Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Departamento de Computación

Énfasis:
Programación Orientada a Aspectos de Grano Fino

Tesis que presenta el

M. en C. Ulises Juárez Martínez

para obtener el Grado de

Doctor en Ciencias
en la Especialidad de Ingeniería Eléctrica

Director de Tesis
Dr. José Oscar Olmedo Aguirre

México D. F. Octubre 2008

II

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

De manera muy especial a mis padres, por todo el apoyo y entusiasmo para la realización de esta tesis, por cada una de sus palabras, por su tiempo, por su tolerancia, por los sabios consejos, por la vida y sobre todo por haberme escuchado en esos momentos difíciles... por tantas cosas que hacen una vida en un hijo.

Al Dr. Jorge Buenabad Chávez, a los maestros Ivonne Rodríguez, Job Morales, José Rogelio Reyes Reyes y al ingeniero José Eduardo Terán Rendón por sus valiosos comentarios sobre la introducción de la tesis.

En especial quiero agradecer a la Dra. Leticia Dávila Nicanor por su tiempo y dedicación en la revisión de toda la tesis y por su valiosa crítica, retroalimentación y apoyo para mejorar el trabajo.

A Sofía Reza Cruz por su invaluable apoyo en trámites académicos y administrativos, principalmente por su constante dedicación hacia uno como estudiante.

A Flor Córdova González y Felipa Rosas López por su apoyo administrativo en salidas a congresos y a Graciela Meza Castellanos por sus finas atenciones en biblioteca.

Al maestro, amigo y mejor analista y diseñador de objetos que he conocido, Manuel Cota Aguilar, quien por una conversación telefónica me permitió conocer e interesarme en la programación orientada a aspectos.

Al Consejo Nacional de Ciencia y Tecnología por el apoyo de becas para la realización de mis estudios doctorales.

Resumen

La Programación Orientada a Aspectos es un nuevo paradigma de programación que permite modularizar atributos (requerimientos no funcionales) de un sistema como persistencia, desempeño, calidad, registro (logging), concurrencia, entre otros. Estos atributos requieren de algunos miembros de instancia o de clase para llevar a cabo su función. Sin embargo, para modularizar atributos como prueba, delineado (tracing), visualización de programas, monitoreo y complejidad, se requieren elementos más finos de un programa, específicamente las variables locales. Los lenguajes de programación orientados a aspectos como AspectJ y similares no soportan la modularización de atributos a nivel de variables locales, por lo que encapsular y reutilizar un atributo como la verificación de variables no es posible. En este trabajo se presenta a ÉNFASIS, una plataforma en Java para definir aspectos de grano fino en donde la localización de variables locales se basa en un mecanismo de rutas que considera los árboles de sintaxis abstracta del código fuente. Mediante este mecanismo se establece un mapeo entre las variables locales disponibles en el código fuente y las correspondientes instrucciones de bytecode permitiendo mantener una descripción de alto nivel y al mismo tiempo definir un modelo computacional basado en teoría de conjuntos para este tipo de aspectos finos. Entre las aplicaciones de aspectos de grano fino se pueden mencionar el análisis de flujo de datos (información sobre posibles conjuntos de valores en varios puntos del programa), comprensión de programas, manejo de aseveraciones y la cobertura de código (prueba de software para funciones, sentencias, condiciones, rutas, etc.).

Abstract

Aspect-Oriented Programming is a new programming paradigm that allows system attributes (non-functional requirements) modularization such as persistence, performance, quality, logging, concurrency, among others. These attributes require some class or instance members to carry out its function. However, modularizing attributes like testing, tracing, program visualization, monitoring and complexity, it requires finer elements of a program, specifically the local variables. Aspect-oriented languages such as AspectJ and AspectJ-like languages do not support attribute modularization for local variables, thus the encapsulation and reuse of an attribute such as verification for variables is not possible. This thesis introduces ÉNFASIS, a Java platform for defining fine-grained aspects where the location of local variables is based on a path mechanism. Such mechanism uses the abstract syntax trees of source code. A mapping between local variables available in source code and related bytecode instructions allows a high level description and a computational model based on set theory. Applications of fine-grained aspects include data flow analysis (information on possible sets of values at various points of the program), program comprehension, assertions and code coverage (software testing for functions, sentences, conditions, paths, etc.).

Capítulo 1

Introducción

La complejidad en los sistemas de software, principalmente en el software de nivel industrial, es una característica natural que desafortunadamente no puede eliminarse pero sí es posible alcanzar un conocimiento profundo de ella [12]. Para entender un sistema complejo es necesario dividirlo o descomponerlo en partes suficientemente pequeñas hasta un nivel que permita entender su funcionamiento de manera individual. Esta técnica de descomposición se conoce como *divide et impera* (divide y vencerás) y dependiendo del método de descomposición utilizado, es el tipo de unidades modulares que se obtienen. La descomposición algorítmica y la descomposición orientada a objetos son dos visiones que generan unidades modulares diferentes, funciones (o procedimientos) y objetos respectivamente. La composición del sistema depende del tipo de unidades obtenidas por el proceso de descomposición.

En la ingeniería del software, la técnica de *divide et impera* se conoce como el *principio de separación de asuntos* (*separation of concerns: SOC*). Un asunto[1] o *concern* es un atributo, propiedad, criterio, consideración o requisito específico y relevante para la realización del software [19]. Es decir, un sistema de software requiere la realización de un conjunto de asuntos. Por ejemplo, un sistema bancario implica la realización de cliente, administración de cuenta, cálculo de interés, transacciones interbancarias, persistencia de las entidades, autorización de acceso a servicios, etc. Además el proyecto de software requiere de otros asuntos como capaci-

[1] Materia de que se trata.

dad de mantenimiento, comprensibilidad y fácil evolución. Los asuntos como cliente, cálculo de interés, transacción bancaria y cuenta se conocen como *asuntos primarios* porque capturan la funcionalidad central del sistema, mientras que los demás asuntos se conocen como *secundarios* porque están compartidos entre los asuntos primarios, como lo es la persistencia y la autorización de acceso.

Cada método de descomposición permite separar asuntos en términos de las unidades que producen, limitando su implementación en un sólo tipo de unidad, ya sea procedimiento u objeto. Para los asuntos que capturan la funcionalidad central del sistema, un solo tipo de unidad es adecuado, pero en el caso de los asuntos secundarios normalmente se recurre a su implementación en forma fragmentada, es decir, quedan *dispersos* entre varias unidades. A los asuntos secundarios se les conoce como *asuntos de corte* (*crosscutting concerns*), es decir, asuntos que cortan e interfieren con la funcionalidad de otros. Por ejemplo, el asunto de autorización afecta a cada módulo con requisitos de control de acceso. Otros ejemplos de asuntos de corte son [12, 51]: autenticación, administración, registro (logging), desempeño, administración de almacenamiento, persistencia de datos, seguridad, concurrencia, integridad de transacciones, monitoreo, rendimiento, etc. Los asuntos de corte no pueden encapsularse en un procedimiento u objeto debido a la dispersión de su implementación. Esto causa una reducción en la capacidad de comprensión del sistema, incrementa el impacto al cambio, se genera una mayor facilidad de cometer errores y por consecuencia hay un aumento en la dificultad de la evolución del software [15, 47, 51].

La *programación orientada a aspectos* (POA) es una técnica de programación que permite encapsular asuntos de corte [47]. La POA se construye y trabaja sobre las metodologías existentes, tanto estructuradas como de objetos, permitiendo nuevos niveles de modularidad. Un *aspecto* es un tipo particular de asunto y un asunto es cualquier código relacionado a un objetivo, característica o tipo de funcionalidad [15].

En el ámbito de los objetos, la realización del sistema se basa en el intercambio de mensajes entre los objetos que lo componen. Con el enfoque de aspectos, la realización del sistema se produce mediante la identificación de eventos los cuales permiten coordinar las interacciones

con los objetos. Este esquema de coordinación, conocido como *entrelazado* (*weaving*), se basa en ciertos puntos bien definidos en la ejecución de un programa, los cuales se conocen como *puntos de unión* (*join points*). Un punto de unión puede ser la llamada a un método, la ejecución de un constructor o el acceso a un campo en el momento de una asignación. La capacidad de un lenguaje orientado a aspectos para entrelazar aspectos con objetos define la expresividad de su *modelo de puntos de unión*. Este modelo provee un marco de referencia común para permitir la ejecución de aspectos y objetos en forma transparente [45].

1.1. Definición del problema

En el contexto de la ingeniería del software, la prueba de software es una actividad que pretende evaluar un atributo o la capacidad de un programa o sistema y determinar que se pueden obtener los resultados requeridos [33]. Para realizar esta actividad se requiere de la intervención por parte de un humano, la cual es al mismo tiempo una fuente de introducción de errores. La POA ofrece la alternativa de reducir considerablemente esta fuente de introducción de errores al modularizar las pruebas de software y otros asuntos que están relacionados a actividades similares como la comprensión y visualización de programas. La capacidad y facilidad de tener acceso a cualquier elemento del programa, como las variables locales, se vuelve esencial para estas actividades.

Los lenguajes orientados a aspectos disponibles [4, 11, 39, 45, 61, 62, 71] definen un modelo de puntos de unión que opera esencialmente con los miembros definidos en una clase: campos, métodos y constructores. Muchos aspectos como persistencia, rastreo y rendimiento pueden implementarse fácilmente con este modelo de puntos de unión. Sin embargo, elementos como las variables locales no están disponibles para implementar aspectos tales como complejidad, prueba, monitoreo y manejo de aseveraciones, los cuales requieren de un modelo de puntos de unión de baja granularidad, es decir, identificación de eventos más finos que los miembros de una clase o instancia.

Dentro de la gran diversidad de lenguajes orientados a aspectos, AspectJ es el lenguaje

con uno de los modelos de puntos de unión más expresivos. AspectJ [46, 51] es una extensión de propósito general para el lenguaje Java y es el lenguaje en los que otros lenguajes han basado su modelo de programación [4, 61]. Independientemente del soporte que ofrecen estos lenguajes para definir aspectos, ninguno de ellos define un mecanismo de entrelazado que sea capaz de interactuar con las variables locales, limitando el tipo de aspectos que pueden definirse y encapsularse.

El estudio de nuevas formas de expresividad en los lenguajes orientados a aspectos [69], la definición de aspectos genéricos mediante mecanismos de extensión [21], nuevas formas para identificar puntos de unión mediante la combinación de paradigmas [65] y mejor soporte para aplicar aspectos en dominios específicos [82] muestran la evidente carencia de los actuales lenguajes de aspectos para soportar el entrelazado con variables locales, es decir, definir e implementar *aspectos de grano fino*. Sin embargo, definir un modelo de puntos de unión con estas características afronta varios retos con las variables locales, principalmente porque un método puede contener una gran cantidad de *ámbitos* definidos por un bloque o sentencia de código. En este sentido, las variables locales poseen las siguientes características:

1. Son válidas sólo en determinados ámbitos de un método, como en el interior de un ciclo `for`, en el interior de un `if` o en un bloque arbitrario como es el caso de las iniciaciones de clase e instancia.

2. Aunque tengan el mismo tipo y nombre, las declaraciones de variables son diferentes en cada ámbito del método.

3. No son visibles fuera del método donde se declaran, ni son visibles fuera de algún ámbito del mismo método.

4. Pueden aparecer múltiples veces en una misma línea de código, sentencia o expresión.

5. Su manipulación no es flexible. Una variable local que forma parte de una expresión de una sentencia `if` no puede contener otro tipo de código que no sea la expresión misma. Esto aplica de la misma forma en expresiones complejas.

6. Conocer sus valores implica tener acceso al entorno de ejecución del programa, donde generalmente no se cuenta con información para la identificación de las variables.

7. Diferentes compiladores del lenguaje Java como Jikes de IBM [36] y otros como Kopi [50] generan diferentes secuencias de instrucciones de código intermedio para un mismo programa. En estos casos las clases generadas por cada compilador tienen la misma funcionalidad pero con instrucciones similares o en diferente orden.

Un modelo de puntos de unión que permita la definición de aspectos de grano fino debe solventar los retos de las variables locales mediante un mecanismo que permita extraer información estática (código fuente) e información dinámica (código intermedio).

1.2. Objetivos de la investigación

El principal objetivo de la investigación es definir un modelo de puntos de unión para modularizar aspectos de grano fino. Para el alcance de este objetivo se requiere realizar las siguientes acciones:

- Diseñar una solución computacional que permita identificar puntos de unión sobre variables locales y permitir el entrelazado de aspectos de baja granularidad.

- Definir un modelo de corte que identifique ámbitos de variables locales sin importar el nivel de profundidad de los bloques que las contienen. El modelo debe proveer un conjunto mínimo de primitivas.

- Mostrar la validez del nuevo modelo de puntos de unión mediante la aplicación de varios casos de aspectos de grano fino en la solución de problemas de diferentes dominios.

Lo anterior se justifica porque AspectJ y lenguajes similares no ofrecen un modelo con estas características a la vez que es importante por sus aplicaciones tales como prueba de software, comprensión y visualización de programas, manejo de aseveraciones, depuración, entre otras.

1.3. Solución propuesta

La identificación de variables locales dentro del código de un método, sin ambigüedad y principalmente dentro de cualquier ámbito anidado (estructuras de control dentro de estructuras de control a diferentes niveles de profundidad) requiere de la estructura del código fuente. Mediante el análisis del código a través de los correspondientes *árboles de sintaxis abstracta* (*abstract sintax tree: AST*) se puede definir una ruta que identifique de manera única a cualquier elemento deseado. De manera análoga, en el código intermedio cada instrucción con su dirección tiene correspondencia directa con los elementos del AST. De esta forma se puede establecer una relación entre ambas perspectivas del programa.

En esta tesis se presenta a ÉNFASIS, una herramienta diseñada específicamente para trabajar con aspectos de grano fino. ÉNFASIS define esencialmente:

- Un modelo computacional basado en la relación (mapeo) de la información recuperada de los AST y su contraparte en las instrucciones del código intermedio.
- Un modelo de puntos de unión específico para entrelazar aspectos de grano fino con los objetos del sistema.

Al igual que en los demás lenguajes de aspectos, ÉNFASIS introduce un mecanismo de *patrones de firmas* (*signature patterns*) que permite describir y localizar desde una variable local en particular hasta conjuntos de ellas. ÉNFASIS se basa en mecanismos de instrumentación de *bytecode* para lograr la implementación del mecanismo de entrelazado. Sus características se apoyan en el modelo computacional que lo define, hace uso de operaciones de conjuntos para identificar, seleccionar y manipular las variables locales requeridas por el proceso de entrelazado de aspectos de baja granularidad.

1.4. Contribuciones

La tesis provee varias contribuciones en el ámbito de aspectos de grano fino y sus aplicaciones, las cuales se describen a continuación.

1. *Un modelo de puntos de unión con soporte para el entrelazado al nivel de variables locales.*

   ÉNFASIS define un modelo que reduce algunas de las limitaciones de los demás lenguajes orientados a aspectos al permitir la modularización de aspectos de grano fino. El modelo computacional utiliza teoría de conjuntos para definir esquemas de entrelazados sofisticados, lo que permite aplicar aspectos de manera flexible sobre las aplicaciones. Aunque ÉNFASIS sólo soporta el corte para aspectos de grano fino, el modelo es lo suficientemente robusto para incorporar el soporte para aspectos de mayor granularidad, tal y como lo hacen otros lenguajes orientados a aspectos.

2. *La posibilidad de encapsular aspectos que son aplicables en áreas de verificación, prueba y validación.*

   Los aspectos de grano fino tienen la característica de indagar más allá de los límites de la interfaz de un objeto. Su funcionalidad reside en la interacción con elementos muy finos y específicos de un programa. Ésta es una propiedad natural que toma relevancia sobre los criterios de encapsulación tradicionales de objetos, principlamente cuando se trata de separar la funcionalidad que concierne a una propiedad que está definida en el interior de los objetos, no en el intercambio de mensajes entre dichos objetos. La encapsulación de aspectos de grano fino conlleva a la reutilización de partes o fracciones de objetos en otros objetos, en otras clases e incluso en otros sistemas.

3. *El procesamiento de anotaciones para variables locales como una alternativa a las especificaciones de corte.*

   Las anotaciones proveen un medio para especificar descriptores de archivos. El problema principal es la falta de soporte para su procesamiento a nivel de variables locales. Con la solución propuesta se tiene una alternativa para procesar este tipo de anotaciones aplicado para asuntos relacionados a monitoreo.

4. *Una herramienta de instrumentación de bytecode de alto nivel para el programador.*

ÉNFASIS fue diseñado con una arquitectura en capas. Cada capa puede utilizarse por separado, permitiendo mayor flexibilidad según las necesidades del programador. Se puede utilizar la capa de instrumentación sin necesidad de tener nociones de aspectos o se puede utilizar la capa de más alto nivel para aplicar incluso las mismas transformaciones mediante un enfoque con aspectos. Esto hace a ÉNFASIS bastante flexible para poder instrumentar clases de Java sin conocer los detalles de los *bytecodes*.

5. *Implementación del modelo de puntos de unión en una biblioteca de Java.*

   Aprender un nuevo lenguaje implica siempre una curva de aprendizaje que puede ser costosa. En este sentido, ÉNFASIS se provee como una biblioteca de Java para incorporar los conceptos de aspectos en el mismo lenguaje base. El programador solo tiene que definir las reglas de entrelazado requeridas para sus aspectos y crear nuevos objetos de la manera tradicional en que está acostumbrado. Este enfoque se ha utilizado anterioremente con otros lenguajes [62].

1.5. Organización de la tesis

El capítulo 2 ofrece una introducción a la programación orientada a aspectos en general y particularmente en el contexto de AspectJ. Se describen conceptos esenciales a partir de principios de diseño de la ingeniería del software, se describen brevemente las principales extensiones al modelo orientado a objetos que trataron previamente la problemática de los aspectos y se presenta una introducción al lenguaje AspectJ la cual muestra el alcance que tiene su modelo de puntos de unión.

En el capítulo 3 se presenta el trabajo relacionado con la granularidad fina. Se abordan extensiones basadas en AspectJ y las propuestas que incorporan nuevas primitivas de corte. Se revisan los modelos que ofrecen nuevos puntos de unión y mecanismos de refinación de avisos y los modelos que ofrecen diferentes enfoques en el modelo de puntos de unión.

El capítulo 4 introduce a ÉNFASIS, un framework que permite definir aspectos de baja granularidad. Se discuten los patrones de firmas que se proveen, las primitivas de corte y su

composición, así como el uso de avisos (*advice*) y la forma de definir aspectos. Particularmente, ÉNFASIS se provee como un conjunto de bibliotecas Java las cuales pueden utilizarse de manera independiente para instrumentar bytecode.

El modelo y los detalles más importantes de la implementación de Énfasis se discuten en el capítulo 5. Particularmente se ofrece información sobre la manera de identificar variables locales tanto en código fuente como en código intermedio. Se revisan brevemente las herramientas empleadas para este propósito y se discute el modelo formal de corte.

Finalmente el capítulo 6 presenta un análisis y discusión de la implementación de Énfasis, su modelo computacional y las principales contribuciones. También se mencionan las limitaciones y el trabajo futuro.

Capítulo 2

Programación orientada a aspectos

La Programación Orientada a Aspectos (POA) es una técnica de programación que permite separar y encapsular diferentes conceptos usados como criterios de descomposición para proveer una mejor modularidad en el software. Para poder comprender los conceptos en los que se basa este paradigma es necesario revisar las ideas detrás del *principio de separación de asuntos* (*separation of concerns principle*), el cual se considera uno de los principios fundamentales en la ingeniería del software.

2.1. Requerimientos y asuntos

Un *requerimiento* (*requirement*) es una propiedad del software necesaria por un usuario o conocida por un sistema para satisfacer un objetivo o documentación impuesta formalmente como contrato, estándar o especificación [53]. Existen tres tipos de requerimientos:

- Requerimientos funcionales los cuales expresan cómo se comporta el sistema.
- Requerimientos no funcionales, utilizados para expresar algunos atributos del sistema o atributos del ambiente del sistema. Éstos no están asociados al comportamiento, solo especifican otros aspectos del software como capacidad de uso, fiabilidad, desempeño y soporte [12, 53].

- Restricciones de diseño las cuales imponen límites sobre el diseño de los sistemas.

El objetivo de un sistema es satisfacer requerimientos o, en forma más general, asuntos. Un *asunto* es algo de interés para las personas involucradas en un proyecto de software, considerando cualquier tipo de requerimiento o incluso, más que un requerimiento, algún fragmento de código o algún concepto [15, 38].

Los requerimientos de una aplicación son exactamente los mismos, independientemente del paradigma utilizado. Los asuntos también son los mismos, sin embargo, de acuerdo al paradigma utilizado, éstos estarán implementados de diferente forma. Por ejemplo, el asunto de *registro* (*logging*) con un enfoque estructurado queda modelado mediante procedimientos o funciones, mientras que con un enfoque orientado a objetos se representa mediante objetos y clases.

Cada asunto es relevante para un problema particular, es decir, dicho asunto puede no tener sentido en otro problema o contexto. De forma más concreta, un asunto es una abstracción que es relevante para un problema dado [80].

2.1.1. Principio de separación de asuntos

En el desarrollo de software es deseable satisfacer todo tipo de requerimientos, incluyendo aquellos relacionados a la calidad como son robustez, adaptabilidad, mantenimiento y reutilización para que el sistema sea útil a los clientes o a los usuarios finales. Las abstracciones adecuadas que representan a cada *aspecto* dentro del dominio del problema deben separarse en forma adecuada para abatir la complejidad. Al separar cada aspecto es posible razonar de manera inteligente: estudiar a profundidad cada uno de ellos y de manera aislada, permitiendo claridad y consistencia en el *concepto* involucrado [19].

El *principio de separación de asuntos* (*SDA*), actualmente uno de los más importantes en la ingeniería del software, establece que un problema está compuesto por diferentes asuntos. Para la realización del software, cada asunto debe identificarse, separarse y posiblemente encapsularse [16]. Los beneficios asociados incluyen un mejor análisis y comprensión de los

sistemas, mayor nivel de abstracción, alta legibilidad en el código, alto nivel de reutilización, fácil adaptación y mantenimiento [1].

2.1.2. Asuntos de corte

El desarrollo de software involucra la implementación de requerimientos funcionales, los cuales proveen la *funcionalidad primaria* o central del sistema, y no funcionales para implementar los atributos del sistema o la *funcionalidad secundaria*. Por ejemplo, una aplicación de negocios tiene su funcionalidad central en la lógica del negocio, mientras que su funcionalidad secundaria puede estar formada por los atributos de autorización y persistencia.

Un atributo se comparte entre diferentes elementos de la funcionalidad primaria al momento de su implementación, lo cual genera que dicho atributo no pueda mantenerse aislado. Esta característica se conoce como un *asunto de corte* (*crosscutting concern*). La POA permite definir, programar y mantener separados los asuntos corte.

La Programación Orientada a Objetos (POO) y la Programación Estructurada (PE) son metodologías utilizadas que permiten trabajar esencialmente con asuntos primarios pero no con asuntos de corte, principalmente en aplicaciones complejas. Esto genera un fuerte acoplamiento entre ambos tipos de asuntos limitando la modificación en forma aislada en alguno de ellos.

2.2. Extensiones al modelo clásico de objetos

Diferentes problemas y complejidades relacionados al principio SDA fueron identificados observando diferentes problemas en el modelo clásico de objetos. La solución a cada problema permitió definir un conjunto de modelos conocidos en general como *tecnologías en separación de asuntos*. Las siguientes secciones describen en forma breve cada uno de los modelos propuestos.

2.2.1. Evolución funcional

Uno de los primeros trabajos donde se reporta la necesidad de contar con un adecuado soporte lingüístico para separar asuntos y facilitar modificaciones al vuelo sobre los objetos en ejecución, es en el área de sistemas de tiempo real. En estos sistemas se estudió el impacto que se tiene al introducir cambios que afectan el comportamiento del sistema mediante la clasificación de los nuevos servicios (características que modelan al sistema) en dos tipos, los nuevos servicios (extensiones funcionales) y los servicios que cambian (cambios funcionales) [37, 38].

En el caso de los nuevos servicios o extensiones funcionales, se consideran a los elementos existentes como *existiones* y a los elementos de extensión como *extensiones*. Mientras un existión se ejecuta, una extensión puede intervenir en puntos específicos. Cuando la extensión termina su ejecución, el control se regresa al existión continuando con su actividad. Más de una extensión pueden intervenir sobre un existión. La idea fue proveer a un objeto o servicio de extensión con una lista de *sondas*, elementos que especifican los lugares de intervención en los existiones. Cada sonda especifica un *punto de inserción* en el grafo del servicio extendido. Durante la interpretación de una *ruta de transición* (los lugares de intervención), un servicio existente permite contener sentencias de extensión para que sea intervenido. Una extensión puede ella misma tratarse como un existión y ser intervenida por otra extensión.

Para los servicios que cambian o cambios funcionales, un existión se puede transformar en uno nuevo. Las nuevas definiciones para un existión permiten *decrementar* o *incrementar* su comportamiento, estado, e incluso cambiar su referencia.

2.2.2. Programación orientada a sujetos

Para que la tecnología orientada a objetos pudiera ser exitosa no solo en el desarrollo de aplicaciones, sino en el desarrollo de múltiples conjuntos de aplicaciones, se observó que lo más importante no es el objeto mismo, sino la manera de cómo ese objeto se puede crear de acuerdo al dominio donde tendrá validez [32].

En el modelo clásico de objetos, una abstracción está caracterizada por un conjunto

de propiedades y comportamientos intrínsecos y esenciales que obedecen a las necesidades específicas de un dominio en particular. Esto genera un problema que impide la reutilización en otros dominios debido a que esa misma abstracción no cubre las necesidades de todos ellos, es decir, cada dominio requiere su propia *vista subjetiva*. Esto hace que los diseñadores tomen decisiones inadecuadas tratando toda propiedad y comportamiento como si fuesen intrínsecas a la naturaleza de los objetos.

La *programación orientada a sujetos* propone generar aplicaciones cooperativas que compartan los objetos y que contribuya cada una con la ejecución específica de sus operaciones para satisfacer las necesidades de cada dominio. Un *sujeto* queda definido como la colección de estados y comportamientos específicos que reflejan un todo particular. Los sujetos no son clases, ya que ellos introducen nuevas clases en el universo, sin embargo describen estado y comportamiento de objetos en muchas clases. Si dentro de un universo existen ciertos individuos o *sujetos* que definen vistas diferentes para una abstracción, entonces el universo debe ser capaz de permitir aplicaciones débilmente acopladas, introducir nuevas aplicaciones sin modificar las existentes, extender las aplicaciones existentes y en determinados casos permitir la recompilación sólo con propósitos de optimización y eficiencia. En todos los casos deben mantenerse los conceptos fundamentales del modelo de objetos: encapsulación, herencia y polimorfismo. Mediante *reglas de composición* se define cómo se construyen los sujetos y de qué forma va a ser la interacción de unos con otros.

2.2.3. Filtros de composición

El intercambio de mensajes entre los objetos de un sistema, permite establecer los mecanismos de colaboración necesarios para cubrir con la funcionalidad del sistema. El modelo de objetos define una semántica que sólo permite representar la *interacción* que ocurre entre dos objetos a la vez. Desde el punto de vista de sistemas concurrentes, se requiere de construcciones que permitan modelar la interacción entre dos o más objetos a la vez y al mismo tiempo poder reutilizar dichos esquemas de interacción para facilitar la evolución de sistemas distribuidos concurrentes [2]. Por otro lado, el mecanismo de herencia permite definir rela-

ciones entre clases para la construcción de objetos en términos de comportamiento, pero no en términos de interacciones entre objetos. Esto trae como consecuencia que la concurrencia se implemente en partes pequeñas y específicas como parte del comportamiento de los objetos, limitando la capacidad de reutilización [9].

El *modelo de filtros de composición* (de comportamiento) es una extensión del modelo de objetos mediante una modificación a la interfaz tradicional de un objeto agregando objetos adicionales conocidos como *filtros* que permiten seleccionar y manipular las peticiones de objetos. Esta interfaz extendida tiene *filtros de entrada* y *filtros de salida* que aplican mecanismos de reflexividad para manipular los mensajes tanto de entrada como de salida. Estos filtros tienen una semántica que los hace independientes unos de otros, facilitando la selección de mensajes invocando a otros filtros para obtener un nuevo comportamiento [7].

La interfaz extendida de este modelo permite la especificación de cómo pueden operar los filtros sobre los mensajes de otros filtros. Esta característica se conoce como *superimposición* y permite alterar el proceso de instanciación de los filtros para cubrir necesidades específicas. La *superimposición* también evita la dispersión de código entre varios objetos [8].

2.2.4. Software orientado a objetos adaptables

Cualquier comportamiento necesario para la realización de un sistema se hace mediante la implementación de métodos que se codifican en alguna clase. Aunque este enfoque permite la creación de objetos para cumplir con la funcionalidad del sistema, la estructura de la clase queda codificada con detalles que no son relevantes para otros programas, generando redundancia y limitaciones en la reutilización de dichos programas. Esto se hace más evidente cuando muchos métodos, principalmente los que son de pocas líneas de código, deben definirse en varias clases porque están relacionadas entre ellas para proveer la funcionalidad requerida.

El enfoque de *objetos adaptables* lleva el desarrollo de software orientado a objetos a un nivel más alto de abstracción (generalización) considerando familias de programas orientados a objetos [55]. La idea parte del hecho de que resolver problemas generales es más fácil que la resolución de problemas específicos. El *software adaptable* se ajusta automáticamente al

cambio de contexto, considerando bajo acoplamiento con fragmentos de código cooperativos, los cuales describen solo los asuntos de un contexto.

El *software orientado a objetos adaptables* considera la estructura de las clases definidas sólo de manera parcial. Estas definiciones parciales deben estar acompañadas de un número de *restricciones* que permiten o satisfacen la personalización de la clase, e incluso, permiten satisfacer una gran cantidad de estructuras de clases. De la misma manera, el comportamiento no se implementa por completo, solo se especifica cuando se necesita. La definición completa de la clase se complementa con el código resultante del contexto específico del dominio del problema. Para crear la definición completa de la clase, se utilizan las restricciones que definen los puntos en donde los diferentes fragmentos de programas se combinarán. A las restricciones se les conoce como *patrones de propagación* debido a que especifican en qué clases deben propagar (insertar) un fragmento de código. El ensamblaje de código se lleva a cabo mediante *envolventes* (*wrappers*) con relaciones de tipo *antes* (*prefix*) o *después* (*suffix*) para indicar en qué parte del código de la clase, respecto a un punto en particular, se agregará el código específico del contexto.

2.2.5. Hiperespacios y separación multidimensional de asuntos

En este enfoque se precisa que para satisfacer los objetivos de la ingeniería de software se depende esencialmente de la habilidad para mantener separados todos los asuntos de importancia en los sistemas de software. Las técnicas modernas de software soportan de alguna forma la separación de asuntos, pero de forma restringida en una sola dimensión de descomposición y composición, por ejemplo los objetos. A esto se le conoce como la *tiranía de la descomposición dominante* [79].

La *separación multidimensional de asuntos* (*multidimensional separation of concerns: MSOC*) es un modelo que permite modelar los conceptos de importancia dentro de un domino. Estos conceptos incluyen objetos, funcionalidad y propiedades, así como conceptos derivados del dominio del problema como estructuras de datos. También se consideran diferentes recursos y artefactos como especificaciones de requerimientos, diseños y código. El

modelo MSOC define el concepto de *hipercorte* (*hyperslice*) el cual tiene por objetivo encapsular un asunto codificado en algún formalismo: concepto, función, artefacto o característica [64].

Mediante reglas de composición, los hipercortes se sintetizan y se integran en un *hipermódulo* (*hypermodule*) el cual puede convertirse en un nuevo hipercorte. Como este modelo es la evolución de la *programación orientada a sujetos*, cada sujeto es un hipercorte y las reglas de composición son las mismas que permiten definir a nuevos individuos. Finalmente, el *hiperespacio* (*hyperspace*) es un *espacio de asuntos* donde se puede realizar la separación multidimensional [63].

2.2.6. Programación orientada a la variación

El comportamiento de un objeto depende de todos los comportamientos que están definidos en cada nivel de la jerarquía de herencia a la que pertenece. Entre más especializada es la clase, mayor será el comportamiento que puede exhibir el objeto. Esto se conoce como *variaciones dependientes del tipo* (clase), es decir, todas las variaciones de comportamiento que puede tener un objeto (por la jerarquía de herencia) se conocen hasta la creación del mismo. Sin embargo, existen otras fuentes de variaciones de comportamiento como el estado interno de un sistema o las diferentes características del ambiente en donde opera el objeto. Estas variaciones se conocen como *variaciones dependientes del contexto* las cuales no dependen de una jerarquía de herencia. Esto genera problemas de extensión, ya que no es posible establecer variaciones una vez que le objeto se ha instanciado, quedando ligado a un determinado contexto [60].

El *modelo de variación* soporta la separación de la descripción de un comportamiento base y sus variaciones dependientes del contexto a nivel semántico. La propuesta mantiene el comportamiento intrínseco de un objeto como comportamiento predeterminado y las variaciones de comportamiento como dependientes del contexto. Las clases y los *ajustes* (*variaciones*) se consideran como repositorios de comportamiento y mediante *administradores* se establecen las relaciones entre ellos. La composición se realiza mediante *combinadores* los cuales

se responsabilizan de conectar definiciones desde diferentes módulos para producir comportamientos completos. La variación o cambio de comportamiento se hace tras la identificación de un evento que informa a los administradores [59].

2.3. Introducción a la POA

La POA es el resultado de varias líneas de investigación que han abordado de diferente forma el principio SDA para lograr mejores niveles de modularidad en el software. La tecnología orientada a aspectos se apoya al igual que otros modelos, en principios sólidos de la ingeniería del software como abstracción, encapsulación, herencia, polimorfismo, modularidad, interfaces y contratos. A doce años de la aparición de este modelo como nuevo paradigma, actualmente la *tecnología orientada a aspectos* se encuentra con la suficiente madurez para su aplicación en el desarrollo de software a nivel industrial [11, 28, 39, 75], pero al mismo tiempo aún con muchos retos.

Las tecnologías emergentes han permitido formalizar el área de *Desarrollo de Software Orientado a Aspectos*, *DSOA* (*Aspect-Oriented Software Development: AOSD*). La *Ingeniería del Software Orientada a Aspectos* o *ISOA* (*Aspect-Oriented Software Engineering: AOSE*) está emergiendo para administrar el proceso de desarrollo de los sistemas con este paradigma [24, 26].

2.3.1. Antecedentes

El modelo de la POA surge tras observar que la programación orientada a objetos no puede capturar todas las *decisiones de diseño* importantes que un programa debe implementar. Este problema no es exclusivo de los objetos, también la programación estructurada presenta el mismo tipo de inconvenientes. El problema fundamental radica en que ciertas *propiedades* de un sistema interfieren con otras generando programas con código perteneciente a diferentes funcionalidades, dificultando la reutilización de cada una de ellas. Finalmente se distingue que el código resultante con estas problemáticas conduce a la generación de dos tipos de

sistemas, aquellos que son fáciles de comprender pero al mismo tiempo son ineficientes, y por el contrario, los sistemas que son eficientes pero difíciles de comprender [47].

Al no poder expresar adecuadamente todas las decisiones de diseño, cuando dos o más propiedades se implementan, se tiene que lidiar con la *interferencia* que causan en la funcionalidad del sistema, lo que lleva a desarrollar una esquema de coordinación (el cual generalmente es ineficiente) para lograr la funcionalidad requerida. Cuando ocurre esta situación se dice que una propiedad *corta* a la otra. Formalmente, el término para describir el *corte entre propiedades* se conoce como *corte de funcionalidad* o *crosscutting*. Desde la perspectiva del principio SDA, el *crosscutting* corresponde con los *asuntos secundarios o de corte* (*crosscutting concerns*), aquellos que están compartidos entre los asuntos primarios.

El *crosscutting* o simplemente corte (de aquí en adelante) genera una serie de problemáticas asociadas a la falta de claridad en el código [47, 51], distinguiéndose principalmente:

1. Problemas de *código disperso* (*code scattering*) los cuales se generan al momento de implementar dos propiedades que se interfieren o cortan una a la otra. La implementación de estas propiedades queda fragmentada o en partes a lo largo de diferentes módulos del sistema.

2. Problemas de *código repetido* los cuales aparecen como una consecuencia del código disperso ya que los bloques de código tienen que repetirse en cada parte del sistema donde tiene efecto el corte de la propiedad en cuestión.

3. Problemas de *código invasivo* (*invasiveness*) que surgen cuando el software tiene nuevos requerimientos o nuevas propiedades y éstos tienen relación con las propiedades de corte, por lo que deberán incorporar dicha funcionalidad para garantizar la consistencia del sistema.

4. Problemas de *código mezclado* (*code tangling*) que ocurren cuando dos o más propiedades del sistema tienen relación y debido al efecto de dispersión, el código se comparte o se *mezcla* entre las propiedades involucradas dificultando su comprensión.

5. Problemas de mantenimiento que se presentan de manera natural por la aparición de errores o cambio en los requerimientos del sistema. El mantenimiento que se le tiene que dar a cada una de las propiedades de corte, o a aquellas que están interferidas por un corte, generan una actividad considerablemente mayor de mantenimiento debido a la dispersión. Cada cambio debe reflejarse en todos los lugares donde aplica el corte para garantizar uniformidad. Sin embargo el problema puede agravarse aún más porque al mismo tiempo este proceso es fuente de introducción de nuevos errores.

Desde la perspectiva de la POA, las propiedades de corte se conocen como *aspectos* y para una mejor comprensión del sistema, éste se tiene que descomponer tanto en componentes (objetos, procedimientos o funciones) como en aspectos [47]. La POA ofrece mecanismos adecuados para abstraer, componer y reutilizar aspectos. Estos conceptos y los mecanismos asociados se presentan en las siguientes secciones.

2.3.2. Descomposición y composición orientada a aspectos

La POA permite la implementación de determinados conceptos relevantes para la realización de un sistema. La metodología de la POA para el desarrollo de sistemas orientados a aspectos se realiza de forma similar a las otras metodologías. Se identifican tres etapas importantes [18, 51]:

- **Descomposición de aspectos.** A partir de los requerimientos, las actividades de análisis y diseño se enfocan en identificar y especificar aspectos y componentes involucrados. Los componentes se identifican de acuerdo al proceso de descomposición funcional utilizado (objetos o procedimientos) y los aspectos se identifican a partir de aquellos requerimientos o conceptos que se expanden o afectan a otros módulos. Se ha propuesto esencialmente identificar el impacto del aspecto, la resolución de conflictos y el mapeo adecuado bajo la influencia de cada criterio o concepto utilizado como elemento de descomposición [73].

- **Implementación de asuntos.** La programación de componentes se realiza de forma

tradicional de acuerdo al paradigma utilizado y la programación de aspectos se realiza de forma independiente a otros aspectos.

- **Recomposición de aspectos.** La creación completa del sistema comprende la integración de componentes y aspectos, así como la interacción entre los mismos aspectos para proveer la funcionalidad requerida. Cada aspecto debe tener *reglas de recomposición o entrelazado* (*weaving rules*) para poder producir el sistema final. Los componentes y aspectos se compilan con una entidad conocida como *entrelazador* (*weaver*), el cual utiliza las reglas de recomposición para habilitar a los componentes compilados y permitir la interacción con los aspectos.

2.3.3. Análisis y diseño orientado a aspectos

En el terreno de la descomposición, se han desarrollado planteamientos que permiten aproximarse hacia el *análisis y diseño orientado a aspectos* (*ADOA*). A partir de la riqueza conceptual del modelo de Programación Orientada a Sujetos (sección 2.2.2) se propuso el enfoque de *temas* para observar las relaciones entre comportamientos y aproximarse hacia el ADOA [5, 15]. Por otro lado, a partir de diferentes enfoques de ingeniería en la resolución de problemas se propusieron planteamientos para considerar *arquitecturas orientadas a aspectos* [80]. Y en el contexto de la modelación con UML, Jacobson identificó que los casos de uso capturan adecuadamente aspectos y su estructura de corte [38].

2.4. Terminología

Debido a la riqueza conceptual de las diferentes líneas de investigación, es necesario precisar el significado de algunos conceptos básicos para establecer la terminología adecuada.

2.4.1. Definición de aspecto

En la página 2 se definió un aspecto como un tipo particular de asunto y al mismo tiempo se mencionó que un asunto es cualquier código relacionado a un objetivo, característica o tipo de funcionalidad [15]. Además, frecuentemente se asocia un requerimiento con un asunto [51], sin embargo como se vió en la sección 2.1 son dos cosas diferentes.

Kiczales define un aspecto como una propiedad que afecta el desempeño o la semántica de los componentes en forma sistémica [47]. Considerando los modelos que extienden el modelo de objetos (sección 2.2), Krysztof Czarnecki amplía la definición de aspecto mediante el concepto de *dominio*. Un *dominio* es un área de conocimiento caracterizada por un conjunto de conceptos, un asunto es un dominio utilizado como criterio de descomposición y un aspecto es una especificación parcial de un concepto o conjunto de conceptos respecto a un asunto [18].

Ésta última definición clarifica el alcance que tiene un asunto de corte desde cualquier perspectiva de las diferentes tecnologías en separación de asuntos, incluyendo a la POA. De esta manera, le definición que se sugiere para aspecto es la siguiente:

> **Aspecto.** *Un criterio utilizado para especificar parcialmente un conjunto de conceptos asociados a un área de conocimiento.*

Esta definición permite identificar tres elementos esenciales:

1. El *criterio* (abstracción) permite distinguir una característica esencial de otra y señalar la diferencia que hay entre ellas. Para poder aplicar un criterio, se necesita de un *proceso de abstracción*. El criterio también aplica para poder distinguir entre *conjuntos* de características esenciales.

2. La *especificación* como una forma de determinar o precisar la individualidad de algo. Se distingue la *parcialidad* como relativa a un todo o que es partícipe de algo.

3. El *concepto* que permite una descripción en términos de resolución.

Cualquier aspecto tendrá inherentemente asociado un *mecanismo de interacción* con al menos un componente que le permita *activar* su funcionalidad, ya sea de manera individual

o en combinación con la funcionalidad del componente, para proveer una funcionalidad que satisfaga un requerimiento. Por lo tanto este esquema de interacción define la manera de cómo proveer funcionalidad en diferentes situaciones representadas por diferentes componentes.

2.4.2. Definición de POA

Gregor Kiczales [47] define a la POA como una técnica de programación que provee al programador de mecanismos adecuados para separar componentes y aspectos, unos de otros, permitiendo su abstracción y composición para producir todo el sistema.

Con la *orientación a aspectos* se cuenta con un mecanismo para componer *comportamiento de corte* sobre operaciones y clases de un sistema permitiendo liberar al código fuente de cualquier propiedad de corte [38] reflejando una mejor y mayor aplicación del principio SDA. El mecanismo de composición lo realiza una entidad llamada *entrelazador* o *tejedor* (*weaver*), el cual es una entidad similar a un compilador que permite generar el sistema completo. Este proceso de composición o *entrelazado* (*weaving*) puede llevarse a cabo en tiempo de compilación, carga o ejecución.

Debido a las características que presenta este modelo, la POA ha tenido una relevancia importante en la industria. La POA no substituye a las metodologías de objetos y procedimientos, en su lugar, las complementa facilitando la aplicación del principio SDA [51]. Actualmente la POA se considera una metodología completa y uno de los primeros investigadores en favor de este hecho fue Karl J. Lieberherr [56] quien declara que "*el modelo de objetos adaptables es una caso especial de POA, ... ambos enfoques parten al programa en comportamiento básico y en varias descripciones de aspectos, ... por lo que la POA es una generalización de la programación adaptable*".

Pero para poder definir con mayor precisión qué es la POA, es necesario identificar cuáles son sus elementos esenciales. Filman sugiere que la POA requiere de dos propiedades básicas [27]:

1. *Cuantificación* (*quantification*). Especificar la ejecución de sentencias que tienen efectos en varios lugares. Por ejemplo, la herencia es una forma (limitada) de cuantificación

porque el comportamiento de una superclase tendrá efecto en algunas de sus subclases.

2. *Inconsciencia* (*obliviousness*). Liberar al programador de las especificaciones puntuales donde una sentencia o conjunto de sentencias del programa tendrá efecto. Retomando el ejemplo de herencia, ésta obliga a un programador a saber que debe ser *consciente* de hacer algo en las subclases. Por ejemplo, la redefinición de métodos que es un proceso puntual y local en cada subclase involucrada.

En este sentido, la POA se caracteriza por *cuantificar* de manera *inconsciente*, es decir, el programador solo especifica lo que quiere hacer sin que tenga que puntualizar dónde debe tener efecto el comportamiento del aspecto.

Al respecto, Filman dice: "*En general, la POA se puede entender como el deseo de hacer sentencias cuantificadas sobre el comportamiento de los programas y tener estas cuantificaciones en programas escritos por programadores no conscientes*". Y agrega: "*Este proceso rompe por completo las demandas de unidad (efecto en algún lugar del programa) y localidad (una sentencia que debe ejecutarse dentro de la proximidad de otra), es decir, se puede organizar un programa en la forma más apropiada para codificar y mantener. Ya no son necesarias las marcas locales de cooperación. El mecanismo de entrelazado del sistema POA puede, por sí mismo, tomar las sentencias cuantificadas y el programa base para producir las instrucciones primitivas a desempeñar*".

La cuantificación es el mecanismo que permite establecer la interacción entre componentes y aspectos permitiendo liberar al programador de tener que especificar cómo construir el sistema completo en cada una de sus partes. Por consiguiente, se sugiere la siguiente definición para la POA:

> **Programación orientada a aspectos.** *Método de implementación en el que los programas se organizan en componentes, aspectos y especificaciones de cuantificación para permitir la realización del sistema.*

Hay cuatro puntos importantes en la definición:

1. La construcción del sistema utiliza componentes y aspectos.

2. Los componentes y aspectos establecen un esquema de *coordinación* entre ellos y un *grado de interacción* mediante la cuantificación.

3. Los componentes (objetos, procedimientos o funciones) definen especificaciones completas y conscientes.

4. Los aspectos y sus cuantificaciones definen las especificaciones parciales e inconscientes.

2.4.3. Definición de lenguaje orientado a aspectos

La mayoría de los lenguajes orientados a aspectos se han reportado solamente como extensiones de algún otro lenguaje, orientado a objetos o estructurado, con el soporte para implementar los conceptos de aspectos [4, 11, 45]. Esto es coherente con los diferentes modelos que extienden el modelo clásico de objetos. Otros lenguajes han podido unificar clases y aspectos [72] o incluso combinar componentes, aspectos y paquetes en una sola unidad coherente [61]. Considerando el estado del arte, se sugiere la siguiente definición:

> **Lenguaje orientado a aspectos.** *Es un lenguaje de programación que provee los mecanismos adecuados de cuantificación para explícitamente capturar la estructura de corte requerida por los aspectos, encapsular el criterio empleado y proveer la funcionalidad requerida de acuerdo a los conceptos involucrados.*

Algunos lenguajes han incorporado la orientación a aspectos en el mismo lenguaje de componentes [62] donde en vez de proveer una extensión al lenguaje se provee una biblioteca con todas las bondades descritas. Este es el enfoque que se sigue en la presente tesis.

2.4.4. Definición de granularidad

Un modelo de puntos de unión define su capacidad de corte en base a los tipos de puntos de unión que puede identificar. El lenguaje AspectJ identifica puntos de unión para campos, métodos y constructores, los cuales pueden especificarse de manera puntual o mediante un patrón que los pueda cuantificar. Estos puntos de unión son adecuados para trabajar a nivel

de las interfaces que definen tanto clases como objetos, por lo que su alcance o *granularidad* abarca solamente mensajes y operaciones. La definición de granularidad[1] es la siguiente:

> **Granularidad.** *Definición de la unidad, parte o porción que puede identificarse o considerarse para un propósito.*

En el contexto de AspectJ, la granularidad y expresividad que caracteriza a su modelo de puntos de unión no es adecuada para todo tipo de dominios, principalmente para aquellos en donde se requiere identificar puntos de unión que no tienen relación con mensajes y operaciones.

2.4.5. Definición de corte

Considerando la definiciones propuestas de aspecto (sección 2.4.1) y de POA (sección 2.4.2), la definición sugerente de corte es la siguiente:

> **Corte.** *Especificación que permite cuantificar los puntos de unión para la interacción entre aspectos y componentes.*

De esta definición se desprende directamente que:

- La *cuantificación* es, al mismo tiempo, la regla de composición (entrelazado) para producir el sistema completo.

2.5. Lenguaje AspectJ

En un inicio, los lenguajes orientados a aspectos fueron implementaciones específicas de algún dominio en particular. Tras la aplicación de diversas técnicas y enfoques, tales como flujos de datos o invocaciones de métodos en tiempo de ejecución, se observó que lo más importante para el diseño de un buen lenguaje de aspectos era el concepto de *puntos de unión*, los cuales son los elementos de la semántica de un lenguaje de componentes con los que interactúan los aspectos [47].

[1] Diccionarios de la Real Academia Española y Merriam-Webster.

El primer lenguaje orientado a aspectos de propósito general fue *AspectJ* [58], el cual es una extensión del lenguaje Java para trabajar con la orientación a aspectos. AspectJ fue diseñado con un *modelo de puntos de unión* basado originalmente en las invocaciones a métodos. Este modelo mostró su efectividad para encapsular aspectos y especificaciones de corte lo cual hizo que otros lenguajes[2] y plataformas[3] de desarrollo siguieran la misma filosofía.

El lenguaje AspectJ permite encapsular asuntos de corte junto con las reglas de entrelazado necesarias para coordinarse con los objetos del sistema. De esta manera se pueden especificar *qué* acciones se van a desarrollar *cuando* se identifiquen ciertos puntos de unión.

2.5.1. Tipos de corte

La implementación de la reglas de entrelazado definen el corte sobre diferentes módulos del sistema para poder modularizar los asuntos de corte. AspectJ extiende a Java con soporte para dos *tipos de corte* (*crosscutting*). El *corte dinámico* permite definir implementaciones adicionales que se ejecutan en ciertos puntos bien definidos en la ejecución del programa. El *corte estático* hace posible definir nuevas operaciones y atributos sobre tipos existentes.

- **Corte dinámico.** El corte dinámico permite reemplazar o agregar nuevo comportamiento a la ejecución de un programa. Este tipo de corte es el más utilizado en AspectJ y se basa en la *expresividad* del modelo de puntos de unión. Como ejemplo, considérese la especificación de realizar cierta acción antes de la ejecución de algún método; basta con especificar los puntos de entrelazado y la acción a realizar al alcanzar dichos puntos.

- **Corte estático.** El corte estático permite hacer modificaciones a la estructura de las clases, interfaces y aspectos mismos y no afectan la ejecución de la aplicación. Puede utilizarse para declarar notificaciones y errores visibles en tiempo de compilación. En AspectJ, este tipo de corte se utiliza para facilitar la implementación del corte dinámico.

[2] Algunos de estos lenguajes son *AspectWerkz* [28] (actualmente fusionado con AspectJ) y *CaesarJ* [61].

[3] Dos de los frameworks más sobresalientes son *JBoss-AOP* [39] y *Spring* [75].

2.5.2. Elementos de corte dinámico

Los elementos que intervienen en el corte dinámico son tres: *puntos de unión*, *corte en puntos* y *avisos*.

Punto de unión

Un *punto de unión* (*join point*) es un punto identificable en la ejecución de un programa. Su función es definir los lugares en donde se entrelazará el código de los aspectos.

Corte en puntos

Un *corte en puntos* (*pointcut*) o simplemente *corte*, es una construcción del programa que selecciona puntos de unión y opcionalmente toma los valores de la ejecución del contexto de esos puntos de unión. Un corte define una colección de puntos de unión y al mismo tiempo especifica la interacción o reglas de entrelazado.

Aviso

Un *aviso* (*advice*) es el código que se ejecutará antes o después del punto de unión que ha sido seleccionado por un corte. También puede modificar, reemplazar u omitir la ejecución del código que corresponde a dicho punto de unión. El aviso encapsula la lógica que se ejecutará al alcanzar un punto de unión.

2.5.3. Elementos de corte estático

Los elementos del corte estático son *introducciones* y *declaraciones a tiempo de compilación*.

Introducciones

Una *introducción* (*introduction*), como su nombre lo dice, permite introducir cambios en las clases, interfaces y aspectos del sistema, los cuales no afectan el comportamiento. Por

ejemplo, se puede hacer que una nueva interfaz sea implementada por una clase existente en el sistema. A las introducciones también se les conoce como *declaraciones entre-tipos* (*inter-type declarations*).

Declaraciones a tiempo de compilación

Estas declaraciones (*compile-time declarations*) permiten agregar notificaciones y errores detectables a tiempo de compilación. Por ejemplo, para notificar al programador que un método se ha depreciado y que debe utilizar una nueva versión u otro método.

2.5.4. Modelo de puntos de unión

El *modelo de puntos de unión* (*join point model*) es un elemento importante en el diseño de cualquier mecanismo de un lenguaje orientado a aspectos. Este modelo provee un marco de referencia para hacer posible la ejecución de los programas de aspectos y componentes en una forma coordinada [45]. El modelo de puntos de unión se explica en base al modelo ofrecido por el lenguaje AspectJ, el cual se ha considerado como referencia para su implementación en otros lenguajes.

Para ilustrar los conceptos del modelo de puntos de unión se usará un ejemplo de algoritmos para ordenación y búsqueda de datos. La figura 2.1 muestra un diagrama UML para un fragmento del sistema. Cada uno de los métodos de ordenación es una subclase de la clase abstracta `Sorting` y dependiendo de las condiciones de número de datos y recursos de la computadora, el sistema ofrece el mejor algoritmo para llevar a cabo el proceso de ordenación. Por ejemplo, las partes principales del código de *QuickSort*[4] está codificado como sigue.

```
class Quick extends Sorting {
  public int[] sort(int[] array) {
    sort(array, 0, array.length - 1);
```

[4]En el apéndice B.1 se muestra el código completo.

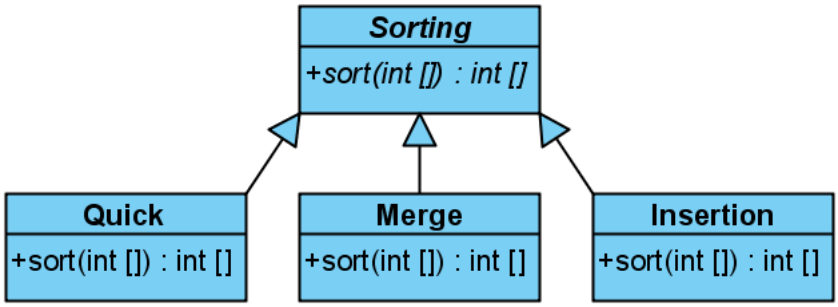

Figura 2.1: Descripción UML de una parte de un sistema de algoritmos de ordenación y búsqueda.

```
    return array;
  }
  private void sort(int[] a, int lo0, int hi0) {
    //código de ordenación
  }
}
```

En este código se distinguen los siguientes elementos: la declaración de una clase llamada `Quick`, la cual hereda de la clase abstracta `Sorting`. Un método público `sort`, el cual toma como argumento un arreglo de tipo `int` y devuelve un arreglo de tipo `int`. Un método privado llamado `sort` que recibe un arreglo de tipo `int`, dos argumentos más de tipo `int` y no devuelve valor alguno. Cada uno de estos elementos son parte del modelo de puntos de unión de AspectJ y permiten definir los lugares de entrelazado.

Considérese la ejecución de las siguientes sentencias en alguna parte de la aplicación, por ejemplo en un método `main()`.

```
01 Sorting q = new Quick();
02 int[] a = new int[] {6,1,5,2,8,2,9,3,7,6,8,9,1,4,2,5,2};
03 int[] ordenado = q.sort(a);
```

La línea 3 desencadena una serie de invocaciones que se describen a continuación.

1. Se realiza una llamada al método `sort(int[])`.
2. Se ejecuta el método `sort(int[])`.
3. Se llama al método privado `sort(int[], int, int)`.
4. Se ejecuta el método privado `sort(int[], int, int)`. Dentro de éste método privado se realizan una serie de invocaciones y ejecuciones recursivas hasta que el arreglo está ordenado.
5. Al terminar la ejecución de la primera invocación al método privado, el control se devuelve al método público `sort(int[])`, terminando la llamada (en el punto 1).

El modelo considera a los puntos de unión como los nodos de un grafo de llamadas del objeto en ejecución y las aristas formadas entre los nodos representan flujos de control. En este modelo, el control pasa en cada punto dos veces, una cuando se inicia un procesamiento ligado a ese punto y otra cuando termina. Concretamente, en esta descripción se identifican los puntos de unión para la llamada a los métodos y los puntos de unión para la ejecución de los métodos. Por ejemplo, la sentencia

```
03 int[] ordenado = q.sort(a);
```

especifica la llamada al método, mientras que las sentencias que conforman el método corresponden a su ejecución.

```
public int[] sort(int[] array) {
  /* sentencias */
  /* de ejecución */
}
```

En la figura 2.2 se muestra la secuencia de invocaciones entre los métodos `sort()`. Del lado izquierdo se encuentra la definición que recibe el arreglo completo a ordenar y del lado derecho el método que recibe el subarreglo a analizar con sus respectivos índices.

La siguiente sección describe algunos de los tipos de puntos de unión que ofrece AspectJ.

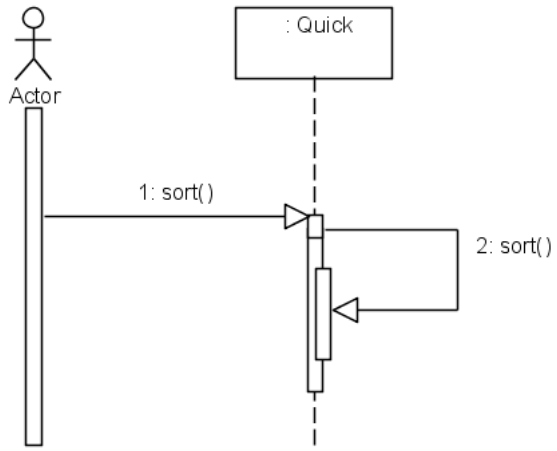

Figura 2.2: Diagrama de secuencia para las invocaciones de los métodos `sort()` de la clase `Quick`.

2.5.5. Categorías de puntos de unión

AspectJ define varias categorías de puntos de unión, las cuales se describen a continuación.

Puntos de unión para métodos

Hay dos tipos de puntos de unión para métodos, la *ejecución* (*execution*) y la *llamada* (*call*) de un método. El punto de unión para la ejecución de un método se refiere a todo el cuerpo del método. En base al ejemplo de la clase `Quick`, el siguiente fragmento ilustra este punto de unión.

```
public int[] sort(int[] array) {
  sort(array, 0, array.length - 1);   | punto de unión para la ejecución
  return array;                       | de sort(int[])
}
```

Un punto de unión para la llamada de un método solo se refiere al lugar, dentro de otro método, donde se manda llamar o se invoca al primero. Por ejemplo, la siguiente línea solo se refiere a la llamada del método sort() con tres argumentos:

```
public int[] sort(int[] array) {
  sort(array, 0, array.length - 1);
  return array;
}
```

punto de unión para la ejecución de `sort(int[],int,int)`

Puntos de unión para constructores

De manera análoga, un punto de unión para la ejecución de un constructor es todo el cuerpo que lo define, mientras que un punto de unión para la llamada es la instanciación de un objeto:

```
Quick quick = new Quick();
```

punto de unión para la llamada de `Quick()`

Puntos de unión para acceso a campos

Este tipo de puntos de unión permiten identificar el instante en que se asigna (escritura) un valor a un campo o se recupera (lectura) el valor que tiene almacenado.

```
int i;
...
i = (int)Math.random()*15;
...
a = i;
```

punto de unión para escritura de `i`

punto de unión para lectura de `i`

Otras categorías de puntos de unión

AspectJ define otras categorías de puntos de unión, las cuales se resumen en la siguiente lista[5]:

- Manejadores de excepciones (sentencias `catch`).

[5]Para una lista detallada de los tipos de puntos de unión disponibles en AspectJ se puede consultar la documentación en línea, disponible en: http://www.eclipse.org/aspectj/.

- Inicialización: objetos y clases.

- Preinicialización de objetos (llamadas a `super` en el proceso de construcción).

2.5.6. Cortes

Un *corte* (*pointcut*)[6] es un conjunto de puntos de unión, más opcionalmente algunos de los valores en el contexto de su ejecución. En otras palabras, el corte captura o identifica los puntos de unión en el flujo del programa. Un *designador* (*pointcut designator*) identifica el corte mediante nombres o expresiones, los cuales a su vez identifican puntos de unión a tiempo de ejecución. Ambos términos, *designador de corte* y *corte*, suelen usarse indistintamente.

Patrones de firmas

La identificación de puntos de unión se consigue mediante una serie de *patrones de firmas* (*signature patterns*) que describen colectivamente clases, interfaces, miembros de instancia o clase y tipos primitivos. Estos patrones permiten especificar los puntos de unión sobre los que se quiere aplicar acciones de corte. Los patrones se complementan con caracteres comodines (`*`, `?`, `+` y `..` -dos puntos-) que permiten denotar conjuntos más grandes de puntos de unión. El cuadro 2.1 contiene varios ejemplos que muestran cómo se definen diferentes patrones.

Como se puede apreciar, los patrones de firmas son un mecanismo flexible para describir el corte en puntos de campos, métodos, constructores o cualquier otro elemento de interés. En AspectJ es posible la especificación de conjuntos de puntos de unión más robustos mediante operadores de composición como `&&` (*and*), `||` (*or*) y `!` (*not*). El último patrón de la tabla 2.1 muestra el uso del operador `not`. Otros ejemplos se muestran en la tabla 2.2.

[6] El término *pointcut* puede traducirse como *puntos de corte* (en plural). Sin embargo, debido a la cognotación de la POA, un corte se realiza *sobre puntos* de algún elemento de la ejecución del programa, dando lugar a una traducción más apropiada como *corte en puntos* o simplemente *corte*.

Patrón	**Identificación**
`Quick`	Tipo de nombre `Quick`.
`Sort*`	Cualquier tipo que empiece con `Sort`.
`java.*.Date`	Tipo `Date` en cualquier paquete que tenga definido un espacio de nombre `java`, tal como `java.util.Date` y `java.sql.Date`.
`private void Quick.sor*(..)`	Cualquier método de la clase `Quick` que empiece con `sor` y sin importar los argumentos, que no devuelva valor alguno y sea privado.
`public * Sorting+.*(*)`	Cualquier método de la clase `Sorting` o de sus subclases con un solo argumento y sin importar el valor devuelto pero que sean públicos.
`public Punto.new()`	Constructor público sin argumentos de la clase `Punto`.
`public !final *.*`	Cualquier campo de cualquier clase que sea público pero que no esté declarado como `final`.

Tabla 2.1: Ejemplos de patrones de firmas.

Designadores de corte

Las diferentes categorías de puntos de unión listadas en la sección anterior se pueden identificar mediante los siguientes designadores:

- `call`: llamada de métodos y constructores.
- `execution`: ejecución de métodos y constructores.
- `get`: acceso de lectura sobre los campos.

Patrón	**Identificación**
`Vector || Hashtable`	Tipo `Vector` o `Hashtable`.
`java.util.RandomAccess+ && java.util.List+`	Todos los tipos que implementan las interfaces especificadas.

Tabla 2.2: Ejemplos de composición de cortes.

- `set`: acceso de escritura (asignación) sobre los campos.
- `handler`: manejadores de excepciones (sentencias `catch`).
- `initialization`: inicialización de objetos.
- `staticinitialization`: inicialización de clases.

Como ejemplo, los puntos de unión para los métodos `sort()` de la clase `Quick` se pueden identificar de diferentes formas:

- `execution(public int[] Quick.sort(int[]))`. Identificación puntual de ejecución.
- `call(public void Quick.sort(int[], int, int))`. Identificación puntual de llamada.
- `execution(public * Quick.sort(..))`. Identificación para la ejecución de cualquier método `sort()` en la clase `Quick`.
- `call(* Sorting+.sort(..))`. Identificación para la llamada de cualquier método `sort()` de cualquier subclase de la clase `Sorting`, incluyendo a ella misma.
- `execution(* *.sort(..))`. Identificación de ejecución para cualquier método `sort()` en cualquier clase del sistema.

AspectJ define además otros designadores de corte. Unos permiten identificar cualquier punto de unión dentro de un bloque de código. A estos se les conoce como *designadores léxicos*

porque se basan en la estructura de léxica del código. Por ejemplo el designador `within` permite identificar sobre las clases y `withincode` identifica sobre métodos y constructores. Los designadores para objetos en ejecución `this` y `target` identifican sobre el objeto en ejecución y el objeto en el que se invoca a un método, respectivamente. El designador de argumentos `args` permite identificar sobre argumentos de puntos de unión, como métodos y campos. Finalmente, el designador condicional `if` identifica sobre expresiones booleanas.

2.5.7. Avisos

Un *aviso* (*advice*) es un mecanismo similar a un método para declarar cierto código que debe ejecutarse al alcanzar un punto de unión capturado por un corte. Los avisos definen la funcionalidad de un aspecto. Cuando un punto de unión se identifica, el mecanismo de entrelazado permite alterar la ejecución normal intercalando la ejecución del aviso. El lenguaje AspectJ soporta tres tipos de avisos: *antes* (*before*), *después* (*after*) y *alrededor* (*around*).

Los avisos de tipo *before* y *after* son completamente aditivos a la ejecución de la aplicación y trabajan al inicio y final de los flujos de control que se crean al iniciar los procesamientos dado un punto de unión. El aviso de tipo *around* rodea la ejecución del punto de unión y tiene la capacidad de apropiarse del procesamiento que se inicia en dicho punto de unión, cambiarlo por completo y opcionalmente invocar el comportamiento original.

2.5.8. Información del punto de unión

En AspectJ cada punto de unión provee un objeto que contiene información dinámica y dos objetos que contienen información estática. Estos objetos se basan en información reflexiva y están disponibles de la siguiente forma:

- **thisJoinPoint.** Contiene información dinámica del punto de unión identificado. Provee acceso al objeto destino, al objeto de ejecución y a los argumentos del método. Este objeto es el más utilizado.

- **thisJoinPointStaticPart.** Provee información estática acerca de la ubicación, del tipo y del patrón del punto de unión.

- **thisEnclosingJoinPointStaticPart.** Contiene información del objeto envolvente del punto de unión en cuestión. Para la llamada a un método, este objeto contiene información del método donde se origina dicha llamada.

2.5.9. Precedencia de aspectos

Un aviso de tipo *around* tiene precedencia más alta que uno de tipo *before* o *after*. Sin embargo, diferentes cortes en un mismo aspecto o en diferentes aspectos pueden identificar al mismo punto de unión. Cuando se tienen estos casos, puede ser necesario especificar el orden en el que los avisos podrán aplicarse sobre un punto de unión. Para los avisos de tipo *before*, se ejecuta primero el que tiene la precedencia más alta, mientras que en los avisos de tipo *after* se ejecuta primero el que tiene la precedencia más baja. En los avisos de tipo *around* se ejecuta primero el de precedencia más alta.

2.5.10. Entrelazado de aspectos

El *entrelazado de aspectos* (*weaving*) se realiza de acuerdo a las reglas de composición declaradas en los cortes. El proceso puede realizarse a tiempo de compilación, en tiempo de carga y a tiempo de ejecución. Para entender el proceso de entrelazado, es necesario mencionar que los aspectos se traducen a clases y los avisos a métodos, posteriormente el *entrelazador* incrusta llamadas al código de los avisos en los lugares especificados por los puntos de unión detectados.

- **Entrelazado a tiempo de compilación.** Este tipo de entrelazado también se conoce como *entrelazado estático* (*static weaving*) [47, 61, 71]. En este caso, el proceso se realiza modificando una copia del código fuente de la aplicación y los ajustes van desde insertar invocaciones a métodos que representan a los avisos hasta la modificación del código de las clases para insertar nuevos miembros, tanto de clase como de instancia.

- **Entrelazado a tiempo de carga.** Aprovechando las facilidades que tienen los cargadores de clases en Java, el código intermedio de las clases se copia y se modifica (instrumentación) de la misma forma que el código fuente, pero trabajando al nivel de bytecode [34]. El proceso también cae en la categoría de entrelazado estático.

- **Entrelazado a tiempo de ejecución.** Este proceso también se conoce como *entrelazado dinámico* (*dynamic weaving*). En este caso el entrelazado se realiza por diferentes técnicas: mediante la instrumentación de clases en el momento en que se ejecutan [34, 46], a través de la detección de eventos y manejo de *breakpoints* [67, 68], con técnicas de depuración o mediante el soporte directo de una máquina virtual modificada [10].

2.5.11. Ejemplo de corte dinámico

Considérese el caso en el que se requiere *cronometrar* (*timing*) el tiempo que tarda la ordenación de un arreglo. Esto implica incluir en el código de cada uno de los métodos de ordenación las sentencias adecuadas para ello.

```
01 class Quick extends Sorting {
02   public int[] sort(int[] array) {
03     long ini = System.currentTimeMillis();
04     //código de ordenación
05     long end = System.currentTimeMillis();
06     System.out.println("Tiempo de ejecución: " + (end - ini));
07     return array;
08   }
09 }
```

Sin el enfoque de aspectos, el código de las líneas 3, 5 y 6 debe insertarse en todos los métodos de ordenación, lo cual genera el fenómeno de *invasividad*, *mezclado* y *dispersión* entre las clases que conforman los diferentes métodos de ordenación.

De acuerdo con el modelo de puntos de unión, hay que definir los cortes necesarios que permitan seleccionar los métodos de ordenación en cada una de las clases (ver figura 2.1). El patrón que permite identificar los puntos de unión en forma adecuada es:

```
public * Sorting+.sort(int[])
```

Este patrón especifica a todos los métodos `sort()` con visibilidad pública y con argumento de tipo `int[]`, localizados en las subclases de `Sorting`. Aquí, el comodín `+` identifica a la misma clase `Sorting`, pero como ésta es abstracta al igual que su método `sort()` no se considera en la identificación. Ahora, como lo que interesa es cronometrar el tiempo de ejecución sobre la ordenación, se debe utilizar el designador `execution`:

```
execution(public * Sorting+.sort(int[]))
```

Para definir un corte en AspectJ, se utiliza la palabra reservada `pointcut`, seguido del nombre del corte.

```
pointcut time():
  execution(public * Sorting+.sort(int[]));
```

Para asignar el tiempo al momento de iniciar la ordenación, se utiliza un aviso de tipo *before* el cual contiene el código necesario para ello, adicionalmente se desea mostrar un mensaje de que dará inicio el proceso. En este caso se utilizan las palabras reservadas `before`, para especificar el aviso, y la variable reflectiva `thisJoinPoint` que permite mostrar el punto de unión identificado.

```
before(): time() {
  begin = System.currentTimeMillis();
  System.out.println("Antes  :" + thisJoinPoint);
}
```

Para asignar el tiempo al momento de concluir con la ordenación, se utiliza de manera análoga un aviso de tipo *after*. El código completo del aspecto se muestra a continuación.

```
01 public aspect TimingOnSorting {
02   long begin;
03   long end;
04   pointcut time():
05     execution(public * Sorting+.sort(int[]));
06   before(): time() {
07     begin = System.currentTimeMillis();
08     System.out.println("Antes  :" + thisJoinPoint);
09   }
10   after(): time() {
11     end = System.currentTimeMillis();
12     System.out.println("Despues:" + thisJoinPoint);
13     System.out.println("Tiempo: " + (end - begin));
14   }
04 }
```

La línea 1 define el aspecto mediante la palabra reservada `aspect`. Las líneas 2 y 3 son dos campos del aspecto, `begin` y `end` para guardar los valores de inicio y término de la ejecución de los métodos. Las líneas 4 y 5 definen el corte. Las líneas 6 a 9 y 10 a 14 definen los avisos que se activarán al alcanzar un punto de unión especificado por el corte `time`.

AspectJ ofrece el compilador *ajc*, el cual puede compilar archivos de Java y archivos de AspectJ. Este compilador es al mismo tiempo el *entrelazador* que permite componer el sistema completo de acuerdo a las especificaciones de corte. Para compilar las clases y aspectos se especifica:

```
ajc Sorting.java Quick.java ... TimingOnSorting.aj
```

Donde por convención, los archivos de AspectJ tienen la extensión `aj`. Todo aspecto se convierte en una clase totalmente compatible con el intérprete de Java. La salida de la ejecución es:

```
Antes  :execution(Quick.sort(int[]))
Despues:execution(Quick.sort(int[]))
Tiempo de ejecucion: 16
```

Nótese que la variable `thisJoinPoint` permite obtener la información exacta del punto de unión alcanzado, en este caso `execution(Quick.sort(int[]))`. Con el enfoque de aspectos, la clases se conservan intactas, como se puede ver en la clase `Quick`:

```
01 class Quick extends Sorting {
02   public int[] sort(int[] array) {
03     //código de ordenación
04     return array;
05   }
06 }
```

2.5.12. Ejemplo de corte estático

Cuando un método de ordenación previamente programado en su respectiva clase no forma parte de la jerarquía de herencia existente y se desea integrarlo a la misma, un aspecto con *corte estático* permite hacerlo sin modificar nuevamente el código. Por ejemplo, la clase `Bubble` no es parte de la jearquía:

```
class Bubble {
  public int[] sort(int[] array) {
    //código de ordenación
    return array;
  }
}
```

El aspecto `NewSorting` muestra cómo mediante una simple declaración, la clase se incorpora en la jerarquía de herencia.

```
01 public aspect NewSorting {
02   declare parents: Bubble extends Sorting;
03 }
```

La línea 2 contiene las palabras `declare` y `parents`, las cuales permiten especificar que la clase `Bubble` deberá heredar de la clase abstracta `Sorting`. Este mecanismo también es adecuado para especificar la implementación de interfaces por determinadas clases. Mediante `declare parents` los aspectos pueden modificar la estructura jerárquica de las clases. Las reglas para heredar e implementar aplican de acuerdo a lo establecido en el lenguaje Java.

Ahora, supóngase que el algoritmo de ordenación *Bubble* se quiere mejorar mediante la propuesta de ordenación *CombSort11*, la cual introduce un *factor de compactación* para hacer más eficiente el proceso de ordenación. Dicho factor se puede representar en un campo de la clase `Bubble`. Para esto, nuevamente se requiere aplicar el corte estático, ahora para introducir nuevos miembros en la clase. El aspecto es el siguiente:

```
01 public aspect Comb11ImprovesBubble {
02   final float Bubble.SHRINKFACTOR = 1.3F;
03   public int[] Bubble.sortImproved(int[] array) {
04     //código de ordenación
05     return array;
06   }
07 }
```

Como lo que se requiere es mantener la implementación original junto con el esquema mejorado, ambos, el campo y nuevo método deben integrarse a la clase `Bubble`. En la línea 2 la constante `SHRINKFACTOR` va precedida por el nombre de la clase `Bubble`, indicándole al aspecto que dicho campo deberá insertarse en esa clase. Lo mismo sucede en la línea 3 con el método `sortImproved()`. Las inserciones de nuevos miembros en una clase se conocen como *introducciones* (*introductions* en general o *inter-type declarations* en el caso particular de AspectJ).

Con estos cambios en la estructura de la clase, es posible programar las siguientes sentencias:

```
Sorting r = new Bubble();
ordenado = r.sort(a);
System.out.println("Bubble: " + Arrays.toString(ordenado));

ordenado = ((Bubble)r).sortImproved(a);
System.out.println("Bubble improved: " +
  Arrays.toString(ordenado));
```

2.5.13. Implementaciones de AspectJ

AspectJ cuenta con dos implementaciones, *ajc* que es el compilador original del lenguaje y *abc* que es un compilador extensible. Ambas implementaciones se describen a continuación.

Compilador ajc

El compilador *ajc* es también el entrelazador de bytecode para AspectJ y Java. *ajc* puede compilar archivos de código con extensión `.java` para clases y extensión `.aj` para aspectos. Esta es una convención, pero los archivos de aspectos pueden tener también la extensión `.java`. La compatibilidad del código producido por el compilador *ajc* se basa en las especificaciones del Lenguaje Java [29] y de la Máquina Virtual de Java [57].

Compilador abc

El compilador extensible *abc* (*AspectBench Compiler*) es una implementación del lenguaje AspectJ 1.2 diseñada para realizar extensiones al lenguaje en forma modular. *abc* incluye varias mejoras como el refinamiento del mecanismo de abstracción para patrones de nombre, introducciones con parámetros los cuales se evalúan al momento de realizar el entrelazado, cortes basados en el flujo de datos y aspectos basados en rastreo e historia [4].

abc permite hacer extensiones al modelo de puntos de unión del lenguaje AspectJ. Entre las extensiones incluidas en la misma implementación de *abc* se encuentra el corte sobre conversiones (cast) de objetos y tipos primitivos, así como los cortes globales que permiten a un corte incluir o excluir ciertos elementos (posiblemente) en todo el sistema. Por ejemplo, la siguiente declaración se asegura de que ningún aviso en los aspectos aplique dentro (`within`) de la clase `Hidden`.

```
global : * : !within(Hidden);
```

Capítulo 3

Granularidad en aspectos

3.1. Introducción

Desde sus inicios en 1997, AspectJ ha evolucionado hasta convertirse probablemente en el lenguaje POA con mayor madurez. Se ha tomado como referencia y su terminología ha influenciado el desarrollo y comprensión de la POA. AspectJ utiliza el corte dinámico para alterar el comportamiento de la aplicación mediante la especificación de cortes y avisos. La expresividad de su modelo dinámico de puntos de unión abarca llamadas y ejecuciones de métodos, acceso de lectura/escritura a campos y ejecuciones de inicialización. El modelo de avisos soporta los tipos *before*, *around* y *after*.

Las diferentes implementaciones de POA mantienen esencialmente el mismo modelo de puntos de unión de AspectJ. Algunos enfoques como Arachne [20], JAsCo [78], PROSE [62], y Spring AOP [75] restringen el modelo para facilitar la implementación de cortes y avisos. Al mismo tiempo sólo pocas implementaciones agregan variantes al modelo de puntos de unión:

- Arachne. Define un corte llamado `seq` que identifica secuencias de puntos de unión exactamente en el orden especificado. Los avisos pueden asociarse a cada corte especificado por `seq`, de tal forma que se puede tener una secuencia de ejecuciones de avisos durante una secuencia de cortes específicos.

- JAsCo (*Java Aspect Components*). Soporta la identificación de *secuencias de puntos de unión*.
- PROSE (*PROgrammable extenSions of sErvices*). Permite trabajar con el corte `throw` para identificar puntos de unión al momento de lanzar excepciones.
- AspectWerkz [11]. Define dos cortes adicionales a los de AspectJ: `hasmethod` y `hasfield` para limitar la aplicación de cortes en clases sólo si existen ciertos métodos o campos. Este corte puede llegar a coincidir con `within` de AspectJ.
- abc (AspectBench Compiler) [4]. Esta implementación de AspectJ introduce varias primitivas de corte:
  - `cast` permite capturar conversiones entre objetos.
  - `global` define cortes implícitamente unidos a todos los avisos de un aspecto.
  - `monitorenter/exit` para capturar la adquisición de candados en bloques sincronizados.
  - `maybeShared` para identificar acceso a campos compartidos entre varios hilos.
  - `contains` permite expresar que un punto de unión debe estar léxicamente contenido en otro.
  - `arrayget/set` define cortes sobre el acceso de lectura y escritura en arreglos y expone sus respectivos índices.
  - `let` para exponer la información en *trace-matches*, elementos que permiten activar un aviso después de cierto número de eventos detectados. Esto se logra mediante el enlace de cualquier campo estático con el parámetro del corte.

Como puede apreciarse, independientemente de las adiciones y mejoras para definir cortes, el modelo de puntos de unión se mantiene ligado a los campos y métodos incluyendo `arrayget` y `arrayset` de *abc*. En el caso de Arachne cuya implementación POA es para el lenguaje C, los cortes se aplican sobre funciones.

La expresividad que mantienen estos modelos no permite trabajar con asuntos como *visualización de programas*, *prueba* y *complejidad*, los cuales requieren de otros elementos identificables a tiempo de ejecución: las *variables locales*. Para trabajar con asuntos que están asociados a las variables locales se debe definir un modelo de puntos de unión que soporte una *granularidad fina* o *baja granularidad*.

3.2. Necesidad de granularidad fina

En general, varias propuestas sugieren nuevos modelos, otras proponen extensiones para cuantificar nuevos puntos de unión y otras más se apoyan en refinar el mecanismo de avisos. Independientemente de la aproximación, muchas propuestas coinciden en la necesidad de trabajar con puntos de unión detectables en el interior de los métodos o puntos de unión que describen un conjunto de condiciones que corresponden también a valores en el interior del método [40, 41].

De acuerdo con el concepto de granularidad (sección 2.4.4), los puntos de unión en AspectJ tienen una *granularidad gruesa*, mientras que las cuantificaciones sugeridas en los nuevos enfoques se refieren a la necesidad de una *granularidad fina*, aquella que cuantifica sobre el interior de los métodos. Las siguientes secciones describen las problemáticas que justifican un modelo de puntos de unión más expresivo mediante granularidad fina.

3.2.1. Cuantificación en sistemas de tiempo real

El modelo de puntos de unión de los lenguajes orientados a aspectos carece de expresividad para la modularización de sistemas de tiempo real por dos razones principales:

- Los lenguajes orientados a aspectos se basan en texto y no tienen el soporte adecuado para mostrar (y dar seguimiento) con claridad cómo se entrelazan los aspectos. Esta última característica es importante en el diseño de estos sistemas ya que es necesario ver la ejecución del programa del código entrelazado y ver cuáles aspectos se aplican y en qué orden sin saltar entre el código de los aspectos.

- Los lenguajes proveen soporte sólo para granularidad a nivel de método, el cual es una unidad muy grande (burdo) para separar aspectos en sistemas de tiempo real (STR). Los métodos en STR comúnmente son muy grandes lo cual hace que tengan muchas características mezcladas de forma compleja. Para poder modularizar aspectos en estos sistemas se requiere una granularidad más fina que la ofrecida a nivel de métodos.

Ante esta problemática, se propone producir una *granularidad a nivel de submétodo*. La técnica consiste en descomponer un método en *bloques básicos*, donde cada bloque puede pertenecer a un aspecto. Los bloques básicos pueden ser *bloques de código* o *bloques condicionales*. Un bloque de código es una secuencia de código que no contiene sentencias condicionales y un bloque condicional tiene una sentencia condicional. Si hay dos o más sentencias condicionales, cada una se subdivide para formar nuevos bloques básicos. El flujo de control de un método se estructura mediante bloques básicos y sus conexiones [66].

3.2.2. Cuantificación en pruebas de software

Los mecanismos de cuantificación que tienen los lenguajes orientados a aspectos son muy burdos y además están estrechamente ligados a los conceptos de intercambio de mensajes. Esto hace que el código disperso que está dentro de los métodos no pueda seleccionarse adecuadamente porque no se cuenta con mecanismos de cuantificación más finos.

Para poder trabajar con pruebas de caja blanca es necesario sobrepasar los límites de la modularidad tradicional. El modelo de puntos de unión disponible en los lenguajes orientados a aspectos no es el adecuado para esto ya que solo se soporta el corte sobre puntos de unión expuestos en los límites del módulo. Para solventar esta situación, se propone el corte con acceso a las estructuras de control y a los mecanismos de iteración de estructuras para colecciones [70].

En este mismo contexto se utiliza el concepto de *cobertura de código* (*code coverage*) que describe el grado en que el código fuente se ha probado [48]. Para este proceso se requiere tener acceso a cada línea de código del programa, pero el problema se complica cuando se requiere aplicarlo a componentes que se distribuyen en forma binaria.

3.2.3. Cuantificación en procesos de recomposición

En procesos de *recomposición* (*refactoring*) principalmente de sistemas legados se requiere frecuentemente introducir nuevas sentencias en el interior de los métodos o considerar expresiones concretas para extender la funcionalidad de la aplicación. Se identifican tres tipos de extensiones en granularidad fina:

- **Extensiones de sentencias.** Como ejemplo se tiene la sincronización de una sola sentencia, lo cual requiere tener acceso a las variables locales.
- **Extensiones de expresiones.** Por ejemplo la extensión de condiciones dentro de un `if`.
- **Cambios en la declaración de los métodos.** Por ejemplo la introducción de un nuevo argumento en el método. Aunque en AspectJ se puede utilizar el aviso *around*, en este caso tiene el gran inconveniente de cambiar todo el comportamiento del método, haciendo que el programador tenga que volver a programar todo el código.

También se ha detectado que las *anotaciones* pueden ayudar en esta problemática pero solo sirven como marcadores, donde se necesita de un preprocesador para poder solventar parte de la problemática. Por otro lado la introducción de anotaciones tiene su propia complejidad ya que se requiere tener *consciencia* de dónde marcar el código, lo cual se contrapone a los esquemas de cuantificación propuestos en la POA. Este tipo de refinamientos no se perciben como alguna forma de modularidad.

Para solventar este tipo de problemáticas, se propone la manipulación de árboles de sintaxis abstracta (*abstract syntax tree: AST*) para poder realizar cambios en las sentencias y expresiones [44].

3.2.4. Extensiones para modelos de puntos de unión

En la propuesta de Ubayashi [81] para desarrollar una estructura conceptual que permita crear nuevos modelos o extender modelos de puntos de unión existentes, se propone el uso de

puntos de enganche (*hooks-points*) como mecanismo de identificación generalizado de puntos de unión. Mediante el uso de puntos de enganche es posible refinar las cuantificaciones, considerando esencialmente variables locales y estructuras de control.

El modelo contempla la definición de puntos de unión mediante las estructuras de datos relacionadas, incluyendo AST. Para la definición de cortes se usan *evaluadores* e *interfaces* que permiten saber si un punto de unión pertenece a algunos de dichos cortes. En el caso del entrelazador, se utilizan los AST para saber si hay nodos registrados como puntos de unión.

3.3. Propuestas de modelos de puntos de unión

En esta sección se describen los modelos que proponen extensiones para considerar el *estado del sistema* y el *tiempo de ejecución de eventos* como puntos de unión para mejorar la expresividad de las cuantificaciones.

3.3.1. Modelo de puntos de unión basado en estados

Considerando la tolerancia a fallos, el *monitoreo de estados* es importante porque permite tomar acciones sobre la ejecución del programa. Los actuales modelos de puntos de unión son deficientes para capturar claramente el estado y la transición de estados de un sistema debido a que los puntos de unión expuestos son de bajo nivel para representar adecuadamente no solo el estado, sino la transición entre ellos. La propuesta de solución [3] consiste en agrupar condiciones que definen estados (*states*) que cortan a múltiples clases y generar una máquina de estados abstracta (el aspecto) para exponer puntos de unión como la *transición* (*trans*) de dichos estados.

3.3.2. Modelo de puntos de unión en el tiempo

El modelo de puntos de unión de AspectJ se considera un *modelo de región en el tiempo* porque los puntos de unión representan la duración de un evento. Para poder hacer que los aspectos sean más flexibles y posibiliten una mejor reutilización, es necesario contar con una

granularidad más fina. Por ejemplo, la llamada a un método en AspectJ considera todo el evento desde el inicio hasta la terminación de dicha llamada, pero no es posible sólo identificar el regreso de un método sin tener que considerar el inicio de su llamada.

Endoh [25] propone un modelo de puntos de unión basado en instantes de un evento, llamado *modelo de puntos de unión en el tiempo*. En este modelo, la terminación de las acciones se puede capturar mediante expresiones de corte y no mediante avisos de tipo *after*. Ante esta propiedad, el modelo considera un solo tipo de aviso similar al aviso *around* de AspectJ lo que facilita una mayor reutilización de avisos.

3.4. Enfoques que abordan la baja granularidad

Esta sección presenta las áreas que se han abordado para tratar la baja granularidad de los aspectos con variables locales. Se exponen también las respectivas soluciones que se han desarrollado para solventar parte de las limitaciones con las implementaciones de POA disponibles.

3.4.1. Depuración

La depuración es un proceso metódico para encontrar y reducir el número de errores en el software [52]. En este contexto, el *rastreo* (o la *traza*) de ejecuciones de un programa es útil porque permite detectar y corregir errores complejos [54]. Dicho proceso es un área de aplicación bien conocida de la POA [47] ya que el mecanismo de avisos permite mostrar mensajes para observar tales ejecuciones.

El código que se desarrolla para la depuración corta a través de varios módulos del sistema, e incluso, corta la funcionalidad de las estructuras de control. Tal es el caso cuando se requiere investigar cómo cambia el valor de una variable dentro de un ciclo. Cuando se considera a la depuración como un asunto de corte, el código se puede separar y mejorar la comprensión del sistema. Además, se evita la introducción de nuevos errores al remover el código de depuración de la aplicación.

Sin embargo, aún con los beneficios de poder encapsular el código de este asunto, se tienen restricciones para su aplicación mediante los actuales lenguajes POA ya que éstos solo pueden realizar cuantificaciones sobre campos y métodos. Para tener acceso a las variables locales se requiere exponer los puntos de unión correspondientes.

Bugdel

Bugdel [82] es un sistema orientado a aspectos especializado para depuración. Se distinguen tres características importantes:

1. Cortes específicos para depuración: `line` y `allLines`.

2. Acceso a las variables locales disponibles en el punto de unión.

3. Los cortes y avisos se especifican mediante una interfaz gráfica de usuario.

El designador `line(lineNumber)` corta sobre la línea especificada mientras que `allLines(methodName)` corta a todas las líneas del método especificado.

El mecanismo de avisos provee información del punto de unión interceptado mediante extensiones a la variable reflexiva `thisJoinPoint`:

- `thisJoinPoint.line` ofrece el número de línea correspondiente al punto de unión.

- `thisJoinPoint.filePath` brinda la ruta del código fuente donde se interceptó el punto de unión.

- `thisJoinPoint.variables` da la lista de las variables locales disponibles en el punto de unión.

Bugdel se distribuye como plug-in para Eclipse y se considera un sistema POA de dominio específico.

3.4.2. Paralelismo

En el contexto de la programación paralela, los ciclos juegan un papel importante para mejorar el desempeño de los programas. Harbulot y Gurd [31] plantean un *modelo de puntos de unión para ciclos*, el cual consiste en definir el comportamiento que se quiere reconocer y sus características dinámicas (a tiempo de ejecución).

LoopsAJ

LoopsAJ [30, 31] es la implementación del modelo de puntos de unión para ciclos y es una extensión del compilador *abc*. El mecanismo de cuantificación se construye mediante el designador `loop` en combinación con `args` y/o con `within` para poder cuantificar en forma precisa cierto grupo de ciclos ya que se carece de un identificador o nombre que pueda diferenciar unos de otros.

Para poder identificar los ciclos se analiza el flujo de control a nivel de *bytecode* para evitar ambigüedades causadas por formas alternativas de código fuente que pueden producir código idéntico. El esquema de implementación inserta avisos de tipo *before* y *after* solamente fuera de los límites del ciclo, ya que la identificación de inicio y terminación no se puede garantizar debido a las condiciones de cada ciclo.

3.4.3. Genericidad

El término *genericidad* hace referencia a aquellas características de un lenguaje POA (especialmente su mecanismo de cuantificación) para poder expresar aspectos en forma genérica, es decir, que puedan aplicarse en diferentes contextos.

La genericidad implica conocer acerca de los *efectos de un aspecto*. Un *efecto de aspecto* denota los cambios que un aspecto desarrolla sobre un programa o sobre la ejecución de un programa. Por ejemplo, las especificaciones de efecto en AspectJ corresponden a las introducciones y los avisos.

Kniesel y Rho [49, 74] establecen que un lenguaje de aspectos es *genérico* si sus aspectos

pueden especificar múltiples variantes de un efecto de aspecto (sin requerir de mecanismos de reflexividad).

La genericidad es una propiedad deseable en un lenguaje orientado a aspectos porque permite eliminar las limitaciones para identificar puntos de unión mediante mecanismos de patrones de firmas, tal y como lo hace AspectJ y los lenguajes basados en su modelo de puntos de unión. Si los nombres en los programas base cambian, los aspectos que dependen de dichos nombres tendrán que cambiar la especificación de cuantificación.

LogicAJ2

LogicAJ2 es una implementación de aspectos genéricos [74]. Este modelo se desarrolló para incrementar la expresividad de corte mediante elementos extensibles, verificables estáticamente, disponibles como parte del lenguaje de aspectos y capaces de construir cuantificaciones más robustas y complejas.

La implementación de este modelo define solo tres cuantificadores básicos:

- `expr(join_point, expression_code_pattern)`
- `stmt(join_point, statement_code_pattern)`
- `decl(join_point, declaration_code_pattern)`

El primer argumento de cada corte es una representación explícita de cada punto de unión identificado. El segundo argumento es un patrón que describe el punto de unión.

El corte `expr` cuantifica sobre cualquier expresión de acuerdo a un patrón de código fuente. El corte `stmt` cuantifica en la misma forma pero para sentencias. Ambos permiten identificar cualquier elemento dentro de un método. El corte `decl` cuantifica clases, interfaces, métodos y campos. Los tres cortes permiten identificar cualquier estructura de un programa Java.

Aunque la cuantificación puede llevarse a cabo a cualquier nivel de granularidad, alta o baja, los autores los llaman *cortes de grano fino*.

Los patrones para identificar puntos de unión utilizan *meta-variables lógicas*, las cuales pueden tomar valores sobre las entidades sintácticas del lenguaje y cuyos valores los obtiene

por evaluación de predicados. Esto permite el incremento en la expresividad, reutilización y modularidad de los aspectos.

Capítulo 4

ÉNFASIS: Un framework para aspectos de grano fino

ÉNFASIS es un framework diseñado específicamente para programar extensiones orientadas a aspectos de baja granularidad. Programado totalmente en Java, el framework puede utilizarse en forma tradicional a un compilador o como biblioteca para aplicar:

- Corte sobre variables locales.
- Corte sobre la estructura estática de las clases para agregar miembros.
- Instrumentación de clases.

En este capítulo se muestra cómo programar aspectos de baja granularidad. Se explica en detalle la librería de ÉNFASIS, las clases que la componen y se ilustra el potencial de la herramienta mediante varios ejemplos que encapsulan asuntos. Algunos ejemplos están acompañados del código de *bytecode* desensamblado para poder visualizar el efecto del entrelazado[1]. Se introducen tres conceptos novedosos:

- *Patrones de rutas* para localizar variables locales en el interior de un método.

[1]Los detalles acerca del proceso de entrelazado serán revisados exhaustivamente en el capítulo 5.

- Selección de *conjuntos* específicos de variables para denotar solo algunas de sus ocurrencias.

- *Exposición automática del contexto*, una técnica para exponer el contexto de los cortes sin hacer uso del paso de argumentos entre cortes y avisos.

4.1. Introducción a Énfasis

Como es costumbre, el ejemplo `HolaMundo` se ha utilizado para poder iniciarse en la programación de un nuevo lenguaje. En este caso un ejemplo sencillo es conveniente para poder introducir rápidamente gran parte de los conceptos que Énfasis provee, además de poder discutir con mayor claridad las implicaciones que se tienen en la instrumentación de *bytecode* como parte del proceso de entrelazado. La clase `HolaMundo` es la siguiente:

```
03 public class HolaMundo {
04   public static void main(String[] args) {
05     double d = Math.random();
06     System.out.println(d);
07   }
08 }
```

y una posible salida de la ejecución es:

```
0.5183742931639159
```

La parte importante en este ejemplo es la variable local `d` de tipo `double`. Ésta es una variable disponible, a partir de su declaración, en todo el cuerpo del método `main()`. Considérese el asunto de *monitoreo de variables locales*. La intención es poder mostrar un mensaje previo a la impresión de su valor. El mensaje estará compuesto por una cadena y el mismo valor de la variable. El aspecto que encapsulará dicho asunto es el siguiente:

```
01 import enfasis.lang.*;
02
03 public class HolaMundoAspect extends Aspect {
04   Pointcut p = new GetLocal(
       "void HolaMundo.main(java.lang.String[])/double d");
05   String beforeBody =
       "System.out.println(\"Variable local: \" + d)";
06   Advice a = new Before(beforeBody);
07
08   public void weaving() {
09     advising(a, p);
10   }
11 }
```

Hay varias cosas importantes en este pequeño código del aspecto:

1. Para poder utilizar ÉNFASIS, es necesario utilizar la librería `enfasis.lang`. Ella provee la infraestructura necesaria para poder incorporar aspectos.

2. Todo aspecto se declara extendiendo la definición de la clase `Aspect` (línea 3).

   - La clase `Aspect` provee el método abstracto `weaving()`, el cual debe implementarse para poder especificar las reglas de entrelazado (líneas 8 a la 10).
   - Las reglas de entrelazado se definen en el interior del método `weaving()` invocando al método estático `advising()`, también disponible en la clase `Aspect`.

3. La estructura de corte y los avisos se crean mediante instanciación.

   - El corte `p` es una instancia de la clase `GetLocal`, la cual representa la primitiva de corte para acceso de lectura sobre la variable local (línea 4).

- El aviso `a` es una instancia de la clase `Before`, la cual define el cuerpo del aviso (línea 6).

4. El patrón de cuantificación para la variable local está asociado junto con el patrón del método correspondiente donde se localiza (argumento de `GetLocal` en línea 4). Entre el patrón del método y el de la variable local existe una ruta, en este caso la diagonal, que indica el alcance o contexto donde se quiere localizar a la variable. En este caso la ruta especifica la raíz (`/`) del método.

5. El código que implementarán los avisos son cadenas que contienen instrucciones Java. Solo es necesario escribir el fragmento de código que se necesita para el aviso.

   - Es necesario utilizar secuencias de escape (`\"`) para delimitar las cadenas (`String`) que se utilizan dentro de la cadena que representa al fragmento de código.
   - Todo fragmento de código debe seguir las convenciones del lenguaje Java para declarar sentencias y bloques: terminar con su correspondiente punto y coma (`;`) o en su caso la apertura y cierre de llaves para bloques.

6. Cualquier código del aviso puede utilizar los campos y las variables que estén disponibles en el punto de intercepción. En este caso, el corte aplicará el aviso antes de la lectura de la variable `d` (línea 6 de la clase `HolaMundo`), por lo que previo a su lectura es completamente válido usar la variable en la forma mostrada. Nótese que dicha lectura no se hace antes de la línea 6 de `HolaMundo` (ver sección 4.1.1).

7. Lo anterior evita el uso de argumentos entre los cortes y los avisos para exponer el contexto. A esto se le conoce en ÉNFASIS como *exposición automática del contexto*. En algunas circunstancias, esta característica evita el uso de la variable reflexiva `thisJoinPoint` para conocer el valor del punto de unión.

8. El método estático `advising()` permite indicar cuáles son los avisos que trabajarán con determinados cortes. En este caso, el aviso `a` se activará con el corte `p` (línea 9).

Por cada aviso requerido en algún corte, es necesario declarar una invocación a este método. Si no se declara, entonces no habrá ningún entrelazado de aspectos con las clases.

Cuando el aspecto se ha entrelazado con la clase, el comportamiento se ha alterado y una posible salida de ejecución es la siguiente:

```
Variable local: 0.48497879171688374
0.48497879171688374
```

Como puede apreciarse, se ha obtenido el mensaje deseado antes de la salida original del sistema. Aparentemente esto puede interpretarse como un equivalente a haber insertado en la clase original una línea que imprime el mensaje requerido:

```
double d = Math.random();
System.out.println("Variable local: " + d);
System.out.println(d);
```

Sin embargo no es así. Entre ambas salidas hay una serie de eventos que no es posible manejar con el código fuente. Esto se explica en la siguiente sección.

4.1.1. El bytecode de `HolaMundo`

Es importante mencionar que un punto de unión no está asociado a alguna sentencia o expresión en particular, aunque a veces pueda corresponder. Los puntos de unión están asociados a eventos mas pequeños que no es posible cuantificar usando código fuente, en estos casos es necesario conocer los detalles del bytecode[2].

El siguiente fragmento de bytecode muestra el método `main()` original de la clase `HolaMundo`, junto con las tablas de números de línea (LNT) y de variables locales (LVT).

[2]Para detalles adicionales del bytecode, se puede consultar la sección 5.2.1.

```
public static void main(java.lang.String[]);
  Code:
   Stack=3, Locals=3, Args_size=1
   0:   invokestatic    #16; //Method java/lang/Math.random:()D
   3:   dstore_1
   4:   getstatic       #22;
        //Field java/lang/System.out:Ljava/io/PrintStream;
   7:   dload_1
   8:   invokevirtual   #28;
        //Method java/io/PrintStream.println:(D)V
   11:  return

  LineNumberTable:
   line 5: 0
   line 6: 4
   line 7: 11

  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      12      0     args   [Ljava/lang/String;
   4      8       1     d      D
```

La secuencia original de instrucciones muestra que se invoca al método `random()` (dirección 0) y su valor devuelto se almacena en la posición 1 de la pila de variables locales (`dstore_1`), lo cual corresponde con la línea 5 de la clase `HolaMundo`. Posteriormente se coloca en la pila el valor de la variable local `d` seguido del campo `System.out` (direcciones 4 y 7). Finalmente se invoca el método `println()` que consumirá el valor de la variable y lo mostrará (dirección 8).

Cuando se aplica el aspecto `HolaMundoAspect`, la secuencia de instrucciones cambia por

las siguientes:

```
public static void main(java.lang.String[]);
  Code:
   Stack=5, Locals=3, Args_size=1
   0:   invokestatic    #16; //Method java/lang/Math.random:()D
   3:   dstore_1
   4:   getstatic       #22;
        //Field java/lang/System.out:Ljava/io/PrintStream;
   7:   getstatic       #41;
        //Field java/lang/System.out:Ljava/io/PrintStream;
   10:  new     #43; //class java/lang/StringBuffer
   13:  dup
   14:  invokespecial   #45;
        //Method java/lang/StringBuffer."<init>":()V
   17:  ldc     #47; //String Advising a local variable!
   19:  invokevirtual   #51;
        //Method java/lang/StringBuffer.append:
            (Ljava/lang/String;)Ljava/lang/StringBuffer;
   22:  dload_1
   23:  invokevirtual   #54;
        //Method java/lang/StringBuffer.append:
            (D)Ljava/lang/StringBuffer;
   26:  invokevirtual   #58;
        //Method java/lang/StringBuffer.toString:
            ()Ljava/lang/String;
   29:  invokevirtual   #61;
        //Method java/io/PrintStream.println:
            (Ljava/lang/String;)V
```

```
   32:  dload_1
   33:  invokevirtual   #28;
        //Method java/io/PrintStream.println:(D)V
   36:  return

  LineNumberTable:
   line 5: 0
   line 6: 4
   line 7: 36

  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      37      0     args   [Ljava/lang/String;
   4      33      1     d      D
```

Las direcciones 0, 3 y 4 se mantienen intactas. Nótese que originalmente las direcciones 4, 7 y 8 formaban la sentencia

```
System.out.println(d);
```

y ahora se ven separadas por otro conjunto de instrucciones. En la dirección 7 se coloca el campo `System.out` para preparar una nueva salida. En la dirección 10 se crea una referencia simbólica de la clase `StringBuffer` y para realizar las operaciones requeridas, se duplica la referencia a ella misma (dirección 13). En seguida se crea una instancia de esta clase, se llama a la constante de tipo `String` y se invoca al método `append()` para anexar dicha cadena en el objeto de `StringBuffer` (direcciones 14, 17 y 19). Posteriormente se llama a la variable local y se agrega al contenido de `StringBuffer` (dirección 23). Dicho objeto se convierte a `String` y se muestra en pantalla (direcciones 26 y 29). Hasta este momento el mensaje junto con el valor de la variable `d` se han mostrado por efecto de las instrucciones del aspecto. Finalmente, las direcciones 32 a 36 corresponden a las instrucciones originales de la clase `HolaMundo`.

Concretamente, las direcciones 4 y 7 son dos invocaciones consecutivas para colocar dos veces el campo `System.out` en la pila, las cuales nunca serán generadas mediante la compilación tradicional de Java. Todas estas modificaciones actualizan automáticamente las constantes y atributos necesarios para mantener la integridad de la clase, tal y como se observa en las entradas de las tablas LNT y LVT.

El trabajo con *bytecode* es necesario en ÉNFASIS para poder identificar el instante exacto (punto de unión) de lectura o escritura de una variable local, los cuales corresponden específicamente a las instrucciones de tipo `load` y `store`, respectivamente. Este tipo de control no es posible mediante código Java.

4.2. Patrones de firmas

Los patrones de firmas para aplicar corte sobre las variables locales requieren una notación que pueda identificarlas sin ambigüedad. ÉNFASIS hace uso de nombres y de la estructura léxica del programa para especificar el lenguaje de puntos de unión.

4.2.1. Variables locales

Una variable local no puede tener un modificador de acceso, lo cual hace que carezca de visibilidad, solo se define por su tipo y nombre. El único modificador que puede aparecer en una declaración de variable es `final`, para definir una constante. A partir de estos elementos, se puede establecer el patrón para identificarlas:

```
[modificador] [tipo] [nombre]
```

Donde `modificador` toma el valor único de `final`, `tipo` es un tipo primitivo o un tipo de alguna clase y `nombre` es propiamente el nombre o identificador de la variable local. Para capturar múltiples puntos de unión se utiliza el comodín `*` el cual denota cualquier número de caracteres. La tabla 4.1 muestra algunos ejemplos de identificación con estos patrones de firmas.

Patrón	**Cuantificación**
`int temp`	Variable entera `temp`
`String cadena`	Variable `cadena` de tipo `String`
`Object *`	Todas las variables de tipo `Object`
`* x`	Cualquier variable llamada `x` de cualquier tipo
`*`	Todas las variables locales sin importar su tipo
`final double a`	Constante `a` de tipo `double`

Tabla 4.1: Ejemplos de patrones de firmas para variables locales.

Aunque se puede escribir `* *` para denotar todas las variables locales de un método sin importar su tipo, se puede simplificar su notación mediante un solo asterisco `*` sin generar ambigüedades, tal y como se muestra en la tabla anterior.

La cuantificación de variables locales debe estar acompañada de la parte correspondiente del método al que pertenecen. En este caso se sigue la especificación de AspectJ:

```
[visibilidad] [modificadores] [tipo retorno]
  [Clase].[método([argumentos])]
```

Los argumentos del método son variables locales que siguen el mismo patrón descrito previamente.

Ámbito de variables locales

La combinación de ambos patrones permite identificar claramente a las variables locales correspondientes a un método, sin embargo ambas notaciones no son suficientes. Considérese el siguiente método:

```
void m(int i, int[] ad) {
  if(i < 0) {
    for(int k = 0; k < ad.length; k++)
      ad[k]++;
```

```
  }
  else {
    for(int k = 0; k < ad.length; k++)
      ad[k] = ad[k] * 2;
  }
}
```

Las variables locales que tiene el método son:

- Los argumentos `i` y `ad`. Estas variables son válidas en todo el método. Se dice que estas variables tienen un ámbito válido para todo el cuerpo del método.
- Variable `k`. Esta variable tiene dos ocurrencias, una por cada ciclo `for` y cada una es totalmente independiente de la otra y válida sólo en su respectivo ciclo.

El ámbito y validez de una variable local está determinado por el bloque en donde se define.

4.2.2. Rutas

En base al código previo, el primer ciclo `for` está ubicado en el interior de la sentencia `if`, mientras que el segundo está localizado en la sentencia `else` del `if`. Para poder identificar a la primera variable `k`, es necesario poder especificar que ésta *se encuentra en la sentencia* `if` *y en el ciclo* `for`.

No obstante, los métodos pueden tener cualquier cantidad de sentencias y anidamientos de ciclos. Por esta razón, es necesario especificar las partes del método que son de interés para localizar a la variable local, es decir, los ámbitos relevantes para cuantificar variables locales.

ÉNFASIS propone el uso de *patrones de rutas*. Una *ruta* permite identificar a una variable local mediante la estructura léxica del programa. Dicha estructura corresponde al árbol de sintaxis abstracta (AST) del programa o fragmento del programa.

Las rutas que pueden identificarse en el código anterior se muestran a continuación.

```
void m(...) {   /
  if(...) {     /if
                /if/then
    for(...)    /if/then/for
      ...;      /if/then/for/do
  }
  else {        /if/else/do
    for(...)    /if/else/do/for
      ...;      /if/else/do/for/do
  }
}
```

El caso más simple ocurre cuando se hace referencia a los argumentos, seguido del caso cuando la variable local se define una sola vez. Sin embargo cuando se consideran múltiples ámbitos en el método, se requiere utilizar la estructura léxica del programa. Cada ruta permite hacer referencia a los diferentes ámbitos donde puede ser válida una variable local.

En Java 1.5 se introduce una nueva estructura de control, representada como una variante del ciclo `for`. Se le conoce como un *for mejorado* (*enhanced for*) y en términos prácticos suele llamarse también `foreach`. Ésta palabra es la que se utiliza en ÉNFASIS para especificar rutas que hacen referencia a esta mejora.

Constructores y selectores

La especificación de variables locales por ámbitos a partir de los AST del programa tiene aún el inconveniente de no poder indicar el ámbito específico cuando se tienen múltiples rutas que coinciden al mismo nivel. El siguiente fragmento de código muestra tal situación.

```
for(int i = 0; i < a.length; i++) {
  a[i]++;
}
```

```
System.out.println("Incremento realizado");
for(int i = 0; i < a.length; i++) {
  if(a[i] < 0)
    a[i] = -a[i];
}
System.out.println("Arreglo sin negativos listo");
```

En caso de requerir identificar solamente a la variable local `i` del segundo ciclo `for`, con los patrones de rutas descritos no es posible hacerlo. Se requiere un mecanismo adicional para especificar puntualmente que solo se desea el segundo ciclo `for`.

Se propone el uso de *constructores* y *selectores* como mecanismo de filtrado para sentencias. Un *constructor* corresponde con el nombre simbólico de cada sentencia Java. Puede tener asociado un índice que permite saltar a un determinado AST del mismo nivel. Un *selector* indica la parte de la sentencia que puede escogerse una vez identificado el constructor correspondiente. Los selectores corresponden con las partes restantes de una sentencia, por ejemplo: un constructor para `for` tiene como selector su parte *do*, mientras que un constructor para un `if` tiene como selector la parte `then` o `else`.

De esta manera, para cualquier AST dado, un constructor brincará todos los subárboles que no correspondan con él. Si se especificó un índice, entonces se brincará los subárboles del mismo nivel.

Una *ruta* es entonces, una *combinación de constructores y selectores* que permite identificar la estructura de control precisa a cualquier nivel de anidamiento y con cualquier cantidad de estructuras ubicadas al mismo nivel. Nuevamente, el código se muestra con los constructores C y selectores S correspondientes.

```
                                      C     S    C    S
for(int i = 0; i < a.length; i++) {   /for[1]
  a[i]++;                             /for[1]/do
}
```

```
System.out.println("Incremento");     /
for(int i = 0; i < a.length; i++) {   /for[2]
  if(a[i] < 0)                        /for[2]/do/if[1]
    a[i] = -a[i];                     /for[2]/do/if[1]/then
}
System.out.println("No negativos");   /
```

4.2.3. Ocurrencias

La cuantificación de variables locales por medio de rutas permite identificarlas en sus correspondientes ámbitos donde son válidas. Sin embargo, para trabajar con asuntos de baja granularidad es necesario localizar variables en lugares específicos y no continuos en sus sentencias (diferente *localidad*). Por ejemplo, un asunto de *visualización de algoritmos de ordenación no recursivos* solo requiere información de tres puntos específicos: los valores del arreglo antes y después de la ordenación y antes de cada paso de ordenación. Sin un mecanismo que permita seleccionar sólo las ocurrencias específicas, todas las ocurrencias de la variable local serán monitoreadas. La ordenación con el método de la burbuja ejemplifica esta necesidad.

```
01 class Bubble {
02   public int[] sort(int[] array) {
03     for(int i = array.length; --i >= 0;) {
04       boolean flipped = false;
         //1: acceso antes de la ordenación
05       for(int j = 0; j < i; j++) {
06         if(array[j] > array[j + 1]) {
             //2: accceso antes de cada paso de ordenación
07           int t = array[j];
08           array[j] = array[j + 1];
09           array[j + 1] = t;
```

```
10            flipped = true;
11          }
12        }
          //3: acceso después de la ordenación
13        if(!flipped) {
14          return array;
15        }
16      }
17      return array;
18    }
19 }
```

Los cortes necesarios en los puntos marcados junto con las rutas que podrían ayudar a definirlos son:

1. `after SetLocal(flipped)`, en línea 4 con la ruta: `/for/do`.
2. `before GetLocal(array[i])`, en línea 7 con la ruta: `/for/do/if`.
3. `before GetLocal(flipped)`, en línea 13 con la ruta: `/for/do`.

Como se puede observar, los ámbitos definidos por las rutas no son de gran ayuda porque ambas variables, `flipped` y `array` tienen más ocurrencias, las cuales serán cuantificadas automáticamente si no se restringe la capacidad de selección en los cortes. Para estos casos, ÉNFASIS permite la cuantificación con diferente localidad mediante *ocurrencias* de una variable local y por *número de línea*.

Cuantificación por ocurrencias de variables

En este tipo de cuantificación es necesario realizar un conteo de las ocurrencias de la variable requerida para identificar cuáles son las de interés. El conteo se realiza a partir de la declaración de la variable y es necesario distinguir entre las ocurrencias de acceso de

lectura y escritura. En base al código de la clase `Bubble`, la variable `flipped` tiene un total de 3 ocurrencias. Las dos primeras son de escritura (asignaciones) y la última de lectura. La variable `array` tiene un total de 9 ocurrencias, la 5 (línea 8) y la 7 (línea 9) son de escritura y el resto de lectura. La clase `VisualisationAspect` muestra cómo utilizar la cuantificación por ocurrencias de variables.

```
01 import enfasis.lang.*;
02
03 public class VisualisationAspect extends Aspect {
04   Pointcut p1 = new SetLocal(
       "int[] Bubble.sort(int[])/boolean flipped@{1}");
05   Pointcut p2 = new GetLocal(
       "int[] Bubble.sort(int[])/int[] array@{4}");
06   Pointcut p3 = new GetLocal(
       "int[] Bubble.sort(int[])/boolean flipped@{1}");
08   String adviceBody =
       "System.out.println(java.util.Arrays.toString(array));";
09   Advice a1 = new After(adviceBody);
10   Advice a2 = new Before(adviceBody);
11   Advice a3 = new Before(adviceBody);
12   public void weaving() {
13     advising(a1, p1);
14     advising(a2, p2);
15     advising(a3, p3);
16   }
17 }
```

Nótese que el código que simula la visualización del arreglo (línea 8) es el mismo para cada uno de los avisos (líneas 9 a 11). Las líneas 13 a 15 muestran la implementación de diferentes reglas de entrelazado.

Cuantificación por número de línea

La cuantificación por número de línea es bastante sencilla. Basta con solo poner el número de línea con el selector # en lugar de @.

```
04 Pointcut p1 = new SetLocal(
     "int[] Bubble.sort(int[])/boolean flipped#{4}");
05 Pointcut p2 = new GetLocal(
     "int[] Bubble.sort(int[])/int[] array#{7}");
06 Pointcut p3 = new GetLocal(
     "int[] Bubble.sort(int[])/boolean flipped#{13}");
```

Aunque Énfasis es capaz de identificar correctamente esta notación para líneas, la implementación actual no soporta este caso. En lugar de ello se sugiere utilizar la primitiva `Lines` (ver sección 4.3.2).

4.3. Primitivas de corte

Esta sección describe las primitivas de corte que se proveen en Énfasis. Cada primitiva es un objeto que toma información de acuerdo al corte que representa.

4.3.1. `GetLocal` y `SetLocal`

Los cortes mostrados en las secciones previas ejemplifican el uso de dos primitivas de corte muy importantes: `GetLocal` y `SetLocal`. Estas dos primitivas permiten interceptar los puntos de unión correspondientes e introducir nuevo comportamiento.

No obstante, la cuantificación de variables locales es un proceso que consume muchos recursos, principalmente si se trata de métodos muy grandes. El proceso de entrelazado puede verse mejorado mediante un mecanismo que pueda indagar sobre la existencia de las variables en el método, antes de proceder a buscar alguna ocurrencia. En este sentido, las primitivas `Get/SetLocal` carecen de este potencial.

4.3.2. Lines

Esta primitiva ofrece la capacidad de insertar código directamente en los números de línea especificados. Los números de línea deben corresponder a líneas que contengan instrucciones Java. Las líneas que solo contienen llaves por cierre de bloques o líneas en blanco, no son válidas (debido a que no generan entradas en la tabla de números de línea). A diferencia del número de línea para las ocurrencias, `Lines` no considera alguna variable particular sobre la línea de código. El código de `VisualisationAspect` puede simplificarse utilizando esta primitiva.

```
import enfasis.lang.*;

public class VisualisationAspectLines extends Aspect {
  Pointcut p = new Lines("Bubble", "{4,6,13};");
  String afterBody =
    "System.out.println(\"Tracing array: \" +
    java.util.Arrays.toString(array));";
  Advice a = new After(afterBody);

  public void weaving() {
    advising(a, p);
  }
}
```

El argumento de `Lines` debe especificarse bajo la notación de iniciación de arreglo: una lista de enteros, separados por coma y delimitados por llaves, terminando con punto y coma. Adicionalmente la lista debe estar ordenada ascendentemente.

4.3.3. Execution

Esta primitiva funciona de manera análoga a la de AspectJ. Se provee para tener funcionalidad completa en el modelo de puntos de unión y como complemento a la especificación de la estructura de corte. El aviso de tipo *around* no ofrece soporte para la directiva `proceed()`. El siguiente es un ejemplo que muestra el uso del aviso *around*.

```
import enfasis.lang.*;

public class TestingExecution extends Aspect {
  Pointcut p = new Execution("void HolaExecution.m()");
  String body = "System.out.println(\"A complete around!\");";
  Advice a = new Around(body, null);

  public void weaving() {
    advising(a, p);
  }
}
```

4.4. Composición de cortes

La composición de cortes permite especificar una estructura de corte más robusta y elaborada a partir de las primitivas ya mencionadas. Énfasis soporta los operadores `and`, `or` y `not` cuya precedencia entre ellos es la misma que en Java. Cabe mencionar que el orden de precedencia de la composición no puede cambiarse mediante paréntesis debido al uso de objetos para representar a las primitivas de corte.

La composición de cortes no está basada en el flujo de control como lo hace AspectJ. En Énfasis la composición se refiere a las operaciones de *unión* (or), *intersección* (and) o *diferencia relativa* (not) de la cuantificación realizada exclusivamente con las primitivas `Get/SetLocal` (para ocurrencias) y `Lines` (para líneas de código) sobre el ámbito de todo el

método. Esto abre las posibilidades de poder expresar conjuntos de puntos de unión en forma más flexible. El siguiente fragmento de código muestra la notación y el uso de los operadores de composición.

```
Pointcut pc1 = new GetLocal(int i);
Pointcut pc2 = new SetLocal(double d);
Pointcut composition = pc1.AND(pc2).NOT();
```

Cada primitiva de corte tiene un método que hace la función de operador condicional, el cual para los casos de `and` y `or` reciben como argumento un objeto de tipo `Pointcut` mientras que el operador `not` no requiere argumento. Nótese que el operador `not` está ubicado al final de la expresión, lo cual es una particularidad de la implementación del modelo de corte en ÉNFASIS. El código previo es equivalente a la expresión:

```
!(pc1 && pc2)
```

La evaluación de expresiones lógicas (como cortes) es de izquierda a derecha. Por lo tanto, el resultado de esta expresión es el conjunto de todas las ocurrencias de variables locales que no corresponden con la lectura de `i` ni con la escritura de `d`. Con el corte `pc3`:

```
Pointcut pc3 = new SetLocal(boolean b);
```

la composición

```
pc1.AND(pc2).NOT().OR(pc3);
```

es equivalente a:

```
!(pc1 && pc2) || pc3
```

En términos de cuantificación, esta composición dará como resultado el conjunto de todas las ocurrencias de las variables en un método m expresadas por el resultado de las siguientes operaciones[3]:

$$m \backslash !(pc1 \cap pc2) \cup pc3$$

[3] Ver sección 5.1 para los detalles formales.

4.5. Variable thisJoinPoint

Esta variable ofrece información acerca del punto de unión que se intercepta por una cuantificación. Particularmente en el caso de Énfasis, la información que se provee es específicamente el tipo y nombre de la variable local especificada en una ruta con su correspondiente identificación de acceso de lectura o escritura. Esta restricción se debe a las implicaciones del modelo computacional de Énfasis, donde mantener la información de variables locales a partir de *bytecode* es sumamente costosa.

Retomando el ejemplo de la sección 4.1, la línea 5 del aspecto `HolaMundoAspect` puede cambiarse por:

```
05 String beforeBody = "System.out.println(thisJoinPoint)";
```

Al ejecutar el aspecto, la salida es:

```
getlocal(double d)
```

4.6. Corte estático

El corte estático que soporta Énfasis consiste en la inserción de campos, métodos, constructores e implementación de interfaces. Estas capacidades prácticamente igualan a las de AspectJ para manejar introducciones. Por ejemplo, para insertar un nuevo campo en una clase basta con especificar el tipo y nombre:

```
Introduction f = new Field("short", "s");
```

En el caso de métodos y constructores, éstos se introducen mediante el código completo:

```
Introduction m = new Method("void m() {}");
```

Y para implementar nuevas interfaces, solo se requiere el nombre de la interfaz:

```
Introduction i = new Interface("AnInterface");
```

4.7. Instrumentación

ÉNFASIS permite utilizar su mecanismo de instrumentación directamente en otra biblioteca llamada `enfasis.bytecode`. Mediante esta biblioteca, el programador puede explorar y modificar clases y métodos de acuerdo a sus necesidades. El siguiente código muestra cómo utilizar la API:

```
01 import enfasis.bytecode.ClassModifier;
02
03 public class TestingClassModifier {
04   protected boolean c;
05   public static void main(String[] args) {
06     ClassModifier cmod = new ClassModifier("d:\\", "Example");
07
08     //adding fields
09     cmod.addField("public static int a = 8;");
10     cmod.addField("private int d = 2800;");
11     cmod.addField("Object s = new String(\"Java 1.6......\");");
12
13     //adding methods
14     String method = "public void superMethod(int i) { i += 250;" +
         "System.out.println(i); }";
15     cmod.addMethod(method);
16     String method2 = "static void anStatic() {"+
         "System.out.println(\"init...\"); }";
17     cmod.addMethod(method2);
18
19     //adding constructors
20     String constructor = "public Example(String cadena) {" +
```

```
          " superMethod(8); anStatic(); }";
21      cmod.addConstructor(constructor);
22
23      //saving file
24      cmod.write();
25    }
26 }
```

Esta biblioteca es bastante útil para conocer acerca de las de variables locales. Los siguientes dos códigos muestran su utilización. El primero es una clase que muestra un método con varios ámbitos definidos.

```
public class LocalVariableDemo {
  public void m(short implicitVar) {
    {
      int a = 5, b = 3, c = -9, d = 0, e = 11, f = -999;
      int explicitVar = 3;
      explicitVar++;
      System.out.println(implicitVar);
      explicitVar++;
      System.out.println(explicitVar);
      explicitVar += 128;
      System.out.println(explicitVar);
      f = f + 5;
      c -= 10 + a;
      System.out.println(f + b + c + a + d + e);
    }

    {
      String explicitVar = null;
```

```
      System.out.println(explicitVar);
    }

    {
      int x = 99;
      int explicitVar = 8 + x;
      System.out.println(implicitVar);
      explicitVar += 2000;
      System.out.println(explicitVar);
    }

    {
      if(Math.random() < 0.5) {
        int explicitVar = 10;
        explicitVar++;
        System.out.println(explicitVar);
      }
      int explicitVar = 100;
      System.out.println(explicitVar);
    }
  }
}
```

Y el siguiente código muestra cómo obtener información de las variables locales y sus ámbitos.

```
import enfasis.bytecode.*;

public class TestingLocalVariable {
  public static void main(String[] args) {
```

```
    ClassExplorer ce = new ClassExplorer("d:\\", "LocalVariableDemo");
    MethodExplorer me =
      new MethodExplorer(ce.getMethod("void", "m", "short"));
    LocalVariable var = me.getVariable("int", "explicitVar");

    if(var == null)
      System.exit(0);
    System.out.println("Name: " + var.getName());
    System.out.println("Signature: " + var.getSignature());
    System.out.println("Java type: " + var.getJavaType());
    System.out.println("Is argument: " + var.isArgument());
    for(int i = 0; i < var.getScopeSize(); i++) {
      System.out.println("\n<<< scope: " + i + " >>>");
      System.out.println("Start: " + var.getStart(i));
      System.out.println("Length: " + var.getLength(i));
      System.out.println("Slot: " + var.getSlot(i));
      System.out.println("From line number: " + var.getFromLine(i));
      System.out.println("To line number: " + var.getToLine(i));
      int[] reads = var.getReadIndexes(i);
      int[] writes = var.getWriteIndexes(i);
      for(int r = 0; r < reads.length; r++)
        System.out.println("Read index: " + reads[r]);
      for(int r = 0; r < writes.length; r++)
        System.out.println("Write index: " + writes[r]);
    }
  }
}
```

La salida es la siguiente:

```
Name: explicitVar
Signature: I
Java type: int
Is argument: false

<<< scope: 0 >>>
Start: 23
Length: 71
Slot: 8
From line number: 6
To line number: 18
Read index: 39
Read index: 53
Write index: 21
Write index: 23
Write index: 33
Write index: 44

<<< scope: 1 >>>
Start: 111
Length: 20
Slot: 3
From line number: 25
To line number: 31
Read index: 127
Write index: 110
Write index: 118
```

```
<<< scope: 2 >>>
Start: 144
Length: 10
Slot: 2
From line number: 33
To line number: 36
Read index: 150
Write index: 143
Write index: 144

<<< scope: 3 >>>
Start: 157
Length: 7
Slot: 2
From line number: 37
To line number: 39
Read index: 160
Write index: 156
```

4.8. Especificaciones de corte mediante anotaciones

Las anotaciones proveen información que se necesita para describir un programa, pero que no se puede expresar en términos del mismo lenguaje. Las anotaciones son también conocidas como *meta-datos* y entre sus múltiples aportaciones al lenguaje Java se encuentra la capacidad de especificar valores por medio de *elementos* los cuales se pueden procesar con un programa o herramienta [22]. Una anotación sin elementos se conoce como una *anotación marcador*.

Limitaciones de las anotaciones

Las anotaciones tienen políticas de retención para código fuente, clases y a tiempo de ejecución y pueden aplicarse a tipos, miembros de clases, argumentos y variables locales. Sin embargo, las políticas de retención no son las mismas para diferentes elementos en un programa, lo que resulta en inconsistencias como:

- **Variables locales.** Las anotaciones para variables locales están disponibles solo para código fuente y no pueden utilizarse durante la ejecución del programa. Sin el soporte a tiempo de ejecución, las anotaciones de variables locales se convierten en sólo marcadores.

- **Reflexividad.** Debido al hecho de que Java soporta reflexión solo sobre los miembros de una clase, las anotaciones de variables locales no se soportan a tiempo de ejecución ni en los archivos de clases [77].

- **Construcción de procesadores de anotaciones.** Cuando las anotaciones tienen retención a tiempo de ejecución, no es difícil construir sus correspondientes procesadores (ver apéndice C para un ejemplo). Para las anotaciones de código fuente, la herramienta *apt* (*annotation processing tool*) es de utilidad pero no soporta anotaciones sobre variables locales.

- **Las anotaciones no tienen acceso al contexto dinámico del programa.** Aunque las anotaciones se utilizan para generar archivos, esto es totalmente independiente de la ejecución del programa.

Marcadores: anotaciones como especificaciones de corte

Aunque las anotaciones fueron diseñadas para realizar descripciones acerca de algún elemento del programa, no como directiva de edición de código [35], ÉNFASIS considera específicamente las anotaciones sobre variables locales como *marcadores de corte*[4]. En este caso,

[4]Parte del material cubierto en esta sección fue publicado en [42].

los marcadores de corte operan sobre todas las ocurrencias de lectura o escritura de variables para aplicar avisos.

Un *marcador de corte* es una anotación para variable local que toma como argumento el tipo y cuerpo del aviso a aplicar. Un marcador de corte solo trabaja con avisos de tipo *before* y *after*. Hay dos tipos de marcadores de corte, `GetLocal` y `SetLocal`, los cuales trabajan sobre todas las ocurrencias de la variable local marcada y están diseñados para especificar cortes asociados a actividades de monitoreo y rastreo, aunque pueden tener cualquier cantidad de sentencias Java para implementar cualquier asunto. El siguiente es un ejemplo de una clase Java que usa marcadores para aplicar corte sobre variables locales.

```
import enfasis.lang.markers.GetLocal;

class GetLocalMarker {
  void m() {
    @GetLocal("After(System.out.println(thisJoinPoint);)")
    int i = 7;
    System.out.println(i);

    @GetLocal("Before(System.out.println(j);)")
    int j = i + 7;
    j++;
    System.out.println(j);
  }

  public static void main(String[] args) {
    new GetLocalMarker().m();
  }
}
```

Un marcador solo funciona para la variable local anotada, permite aplicar un único aviso

sobre dicha variable, no requiere especificar rutas ni la creación de una clase para representar al aspecto. El compilador de Énfasis automáticamente indaga las clases a compilar para saber cuáles son las que contienen marcadores de corte y aplicar los avisos correspondientes.

Capítulo 5

Modelo e implementación de Énfasis

Énfasis puede realizar la cuantificación de variables locales a nivel de código fuente mediante *patrones de rutas* y a nivel de *bytecode* por medio del rango de instrucciones donde son válidas las variables locales.

Para la identificación de *rutas* y las respectivas variables locales, es necesario recorrer el AST del código fuente. Esto se logra con la herramienta *JDT* (*Java Development Tooling*) [23], la cual ofrece una API, la *JDT Core*, para manipular código Java. En el caso de la identificación de variables por medio de *bytecode*, se utiliza la herramienta *Javassist* [14] la cual ofrece APIs de alto y bajo nivel para la manipulación del código intermedio.

Este capítulo presenta el modelo base (inicial) sobre el cual se ha formalizado una parte de Énfasis: la selección de ocurrencias de variables locales como conjuntos de índices del registro `pc` de *bytecode* y el filtrado mediante rutas. También se exponen los detalles de implementación mediante las herramientas mencionadas junto con las consideraciones previas para el manejo de *bytecode*.

5.1. Modelo formal de Énfasis

El modelo formal de Énfasis abarca específicamente el modelo de programación, así como la sintaxis y la semántica de los descriptores de corte.

5.1.1. Modelo de programación

El modelo de programación[1] define un punto de unión para una variable local como la dirección de la instrucción que tiene acceso a la variable. Con esta definición, se puede ver un mecanismo que recupere información a tiempo de ejecución de la variable local tal como su nombre, tipo, ubicación (*slot* de la tabla de variables locales: LVT) y la instrucción de acceso.

Esta noción de punto de unión también provee una interpretación operacional basada en el mapeo entre un descriptor de corte y los puntos de unión (dirección de la instrucción) que especifica.

En este modelo, la descripción de alto nivel de los cortes para variables locales se produce mediante la introducción de una notación en el programa fuente para puntos de unión y por el enlace de cada punto de unión a su correspondiente acceso a la dirección de la instrucción. El enlace se puede producir visitando en post-orden el árbol de sintaxis abstracta (AST) del programa fuente y visitando el programa de bytecode en forma sincronizada. La sincronización consiste en unir un fragmento de programa (como subárbol del AST) con el correspondiente segmento de código intermedio generado por el compilador. Como ejemplo, el fragmento:

$$S_3 : \texttt{m} \texttt{ = } (L_5 : \texttt{hi} + L_6 : \texttt{lo})/2;$$

contiene decoraciones usando etiquetas S_3 (para una instrucción `store` y L_5 y L_6 (para instrucciones `load`) sobre las variables locales `m`, `hi` y `lo` respectivamente, para indicar los puntos de unión que son de interés. El AST para este fragmento se puede escribir como sigue:

$$L_5 : \texttt{hi}, L_6 : \texttt{lo}, \texttt{+}, \texttt{2}, /, S_3 : \texttt{m}$$

donde cualquier nodo del AST se visita en post-orden. Después de comparar este fragmento con el bytecode generado por el compilador:

[1] Parte del material cubierto en esta sección fue publicado en [43].

```
12: load_hi
13: load_lo
14: iadd
15: iconst_2
16: idiv
17: istore 4
```

se puede mostrar para cada variable local, que un punto de unión L_i (S_i) se puede asociar a la dirección de la correspondiente instrucción `load` (`store`). Esto es posible porque el orden en que las variables aparecen en el programa fuente se conserva en el código generado por el compilador, excepto para la sentencia de asignación. En este caso, la variable asignada se coloca después de la expresión:

$$[L_5 \rightarrow 12, L_6 \rightarrow 13, S_3 \rightarrow 17, ...]$$

Asociando las etiquetas de puntos de unión a las direcciones de las instrucciones ayudan a mantener una perspectiva de alto nivel para los descriptores de cortes. ÉNFASIS introduce un lenguaje de descriptores de corte para reconocer los patrones de puntos de unión detectados a tiempo de ejecución considerando las rutas.

5.1.2. Sintaxis del descriptor de cortes

La sintaxis abstracta del subconjunto de Java incluye solo una extensión mínima que puede ocultarse durante el análisis lexicográfico. La extensión consiste en decorar cada variable local con una etiqueta única que fija la posición de un punto de unión para la variable.

$$
\begin{array}{lcl}
J & ::= & [L:]\ x\ \texttt{=}\ E\texttt{;} \mid J_1\ J_2 \mid \texttt{if(}B\texttt{)}\ J_1\ \texttt{else}\ J_2 \mid ... \\
E & ::= & c \mid [L:]\ x \mid E_1\ \texttt{+}\ E_2 \mid ... \\
B & ::= & \texttt{true} \mid \texttt{false} \mid E_1\ \texttt{<}E_2 \mid ... \mid B_1\ \texttt{and}\ B_2 \mid ... \\
V & ::= & M[/P][::F\texttt{(}\ \texttt{int}\ x\texttt{)}]\,[\texttt{@}N]\,[\texttt{\#}N] \\
P & ::= & P/K \mid K \mid * \\
K & ::= & C\texttt{[}n\texttt{]} \mid C \mid S \\
C & ::= & \texttt{assign} \mid \texttt{if} \mid \texttt{while} ... \\
S & ::= & \texttt{then} \mid \texttt{else} \mid \texttt{do} \mid ... \\
F & ::= & \texttt{hasLocal} \mid \texttt{getLocal} \mid \texttt{setLocal} \\
D & ::= & V \mid D_1\ \texttt{\&\&}\ D_2 \mid D_1\ \texttt{||}\ D_2 \mid \texttt{!}\ D
\end{array}
$$

En las reglas gramaticales de arriba, v y x toman valores de nombres de variables, c y n toman valores sobre los enteros y naturales respectivamente, y N toma valores sobre el conjunto ordenado de los números naturales. La categoría sintáctica M describe una forma restringida de las declaraciones de métodos que comprende sólo el nombre del método, el tipo y la lista de parámetros. Por simplicidad los tipos se restringen solo a `int` o `int[]`.

Las sentencias del programa son las usuales excepto que las variables pueden estar etiquetadas. Por lo tanto, las etiquetas aparecen en el lado izquierdo de las asignaciones en cualquier parte del programa en el que las expresiones (aritméticas o booleanas) pueden ocurrir. No obstante, las expresiones que no están permitidas en esta versión del modelo de ÉNFASIS son aquellas obtenidas por los operadores de incremento y decremento debido a su difícil semántica. Los paréntesis sintácticos $[\![P]\!]$ denotan el conjunto ordenado de puntos en un fragmento de programa (representado por su árbol de sintaxis abstracta).

5.1.3. Semántica del descriptor de cortes

Las expresiones de rutas son una característica novedosa introducida en ÉNFASIS para lograr una granularidad fina en los descriptores de corte. Una expresión de ruta describe la secuencia de selecciones de subárboles en un AST. La categoría sintáctica P describe la

estructura de una ruta como una secuencia de constructores y selectores. Un constructor en C corresponde a un nombre simbólico de cada sentencia Java. Un constructor puede ir seguido de un índice de salto encerrado en corchetes $C[i]$ para visitar el AST en el i-esimo constructor hermano de C, de otra forma no tiene efecto. Un selector en S indica la parte de una sentencia que se seleccionará enseguida. Por lo tanto, en cualquier nivel dado del AST (por ejemplo un bloque), un constructor salta todos los subárboles cuya raíz no corresponde al mismo constructor. El proceso de selección del subárbol comienza con el AST que representa el código completo de un método dado. Por ejemplo, la expresión de ruta

```
BinarySearch.search(int[], int)/while/do
```

indica que la primer sentencia `while` deberá seleccionarse y después su parte `do`. Recorriendo la definición del método `search()` por el cuerpo del método, inicia el proceso de selección del subárbol. El significado de las expresiones de ruta está dado por las siguientes expresiones:

$$
\begin{aligned}
[\![M]\!] &= [\![J]\!] \text{ para cualquier} \\
&\quad \text{definición de método } M\{J\} \\
[\![J]\!]\,\texttt{/*} &= [\![J]\!] \\
[\![L : x\ \texttt{=}\ E\texttt{;}]\!]\,\texttt{/assign} &= [\![L : x\ \texttt{=}\ E\texttt{;}]\!] \\
C\texttt{[0]} &= C \\
[\![J_1\ J_2]\!]\,/(C\texttt{[}i\texttt{]}/P) &= [\![J_2]\!]\,/(C\texttt{[}i\texttt{]}/P) \\
&\quad \textbf{if}\, \mathit{nombre}\, J_1 \neq C \\
[\![J_1\ J_2]\!]\,/(C\texttt{[}i\texttt{]}/P) &= [\![J_2]\!]\,/(C\texttt{[}i-1\texttt{]}/P) \\
&\quad \textbf{if}\, \mathit{nombre}\, J_1 = C \wedge i > 0 \\
[\![J_1\ J_2]\!]\,/(C/P) &= [\![J_1]\!]\,/C/P \\
&\quad \textbf{if}\, \mathit{nombre}\, J_1 = C \\
[\![\texttt{if (}B\texttt{)}\ J_1\ \texttt{else}\ J_2]\!]\,\texttt{/if/cond} &= [\![B]\!] \\
[\![\texttt{if (}B\texttt{)}\ J_1\ \texttt{else}\ J_2]\!]\,/(\texttt{if/then/}P) &= [\![J_1]\!]\,/P \\
[\![\texttt{if (}B\texttt{)}\ J_1\ \texttt{else}\ J_2]\!]\,/(\texttt{if/else/}P) &= [\![J_2]\!]\,/P
\end{aligned}
$$

donde la función intrínseca $\mathit{nombre}()$ devuelve el nombre del constructor colocado en la raíz de árbol de sintaxis abstracta dado.

Las expresiones de filtro seleccionan solo puntos de unión que corresponden a la variable local dada y también para el tipo de acceso a la instrucción (para lectura o escritura). La estrucura sintáctica de los filtros está dada por la categoría F y se usa en el contexto de la categoría V de la sintaxis abstracta. El significado de una expresión de filtro está dada por la siguientes ecuaciones:

$$
\begin{aligned}
[\![J]\!]\,\texttt{::hasL(int}\,x\texttt{)} &= \begin{cases} [\![J]\!]\,\texttt{::setL(int}\,x\texttt{)} \cup \\ [\![J]\!]\,\texttt{::getL(int}\,x\texttt{)} \end{cases} \\
[\![L : x\ \texttt{=}\ E\texttt{;}]\!]\,\texttt{::setL(int}\,v\texttt{)} &= \textbf{if}\, v = x\, \textbf{then}\, \{L\}\, \textbf{else}\, \emptyset \\
[\![L : x\ \texttt{=}\ E\texttt{;}]\!]\,\texttt{::getL(int}\,x\texttt{)} &= [\![E]\!]\,\texttt{::getL(int}\,x\texttt{)} \\
[\![c]\!]\,\texttt{::getL(int}\,v\texttt{)} &= \emptyset \\
[\![L : x]\!]\,\texttt{::getL(int}\,v\texttt{)} &= \textbf{if}\, v = x\, \textbf{then}\, \{L\}\, \textbf{else}\, \emptyset \\
[\![E_1 + E_2]\!]\,\texttt{::getL(int}\,x\texttt{)} &= \begin{cases} [\![E_1]\!]\,\texttt{::getL(int}\,x\texttt{)} \cup \\ [\![E_2]\!]\,\texttt{::getL(int}\,x\texttt{)} \end{cases} \\
[\![J_1\ J_2]\!]\,\texttt{::xL(int}\,x\texttt{)} &= \begin{cases} [\![J_1]\!]\,\texttt{::xL(int}\,x\texttt{)} \cup \\ [\![J_2]\!]\,\texttt{::xL(int}\,x\texttt{)} \end{cases} \\
[\![\texttt{if (}B\texttt{)}\ J_1\ \texttt{else}\ J_2]\!]\,\texttt{::xL(int}\,x\texttt{)} &= \begin{cases} [\![B]\!]\,\texttt{::xL(int}\,x\texttt{)} \cup \\ [\![J_1]\!]\,\texttt{::xL(int}\,x\texttt{)} \cup \\ [\![J_2]\!]\,\texttt{::xL(int}\,x\texttt{)} \end{cases}
\end{aligned}
$$

donde `xL` significa `setL` o `getL` (la abreviatura de `setLocal` y `getLocal`).

La selección de ocurrencias específicas de un conjunto de puntos de unión es una operación muy útil que evita tener que descomponer expresiones aritméticas y booleanas en todos sus fragmentos. El programador no necesita analizar el AST de una expresión, solo necesita contar el número de ocurrencias de interés, siguiendo el orden en el que aparecen en el programa. De esta forma se simplifica considerablemente la notación de los descriptores de corte, evitando expresiones de rutas muy grandes e ilegibles. El selector de número de ocurrencia `@` y el selector de número de línea `#` se definen por las siguientes descripciones:

$$
\begin{aligned}
[\![J]\!]\,\texttt{@}N &= \{\, j \,!!\, i \mid j \in J \wedge i \in N \,\} \\
[\![J]\!]\,\texttt{\#}N &= [\![J]\!]\,\texttt{@}\,(\textstyle\bigcup_{l \in N} OccsAtLine(l))
\end{aligned}
$$

donde la notación $j \,!!\, i$ toma un elemento i del conjunto ordenado j. El selector de número de ocurrencias `#` colecciona el conjunto de todos los puntos de unión del conjunto ordenado $[\![J]\!]$ que ocurre en las posiciones N. El selector de número de línea `@` primero colecta todas las ocurrencias de cada línea l del conjunto N, utilizando la función $OccsAtLine$, y después aplica el selector de número de ocurrencia.

De la definición del punto de unión también se produce la definición natural de la composición de descriptores de corte. Las reglas de composición están dadas por la categoría D de la gramática. Las conectivas lógicas básicas como conjunción, disyunción y negación corresponde a las operaciones de intersección, unión y complemento de un conjunto ordenado de puntos de unión. El complemento de un conjunto se obtiene con respecto al conjunto completo de puntos de unión de un método.

$$\begin{aligned} [\![D_1]\!] \texttt{\&\&} [\![D_2]\!] &= [\![D_1]\!] \cap [\![D_2]\!] \\ [\![D_1]\!] \texttt{||} [\![D_2]\!] &= [\![D_1]\!] \cup [\![D_2]\!] \\ \texttt{!}\, [\![D]\!] &= [\![M]\!] \setminus [\![D]\!] \end{aligned}$$

Estas operaciones proveen el fundamento para generar descriptores de corte más complejos a partir de unos sencillos.

5.2. Consideraciones previas de implementación

Esta sección ofrece una explicación breve acerca del formato de los archivos `class`, se introduce el concepto de *sombras*, elemento importante para lograr la intercepción de puntos de unión y se describen las herramientas empleadas en la implementación de Énfasis.

5.2.1. Bytecode de archivos `class`

La *máquina virtual de java* (JVM) tiene un conjunto de instrucciones y manipula varias áreas de memoria a tiempo de ejecución. Solo reconoce el formato binario del archivo `class`, el cual contiene las instrucciones de la máquina virtual conocidos como *bytecodes*, la tabla de símbolos, entre otra información complementaria [57].

La JVM define varias áreas de ejecución, algunas se crean al arrancar la máquina virtual y existen mientras no finalice su ejecución. Otras áreas se crean por cada hilo de ejecución. En el caso de los hilos, cada uno tiene su propio registro `pc` (*program counter*) el cual contiene la dirección actual de la instrucción que se está ejecutando. Este registro es bastante robusto para soportar tanto operaciones de *bytecode* como operaciones nativas.

Cuando se crea un hilo, se crea una pila privada que almacena *marcos*. Un *marco* (*frame*) almacena variables locales y resultados parciales, juega un papel importante en las invocaciones a métodos (enlace dinámico), valores de retorno de métodos y despachado de excepciones. El almacenamiento de datos puede requerir una o dos variables locales, dependiendo de su longitud. Las variables locales se direccionan por indexado y el valor de la primera posición es cero. Considérese el siguiente ejemplo en lenguaje Java.

```
public class Base {
  void showInfo(String s) {
    System.out.println(s.length());
  }
}
```

El *bytecode* generado[2] para el método `showInfo(String)` es el siguiente:

```
void showInfo(java.lang.String);
  Signature: (Ljava/lang/String;)V
  Code:
   0:   getstatic       #2;
        //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   aload_1
   4:   invokevirtual   #3; //Method java/lang/String.length:()I
   7:   invokevirtual   #4;
        //Method java/io/PrintStream.println:(I)V
```

[2]La información fue extraída con la herramienta `javap`.

```
10:  return
```

La primera línea indica el método, sus argumentos y el tipo de valor de retorno. La información se muestra como método abstracto y cualquier tipo de alguna clase aparece con su nombre calificado de acuerdo con el paquete al que pertenece. Por ejemplo `String` aparece como `java.lang.String`.

La segunda línea indica la signatura del método, representada en el formato de la JVM. Cada tipo de clase se delimita con una letra `L` al inicio, un punto y coma al final y cada punto del nombre calificado se remplaza por una diagonal. Los datos primitivos se convierten a su representación de JVM, en este caso `void` se representa con la letra `V`. Los modificadores de acceso se colocan a la derecha de los paréntesis que contienen a los argumentos (si hubiera). En caso de existir arreglos, cada dimensión se identifica con un corchete de apertura `[` y se coloca antes del tipo del arreglo. Por ejemplo, `int[]` se representa como `[I`, mientras que `String[][]` equivale a `[[Ljava/lang/String;`.

En seguida aparece el código del *bytecode*. Los números de la columna izquierda representan el registro `pc` o *índice*. En seguida aparece el mnemónico de la instrucción correspondiente, conocido como `opcode`. En caso de existir, una tercera columna mostrará los operandos de la instrucción. Los números precedidos por un símbolo `#` indican la entrada en el `constant_pool`. La información que aparece como comentario (`//`) es la representación completa de las entradas del `constant_pool`, las cuales forman la operación a ejecutar.

Como parte de los posibles atributos que puede tener el archivo `class`, se tiene la *tabla de números de línea* y la *tabla de variables locales*. La primera indica por cada línea de código fuente el rango de instrucciones de *bytecode* correspondientes. La segunda tabla ofrece información acerca de qué variables locales existen en el método, el rango de instrucciones donde son válidas (`start` y `length`), el índice asignado (`slot`), nombre y tipo (`signature`).

```
LineNumberTable:
 line 5: 0
 line 6: 10
```

```
LocalVariableTable:
  Start  Length  Slot  Name   Signature
  0      11      0     this   LBase;
  0      11      1     s      Ljava/lang/String;
```

5.2.2. Sombras para puntos de unión

El proceso de entrelazado en AspectJ utiliza instrumentación para realizar inserción de llamadas a métodos que representan a los avisos. La inserción se realiza en ciertos lugares del *bytecode* que representan posibles puntos de unión. Estos lugares se conocen como *sombras estáticas* del punto de unión. Todas las inserciones alrededor de una sombra permiten modificar el comportamiento dinámico del programa [34].

Definición de sombras para puntos de unión

Un corte que se aplica a la ejecución de un método (`execution`) define una sombra que corresponde a todas las instrucciones del método. Las sombras para campos hacen referencia a una sola instrucción de *bytecode*, por ejemplo `getfield`.

En AspectJ cada tipo de sombra está definida por un tipo, una firma y una región de *bytecode*. Una sombra también tiene una ubicación en el código fuente, cuyo número de línea se está determinado por la tabla de números de línea. Cada punto de unión expone hasta tres estados:

- `this` representa al objeto en ejecución. Solamente existe en el contexto de una instancia y se identifica por medio de la instrucción `aload_0`, la cual siempre representa a la referencia `this`.
- `target` es el objeto destino del punto de unión.
- `args` representa los argumentos del punto de unión.

Tipo	`this`	`target`	`args`	**Sombra**
`execution`	`aload_0` o ninguna	misma que `this`	variables locales	Todo el método
`call`	`aload_0` o ninguna	de la pila	de la pila	`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`
`get`	`aload_0` o ninguna	de la pila	ninguno	`getfield`, `getstatic`
`set`	`aload_0` o ninguna	de la pila	de la pila	`putfield`, `putstatic`

Tabla 5.1: Sombras para métodos y campos en AspectJ.

La tabla 5.1 muestra los estados posibles para las sombras de métodos y campos en AspectJ.

5.2.3. Herramientas empleadas

Una variable local puede aparecer en cualquier parte de una expresión, como parte de las condiciones de un ciclo `for` o `while`, o como datos para poder satisfacer las invocaciones a métodos. Bajo estas consideraciones, el código fuente no es suficiente para poder tener control sobre los puntos de unión requeridos en las especificaciones de cortes. Por ejemplo, para entrelazar una variable dentro de una sentencia `if` no es posible colocar código que no pertenezca a la propia expresión que se evalúa. Es evidente que no se puede utilizar el código fuente para estos casos, lo cual obliga a trabajar con el formato de la clase. A nivel de *bytecode* se tiene control completo de cada una de las instrucciones equivalentes al código fuente y es posible, siempre y cuando no se altere el estado de la pila, introducir nuevas instrucciones sobre cada una de las instrucciones originales del método.

Debido a estas razones, el mecanismo de entrelazado para aspectos de grano fino debe op-

erar obligadamente con el código de *bytecode* y por consecuencia, es necesario hacer transformaciones de las instrucciones mediante herramientas de instrumentación. Al respecto existen herramientas que permiten instrumentar *bytecode* con pleno conocimiento del formato de las clases de Java como lo es BCEL [6] y ASM [17]. También hay herramientas que permiten hacer modificaciones de una forma flexible con apenas una mínima comprensión de la estructura de las clases, en este caso se tiene como ejemplo a *Javassist*.

Para el desarrollo de la investigación se seleccionó Javassist, una herramienta que permite instrumentar a nivel de *bytecode* mediante API's de bajo y alto nivel. Esto es bastante conveniente ya que ofrece adicionalmente un compilador que permite transformar sentencias de código Java en su equivalente *bytecode* y posteriormente insertarlo en algún lugar de la clase. Las siguientes secciones describen las bondades de esta y otras herramientas.

Javassist

Javassist (*Java Programming Assistant*) es una biblioteca para instrumentar *bytecode* de Java [13, 14]. La herramienta provee métodos de acceso a archivos `class` mediante objetos que representan clases, métodos, campos y constructores. Las clases para crear estos objetos son `CtClass`, `CtMethod`, `CtConstructor` y `CtField`. Cada uno de estos objetos proveen métodos para inspeccionar las definiciones de las clases con respecto a la estructura estática de la clase. Estos métodos ofrecen acceso en forma reflexiva y son homólogos a los que se proveen en la API de reflexividad de Java.

Dado que Javassist se creó para la manipulación de clases que no se han cargado en la máquina virtual, no se soporta la creación de objetos, invocación de métodos o acceso a campos. Javassist permite alterar la definición de clases y el diseño de la API se basa en reflexividad estructural, la cual fue primeramente desarrollada en SmalTalk-80.

javap

La herramienta `javap` [76] es un desensamblador de archivos `class`. Permite obtener una representación de alto nivel de las instrucciones de *bytecode*. Entre las múltiples opciones

que tiene, se puede obtener información detallada acerca del *constant_pool*, mnemónicos de las instrucciones correspondientes a cada método, tamaño de la pila por método, datos contenidos en las tablas de números de líneas y de variables locales, entre otros atributos de interés.

La información que puede mostrarse mediante esta herramienta depende de las opciones de usadas para la compilación y/o generación del archivo `class`. Por ejemplo, el compilador `javac` necesita de la opción `-g` para guardar información de depuración, la cual se almacena mediante atributos, específicamente en `LocalVariableTable`.

Java Development Tooling (JDT)

La API de la *Java Development Tooling* (*JDT*) es una herramienta que permite escribir, compilar, probar, depurar y editar programas escritos en Java [23]. La JDT se utiliza para definir extensiones a la plataforma Eclipse, abarcando el lenguaje y la interfaz gráfica.

Particularmente se seleccionó la biblioteca *Java DOM/AST*, la cual contiene un conjunto de clases que modelan la estructura del código fuente como un documento estructurado. Esta herramienta se aplica para implementar el mecanismo de identificación de rutas para localizar variables locales, información que está disponible únicamente en el código fuente.

5.3. Sombras para variables locales

El mecanismo de entrelazado en Énfasis hace uso de sombras estáticas para variables locales, las cuales están representadas por dos familias de instrucciones denominadas `load` y `store`. Las instrucciones `load` permiten colocar una variable local en el *operand stack* mientras que las instrucciones `store` almacenan operandos en una variable local. El conjunto de instrucciones se muestra en la tabla 5.2.

Cada grupo de instrucciones está precedido por una letra que representa un tipo primitivo o una referencia:

- `a` es una referencia a un objeto.

Tipo	**Sombra**
`getlocal`	`aload`, `dload`, `fload`, `iload`, `lload`
`setlocal`	`astore`, `dstore`, `fstore`, `istore`, `lstore`, `iinc`

Tabla 5.2: Sombras estáticas para variables locales.

- `d` es un valor `double`.
- `f` es un valor `float`.
- `i` representa primitivos de tipo `boolean`, `byte`, `short` e `int`.
- `l` es un valor `long`.

Y cada instrucción puede aparecer en dos variantes:

- `xload_<n>` y `xload <n>`.
- `xstore_<n>` y `xstore <n>`.

donde `n` es un índice del arreglo de variables locales, el guión bajo (`_`) se utiliza para índices del 0 al 4 cuyo operando es implícito y `x` es alguna de las letras que representan el tipo de la variable.

Adicionalmente, se considera a la instrucción `iinc` la cual realiza un incremento de una constante a la variable local correspondiente. Esta instrucción modifica directamente el valor de la variable en la pila de variables, por lo que la sombra estática que puede identificarse representa exclusivamente a una operación de asignación (`SetLocal`).

La instrucción `wide` modifica el comportamiento de otra instrucción, en este caso las ya descritas, para extender el índice de la variable y permitir la lectura de *bytes* adicionales que representan al dato. Esta instrucción no representa a una sombra, pero debe considerarse para identificar correctamente los *bytes* implicados en la instrucción.

5.4. Análisis de sombras

Para identificar correctamente los puntos de unión mediante sombras estáticas de variables locales, es necesario conocer en detalle cómo se genera el *bytecode* con los operadores que provee el lenguaje. La implementación de Énfasis no considera operaciones que involucran desplazamiento de bits. Para este análisis se usó la versión 1.6 de Java.

5.4.1. Operadores Java

El primer caso de estudio es la generación de *bytecode* para los operadores de incremento (++), asignación con incremento (+=) y asignación explícita (=). La operación básica es incrementar en 1 una variable de tipo entero primitivo, donde las tres variantes son:

```
int i = 10;
i = i + 1;
```

```
int i = 10;
i++;
```

```
int i = 10;
i += 1;
```

El bytecode generado para las tres variantes es el siguiente:

```
0:   bipush  10     // valor 10
2:   istore_1       // almacenamiento en el slot 1
3:   iinc    1, 1   // incremento del valor en el slot 1 en 1
6:   return
```

El segundo caso es cuando se incrementa una variable con un número grande.

```
int i = 10;
i = i + 50000;
```

```
int i = 10;
i += 50000;
```

El bytecode generado para ambos casos es el siguiente:

```
0:   bipush  10   // valor 10
2:   istore_1     // almacenamiento en el slot 1
3:   iload_1      // recupera el valor del slot 1
4:   ldc     #20; // int 50000
6:   iadd         // suma entera
7:   istore_1     // almacena el resultado en el slot 1
```

Para comprender cómo cambian las instrucciones generadas cuando cambia el tamaño de la constante a incrementar, obsérvese lo siguiente:

- Cuando el incremento ocurre en el rango de 1 byte (-128 a 127) la instrucción generada es `iinc`.
- En el rango de 2 bytes (`short`) la instrucción es `iinc_w`.
- En el rango de 4 bytes (`int`) se generan las instrucciones: `iload`, `ldc`, `iadd` e `istore`.
- En el rango de 8 bytes (`long`) se generan las instrucciones: `lload`, `ldc2_w`, `ladd` y `lstore`.

Si las variables locales son de tipo `byte` o `short`, no se generan instrucciones `iinc`. En su lugar, se generan instrucciones explícitas de `iload` e `istore`, con instrucciones adicionales como `i2b` e `i2s` para las conversiones correspondientes.

Es importante observar el comportamiento cuando se utilizan las clases envolventes de los primitivos anteriores.

```
Byte i = new Byte((byte)0);
i++;
```

```
0:   new             [java/lang/Byte]
3:   dup
4:   iconst_0
5:   invokespecial   [java/lang/Byte."<init>":(B)V]
8:   astore_1
9:   aload_1
10:  invokevirtual   [java/lang/Byte.byteValue:()B]
13:  iconst_1
14:  iadd
15:  i2b
16:  invokestatic    [java/lang/Byte.valueOf:(B)Ljava/lang/Byte;]
19:  astore_1
```

Cuando se utiliza *autoboxing* se tiene:

```
Byte i = 0;
i++;
```

```
0:   iconst_0
1:   invokestatic    [java/lang/Byte.valueOf:(B)Ljava/lang/Byte;]
4:   astore_1
5:   aload_1
6:   invokevirtual   [java/lang/Byte.byteValue:()B]
9:   iconst_1
10:  iadd
11:  i2b
12:  invokestatic    [java/lang/Byte.valueOf:(B)Ljava/lang/Byte;]
15:  astore_1
```

Para los tipos `Short`, `Integer` y `Long` la generación de código es similar, excepto las instrucciones intermedias que hacen las conversiones a los tipos correspondientes. En estos

casos la instrucción `aload` y `astore` son las mismas para todos los casos. Para los tipos primitivos `boolean`, `char` y para los envolventes respectivos, se aplican las mismas reglas.

Los tipos `float`, `double` y los envolventes respectivos generan llamadas explícitas con las instrucciones `fload`, `fstore`, `dload`, `dstore`, `aload` y `astore`. Las dos últimas instrucciones aplican para cualquier tipo de objeto, por lo cual las lecturas y escrituras son fácilmente identificables.

5.4.2. Sombras ocultas: instrucción `iinc`

Una sentencia como:

```
i = i + 25;
```

genera la instrucción de *bytecode*:

```
iinc 8 = 25
```

La instrucción `iinc` se utiliza para hacer incrementos directamente sobre la pila de variables locales, es decir, no se utiliza el *operand stack* para realizar las operaciones. Bajo estas condiciones, `iinc` representa un punto de unión natural solo para operaciones de escritura sobre la variable.

En el contexto de ÉNFASIS, a la instrucción `iinc` se le ha denominado como una *sombra estática oculta* debido a que no permite la identificación de sombras estáticas explícitas de lectura y escritura. Una alternativa para poder generar el punto de unión correspondiente a la lectura en instrucciones `iinc`, es transformar dicha instrucción con soporte de `load` y `store`. Por ejemplo, la instrucción previa de *bytecode*:

```
iinc 8 25
```

puede transformarse en:

```
iload  8
bipush 25
iadd
istore 8
```

De esta forma queda expuesto el punto de unión con sombras para la lectura y escritura de la variable. Es importante notar que este tipo de transformaciones genera una sobre carga en las operaciones de la pila. Su justificación se basa en el hecho de que la utilización de incrementos con valores pequeños hace que los accesos de lectura se pierdan, lo que genera una cuantificación errónea cuando se usa la primitva de corte `GetLocal`.

Soporte para exponer sombras ocultas

El paquete `enfasis.util` contiene la clase `InstructionRefactor` la cual se utiliza como mecanismo de recomposición (*refactoring*) para traducir las instrucciones `iinc` e `iinc_w` (`wide` seguido de `iinc`) en su correspondiente grupo de instrucciones compuestas por `load`, la constante, el operador involucrado y `store`.

La clase define dos métodos que hace una transformación a la vez de cada tipo de instrucción `iinc` e `iinc_w`. Todos los índices, contadores `pc`, direcciones y entradas en las tablas de variables locales y de números de línea se actualizan automáticamente. Por lo tanto no se pierde ninguna referencia entre el nuevo código modificado y su contraparte en el código fuente.

Esta opción de recomposición debe activarse explícitamente en las opciones del compilador de Énfasis. La implementación soporta la exposición de sombras ocultas en combinación con los cortes expresados como marcadores.

5.4.3. Constantes locales (`final`)

Una variable local puede declararse como `final` y tiene implicaciones de sombras ocultas como se muestra en el siguiente ejemplo. El siguiente código muestra la versión con variable local.

```
public void m() {
  int i = 8000;
  System.out.println(i);
}

public void m();
  Code:
   Stack=2, Locals=2, Args_size=1
   0:   sipush  8000
   3:   istore_1
   4:   getstatic
        #15; //Field java/lang/System.out:Ljava/io/PrintStream;
   7:   iload_1
   8:   invokevirtual
        #21; //Method java/io/PrintStream.println:(I)V
   11:  return
```

El siguiente código muestra la versión anterior con la misma variable pero declarada como `final`.

```
public void m() {
  final int i = 8000;
  System.out.println(i);
}

public void m();
  Code:
   Stack=2, Locals=2, Args_size=1
   0:   sipush  8000
   3:   istore_1
```

```
4:   getstatic
     #15; //Field java/lang/System.out:Ljava/io/PrintStream;
7:   sipush  8000
10:  invokevirtual
     #21; //Method java/io/PrintStream.println:(I)V
13:  return
```

Nótese que la instrucción en la dirección 7 coloca directamente la constante sin llamar a la instrucción `iload_1`. La excepción a esta situación ocurre cuando el valor de la constante proviene como resultado de la invocación de un método o es un valor proveniente de otra variable o campo. En este caso, se generarán instrucciones `load` que automáticamente permiten la identificación de puntos de unión.

```
public void m() {
  final int i = (int)(Math.random()*8000);
  System.out.println(i);
}

public void m();
  Code:
   Stack=4, Locals=2, Args_size=1
   0:   invokestatic
        #15; //Method java/lang/Math.random:()D
   3:   ldc2_w  #21; //double 8000.0d
   6:   dmul
   7:   d2i
   8:   istore_1
   9:   getstatic
        #23; //Field java/lang/System.out:Ljava/io/PrintStream;
   12:  iload_1
```

```
13:  invokevirtual
     #29; //Method java/io/PrintStream.println:(I)V
16:  return
```

En la dirección 12 se puede observar cómo la constante `i` se recupera mediante una instrucción `iload_1`. A nivel de *bytecode* no hay un mecanismo que permita identificar una variable local como constante (para los campos sí, ya que la declaración de tipo final existe en el mismo *bytecode*). Una opción para controlar esta situación es obtener información desde el código fuente, detectando si la variable tiene en su declaración la palabra `final`. La implementación de ÉNFASIS no soporta la exposición de este tipo de sombras ocultas.

5.5. Instrumentación de bytecode

Aquí se describen los detalles más importantes acerca de la instrumentación empleada en ÉNFASIS.

5.5.1. Aviso *before*

Como ejemplo de instrumentación, considérese la asignación de la variable entera `m`.

```
m = 10;
```

El compilador genera la siguiente secuencia de bytecode.

```
18: bipush  10
20: istore_1
```

Un punto de unión se puede asociar a la dirección 20 que apunta a la instrucción `istore`, en este caso para `m`. La implementación que soporta el aviso de tipo *before*, puede transformar el código (insertando el código del aviso) de la siguiente manera.

```
18: bipush  10
20: getstatic       [java/lang/System.out]
23: ldc             [String before a local variable!]
25: invokevirtual   [java/io/PrintStream.println]
28: istore_1
```

5.5.2. Aviso after

La implementación del aviso *after* sigue un mecanismo análogo.

```
18: bipush  10
20: istore_1
21: getstatic       [java/lang/System.out]
24: ldc             [String after a local variable!]
26: invokevirtual   [java/io/PrintStream.println]
```

5.5.3. Aviso around

Al aviso *around* permite cambiar y sustituir el comportamiento de la aplicación. En el caso de las variables locales, dicha substitución tiene el efecto de cambiar el valor original de la variable, tanto para el instante de la lectura como el de la escritura, con la ejecución adicional de código.

Para lograr este efecto, ÉNFASIS inserta instrucciones que reemplaza el valor de la variable justo antes de llamarla o almacenarla. Si el aviso contiene código a ejecutar, éste se inserta primero, seguido de las instrucciones de reemplazo.

Directiva `proceed`

Esta directiva permite la invocación original del valor de la variable. En el caso particular de ÉNFASIS, el soporte consiste solamente en utilizarla como indicador de que el valor original de la variable no deberá alterarse después de la ejecución el código del aviso.

Así, las opciones para utilizar el aviso de tipo *around* son las siguientes:

```
Advice a = new Around(aroundBody, "newValue");
```

```
Advice a = new Around(aroundBody, proceed);
```

El programador es responsable de asignar un valor correcto para el tipo de variable, así como de utilizar la directiva `proceed` en forma adecuada. Cualquier variación generará excepciones a causa de posibles valores nulos o incorrectos. El siguiente es un ejemplo.

```
01 import enfasis.lang.*;
02 public class HolaMundoAspect3 extends Aspect {
03   Pointcut p = new GetLocal(
       "void HolaMundo3.main(java.lang.String[])/double d");
04   String aroundBody = "System.out.println(\"Around: \");";
05   //Advice a = new Around(aroundBody, "88.88");
06   Advice a = new Around(aroundBody, proceed);
07
08   public void weaving() {
09     advising(a, p);
10   }
11   public static void main(String[] args) {
12     new HolaMundoAspect3().weaving();
13   }
14 }
```

Nótese que en la línea 5 se tiene el caso correspondiente al cambio de valor. Parte de la implementación del entrelazador toma directamente los valores correspondientes:

```
...
if(aro.getReturnValue().equals("proceed")) //dejar valor actual
```

```
  adviceBody = aro.getBody();
else                                          //aplicar nuevo valor
  adviceBody = aro.getBody() + var.getName() +
    "=" + aro.getReturnValue() + ";";
...
```

Al aplicar el cambio de valor en formato de código fuente (líneas del `else`), ÉNFASIS le pasa el control a Javassist para hacer la inserción en forma transparente. De esta forma, el aviso *around* se convierte en un sencillo esquema similar al aviso *before*.

5.6. Arquitectura de Énfasis

ÉNFASIS tiene una arquitectura basada en capas. La primera capa ofrece información acerca de la estructura de las clases: paquetes, clases, métodos, campos, variables locales, números de línea y datos asociados a otros atributos. La segunda capa permite la instrumentación de *bytecode* sobre cualquier instrucción. Esta capa considera que las inserciones de código están libres de efectos colaterales (la pila no se altera con las nuevas instrucciones). La tercera capa permite la definición de aspectos. Cada capa pasa información a la siguiente para poder crear un framework completo y soportar los aspectos de grano fino, así como el mecanismo de patrones para variables locales.

5.6.1. Capa de instrumentación

Con esta arquitectura es possible usar cada capa de ÉNFASIS como una aplicación individual. Cada capa tiene su propia interfaz de programación, así como la manera de utilizarla. Por ejemplo, el siguiente código muestra cómo encontrar información acerca de la existencia de una variable local.

```
ClassExplorer ce = new ClassExplorer("BinarySearch");
MethodExplorer me = new MethodExplorer(
```

```
ce.getMethod("int", "search", "int[], int"));
if(me.hasVariable("int", "hi"))
System.out.println(me.getVariable("int", "hi"));
```

Aquí, `ClassExplorer` es una clase que busca dentro de la estructura del archivo de la clase y recupera un método en particular. La clase `MethodExplorer` permite preguntar sobre la existencia de una variable. Si la variable se encuentra, entonces toda la información sobre dicha variable se almacena en un objeto de tipo `LocalVariable`. Todas las clases mostradas en el ejemplo pertenecen al paquete `enfasis.bytecode`. Si se quiere instrumentar una clase, se usa la capa de instrumentación.

```
ClassExplorer ce = new ClassExplorer("BinarySearch");
MethodExplorer method = new MethodExplorer(
ce.getMethod("int", "search", "int[], int"));
MethodModifier modif =
new MethodModifier(ce.getKlass(), method.getMethodInfo());
modif.addLocalVariable("int", "c");
modif.insertBefore("{ c = 0; }");
modif.insertAt(5, "{c++;}");
modif.insertAt(12, "{System.out.println();}");
ClassModifier klassMod = new ClassModifier(ce);
klassMod.write("c:\\enfasistesting");
```

La clase `MethodModifier` habilita a una clase para su instrumentación. La clase tiene varios métodos útiles para soportar este proceso de manera flexible. El método `addLocalVariable` inserta una nueva variable local, `insertBefore` permite insertar código al inicio de un método e `insertAt` permite insertar código en una línea de código en particular. La clase `ClassModifier` permite guardar cualquier cambio realizado.

5.6.2. Capa de análisis léxico

La capa de análisis léxico utiliza la especificación de Eclipse para modelar código fuente de Java como un documento estructurado. Se utiliza la API `org.eclipse.jdt.core.dom` la cual ofrece un conjunto de clases para manipular los AST. Particularmente, ÉNFASIS la utiliza para recuperar información a partir de unidades de compilación (cada uno de los archivos `.java`). Al igual que la biblioteca de *bytecode*, la parte léxica también puede utilizarse de manera independiente con fines exclusivamente para explorar el código fuente y determinar las rutas de las variables locales. El siguiente es un ejemplo.

```
import org.eclipse.jdt.core.dom.Statement;
import enfasis.ast.SCClassExplorer;
import enfasis.ast.SCMethodExplorer;
import enfasis.ast.SCUnitExplorer;

public class TestingSCMethodExplorer {
  public static void main(String[] args) {
    SCUnitExplorer unit = new SCUnitExplorer(
      "d:\\", "Example.java");
    SCClassExplorer klass =
      new SCClassExplorer(unit.getTypeClass("Code"));
    SCMethodExplorer method = new SCMethodExplorer(unit,
      klass.getMethod("insertionSort", "int[]"));

    System.out.println("/for/while: " +
      method.pathFinder("/for/while"));
    System.out.println(method.getStatementFound());

    method = new SCMethodExplorer(unit,
      klass.getMethod("method2", "double"));
```

```
    System.out.println("/if/for: " + method.pathFinder("/if/for"));
    System.out.println("/if: " + method.pathFinder("/if"));
    Statement stm = method.getStatementFound();
    System.out.println(stm);

    method = new SCMethodExplorer(unit,
      klass.getMethod("method5", "byte, String, int[], String[]"));
    System.out.println(method.pathFinder("/"));
    stm = method.getStatementFound();
    System.out.println(stm);
    System.out.println("ini: " + method.getStartLine());
    System.out.println("end: " + method.getFinishLine());
  }
}
```

El código sobre el cual opera el ejemplo se anexa en el apéndice B.3. Nótese que en la salida es posible observar algunos de los bloques de código identificados.

```
/for/while: true
{
  arreglo[pos + 1]=arreglo[pos];
  pos--;
}

/if/for: false
/if: true
null
true
{
  int i;
```

```
  i=b;
  double d=3.3;
  i=8;
  int a, e=8, c;
  int x;
  int y;
  x=0;
  y=0;
  field=88;
  String s="H";
}

ini: 198
end: 208
```

Análisis de ámbitos y líneas de código

Esta sección proporciona detalles acerca de la identificación de ámbitos para variables locales.

El siguiente método muestra un ámbito definido por la estructura `if`. La variable `i` es válida desde la línea 12 hasta la 20, mientras que la variable `j` es válida de la 14 a la 17.

```
11 void method0() {
12   int i = 5;
13   if(true) {
14     int j = 9;
15     i++;
16     System.out.println(j);
17   }
18   i++;
```

```
19   System.out.println(i);
20 }
```

El *bytecode* y las correspondientes tablas son:

```
Code:
   Stack=2, Locals=3, Args_size=1
   0:   iconst_5
   1:   istore_1
   2:   bipush 9
   4:   istore_2
   5:   iinc   1, 1
   8:   getstatic      #1; //Field java/lang/System.out:Ljava/io/PrintStream;
   11:  iload_2
   12:  invokevirtual  #2; //Method java/io/PrintStream.println:(I)V
   15:  iinc   1, 1
   18:  getstatic      #1; //Field java/lang/System.out:Ljava/io/PrintStream;
   21:  iload_1
   22:  invokevirtual  #2; //Method java/io/PrintStream.println:(I)V
   25:  return

  LineNumberTable:
   line 12: 0
   line 14: 2
   line 15: 5
   line 16: 8
   line 18: 15
   line 19: 18
   line 20: 25
```

```
  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   5      10      2     j      I
   0      26      0     this   LCode;
   2      24      1     i      I
```

La variable `i` ocupa la ranura 1, con direcciones de la 2 a la 25 y líneas 14 a 20. La variable `j` ocupa la ranura 2, direcciones 5 a la 15 La limitante de trabajar con números de línea para determinar si una variable local está dentro de una ruta es que cuando se escriben múltiples sentencias en una misma línea, se pierde el control de los ámbitos al no disponer de suficiente información en la tabla de números de línea. La solución es manipular mayor cantidad de información mediante los AST, sin embargo, esto implica un deterioro en el desempeño, ya que el recorrido de los AST consume demasiado tiempo.

Por otro lado, disponer de código fuente para detectar rutas por medio de líneas tiene la gran ventaja de preservar la mínima información para la identificación de contextos. El ejemplo muestra cómo el bytecode generado no tiene absolutamente ningún registro de una estructura `if`, debido a que la condición siempre verdadera ejecutará sin lugar a dudas las sentencias. Por lo tanto, un código que genera *bytecode* equivalente puede ser el siguiente:

```
void method0() {
  int i = 5;
  int j = 9;
  i++;
  System.out.println(j);
  i++;
  System.out.println(i);
}
```

Nótese que la tabla de variables locales muestra diferente ámbito para la variable `j`.

```
LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      26      0     this   LCode;
   2      24      1     i      I
   5      21      2     j      I
```

5.6.3. Capa de entrelazado

La capa de entrelazado hace uso de las dos capas revisadas previamente, la capa de instrumentación de *bytecode* y la capa de manipulación de código. Ésta última se utiliza sólo para recuperar información de las rutas para las variables debido al alto costo de procesamiento de AST's. De ésta manera se forma la biblioteca `enfasis.lang`.

Capítulo 6

Conclusiones y trabajo futuro

Los aspectos de grano fino no se soportan en otros lenguajes como AspectJ, ni en otras aproximaciones que difieren en el modelo de puntos de unión. Aunque se han reportado trabajos que la han abordado, la granularidad fina se restringe a un dominio particular [82] o a la selección de expresiones [74].

6.1. Principales contribuciones

Las principales contribuciones del presente trabajo son:

- El diseño de un modelo de puntos de unión para trabajar con aspectos de granularidad fina.
- La implementación de dicho modelo como una biblioteca de Java para programar aspectos sin recurrir al aprendizaje de un nuevo lenguaje.
- El uso de la arquitectura de ÉNFASIS para instrumentar clases de manera flexible.

Estas contribuciones permiten definir aspectos de granularidad fina, los cuales son importantes para dominios específicos como prueba, monitoreo, visualización de programas, desempeño, análisis y comprensión de programas, depuración, entre otros.

6.2. Análisis y discusión

Énfasis es una extensión orientada a aspectos para Java diseñada específicamente para trabajar con asuntos de baja granularidad. El framework de Énfasis está totalmente programado en Java.

6.2.1. Implementación

Énfasis soporta corte estático y dinámico. El *corte estático* permite alterar la estructura de las clases para agregar nuevos miembros, incluyendo constructores. El *corte dinámico* se utiliza para alterar el comportamiento de aplicaciones mediante la especificación de cortes y avisos a nivel de variables locales. El modelo dinámico de puntos de unión es lo suficientemente expresivo para cuantificar ocurrencias específicas de variables locales. Los puntos de unión soportan acceso de lectura y escritura sobre las variables locales.

Para la cuantificación se utiliza el concepto de *patrones de rutas*, los cuales permiten identificar variables locales en partes específicas de un método. Estos patrones evitan la selección completa de todas las variables disponibles en el método, lo cual define *subconjuntos* de elementos cuantificables por la estructura de corte. El concepto de subconjuntos es una característica novedosa de Énfasis y no disponible en otros enfoques orientados a aspectos.

La definición de la estructura de corte se hace en el mismo lenguaje Java. Este enfoque tiene la gran ventaja de utilizar Énfasis como una biblioteca de Java sin la necesidad de aprender nuevas construcciones del lenguaje.

Arquitectura del modelo de ejecución

Énfasis consiste de tres capas:

1. **Capa de instrumentación.** Se hace cargo de explorar y modificar el *bytecode* de las clases de acuerdo a las necesidades de corte. Esta capa cuenta con un mecanismo adicional diseñado específicamente para poder exponer puntos de unión no disponibles de manera natural en ciertas instrucciones de *bytecode*.

2. **Capa de AST.** Trabaja a nivel de código fuente para cuantificar las variables locales por conjunto de ocurrencias o por rutas específicas. En esta capa se incluye toda la infraestructura léxica requerida.

3. **Capa de entrelazado.** Lleva a cabo todo el proceso de administración y control del proceso de entrelazado.

El framework hace uso del *JDT Core* (*Java Development Tooling*) de Eclipse, el cual permite escribir, compilar, probar y editar programas en Java [23]. Esta herramienta es necesaria porque ÉNFASIS identifica variables locales desde el código fuente, aunque también puede hacerlo directamente sobre el *bytecode* haciendo uso de la capa de instrumentación.

El proceso de entrelazado trabaja esencialmente con código fuente generando archivos `class` totalmente compatibles con la JVM. Mediante la capa de bajo nivel se puede instrumentar directamente el *bytecode* sin necesitar del código fuente. Para lograr la instrumentación requerida por el proceso de entrelazado se utiliza la herramienta Javassist [14], la cual permite incluso compilar fragmentos de código Java. Los mecanismos POA se aplican en forma estática y posterior al proceso de compilación (si existe).

Implementación del modelo de programación

A diferencia de otros enfoques, ÉNFASIS no genera archivos `class` adicionales para representar a los aspectos. En su lugar, el proceso de entrelazado inserta las instrucciones que representan a los avisos cuantas veces sea necesario en todo los puntos de la aplicación cuantificados por los cortes.

Por esta misma razón, los avisos no existen como métodos compilados. El código de los avisos de tipo *before* y *after* son completamente aditivos en los lugares de las sombras correspondientes. El aviso de tipo *around* está restringido a reemplazar las instrucciones de *bytecode*, por lo que no es posible hacer uso de la directiva `proceed()` ni se soporta algún mecanismo similar. Por consecuencia, al no contar con una representación de aspectos como clases, no se pueden ensamblar aspectos según se requiera.

Las capacidades de corte estático están restringidas solo a agregar nuevos miembros y constructores a las clases. Los elementos de corte estático para alterar la jerarquía de herencia no están permitidos.

Implementación del modelo de puntos de unión

El modelo de aplicación, que se usa para exponer los puntos de unión, utiliza dos enfoques. El primero usa los AST del código fuente para localizar las ocurrencias de las variables locales, ya sea en todo el método o en secciones específicas delimitadas por los bloques de las estructuras de control disponibles. Aquí se hace uso de los patrones de rutas para delimitar las cuantificaciones.

El segundo enfoque expone puntos de unión que no están disponibles de forma natural en el *bytecode*. Esta situación se presenta cuando la variable local no se carga en la pila de ejecución y su valor se modifica directamente en la pila de variables locales. Este segundo enfoque debe ser habilitado en forma explícita.

Implementación del modelo de corte

El mecanismo de entrelazado utiliza varios objetos que permiten modificar la estructura de las clases y realizar las inserciones en las sombras para variables locales. Para localizar las sombras se debe preguntar primero por la existencia del método que las contiene. Una vez que se corrobora dicha existencia se procede a realizar las inserciones en forma directa sobre cada una de las sombras, en este caso representadas por los `pc` de sus respectivas ocurrencias. El proceso de entrelazado no realiza ningún tipo de optimización sobre el bytecode modificado, solamente se inserta el bytecode necesario.

En caso de habilitarse la exposición de *sombras ocultas*, previo al proceso normal de instrumentación se realiza otro proceso de instrumentación que transforma las instrucciones que ocultan puntos de unión. Una vez concluido este proceso se procede con la instrumentación normal.

Implementación del mecanismo de entrelazado

El proceso de entrelazado hace que todos los avisos sean automáticamente insertados en la sombra correspondiente (*inlined*). Se considera que toda inserción no genera efectos colaterales sobre la pila, es decir, después de la inserción de las instrucciones de los avisos, éstas se consumen hasta llegar al mismo estado previo a la inserción.

En la gran mayoría de los casos, el *bytecode* instrumentado no tiene correspondencia con el código de alto nivel, ya que las inserciones pueden hacerse en sombras que corresponden a variables locales que están como argumentos de métodos, ciclos o en el interior de expresiones.

Particularmente, el proceso de entrelazado no permite la aplicación de avisos sobre clases entrelazadas previamente, excepto cuando se utiliza de manera independiente la biblioteca de instrumentación. El proceso normal de compilación siempre volverá a compilar las clases a partir de su código fuente. Tampoco se hacen inserciones de código condicional para efectuar evaluaciones a tiempo de ejecución sobre algún comportamiento de avisos o aspectos.

Administración de avisos

AspectJ ofrece varias posibilidades de asociación de avisos: por máquina virtual (opción asumida), por hilo y por objeto. Énfasis no soporta ningún tipo de asociación de aspectos, sin embargo, si se considera que el entrelazado distribuye el código de los avisos sobre todos los puntos cuantificados de la aplicación, se tiene el equivalente a una asociación por máquina virtual.

6.2.2. Modelo

El modelo formal de Énfasis abarca sólo las descripciones de corte mediante rutas, las correspondientes variables locales y la composición de cortes.

Modelo de programación

El modelo de programación permite describir la relación entre el código fuente y las instrucciones de bytecode. La relación se establece mediante el mapeo de la cuantificación (el descriptor de corte) y las sombras estáticas para variables locales (los puntos de unión). Este mapeo considera el recorrido de los AST en post-orden para sincronizarse con los *bytecodes*.

Sintaxis de corte

La sintaxis para la descripción de cortes considera la cuantificación de variables mediante expresiones de rutas, constructores y selectores. En el modelo se utilizan etiquetas que permiten fijar la posición de un punto de unión y se incluyen en las reglas gramaticales para extender la notación de Java.

Semántica de cortes

La selección de variables locales depende del contexto que se especifica mediante los patrones de rutas. Para descartar rutas que no son de interés en la cuantificación, la semántica de los cortes incluye la modelación de subárboles AST. Mediante la incorporación de índices para marcar cada AST se introduce el concepto de filtros para rápidamente llegar a la ruta requerida. La selección puntual de variables se hace mediante los selectores # y @ para número de ocurrencias y líneas respectivamente.

Composición de cortes

La composición de cortes se realiza mediante operaciones de conjuntos de variables. Los conjuntos se forman a partir de las direcciones de *bytecode* recuperadas mediante el mapeo del modelo de programación. Las operaciones básicas consideradas en el modelo de composición son unión, intersección y diferencia relativa.

6.3. Limitaciones y trabajo futuro

Las limitaciones y trabajo futuro son:

- Énfasis trabaja con nombres y comodines para especificar cuantificaciones, lo cual hace que los aspectos establezcan dependencias léxicas convirtiéndolos en entidades altamente sensitivas a cambios en el programa base. A esto se le conoce como el problema de *fragilidad en cortes* y debido al uso de patrones de rutas el problema es aún mucho más grave. Es necesario incorporar otros enfoques como por ejemplo el uso de *variables lógicas*, las cuales permiten tomar valores de acuerdo a las entidades del lenguaje, no en cadenas de texto arbitrarias.

- La cuantificación de las variables mediante el conteo de ocurrencias está limitado a conteos de lectura o escritura. Una forma de mejorar las especificaciones de corte es permitir los conteos en forma natural, sin importar si son de lectura o escritura, y dejar a ÉNFASIS la separación de las ocurrencias de lectura y escritura.

- La implementación del modelo de corte no soporta la directiva `proceed()`. Para solventar esta situación, es necesario incorporar *residuos* al momento de realizar la instrumentación. Los *residuos* son instrucciones de lógica condicional para evaluar condiciones de ejecución y permitir la invocación de código, el cual corresponde al comportamiento original.

- Los patrones de rutas solo soportan los ciclos `for`, `foreach`, `while` y `do-while`, así como `if` y `else`. La estructura `switch` no se soporta.

- Las sombras ocultas para constantes no siempre se pueden identificar debido a la forma de cómo se inicializa una variable. La cuantificación sobre estas constantes normalmente será incorrecta. Para solventarlo, es necesario utilizar información del código fuente para definir un mecanismo que permita identificar adecuadamente la asignación de constantes en el *operand stack* y exponer los puntos de unión. Esto llevará a definir

otras instrucciones de bytecode como sombras estáticas para puntos de unión, tales como `bipush`, `sipush`, `ldc` y `ldc2_w`.

- El modelo de corte no considera genéricos ni anotaciones. La única excepción es el soporte para expresar rutas con el ciclo `for` mejorado. Estas capacidades de Java deben incorporarse en ÉNFASIS para compatibilizarlo con las versiones recientes del lenguaje.

- El modelo de puntos de unión se provee como una biblioteca para programar aspectos en el mismo lenguaje Java. Esto restringe la facilidad para expresar avisos con gran cantidad de líneas de código, ya que el código mismo debe escribirse como una cadena de texto. Frecuentemente debe hacerse uso de secuencias de escape para diferencias los tipos `String` dentro de la cadena. La intención es desarrollar un parser que permita llevar a ÉNFASIS a la categoría de lenguaje completo.

- La exposición automática del contexto hace que todo el código del aviso se inserte en todos los puntos cuantificados por el corte. Este enfoque tiene la ventaja de ser altamente eficiente pero al mismo tiempo hace que las clases instrumentadas crezcan de manera considerable, principalmente si se cuantifica sobre todas las ocurrencias de todas las variables y además se aplican múltiples aspectos. Una alternativa para reducir el crecimiento de las clases, es generar clases que implementen avisos como métodos e insertar en la aplicación solo llamadas a estos métodos. El criterio escogido debe permitir un balance entre desempeño y crecimiento de clases.

- La capa de instrumentación que tiene ÉNFASIS permite explorar los archivos `class` en forma flexible, con capacidades que permiten indagar sobre cualquier elemento disponible en el bytecode. Esta capa está restringida a las necesidades de corte lo que limita su utilización como herramienta general de instrumentación. El refinamiento de esta capa puede aportar elementos de análisis de código intermedio a alto nivel.

- El modelo formal de ÉNFASIS solo considera el mapeo de ocurrencias de variables para establecer la composición de cortes. Para formalizar a ÉNFASIS, principalmente en la

categoría de lenguaje completo, es necesario desarrollar una gramática completa del lenguaje.

Apéndice A

Lista de publicaciones

- Juárez Martínez, U. y Olmedo Aguirre, J. O. Entorno dinámico para mejorar la expresividad en lenguajes orientados a aspectos. *En CNCIIC-ANIEI* (México, 2004).
- Juárez Martínez, U. y Olmedo Aguirre, J. O. Soporte dinámico para mejorar la expresividad en lenguajes orientados a aspectos. *En SICI* (México, 2004).
- Juárez Martínez, U. and Olmedo Aguirre, J. O. Énfasis: A Model for Local Variable Crosscutting. *In Proceedings of the ACM Symposium on Applied Computing (SAC)* (Ceará, Brazil, March, 2008), pp 261-265.
- Juárez Martínez, U. and Olmedo Aguirre, J. O. A Join Point Model for Fine-Grained Aspects. *In Proceedings of 2nd. European Computing Conference (ECC)* (Malta, September, 2008).
- Juárez Martínez, U. and Olmedo Aguirre, J. O. Annotation Processing as Local Variable Crosscutting. *In IEEE Andescon International Conference* (Perú, October, 2008).
- Juárez Martínez, U. and Olmedo Aguirre, J. O. Monitoring Assertions with Local Variables in Énfasis. *Computación y Sistemas* (En revisión).

Apéndice B

Código fuente

B.1. Algoritmos de ordenación

```
public abstract class Sorting {
  abstract int[] sort(int[] array);
}

public class Quick extends Sorting {
  private static final long serialVersionUID = -6457156212040719031L;

  public int[] sort(int[] array) {
    sort(array, 0, array.length - 1);
    return array;
  }

  private void sort(int[] a, int lo0, int hi0) {
    int lo = lo0;
    int hi = hi0;
    if(lo >= hi)
```

```
  return;
else if(lo == hi - 1) {
  if(a[lo] > a[hi]) {
    int t = a[lo];
    a[lo] = a[hi];
    a[hi] = t;
  }
  return;
}

int pivot = a[(lo + hi) / 2];
a[(lo + hi) / 2] = a[hi];
a[hi] = pivot;
while(lo < hi) {
  while(a[lo] <= pivot && lo < hi)
    lo++;
  while(pivot <= a[hi] && lo < hi )
    hi--;
  if(lo < hi) {
    int T = a[lo];
    a[lo] = a[hi];
    a[hi] = T;
  }
}
a[hi0] = a[hi];
a[hi] = pivot;

sort(a, lo0, lo-1);
```

```
    sort(a, hi+1, hi0);
  }
}

public class Bubble {
  public int[] sort(int[] array) {
    for(int i = array.length; --i >= 0;) {
      boolean flipped = false;
      for(int j = 0; j < i; j++) {
        if(array[j] > array[j + 1]) {
          int t = array[j];
          array[j] = array[j + 1];
          array[j + 1] = t;
          flipped = true;
        }
      }
      if(!flipped) {
        return array;
      }
    }
    return array;
  }
}
```

B.2. Aspectos

```
public aspect TimingOnSorting {
  long begin;
```

```
  long end;

  pointcut time():
    execution(public * Sorting+.sort(int[]));

  before(): time() {
    begin = System.nanoTime();
    System.out.println("Antes  :" + thisJoinPoint);
  }
  after(): time() {
    end = System.nanoTime();
    System.out.println("Despues:" + thisJoinPoint);
    System.out.println("Tiempo de ejecucion (ns): " + (end - begin));
  }
}

public aspect NewSorting {
  declare parents: Bubble extends Sorting;
}

public aspect Comb11ImprovesBubble {
  final float Bubble.SHRINKFACTOR = (float)1.3;

  public int[] Bubble.sortImproved(int[] array) {
    boolean flipped = false;
    int gap, top, i, j;

    gap = array.length;
```

```
do {
  gap = (int)((float) gap / SHRINKFACTOR);
  switch(gap) {
    case 0: /* the smallest gap is 1 - bubble sort */
      gap = 1;
      break;
    case 9: /* this is what makes this Combsort11 */
    case 10:
      gap = 11;
      break;
    default:
      break;
  }
  flipped = false;
  top = array.length - gap;
  for(i = 0; i < top; i++) {
    j = i + gap;
    if(array[i] > array[j]) {
      int t = array[i];
      array[i] = array[j];
      array[j] = t;
      flipped = true;
    }
  }
}
while(flipped || (gap > 1))
  ;
return array;
```

```
  }
}

public class Algoritmos {
  public static void main(String[] args) {
    Sorting q = new Quick();
    int[] a = new int[] {9,8,7,6,5,4,3,2,1};
    int[] ordenado = q.sort(a);
    System.out.println("Quick: " + Arrays.toString(ordenado));

    Sorting r = new Bubble();
    ordenado = r.sort(a);
    System.out.println("Bubble: " + Arrays.toString(ordenado));

    ordenado = ((Bubble)r).sortImproved(a);
    System.out.println("Bubble improved: " + Arrays.toString(ordenado));

    Searching b = new Binary();
    boolean f = b.search(5, ordenado);
    System.out.println("Existencia de 5 en el arreglo: " + f);
  }
}
```

B.3. Código para exploración de rutas

El nombre de la unidad de compilación es `Example.java`.

```
import java.util.List;
import java.util.ArrayList;
```

```
class Demo {
  void m() {
    System.out.println("Solo por verificar el codigo...");
  }
}

class Code {
  void method0() {
    int i = 5;
    if(true) {
      int j = 9;
      i++;
      System.out.println(j);
    }
    i++;
    System.out.println(i);
  }
  void method00() {
    int i = 5;
    int j = 9;
    i++;
    System.out.println(j);
    i++;
    System.out.println(i);

  }
  public void method1() {
```

```
    for(int i = 0; i < 10; i++) {
      System.out.println(i);
      System.out.println(":::" + i);
    }
    System.out.println("method1()");
  }
  static void method2(double d) {
    System.out.println("method2()");
    if(Math.random() < d + 0.05) {
      System.out.println("excelente");
      d++;
      System.out.println(d);
    }
    else {
      System.out.println("deficiente");
      int i = 8 + (int)d;
      i++;
      i += 2;
      System.out.println(i);
    }
  }
  public void method3() {
    if(true) {
      System.out.println("true true");
      if(true)
        System.out.println("true true true");
      else  {
        for(int i = 0; i < 10; i++) {
```

```
      for(int j = 2; j > -20; j++)
        System.out.println();
      while(true) {
        System.out.println("...");
        break;
      }
    }
  }
}
else
  if(true)
    ;
List<Object> list = new ArrayList<Object>();
list.add(new Object());
list.add(new Object());
list.add(new Object());
int a = 10;
while(a > 1) {
  --a;
}
//foreach[1]
for(Object o : list) {
  //foreach or foreach[1]
  for(int i = 0; i < 3; i++)
    System.out.println(o);
}
System.out.println(a);
```

```
a = 10;
do {
  --a;
} while(a > 1);
System.out.println(a);
for(int z = 0; z < 2; z++)
  ;
if(true)
  System.out.println("second if...");
//foreach[2]
for(Object o : list) {
  for(int i = 0; i < 3; i++)
    System.out.println("Hola..." + o);
  if(true)
    System.out.println("kkkkkkk");
  for(int i = 0; i < 3; i++)
    System.out.println("Hola..." + o);
}
a = 10;
while(a > 1) {
  --a;
}
for(int z = 0; z < 2; z++)
  ;
a = 10;
do {
  --a;
} while(a > 1);
```

```
    a = 10;
    do { //do[3]
      --a;
      for(int z = 0; z < 2; z++)
        ;
      for(int z = 0; z < 2; z++) { //for[2]
        for(int zz = 0; zz < 2; zz++)
          ;
        for(int zz = 0; zz < 2; zz++) { //for[2]
          a = 10;
          while(a > 1) {
            a--;
            System.out.println("intemediate");
            --a;
          }
        }
      }
      for(int z = 0; z < 2; z++)
        ;
    } while(a > 1);
    a = 10;
    do {
      --a;
    } while(a > 1);
  }
  public Code(String s) {}
  int method3(double d, float f) {
    return (int)(d + f);
```

```
}
Code() {}
private Code(byte b) {}
public void insertionSort(int[] arreglo) {
  int temp;
  int pos;
  for(int i = 1; i < arreglo.length; i++) {
    temp = arreglo[i];
    pos = i - 1;
    while((pos >= 0) && (temp < arreglo [pos])) {
      arreglo[pos + 1] = arreglo[pos];
      pos--;
    }
    arreglo[pos + 1] = temp;
  }
}
public String demoArrays(double[][][][] a, String s, Object o, int i) {
  return null;
}
void superPrueba() {
  new Code((byte)2);
}

int field = 8;

public void method4() {
  int a;
  a = 5;
```

```
    System.out.println(a);
    a++;
    --a;
    int b = a;
    String s;
    s = "Hola";
    s = new String(s + "Bola");
    long l = 7L;
    a = (int)l;
    boolean bb = true;
    char c = 'x';
    short ss = 2;
    byte by = 1;
    float f = 3.3F;
    double d = 3.3;
    d = c;
    b = (ss + (int)f) * 5;
    int[] ia = {1,2};
    ia[1] = 8;
    String[] arrays = new String[] {"buu"};
    arrays = null;
    System.out.println((int)(Math.random() * 10));
    for(int x = 0; x < 10; x++)
      a += x;
    a += field;
  }

  void method5(byte b, String ss, int[] ai, String[] as) {
```

```
    int i;
    i = b;
    double d = 3.3;
    i = 8;
    int a, e = 8, c;
    int x; int y;
    x = 0; y = 0;
    field = 88;
    String s = "H";
  }
}

public class Example {
  public static void main(String[] args) {
    Code c = new Code();
    c.method1();
    Code.method2(3.1569);
    System.out.println(c.method3(3.33, 4.56f));
    c.method00();
  }
}
```

Apéndice C

Anotaciones

La clase `Testable`, mostrada abajo, tiene un campo `s` que está anotado con la etiqueta `@Test`.

```
public class Testable {
  @Test String s = "An annotated field";
}
```

La anotación no cambia el significado de cualquier miembro definido en la clase. A nivel de código fuente, las anotaciones son muy similares a un modificador adicional para los miembros de la clase y pueden identificarse y procesarse mediante herramientas para el procesamiento de anotaciones. La anotación mostrada en el ejemplo anterior se define como sigue:

```
import java.lang.annotation.*;
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
}
```

Una anotación es similar a una interfaz con un signo `@` como prefijo junto con una serie de anotaciones adicionales conocidas como meta-anotaciones. `@Target` especifica el tipo de

elemento, en este caso, un campo. Otros valores son: `PARAMETER`, `ANNOTATION`, `TYPE`, `CLASS`, `CONSTRUCTOR`, `METHOD`, `LOCAL VARIABLE`, `PACKAGE` y `TYPE`. `@Retention` especifica en dónde pueden estar disponibles las anotaciones: a nivel de código fuente (`SOURCE`), en los archivos de clase (`CLASS`) o a tiempo de ejecución (`RUNTIME`). Las anotaciones disponibles a tiempo de ejecución se leen reflectivamente por la máquina virtual. Las anotaciones de clase están disponibles en el archivo pero la máquina virtual puede descartarlas, al igual que las anotaciones disponibles en el código fuente son descartadas por el compilador. Independientemente de las políticas de retención, cada anotación requiere su propio procesador. Las anotaciones a tiempo de ejecución requieren un procesador que trabaje con la API de reflexión de Java como `java.lang.reflect`. Las anotaciones en el código fuente se pueden procesar con la herramienta de procesamiento de anotaciones `apt` (annotation processing tool) o directamente con el compilador de Java. En este último caso, el programador utiliza la API Mirror para modelar la estructura semántica de un programa. Finalmente, las anotaciones de clase son básicamente las mismas que las anotaciones a tiempo de ejecución pero están etiquetadas como invisibles por la máquina virtual, lo que hace que se puedan leer con herramientas de instrumentación. El siguiente fragmento de programa para procesar anotaciones en el código fuente muestra cómo verificar la anotación `@Test` en la clase `Testable`.

```
public class CheckAnnotation {
  public static void main(String[] args) throws Exception {
    Class klass = Testable.class;
    Field f = klass.getDeclaredField("s");
    Test t = f.getAnnotation(Test.class);
    System.out.println(t);
  }
}
```

En este ejemplo, el campo anotado `s` de tipo `String` sobre la clase `Testable` se recupera vía reflexión utilizando el método `getAnnotation(Test.class)`, el cual se muestra como `@Test()` en el código fuente.

Bibliografía

[1] Aksit, M. Separation and Composition of Concerns in the Object-Oriented Model. *ACM Comput. Surv. 28* (December 1996).

[2] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A. Abstracting Object Interactions Using Composition Filters. In *Object-based Distributed Processing* (1993), Springer-Verlag, pp. 152–184.

[3] Ali, N. M., and Rashid, A. A State-based Join Point Model for AOP. In *Proceedings of the 1st ECOOP Workshop on Views, Aspects and Role (VAR 05), in 19th European Conference on Object-Oriented Programming* (2005).

[4] Avgustinov, P., Christensen, A. S., Hendren, L. J., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. abc : An Extensible AspectJ Compiler. *Transactions on Aspect-Oriented Software Development I 3880* (October 2005), 293–334.

[5] Baniassad, E., and Clarke, S. Theme: An Approach for Aspect-Oriented Analysis and Design. In *International Conference on Software Engineering (ICSE)* (Washington, DC, USA, 2004), IEEE Computer Society Press.

[6] BCEL. Byte Code Engineering Library. http://jakarta.apache.org/bcel/.

[7] Bergmans, L., and Aksit, M. The Composition-Filters Object Model. Tech. rep., TRESE group, Department of Computer Science, University of Twente, Enschede, The Netherlands, 1994.

[8] Bergmans, L., and Aksit, M. Composing Multiple Concerns Using Composition Filters. Tech. rep., TRESE group, Department of Computer Science, University of Twente, Enschede, The Netherlands, 2001.

[9] Bergmans, L. M. J. *Composing Concurrent Objects*. PhD thesis, University of Twente, Enschede, The Netherlands, 1994.

[10] Bockisch, C., Haupt, M., Mezini, M., and Ostermann, K. Virtual Machine Support for Dynamic Join Points. In *3th Aspect-Oriented Software Development Conference* (2004), ACM, pp. 83–92.

[11] Bonér, J. AspectWerkz: Dynamic AOP for Java. http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf, 2004. AOSD Technical paper.

[12] Booch, G., Maksimchuk, R. A., Engel, M. W., Young, B. J., Conallen, J., and Houston, K. A. *Object-Oriented Analysis and Design with Applications*, 3th ed. Addison-Wesley Professional, April 2007.

[13] Chiba, S. Load-Time Structural Reflection in Java. *Lecture Notes in Computer Science 1850* (2000).

[14] Chiba, S., and Nishizawa, M. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of Generative Programming and Component Engineering* (2003), Springer-Verlag, pp. 364–376.

[15] Clarke, S., and Baniassad, E. *Aspect-Oriented Analisys and Design - The Theme Approach*. Addison-Wesley, March 2005.

[16] Colyer, A., Rashid, A., and Blair, G. On the Separation of Concerns in Program Families. Tech. Rep. COMP-001-2004, Computing Department, Lancaster University, 2004.

[17] Consortium, O. ASM. http://asm.objectweb.org/.

[18] Czarnecki, K. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration abd Fragment-Based Component Models.* PhD thesis, Technische Universitat Ilmenau, Germany, 1999.

[19] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computation. Prentice Hall, Inc., October 1976.

[20] Douence, R., Fritz, T., Loriant, N., Menaud, J.-M., Segura-Devillechaise, M., and Sudholt, M. An Expressive Aspect Language for System Applications with Arachne. In *4th International Conference on Aspect-Oriented Software Development* (March 2005), ACM.

[21] Eaddy, M., and Aho, A. V. Statement Annotations for Fine-Grained Advising. In *Reflection, AOP and Meta-Data for Software Evolution* (2006), pp. 89–99.

[22] Eckel, B. *Thinking in Java*, 4th ed. Prentice Hall, 2006.

[23] Eclipse. Jdt core. http://help.eclipse.org/help32/index.jsp.

[24] Elrad, T., Fillman, R. E., and Bader, A. Aspect-Oriented Programming. Communications of ACM, October 2001.

[25] Endoh, Y., Masuhara, H., and Yonezawa, A. Continuation Join Points. In *Foundations of Aspect-Oriented Languages Workshop at AOSD* (March 2006), pp. 1–10.

[26] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[27] Filman, R. E., and Friedman, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. Tech. Rep. 01.12, RIACS, May 2001. Presented at Workshop on Advanced Separation of Concerns, OOPSLA October 2000.

[28] Foundation, T. E. AspectJ. http://www.eclipse.org/aspectj/.

[29] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language™ Specification Third Edition*. Addison-Wesley, 2005. http://java.sun.com /docs/books/jls/index.html.

[30] Harbulot, B. LoopsAJ. http://intranet.cs.man.ac.uk/cnc/projects/loopsaj.php.

[31] Harbulot, B., and Gurd, J. R. A join point for loops in AspectJ. In *Foundations of Aspect-Oriented Languages workshop, AOSD* (2006), R. E. Filman, Ed., ACM, pp. 63–74.

[32] Harrison, W., and Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects). *ACM SIGPLAN Notices 28*, 10 (1993), 411–428.

[33] Hetzel, W. C. *The Complete Guide to Software Testing*, 2nd ed ed. Mass Wellesley: QED Information Sciences, 1988.

[34] Hilsdale, E., and Hugunin, J. Advice Weaving in AspectJ. In *AOSD'04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development* (2004), pp. 26–35.

[35] Horstmann, C., and Cornell, G. *Core Java*, 7th ed., vol. 2. Prentice Hall PTR, 2004.

[36] IBM. Jikes Compiler. http://jikes.sourceforge.net/.

[37] Jacobson, I. Language Support for Changeable Large Real Time Systems. In *OOPLSA'86: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, USA, 1986), ACM, pp. 377–384.

[38] Jacobson, I., and Ng, P.-W. *Aspect-Oriented Sfotware Development with Use Cases*. Addison Wesley, 2005.

[39] JBoss. JBoss AOP. http://www.jboss.org/jbossaop/.

[40] Juárez-Martínez, U., and Olmedo-Aguirre, J. O. Entorno dinámico para mejorar la expresividad en lenguajes orientados a aspectos. In *CNCIIC-ANIEI* (México, 2004).

[41] Juárez-Martínez, U., and Olmedo-Aguirre, J. O. Soporte dinámico para mejorar la expresividad en lenguajes orientados a aspectos. In *SICI* (México, 2004).

[42] Juárez-Martínez, U., and Olmedo-Aguirre, J. O. A Join Point Model for Fine-Grained Aspects. In *Proceedings of 2nd. European Computing Conference* (Malta, September 2008).

[43] Juárez-Martínez, U., and Olmedo-Aguirre, J. O. Énfasis: A Model for Local Variable Crosscutting. In *Proceedings of the ACM Symposium on Applied Computing (SAC)* (Ceará, Brazil, March 2008), pp. 261–265.

[44] Kastner, C., and Apel, S. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)* (Leipzig, Germany, May 2008).

[45] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ. In *European Conference on Object-Oriented Promgramming 2001, Lecture Notes in Computer Science 2072* (June 2001), Springer-Verlag, pp. 327–353.

[46] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting Started with AspectJ. *Communications of the ACM 44*, 10 (October 2001), 59–65.

[47] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C. V., Maeda, C., and Mendhekar, A. Aspect-Oriented Programming. *ACM Comput. Surv. 28* (1996).

[48] Kniesel, G., and Austermann, M. CC4J - Code Coverage for Java A Load-Time Adaptation Success Story. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment* (2002), Springer-Verlag, pp. 155–169.

[49] Kniesel, G., and Rho, T. Generic Aspect Languages - Needs, Options and Challenges. Tech. rep., University of Bonn, Germany, 2005. Presented at 2nd Francophone Day on Aspect-Oriented Software Development.

[50] Kopiright. Kopi Java Compiler. http://www.kopiright.com/kopi_projekt.php.

[51] Laddad, R. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications, July 2003.

[52] Law, R. An Overview of Debugging Tools. *SIGSOFT Software Engineering Notes 22*, 2 (March 1997), 43–47.

[53] Leffingwell, D., and Widrig, D. *Managing Software Requirements: A Use Case Approach*, 2nd ed. Addison-Wesley, 2003.

[54] Liang, D., and Xu, K. Debugging Object-Oriented Programs with Behavior Views. In *AADEBUG'05: Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (September 2005), ACM, pp. 133–142.

[55] Lieberherr, K. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, 1st ed. Pws Pub Co, August 1995.

[56] Lieberherr, K. Demeter and aspect-oriented programming. http://www.ccs.neu.edu/home/lieber/connection-to-aop.html, 1998.

[57] Lindholm, T., and Yellin, F. *The Java™ Virtual Machine Specification Second Edition.* Prentice Hall, 1999. http://java.sun.com/docs/books/jvms/index.html.

[58] Lopes, C. V., and Kiczales, G. Recent Developments in AspectJ. In *Proceedings of European Conference on Object-Oriented Programming - Workshop on Aspect-Oriented Programming* (Brussels, Belgium, July 1998), Springer, pp. 398–401.

[59] Mezini, M. *Variation-Oriented Programming: Beyond Classes and Inheritance.* PhD thesis, Fachbereich Elektrotechnik und Informatik, Siegen, Germany, 1997.

[60] Mezini, M. Towards Variational Object-Oriented Programming: The Rondo Model. Tech. rep., Darmstadt University of Technology, 1998.

[61] Mezini, M., and Ostermann, K. Conquering Aspects with Caesar. In *AOSD'03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development* (Boston, MA, USA, March 2003), ACM, pp. 90–100.

[62] Nicoara, A., and Alonso, G. Dynamic AOP with PROSE. In *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications* (Porto, Portugal, June 2005), J. Castro and E. Teniente, Eds., pp. 125–138.

[63] Ossher, H., and Tarr, P. Multi-Dimensional Separation of Concerns in Hyperspace. Tech. rep., IBM T. J. Watson Research Center, 1999.

[64] Ossher, H., and Tarr, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development* (2000), IBM T. J. Watson Research Center, Kluwer.

[65] Ostermann, K., Mezini, M., and Bockisch, C. Expressive Pointcuts for Increased Modularity. In *European Conference on Object-Oriented Programming* (2005), pp. 214–240.

[66] Park, J., and Hong, S. Customizing Real-Time Operating Systems with Aspect-Oriented Programming Framework. In *Separation of Concerns Design Conference* (2003), pp. 966–970.

[67] Popovici, A., Gross, T., and Alonso, G. Dynamic Homogenous AOP with PROSE. Tech. rep., Department of Computer Science, ETH Zurich, Switzerland, March 2001.

[68] Popovici, A., Gross, T. R., and Alonso, G. Dynamic Weaving for Aspect-Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development* (Enschede, The Netherlands, April 2002), pp. 141–147.

[69] Rajan, H., and Sullivan, K. Need for Instance Level Aspect Language with Rich Pointcut Language. *Software-Engineering Properties of Languages for Aspect Technologies in conjunction with the Second International Conference on Aspect-Oriented Software Development* (March 2003).

[70] Rajan, H., and Sullivan, K. Generalizing AOP for Aspect-Oriented Testing. In *In the proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005)* (Chicago, IL, USA., March 2005).

[71] Rajan, H., and Sullivan, K. J. Eos: Instance-Level Aspects for Integrated System Design. In *ESEC / SIGSOFT FSE* (2003), pp. 291–306.

[72] Rajan, H., and Sullivan, K. J. Classpects: Unifying aspect- and object-oriented language design. In *Proceedings of 27th International Conference on Software Engineering (ICSE)* (May 2005), IEEE Computer Society Press, pp. 59–68.

[73] Rashid, A., Sawyer, P., Moreira, A. M. D., and Araujo, J. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *Proceedings of IEEE Joint International Conference on Requirements Engineering (RE)* (2002), pp. 199–202.

[74] Rho, T., Kniesel, G., and Appeltauer, M. Fine-Grained Generic Aspects. In *Foundations Of Aspect-Oriented Languages workshop* (Bonn, Germany, March 2006).

[75] SpringSource. Spring Framework. http://www.springframework.org/.

[76] Sun Microsystems. javap. http://java.sun.com/javase/6/docs/technotes/tools/windows/javap.html.

[77] Sun Microsystems. *Mirror API: http://java.sun.com/javase/6/docs/jdk/api/apt/mirror/overview-summary.html.*

[78] Suvée, D., Vanderperren, W., and Jonckers, V. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *2nd International Conference on Aspect-Oriented Software Development* (2003), ACM, pp. 21–29.

[79] Tarr, P., Ossher, H., Harrison, W., and Sutton, J. S. M. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of International Conference on Software Engineering (ICSE)* (1999), IEEE Computer Society Press, pp. 107–119.

[80] Tekinerdogan, B. ASAAM: Aspectual Software Architecture Analysis Method. In *Working IEEE/IFIP Conference on Software Architecture* (2004), IEEE Computer Society, pp. 5–14.

[81] Ubayashi, N., Masuhara, H., and Tamai, T. An AOP Implementation Framework for Extending Join Point Models. In *Workshop on Reflection, AOP, and Meta-Data for Software Evolution* (Oslo, June 2004), pp. 71–81.

[82] Usui, Y., and Chiba, S. Bugdel: An Aspect-Oriented Debugging System. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)* (Taipei, Taiwan, December 2005), IEEE Computer Society, pp. 790–795.