awt.datatransfer o java.awt.peer, entre otras.

A través del wildcard "+" es posible agrupar todas las subclases de una clase o conjunto de clases. Este wildcard sigue inmediatamente al nombre de una clase. Entonces mientras que

call (Pila.new())

captura todas las llamadas a constructores donde una instancia exclusivamente de la clase *Pila* se crea,

call(Pila+.new())

captura todas las llamadas a constructores donde una instancia de cualquier subclase de la clase *Pila*, incluyendo a ella misma, es creada.

Los patrones de clase pueden componerse utilizando los operadores lógicos **&&** ("y"), || ("o"), **!**("no") y los paréntesis.

Formalmente, la sintaxis de un PatróndeClase es especificada por la siguiente BNF.

<PatróndeClase>::= <PatróndeIdentificador> [+] [**[] .. ***] |
!<PatróndeClase> |
<PatróndeClase> **&&** <PatróndeClase> |
<PatróndeClase> || <PatróndeClase>
(<PatróndeClase>**)**

BNF 3: Patrones de clase.

3.9.3 Avisos

Los avisos definen el código adicional que se ejecuta en los puntos de enlace capturados por los cortes. Los avisos definen el comportamiento que entrecruza toda la funcionalidad, están definidos en función de los cortes . La forma en que el código del aviso es ejecutado depende del tipo del aviso.

AspectJ soporta tres tipos de avisos. El tipo de un aviso determina cómo éste interactúa con el punto de enlace sobre el cual está definido. AspectJ divide a los avisos en:

- Aviso "Antes"(Before): aquellos que se ejecutan antes del punto de enlace.
- Aviso "Después" (After): aquellos que se ejecutan después del punto de enlace.
- Aviso "Durante" (Around): aquellos que se ejecutan en lugar del punto de enlace.

El aviso "antes" no agrega mayores dificultades. Por ejemplo, consideremos ahora agregar los controles al método *Apilar* de la clase *Pila* definida anteriormente para que no efectúe la operación *Apilar* si es el caso de qué la pila esté llena.

pointcut al_apilar(Pila p) : call(void Pila.Apilar(..)) && target (p);

```
before(Pila p):al_apilar(p){
    if(p.Pila_Llena()){
        throw new ExcepcionPilaLlena();
    }
}
```

De igual forma podemos controlar también que la operación *Desapilar* no se realice cuando la pila está vacía.

```
pointcut al_desapilar(Pila p) : call(Object Pila.Desapilar(..))
                                && target(p);

before(Pila p):al_desapilar(p){
    if(p.Pila_Vacia()){
        throw new ExcepcionPilaVacia();
    }
}
```

El aviso "después" se puede subdividir según como sea el retorno del método: normal, señalando una excepción o sin importar el retorno. Como ejemplo consideremos nuevamente el problema de mostrar la pila cada vez que se agrega un elemento de manera eficiente.

```
pointcut al_apilar_deauno(Pila p): al_apilar(p)
        &&  !withincode(void Pila.Apilar_Multiple(..));

pointcut al_apilar_multiple(Pila p): call(void Pila.Apilar_Multiple(..))
                                && target(p);

pointcut al_apilar_eficiente(Pila p):
                    al_apilar_deauno(p) && al_apilar_multiple(p);
```

Completemos el ejemplo agregando el comportamiento adicional para mostrar la pila.

```
after (Pila p):al_apilar_eficiente(p){
    dibujaEstructura dibujante =new dibujaEstructura();
    dibujante.dibujarPila(p);
}
```

Una vez que se efectúe una modificación en la pila, en este caso agregar un elemento, se insertará el código necesario para dibujar la pila. Suponemos definida una clase *dibujaEstructura* la cual muestra en pantalla diversas estructuras como pilas, listas, tablas hash, etc. En este aviso no importa la forma en que retornan los métodos *Apilar* y *Apilar_Multiple*.

```
After () throwing  (Exception e):call( public *.*(..)) {
    contadorDeExcepciones++;
    System.out.println(e);  }
```

Este aviso se ejecuta cada vez que se genera una excepción en un método público definido en cualquier clase, aumenta el contador de excepciones y muestra en pantalla la excepción. Este aviso tiene en cuenta la forma de retorno de los métodos, cuando retornan señalando una excepción, y señala nuevamente la excepción una vez que se ejecuta su código.

```
After () returning (int cantidaddeelementos):
     call(int Pila.Cantidad_Elementos(..)){
System.out.println("Retornando un valor entero"+cantidaddeelementos);
}
```

Este aviso se ejecuta luego de cada punto de enlace capturado por el corte pero sólo en el caso de que su retorno sea normal. El valor retornado puede accederse a través del identificador *cantidaddeelementos*. Una vez que se ejecutó el aviso se retorna el valor.

El aviso "durante" se ejecuta en lugar del punto de enlace asociado al mismo y no antes o después de éste. El aviso puede retornar un valor, como si fuera un método y por lo tanto debe declarar un tipo de retorno. El tipo de retorno puede ser **void**, indicando el caso en que en el aviso no se desee devolver valor alguno; en esta situación el aviso retorna el valor que devuelve el punto de enlace.

En el cuerpo del aviso se puede ejecutar la computación original del punto de enlace utilizando la palabra reserva **proceed(**<Lista_Argumentos>**)**, que toma como argumentos el contexto expuesto por el aviso y retorna el valor declarado en el aviso.

```
int around(int i) : call(int Numero.suma(int)) && args(i) {
                    int  nuevoRetorno = proceed(i+1);
                    return nuevoRetorno*3;
                     }
```

Este aviso captura todas las llamadas al método suma, y en vez de ejecutar directamente el cuerpo del método, empieza a computar el código del aviso. Este código reformula la llamada, cambiando el argumento de entrada *i* por *i+1*; invoca al método *suma* a través de **proceed** y finalmente el aviso retorna la respuesta multiplicada por tres.

Si el valor de retorno del aviso es de clase Object, entonces el resultado del **proceed** es convertido a una representación Object, aún cuando ese valor sea un tipo primitivo. Sin embargo, cuando un aviso retorna un valor de clase Object, ese valor es convertido nuevamente a su representación original. Luego, el siguiente aviso es equivalente al anterior:

```
Object around(int i) : call(int Numero.suma(int)) && args(i) {
                    Integer  nuevoRetorno = proceed(i+1);
                    return new Integer(nuevoRetorno*3);
                     }
```

En todos los tipos de avisos todos los parámetros se comportan exactamente como los parámetros de los métodos. La semántica del pasaje de parámetros es

pasaje de parámetros por valor. La única manera de cambiar los valores de los parámetros o los valores de retorno de los puntos de enlace es utilizando el aviso "durante".

La declaración de avisos puede describirse formalmente a partir de la siguiente BNF.

<Aviso>::=<Tipo_aviso> **:** <Corte> **{**<Cuerpo>**}**
<Tipo_aviso>::= <Aviso_antes> | <Aviso_despues> | <Aviso_durante>
<Aviso_antes>::= **before (**<Parametros_formales>**)**

<Aviso_despues> ::= **after (**<Parametros_formales>**)** <Forma_terminacion>
<Forma_terminacion>::= **returning** [**(**<Parametro_formal>**)**] |
throwing [**(**<Parametro_formal>**)**] |
λ

<Aviso_durante>::= <Nombre_Clase> **around (**<Parametros_formales>**)**
[**throws** <Lista_Clases>]

BNF 4: Definición de avisos

Modelo de comportamiento

Múltiples avisos se pueden aplicar a un mismo punto de enlace. Para determinar el orden de aplicación de los avisos se define una relación de precedencia entre ellos. Las reglas de precedencia se describirán en la sección 3.9.5 Precedencia de aspectos.

Una vez que se arriba a un punto de enlace, todos los avisos del sistema son examinados para ver si alguno se aplica al punto de enlace. Aquellos que sí, son agrupados, ordenados según las reglas de precedencia, y ejecutados como sigue:

1) Primero, cualquier aviso "durante" es ejecutado, los de mayor precedencia primero. Dentro del código del aviso "durante", la llamada a **proceed()** invoca al próximo aviso "durante" que le sigue en precedencia. Cuando ya no quedan más avisos "durante" se pasa al punto 2).
2) Se ejecutan todos los avisos "antes" según su orden de precedencia (de mayor a menor).
3) Se ejecuta la computación del punto de enlace.
4) La ejecución de los avisos "después normal" y "después con excepción" depende de cómo resultó la ejecución en el punto 3) y de la terminación de avisos "después normal" y "después con excepción" ejecutados anteriormente.
    - Si la terminación es normal todos los avisos "después normal" se ejecutan, los de menor precedencia primero.
    - Si ocurre una excepción todos los avisos "después con excepción" que coinciden con la excepción se ejecutan, los de menor precedencia primero. (esto significa que los

avisos "después con excepción" pueden manejar excepciones señaladas por avisos "después normal" y "después con excepción" de menor precedencia.

5) Se ejecutan todos los "avisos después", los de menor precedencia primero.
6) Una vez que se ejecutan todos los avisos "después" el valor de retorno del punto 3) (si es que hay alguno) es devuelto a la llamada **proceed** correspondiente del punto 1) y ese aviso "durante" continúa su ejecución.
7) Cuando el aviso "durante" finaliza, el control pasa al próximo aviso "durante" con mayor precedencia.
8) Cuando termina el último aviso "durante", el control retorna al punto de enlace.

Acceso reflexivo

AspectJ provee un variable especial, thisJointPoint, que contiene información reflexiva sobre el punto de enlace actual para ser utilizada por un aviso. Esta variable sólo puede ser usada en el contexto de un aviso y está ligada a un objeto de clase JoinPoint que encapsula la información necesaria. La razón de su existencia es que algunos cortes pueden agrupar un gran número de puntos de enlaces y es deseable acceder a la información de cada uno.

```
before(Punto p): target(p) && call(*.*(..)){
    System.out.println("accediendo"+ thisJointPoint +"en"+p);
}
```

El ejemplo anterior muestra una manera "bruta" de efectuar trazas sobre todas las llamadas a métodos de la clase *Punto*.

La variable thisJointPoint puede usarse para acceder tanto a información estática como dinámica sobre los puntos de enlace. Para aquellos casos en los que sólo se desea la información estática, AspectJ provee otra variable especial thisJoinPointStaticPart, de la clase JoinPoint.StaticPart, la cual está ligada a un objeto que contiene únicamente la información estática. Dicho objeto es el mismo objeto que se obtendría con la siguiente expresión:

```
thisJointPoint.getStaticPart()
```

3.9.4 Introducciones y declaraciones

Las declaraciones de avisos cambian el comportamiento de sus clases asociadas pero no modifican su estructura estática. Para aquellos conceptos que operan sobre la estructura estática de la jerarquía de clases, AspectJ provee *formas de introducción*.

Cada *forma de introducción* será miembro del aspecto que lo define, pero introduce un nuevo miembro en otra clase.

En el siguiente ejemplo se introduce el atributo booleano *atencion* en la clase *Biblioteca* y lo inicializa con el valor true. Como está declarado **private** solamente puede ser accedido dentro del aspecto que incluye la declaración. Si fuera declarado como **public** podría ser accedido por cualquier código. Por omisión se asume como declarado **package-protected** y en este caso puede ser accedido por cualquier código dentro del paquete.

```
private boolean Biblioteca.atencion=true;
```

También se pueden introducir métodos, incluyendo constructores, en una o más clases utilizando un patrón de clase.

```
public Pila.new(Object o){
    arreglo = new Object[capacidadMaxima];
    ultimo=0;
    this.Apilar (o) ;
    }
```

El ejemplo anterior introduce un nuevo constructor en la clase *Pila*. No se puede introducir un constructor en una interface, y si el patrón de clase incluye una interface se producirá un error.

Las *formas de introducción* permiten también declarar que una o más clases objetivo implementarán una interface o heredarán de otra clase desde ese momento en adelante.

```
/* implementa una interface */
declare parents: Pila implements PilaAbstracta;
/* hereda de un clase */
declare parents: PilaOrdenada extends Pila;
```

Como efecto de estas declaraciones *PilaOrdenada* hereda de la clase *Pila* y *Pila* implementa la interface *PilaAbstracta*.

Una misma declaración puede introducir varios elementos en más de una clase.

```
public  unaFecha (Perro||Gato).fechaUltimaVacuna(){
            return  fechaVacuna;
}
```

Introduce dos métodos, uno en la clase *Perro* y otro en clase *Gato*; ambos métodos tienen el mismo nombre, no tienen parámetros, retornan un objeto de clase *unaFecha* y tienen el mismo cuerpo. En el caso que las clases *Perro* o *Gato* no tengan definido un atributo *fechaVacuna* resultará en un error.

Un aspecto puede introducir atributos y métodos tanto en interfaces como en clases.

Formalmente la declaración de *formas de introducción* se define con la siguiente BNF.

```
<Introducciones>::=<IntroduccionesdeAtributos> |
                   <IntroduccionesdeMetodos> |
                   <IntroduccionesdeMetodosAbstractos>|
                   <IntroduccionesdeConstructores>

<IntroduccionesdeAtributos>::=
[<Modificadores>] <Nombre_Clase> <PatrondeClase>.<Identificador>
 [= <expresión>] ;

<IntroduccionesdeMetodos>::=
[<Modificadores>] <Nombre_Clase>
<PatrondeClase>.<Identificador>(<Parametros_formales>)
  [ throws <Lista_Clases>] {<Cuerpo>}

<IntroduccionesdeMetodosAbstractos>::=
abstract [<Modificadores>] <Nombre_Clase>
<PatrondeClase>.<Identificador>(<Parametros_formales>)
  [ throws <Lista_Clases>] ;

<IntroduccionesdeConstructores>::=
[<Modificadores>]
<PatrondeClase>.new(<Parametros_formales>)
  [ throws <Lista_Clases>] {<Cuerpo>}
```

BNF 5: Definición de *formas de introducción*

La semántica de las *formas de introducción* es la siguiente:

- Introducción de atributos: el efecto de este tipo de introducción es que todas las clases que coincidan con el patrón de clase soportarán el nuevo atributo especificado en la introducción. Las interfaces que coincidan con el patrón de clase también soportarán el nuevo atributo.
- Introducción de métodos: el efecto de este tipo de introducción es que todas las clases que coincidan con el patrón de clase soportarán el nuevo método especificado en la introducción. Las interfaces también soportarán el nuevo método, aún si el método no es público o abstracto.
- Introducción de métodos abstractos: tiene el mismo efecto que la introducción de un método.
- Introducción de constructores: el efecto de este tipo de introducción es que todas las clases que coincidan con el patrón de clase soportarán el nuevo constructor especificado en la introducción. Como hemos mencionado anteriormente no se pueden introducir constructores en una interface, luego es un error si el patrón de clase incluye una interface.

La ocurrencia del identificador **this** en el cuerpo de la introducción de un constructor o método, o en el inicializador de una introducción de atributo, hace referencia a la clase objetivo y no al aspecto.

3.9.5 Aspectos

Los aspectos son declarados en forma similar a la declaración de clases. El aspecto es la unidad que encapsula puntos de enlace, cortes, avisos , introducciones y declaraciones, además de poder definir sus propios métodos y atributos. La diferencia con una clase es que un aspecto puede entrecruzar otras clases o aspectos, y que no son directamente instanciados con una expresión **new**, o procesos de clonado o serialización. Los aspectos pueden incluir la definición de un constructor pero dicha definición no debe contener argumentos y no debe señalar excepciones chequeadas.

Un aspecto puede ser definido en un paquete, o como un aspecto interno, esto es, puede ser miembro de una clase, una interface o un aspecto.

Como primer ejemplo de la definición de aspecto podemos implementar una traza sobre la clase *Pila* a través de un aspecto *TrazadoPila*. Por cada llamada a los métodos de la clase *Pila* se imprime el nombre del método.

```
aspect TrazadoPila{

    pointcut LlamadasaPila():call(* Pila.*(..));

    before():LlamadasaPila(){
     System.out.println("Entrando al método: "+  thisJointPoint);
    }
    after():LlamadasaPila(){
     System.out.println("Retornando del método: "+  thisJointPoint);
    }
}
```

Código 4: Aspecto de traza para la clase pila.

De no utilizar AspectJ , el código de *TrazadoPila* estaría diseminado por toda la clase *Pila* al comienzo y al final de cada uno de sus métodos. Si se quiere modificar el mensaje que se muestra o los métodos involucrados en la traza, implicaría revisar todo el código de la clase y realizar los cambios necesarios. Al tenerlo encapsulado como un aspecto dichas modificaciones son triviales y no corremos el riesgo de afectar la funcionalidad básica.

Reconsideremos el problema de agregar controles en los métodos *Apilar* y *Desapilar* de la clase *Pila*. Sin AspectJ dicho código queda diseminado en ambos métodos con los problemas ya mencionados. En cambio si tomamos al control como un aspecto *ControlPila* quedaría encapsulado en una única unidad.

```
aspect Control Pila{

    pointcut al_apilar(Pila p) : call(void Pila.Apilar(..)) && target (p);
    pointcut al_desapilar(Pila p) : call(Object Pila.Desapilar(..))
                                    && target(p);

    before(Pila p):al_apilar(p){
      if(p.Pila_Llena()){ throw new ExcepcionPilaLlena(); }
    }

    before(Pila p):al_desapilar(p){
      if(p.Pila_Vacia()){ throw new ExcepcionPilaVacia(); }
    }
}
```

Código 5: Aspecto de control para la clase pila.

Estos dos aspectos trabajan independientemente sobre la clase *Pila* abstrayendo dos conceptos diferentes. Está claramente separada la funcionalidad básica, en este caso el código de la clase *Pila* , de los aspectos de control y traza, y los aspectos entre sí. Sin aspectos el código de *Apilar*, por ejemplo, se vería "ensuciado" por el código necesario para la traza y el control.

Extensión de aspectos

Para manejar la abstracción y composición de los conceptos entrecruzados, los aspectos pueden extenderse en una forma similar a las clases. Sin embargo, la extensión de aspectos agrega nuevas reglas.

Un aspecto, abstracto o concreto, puede heredar de una clase e implementar un conjunto de interfaces. Heredar de una clase no le da al aspecto la habilidad de instanciarse con una expresión **new** .

Una clase no puede heredar o implementar un aspecto. Tal situación es considerada un error.

Los aspectos puede heredar de otros aspectos, heredando atributos, métodos y cortes. Una restricción importante es que los aspectos solo pueden heredar aspectos abstractos. Se considera un error que un aspecto concreto herede de otro aspecto concreto.

Como ejemplo podemos definir al aspecto *TrazaPila* en función de un aspecto abstracto *TrazaSimple*.

```
abstract aspect TrazaSimple{
      abstract pointcut PuntosTraza();

      before():PuntosTraza(){
       System.out.println("Entrando "+ thisJoinPoint);
      }

after():PuntosTraza(){
       System.out.println("Retornando "+ thisJoinPoint);
      }
}

aspect TrazaPila extends TrazaSimple{
      pointcut PuntosTraza():call(* Pila.*(..));
}
```

Código 6: Ejemplo de herencia en aspectos.

Privilegio de aspectos

El código escrito en un aspecto está sujeto a las mismas reglas de acceso del código Java para referenciar a miembros de clases o aspectos. Por ejemplo, un aspecto no puede referir a miembros con visibilidad package-protected sino está definido en el mismo paquete.

Existen algunas situaciones donde es necesario para los aspectos acceder a recursos privados o protegidos de otras clases. Para permitir esto se puede declarar al aspecto como privilegiado (**privileged**). Un aspecto privilegiado puede acceder a todos los atributos y métodos, incluso a los privados. Por ejemplo:

```
privileged aspect A {
 . . .
      before(Pila p):Corte1(p){
      . . .
      p.capacidadMaxima=25;
      . . .
}
. . .
}
```

El aspecto *A* al ser declarado como privilegiado puede acceder al atributo privado *capacidadMaxima* de la clase *Pila*.

Precedencia de aspectos

Existen situaciones en donde los avisos de un aspecto deben preceder a los avisos de otro aspecto. Por tal motivo, un aspecto puede declarar que "domina" a

otro aspecto estableciendo que los avisos definidos en él tienen prioridad sobre los avisos en el aspecto que domina.

aspect <Identificador> **dominates** < PatróndeClase> {. . .}

Como ejemplo, supongamos que tenemos definidos dos aspectos *Precondicion* y *Trazado*, y queremos que el aspecto *Precondicion* tenga mayor precedencia que el aspecto *Trazado* para que la traza se efectúe sólo si las precondiciones son satisfechas. Luego

aspect Precondicion dominates Trazado {. . .}

logra que el aspecto *Precondicion*, incluyendo sus avisos, tenga mayor precedencia que el aspecto *Trazado* y sus avisos serán ejecutados primero.

La declaración "dominates" forma parte de las reglas de precedencia de avisos, además de otros elementos como la herencia de aspectos.Las reglas que determinan la precedencia de avisos son las siguientes:

Si dos avisos están definidos en diferentes aspectos entonces:

- Si el aspecto A domina al aspecto B entonces los avisos en A tienen precedencia sobre los avisos en B.
- En otro caso, si el aspecto A hereda del aspecto B entonces todos los avisos en A tienen precedencia sobre los avisos en B.
- En otro caso, si no hay relación entre los aspectos entonces es indefinido quien tiene mayor precedencia.

Si dos avisos están definidos en el mismo aspecto:

- Si ambos son avisos "después" entonces el que esté definido último (según la estructura léxica del programa) tiene mayor precedencia.
- En otro caso, aquel que aparezca primero tiene mayor precedencia.

Cuando múltiples avisos se aplican a un mismo punto de enlace, el orden de resolución se basa en la precedencia de los avisos.

BNF completa

La siguiente BNF describe la declaración de aspectos en AspectJ agrupando todos los conceptos que vimos hasta ahora.

```
<Aspecto>::=[privileged] aspect <Identificador>
                [extends <Nombre_Clase>]
                [implements <Lista_Clases>]
                [dominates <Lista_Clases>]
 {

   {<Atributos>}
   {<Métodos>}
   {<Def_Cortes>}
   {<Introducciones>}
   {<Aviso>}

 }

<Def_Cortes>::= abstract [<Modificadores>]
                    pointcut <Identificador>(<Parametros_formales>); |
              [<Modificadores>]
              pointcut  <Identificador>(<Parametros_formales>):<Corte>;

<Introducciones>::=<IntroduccionesdeAtributos> |
                      <IntroduccionesdeMetodos> |
                      <IntroduccionesdeMetodosAbstractos>|
                      <IntroduccionesdeConstructores>

<IntroduccionesdeAtributos>::=
[<Modificadores>] <Nombre_Clase> <PatrondeClase>.<Identificador>
 [= <expresión>] ;

<IntroduccionesdeMetodos>::=
[<Modificadores>] <Nombre_Clase>
<PatrondeClase>.<Identificador>(<Parametros_formales>)
  [ throws <Lista_Clases>] {<Cuerpo>}

<IntroduccionesdeMetodosAbstractos>::=
abstract [<Modificadores>] <Nombre_Clase>
<PatrondeClase>.<Identificador>(<Parametros_formales>)
  [ throws <Lista_Clases>] ;

<IntroduccionesdeConstructores>::=
[<Modificadores>] <PatrondeClase>.new(<Parametros_formales>)
  [ throws <Lista_Clases>] {<Cuerpo>}
```

<Aviso>::=<Tipo_aviso> : <Corte> {<Cuerpo>}
<Tipo_aviso>::= <Aviso_antes> | <Aviso_despues> | <Aviso_durante>

<Aviso_antes>::= **before (**<Parametros_formales>**)**
<Aviso_despues> ::= **after (**<Parametros_formales>**)** <Forma_terminacion>
<Forma_terminacion>::= **returning** [**(**<Parametro_formal>**)**] |
throwing [**(**<Parametro_formal>**)**] | λ

<Aviso_durante>::= <Nombre_Clase> **around (**<Parametros_formales>**)**
[**throws** <Lista_Clases>]

BNF 6: Definición de aspectos

3.9.6 Evaluación

En esta descripción de AspectJ no han sido tratados detalles de implementación para permitir una mejor comprensión y aprendizaje del lenguaje. Hemos priorizado la simpleza y forma general del lenguaje por sobre detalles de implementación que oscurecen el entendimiento. Además, no es necesario conocer a fondo estos detalles para iniciarse en la programación de AspectJ. En sí, describimos de AspectJ aquellos conceptos necesarios para poder programar con aspectos por primera vez y/o entender la semántica de un programa escrito en AspectJ. En [35], la página oficial de AspectJ, se muestra la documentación del lenguaje donde se describe informalmente los detalles de implementación.

La programación en AspectJ es directa si el programador conoce el lenguaje Java. Los aspectos son muy similares a las clases, los avisos a los métodos. Luego la declaración de aspectos es sencilla, el programador con poco esfuerzo y sin mayores dificultades agrega aspectos a su implementación con las ventajas que esto trae. Al ser una extensión compatible de Java los costos de aprendizaje son mínimos.

AspectJ soporta la facilidad "plug-in plug-out" de aspectos. Los aspectos pueden agregarse o removerse de la funcionalidad básica con sólo incluirlos o no en la compilación. Así el programador puede intentar introducir aspectos en su aplicación y de no satisfacerle los puede remover, manteniendo la aplicación en su estado original.

Uno de los puntos fuerte de AspectJ es la composición de cortes a través de operadores booleanos, la cual es a la vez simple y poderosa. Por lo tanto la expresividad del lenguaje es destacable.

4. Un ejemplo: implementación del protocolo TFTP

Como ejemplo de una implementación, vamos a considerar la implementación del protocolo TFTP (Trivial File Transport Protocol), con y sin aspectos. La versión sin aspectos es la versión TFTP 0.8 de Mark Benvenuto, realizada en Java, de distribución libre y que puede obtenerse de su página [37]. Nosotros implementamos la versión con aspectos en AspectJ; la funcionalidad básica está basada en la versión de Benvenuto pero con las modificaciones que surgen de introducir aspectos en el desarrollo. Para obtener las conclusiones, se compararán ambas implementaciones, y así poder evaluar la verdadera dimensión de las ventajas de la Programación Orientada a Aspectos, y de su principal herramienta, AspectJ.

4.1 El protocolo TFTP

TFTP es un protocolo muy simple para transferir archivos entre procesos. Entre sus principales características principales figuran:

- Sobrecarga mínima.
- Poca seguridad.
- Implementado sobre User Datagram Protocol (UDP).
- Pequeño y fácil de implementar.
- Sólo permite operaciones de lectura y escritura de archivos desde y hacia un servidor remoto.
- Datos de 8 bits (byte).

Las principales áreas donde TFTP es utilizado son:

- Bootless diskless workstation, terminal servers o routers: para obtener la IP del servidor TFTP desde un servidor de RARP (Reverse Address Resolution Protocol), y para la lectura de boot files desde el servidor TFTP.
- Manejo de archivos de configuración: almacenar configuraciones, o restablecer configuraciones desde un servidor TFTP.

Es un protocolo alternativo a FTP, con menos sobrecarga y más simple, pero con seguridad mínima. Otras diferencias con FTP son:

- No provee identificación de usuario.
- Utiliza un único port para la transferencia.
- No provee comandos para operaciones sobre archivos o directorios.
- No tiene incluidos conceptos de seguridad ya que utiliza la seguridad brindada por el sistema operativo.

Dentro del protocolo, existen cinco tipos de mensajes que se mencionan en la tabla 3.

CÓDIGO	DESCRIPCIÓN	FORMATO DEL PAQUETE
01 RRQ	Requerimiento de lectura	01 | NombreArch | 0 | Modo | 0
02 WRQ	Requerimiento de escritura	02 | NombreArch | 0 | Modo | 0
03 DATA	Datos	03 | Nro Bloque | DATA
04 ACK	Reconocimiento	04 | Nro Bloque
05 ERROR	Errores	05 | Código Error | Mensaje | 0

Tabla 3: Paquetes TFTP

Existen ocho tipos de errores:

1 No definido.
2 Archivo inexistente.
3 Violación de acceso.
4 Disco lleno.
5 Operación TFTP ilegal.
6 Puerto desconocido.
7 Archivo ya existente.
8 Usuario desconocido.

4.1.1 Comportamiento general del protocolo TFTP:

El comportamiento del protocolo para los dos tipos de operaciones que soporta son:

1) Lectura:
   i) El cliente envía un RRQ.
   ii) El servidor responde con el bloque de datos número 1.
   iii) El cliente reconoce el bloque número 1.
   iv) El servidor envía el bloque de datos número 2.
   v) ...

2) Escritura:
   i) El cliente envía WRQ.
   ii) El servidor responde con un reconocimiento número 0.
   iii) El cliente envía el bloque de datos número 1.

iv) El servidor reconoce el bloque de datos número 1.
v) ...

Se debe tener en cuenta que todos los bloques son de 512 bytes, una cantidad menor indica la finalización de la transferencia.

En unas pocas líneas, podemos resumir el protocolo TFTP en cuatro ítems:

- Cada paquete de dato debe ser reconocido mediante un paquete ACK.
- Cada transferencia comienza con un pedido de lectura o escritura.
- Ambas máquinas envían y reciben paquetes:
  - Una envía paquetes de datos y recibe paquetes de reconocimiento.
  - La otra, envía paquetes de reconocimiento, y recibe paquetes de datos.

Para mayor información sobre el protocolo TFTP, consultar las RFC[2] (Request For Comment) :

- 783 : The TFTP Protocol (Revision 2), junio 1981.
- 1350: The TFTP Protocol (Revision 2), julio 1992.
- 1785: TFTP Option Negociation Analysis, marzo 1995.
- 2347: TFTP Option Extension, marzo 1998.
- 2348: TFTP Blocksize Option, marzo 1998.
- 2349: TFTP Timeout Interval and Transfer Size Options, marzo 1998.

4.2 Implementación en Java

Existen dos puntos de vista principales para la implementación: el punto de vista del cliente, y el punto de vista del servidor. Nosotros, sólo analizaremos el punto de vista del servidor, por cuestiones de extensión. Estudiando sólo esta rama de análisis, alcanza para mostrar, claramente, los beneficios de este nuevo paradigma y de la utilización de la herramienta AspectJ.

Por otra parte, una vez analizado el punto de vista del servidor, es sencillo comprender el punto de vista del cliente. Por razones similares, dentro del punto de vista del servidor, sólo analizaremos las acciones correspondientes a la escritura de archivos. Esto es, un pedido de lectura por parte del cliente. Las acciones para una conexión de lectura debido a un pedido de escritura por parte del cliente son análogas a una conexión de escritura, y no agrega mucho más al ejemplo. La idea que seguimos fue mantener el modelo lo más sencillo posible y simultáneamente, poder observar claramente las ventajas de la POA.

La figura 11 muestra el camino de análisis que hemos elegido. Dicho camino está marcado con color violeta dentro del árbol que modela la implementación del protocolo TFTP.

[2] Los RFC son reportes técnicos que describen todos los estándares sobre Internet, son definidos por la Internet Engineering Task Force (IETF) y pueden obtenerse en http://www.ietf.org/.

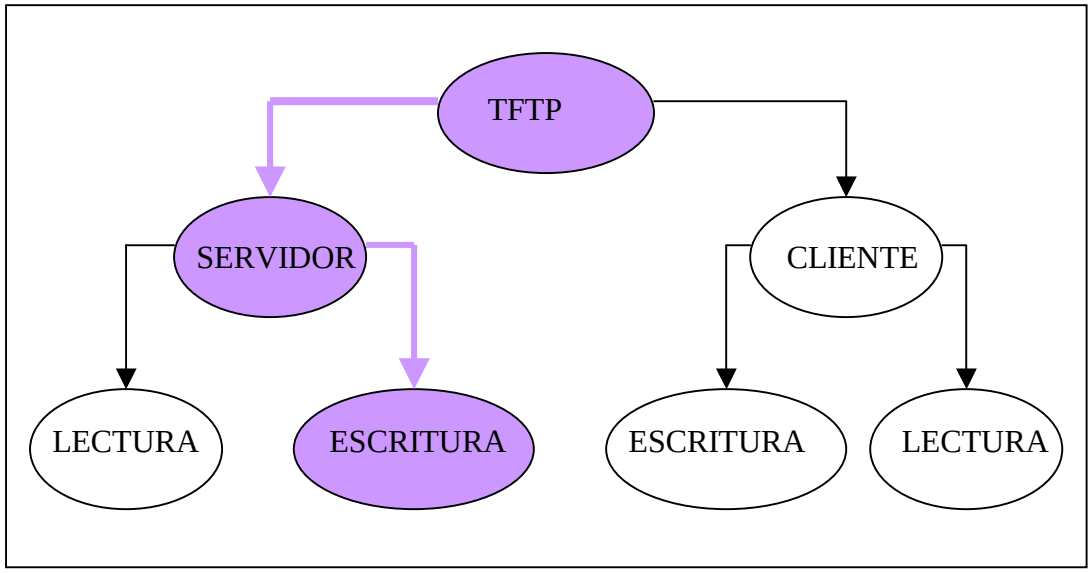


Figura 11: Rama de análisis

4.2.1 Implementación del servidor TFTP

En esta sección explicaremos la implementación del servidor TFTP por Mark Benvenuto. Recordar que según nuestra rama de análisis, sólo nos son de interés las clases: *Tftpd*, *TftpServerConnection* y *TftpWriteConnection*.

La clase principal es *Tftpd*, la cual se encarga de dejar al servidor "listo" para procesar los pedidos por parte de los clientes. Cuando arriba un pedido el servidor crea una instancia de la clase *TftpServerConnection*, la cual realiza los primeros pasos de la conexión. Por último, existen dos clases, las cuales también las utiliza el cliente, *TftpWriteConnection*, y *TftpReadConnection*, que se encargan de procesar toda la conexión, ya sea un pedido de escritura o un pedido de lectura.

El comportamiento del servidor se puede describir con los siguientes pasos:

Como primera acción, se crea una instancia de la clase *Tftpd*, llamada server. Luego, se invoca al método *processConnection*.

↓

Lo que hace el servidor, a través de este método, es crear un socket[3], con número de puerto 69, que es el puerto TFTP por defecto. Este es el socket principal al cual arriban los pedidos de los clientes. El servidor permanece en un loop infinito, esperando por estos pedidos. Cada pedido se ejecutará como un hilo de ejecución.

↓

[3] Un socket es la interface entre una aplicación y un protocolo.

Al arribar un pedido, se obtiene información del mismo, como por ejemplo determinar si es un pedido de lectura o escritura, el número de puerto remoto, el nombre del archivo a utilizar, etc. Con esta información, se crea una instancia de la clase *TftpServerConnection*, y se ejecuta esa instancia como un hilo independiente. Esta instancia recién creada manejará el resto de la conexión de ahora en más, liberando al servidor de manera que continue atendiendo los nuevos pedidos.

La clase *TftpServerConnection* es quien prepara todo para una conexión TFTP en particular. Como primera medida, creará un nuevo socket. De ahí en adelante, la comunicación entre el cliente y el servidor se realizará a través de este socket. Con la información de la conexión, y este nuevo socket, se crea una instancia de la clase *TftpReadConnection*, o de la clase *TftpWriteConnection*, según el pedido del cliente. Estas clases son las encargadas de procesar toda la transferencia, esto es, enviar tanto paquetes de datos como de reconocimiento, efectuando los controles correspondientes.

4.2.2 Código Java

El código de las clases *Tftpd*, *TftpServerConnection*, y *TftpWriteConnection* se detalla en el apéndice A.

4.3 Implementación en AspectJ

Si bien la lógica del protocolo es simple, el código resultante es confuso y complejo. Esto se debe a ciertos aspectos, que nada tienen que ver con el protocolo en sí, sino a cuestiones extras, o agregados.

El más importante, el que "ensucia" más el código, es el aspecto de logging: encargado de llevar las trazas, informar de las excepciones, manejar las estadísticas de las conexiones, entre otras tareas. Todo el código necesario para el logging resulta desparramado por todas las clases, esto es, entrecruza las clases, obstruyendo el entendimiento de la lógica del programa con código que no tiene relación directa con el protocolo TFTP. Este código sucio, que muchas veces se trata de código duplicado, incluye sentencias, mensajes, como también atributos relativos al logging dentro de las clases que implementan el protocolo. Como ejemplo, la variable entera *logging*, o la variable de clase PrintStream *logStream*, ambas son atributos de la clase *Tftpd* y sin embargo no están relacionadas con el protocolo sino con el aspecto de logging. Las graves consecuencias de este tipo de código ya han sido tratadas ampliamente en este trabajo: dificultad de entendimiento y de lectura, menor robustez, dificultades varias a la hora de enfrentar cambios o modificaciones, disminución de la calidad, etc.

4.3.1 El aspecto de logging

Para identificar el aspecto de logging se extraerán las acciones propias del logging del código que implementa la funcionalidad básica. El conjunto de esas acciones extraídas constituirá el aspecto.

Para comenzar, en las clases a analizar, *Tftpd*, *TftpServerConnection* y *TftpWriteConnection*, existen atributos extraños a estas clases, ya que no tienen relación con el protocolo, sino que son relativos al logging. En la clase *Tftpd* tenemos la siguiente declaración:

```
String rootDirectory = "tftpdroot";
int logging = 1;
int logStdout = 1;
int logUsingSyslogd = 0; //to be implemented in next version
int socketTimeout = 3000;
int debugMode = 0;
String logFile = "tftpd.log";
//Logging Stuff
private File lFile = null;
private FileOutputStream fout = null;
private TftpLogStream logTftpStream = null;
private PrintStream logStream = null;
InetAddress iahost = null;
```

Código 7: Atributos extraños en la clase Tftpd.

Los atributos marcados con rojo son los atributos extraños a la clase, utilizados para la tarea de logging. De los once atributos que se introducen, nueve resultan extraños. En las otras dos clases, están presentes los atributos *debug* y *logWriter*, los cuales también "ensucian" la declaración de la clases. Además, en la creación de estas clases, información relativa al logging se pasa como parámetro para la inicialización de estos dos atributos:

```
TftpServerConnection(InetAddress ia, int rport, String localFilename,
                     int rtype, PrintStream logStream, int dstate){
    debug = dstate;
    cia = ia;
    remoteport = rport;
    localfile = localFilename;
    TransferType = rtype;
    logWriter = logStream;
...}
```

Código 8: Constructor de la clase *TftpServerConnection*.

```
TftpWriteConnection(DatagramSocket ds, InetAddress ia,
                    FileInputStream fstream, int dbstate){
    sock = ds;
    debug = dbstate;
    ria = ia;
    istream = fstream;
    logWriter = System.out ;
}
```

Código 9: Constructor de la clase *TftpWriteConnection*.

Las sentencias marcadas en rojo en los constructores de las clases *TftpWriteConnection* y *TftpServerConnection*, de igual forma corresponden a acciones de logging, y formarán parte del aspecto.

El efecto de todos los atributos de logging se extienden en las tres clases, mezclándose con la implementación del protocolo. Al tener a logging como aspecto, todo el código en rojo desaparece, formando parte del aspecto, resultando en un código más limpio y legible.

También existen métodos con el mismo problema, esto es, son acciones de logging insertadas en la funcionalidad básica. En la clase *Tftpd*, se encuentran los métodos *initializeLogging()*, *getLogWriter()*, y en *TftpWriteConnection*, los métodos *setLogWriter(PrintStream pw)*, *printError(byte [] data)*. De la misma forma que los atributos anteriores, estos métodos serán ahora métodos del aspecto.

Analicemos ahora las acciones que se realizan cuando se recibe un paquete, dentro del método *processConnection*, de la clase *TftpWriteConnection*. Nuevamente con rojo marcamos las acciones del logging.

```
if(debug > 0)
   logWriter.println("Receiving Packet");
try{
 sock.receive(ACKp);
    }
catch( InterruptedIOException iioe) {
  logWriter.println("Error Receiving Packet: " + iioe);
  if(RECVtries > maxDATAtries)  {
    logWriter.println("Waited to many times for ACK packet");
    return;
  }
  RECVtries++;
  logWriter.println("Continuing to wait for Packet");
  continue;
 }
catch(IOException e) {
 logWriter.println("Error Receiving Packet: " + e);
 return;
 }
 RECVtries = 0;
 if(remoteport == 0) {
  remoteport = ACKp.getPort();
```

```
 p.setPort(remoteport);
 p.setAddress(ria);
 }
datalength = ACKp.getLength();
if(debug > 0){
 logWriter.println("Received Packet: " + ACKp);
 logWriter.println("Type: " + ACKdata[0] + ACKdata[1]);
 if(ACKdata[1] == 3) {
  logWriter.println("Received DATA Packet( Len: " + datalength + " )");
  logWriter.println(ACKp.getData());
 }
 if(ACKdata[1] == 4) {
  logWriter.println("Received ACK Packet( Len: " + datalength + " )");
  logWriter.println("ACK: " + (ACKdata[2] << 8) + ACKdata[3]);
 }
}
if(isError(ACKdata)) {
 logWriter.println("Received ERROR Packet");
 printError(ACKdata);
 isRead = 0;
 break;
}
```

Código 10: Ejemplo de código mezclado.

Se nota claramente, una vez más, cómo estas acciones que no forman parte del protocolo, oscurecen la legibilidad del código porque es un único código o dimensión que intenta capturar al mismo tiempo el comportamiento del protocolo junto con el logging. AspectJ nos permite aplicar el principio de "dividir y conquistar" con mayor intensidad, abstrayendo por separado el aspecto de logging, logrando más de una dimensión de desarrollo.

Para implementar la situación anterior en AspectJ identificamos un momento en particular, al recibir un paquete, y consideramos las acciones de logging que ocurren antes y después de ese momento. En términos de AspectJ, un momento es un corte, y las acciones son avisos. El siguiente corte captura todas las veces que se recibe un paquete:

```
pointcut receivePacket(DatagramPacket packet):
call(DatagramSocket.receive(DatagramPacket))&& args(packet);
```

El código de los avisos se corresponde con las sentencias marcadas con rojo en el código anterior:

```
before(DatagramPacket packet):receivePacket(packet)&& anyTftp(){
    if(debugMode > 0)
        logStream.println("Receiving Packet");
 }

after(DatagramPacket packet) throwing(InterruptedIOException iioe ):
                        receivePacket(packet) && classWriteConnection(){
    logStream.println("Error Receiving Packet: " + iioe);
}


after(DatagramPacket packet) throwing(IOException ioe):receivePacket(Packet)
                                    && classWriteConnection(){
    logStream.println("Error Receiving Packet: " + ioe);
```

```
}

after(DatagramPacket packet) returning :receivePacket(packet) &&
                                        classWriteConnection(){
    byte ACKdata[] = packet.getData();
    int datalength = packet.getLength();
    …
    if(debugMode > 0) {
        logStream.println("Received Packet: " + packet);
        logStream.println("Type: " + ACKdata[0] + ACKdata[1]);

        if(ACKdata[1] == 3) {
            logStream.println("Received DATA Packet( Len: " + datalength + " )");
            logStream.println(ACKdata);
        }
        if(ACKdata[1] == 4) {
            logStream.println("Received ACK Packet( Len: " + datalength + " )");
            logStream.println("ACK: " + (ACKdata[2] << 8) + ACKdata[3]);
        ...
        }
    }
    if(ACKdata[1] == 5)  {
        logStream.println("Received ERROR Packet");
        printError(ACKdata);
    }
}
```

Es realmente notable el cambio que obtenemos. Fundamentalmente, se pasa de un código mezclado y confuso, donde es muy difícil efectuar un cambio, a tener por un lado una funcionalidad básica limpia y sin impurezas. Por otro lado, se tiene un aspecto de logging donde está claramente identificado un momento particular de ejecución junto con las acciones que ocurren antes y después de él, pudiendo distinguir además entre comportamiento normal y con excepción.

Para visualizar los cambios que se obtienen comparemos los códigos que resultan de aplicar una programación con y sin aspectos, ubicándolos en dos columnas. La columna de la izquierda corresponde a la manera actual de programar, sin aspectos. Esto es una única dimensión en donde se concentra el protocolo y las acciones de logging. La columna de la derecha corresponde al paradigma de aspectos. Esto es un código que implementa únicamente la funcionalidad básica, el protocolo, encargándose el aspecto de todas las acciones de logging. .

Es evidente contrastando las columnas que el código de la derecha es más simple y entendible por el hecho que se refiere solamente a la implementación del protocolo. Observemos que estos importantes beneficios resultan de aplicar el paradigma sólo a una porción del sistema. Obviamente las ganancias son notablemente mayores al aplicar el paradigma a todo el sistema.

SIN ASPECTOS

```
if(debug > 0)
   logWriter.println("Receiving Packet");
try{
 sock.receive(ACKp);
    }
catch( InterruptedIOException iioe) {
  logWriter.println("Error Receiving
Packet: " + iioe);
  if(RECVtries > maxDATAtries)  {
    logWriter.println("Waited to many
times for ACK packet");
    return;
  }
 RECVtries++;
 logWriter.println("Continuing to wait for
Packet");
 continue;
 }
catch(IOException e) {
 logWriter.println("Error Receiving
Packet: " + e);
 return;
 }
 RECVtries = 0;
 if(remoteport == 0) {
    remoteport = ACKp.getPort();
    p.setPort(remoteport);
    p.setAddress(ria);
 }
datalength = ACKp.getLength();
if(debug > 0){
 logWriter.println("Received Packet: " +
                        ACKp);
 logWriter.println("Type: " +
ACKdata[0]
                     + ACKdata[1]);
  if(ACKdata[1] == 3) {
   logWriter.println("Received DATA
         Packet( Len: " + datalength + "
)");
  logWriter.println(ACKp.getData());
  }
  if(ACKdata[1] == 4) {
  logWriter.println("Received ACK
Packet(
                  Len: " + datalength + " )");
  logWriter.println("ACK: " +
(ACKdata[2]
                        << 8) + ACKdata[3]);
  }
}
if(isError(ACKdata)) {
 logWriter.println("Received ERROR
                        Packet");
 printError(ACKdata);
 isRead = 0;
 break;
}
```

CON ASPECTOS

```
try{
    sock.receive(ACKp);
 }
catch( InterruptedIOException iioe) {
 if(RECVtries > maxDATAtries)
      return;
 RECVtries++;
 continue;
}
catch(IOException e) {
 return;
}
RECVtries = 0;
if(remoteport == 0) {
    remoteport = ACKp.getPort();
    p.setPort(remoteport);
    p.setAddress(ria);
}
datalength = ACKp.getLength();
if(isError(ACKdata)){
     isRead = 0;
     break;
}
```

Para implementar el resto del sistema en AspectJ seguimos la misma estrategia que aplicamos a la porción de código que analizamos anteriormente. Dicha estrategia, dado un aspecto consiste en cuatro pasos:

1- Identificar un aspecto en el sistema. Por ejemplo, el aspecto de logging en la implementación del protocolo TFTP.
2- Para el aspecto identificado considerar los siguientes pasos:

2.1 Identificar un momento en particular. Por ejemplo, al recibir o enviar un paquete, al leer o escribir de un archivo, al crear un socket, al obtener información de un paquete, etc.
2.2 Capturar ese momento como un corte de AspectJ. Por ejemplo, el siguiente corte captura el momento de enviar un paquete.

```
pointcut sendPacket(DatagramPacket packet):
    call(DatagramSocket.send(DatagramPacket)) && args(packet);
```

2.3 Identificar las acciones, relativas al aspecto en cuestión, que ocurren antes y después de ese momento. Por ejemplo, avisar que se está enviando un paquete antes de comenzar el envío, señalar la finalización del envío, señalar que ha ocurrido una excepción, etc.
2.4 Codificar esas acciones como avisos "antes" y "después". Por ejemplo, el siguiente aviso "antes" avisa que se está por enviar un paquete.

```
before(DatagramPacket packet):sendPacket(packet){
    logStream.println("Sending  Packet ");
}
```

2.5 Regresar al punto 2.1 mientras existan momentos de interés.

3- Regresar al punto 1 mientras existan aspectos en el sistema.

4.3.2 Código en AspectJ

El código de la funcionalidad básica y del aspecto de logging se encuentran en el apéndice B. En el aspecto de logging se utilizaron distintos colores para distinguir las principales entidades de AspectJ. Con color verde se marcaron los cortes, con rojo los avisos y con violeta los atributos y métodos propios del aspecto.

4.4 Conclusiones

Al utilizar aspectos la primer ventaja que salta a la vista, es que el código que implementa la funcionalidad básica es un código mucho más limpio y entendible. También, al estar todas las acciones de logging encapsuladas en el aspecto y claramente identificadas, se facilita enormemente la realización de cambios o modificaciones, como también acciones para remover o agregar comportamiento. En cuanto al desarrollo, el mismo se vuelve mucho más intuitivo, más natural, dada la naturaleza declarativa de los cortes y los avisos en AspectJ. Como ejemplo sencillo, tomemos el caso de todas las clases de la implementación que al enviar un paquete

imprimen el mensaje en el logging: “Enviando paquete...”. El programador, cada vez que se envía un paquete, en una o más clases, debe codificar él mismo las sentencias necesarias para imprimir el mensaje. Este es un proceso que lleva a errores. Tan sólo basta que el programador se olvide de un caso para producir un error que será difícil de identificar. Más grave aún, imaginemos ahora que se desea cambiar ese mensaje: el programador debe recordar y recorrer todas las clases, buscando los lugares donde efectuar los cambios: claramente, este es otro proceso que conduce a errores. En cambio, considerar las siguientes declaraciones dentro del aspecto logging:

```
pointcut sendPacket(DatagramPacket packet):
     call(DatagramSocket.send(DatagramPacket)) && args(packet);

before(DatagramPacket packet):sendPacket(packet) {
                        logStream.println("Sending  Packet ");
}
```

En apenas dos líneas se consigue el comportamiento deseado, de una manera extremadamente natural y sencilla, ubicado en un lugar específico en vez de resultar repetido y disperso por todo el sistema. Esto constituye un pequeño ejemplo que muestra claramente las bondades de AspectJ, y de la POA.

4.4.1 Evaluación del caso de estudio

Para cuantificar informalmente los beneficios de la POA y AspectJ en nuestro ejemplo tengamos en cuenta el tamaño físico de los archivos de las clases y del aspecto, y un factor de desarrollo que propone Rich Price en [4]. Dicho factor establece que para un sistema pequeño el tiempo de desarrollo promedio en Java es de 10 horas/programador mientras que en AspectJ es de 2 horas/programador. Entonces a partir de los datos podemos construir la siguiente métrica:

$$\text{Beneficio POA} = \frac{TamañoSinAspectos * FactorDeDesarrolloJava}{TamañoConAspectos * FactorDeDesarrolloAspectJ}$$

donde *TamañoConAspectos* = *TamañoFuncionalidadBásica*+*TamañoAspecto* y un resultado mayor que uno de la métrica indica un impacto positivo de la utilización de aspectos.

Aplicado a nuestro ejemplo tenemos que el tamaño físico de las clases es el siguiente:

	Con aspectos	Sin aspectos
Tftpd	6,14 Kb	10,1 Kb
TftpServerConnection	5,09 Kb	6,35 Kb
TftpWriteConnection	8,49 Kb	12,1 Kb
Aspect logging	8,30 Kb	-

Luego,

TamañoSinAspectos = (10,1+6,35+12,1) Kb = 28,55 Kb
TamañoConAspectos = *TamañoFuncionalidadBásica*+*TamañoAspecto*
= (6,14+5,09+8,49) Kb+ 8,30 Kb
= 19,72 Kb + 8,30 Kb = 28,02 Kb

entonces reemplazando los valores en la fórmula de la métrica obtenemos que

$$\text{Beneficio POA} = \frac{28{,}55Kb * 10hp}{28{,}02Kb * 2hp} = \frac{285{,}5}{56{,}04} = 5{,}09$$

También podemos cuantificar la limpieza que se obtiene en el código al utilizar aspectos a través de la siguiente métrica:

$$\text{Limpieza} = \frac{(TamañoSinAspectos - TamañoFuncionalidadBásica) * 100}{TamañoSinAspectos}$$

Aplicado a nuestro ejemplo tenemos que

$$\text{Limpieza} = \frac{(28{,}55 - 19{,}72)Kb * 100}{28{,}55Kb} = 30{,}92$$

Por último comparamos las funcionalidades básicas de las dos alternativas en función del tamaño expresado en líneas de código(ldc).

	Con aspectos	Sin aspectos
Tftpd	199 ldc	318 ldc
TftpServerConnection	176 ldc	212 ldc
TftpWriteConnection	339 ldc	471 ldc
Tamaño total en ldc	*714 ldc*	*1001 ldc*

4.4.2 Conclusiones de la evaluación

Por los resultados que obtuvimos de las métricas podemos concluir que la utilización de aspectos fue ampliamente exitosa. Primero, la métrica *Beneficio POA* arrojó el valor 5,09 que al ser mayor que 1 se traduce en un impacto positivo. Segundo, el valor 30,92 calculado por la métrica *Limpieza* indica que el código original se ha limpiado casi en un 31%. Por último, la cantidad de líneas de código de la funcionalidad básica se redujo en 300 lineas de código que representa el 30% del total, confirmando el valor obtenido en la métrica de *Limpieza*.

Cabe preguntarse cuál es el costo de trabajar con AspectJ, y la respuesta a esta inquietud es lo que hace a la POA, y a AspectJ en particular tan atractiva: el costo es mínimo. Si el programador ya sabe programar en Java, prácticamente ya está programando en AspectJ: es una extensión de Java.

Aprender a trabajar con aspectos es sumamente sencillo, dada la naturaleza declarativa de los mismos. Además, la característica que los aspectos sean fácilmente "enchufados" y "desenchufados" (plug-in, plug-out) a la funcionalidad básica, permiten al programador ir experimentando con aspectos, sin que esto afecte a la funcionalidad básica, resultando en un aprendizaje gradual y efectivo. Todo lo que hemos experimentado puedo resumirse en una sola afirmación, que establece por sí misma la importancia de este nuevo paradigma "*Con muy poco, se obtienen ganancias importantes en la calidad, no solo del producto final, sino también en el proceso que desarrolla el producto*".

5. Conclusiones finales

La POA es un paradigma que recién está naciendo. Todavía 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, en el diseño[4], en el testing y en la documentación; investigar sobre la formalización de los aspectos[5]; observar cómo es la integración de los aspectos con las aproximaciones, métodos, herramientas y procesos de desarrollo existentes; desarrollar herramientas orientadas a aspectos que respeten los principios de diseño; entre otras áreas de investigación.

Estamos en el comienzo de un nuevo paradigma, descubriendo su potencialidad, sus problemas, y las posibles soluciones. A medida que se logre avanzar en la investigación se irán obteniendo conceptos cada vez más sólidos. De esta manera, la POA irá creciendo, madurando como paradigma, y enfrentándose a las debilidades que surjan.

Los resultados encontrados en la bibliografia y los que hemos obtenido en este trabajo son más que prometedores y nos hacen pensar que la POA es una de las ramas con mayor futuro dentro de la ingeniería de software.

En el resto de esta última sección se compara el paradigma de aspectos con su "predecesor", el paradigma de objetos y se analiza la relación entre ellos. También se menciona algunas áreas de investigación relacionadas con el paradigma teniendo en cuenta que sus avances influirán en el futuro de la POA. Por último, se enuncian lo que según nuestro análisis son las mayores ventajas y desventajas de la POA.

5.1 Breve comparación entre POA y POO

A primera vista daría la impresión que la POA y la POO son en realidad el mismo paradigma. Sin embargo, esta primera impresión es errónea. Un análisis más profundo revela las diferencias entre los paradigmas:

A través de la POO los sistemas toman la forma de un conjunto de objetos que colaboran entre sí. En realidad, la mayor fortaleza de la POO se demuestra cuando hay que modelar conceptos comunes. Sin embargo, falla al modelar los conceptos que se entrecruzan. POA logra este objetivo al tratar los conceptos entrecruzados como elementos de primera clase, extrayéndolos horizontalmente de la estructura vertical del árbol de herencia.

Tanto la POA como la POO crean implementaciones modularizadas y con mínimo acoplamiento. La diferencia radica en qué mientras la POA se enfoca en los conceptos que se entrecruzan, la POO se enfoca en los conceptos comunes y los ubica en un árbol de herencia. Gráficamente esto se ve reflejado en la figura 12.

[4] Ver referencias [5,43,44,45,46].
[5] Ver referencias [47,48,49].

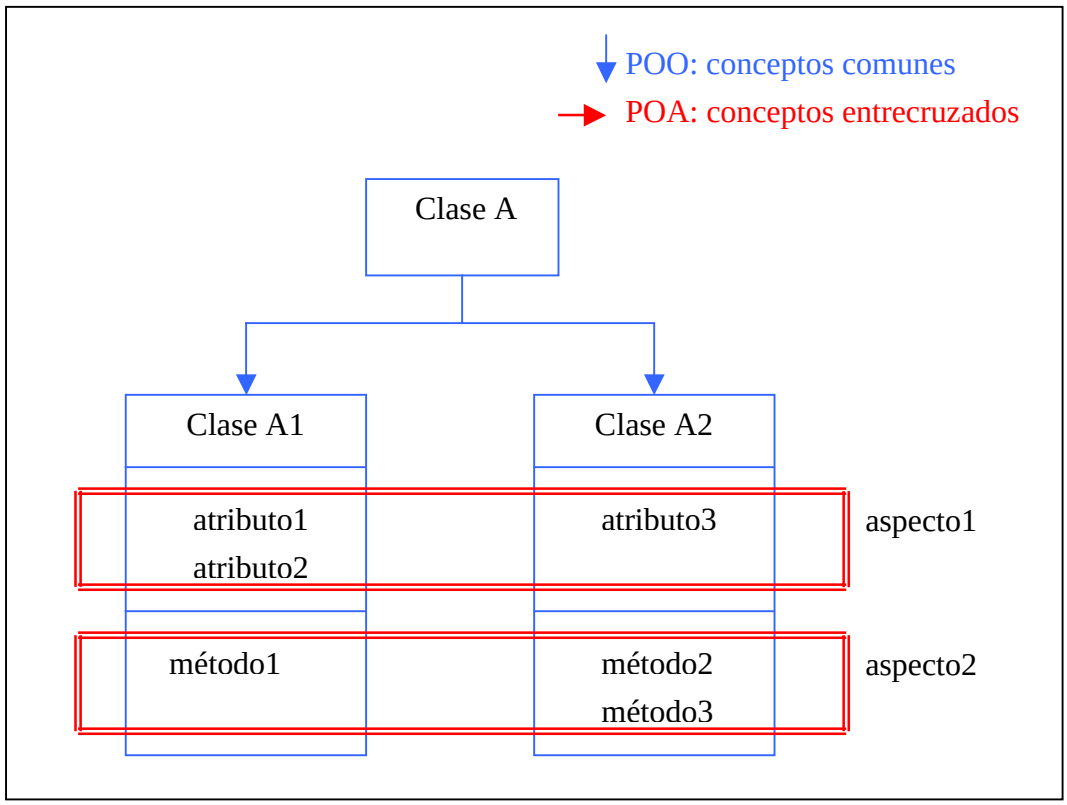


Figura 12: Comparación de POA con POO.

En POA la implementación de los conceptos son independientes. Esta independencia la distingue de las técnicas inherentes a la POO. En POA, el flujo de composición va desde de los conceptos que se entrecruzan al concepto principal; mientras que en la POO el flujo va en dirección opuesta.

Otra de las impresiones erróneas que puede tener quien lee por primera vez sobre aspectos es considerar que la POA está limitada a utilizar como base a la POO. Entonces es primordial aclarar que una implementación POA puede utilizar como metodología base cualquier paradigma de programación, manteniendo intactos los beneficios de ese paradigma de programación. Se elige POO como el sistema base para obtener los beneficios de una mejor implementación de los conceptos comunes. Bajo esta implementación los conceptos individuales pueden emplear técnicas orientadas a objetos. Esto es análogo a la forma en que actúan los lenguajes procedurales como lenguaje base a muchos lenguajes orientados a objetos [18].

5.2 Trabajos relacionados

El tópico de *separación de conceptos* ha sido un tema de investigación fuerte en los últimos años, con distintos nombres en cada caso. En la Universidad de Northeastern ha recibido el nombre de Programación Adaptativa por Karl Lieberherr[40]. En la Universidad de Twente, Holanda, recibe el nombre de Filtros de Composición por Mehmet Aksit[41]. En IBM Research, Yorktown Heights(N.Y.), se la conoce como Programación Subjetiva, y uno de sus proyectos es el sistema HyperJ[28] para programación en Java. También se puede citar el trabajo de Mira Mezini[42], en la Universidad de Siegen, en Alemania, en el

proyecto "Optimización en la Modularidad y Reusabilidad del Software Orientado a Aspectos".

Otra tema de investigación relacionado es la Meta Programación Lógica Orientada a Aspectos(Logic POA), creada por Johan Brichau y Kris De Volder. Esta aproximación representa los programas como hechos lógicos y describe a los aspectos como programas meta-lógicos que entrecruzan la modularidad del nivel base. Esto permite definir lenguajes orientados a aspectos nuevos y más específicos en términos de lenguajes orientados a aspectos ya existentes. Actualmente están investigando cómo un modelo de aspectos encapsulado como programas meta-lógicos puede ser usado para soportar combinaciones de lenguajes de aspectos.

Existen otras áreas de investigación que si bien no están fuertemente relacionadas con aspectos comparten problemas y objetivos. Tres ejemplos de ellas son: [19]

1) Reflexión y protocolos de metaobjetos : Un sistema reflectivo proporciona un lenguaje base y uno o más metalenguajes que proporcionan control sobre la semántica del lenguaje base y la implementación. Los metalenguajes proporcionan vistas de la computación que ningún componente del lenguaje base puede percibir. Los metalenguajes son de más bajo nivel que los lenguajes de aspectos en los que los puntos de enlace son equivalentes a los "ganchos" de la programación reflexiva. Estos sistemas se pueden utilizar como herramienta para la programación orientada a aspectos.

2) Transformación de programas: El objetivo que persigue la transformación de programas es similar al de la orientación a aspectos. Busca el poder escribir programas correctos en un lenguaje de más alto nivel, y luego, transformarlos de forma mecánica en otros que tengan un comportamiento idéntico, pero cuya eficiencia sea mucho mejor. Con este estilo de programación algunas propiedades de las que el programador quiere implementar se escriben en un programa inicial. Otras se añaden gracias a que el programa inicial pasa a través de varios programas de transformación. Esta separación es similar a la que se produce entre los componentes y los aspectos.

3) Programación subjetiva: es muy similar a la programación orientada a aspectos pero no representan exactamente el mismo concepto. La programación subjetiva soporta la combinación automática de métodos para un mensaje dado a partir de diferentes sujetos. Estos métodos serían equivalentes a las componentes de la POA. La diferencia clave entre ambas disciplinas es que mientras que los aspectos de la POA tienden a ser propiedades que afectan al rendimiento o la semántica de los componentes, los sujetos de la programación subjetiva son características adicionales que se agregan a otros sujetos.

5.3 POA: Ventajas y desventajas

5.3.1 Ventajas

R. Laddad en [18], enumera algunas de las principales ventajas del paradigma:

- Ayuda a superar los problemas causados por el *Código Mezclado* y *Código Diseminado* (citados previamente en este trabajo).
- 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 evolucionabilidad: La separación de conceptos permite agregar nuevos aspectos, modificar y / o remover aspectos existentes fácilmente.
- Uno puede retrasar las decisiones de diseño sobre requerimientos actuales o que surjan en el futuro, ya que permite, luego, implementarlos separadamente, e incluirlos automáticamente en el sistema; resolviendo el dilema del arquitecto: ¿Cuántos recursos invertir en el diseño? ¿Cuándo es "demasiado diseño"?
- Mayor reusabilidad: Al ser implementados separadamente, tiene mayor probabilidades de ser reusados en otros sistemas con requerimientos similares.

De nuestro análisis del paradigma, podemos agregar otras tres ventajas:

- Dividir y conquistar: Al separar la funcionalidad básica de los aspectos, se aplica con mayor intensidad el principio de dividir y conquistar.
- N-dimensiones: Ahora se tiene la posibilidad de implementar el sistema con las dimensiones que sean necesarias, no una única dimensión "sobrecargada".
- En pocas palabras, hace énfasis en el principio de mínimo acoplamiento y máxima cohesión.

5.3.2 Desventajas

En el análisis de los LOA se han hallado tres falencias [1], las cuáles surgen del hecho de que la POA está en su infancia:

- Posibles choques entre el código funcional (expresado en el lenguaje base) y el código de aspectos (expresados en los lenguajes de aspectos). Usualmente estos choques nacen de la necesidad de violar el encapsulamiento para implementar los diferentes aspectos, sabiendo de antemano el riesgo potencial que se corre al utilizar estas prácticas.

- 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. Uno de los ejemplos más conocidos de este problema es la *anomalía de herencia* [10]. Dentro del contexto de la POA, el término puede ser usado para indicar la dificultad de heredar el código de un aspecto en la presencia de herencia.

Nuestro análisis nos permite observar también que los lenguajes orientados a aspectos actuales no cuentan con mecanismos lingüísticos suficientemente poderosos para respetar por completo todos los principios de diseño, como por ejemplo, el encapsulamiento.

Apéndice A

Clase Tftpd

```
/*
   Tftpd.java - a TFTP Protocol Version 2 Server
   Copyright (C) 1999 Mark Benvenuto <mcb54@columbia.edu>

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

package gnu.inet.tftp;
import java.io.*;
import java.net.*;
import java.text.SimpleDateFormat;
import java.util.*;
/**

Mark's Tftp Server<BR>
Trival File Transfer Protocol (RFC 1350)<BR>
<BR>
Mark Benvenuto <mcb54@columbia.edu><BR>
Java JDK 1.1.8<BR>
<BR>
A TFTP server conforming to TFTP prtocol version 2 (RFC 1360)<BR>
Under GNU License Version 2.<BR>
<BR>
Version History<BR>
<BR>
      Check NEWS & ChangeLog files<BR>

@author Mark Benvenuto
@version 0.8
@see TftpServerConnection

*/

public class Tftpd
{
    //Properties
    String rootDirectory = "tftpdroot";
    int logging = 1;
    int logStdout = 1;
    int logUsingSyslogd = 0; //to be implmented in next version
    int socketTimeout = 3000;
    //int maxRetries = 5;
```

```
int debugMode = 0;
String logFile = "tftpd.log";
//Logging Stuff
private File lFile = null;
private FileOutputStream fout = null;
private TftpLogStream logTftpStream = null;
private PrintStream logStream = null;
//Date Stuff
//private Date logDate = null;
//private SimpleDateFormat logDateFormat = null;
//Network Socket Stuff
InetAddress iahost = null;

/**
        Constructs a new Tftpd server
*/
public Tftpd()
{
    if( loadProperties("tftpd.properties") == -1)
    {
        System.exit(-1);
    }

    initializeLogging();

    logStream.println("Java tftpd Version 0.8 by Mark Benvenuto <mcb54@columbia.edu>");

    try {
        iahost = iahost.getLocalHost();
        logStream.println("Host: " + iahost.getHostName() + "(" + iahost.getAddress()+ ")");
    } catch( UnknownHostException uhe)
    {
        logStream.println("Unknown local host. Must not be connected " +
                          "to network. Address is localhost(127.0.0.1)");
        logStream.println("Host: localhost(127.0.0.1)");
    }

    logStream.println("Properties Dump...");
    logStream.println("rootDirectory: " + rootDirectory );
    logStream.println("logging: " + logging );
    logStream.println("logFile: " + logFile );
    logStream.println("logStdout: " + logStdout );
    logStream.println("socketTimeout: " + socketTimeout );
    //logStream.println("maxRetries: " + maxRetries );
    logStream.println("debugMode: " + debugMode );

}

/**
        Initializes Logging Based on global variables
*/
private void initializeLogging()
{
    if( logging == 1) {
        System.out.println("Initializing Logging...");

        //First try syslogd if user wants,
        //then file user chose, then stdout
        if( logUsingSyslogd == 1) {
                //TODO: do something here in the next version
```

```
            }
            try {
                if( logStdout == 0)
                {
                    lFile = new File(logFile);
                    fout = new FileOutputStream( lFile );
                    logTftpStream = new TftpLogStream();
                    logTftpStream.setLogPrimary(new PrintStream(new
                                                BufferedOutputStream(fout) ) );
                    logTftpStream.useLogPrimary(true);
                    logStream = new PrintStream( new LineBufferedOutputStream( logTftpStream ) );
                } else if ( logStdout == 1)
                {
                    logStream = System.out;
                }
            } catch ( IOException ioe) {
                System.out.println("Log file not found.");
                System.out.println("Exception: " + ioe.getMessage());
                ioe.printStackTrace(System.out);
                logStream = System.out;
            }
        }
    }

    /**
            Gets the PrintStream attached to the current log stream
    */
    public PrintStream getLogWriter()
    {
        return logStream;
    }

    /**
            Load properties file and initialize global variables

            @param filename File that contains properties
            @return -1 indicates failed to load, otherwise 0
    */
    private int loadProperties( String filename)
    {
        File pfile = new File( filename );
        if( pfile.exists() == false) {
                System.out.println("Properties file not found. Exiting...");
                return -1;
        }
        try {
            FileInputStream fin = new FileInputStream( pfile );
            Properties props = new Properties();
            props.load(fin);
            //Load Properties
            rootDirectory = props.getProperty( "rootDirectory", "tftpdroot" );
            logging = Integer.parseInt( props.getProperty("logging", "1" ), 10);
            logFile = props.getProperty("logFile", "tftpd.log");
            logUsingSyslogd = Integer.parseInt( props.getProperty("logUsingSyslogd", "1" ), 10);
            logStdout = Integer.parseInt( props.getProperty("logStdout", "1" ), 10);
            socketTimeout = Integer.parseInt( props.getProperty( "socketTimeout", "3000"), 10);

            if( socketTimeout < 0)
                        socketTimeout = 3000; //Cannot have negative times now can we
            //maxRetries = Integer.parseInt( props.getProperty("maxRetries", "5"), 10);
```

```
            debugMode = Integer.parseInt( props.getProperty("debugMode", "0          "), 10);
            return 0;
        } catch ( FileNotFoundException fnfe) {
                System.out.println("Properties file not found. Exiting...");
                System.out.println("Exception: " + fnfe.getMessage());
                fnfe.printStackTrace(System.out);
                return -1;
        } catch ( IOException ioe) {
                System.out.println("Properties file failed to load. Exiting...");
                System.out.println("Exception: " + ioe.getMessage());
                ioe.printStackTrace(System.out);
                return -1;
        }
    }

    /**
                Main Server Loop. Processes all incoming RRQ/WRQ requests
                and spawns necessary TftpServerConnections

                @see TftpServerConnection
    */
    public void processConnections()
    {
        if( debugMode > 0 )
                logStream.println("Starting Main Server Loop...");
        int localport = 69; //Default TFTP port
        int maxPacketSize = 516;
        int opcode = 0;
        byte data[] = new byte[maxPacketSize];
        DatagramPacket pdata =new DatagramPacket(data, maxPacketSize);
        String filename = null;
        InetAddress ria=null;
        int remoteport=0;
        TftpServerConnection tftpsc = null;
        try {
            DatagramSocket mainSock = new DatagramSocket( localport );
            if( debugMode > 0 )
                logStream.println("Created Main Socket: " + mainSock);

            mainSock.setSoTimeout( socketTimeout );
            for(;;)
                try {
                    mainSock.receive( pdata );

                    //See what type of Packet it is
                    opcode = data[1];
                    remoteport = pdata.getPort();
                    ria=pdata.getAddress();
                    filename = readString(data,2);

                    //Make a formal log of from who and what they want
                    if( opcode == 1) //We have a RRQ
                            logStream.println("Received Read Request from " + ria + " for " + filename);
                    else if( opcode == 2) //We have a WRQ
                            logStream.println("Received Write Request from " + ria + " for " + filename);
                    if( debugMode > 0 )
                            logStream.println("Received Packet from: " + ria + ":" + remoteport);
                    filename = rootDirectory + File.separator + filename;

                    if( opcode == 1) { //We have a RRQ
```

```
                        //Start up a new ServerConnection, and hand off processing
                        tftpsc = new TftpServerConnection(ria, remoteport, filename, 1, logStream,
                                                          debugMode);
                        tftpsc.setSocketTimeout( socketTimeout );
                        Thread th = new Thread( tftpsc );
                        tftpsc = null;
                        th.start();
                    }
                    else if( opcode == 2) { //We have a WRQ
                        //Start up a new ServerConnection, and hand off processing
                        tftpsc = new TftpServerConnection(ria, remoteport, filename,
                                                          2, logStream, debugMode);
                        tftpsc.setSocketTimeout( socketTimeout );
                        Thread th = new Thread( tftpsc );
                        tftpsc = null;
                        th.start();
                    }
                    else {
                        logStream.println("Unknown opcode: " + opcode + " from " + ria);
                    }
                }catch( InterruptedIOException iioe) {
                        if( debugMode > 1 )
                            logStream.println("Socket Timedout. Trying again.");
                        logStream.flush();
                }catch( IOException ioe) {
                        logStream.println("Error: Main Socket Failed with IO Exception");
                        ioe.printStackTrace( logStream );
                }
            } catch (SocketException se) {
                logStream.println("Error: Failed to open Main Server Socket");
                se.printStackTrace( logStream );
            }
        }
        /*
                Used to read a string from an array of bytes, like a packet

                @param rs Bytes from a packet to read for string
                @param off Offset is the position to start reading from
                @return A String from the packet, could be zero length
        */
        protected String readString(byte[] rs, int off)
        {
            String s = new String();
            off--;
            while(rs[++off]!=(byte)'\0')
                s += String.valueOf((char)rs[off]);
            return s;
        }

        /**
                Creates a Tftpd server and start processing connections
        */
        public static void main(String args[])
        {
            Tftpd server;
            server = new Tftpd();
            server.processConnections();
        }
    }
```

Clase TftpServerConnection

```
/*
   TftpServerConnection.java - a TFTP Protocol Version 2 Server Connection Manager
   Copyright (C) 1999 Mark Benvenuto <mcb54@columbia.edu>

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

package gnu.inet.tftp;
import java.net.*;
import java.io.*;
import java.util.StringTokenizer;

/**
TftpServerConnection

A class to manage either a read or write tftp server side request

@author Mark Benvenuto
@version 0.8
@see Tftpd
@see TftpReadConnection
@see TftpWriteConnection

*/
public class TftpServerConnection implements Runnable
{
    InetAddress cia;
    DatagramSocket conn;
    int localport, remoteport;
    String localfile;
    int error, debug, maxPacketSize = 516;
    int TransferType;
    PrintStream logWriter;
    static final short TFTP_RRQ = 1;
    static final short TFTP_WRQ = 2;
    static final short TFTP_DATA = 3;
    static final short TFTP_ACK = 4;
    static final short TFTP_ERROR = 5;
    static final String TFTP_netascii = "netascii";
    static final String TFTP_octet = "octet";
    static final String TFTP_mail = "mail";

    /**
      Creates a new instance of TftpServerConnection
```

```
   @param ia The internet address of the remote host
   @param rport The remote port of the client
   @param localFilename The filename of the local file
   @param rtype The request type, 1 for read, 2 for write
   @param logStream A PrintWriter to logging all errors to
   @param dbstate A flag to enable debug/verbose support, 0 for no debuging
   @see java.net.InternetAddress
*/
TftpServerConnection(InetAddress ia, int rport, String localFilename, int rtype, PrintStream
                                logStream, int dstate)
{
    debug = dstate;
    cia = ia;
    remoteport = rport;
    localfile = localFilename;
    TransferType = rtype;
    logWriter = logStream;

    //Create Socket
    try{
            conn = new DatagramSocket();
    }catch( SocketException e)
    {
            logWriter.println("Datagram Socket Creation Failed: " + e);
            conn = null;
            return;
    }

    localport = conn.getLocalPort();

    if( debug > 0)
    {
            logWriter.println("Remote Address: " + cia);
            logWriter.println("Remote Port: " + remoteport);
            logWriter.println("Local File: " + localfile);
            logWriter.println("Transfer Type: " + TransferType);
            logWriter.println("Log Writer: " + logWriter);
    }
}
/**
  Processes the TftpServerConnection
*/
public void run()
{
    if(conn == null)
            return;
    if(TransferType == 1)  //Read Request - send file
    {
        //Send the File
        if(debug > 0)
            logWriter.println("Reading the file: " + localfile);

        //Hande Data
        //Check here so that we don't initiate a write even though
        //we might not have the file
        File inFile = new File(localfile);
        if(inFile.exists() == false)
        {
            logWriter.println("File does not exist");
            return;
```

```
            }

            if(debug > 0)
                logWriter.println("Created Socket: " + conn);

            try{
                FileInputStream fstream = new FileInputStream(localfile);
                TftpWriteConnection tftpWConn=new TftpWriteConnection(conn,cia,fstream, debug);
                tftpWConn.setLogWriter( logWriter );
                tftpWConn.setConnectionMode(true);
                tftpWConn.setRemotePort( remoteport );
                tftpWConn.processConnection();
                fstream.close();
                logWriter.println("Transfer Successfully Completed");
            }catch(IOException IOe){
                logWriter.println("FileInputStream Creation failed: " + IOe);
                return;
            }
        }
        else if(TransferType == 2) //Write Request - read the file
        {
            //Read the File
            if(debug > 0)
                logWriter.println("Writing the file: " + localfile);

            if(debug > 0)
                logWriter.println("Created Socket: " + conn);

            //Hande Data
            File outFile = new File(localfile);
            try{
                FileOutputStream fstream = new FileOutputStream(localfile);
                TftpReadConnection tftpRConn=new TftpReadConnection(conn, cia, fstream, debug);
                tftpRConn.setLogWriter( logWriter );
                tftpRConn.setConnectionMode(true);
                tftpRConn.setRemotePort( remoteport );
                tftpRConn.processConnection();
                fstream.close();
                logWriter.println("Transfer Successfully Completed");
            }catch(IOException IOe){
                logWriter.println("FileOutputStream Creation failed: " + IOe);
                return;
            }
        }
    }
    /**
      Sets the socket timeout for the connection socket
      which applies also to TftpReadConnection and TftpWriteConnection.

      @param time The new socket timeout in milliseconds
      @see TftpReadConnection
      @see TftpWriteConnection
    */
    void setSocketTimeout(int time)
    {
        try{
                conn.setSoTimeout(time);
        }catch( SocketException e)
        {
                System.err.println("Error on setting socket timeout: " + e);
```

```
            }
        }
        /**
          Gets the socket timeout for the connection socket
          which applies also to TftpReadConnection and TftpWriteConnection.

          @return The socket timeout in milliseconds
          @see TftpReadConnection
          @see TftpWriteConnection
        */
        int getSocketTimeout() throws SocketException
        {
                return conn.getSoTimeout();
        }
        /*
          Used to read a string from an arrary of bytes, like a packet

          @param rs Bytes from a packet to read for string
          @param off Offset is the position to start reading from
          @return A String from the packet, could be zero length
        */
        protected String readString(byte[] rs, int off)
        {
          String s = new String();
          off--;
          while(rs[++off]!=(byte)'\0')
                s += String.valueOf((char)rs[off]);
          return s;
        }

}
```

Clase TftpWriteConnection

```
/*
   TftpWriteConnection.java - a TFTP Protocol Version 2 Writer
   Copyright (C) 1999 Mark Benvenuto <mcb54@columbia.edu>

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/
package gnu.inet.tftp;
import java.net.*;
import java.io.*;
import java.util.StringTokenizer;

/**
```

```
        TftpWriteConnection

        A class to manage an entire write tftp request.
        This class processes the file as a thread.

        @author Mark Benvenuto
        @version 0.6
        @see Tftp
        @see TftpReadConnection
        @see TftpConnection

    */
    public class TftpWriteConnection
    {
        DatagramSocket sock;
        InetAddress ria;
        FileInputStream istream;
        int remoteport = 0;
        int maxPacketSize = 516;
        int debug=0;
        int maxDATAtries=3;
        boolean connectionMode = false;
        PrintStream logWriter;
        /**
          Creates a new instance of TftpWriteConnection

          @param ds A DatagramSocket for i/o
          @param ia The internet address of the remote host
          @param fstream Stream to read file from
          @param dbstate A flag to enable debug/verbose support, 0 for no debuging
          @see java.net.DatagramSocket
          @see java.net.InternetAddress
        */
        TftpWriteConnection(DatagramSocket ds, InetAddress ia, FileInputStream fstream, int dbstate)
        {
                sock = ds;
                debug = dbstate;
                ria = ia;
                istream = fstream;
                logWriter = System.out ;
        }
        /**
          Sets PrintStream to use for logging and error reporting

          @param pw PrintStream to log errors to
        */
        void setLogWriter( PrintStream pw)
        {
          if( pw != null)
                logWriter = pw;
          else
                logWriter = System.out ;
        }
        /**
          Sets this connection in a server mode. It means that in
          the main loop it will skip the first part of the loop
          when it first goes through the loop

          @param connMode true is server, false is not server (default)
        */
```

```
void setConnectionMode( boolean connMode)
{
  connectionMode = connMode;
}
/**
  Sets the remote port for this connection in a server mode.
  It is needed for the server as it must know from where to start
  sending messages to while the client automatically determines this itself.

  @param rport Remote Port to send datagrams too
*/
void setRemotePort( int rport)
{
  if( rport > 0)
        remoteport = rport;
}
/**
  Sets maximum retries for sending DATA, default 3

  @param tries Number of retries
*/
void setMaxTries(int tries)
{
  if(tries > 0)
        maxDATAtries = tries;
}
/**
  Sends a DatagramPacket representing an DATA packet

  @param tries Number of retries
  @return true if success, false otherwise
  @see java.net.DatagramPacket
*/
private boolean sendDATA(DatagramPacket p)
{
  try{
        sock.send(p);
        return true;
  }catch(IOException e)
  {
        logWriter.println("Failed to Send DATA packet: " + e);
  }
  return false;
}
/**
  Processes the write connection
*/
public void processConnection()
{
    int isFinished = 0;
    int isRead = 1, blocknum = 1, oldblocknum = 0, pblocknum = 0;
    byte[] ACKdata = new byte[maxPacketSize]; //Also used to receive errors
    int DATAtries = 0, RECVtries = 0;

    byte data[] = new byte[maxPacketSize];
    data[0] = 0;
    data[1] = 3;
    int datalength = 516;
    boolean DATAretcode = false;
    DatagramPacket ACKp = new DatagramPacket(ACKdata, ACKdata.length);
```

```
DatagramPacket p = new DatagramPacket(data, maxPacketSize);
boolean skipLoop = connectionMode;
if( skipLoop == false)
{
        remoteport = 0;
} else {
    oldblocknum = 0;
    pblocknum = 1;
    blocknum = 1;
    skipLoop = false;
    p.setAddress(ria);
    if(remoteport == 0)
    {
      logWriter.println("Failed to set remote port. Exiting");
      return;
    }
    p.setPort(remoteport);
    //Read DATA from File
    if(debug > 0)
        logWriter.println("Read DATA from File");
    try{
        datalength = istream.read(data, 4, 512);
    }catch(IOException IOe){
        logWriter.println("Failed to read DATA packet from file: " + IOe);
        return;
    }
    if(debug > 0)
        logWriter.println("Read " + datalength + " bytes");
    if(datalength < 512) //Read the last packet
    {
        isFinished = 1;
        if(datalength < 0)
        {
                datalength = 0;
        }
    }
    p.setLength(datalength + 4);

    DATAtries = 0;
    if(debug > 0)
        logWriter.println("Sending the DATA Packet");
    data[2] = (byte)(0xFF & (blocknum >> 8));
    data[3] = (byte)(0xFF & blocknum);
    if(debug > 0)
        logWriter.println("Sending Packet #: " + data[2] + data[3]);
    DATAretcode = sendDATA(p);
    //Exit if sending DATA packet failed here
    if(!DATAretcode)
        return;

    //This is the block we were expecting
    oldblocknum++;
    blocknum++;
}

while(isRead == 1)
{
    if(debug > 0)
        logWriter.println("Receiving Packet");
    try{
```

```
        sock.receive(ACKp);
    }
    catch( InterruptedIOException iioe) {
        logWriter.println("Error Receiving Packet: " + iioe);
        if(RECVtries > maxDATAtries)
        {
                logWriter.println("Waited to many times for ACK packet");
                return;
        }
        RECVtries++;
        logWriter.println("Continuing to wait for Packet");
        continue;
    }
    catch(IOException e) {
        logWriter.println("Error Receiving Packet: " + e);
        return;
    }
    RECVtries = 0;
    if(remoteport == 0)
    {
        remoteport = ACKp.getPort();
        p.setPort(remoteport);
        p.setAddress(ria);
    }
    datalength = ACKp.getLength();
    if(debug > 0)
    {
        logWriter.println("Received Packet: " + ACKp);
        logWriter.println("Type: " + ACKdata[0] + ACKdata[1]);
        if(ACKdata[1] == 3)
        {
                logWriter.println("Received DATA Packet( Len: " + datalength + " )");
                logWriter.println(ACKp.getData());
        }
        if(ACKdata[1] == 4)
        {
                 logWriter.println("Received ACK Packet( Len: " + datalength + " )");
                logWriter.println("ACK: " + (ACKdata[2] << 8) + ACKdata[3]);
        }
    }
    if(isError(ACKdata))
    {
        logWriter.println("Received ERROR Packet");
        printError(ACKdata);
        isRead = 0;
        break;
    }
    if(ACKdata[1] != 4)
    {
        //Failed to get ACK packet
        sendError((short)4, "Expected ACK Packet");
        logWriter.println("Expected ACK Packet");
        break;
    }
    pblocknum = (ACKdata[2] << 8) + ACKdata[3];

    if (pblocknum == oldblocknum)
    {

        if(isFinished == 1)
```

```
                return;
        //Send next DATA packet
        if(debug > 0)
                logWriter.println("Received Correct Block Num: " + pblocknum);

        //Read DATA from File
        if(debug > 0)
                logWriter.println("Read DATA from File");
        try{
                datalength = istream.read(data, 4, 512);
        }catch(IOException IOe)
        {
                logWriter.println("Failed to read DATA packet from file: " + IOe);
                return;
        }
        if(debug > 0)
                logWriter.println("Read " + datalength + " bytes");
        if(datalength < 512) //Read the last packet
        {
                isFinished = 1;
                if(datalength < 0)
                {
                        datalength = 0;
                }
        }
        p.setLength(datalength + 4);

        DATAtries = 0;
        if(debug > 0)
                logWriter.println("Sending the DATA Packet");
        data[2] = (byte)(0xFF & (blocknum >> 8));
        data[3] = (byte)(0xFF & blocknum);
        if(debug > 0)
                logWriter.println("Sending Packet #: " + data[2] + data[3]);
        DATAretcode = sendDATA(p);

        //Exit if sending DATA packet failed here
        if(!DATAretcode)
        return;

        //This is the block we were expecting
        oldblocknum++;
        blocknum++;

        continue;
    }
    if( pblocknum < oldblocknum)
    {
        if(debug > 0)
                logWriter.println("Received Duplicate Block Num: " + pblocknum);

        //Last DATA failed to receive
        if(DATAtries > maxDATAtries)
        {
                //We have retried to many times
                //Send an error message to be nice
                sendError((short)4, "Too Many Retries");

                logWriter.println("Failed to Receive proper ACK for DATA packet");
                if(isFinished == 1)
```

```
                                logWriter.println("File was sent off completely");
                        return;
                    }
                    //Send it again
                    if(debug > 0)
                            logWriter.println("Sending the DATA Packet");
                    data[3] = (byte)(data[3] + 1);
                    if(!sendDATA(p))
                            return;
                    DATAtries++;
                }
                else if( pblocknum > oldblocknum)
                {
                    if(debug > 0)
                            logWriter.println("Received Incorrect Block Num: " + pblocknum);
                    //We have an invalid block number
                    sendError((short)4, "Invalid BLOCK number");
                    return;
                }
            }
        }
        /**
          Creates a proper Error packet

          @param ErrorCode A code representing the error
          @param str A string describing the error, could be null
          @return Returns a DatagramPacket representing the packet
          @see java.net.DatagramPacket
        */
        DatagramPacket BuildErrorPacket(short ErrorCode, String str)
        {
            byte data[];
            int pos;
            data = new byte[5 + str.length()];


            data[0] = 0;
            data[1] = 5;

            data[2] = (byte)(0xFF & (ErrorCode >> 8));
            data[3] = (byte)(0xFF & ErrorCode);

            //Store String
            byte bytes[] = str.getBytes();
            System.arraycopy(bytes, 0, data, 4, bytes.length);
            data[4 + bytes.length] = 0;

            return new DatagramPacket(data, data.length, ria, remoteport);
        }
        /**
          Send remote computer error packet using <B>BuildErrorPacket</B>

                    @param ErrorCode A code representing the error
                    @param str A string describing the error, could be null
        */
        void sendError(short ErrorCode, String str)
        {
          try{
                    sock.send(BuildErrorPacket(ErrorCode, str));
          }catch( IOException e)
```

```
  {
        logWriter.println("Error on Sending ERROR Packet: " + e);
  }
}
/**
  Checks if packet is error packet

  @param data Bytes from a packet to check if it is an error packet. Minimum length 2
  @return True if error packet, false otherwise
*/
boolean isError(byte[] data)
{
  if(data[1] == 5)
        return true;
  return false;
}
/**
  Prints error packet. If there is not string in the packet, it uses a standard string defined
        by the RFC.

  @param data Bytes from an error packet
*/
void printError(byte[] data)
{
  logWriter.println("Error: " + (short)((data[2] << 8) + data[3]));
  String str = readString(data, 4);
  if(str.length() == 0)
        logWriter.println("ErrorStr: " + str);
  else
  {
        switch(data[3])
        {
              case 0:
                    logWriter.println("Not defined, see error message (if any).");
                    break;
              case 1:
                    logWriter.println("File not found.");
                    break;
              case 2:
                    logWriter.println("Access violation.");
                    break;
              case 3:
                    logWriter.println("Disk full or allocation exceeded.");
                    break;
              case 4:
                    logWriter.println("Illegal TFTP operation.");
                    break;
              case 5:
                    logWriter.println("Unknown transfer ID.");
                    break;
              case 6:
                    logWriter.println("File already exists.");
                    break;
              case 7:
                    logWriter.println("No such user.");
                    break;
              default:
                    logWriter.println("File not found.");
                    break;
        }
```

```
        }
      }
      /*
        Used to read a string from an arrary of bytes, like a packet

        @param rs Bytes from a packet to read for string
        @param off Offset is the position to start reading from
        @return A String from the packet, could be zero length
      */
      protected String readString(byte[] rs, int off)
      {
        String s = new String();
        off--;
        while(rs[++off]!=(byte)'\0')
                s += String.valueOf((char)rs[off]);
        return s;
      }
  }
```

Apéndice B

1: Funcionalidad básica

Clase Tftpd

```
package gnu.inet.tftp;
import java.io.*;
import java.net.*;
import java.text.SimpleDateFormat;
import java.util.*;

/**

@version 0.8
@see TftpServerConnection

*/

public class Tftpd
{
    //Properties
    String rootDirectory = "tftpdroot";
    int socketTimeout = 3000;
    //int maxRetries = 5;
    //Date Stuff
    //private Date logDate = null;
    //private SimpleDateFormat logDateFormat = null;
    //Network Socket Stuff
    InetAddress iahost = null;

    /**
      Constructs a new Tftpd server
    */
    public Tftpd()
    {
      if( loadProperties("tftpd.properties") == -1)
      {
                System.exit(-1);
      }

      try {
                iahost = iahost.getLocalHost();
       }catch( UnknownHostException uhe){}
    }

    /**
      Load properties file and initialize global variables

      @param filename File that contains properties
      @return -1 indicates failed to load, otherwise 0
    */
    private int loadProperties( String filename)
    {
        File pfile = new File( filename );
        if( pfile.exists() == false) {
```

```
            System.out.println("Properties file not found. Exiting...");
            return -1;
        }
        try {
            FileInputStream fin = new FileInputStream( pfile );
            Properties props = new Properties();
            props.load(fin);
            //Load Properties

            rootDirectory = props.getProperty( "rootDirectory", "tftpdroot" );

            socketTimeout = Integer.parseInt( props.getProperty( "socketTimeout", "3000"), 10);
            if( socketTimeout < 0)
                socketTimeout = 3000; //Cannot have negative times now can we
                //maxRetries = Integer.parseInt( props.getProperty("maxRetries", "5"), 10);

            return 0;

        } catch ( FileNotFoundException fnfe) {
                System.out.println("Properties file not found. Exiting...");
                System.out.println("Exception: " + fnfe.getMessage());
                fnfe.printStackTrace(System.out);
                return -1;
        } catch ( IOException ioe) {
                System.out.println("Properties file failed to load. Exiting...");
                System.out.println("Exception: " + ioe.getMessage());
                ioe.printStackTrace(System.out);
                return -1;
        }
    }

    /**
      Main Server Loop. Processes all incoming RRQ/WRQ requests
      and spawns necessary TftpServerConnections

      @see TftpServerConnection
    */
    public void processConnections()
    {
        int localport = 69; //Default TFTP port
        int maxPacketSize = 516;
        int opcode = 0;
        byte data[] = new byte[maxPacketSize];
        DatagramPacket pdata =new DatagramPacket(data, maxPacketSize);
        String filename = null;
        InetAddress ria=null;
        int remoteport=0;
        TftpServerConnection tftpsc = null;

        try {
            DatagramSocket mainSock = new DatagramSocket( localport );
            mainSock.setSoTimeout( socketTimeout );
            for(;;)
                try {
                    mainSock.receive( pdata );
                    //See what type of Packet it is
                    opcode = data[1]; // opcode = getOpcode(data);
                    remoteport = pdata.getPort();
                    ria=pdata.getAddress();
                    filename = readString(data,2);
```

```
                    filename = rootDirectory + File.separator + filename;

                    if( opcode == 1) { //We have a RRQ
                        //Start up a new ServerConnection, and hand off processing
                        tftpsc = new TftpServerConnection(ria, remoteport, filename, 1);
                        tftpsc.setSocketTimeout( socketTimeout );
                        Thread th = new Thread( tftpsc );
                        tftpsc = null;
                        th.start();
                    }else if( opcode == 2) { //We have a WRQ
                        //Start up a new ServerConnection, and hand off processing
                        tftpsc = new TftpServerConnection(ria, remoteport, filename, 2);
                        tftpsc.setSocketTimeout( socketTimeout );
                        Thread th = new Thread( tftpsc );
                        tftpsc = null;
                        th.start();
                    }

                } catch( InterruptedIOException iioe) {}
                catch( IOException ioe) {}

            } catch (SocketException se) {}
        }
        /*
          Used to read a string from an array of bytes, like a packet

          @param rs Bytes from a packet to read for string
          @param off Offset is the position to start reading from
          @return A String from the packet, could be zero length
        */
        protected String readString(byte[] rs, int off)
        {
          String s = new String();
          off--;
          while(rs[++off]!=(byte)'\0')
                s += String.valueOf((char)rs[off]);
          return s;
        }

        /**
          Creates a Tftpd server and start processing connections
        */
        public static void main(String args[])
        {
          Tftpd server;
          server = new Tftpd();
          server.processConnections();
        }
    }
```

Clase TftpServerConnection

```
    package gnu.inet.tftp;
    import java.net.*;
    import java.io.*;
    import java.util.StringTokenizer;

    /**
```

```
TftpServerConnection

A class to manage either a read or write tftp server side request

@version 0.8
@see Tftpd
@see TftpReadConnection
@see TftpWriteConnection

*/
public class TftpServerConnection implements Runnable
{
    InetAddress cia;
    DatagramSocket conn;
    int localport, remoteport;
    String localfile;
    int error,maxPacketSize = 516;
    int TransferType;
    static final short TFTP_RRQ = 1;
    static final short TFTP_WRQ = 2;
    static final short TFTP_DATA = 3;
    static final short TFTP_ACK = 4;
    static final short TFTP_ERROR = 5;
    static final String TFTP_netascii = "netascii";
    static final String TFTP_octet = "octet";
    static final String TFTP_mail = "mail";

    /**
      Creates a new instance of TftpServerConnection

      @param ia The internet address of the remote host
      @param rport The remote port of the client
      @param localFilename The filename of the local file
      @param rtype The request type, 1 for read, 2 for write
      @see java.net.InternetAddress
    */
    TftpServerConnection(InetAddress ia, int rport, String localFilename, int rtype)
    {
      cia = ia;
      remoteport = rport;
      localfile = localFilename;
      TransferType = rtype;

      //Create Socket
      try{
                conn = new DatagramSocket();
      }catch( SocketException e){
                        conn = null;
                        return;
      }
      localport = conn.getLocalPort();
    }
    /**
      Processes the TftpServerConnection
    */
    public void run()
    {
        if(conn == null)
                return;
        if(TransferType == 1)  //Read Request - send file
```

```
        {
            //Send the File
            //Handle Data
            //Check here so that we don't initiate a write even though
            //we might not have the file
            File inFile = new File(localfile);
            if(inFile.exists() == false)
                return;

            try{
                FileInputStream fstream = new FileInputStream(localfile);
                TftpWriteConnection tftpWConn = new TftpWriteConnection(conn, cia, fstream);
                tftpWConn.setConnectionMode(true);
                tftpWConn.setRemotePort( remoteport );
                tftpWConn.processConnection();
                fstream.close();

            }catch(IOException IOe){
                return;
            }
        }else if(TransferType == 2) //Write Request - read the file
        {

            //Read the File
            //Handle Data
            File outFile = new File(localfile);
            try{
                FileOutputStream fstream = new FileOutputStream(localfile);
                TftpReadConnection tftpRConn = new TftpReadConnection(conn, cia, fstream);
                tftpRConn.setConnectionMode(true);
                tftpRConn.setRemotePort( remoteport );
                tftpRConn.processConnection();
                fstream.close();
            }catch(IOException IOe)
            {
                return;
            }
        }
    }
    /**
      Sets the socket timeout for the connection socket
      which applies also to TftpReadConnection and TftpWriteConnection.

      @param time The new socket timeout in milliseconds
      @see TftpReadConnection
      @see TftpWriteConnection
    */
    void setSocketTimeout(int time)
    {
      try{
                conn.setSoTimeout(time);
      }catch( SocketException e){
                System.err.println("Error on setting socket timeout: " + e);
      }
    }
    /**
      Gets the socket timeout for the connection socket
      which applies also to TftpReadConnection and TftpWriteConnection.

      @return The socket timeout in milliseconds
```

```
      @see TftpReadConnection
      @see TftpWriteConnection
    */
    int getSocketTimeout() throws SocketException
    {
      return conn.getSoTimeout();
    }
    /*
      Used to read a string from an arrary of bytes, like a packet

      @param rs Bytes from a packet to read for string
      @param off Offset is the position to start reading from
      @return A String from the packet, could be zero length
    */
    protected String readString(byte[] rs, int off)
    {
      String s = new String();
      off--;
      while(rs[++off]!=(byte)'\0')
            s += String.valueOf((char)rs[off]);
      return s;
    }

}
```

Clase TftpWriteConnection

```
package gnu.inet.tftp;
import java.net.Inet*;
import java.io.*;
import java.util.StringTokenizer;

/**
TftpWriteConnection

A class to manage an entire write tftp request.
This class processes the file as a thread.

@see Tftp
@see TftpReadConnection
@see TftpConnection

*/
public class TftpWriteConnection
{
    DatagramSocket sock;
    InetAddress ria;
    FileInputStream istream;
    int remoteport = 0;
    int maxPacketSize = 516;
    int maxDATAtries=3;
    boolean connectionMode = false;
    /**
      Creates a new instance of TftpWriteConnection

      @param ds A DatagramSocket for i/o
      @param ia The internet address of the remote host
      @param fstream Stream to read file from
      @see java.net.DatagramSocket
      @see java.net.InternetAddress
```

```
*/
TftpWriteConnection(DatagramSocket ds, InetAddress ia, FileInputStream fstream)
{
  sock = ds;
  ria = ia;
  istream = fstream;
}
/**
  Sets this connection in a server mode. It means that in
  the main loop it will skip the first part of the loop
  when it first goes through the loop

  @param connMode true is server, false is not server (default)
*/
void setConnectionMode( boolean connMode)
{
  connectionMode = connMode;
}
/**
  Sets the remote port for this connection in a server mode.
  It is needed for the server as it must know from where to start
  sending messages to while the client automatically determines this itself.

  @param rport Remote Port to send datagrams too
*/
void setRemotePort( int rport)
{
  if( rport > 0)
        remoteport = rport;
}
/**
  Sets maximum retries for sending DATA, default 3

  @param tries Number of retries
*/
void setMaxTries(int tries)
{
  if(tries > 0)
        maxDATAtries = tries;
}
/**
  Sends a DatagramPacket representing an DATA packet

  @param tries Number of retries
  @return true if success, false otherwise
  @see java.net.DatagramPacket
*/
private boolean sendDATA(DatagramPacket p)
{
  try{
        sock.send(p);
        return true;
  }catch(IOException e){}
  return false;
}
/**
  Processes the write connection
*/
public void processConnection()
{
```

```
int isFinished = 0;
int isRead = 1, blocknum = 1, oldblocknum = 0, pblocknum = 0;
byte[] ACKdata = new byte[maxPacketSize]; //Also used to receive errors
int DATAtries = 0, RECVtries = 0;
byte data[] = new byte[maxPacketSize];
data[0] = 0;
data[1] = 3;
int datalength = 516;
boolean DATAretcode = false;
DatagramPacket ACKp = new DatagramPacket(ACKdata, ACKdata.length);
DatagramPacket p = new DatagramPacket(data, maxPacketSize);
boolean skipLoop = connectionMode;
if( skipLoop == false)
{
        remoteport = 0;
} else {
    oldblocknum = 0;
    pblocknum = 1;
    blocknum = 1;
    p.setAddress(ria);
    if(remoteport == 0)
    {
          return;
    }
    p.setPort(remoteport);
    //Read DATA from File
    try{
        datalength = istream.read(data, 4, 512);
    }catch(IOException IOe){
        return;
    }

    if(datalength < 512) //Read the last packet
    {
        isFinished = 1;
        if(datalength < 0)
        {
                datalength = 0;
        }
    }
    p.setLength(datalength + 4);

    DATAtries = 0;

    data[2] = (byte)(0xFF & (blocknum >> 8));
    data[3] = (byte)(0xFF & blocknum);
    DATAretcode = sendDATA(p);
    //Exit if sending DATA packet failed here
    if(!DATAretcode)
        return;

    //This is the block we were expecting
    oldblocknum++;
    blocknum++;
}

while(isRead == 1)
{
    try{
        sock.receive(ACKp);
```

```
}
catch( InterruptedIOException iioe) {
    if(RECVtries > maxDATAtries)
            return;
    RECVtries++;
    continue;
}
catch(IOException e) {
            return;
}
RECVtries = 0;
if(remoteport == 0)
{
    remoteport = ACKp.getPort();
    p.setPort(remoteport);
    p.setAddress(ria);
}
datalength = ACKp.getLength();

if(isError(ACKdata))
{
    isRead = 0;
    break;
}
if(ACKdata[1] != 4)
{
    //Failed to get ACK packet
    sendError((short)4, "Expected ACK Packet");
    break;
}
pblocknum = (ACKdata[2] << 8) + ACKdata[3];

if (pblocknum == oldblocknum) {
    if(isFinished == 1)
            return;
    //Send next DATA packet
    //Read DATA from File
    try{
            datalength = istream.read(data, 4, 512);
    }catch(IOException IOe)  {
            return;
    }
    if(datalength < 512) //Read the last packet
    {
            isFinished = 1;
            if(datalength < 0){
                    datalength = 0;
            }
    }
    p.setLength(datalength + 4);

    DATAtries = 0;
    data[2] = (byte)(0xFF & (blocknum >> 8));
    data[3] = (byte)(0xFF & blocknum);
    DATAretcode = sendDATA(p);

    //Exit if sending DATA packet failed here
    if(!DATAretcode)
            return;
```

```
                //This is the block we were expecting
                oldblocknum++;
                blocknum++;

                continue;
            }
            if( pblocknum < oldblocknum)
            {
                //Last DATA failed to receive
                if(DATAtries > maxDATAtries){
                        //We have retried to many times
                        //Send an error message to be nice
                        sendError((short)4, "Too Many Retries");
                        return;
                }
                //Send it again

                data[3] = (byte)(data[3] + 1);
                if(!sendDATA(p))
                        return;
                DATAtries++;
            }
            else if( pblocknum > oldblocknum)
            {
                //We have an invalid block number
                sendError((short)4, "Invalid BLOCK number");
                return;
            }
        }
    }
    /**
      Creates a proper Error packet

      @param ErrorCode A code representing the error
      @param str A string describing the error, could be null
      @return Returns a DatagramPacket representing the packet
      @see java.net.DatagramPacket
    */
    DatagramPacket BuildErrorPacket(short ErrorCode, String str)
    {
        byte data[];
        int pos;
        data = new byte[5 + str.length()];
        data[0] = 0;
        data[1] = 5;
        data[2] = (byte)(0xFF & (ErrorCode >> 8));
        data[3] = (byte)(0xFF & ErrorCode);
        //Store String
        byte bytes[] = str.getBytes();
        System.arraycopy(bytes, 0, data, 4, bytes.length);
        data[4 + bytes.length] = 0;
        return new DatagramPacket(data, data.length, ria, remoteport);
    }
    /**
      Send remote computer error packet using <B>BuildErrorPacket</B>

      @param ErrorCode A code representing the error
      @param str A string describing the error, could be null
    */
    void sendError(short ErrorCode, String str)
```

```
    {
      try{
                sock.send(BuildErrorPacket(ErrorCode, str));
      }catch( IOException e){}
    }
    /**
      Checks if packet is error packet

      @param data Bytes from a packet to check if it is an error packet. Minimum length 2
      @return True if error packet, false otherwise
    */
    boolean isError(byte[] data)
    {
      if(data[1] == 5)
                return true;
      return false;
    }
    /*
      Used to read a string from an arrary of bytes, like a packet

      @param rs Bytes from a packet to read for string
      @param off Offset is the position to start reading from
      @return A String from the packet, could be zero length
    */
    protected String readString(byte[] rs, int off){
      String s = new String();
      off--;
      while(rs[++off]!=(byte)'\0')
                s += String.valueOf((char)rs[off]);
      return s;
    }
}
```

2: Aspecto de logging

```
aspect logging{

int logging = 1;
int logStdout = 1;
int logUsingSyslogd = 0;
int debugMode = 0;
String logFile = "tftpd.log";
//Logging Stuff
private File lFile = null;
private FileOutputStream fout = null;
private TftpLogStream logTftpStream = null;
private PrintStream logStream = null;

/**
      Initializes Logging Based on global variables
*/
private void initializeLogging()
{
    if( logging == 1) {
        System.out.println("Initializing Logging...");
        //First try syslogd if user wants,
        //then file user chose, then stdout
        if( logUsingSyslogd == 1) {
```

```
                //TODO: do something here in the next version
            }
            try {
                if( logStdout == 0){
                    lFile = new File(logFile);
                    fout = new FileOutputStream( lFile );
                    logTftpStream = new TftpLogStream();
                    logTftpStream.setLogPrimary( new PrintStream(
                                                    new BufferedOutputStream(fout) ) );
                    logTftpStream.useLogPrimary(true);
                    logStream = new PrintStream( new
                                    LineBufferedOutputStream( logTftpStream ) );
                } else if ( logStdout == 1) {
                    logStream = System.out;
                }
            } catch ( IOException ioe) {
                System.out.println("Log file not found.");
                System.out.println("Exception: " + ioe.getMessage());
                ioe.printStackTrace(System.out);
                logStream = System.out;
            }
        }
    }

    /**
        Prints error packet. If there is not string in the packet, it uses a standard string defined by the
    RFC.

        @param data Bytes from an error packet
    */
    void printError(byte[] data)
    {
        logStream.println("Error: " + (short)((data[2] << 8) + data[3]));
        String str = readString(data, 4);
        if(str.length() == 0)
            logStream.println("ErrorStr: " + str)
        else {
            switch(data[3]) {
                case 0:
                    logStream.println("Not defined, see error message (if any).");
                    break;
                case 1:
                    logStream.println("File not found.");
                    break;
                case 2:
                    logStream.println("Access violation.");
                    break;
                case 3:
                    logStream.println("Disk full or allocation exceeded.");
                    break;
                case 4:
                    logStream.println("Illegal TFTP operation.");
                    break;
                case 5:
                    logStream.println("Unknown transfer ID.");
                    break;
                case 6:
                    logStream.println("File already exists.");
                    break;
                case 7:
```

```
                logStream.println("No such user.");
                break;
            default:
                logStream.println("File not found.");
                break;
        }
    }
}



// pointcuts

pointcut classTftpd():within(Tftpd);
pointcut classServerConnection():within(TftpServerConnection);
pointcut classWriteConnection():within(TftpWriteConnection);
pointcut anyTftp():classTftpd()||classServerConnection()||classWriteConnection();

pointcut localPort():call(InetAddress.getLocalHost());

before():localPort() && classTftpd(){

    initializeLogging();
    logStream.println("Java tftpd Version 0.8 by Mark Benvenuto
                        <mcb54@columbia.edu>");
    logStream.println("Properties Dump...");
    logStream.println("rootDirectory: " + this.rootDirectory );
    logStream.println("logging: " + logging );
    logStream.println("logFile: " + logFile );
    logStream.println("logStdout: " + logStdout );
    logStream.println("socketTimeout: " + this.socketTimeout );
    logStream.println("debugMode: " + debugMode );
}

after() returning(InetAddress iahost):localPort() && anyTftp(){
    logStream.println("Host: " + iahost.getHostName() +
                        "(" + iahost.getAddress() + ")");
 }

after() throwing(UnknownHostException uhe):localPort()&& anyTftp(){
    logStream.println("Unknown local host. Must not be connected " +
                "to network. Address is localhost(127.0.0.1)");
    logStream.println("Host: localhost(127.0.0.1)");
 }

pointcut loadProp(Properties props):call(Properties.load(FileInputStream))
                                                && target(props);

after(Properties props):loadProp(props) && classTftpd(){
    logging = Integer.parseInt( props.getProperty("logging", "1" ), 10);
    logFile = props.getProperty("logFile", "tftpd.log");
    logUsingSyslogd = Integer.parseInt( props.getProperty("logUsingSyslogd", "1" ),
                                    10);
    logStdout = Integer.parseInt( props.getProperty("logStdout", "1" ), 10);
    debugMode = Integer.parseInt( props.getProperty("debugMode", "0"), 10);
}


pointcut processConnections():call(void processConnections());
```

```
before():processConnections() && classTftpd(){
    if( debugMode > 0 )
        logStream.println("Starting Main Server Loop...");
}

pointcut createSocket():call(DatagramSocket.new(..));

after() returning (DatagramSocket socket):createSocket()&& classTftpd(){
    if( debugMode > 0 )
        logStream.println("Created Main Socket: " + socket);
}
after() throwing(SocketException se):createSocket()&& classTftpd(){
    logStream.println("Error: Failed to open Main Server Socket");
    se.printStackTrace( logStream );
}
after() throwing(SocketException se):createSocket()&& classServerConnection(){
    logStream.println("Datagram Socket Creation Failed: " + se);
}

pointcut receivePacket(DatagramPacket packet):
      call(DatagramSocket.receive(DatagramPacket))&& args(packet);

before(DatagramPacket packet):receivePacket(packet)&& anyTftp(){
    if(debugMode > 0)
        logStream.println("Receiving Packet");
 }

after(DatagramPacket packet) returning :receivePacket(packet) && classTftpd(){
    byte data[] = packet.getData();
    int opcode = data[1];
    int remoteport = packet.getPort();
    InetAddress ria=packet.getAddress();
    string filename = Tftpd.readString(data,2);
    filename = this.rootDirectory + File.separator + filename;
    //Make a formal log of from who and what they want
    if( opcode == 1) //We have a RRQ
        logStream.println("Received Read Request from " + ria + " for " + filename);
    else if( opcode == 2) //We have a WRQ
        logStream.println("Received Write Request from " + ria + " for " + filename);
    else {
        logStream.println("Unknown opcode: " + opcode + " from " + ria);
    }
    if( debugMode > 0 )
        logStream.println("Received Packet from: " + ria + ":" + remoteport);
 }
 after(DatagramPacket packet) returning :receivePacket(packet) &&
                                              classWriteConnection(){
    byte ACKdata[] = packet.getData();
    int datalength = packet.getLength();
    int sentBlockNumber = checkBlockNumber.aspectof(this).getSentBlockNumber();
    int receivedBlockNumber =
                          checkBlockNumber.aspectof(this).getReceivedBlockNumber();
    if(debugMode > 0) {
        logStream.println("Received Packet: " + packet);
        logStream.println("Type: " + ACKdata[0] + ACKdata[1]);

        if(ACKdata[1] == 3) {
            logStream.println("Received DATA Packet( Len: " + datalength + " )");
            logStream.println(ACKdata);
        }
```

```
            if(ACKdata[1] == 4) {
                logStream.println("Received ACK Packet( Len: " + datalength + " )");
                logStream.println("ACK: " + (ACKdata[2] << 8) + ACKdata[3]);

                if (sentBlockNumber == receivedBlockNumber) {
                    logStream.println("Received Correct Block Num: " +
                                      receivedBlockNumber);
                }
                if (sentBlockNumber > receivedBlockNumber) {
                    logStream.println("Received Duplicate Block Num: " +
                                      receivedBlockNumber);
                }
                if (sentBlockNumber < receivedBlockNumber) {
                    logStream.println("Received Incorrect Block Num: " +
                                       receivedBlockNumber);
                }
            }
        }
        if(ACKdata[1] == 5)  {
            logStream.println("Received ERROR Packet");
            printError(ACKdata);
        }
    }

    after(DatagramPacket packet) throwing(InterruptedIOException iioe):
                                          receivePacket(packet) && classTfptd(){
        if( debugMode > 0 )
            logStream.println("Socket Timedout. Trying again.");
        logStream.flush();
      }


    after(DatagramPacket packet) throwing(IOException ioe):
                                          receivePacket(packet) && classTfptd(){
        logStream.println("Error: Main Socket Failed with IO Exception");
        ioe.printStackTrace( logStream );
    }

    after(DatagramPacket packet) throwing(InterruptedIOException iioe ):
                            receivePacket(packet) && classWriteConnection(){
        logStream.println("Error Receiving Packet: " + iioe);
    }

    after(DatagramPacket packet) throwing(IOException ioe):receivePacket(Packet)
                                          && classWriteConnection(){
        logStream.println("Error Receiving Packet: " + ioe);
    }


    pointcut sendPacket(DatagramPacket packet):
                    call(DatagramSocket.send(DatagramPacket)) && args(packet);

    before(DatagramPacket packet):sendPacket(packet) && classWriteConnection()
                                          && withincode(boolean sendData(DatagramPacket)){
        byte data[] = packet.getData();
        if(debugMode > 0)
            logStream.println("Sending the DATA Packet #: " + data[2] + data[3]);
    }

    after(DatagramPacket packet, String s) returning :sendPacket(packet) && anyTftp()
```

```
                                        && withincode(void sendError(short,String))
                                        && args(*,s){
    logStream.println(s);
}

after(DatagramPacket packet) throwing(IOException ioe):
                                        sendPacket(packet) && anyTftp()
                                        && withincode(boolean sendData(DatagramPacket)) {
    logStream.println("Failed to Send DATA packet: " + ioe);

}

after(DatagramPacket packet) throwing(IOException ioe):
                                        sendPacket(packet) && anyTftp()
                                        && withincode(void sendError(short,String)) {
    logStream.println("Error on Sending ERROR Packet: " + ioe);
}


pointcut createServerConnection():call(TftpServerConnection.new(..));

after() returning(TftpServerConnection tsc):createServerConnection(){
    if( debugMode > 0)
    {
        logStream.println("Remote Address: " + tsc.cia);
        logStream.println("Remote Port: " + tsc.remoteport);
        logStream.println("Local File: " + tsc.localfile);
        logStream.println("Transfer Type: " + tsc.TransferType);
        logStream.println("Log Writer: " + logStream);
    }
}

pointcut run():call(TftpServerConnection.run());

before(TftpServerConnection tsc):run()&& target(tsc){
    if(tsc.conn!=null){
        logStream.println("Created Socket: " + tsc.conn);
        if(tsc.TransferType==1){
            logStream.println("Reading the file: " + tsc.localfile);
        }else if(tsc.TransferType==2){
            logStream.println("Writing the file: " + tsc.localfile);
        }
    }
}

pointcut createFile(String filename):call(File.new(String)) && args(filename);

after(String filename) returning(File f):createFile(filename)&& anyTftp(){
    if(f.exists() == false){
        logStream.println("File "+ filename +" does not exist");
    }
}

pointcut transferCompleted():call(File*Stream.close());

after():transferCompleted()&& classServerConnection {
    logStream.println("Transfer Successfully Completed");
}

pointcut createInputStream():call(FileInputStream.new(..));
```

```
after() throwing(IOException ioe): createInputStream()&& anyTftp(){
    logStream.println("FileInputStream Creation failed: " + ioe);
}


pointcut createOutputStream():call(FileOutputStream.new(..));

after() throwing(IOException ioe): createOutputStream()&& anyTftp(){
    logStream.println("FileOutputStream Creation failed: " + ioe);
}

pointcut processConnection(TftpWriteConnection twc):call(void processConnection())
                                                        && target(twc);

before(TftpWriteConnection twc): processConnection(twc){
    if( (twc.remoteport==0) && (twc.connectionMode) ){
        logStream.println("Failed to set remote port. Exiting");
    }
}

pointcut readFile():call(FileInputStream.read(..));

before():readFile()&& anyTftp(){
    if(debugMode > 0)
        logStream.println("Read DATA from File");
}

after() returning(int length):readFile()&& anyTftp(){
    if(debugMode > 0)
        logStream.println("Read " + length + " bytes");
}

after() throwing(IOException ioe):readFile()&& anyTftp(){
    logStream.println("Failed to read DATA packet from file: " + ioe);
}

aspect checkBlockNumber pertarget (TftpWriteConnection.new(..)) {

    private int rbn, sbn = 0;  /* Received Block Number,  Sent Block Number */

    /* Método que devuelve, a partir de un paquete de entrada,
       el número de bloque asociado al mismo        */

    int getBlockNumber(DatagramPacket p) {
        byte data[] = p.getData();
        return (data[2] << 8) + data[3];
    }

     int getReceivedBlockNumber() {return rbn;}
     int getSentBlockNumber() {return sbn;}


    after (DatagramPacket p) returning : sendPacket(p)&& classWriteConnection(){
        sbn = getBlockNumber(p);
    }
    after (DatagramPacket p) returning : receivePacket(p)&& classWriteConnection(){
        rbn = getBlockNumber(p);
    }
}
```

```
}
```

Referencias

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

[2] K. Mehner, A. Wagner, *"An Assessment Of Aspect Language Design"*, Position Paper in Young Researchers Workshop, GCSE ´99, 1999.

[3] Ken Anderson, *"Thoughts On Aspect-Oriented Programming"*, Summarizing Cristina Lopes' work, Northeastern University, 1996.

[4] Claire Tristram, "*Untangling Code"*, in the January/February 2001 issue of the Technology Review.

[5] Junichi Suzuki, Yoshikazu Yamamoto, *"Extending UML with Aspects: Aspect Support in the Design Phase"*, 3er Aspect-Oriented Programming(AOP) Workshop at ECOOP ´99.

[6] Pascal Fradet, Mario Südholt, *"An Aspect Language for robust programming"*, in Proceedings of the European Conference on Object-Oriented Programming (ECOOP) Workshops 1999.

[7] Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga, "*Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming"*, in the Workshop su Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi (WSDAAL). Septiembre 1999.

[8] Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, *"Aspect-Oriented Programming"*, in Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. Junio 1997.

[9] Gregor Kickzales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Grisnold, *"Getting Started with AspectJ"*, in Communications of the ACM (CACM) , Vol 44, Nº 10, Octubre 2001.

[10] S. Matsuoka, A. Yonezawa, *"Analysis of inheritance anomaly in object-oriented concurrent programming languajes"*, in Research Directions in Concurrent Object-Oriented Programming (G. Agha, P.Wegner, and A. Yonezawa, eds.),pp. 107-150, Cambridge, MA: MIT Press, 1993.

[11] Parnas D.L., *"On the Criteria to be used in decomposing Systems into Modules"*, in Communications of the ACM,vol. 15(2),1972.

[12] Parnas D.L., *"Designing Software for Extension and Contractions"*, in Proceedings 3er International Conference on Software Engineering,pp. 264-277,1978.

[13] Página del grupo Demeter: http://www.ccs.neu.edu/research/demeter/ .

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

[15] Timothy Highley, Michael Lack, Perry Myers, *"Aspect-Oriented Programming: A Critical Analysis of a new programming paradigm"* , University of Virginia, Department of Computer Science, Technical Report CS-99-29, Mayo 1999.

[16] MacLennan B, *"Principles of Programming Languages: Design, Evaluation and Implementation"*, Oxford: NY, 1987.

[17] Carlo Ghezzi, Mehdi Jazayeri, *"Programming language concepts"*, 3º Edición, John Wiley & Sons, 1998.

[18] Ramnivas Laddad, *"I want my AOP", Part 1,2 and 3*, from JavaWorld, Enero-Marzo-Abril 2002.

[19] Antonia Mª Reina Quintero, *"Visión General de la Programamción Orientada a Aspectos"*, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, diciembre de 2000.

[20] L.Berger, *"Junction Point Aspect: A Solution to Simplify Implementation of Aspect Languages and Dynamic Management of Aspect Programs"*, in Proceedings of ECOOP 2000, Junio de 2000, Francia.

[21] 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.

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

[23] Cristina Lopes, *"D: A Language Framework for Distributed Programming"*, Ph.D. thesis, Northeastern University, Noviembre de 1997.

[24] Mattia Monga, *"Concern Specific Aspect-Oriented Programming with Malaj"*, Politecnico de Milano, Dip. Di Elettronica e Informazione, Milán, Italia.

[25] Gianpaolo Cugola, Carlo Ghezzi, Mattia Monga, Gian Pietro Picco, *"Malaj: A Proposal to Eliminate Clashes Between Aspect-Oriented and Object-Oriented Programming"*, Politecnico de Milano, Dip. Di Elettronica e Informazione, Milán, Italia.

[26] Robert Hirschfeld, *"AspectS – AOP with Squeak"*, in Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented System, Agosto de 2001.

[27] Yvonne Coady, Gregor Kickzales, Mike Feeley, Greg Smolyn, *"Using AspectC to improve the modularity of path-specific customization in operating system code"*, in Proceedings of Joint ESEC and FSE-9, 2001.

[28] Harold Ossher, Peri Tarr, *"Multi-Dimensional Separation of Concerns and the Hyperspace Approach"*, in Proceedings of the Symposium on Software Arquitectures and Component Technology: The state of the Art in Sofware Development. Kluwer, 2001.

[29] Andreas Gal, Wolfgan Schroder, Olaf Spinczyk, *"AspectC++: Language Proposal and Prototype Implementation"*, University of Magdeburg, Alemania, 2001.

[30] Kris de Volder, Theo D' Hondt, "*Aspect-Oriented Logic Meta Programming*", in Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99. LNCS 1616, pp. 250-272, Springer-Verlag,1999.

[31] Johan Brichau, *"Declarative Composable Aspects"* , in Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns, Septiembre de 2000.

[32] Ralf Lammel, Gunter Riedewald, Wolfgang Lohmann, *"Adaptation of functional object programs"*, Position Paper at the ECOOP ´99 Workshop on Aspect-Oriented Programming.

[33] Ralf Lammel, "*Declarative aspect-oriented programming"*, in Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1, páginas 131-146, Enero de1999.

[34] B. Wulf, M. Shaw, *"Global Variables Considered Harmful"*, SIGPLAN Notices, vol.8, No. 2, February, 1972.

[35] El sitio de AspectJ: www.aspectj.org . Palo Alto Research Center.

[36] Johan Brichau, Wolfgang De Meuter, Kris de Volder, *"Jumping Aspect"*, Programming Technology Lab, Vrije Universiteit Brussel, Bélgica, Abril de 2000.

[37] Página de Mark Benvenuto: *http://www.cs.columbia.edu/~markb/* .

[38] Página de Squeak: http://squeak.org .

[39] Página de TyRuBa: http://tyruba.sourceforge.net .

[40] Página de Karl Lieberherr: http://www.ccs.neu.edu/home/lieber/ .

[41] Página de Mehmet Aksit: http://trese.cs.utwente.nl/aksit/aksit_trese.htm .

[42] Página de Mira Mezini: http://www.informatik.uni-siegen.de/~mira/ .

[43] Wai-Ming Ho, François Pennaneach, Jean Marc Jezequel, Noël Plouzeau, *"Aspect-Oriented Design with the UML"*, in Proceedings of Multi-Dimensional Separation of Concerns Workshop at ICSE, junio de 2000.

[44] Siobhan Clarke, John Murphy. *"Developing a Tool to support Aspect-Oriented Programming principles to the Design phase"*, in Proceedings of ICSE '98 Workshop on Aspect-Oriented Programming.

[45] José Luis Herrero, Fernando Sánchez, Fabiola Lucio, Miguel Toro, *"Introducing Separation of Aspects at Design Time"*, in Proceedings of AOP Workshop at ECOOP '00, Cannes, France, junio de 2000.

[46] R. Buhr, *"A possible design notation for aspect oriented programming"*, in Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, julio de 1998.

[47] Pascal Fradet, Mario Südholt, *"AOP: towards a generic framework using program transformation and analysis"*, in Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, julio de 1998.

[48] Wolfgang De Meuter, *"Monads as a theoretical foundation for AOP"*, in Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97.

[49] James H. Andrews, *"Using process algebra as a foundation for programming by separation of concerns"*, Department of Computer Science, University of Western Ontario, Ontario, Canada.