UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL

SEPARACIÓN DINÁMICA DE ASPECTOS INDEPENDIENTE DEL LENGUAJE Y PLATAFORMA MEDIANTE EL USO DE REFLEXIÓN COMPUTACIONAL

Presentada por

Luis Vinuesa Martínez

para la obtención del título de Doctor por la Universidad de Oviedo

Dirigida por el

Profesor Doctor D. Francisco Ortín Soler

Oviedo, 5 de octubre de 2007

Resumen

El principio de la Separación de incumbencias o competencias se fundamenta en separar las partes principales de una aplicación de aquellas con un concepto o propósito especial típicamente ortogonal a la funcionalidad principal, construyendo las aplicaciones finales mediante la unión del código de su funcionalidad más el de sus incumbencias de dominio específico.

El Desarrollo de Software Orientado a Aspectos (DSOA) es una de las aproximaciones a este principio. El DSOA ofrece soporte directo en el lenguaje para modularizar incumbencias que cortan transversalmente al código de la funcionalidad básica de la aplicación. Separando esta funcionalidad de los *aspectos* ortogonales, el código de la aplicación no estará entremezclado, siendo más fácil de mantener, depurar y modificar. Ejemplos de incumbencias transversales típicas son la persistencia, autenticación, *logging*, etc.

La mayoría de los sistemas que ofrecen DSOA son estáticos: una vez que la aplicación se ha generado sus aspectos no se pueden adaptar en tiempo de ejecución. Sin embargo, en determinados escenarios es necesario poder adaptar una aplicación en ejecución en respuesta a cambios del entorno. Además, pueden surgir nuevos requerimientos cuando el sistema está ejecutándose, no siendo posible detenerlo.

Existe una serie de sistemas que ofrecen DSOA de forma dinámica, pero restringen la forma en que se pueden adaptar las aplicaciones, y muchos de ellos no ofrecen una adaptabilidad plena en tiempo de ejecución. Adicionalmente, la mayoría de los sistemas existentes (ya sean dinámicos o estáticos) presentan una restricción muy importante como es la dependencia del lenguaje: sólo se puede utilizar un lenguaje fijado por el sistema.

En esta Tesis Doctoral se presenta un sistema DSOA que ofrece una adaptación dinámica plena sin sufrir dependencias de una plataforma específica, permitiendo la utilización de diversos lenguajes de programación. La reflexión computacional de un sistema permite modificar su propia semántica (comportamiento) en ejecución, pudiendo así adaptar su comportamiento ante sucesos no previstos en el diseño.

La utilización de una máquina abstracta aporta múltiples beneficios al desarrollo de nuestro sistema, destacando la independencia de la plataforma y lenguaje de programación.

Mediante un mecanismo de reflexión computacional basado en un protocolo de meta-objeto (MOP) utilizando técnicas de instrumentación de código aplicadas a una máquina virtual comercial, se obtiene reflexión computacional dinámica independiente del lenguaje y plataforma, consiguiendo así un sistema que soporta DSOA de forma dinámica e independiente del lenguaje y de la plataforma. La utilización de una plataforma profesional estándar ofrece una integración con un elevado número de lenguajes, herramientas y sistemas reales, así como un rendimiento adecuado.

Palabras Clave

Desarrollo de Software Orientado a Aspectos, Separación de Incumbencias, Reflexión Computacional, Máquina Abstracta, Dinamismo, Independencia del Lenguaje, Independencia de la Plataforma.

Abstract

The Separation of Concerns (SoC) principle is based on modularizing crosscutting concerns separate from the main application functionality. Final applications are built joining its functional components plus different reusable domain-specific aspects that are commonly orthogonal to the application functionality.

The Aspect Oriented Software Development (AOSD) is a well-known approach that follows the SoC principle. AOSD offers a language support to separately modularize concerns that crosscut the applications main functionality. The code of each aspect will not be tangled, nor spread out, all over the application, being easier to maintain, debug and modify. Typical examples of crosscutting concerns are persistence, authenticating, logging or tracing.

Most systems that offer AOSD are static. This means that, once the application has been built, their aspects cannot latter be adapted at runtime. Nevertheless, there are scenarios where it is necessary to adapt an application that is being executed, in response to events emerged at runtime. In addition, new requirements can arise when the system is running, not being possible to stop the application.

There are existing systems that offer AOSD in a dynamic way, but they restrict the form applications can be adapted at runtime. Many of them do not offer a complete adaptability of running applications. Additionally, most of AOSD systems (whether they are dynamic or static) are language dependent. This important lack implies that only one fixed programming language can be used.

In this PhD thesis a dynamic AOSD system that overcomes the previously mentioned drawbacks is presented. This system allows the use of multiple programming languages, offering a complete dynamic AOSD Separation of Concerns. Computational reflection has been the technique used to obtain dynamic modification of system semantics (behaviour). Our system is capable of adapting the behaviour of applications in response to runtime-emerging aspects, unpredictable at design time.

The use of an abstract machine has produced many benefits in the development of our dynamic AOSD system, emphasizing the independence of the platform and programming language. Applying code instrumentation techniques to a commercial virtual machine, we have implemented an efficient Meta-Object Protocol (MOP) that offers computational reflection. Since this MOP has been designed at the abstract machine

level, dynamic adaptation of programs is achieved in a platform and language neutral way. The use of a standard professional platform offers a real integration with many programming languages, tools, frameworks and systems, offering an appropriate performance.

Keywords

Aspect Oriented Software Development, Separation of Concerns, Computational Reflection, Abstract Machine, Dynamism, Language Independence, Platform Independence.

Tabla de Contenidos

Tabla de Figuras

CAPÍTULO 1
Introducción

A lo largo de este capítulo se describen los principales objetivos buscados en el desarrollo de esta Tesis Doctoral, estableciendo un marco de requisitos generales a cumplir para, posteriormente, demostrar su viabilidad mediante la creación y evaluación de un prototipo. Posteriormente se presenta la organización de la memoria, estructurada tanto en secciones como en capítulos.

1.1 Introducción

La Programación Orientada a Objetos (POO) es hoy en día el paradigma que se utiliza para la mayoría de los nuevos proyectos de desarrollo de software. De hecho la POO ha demostrado su fuerza a la hora de modelar la funcionalidad básica dominante del sistema (normalmente la funcionalidad original para la que surge la aplicación). Sin embargo no es capaz de tratar de forma adecuada aspectos que no forman parte de la funcionalidad básica o dominante del sistema, encontrándose éstos en ocasiones diseminados por muchos módulos, a menudo sin relación entre ellos, con el consiguiente problema de pérdida de claridad en el código (lo que afecta negativamente a la calidad del software). Se puede afirmar por lo tanto que las técnicas tradicionales no gestionan bien la "separación de incumbencias" para determinados aspectos que no forman parte de la funcionalidad básica del sistema [Tarr99].

Surge así el principio de la separación de incumbencias[1] –*Separation of Concerns*–(SoC) [Parnas72] [Dijkstra76] [Hürsch and Lopes95] con el objetivo de conseguir que los distintos componentes de una aplicación puedan ser identificados en el diseño, en el código y puedan ser tratados explícitamente de forma individual. De esta forma cada función del sistema podrá ser fácilmente comprendida, modificada, añadida y reutilizada.

El principio de la SoC separa los algoritmos principales de una aplicación de aquellas incumbencias con un propósito especial (típicamente ortogonal a la funcionalidad principal), construyendo las aplicaciones finales mediante la composición del código de su funcionalidad principal más el de sus incumbencias de dominio específico.

[1] También es traducida por separación de competencias o de intereses. En este texto se utilizarán indistintamente los términos incumbencia y competencia.

El Desarrollo de Software Orientado a Aspectos[2] (DSOA) – *Aspect Oriented Software Development*– [Kiczales97] es una de las aproximaciones existentes de este principio. El DSOA ofrece soporte directo en el lenguaje para modularizar incumbencias que cortan transversalmente al código de la funcionalidad básica de la aplicación. Separando este código del de los aspectos [Kiczales97] transversales, el código de la aplicación no estará entremezclado, siendo más fácil de mantener, depurar y modificar. Ejemplos de incumbencias trasversales típicas son la persistencia, autenticación, *logging*, etc.

La mayoría de los sistemas que ofrecen DSOA son estáticos: una vez que la aplicación se ha generado no es posible adaptar sus aspectos en tiempo de ejecución. Sin embargo, en determinados escenarios es necesario poder adaptar una aplicación en ejecución en respuesta a cambios del entorno [Popovici01] [Zinki97] [Segura–Devillechaise03] [Matthijs97]. Además, pueden surgir nuevos requerimientos cuando el sistema está ejecutándose, no siendo posible detenerlo.

En sistemas que usan tejido dinámico de aspectos, la funcionalidad básica permanece separada de los aspectos en todo el ciclo de vida del software, incluso en la ejecución del sistema. El código resultante es más adaptable y reutilizable, y los aspectos y la funcionalidad básica pueden evolucionar de forma independiente [Pinto02].

Existe una serie de sistemas que ofrecen DSOA de forma dinámica, pero restringen la forma en que se pueden adaptar las aplicaciones, y muchos de ellos no permiten realmente adaptar el sistema en tiempo de ejecución. La mayoría de los sistemas existentes (ya sean dinámicos o estáticos) presentan una restricción muy importante como es la dependencia del lenguaje: sólo se puede utilizar un lenguaje fijado por el sistema [Lam02].

Justificamos así la necesidad de estudiar las alternativas en la creación de un sistema que ofrezca soporte al DSOA de forma completamente dinámica, independiente del lenguaje y de la plataforma, sin imponer restricciones en la forma de adaptar a las aplicaciones. El dinamismo del sistema, junto a la no existencia de restricciones, permitiría adaptar una aplicación a contextos surgidos en tiempo de ejecución, imprevistos en tiempo de desarrollo. Además, su independencia del lenguaje permitiría adaptar cualquier aplicación, pudiéndose beneficiar de ello las aplicaciones ya implementadas.

A lo largo de esta Tesis estudiaremos las distintas alternativas, y enunciaremos otras, para crear un sistema que ofrezca soporte al DSOA, en el que se satisfagan todos los objetivos enunciados someramente en el párrafo anterior.

1.2 Objetivos

Enunciaremos los distintos objetivos generales propuestos para la creación del sistema que ofrezca separación dinámica de aspectos independiente del lenguaje y plataforma previamente mencionado, posponiendo hasta el siguiente capítulo la especificación formal del conjunto integral de requisitos impuestos a nuestro sistema.

[2] Inicialmente se denominaba Programación Orientada a Aspectos, POA. En este texto se utilizarán indistintamente.

1.2.1 Dinamismo

El sistema aquí propuesto debe ser completamente dinámico, es decir, debe ser capaz de adaptar sin restricciones una aplicación que esté ejecutándose sin necesidad de interrumpir su ejecución.

En contraposición a muchos de los sistemas existentes que imponen restricciones a la adaptación de las aplicaciones tales como la no posibilidad de desactivación o eliminación de un aspecto una vez tejido, o la necesidad de conocer los aspectos que van a adaptar a una aplicación en el momento de comenzar a ejecutarla, debe ser posible activar y desactivar aspectos en cualquier momento de la ejecución de la aplicación a adaptar, y no es necesario que estos aspectos sean conocidos durante la fase de desarrollo de la aplicación.

De esta manera se consigue un verdadero dinamismo, permitiendo adaptar la aplicación a nuevos requerimientos no conocidos durante su desarrollo.

1.2.2 Independencia del Lenguaje

Tanto las aplicaciones a adaptar como los aspectos que las adapten podrán estar implementados en cualquier lenguaje de programación, sin restricciones. La mayoría de los sistemas existentes en la actualidad son dependientes del lenguaje (Java generalmente), imponiendo la necesidad de que tanto el lenguaje utilizado para implementar la aplicación base, como el utilizado para implementar los aspectos sean los fijados por el sistema.

Nuestro sistema debe permitir implementar las aplicaciones y los aspectos en cualquier lenguaje. De hecho debe permitir que un aspecto implementado en un lenguaje A adapte a una aplicación implementada en cualquier otro lenguaje B.

Gracias a esta característica, la elección del lenguaje de programación estará únicamente determinada por la adecuación del mismo al problema a resolver y las preferencias del programador. Igualmente se posibilita aprovechar el código previamente implementado (código heredado) independientemente del lenguaje en el que lo haya sido.

1.2.3 Independencia de la Plataforma

El sistema a desarrollar debe ser completamente independiente de la plataforma (entendiendo plataforma como la unión del Hardware y el Sistema Operativo) donde se ejecute.

Esta característica permitirá ejecutar el sistema en cualquier plataforma, ampliando así el conjunto de potenciales usuarios. De igual forma facilitará el despliegue del sistema al no tener que mantener diferentes versiones del mismo para las diferentes plataformas donde se ejecute.

1.3 Organización de la Memoria

A continuación mostramos la estructura de este documento, agrupando los capítulos en secciones con un contenido acorde.

1.3.1 Introducción y Requisitos del Sistema

En este capítulo narramos la introducción, objetivos y organización de esta memoria. En el capítulo siguiente establecemos el conjunto de requisitos impuestos a nuestro sistema, que serán utilizados principalmente para:

- Evaluar las aportaciones y carencias de los sistemas estudiados en la siguiente sección.
- Fijar la arquitectura global de nuestro sistema.
- Evaluar los resultados del sistema propuesto, comparándolos con otros sistemas existentes estudiados.

1.3.2 Sistemas Existentes y Técnicas Estudiados

En esta sección se lleva a cabo un estudio del estado del arte de los sistemas similares al buscado y de diferentes técnicas que pueden utilizarse para implementarlo. En el capítulo 3 presentamos el principio de la Separación de Incumbencias – *Separation of Concerns* – (SoC) y diversas aproximaciones existentes para conseguir los objetivos enunciados en este principio, entre ellas el Desarrollo de Software Orientado a Aspectos (DSOA). Un estudio detallado del DSOA se lleva a cabo en el capítulo 4.

El capítulo 5 presenta un estudio pormenorizado de una serie de sistemas que ofrecen soporte a la Programación Orientada a Aspectos (POA), explicando sus características principales y evaluando sus fortalezas y debilidades en relación a los requisitos previamente establecidos para el sistema.

Un estudio sobre las máquinas abstractas es presentado en el capítulo 6, especificando sus características, ámbitos de aplicación y las aportaciones que puede tener su uso para construir el sistema propuesto. También se realiza un estudio de una serie de máquinas abstractas utilizadas empresarialmente identificando sus aportaciones y carencias.

La técnica de la reflexión, sus conceptos principales y distintas clasificaciones, son introducidos en el capítulo 7. Finalmente, la evaluación de múltiples sistemas reflectivos, sus aportaciones y limitaciones frente a los requisitos impuestos, son presentadas en el capítulo 8.

1.3.3 Diseño del Sistema

En esta sección se introduce el sistema propuesto basándose en los requisitos fijados y los sistemas estudiados. La arquitectura global del sistema y su descomposición en distintos componentes es presentada en el capítulo 9. En los capítulos siguientes presentamos de forma detallada los distintos componentes del sistema.

En el capítulo 10 se identifica el empleo de una máquina abstracta como beneficioso para la implementación del sistema, se fijan las características que debe cumplir para que el sistema satisfaga los requisitos generales y se procede a la elección de una en concreto.

A continuación, en el capítulo 11, se procede a especificar el lenguaje de definición de puntos de corte, identificando previamente el conjunto de puntos de enlace que soportará el sistema.

En el capítulo 12 se selecciona la reflexión como una técnica útil para desarrollar el sistema propuesto y detallamos los pasos necesarios para obtenerla.

El servidor de aplicaciones, elemento central del sistema, es especificado en el capítulo 13 y, en el capítulo 14, definimos el Framework de Aspectos. En el capítulo 15 describimos la implementación de un prototipo del sistema propuesto.

1.3.4 Aplicaciones, Evaluación, Conclusiones y Trabajo Futuro

Un conjunto de posibles aplicaciones prácticas de nuestro sistema se presenta en el capítulo 16. La evaluación del sistema presentado en esta Tesis, comparándolo con otros existentes, es llevada a cabo en el capítulo 17 bajo los requisitos establecidos al comienzo de éste.

En el capítulo 18 se muestran las conclusiones globales de la investigación llevada a cabo y las principales aportaciones realizadas frente a los sistemas estudiados. Así mismo se presentan las futuras líneas de investigación y el trabajo a realizar a partir de los resultados obtenidos.

1.3.5 Apéndices

Como apéndices de esta memoria se presentan:

- El apéndice A, que contiene la especificación del lenguaje de puntos de corte definido en el sistema.
- El apéndice B, que constituye el conjunto de referencias bibliográficas utilizadas en este documento.

CAPÍTULO 2
REQUISITOS

En este capítulo especificaremos los requisitos del sistema buscado en esta Tesis. Se identificarán los distintos requisitos y se describirán brevemente. En un sistema que ofrece orientación a aspectos hay que tener en cuenta una serie de elementos como son la plataforma sobre la que se implementa el sistema, las aplicaciones que pueden hacer uso de él, los aspectos que utilizan el sistema para adaptar a las aplicaciones y el propio sistema en sí. Por lo tanto, hemos agrupado los requisitos en base a estos cuatro elementos.

La especificación de los requisitos del sistema llevada a cabo en este capítulo tiene por objetivo la validación del sistema diseñado así como el estudio de los distintos sistemas existentes similares al buscado. Los requisitos del sistema nos permiten reconocer los puntos positivos de sistemas reales –para su futura adopción o estudio – así como cuantificar sus carencias y justificar determinadas modificaciones y/o ampliaciones.

La especificación de los requisitos nos ayudará a establecer los objetivos buscados en el diseño de nuestro sistema, así como a validar la consecución de dichos objetivos una vez que éste haya sido desarrollado.

2.1 Requisitos Generales del Sistema

El objetivo de esta Tesis es crear un *sistema que ofrezca soporte para el desarrollo de software orientado a aspectos de forma dinámica y completamente independiente del lenguaje y de la plataforma*. En base a esta definición establecemos una serie de requisitos generales que debe cumplir el sistema. En los apartados siguientes examinaremos los requisitos generales del sistema que no entran dentro de las categorías de la plataforma, aplicaciones o aspectos, que veremos de forma individual en las siguientes secciones.

2.1.1 Sistema Completamente Dinámico

Existen sistemas estáticos, que permiten la modificación de la aplicación base en un momento anterior a su ejecución, y sistemas dinámicos, que permiten la modificación (en mayor o menor medida) de la aplicación durante la ejecución de ésta. El sistema que queremos diseñar deberá ser totalmente dinámico, permitiendo la modificación

de la aplicación base durante su ejecución, sin imponer restricción alguna en su adaptación.

Existen sistemas que, pese a soportar cierto dinamismo, únicamente permiten activar aspectos durante la ejecución de la aplicación base, pero no permiten desactivarlos. En general, en estos casos, se impone además la restricción de que los aspectos deban ser conocidos durante el diseño. Nuestro sistema deberá poder activar y desactivar en ejecución cualquier aspecto, sin que éste haya sido previsto previamente durante el diseño.

Gracias a esta característica del sistema, una aplicación que esté ejecutándose dentro de él podrá adaptarse, de forma dinámica, a cambios en el entorno de ejecución, o ante nuevos requerimientos sobrevenidos durante su ejecución. Todo esto se deberá poder realizar sin tener que detener la ejecución de la aplicación en ningún momento.

2.1.2 Independencia del Lenguaje

El sistema debe ser independiente del lenguaje de programación, tanto de la aplicación base como de los posibles aspectos que vayan a modificarla; por lo tanto, su diseño no puede enfocarse a peculiaridades propias de un lenguaje de programación concreto.

Cualquier aplicación base y cualquier aspecto podrán ser implementados en cualquier lenguaje. Gracias a este requerimiento, la adopción del sistema por parte de una empresa puede realizarse de forma rápida y productiva desde el primer momento, ya que los programadores experimentados en un lenguaje pueden seguir utilizando dicho lenguaje para implementar las aplicaciones y los aspectos.

De igual manera, los programas que hayan sido desarrollados previamente podrán ser utilizados en el sistema, independientemente del lenguaje en el que estén implementados.

2.1.2.1 Interacción entre distintos lenguajes de programación

El sistema no sólo debe permitir trabajar con cualquier lenguaje de programación sino que no debe limitar la interacción de componentes y aspectos a aquellos implementados en el mismo lenguaje. Es decir, debe ser posible modificar el comportamiento de una aplicación base implementada en un lenguaje A, por medio de aspectos implementados en lenguajes B o C (de hecho, debe ser posible modificar el comportamiento mediante diversos aspectos, unos implementados en un lenguaje B, y otros en un lenguaje C, de forma simultánea).

2.1.3 Sistema de Propósito General

El diseño del sistema se debe realizar de tal forma que no esté orientado a la resolución de un tipo de problema concreto. Existen implementaciones que describen una plataforma para abordar un problema específico como la persistencia, la autenticación o la intercomunicación de aplicaciones distribuidas (son conocidos como sistemas de propósito específico). Este tipo de plataforma es descrito de una forma específica (y opti-

mizada) para la resolución del problema planteado y no es comúnmente apto para resolver otro tipo de problemas.

El sistema aquí planteado debe ser capaz de resolver cualquier tipo de problema, es decir, debe ser de propósito general. No se trata de implementar una solución para la mayoría de los problemas existentes, sino una que permita resolver de forma flexible cualquier problema que pudiese surgir.

2.1.4 Amplio Conjunto de Puntos Enlace

El conjunto de puntos de enlace de un sistema define los diferentes puntos de la aplicación base donde se podrá añadir comportamiento (por ejemplo en la invocación a un método o el acceso de lectura a un atributo). Cuanto mayor sea el conjunto de puntos de enlace que soporte el sistema mayor será su potencia y las posibilidades de uso que ofrezca.

Muchos sistemas dinámicos ofrecen un conjunto de puntos de enlace muy reducido en comparación con los sistemas estáticos. El sistema a desarrollar debe soportar un conjunto de puntos de enlace amplio, similar al que soporta AspectJ [AspectJ], que es un estándar de facto y patrón de comparación de los sistemas que ofrecen Programación Orientada a Aspectos (POA).

2.1.5 Expresividad del Lenguaje de Definición de Puntos de Corte

Un punto muy importante en la potencia de los sistemas que ofrecen POA es el modo de expresar los puntos de corte. Aspectos tales como la facilidad de uso y la riqueza expresiva son muy importantes.

Nuestro sistema debe soportar un lenguaje de definición de puntos de corte que sea sencillo de utilizar y, por tanto, de aprender, y que a la vez sea potente. Debe gozar de una gran expresividad, que permita la selección de los puntos de enlace de forma explícita, es decir, nombrando completamente el punto de enlace, y también de forma implícita, mediante el uso de expresiones regulares. De igual forma debe permitir construir sentencias condicionales complejas a partir de otras sentencias simples.

2.2 Requisitos de la Plataforma

En esta sección se engloban los requisitos que están relacionados con la plataforma sobre la que se debe implementar el sistema propuesto.

2.2.1 Independencia de la Plataforma

El sistema no debe imponer ninguna restricción en referencia a la plataforma (entendiendo plataforma como la unión del *Hardware* y Sistema Operativo) sobre la que se ejecute. Debido a este requisito el sistema no podrá hacer uso de ninguna característica particular de una plataforma Hardware o de un Sistema Operativo concreto donde vaya a ejecutarse.

Con esta restricción se pretende que el sistema pueda ejecutarse sobre cualquier plataforma existente, con lo que se facilita su adopción por parte de los usuarios.

2.2.2 Plataforma Comercial Ampliamente Difundida

Nuestro sistema debe ser desarrollado sobre una plataforma comercial con una amplia base de usuarios. Por comercial debe entenderse el que se utilice efectivamente en el mundo empresarial, en contraposición a otras plataformas, de investigación, que cuentan con poca o nula presencia en los sistemas informáticos de las empresas.

Si deseamos crear un sistema que pueda ser utilizado de forma real es necesario que se desarrolle sobre una plataforma con aceptación, de esta forma la base de usuarios potenciales será amplia. Además se beneficia de la utilización de herramientas, compiladores, APIs, componentes, etc. que existan sobre la plataforma.

2.2.3 Uso de Plataforma con Buen Rendimiento en Tiempo de Ejecución

Un objetivo de esta Tesis es crear un sistema eficiente, que pueda ser utilizado para resolver problemas comerciales. Un punto muy importante a la hora de evaluar un sistema es su rendimiento. Por muy buenas prestaciones con respecto a capacidades de adaptación, flexibilidad o potencia que ofrezca un sistema, si el rendimiento en ejecución es bajo es muy posible que no se pueda aplicar a muchos problemas que requieren un buen desempeño.

Puesto que la obtención de la adaptación dinámica que buscamos en nuestro sistema conlleva un coste en el rendimiento (respecto al sistema original que no soporta dicha adaptación dinámica), es necesario que la plataforma sobre la que se desarrolle tenga un buen rendimiento, minimizando así el impacto de este coste.

2.2.4 Portabilidad del Sistema – Sistema Estándar

El entorno sobre el que se implemente nuestro sistema debe ser completamente estándar, es decir, nuestro sistema no puede imponer restricciones o modificaciones al entorno. Por ejemplo, si se implementa haciendo uso de una máquina virtual, ésta no debe ser modificada bajo ningún concepto (lo mismo es aplicable para un sistema operativo, lenguaje de programación, etc.).

Gracias a esto, el sistema podrá implantarse en cualquier lugar donde se encuentre disponible el entorno sobre el que se implementa, sin tener que realizar ninguna adaptación ni en el sistema ni en el propio entorno.

2.3 Requisitos sobre las Aplicaciones

En esta sección explicamos los requisitos que están relacionados con las aplicaciones que pueden hacer uso del sistema, es decir, aplicaciones que pueden ser adaptadas por medio del sistema.

2.3.1 Adaptación de Código Binario

Nuestro sistema debe ser capaz de modificar el comportamiento de una aplicación sin disponer de su código fuente.

Gracias a esta imposición, es posible modificar el comportamiento de aplicaciones desarrolladas por terceros y de las que no disponemos de su código fuente (por ejemplo, los conocidos como COTS *Commercial off–the–shelf*, aplicaciones o componentes ya implementados y que se venden para su uso).

2.3.2 No Imposición de Condiciones a las Aplicaciones a ser Adaptadas

Muchos sistemas imponen ciertas condiciones a las aplicaciones para poder ser adaptadas. Hay sistemas que obligan a que la aplicación se enlace con alguna librería concreta del sistema, otros imponen que las clases de la aplicación hereden de alguna clase propia del sistema, otros ponen condiciones respecto a los constructores (por ejemplo que sólo exista el constructor por defecto, o de existir otro tiene que tener una signatura determinada), etc.

Estas condiciones limitan la utilidad de estos sistemas, ya que sólo aquellas aplicaciones que cumplan esas condiciones podrán utilizarse en ellos. Normalmente, esto implica disponer del código fuente de la aplicación y tener que adaptarla para que pueda funcionar en el sistema. Además, se limita la posibilidad de uso de la aplicación de forma directa (sin necesidad de cambiar nada) en otros sistemas.

Nuestro sistema no debe imponer condición alguna a las aplicaciones que vayan a ser adaptadas, de esta forma cualquier aplicación podrá ejecutarse en él sin necesidad de realizar adaptaciones al mismo, lo que implicaría, en el caso de aplicaciones ya implementadas, disponer de su código fuente o limitar el uso del sistema a aquellas aplicaciones que cumplan originalmente las condiciones. De igual forma, una aplicación desarrollada para ejecutarse en el sistema podrá ejecutarse en otro sistema sin necesidad de ser modificada (excepto por los condicionantes que pueda imponer el otro sistema).

2.4 Requisitos sobre los Aspectos

A continuación describimos los requisitos que debe cumplir el sistema con respecto a los aspectos que puedan ejecutarse en el mismo para adaptar a las aplicaciones.

2.4.1 Adaptación Simultánea de Aplicaciones

Existen sistemas que restringen a un único aspecto la modificación del comportamiento de la aplicación base en un punto de enlace determinado. Debido a esto, si en ese punto fuese necesario tener varias funcionalidades transversales a la funcionalidad principal, como pueden ser persistencia, autenticación o una traza, éstas deberían implementarse en el único aspecto o en la aplicación base.

En cualquier caso esta limitación es contraria al paradigma de Orientación a Aspectos puesto que el código permanecería enmarañado y diseminado, anulando los posibles beneficios de su utilización.

Nuestro sistema debe permitir que en cualquier punto de enlace de la aplicación base se pueda asociar un número de aspectos cualquiera, sin límite, de tal forma que se tengan totalmente separadas la funcionalidad principal de la aplicación de la de cada

una de las funcionalidades transversales a la misma y las de éstas últimas entre sí (encontrándose cada funcionalidad en un aspecto individual).

Este requisito conlleva la aparición de un nuevo requerimiento originado por la posibilidad de la existencia de varios aspectos en un mismo punto de enlace.

2.4.1.1 Existencia de un mecanismo de coordinación de múltiples aspectos ligados a un mismo punto de enlace

Como se ha mencionado, el requisito de que el sistema permita el que varios aspectos se asocien a un mismo punto de enlace de la aplicación base, de forma simultánea, hace surgir la cuestión de gestionar el orden de ejecución de dichos aspectos en el momento en que se alcance el punto de enlace al que están asociados.

Existen sistemas que no permiten al usuario tener control alguno sobre el orden de ejecución de los distintos aspectos asociados cuando hay más de uno en el mismo punto de enlace. Estos sistemas, en general, o establecen el orden de ejecución basándose en el orden de enlace de los aspectos o no garantizan ningún orden de ejecución de los mismos, limitando las posibilidades del usuario ya que no puede implementar código que dependa del orden de ejecución de los diferentes aspectos.

Nuestro sistema debe dotar al usuario de un mecanismo que le permita tener control absoluto sobre el orden de ejecución de diversos aspectos, en el caso de que coexistan sobre el mismo punto de enlace, ofreciéndole garantías de que la ejecución se realizará en el orden que se haya solicitado

2.4.2 Aspectos como Aplicaciones

Los aspectos deben ser considerados como aplicaciones dentro del sistema, con todas las posibilidades que tienen éstas y sin ninguna limitación.

Un aspecto modifica el comportamiento de una aplicación base, pero, a su vez, y debido a que tiene que ser una aplicación estándar, sin limitaciones, puede ser, de forma simultánea, la aplicación base para que otro aspecto la modifique a ella.

Gracias a este requerimiento los aspectos pueden beneficiarse del uso de la Programación Orientada a Aspectos como cualquier otra aplicación

2.4.3 Reutilización de los Aspectos

Los aspectos implementados en nuestro sistema deben poder reutilizarse, tanto dentro del sistema, como por parte de otros sistemas externos, sin tener que ser modificados. Este requisito impone dos requisitos derivados que se explican a continuación.

2.4.3.1 Uso de lenguaje estándar, sin extensiones, para definir los aspectos

Existen sistemas que definen su propio lenguaje de programación completamente nuevo para definir sus aspectos. Esto impide la reutilización de los aspectos implementados en estos sistemas en cualquier otro sistema que no soporte el lenguaje defini-

do, además de no respetar el requisito 2.1.2 Independencia del Lenguaje, al imponer la obligatoriedad de utilizar cierto lenguaje para definir los aspectos.

Otros sistemas utilizan un lenguaje de programación existente añadiéndole extensiones al mismo. Esto tiene el mismo inconveniente que el caso anterior, por lo que cualquier otro sistema no podrá utilizar los aspectos definidos en este lenguaje, al no entender el programa codificado en el lenguaje con las extensiones añadidas (necesitaría un compilador que entendiese las extensiones del lenguaje). También invalida el requisito 2.1.2, puesto que, para permitir al usuario la posibilidad de utilizar cualquier lenguaje para implementar la aplicación o los aspectos, habría que definir extensiones para todos los lenguajes de programación que pudiesen ser utilizados por los usuarios (y sus respectivos compiladores que entendiesen estas extensiones), lo cual no es factible.

Nuestro sistema debe hacer uso de lenguajes estándar, ya definidos previamente, y sin realizar ninguna adición al lenguaje, de tal forma que un aspecto implementado para el sistema pueda ser entendido (compilado) por cualquier compilador del lenguaje en el que se haya implementado.

2.4.3.2 Separación del código del aspecto y de los puntos de corte

Existen sistemas en los que es preciso señalar en el código del propio aspecto los puntos de corte con la aplicación base. Es decir, especifican dónde y de qué forma van a modificar los aspectos a la aplicación base. Esto implica que existe dependencia entre el aspecto y la aplicación base a la que va a modificar, limitando las posibilidades de reutilización del aspecto de forma directa, sin tener que modificarlo, para otras aplicaciones base (y, por supuesto, en otros sistemas que definan los puntos de corte de otra manera).

Nuestro sistema deberá separar completamente el código de los aspectos de la especificación de los puntos de corte entre éstos y la aplicación base, de tal forma que si se desea reutilizar un aspecto con otra aplicación base sólo haya que definir los puntos de corte correspondientes. De la misma manera, al no contener el aspecto nada en su código que especifique los puntos de corte, podrá ser reutilizado en otro sistema de forma directa (obviamente puede ser necesario adaptarlo a las características o restricciones del nuevo sistema).

CAPÍTULO 3
SEPARACIÓN DE INCUMBENCIAS *(SEPARATION OF CONCERNS)*

La Ingeniería del Software ha utilizado el principio de la separación de incumbencias (también conocida como separación de competencias) (SoC) –Separation of Concerns– [Parnas72][Dijkstra76][Hürsch and Lopes95] para gestionar la complejidad del desarrollo de software mediante la separación de las funcionalidades principales de la aplicación de otras partes con un propósito específico, ortogonales a la funcionalidad principal (autenticación, administración, rendimiento, gestión de memoria, logging, etc.). La aplicación final se construye con el código de las funcionalidades principales más el código de propósito específico. Los principales beneficios de esta aproximación son:

- Un nivel de abstracción más alto, debido a que el desarrollador se puede centrar en incumbencias concretas y de forma aislada.
- Una mayor facilidad a la hora de entender la funcionalidad de la aplicación. El código que implementa ésta no está entremezclado con código de otras incumbencias.
- Una mayor reusabilidad al haber un menor acoplamiento.
- Una mayor mantenibilidad del código, al ser éste menos complejo.
- Una mayor flexibilidad en la integración de componentes.
- Un incremento de la productividad en el desarrollo.

La base de una buena SoC es la capacidad de identificar y manejar de forma individual los distintos componentes del sistema tanto en la fase de diseño como en la codificación.

A lo largo de los años se han desarrollado aproximaciones y técnicas para conseguir SoC, que han ido evolucionando con el objetivo de hacer frente a las características cambiantes, ámbito y complejidad de las aplicaciones software. En términos generales estas propuestas ofrecen soluciones para la especificación, asociación y composición de incumbencias ortogonales a la funcionalidad básica a nivel de código. Estas aproximaciones están todavía en una etapa de propuesta o de experimentación. Adicionalmente ofrecen soporte limitado, o simplemente no lo ofrecen, a la definición, y asociación, dinámica de incumbencias.

Algunas de las técnicas existentes son: Separación multidimensional de incumbencias [Tarr99], lenguajes de programación de MOP reflectivos [Kiczales91] o programación orientada a aspectos (POA) [Kiczales97]

A continuación vamos a ver brevemente algunas de estas técnicas y otras que se consideran importantes.

3.1 Filtros de Composición

El origen de los **filtros de composición** – *composition filters*–[Aksit92] [Composition Filters] está motivado por la necesidad de aumentar la adaptabilidad del paradigma de la orientación a objetos. Por lo tanto surge como una alternativa a las limitaciones del modelo de objetos convencional, tales como las anomalías de herencia (herencia múltiple) o la carencia de mecanismos adecuados para separar la funcionalidad de la coordinación de mensajes [Pryor02].

Los filtros de composición constituyen una extensión del modelo de programación orientada a objetos convencional añadiendo los filtros (en forma de *wrappers* – envoltorios; código que se combina con otra pieza de código y que determina cómo se ejecuta la primera. El *wrapper* actúa como una interfaz entre el código envuelto y el que lo llama [Foldoc98]) a la abstracción de objeto –[Aksit92] [Composition Filters], donde los objetos pueden asociarse a uno o más filtros que deben ser modulares y ortogonales entre sí [Bergmans94].

Un ejemplo comparativo de adaptabilidad de objetos mediante filtros de composición puede ser la modificación del funcionamiento de una cámara de fotos. Si las condiciones de luz son pobres y el cuerpo a fotografiar se encuentra alejado, podremos modificar el funcionamiento de la cámara (objeto) añadiéndole filtros de color y lentes de aumento (filtros de composición); estas dos extensiones son modulares porque no es necesario modificar el funcionamiento de la cámara para acoplarlas, y son ortogonales porque sus funcionalidades son independientes entre sí.

En este modelo, los objetos se definen como una instancia de una clase que consiste en métodos, variables de instancia o atributos, condiciones, objetos internos y externos y uno o más filtros. El objeto en este modelo se compone de dos partes: un núcleo u *objeto interno* y una *capa de interfaz*. El objeto interno se puede ver como un objeto normal (en el sentido tradicional), y la capa de interfaz envuelve al objeto interno y gestiona los mensajes entrantes y salientes (Figura 1).

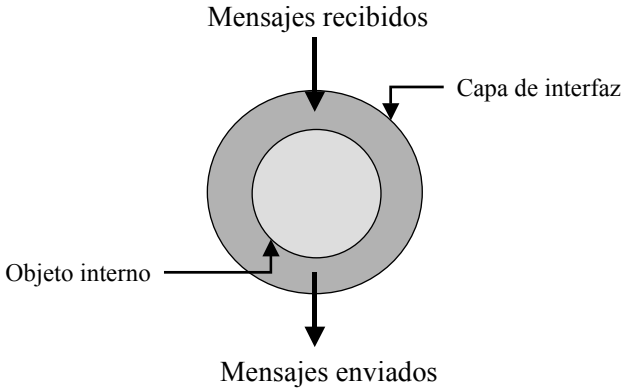

Figura 1: Objeto en el modelo de filtros de composición

Los filtros son objetos instanciados de clases de tipo filtro. El propósito de los filtros es el manejo y control del envío y recepción de mensajes. Especifican condiciones de aptitud o rechazo de los mensajes y determinan así la acción resultante. Cada filtro se puede programar en cada clase que lo utilice. El sistema se asegura de que un mensaje sea procesado por el filtro apropiado antes de que el método correspondiente sea ejecutado: cuando se recibe tiene que ser filtrado por un filtro de entrada y cuando se envía por un filtro de salida.

En términos generales, cuando un objeto envía un mensaje a otro objeto, el mensaje es filtrado por los filtros de salida del objeto emisor, y posteriormente por los filtros de entrada del objeto receptor. Cada filtro decide qué hacer con el mensaje, aceptarlo o rechazarlo. Cuando se acepta el mensaje se invoca a un método, mientras que un mensaje rechazado pasa al siguiente filtro.

Se han desarrollado filtros para aplicaciones con delegación en la herencia, tiempo real [Aksit94], manejo de errores y sincronización de procesos [Bergmans94].

Los filtros aportan al programador la oportunidad de poder atrapar el envío y recepción de mensajes y poder llevar a cabo determinadas acciones antes de que el método sea realmente ejecutado. Se separan así dos niveles de abstracción, el de mayor nivel (método) y el dependiente de la implantación o de bajo nivel (filtro). Un ejemplo de la distinción clara de estos dos niveles puede ser una aplicación en tiempo real.

La forma en la que se consigue una aplicación flexible es permitiendo la modificación de la semántica de los pasos de mensajes y de la ejecución de los métodos, mediante el uso de filtros de composición. La flexibilidad está pues limitada al paso y recepción de mensajes. Además pueden presentarse inconvenientes cuando los filtros no son ortogonales entre sí [Pryor02].

3.2 Programación Adaptable

El objetivo de la **programación** adaptable –*adaptative programming*– es expresar la intención general de un programa sin tener que especificar todos los detalles de las estructuras de los objetos [Lieberherr96]. La programación adaptable, de este modo, eleva el nivel de abstracción de la programación orientada a objetos (POO). El principal

caso práctico de programación adaptable es el proyecto Demeter [Demeter], el cual se basa en el desarrollo de software mediante un método adaptable.

Utilizando el método adaptable las aplicaciones se diseñan con un mayor nivel de abstracción y un menor formalismo que en la POO. Inicialmente se identifican las clases y su comportamiento básico, sin necesidad de definir su estructura interna. Los denominados **patrones de propagación** *(propagation pattern*s) definen las relaciones entre las clases, sus operaciones básicas y un conjunto de restricciones, pero sin llegar a programar todo su comportamiento.

Las partes en las que se divide un patrón de propagación son:

- Signatura de la operación o prototipo de la computación a implementar
- Especificación de un camino. Indica el camino en el grafo de relaciones que va a seguir el cálculo de la operación.
- Fragmento de código. Se especifica el código propio de la realización de la operación.

Una vez definido el grafo de clases y los patrones de propagación, se dispone de una aplicación a un nivel de abstracción tal que permite su formalización en distintas aplicaciones finales (por ejemplo una aplicación en lenguaje C++). Esto se consigue mediante lo que se denomina personalización (*customization)*, de forma que distintas personalizaciones de un grafo dan lugar a distintas aplicaciones.

Por lo tanto en la programación adaptable se distinguen dos niveles. El primero es un alto nivel de abstracción que especifica clases y relaciones entre ellas. El segundo es el que, partiendo de esta especificación, implementa de forma refinada su comportamiento final mediante una personalización.

El equipo involucrado en el grupo Demeter ha estado colaborando con el grupo de Gregor Kiczales, y fruto de esta colaboración surgió la idea de la **Programación Orientada a Aspectos** (POA) [Kiczales97]. En la actualidad el grupo Demeter está trabajando en la integración de la programación adaptable con la POA, habiendo desarrollado los sistemas DJ [DJ] y Demeter AspectJ [DAJ], que integran ambas ideas.

3.3 Separación Multidimensional de Incumbencias

La complejidad en la separación de competencias en un sistema se va incrementando a lo largo de la vida de éste, debido a que van surgiendo nuevas competencias que necesitan ser tratadas. Se puede llegar así al caso extremo en que deba ser posible aplicar cualquier criterio para la descomposición de incumbencias.

La **separación multidimensional de incumbencias** *(Multi–Dimensional Separation of Concern*s) se refiere precisamente a los sistemas dotados de la capacidad de separar de forma incremental, modularizar e integrar aplicaciones software con cualquier tipo y nivel de incumbencias [IBM2000a]. Los principales objetivos de estos sistemas son [Ortin02b]:

1. Permitir la especificación simultanea de cualquier dimensión de incumbencias, sin restricciones de número ni de tipo.

2. Posibilitar la existencia de incumbencias solapadas (no ortogonales) e interactivas (que puedan interactuar entre ellas).
   Estas dos características diferencian este modelo de los estudiados previamente en este capítulo y lo aproxima más al mundo real.
3. Facilitar la "remodularización" de incumbencias, es decir, crear y/o modificar los módulos que manejan ciertas incumbencias sin tener que modificar otros módulos no implicados. Ya hemos dicho al principio que el propio crecimiento del sistema puede provocar la aparición de nuevas incumbencias y cambios en las ya existentes.

Las distintas ventajas de la utilización de este paradigma son:

- Al separarse los distintos aspectos del sistema, se facilita la reutilización de código, no sólo en lo que se refiere a aspectos funcionales de la aplicación sino también las incumbencias adicionales (persistencia, distribución, etc.).
- Aumenta la legibilidad del código, ya que hay una separación y diferenciación de los distintos aspectos del sistema.
- Reduce los impactos debidos a los cambios de requisitos, ya que la posibilidad de añadir nuevas incumbencias y de hacerlo sin modificar otros módulos no relacionados dota al sistema de una mayor flexibilidad antes los cambios.
- Facilita el mantenimiento debido a que los módulos que implementan las incumbencias están perfectamente identificados y separados.
- Aumenta la trazabilidad de una aplicación centrándonos en aquel aspecto específico que deseemos controlar.

Existen diversos sistemas de investigación construidos basándose en este paradigma [ICSE2000][OOPSLA99]. Un ejemplo es *Hyperspaces* [Ossher99], un sistema de separación multidimensional de incumbencias independiente del lenguaje, creado por IBM, así como la herramienta Hyper/J [IBM2000b] que da soporte a *Hyperspaces* en el lenguaje de programación Java [Gosling96].

La principal aportación de los sistemas basados en separación multidimensional de incumbencias, en comparación con los estudiados en este capítulo, es el grado de flexibilidad otorgado: mientras que el resto de sistemas identifican un conjunto limitado de aspectos a describir, en este caso no existen límites.

3.4 Programación Orientada a Aspectos

Como se mencionó anteriormente, existen situaciones en las que los lenguajes orientados a objetos no permiten modelar de forma suficientemente clara las decisiones de diseño tomadas previamente a la implementación. El sistema final se codifica entremezclando el código propio de la especificación funcional del diseño con llamadas a rutinas de diversas librerías encargadas de obtener una funcionalidad adicional (por ejemplo, distribución, persistencia o multitarea). El resultado es un código fuente excesivamente difícil de desarrollar, de entender, y por lo tanto de mantener [Kiczales97].

La **Programación Orientada a Aspectos** (POA) – *Aspect Oriented Programming*– surge en 1997 [Kiczales97], y desde entonces ha suscitado mucho interés por las posibilidades que ofrece. En la POA hay dos términos muy importantes:

- Un componente –*component*– es aquel módulo software que puede ser encapsulado en un procedimiento (un objeto, método, procedimiento o API). Los componentes serán unidades funcionales en las que se descompone el sistema.
- Un **aspecto** –*aspect*– es aquel módulo software que no puede ser encapsulado en un procedimiento. No son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afectan la ejecución o semántica de los componentes. Ejemplos de aspectos son la gestión de la memoria o la sincronización de hilos *(thread*s).

Una aplicación orientada a aspectos es el resultado de adaptar (modificar o ampliar el funcionamiento) los componentes de una aplicación por medio de aspectos. Este proceso se denomina **tejido** – *weave*– y es realizado por el **tejedor de aspectos** –*aspect weaver*–.

En la Figura 2 se puede ver el esquema de funcionamiento de una aplicación orientada a aspectos.

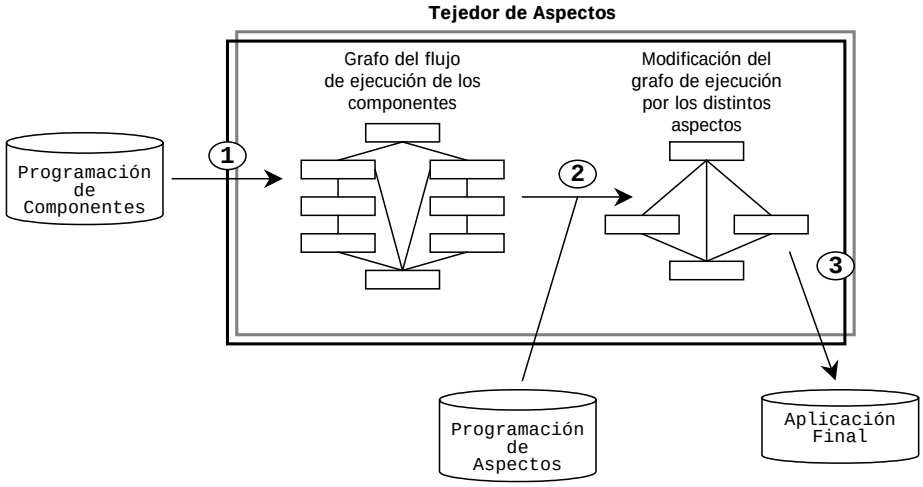

Figura 2: Esquema de la Programación Orientada a Aspectos

1. El tejedor construye un grafo del flujo de ejecución del programa de componentes (esto lo puede hacer a partir del código fuente o del código ya compilado, depende del sistema).
2. A continuación se modifica el grafo anterior realizando las modificaciones o adiciones oportunas (en respuesta a los aspectos).
3. Por último se genera el código final de la aplicación (a partir del nuevo grafo generado, que incluye la funcionalidad de los componentes y de los aspectos).

En la actualidad se está realizando mucho trabajo de investigación sobre esta tecnología y han aparecido muchos sistemas que la soportan, siendo el más importante hasta el momento AspectJ [AspectJ], el cual ha sido utilizado en grandes proyectos con buen resultado. También se ha incorporado a determinados servidores de aplicaciones como Jboss [Jboss] [JBossAOP] o Spring [Spring] [SpringAOP].

La mayoría de los sistemas que ofrecen SoC vistos anteriormente están intentando evolucionar para ser compatibles y complementarios con la POA (por ejemplo la programación adaptable con el sistema Demeter AspectJ [DAJ]).

En el siguiente capítulo se profundiza en esta tecnología.

CAPÍTULO 4
DESARROLLO DE SOFTWARE ORIENTADO A ASPECTOS

El término Programación Orientada a Aspectos – *Aspect Oriented Programming*– (POA – AOP) es propuesto por Gregor Kiczales [Kiczales97] en 1997, aunque el equipo Demeter [Demeter] había estado trabajando sobre este campo desde mucho antes. La primera definición temprana de **aspecto** también la dio este grupo en 1995 [Demeter]. Posteriormente y debido a las connotaciones restrictivas de la palabra *programación* se adopto como nombre el Desarrollo de Software Orientado a Aspectos – *Aspect Oriented Software Development*– (DSOA – AOSD), que es el término empleado actualmente, quedando la POA como un subconjunto de la misma.

4.1 Definición de Aspecto

Puesto que el DSOA es una aproximación de la SoC, antes de definir un aspecto es mejor saber qué es una **incumbencia** (o competencia) –*concern*.

"Incumbencia es todo aquello que incumbe al software" [Tarr99]. Básicamente incumbencia es todo lo que sea importante para la aplicación, ya sea código, infraestructura, requerimientos, elementos de diseño, etc.

En el DSOA hay dos términos muy importantes [Kiczales97]:

- Un **componente** –*component*– es aquel módulo software que puede ser encapsulado en un procedimiento (un objeto, método, procedimiento o API). Los componentes serán unidades funcionales en las que se descompone el sistema.
- Un **aspecto** – *aspect*– es aquel módulo software que no puede ser encapsulado en un procedimiento. No son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afectan a la ejecución o semántica de los componentes. Ejemplos de aspectos son la gestión de la memoria o la sincronización de hilos *(thread*s).

Un aspecto es una clase particular de incumbencia. La definición formal más aceptada es: "Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la

de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa" [Kiczales97]:

De una forma más básica se puede decir que los aspectos son elementos que se diseminan por todo el código y son difíciles de describir con respecto a otros componentes.

En la Figura 3 se pueden ver los requerimientos de un sistema como un haz de luz que pasa a través de un prisma el cual la descompone en las distintas incumbencias. Por un lado está la lógica de negocio, y por otro una serie de incumbencias que se diseminan por todo el código. El conjunto de todas ellas es lo que forma el sistema [Laddad02].

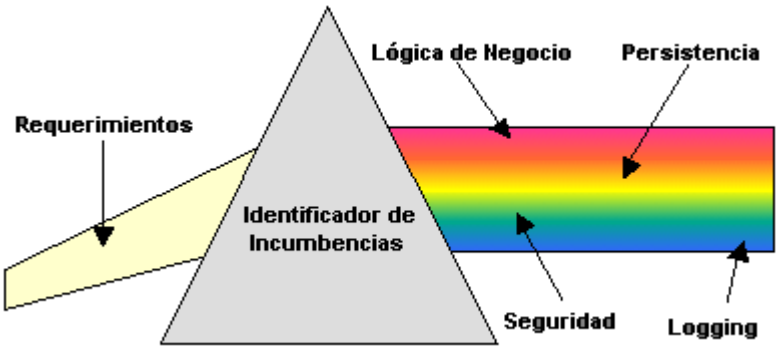

Figura 3: Requerimientos del sistema según POA

La estructura de un programa orientado a aspectos se muestra en la Figura 4. Se puede ver que el programa es una combinación de distintos módulos. Unos contienen la funcionalidad básica (modelo de objetos), mientras que los demás recogen otro tipo de características como son la seguridad, persistencia, *logging* (registro), gestión de memoria, etc.

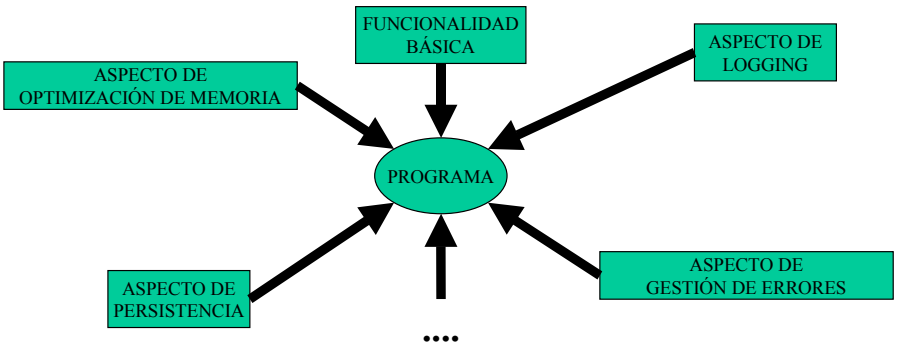

Figura 4: Estructura de un programa orientado a aspectos

En la parte izquierda de la Figura 5 puede verse la forma que tiene el código de un programa siguiendo el paradigma Orientado a Objetos. Se puede observar que el código está entremezclado siendo difícil de entender, depurar y modificar. En cambio

siguiendo el DSOA, mostrada en la parte derecha de la figura, las diferentes incumbencias están separadas con lo que son más fáciles de entender y, por lo tanto, de trabajar con ellas.

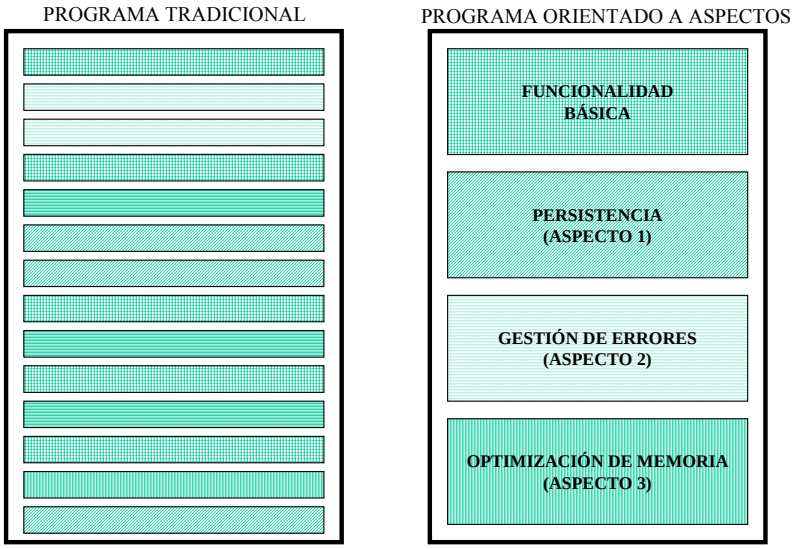

Figura 5: Código programa OO vs. DSOA

4.2 Cómo Funciona la POA

Básicamente en el funcionamiento de la POA se pueden identificar tres pasos (Figura 6):

1. Descomposición en aspectos: se descomponen los requerimientos con el fin de identificar las distintas incumbencias (las comunes y aquéllas que se entremezclan con el resto).
2. Implementación de incumbencias: se implementan de forma independiente las incumbencias detectadas anteriormente, tanto la funcionalidad básica como los aspectos ortogonales.
3. Recomposición de aspectos: este proceso, denominado **tejido** –*weaving*–, toma todos los módulos implementados anteriormente (funcionalidad básica y aspectos) y los teje para formar el sistema completo. Es realizado por el **tejedor de aspectos** –*aspect weaver.*

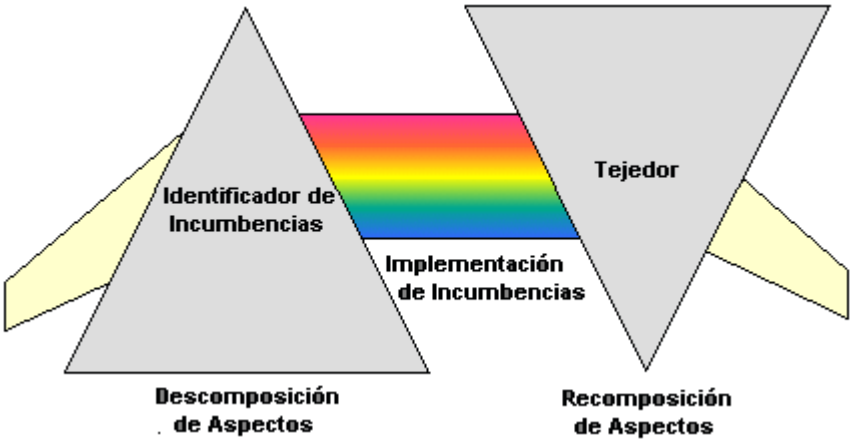

Figura 6: Pasos de la POA

Los lenguajes orientados a aspectos definen una nueva unidad de programación de software, el *aspecto*, para encapsular las funcionalidades que están diseminadas y enmarañadas –*crosscutt* y *tangled*– por todo el código. A la hora de formar el sistema se ve que hay una relación entre los componentes y los aspectos, y que, por lo tanto, el código de los componentes y de estas nuevas unidades de programación tiene que interactuar de alguna manera. Para que los aspectos y los componentes se puedan mezclar, deben tener fijados los puntos donde puedan hacerlo, que son lo que se conoce como **puntos de enlace** –*joinpoints*.

Los puntos de enlace son una interfaz entre los aspectos y los módulos de componentes que define en qué lugares se puede aumentar el comportamiento de los módulos con el comportamiento de los aspectos.

El proceso de realizar esta unión se conoce como tejido –*weave*–, y el encargado de realizarlo es el tejedor de aspectos –*aspect weaver*–.

El tejedor, además de los aspectos, los módulos y los puntos de enlace, recibe unas reglas que le indican cómo debe realizar el tejido. Estas reglas son los llamados **puntos de corte** –*pointcuts*–, que indican al tejedor qué aspecto tiene que aumentar a qué módulo a través de qué punto de enlace.

El código de los aspectos normalmente recibe el nombre de ***advice*** al ser el término utilizado en el sistema AspectJ [AspectJ] y que han adoptado la mayoría de sistemas posteriores.

Realmente los aspectos describen unos añadidos al comportamiento de los objetos, hacen referencia a las clases de los objetos y definen en qué punto se han de colocar los añadidos.

Como se puede ver en la Figura 7, en las metodologías tradicionales el proceso de generar un programa consistía en pasar el código a través de un compilador o un intérprete para así disponer de un ejecutable. En la POA no se tiene un único código del programa sino que está separado el código que implementa la funcionalidad básica y el código que implementa cada uno de los aspectos. Todo este código debe pasar no sólo a través del compilador, sino que debe ser tratado por el tejedor, que es el que se encarga de crear un único programa con toda la funcionalidad, la básica más los aspectos.

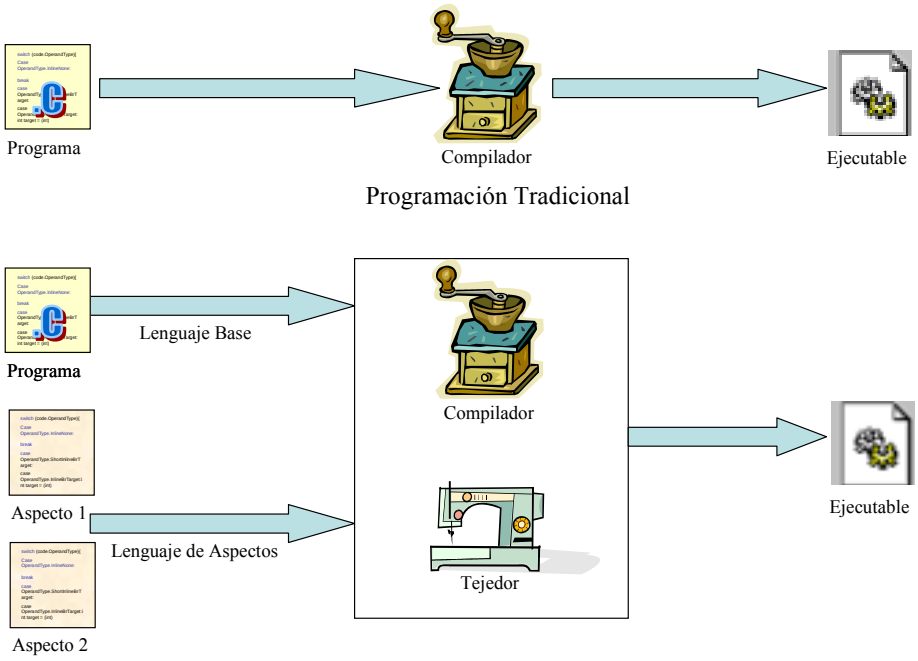

Figura 7: Implementación tradicional vs. POA

4.3 Terminología

Ya se han mencionado algunos de los términos que se emplean en la POA, pero a continuación se van a mostrar los más importantes:

- **Incumbencia**: es todo aquello que resulta importante para una aplicación (requerimientos, infraestructura, código, etc.).
- **Componente**: es aquel módulo software que puede ser encapsulado en un procedimiento (un objeto, método, procedimiento, API). Los componentes son unidades funcionales en las que se descompone el sistema.
- **Aspecto**: es aquel módulo software que no puede ser encapsulado en un procedimiento. Los aspectos no son unidades funcionales en las que se pueda dividir un sistema, sino propiedades que afectan a la ejecución o semántica de los componentes. Ejemplos de aspectos son la gestión de memoria y la sincronización de hilos.
- **Punto de enlace** (*joinpoint*): es un punto de la ejecución de un programa bien definido donde se podrá añadir código (funcionalidad). No todos los puntos de ejecución de un programa son puntos de enlace, sólo aquéllos que puedan ser tratados de forma controlada. Cada sistema que soporta POA ofrece una serie de puntos de enlace distintos. Ejemplos de puntos de enlace pueden ser invocaciones a métodos, acceso a campos, etc. (en cambio la ejecución de la línea 33 de la clase C, no es un punto de enlace).
- **Sombra de punto de enlace** *(joinpoint shadow)*: Una *joinpoint shadow* es la sombra que deja un punto de enlace a bajo nivel (en código intermedio, por ejemplo *bytecode* de Java). Existen puntos de enlace cuya sombra se corresponde con una instrucción y es sencilla de localizar. En cambio, existen puntos de enlace cuya sombra no se corresponde con una única instrucción o, incluso, no se corresponde con ninguna instrucción sino con una zona del código.

- **Punto de corte** (*pointcut*): es un conjunto de instrucciones que se le pasan al tejedor con el fin de que sepa qué código (*advice* del aspecto) se le debe añadir en qué punto de enlace a una aplicación. Un ejemplo podría ser invocar el método X del aspecto cuando se produzca un acceso de lectura al campo Y de la clase C.
- ***Advice***: es el código del aspecto que se ejecuta en los puntos de enlace seleccionados por un punto de corte. Normalmente desde su código se puede acceder al contexto de ejecución del punto de enlace (se puede acceder a los valores de variables, etc.).

A continuación se presentan un conjunto de siglas y nombres que están siendo usados en el ámbito de la programación orientada a objetos (se presentan en inglés por ser éste el idioma en el que se suelen utilizar):

- AOP (*Aspect Oriented Programming*): Programación Orientada a Aspectos (POA), es el término que inicialmente se utilizó para referirse a esta tecnología. En un principio se refería a todo el proceso del desarrollo (análisis, diseño, implementación) pero en los últimos tiempos se refiere más al proceso de codificación (programación propiamente dicha).
- AOSD (*Aspect Oriented Software Development*): Desarrollo de Software Orientado a Aspectos (DSOA), es el término usado actualmente para referirse al uso de técnicas orientadas a aspectos a lo largo del ciclo de vida del software. Incluye todo el proceso de desarrollo del software (análisis, diseño, implementación, etc.).
- AOD (*Aspect Oriented Design*): Diseño Orientado a Aspectos.
- AORE (*Aspect Oriented Requirements Engineering*): Ingeniería de requerimientos Orientado a Aspectos [AORE3].
- AOM (*Aspect Oriented Modeling*): Modelado Orientado a Aspectos[AOM03].
- Early Aspects: este concepto significa que es importante considerar los aspectos de forma temprana en el ciclo de vida de la ingeniería del software, durante el análisis y el diseño, en contraposición a la idea de tenerlos en cuenta únicamente en el momento de la implementación.

En general, a lo largo de este capítulo y el siguiente, nos referiremos a esta técnica principalmente como POA, ya que en general las herramientas específicas existentes ofrecen soporte para la fase de programación, pero no para el resto de fases del desarrollo.

4.4 Un Ejemplo

A continuación mostraremos mediante un ejemplo muy sencillo el uso de la Programación Orientada a Aspectos. Partimos de un programa que gestiona una biblioteca, en el que por simplificación se supone que sólo existen dos operaciones que son: realizar un préstamo (sacar un libro) o una devolución:

```
public class Biblioteca{
public void préstamo(Usuario usuario,Libro libro){
//código de préstamo
      }
public void devolución(Usuario usuario,Libro libro){
```

```
//código de devolución
      }
}
```

Si posteriormente se necesitase añadir un *log* –registro– que indicase el momento de entrada y el de salida de cada método, se tendría que tener un código que implementase el log (el aspecto en sí) y unas reglas que le indiquen al tejedor cómo debe realizar el tejido. Estas reglas en lenguaje natural serían:

- Ejecutar el log antes de que empiece cada método.
- Ejecutar el log después de que se haya ejecutado cada método.

Si esto se tuviese que realizar sin POA se tendría un programa similar al que se muestra (suponiendo que existe una clase Log):

```
//clase Log
public class Log{
public void escribe(String mensaje){
            ......
      }
}
//clase con log
public class BibliotecaConLog{
Log _log;
public void préstamo(Usuario usuario,Libro libro){
      _log.escribe("Entrando en préstamo"+ usuario+libro);
      //código de préstamo
      _log.escribe("Saliendo de préstamo"+ usuario+libro);
      }

public void devolucíón(Usuario usuario,Libro libro){
      _log.escribe("Entrando en devolución"+ usuario+libro);
      //código de préstamo
      _log.escribe("Saliendo de devolución"+ usuario+libro);
      }
```

Si utilizásemos POA, un ejemplo de aspecto (escrito en AspectJ [AspectJ][Kiczales01]) que realizara esto podría ser:

```
public aspect LogBiblioteca{
Log log;//se hace uso de la clase Log
      //define pointcut para cualquier método public de Biblioteca
      //con cualquier tipo de retorno
      //y con cualquier número y tipo de argumentos
pointcut publicBiblioteca():
execution(public * Biblioteca.*(..));
//define pointcut que cumpla publicBiblioteca y que reciba dos
// argumentos, el primero un usuario y el segundo un libro)
pointcut publicBibliotecaArgs(Usuario usuario,Libro libro):
publicBiblioteca() &&args(usuario,libro);
//antes de cada método identificado por publicBibliotecaArgs
// llama a log_acción
      before (Usuario usuario,Libro libro):
      publicBibliotecaArgs(usuario,libro){
      log_acción("entrando",thisJoinPoint.getSignature.toString(),
usuario, libro);
      }
//después de cada método identificado por publicBibliotecaArgs
// llama a log_acción
```

```
      after (Usuario usuario,Libro libro)returning:
      publicBibliotecaArgs(usuario,libro){
      log_acción("saliendo",thisJoinPoint.getSignature.toString(),
usuario, libro);
      }
      //método log_acción
      private log_acción(String cadena,String acción, Usuario usuario,
Libro libro){
      log.escribe(cadena+" "+acción+" "+ usuario+" "libro);
      }
}
```

El programa inicial (sin el log) no se tiene que modificar. El programa y el aspecto pasarían a través del tejedor el cual generaría el programa definitivo.

Sin entrar en detalles sobre el código del aspecto, lo importante es ver que gracias a esta técnica se tiene la posibilidad de adaptar (modificar, ampliar) el código sin necesidad de modificar su código fuente.

Igual que en el ejemplo se ha decidido añadir funcionalidad a los métodos públicos de una clase en concreto, se podía haber decidido añadir esa misma funcionalidad a todos los métodos de todas las clases, haciéndolo en el aspecto, sin tener que modificar para nada el programa original. Para ello se modificaría el *pointcut* `publicBiblioteca` del ejemplo y podría ser algo como:

```
//pointcut para todas las clases
pointcut publicTodos():
execution(public * *.*(..));
```

Algunas de las ventajas que se obtienen al usar POA son que, al no tener que modificar los métodos en el código original introduciendo llamadas al método de log, el código resultante es más reutilizable; más legible por no tener entremezclado código que no se refiera a la funcionalidad de cada método; más rápido de desarrollar; y en caso de que no se necesite realizar el log, bastaría con recompilar sin el aspecto para tener otra vez el programa original.

4.5 Tipos de POA

La programación orientada a aspectos se puede clasificar según varios criterios.

4.5.1 Tejido Estático Vs. Dinámico

El proceso de tejido es el punto más importante para cualquier solución que emplee POA. En la definición original de POA [Kiczales97], Kiczales y su equipo especificaron las siguientes posibilidades para hacer el tejido:

- Un preprocesador que haga las sustituciones pertinentes en el código.
- Un postprocesador que modifique archivos binarios.
- Un compilador que soporte *AOP* y que genere archivos con el proceso de tejido realizado.
- Tejido en tiempo de carga, realizando el proceso cuando las clases son cargadas en memoria, como haría *Java* con la *JVM*.

- Tejido en tiempo de ejecución, capturando cada punto de enlace mientras el programa está en funcionamiento y ejecutando el código que corresponda.

Posteriormente se ha presentado una alternativa denominada tejido en tiempo de despliegue –*deploytime weaving*–[Cohen04]. Este concepto implica un postprocesamiento del código, pero en vez de modificar el código generado, lo que se hace es generar subclases a partir de las clases existentes, introduciendo las modificaciones pertinentes mediante redefinición de métodos. Las clases que ya existían no se modifican en ningún momento con lo que todas las herramientas ya existentes pueden ser usadas durante el desarrollo. Una aproximación similar a esta se ha usado en la implementación de servidores de aplicaciones *J2EE* como IBM WebSphere[IBMWebSphere].

Si se utiliza como criterio el *momento* en el que se realiza el tejido se distinguen entre **tejido estático** y **tejido dinámico**. De las formas de tejido anteriormente mencionadas todas son estáticas, excepto el tejido en tiempo de carga y el tejido en tiempo de ejecución, que son dinámicas.

4.5.1.1 Tejido estático

La mayoría de las actuales implementaciones de POA están basadas en el tejido estático. El tejido estático consiste en la modificación en tiempo de compilación del código fuente, insertando llamadas a las rutinas específicas de los aspectos. Los lugares donde estas llamadas se pueden insertar son los **puntos de enlace** –*joinpoints* (definidos por cada sistema).

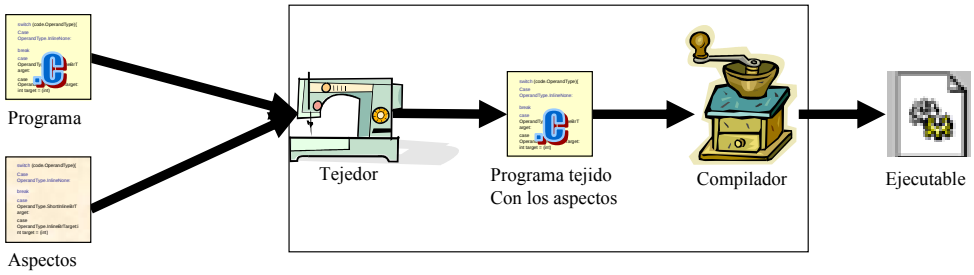

Figura 8: Tejido estático

En la Figura 8 se puede ver el proceso que se sigue en el tejido estático. Se parte del programa con la funcionalidad básica implementada en un lenguaje base (C++, Java, etc.) y además se tiene uno o varios aspectos, escritos en un lenguaje de aspectos (que puede ser una extensión de algún lenguaje normal, o uno específicamente definido con este fin). Por aspecto se entiende el código que se debe añadir –***advice***– y las instrucciones que recibe el tejedor sobre *dónde y cómo* insertar el código de los aspectos en el código principal –***pointcut***– (estas instrucciones pueden venir expresadas en el mismo lenguaje o en otro particular, y pueden encontrarse en el mismo fichero o en otro). El tejedor – *weaver*– realiza la composición de código, insertando el proveniente de los aspectos en el código de la funcionalidad básica siguiendo las instrucciones recibidas (del tipo "insertar llamada a método X antes de la invocación al método Y" o "invocar al método Z después de acceder al campo K de la clase C"). El resultado es un nuevo código fuente, de estilo tradicional, que pasará a través de un compilador el cual generará el programa ejecutable. Hay herramientas que realizan el proceso de tejido–

generación de código con aspectos–compilación de forma interna en un único paso desde el punto de vista del usuario.

El tejido estático presenta algunos inconvenientes como la imposibilidad de adaptarse a cambios[3] en el entorno de ejecución debido a que la aplicación final es generada tejiendo la funcionalidad básica con los aspectos en tiempo de compilación; cualquier funcionalidad que necesite ser adaptada en tiempo de ejecución implica que se debe parar la ejecución de la aplicación, ésta se debe recompilar con los nuevos aspectos (es decir pasar a través del tejedor y del compilador), y debe ser iniciada de nuevo. Ésta es la razón por la que el uso del tejido estático no es factible en aplicaciones que no puedan detenerse y que necesiten ser adaptadas.

Al mismo tiempo, la depuración de una aplicación que haya sido compilada y se le hayan añadido ya los aspectos es una tarea compleja porque el código que se debe analizar es el resultante del proceso de tejido: el código original está entremezclado con el de los aspectos, el cual estará diseminado por toda la aplicación.

La principal ventaja que ofrecen estos sistemas es que se evita que el uso de la POA derive en una penalización en el rendimiento, ya que antes de la compilación se dispone de la totalidad del código pudiendo ser optimizado por el compilador.

Una desventaja muy importante que suelen presentar las herramientas que ofrecen tejido estático (y las que ofrecen tejido dinámico) es que son dependientes del lenguaje; esto implica que el sistema sólo sirve para un lenguaje específico, no pudiendo crear distintos aspectos en distintos lenguajes. Por ejemplo, AspectJ [AspectJ] sólo puede usarse con el lenguaje Java.

4.5.1.2 Tejido dinámico

Usando un tejedor estático, el programa final se genera tejiendo el código de la funcionalidad básica y el de los aspectos seleccionados en la fase de compilación. Si se quiere enriquecer la aplicación con un nuevo aspecto, o incluso eliminar uno de los aspectos actualmente tejidos, el sistema debe ser recompilado y reiniciado de nuevo.

Aunque no todas las aplicaciones necesitan ser adaptadas mediante aspectos en tiempo de ejecución, hay aspectos específicos que se benefician de un sistema con tejido dinámico; puede haber aplicaciones que necesiten adaptar sus competencias específicas en respuesta a cambios en el entorno de ejecución [Popovici01]. Como ejemplo, técnicas relacionadas se han empleado en gestionar requerimientos de calidad del servicio en sistemas distribuidos de CORBA [Zinki97], en la gestión de sistemas de "*cache prefetching*" (precarga en memoria) en un servidor Web [Segura–Devillechaise03], y en distribución de incumbencias basada en el equilibrado de carga [Matthijs97]. Otro ejemplo de uso de POA de forma dinámica es lo que se ha llamado software autónomo – *autonomic software/computing*: software capaz de auto repararse, gestionarse, optimizarse o recuperarse [Autonomic].

[3] Nos referimos a cambios no previstos en el momento del desarrollo del programa, por ejemplo, debidos a requerimientos sobrevenidos.

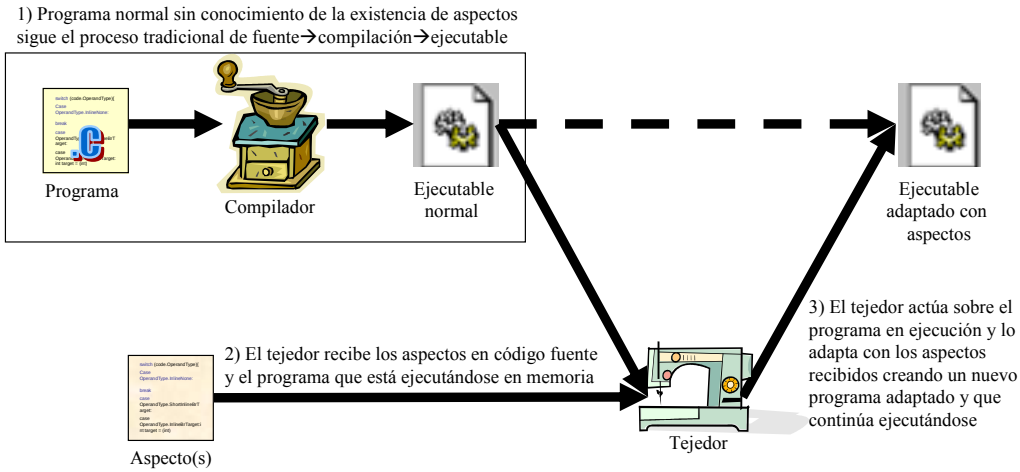

Figura 9: Tejido dinámico

En la Figura 9 se puede ver representado el proceso del tejido dinámico. En un primer paso un programa escrito en un lenguaje normal pasa por el compilador y se genera un ejecutable que se pone en ejecución (este es el proceso tradicional). Este programa no tiene por qué tener ningún conocimiento de que va a ser adaptado. Cuando se necesita adaptar (añadir o modificar funcionalidad) el programa que está ejecutándose (porque surgen nuevos requerimientos o en respuesta a cambios en el entorno), sin detener la ejecución, se procesa el programa en ejecución (no el programa ejecutable, sino el que está en memoria) junto a los aspectos que van a adaptarlo, y por medio del tejedor se modifica el programa en memoria añadiéndole la funcionalidad de los aspectos, dando lugar a una nueva versión del programa que continúa la ejecución. El programa ejecutable inicial no ha sido modificado con lo que se puede utilizar de nuevo sin las modificaciones realizadas en memoria.

En sistemas que usan tejido dinámico de aspectos, la funcionalidad básica permanece separada de los aspectos en todo el ciclo de vida del software, incluso en la ejecución del sistema. El código resultante es más adaptable y reutilizable, y los aspectos y la funcionalidad básica pueden evolucionar de forma independiente [Pinto02].

Con el fin de superar las limitaciones del tejido estático han surgido diferentes sistemas que ofrecen tejido dinámico. Estos sistemas ofrecen al programador la posibilidad de modificar de forma dinámica el código del aspecto asignado a puntos de enlace de la aplicación de forma similar a los sistemas reflectivos en tiempo real basado en **protocolos de meta objetos** –*meta object protocols*– (MOP) [Maes87][Kiczales91] [Sullivan01][Baker02].

Muchas de las herramientas que afirman ofrecer un tejido dinámico de aspectos realmente ofrecen un híbrido entre el tejido estático y el dinámico, pues los aspectos deben conocerse y definirse en el momento del diseño e implementación, para posteriormente, en ejecución, poder instanciarlos. Aunque esta solución aporta alguna ventaja respecto al tejido estático hay casos en que no es suficiente, por ejemplo cuando una vez que la aplicación está funcionando surge un requerimiento nuevo, que no se tuvo en cuenta en el diseño.

Al igual que ocurre con las herramientas de tejido estático, la mayoría de las herramientas que ofrecen tejido dinámico son dependientes del lenguaje como se verá más adelante.

La principal desventaja del tejido dinámico respecto al estático es el menor rendimiento en ejecución causado por la adaptación dinámica de las aplicaciones [Böllert99].

4.5.2 Lenguajes de Aspectos de Dominio Específico Vs. de Propósito General

A la hora de crear un programa orientado a aspectos se utiliza un lenguaje para crear los componentes y otro lenguaje para los aspectos. El lenguaje en el que se crean los componentes es el **lenguaje base**, que puede ser cualquier lenguaje (Java, C/C++, etc.).

El lenguaje en el que se crean los aspectos es lo que se denomina **lenguaje de aspectos**, y puede ser de dominio específico o de propósito general. El lenguaje de aspectos y el lenguaje base pueden ser el mismo (de hecho así ocurre en muchos sistemas). A continuación se comparan ambas alternativas.

4.5.2.1 Lenguajes de aspectos de dominio específico

Los lenguajes de dominio específico soportan alguno de los tipos de aspectos de los que se ha hablado (persistencia, gestión de memoria, distribución, etc.) pero no pueden soportar aspectos para los que no fueron diseñados. Suelen tener un nivel de abstracción superior al del lenguaje base.

Estos lenguajes imponen restricciones al lenguaje base con el fin de garantizar que las incumbencias soportadas por el lenguaje de aspectos se traten sólo mediante él (esto implica que se traten en los aspectos) y que no puedan ser tratadas mediante el lenguaje base (es decir, no pueden tratarse en los componentes), lo que podría provocar interferencias entre ambos.

Existen lenguajes muy diversos de dominio específico, por ejemplo COOL, que trata aspectos de sincronización y RIDL que trata aspectos de distribución (ambos propuestos en [Lopes97]), o la implementación DCOOL (COOL de forma dinámica) [Czarnecki98].

La ventaja del uso de estos lenguajes es que fuerzan la separación de las funcionalidades entre los componentes (escritos en lenguaje base) y los aspectos (escritos en un lenguaje específico). Es decir garantizan la idea central de la POA.

El principal inconveniente que presentan es la necesidad de aprender varios lenguajes distintos por parte de los desarrolladores para poder completar un sistema.

4.5.2.2 Lenguajes de aspectos de propósito general

Los lenguajes de propósito general están pensados para ser utilizados con cualquier clase de aspecto, no con unos específicos. En principio no imponen restricciones

al lenguaje base. La forma de separar los aspectos de los componentes es en base a la separación en unidades de aspectos. Suelen ser extensiones a lenguajes ya existentes, por lo que tienen el mismo nivel de abstracción y conjunto de instrucciones, al ser necesario poder implementar cualquier aspecto en este lenguaje.

El principal inconveniente que presentan estos lenguajes es la imposibilidad de garantizar que tareas propias de los aspectos no sean implementadas en los componentes, lo cual puede llevar al problema de código entremezclado y diseminado.

Por el contrario la gran ventaja es el uso de un lenguaje único, y normalmente conocido, con lo que a nivel empresarial es mucho más atrayente, ya que se evita que los programadores tengan que aprender muchos lenguajes con el consiguiente gasto de tiempo en aprendizaje de lenguajes y cambios de contexto.

El principal ejemplo de este tipo de lenguajes es AspectJ [AspectJ], que es una extensión al lenguaje Java, escribiéndose tanto los componentes como los aspectos en Java.

CAPÍTULO 5
SISTEMAS POA EXISTENTES

A continuación se van a presentar algunos de los sistemas existentes que ofrecen POA. Se va a realizar una clasificación agrupando en sistemas estáticos, sistemas dinámicos, servidores de aplicaciones y basados en componentes. Como veremos, la mayoría de los sistemas que ofrecen el tejido dinámico en realidad lo único que soportan es la instanciación dinámica de los aspectos que tienen que haber sido definidos en el diseño del programa (ya que muchos sistemas necesitan modificar el código del programa base para poder adaptarlo posteriormente).

5.1 Sistemas Estáticos

Entre los sistemas estáticos existentes sólo veremos unos pocos ya que AspectJ es el usado mayoritariamente y muchos otros son "copias" para otros lenguajes (para C++ [AspectC++], para Ruby [AspectR], para Squeak/Smalltalk [AspectS]) o son sistemas de propósito específico. Además, puesto que uno de los requisitos de esta Tesis es el dinamismo, ningún sistema estático puede cumplirlo.

5.1.1 AspectJ

AspectJ [AspectJ] fue el primer sistema comercial que ofreció POA y es el más extendido en la actualidad, siendo un modelo a seguir y de comparación para el resto de sistemas que pretenden ofrecer POA. No es casualidad que haya sido desarrollado inicialmente en Xerox PARC [PARC] por el mismo grupo de personas que propusieron el concepto de programación orientada a aspectos [Kiczales97]. Actualmente forma parte, y se encuentra albergado, en el proyecto Eclipse [Eclipse].

Tradicionalmente AspectJ presenta un tejido estático, pero a partir de la versión 5, y como fruto de la unión de este proyecto con AspectWerkz [AspectWerkz], se incorpora el tejido en tiempo de carga como opción de trabajo.

AspectJ es una especificación de lenguaje y también una implementación de lenguaje que soporta POA. La especificación del lenguaje define varias construcciones y su semántica para soportar los conceptos de la orientación a aspectos. Esto permite que puedan existir diversos entornos que implementen la especificación, ya sea de for-

ma total o parcial. La implementación del lenguaje por parte del equipo "oficial"[4] de AspectJ ofrece herramientas para compilar, depurar y documentar código [Laddad02].

Las construcciones de AspectJ extienden el lenguaje Java [Gosling96], por lo que cualquier programa válido escrito en Java es también válido en AspectJ (no ocurriendo así a la inversa).

El compilador de AspectJ genera clases Java puras (conformes al estándar), por lo que pueden ser ejecutadas en cualquier máquina virtual de Java (JVM).

Entre las herramientas que proporciona AspectJ se encuentra un compilador (que es el tejedor de aspectos), un depurador y un generador de documentación que se pueden integrar en alguno de los entornos de desarrollo más usados actualmente (Eclipse [Eclipse], Jbuilder de Borland [Jbuilder], FORTE o SunONE/NetBeans de Sun [FORTE], y EMACS [EMACS]). Además cuenta con un navegador de aspectos –*browser*–.

Los conceptos que AspectJ ha añadido a Java son:

- ***Joinpoints*** (puntos de enlace).
- *Pointcut designators* o ***Pointcuts*** (puntos de corte).
- ***Advice*** (código del aspecto).

Las definiciones de estos conceptos se han visto en el capítulo anterior. Estos conceptos han sido adoptados por la mayoría de los sistemas que implementan orientación a aspectos.

La forma típica de trabajar de AspectJ se puede ver en la Figura 10. De forma separada se programan los **componentes** (que contendrán la funcionalidad básica de la aplicación) en Java estándar, y por otra parte se programan los **aspectos** (entendiéndose como tal el conjunto de *pointcuts* y *advice*) mediante las extensiones de AspectJ, todo esto (aspectos y componentes) se procesa por parte del **compilador** de AspectJ (ajc, *AspectJ compiler*) [O'Brien01], el cual genera los ficheros de clases (estándar) que son los que se ejecutarán por parte de la JVM.

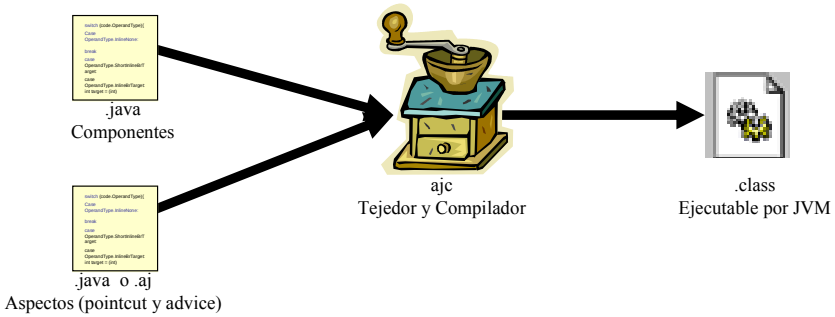

Figura 10: Proceso de compilación en AspectJ

[4] Existen otras implementaciones de la especificación, por eso distinguimos entre la realizada, y soportada, por el equipo de desarrollo de AspectJ y la denominamos "oficial", y el resto de implementaciones.

También se puede usar el compilador estándar de Java, actuando en este caso el compilador de AspectJ como un mero precompilador que genera nuevo código Java, que es el que se le pasa al compilador estándar, para que sea éste el que genere el código ejecutable (en las primeras versiones era así como funcionaba).

Existe, al menos, otro compilador de AspectJ llamado AspectBench Compiler [AspectBench], el cual asegura que es compatible con el compilador original (*ajc*) y está pensado para facilitar las posibles extensiones y optimizaciones. Los autores argumentan que este compilador genera un código rápido y que provee de códigos de error alternativos. Por otro lado, y de cara a la investigación, facilita el añadir extensiones al lenguaje con el fin de obtener nuevas características. Este compilador se presenta bajo licencia *GNU LGPL* [GNULGPL].

En el caso de que se necesite añadir un nuevo aspecto, modificar alguno existente, o eliminarlo es necesario volver a compilar la aplicación. Es decir, es un sistema que soporta orientación a aspectos de forma estática. En la versión 5 se ha añadido cierto grado de dinamismo, como veremos más adelante.

Los aspectos codificados contienen uno o varios puntos de corte y *advice* asociados. La función de los puntos de corte es seleccionar puntos de enlace en el componente. Los *advice* son el código que se debe ejecutar cuando se alcanza un punto de enlace que ha sido seleccionado por el punto de corte asociado.

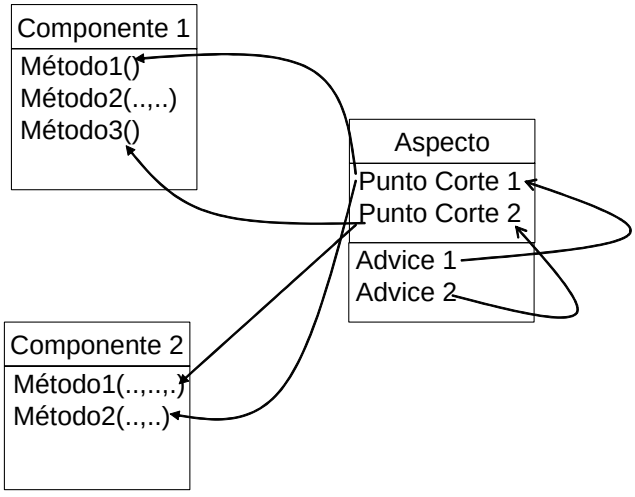

Figura 11: Componentes y aspectos en AspectJ

En la Figura 11 podemos observar como los *advice* están relacionados con un punto de corte en concreto, y el punto de corte selecciona uno o varios (o ninguno) puntos de enlace. Cuando se alcanza un punto de enlace seleccionado por un punto de corte se ejecuta el código del *advice* asociado al punto de corte.

5.1.1.1 Puntos de enlace

Al ser AspectJ el primer sistema que ofreció orientación a aspectos y por ser actualmente el patrón a seguir para el resto de sistemas, tiene mucha importancia el conjunto de puntos de enlace que soporta, ya que los demás sistemas intentan soportar los mismos o un subconjunto de ellos, por lo que ha pasado a ser el patrón de comparación.

Los puntos de enlace que soporta son [AspectJc]:

- Invocación a un método.
- Invocación a un constructor.
- Ejecución de un método.
- Ejecución de un constructor.
- Inicialización de objetos que se crean con el constructor.
- Preinicialización de atributos (asignación en la declaración) realizada antes de la invocación al constructor del padre (`super`).
- Inicialización de bloque estático.
- Acceso de lectura a campo.
- Acceso de escritura a campo.
- Cuando una excepción `IOException` (o un subtipo de la misma) se trata en un bloque catch.
- Ejecución de cualquiera de los advice inyectados.

Estos son los puntos de enlace donde se puede añadir código en un componente en AspectJ. Los puntos de enlace son seleccionados mediante los puntos de corte que se muestran a continuación.

5.1.1.2 Puntos de corte

Los puntos de corte definen una agrupación de puntos de enlace a capturar. Se expresan mediante las primitivas de los puntos de enlace y, parametrizándolas, mediante tipos, modificadores (public, static, ...), expresiones regulares (operador *) y operadores de consulta (|| o &&). A continuación se muestran los puntos de corte existentes:

- `call`, invocación a un método.
- `call`, invocación a un constructor.
- `execution`, ejecución de un método.
- `execution`, ejecución de un constructor.
- `initialization`, inicialización de objetos que se crean con el constructor.
- `preinitialization`, preinicialización de atributos (asignación en la declaración) realizada antes de la invocación al constructor del padre (super).
- `staticinitialization`, inicialización de bloque estático.
- `get`, acceso de lectura a campo.
- `set`, acceso de escritura a campo.
- `handler`, cuando una excepción `IOException` (o un subtipo de la misma) se trata en un bloque catch.
- `adviceexecution`, ejecución de cualquiera de los advice inyectados.

Hasta aquí se han visto puntos de corte que se corresponden con puntos de enlace de forma directa, los puntos de corte que se muestran a continuación sirven para, combinándolos con los anteriores mediante operadores, poder seleccionar puntos de enlace de una forma más potente:

- `within`, selecciona cualquier punto de enlace donde el código asociado se define dentro de una clase, un paquete, etc.
- `withincode`, selecciona cualquier punto de enlace donde el código asociado se define dentro de un método.
- `withincode`, selecciona cualquier punto de enlace donde el código asociado se define dentro del constructor.
- `cflow`, selecciona cualquier punto de enlace que se produce en el flujo de control de una llamada a un método (incluye la llamada).
- `cflowbelow`, selecciona cualquier punto de enlace que se produce a debajo del flujo de control de una llamada a un método (no incluye la llamada).
- `if`, selecciona cualquier punto de enlace donde se cumpla una condición (dada como parámetro).
- `this`, selecciona cualquier punto de enlace en el que el objeto que se está ejecutando es una instancia de una clase determinada (como parámetro).
- `target`, selecciona cualquier punto de enlace en el que el objeto destino es una instancia de una clase determinada (como parámetro).
- `args`, selecciona cualquier punto de enlace donde los argumentos cumplan ciertas reglas (dadas como parámetros) como pueden ser su tipo, su número o su orden.
- `annotation`, selecciona cualquier punto de enlace donde el sujeto contiene una anotación del tipo pasado como parámetro

A continuación se muestran algunos ejemplos de puntos de corte muy sencillos:

- `target(Point) && call(int *()),` selecciona cualquier método que devuelve un int, que no tiene parámetros y que se realiza en un objeto que es una instancia de la clase Point.
- `within(*) && execution(*.new(int))`, selecciona la ejecución de cualquier constructor que recibe exactamente un parámetro de tipo int sin que importe desde donde se realiza la llamada.
- `execution(!static * *(..))`, selecciona cualquier ejecución de cualquier método no estático, con cualquier número de parámetros.

Generalmente los puntos de corte se definen (reciben un nombre) y son posteriormente usados en la declaración de los *advice*, aunque también se pueden usar de forma anónima directamente en la declaración del *advice* (lo veremos a continuación).

5.1.1.3 Advice

En AspectJ a la hora de declarar un advice, además del código que se va a añadir, se selecciona la composición de puntos de corte en los que se va a añadir el código, y además se indica si va a ejecutarse "antes" *–before–*, "después" *–after–* o "en vez de" *–around–* el punto de corte alcanzado.

- `before`, el código se ejecuta antes de ejecutar el punto de enlace (una invocación a un método, un acceso a un campo, etc.).
- `after returning`, el código se ejecuta después si se devuelve lo que se haya declarado.

- `after throwing`, el código se ejecuta después en caso de que se haya producido una excepción.
- `after`, el código se ejecuta después de ejecutar el punto de enlace sin importar como se haya salido (mediante una excepción o salida normal)
- `around`, se ejecuta en vez de ejecutar el punto de enlace.
- `around throws`, se ejecuta en vez de ejecutar el punto de enlace pero puede lanzar una excepción.

Además de esto, en un advice se puede acceder a:

- `thisJoinPoint`, que proporciona información reflectiva sobre el punto de enlace.
- `thisJoinPointStaticPart`, que proporciona información reflectiva únicamente sobre la parte estática del punto de enlace.
- `thisEnclosingJoinPointStaticPart`, que proporciona información reflectiva únicamente sobre la parte estática del punto de enlace que contiene a este.
- `proceed`, (sólo accesible en los de tipo *around*) que permite ejecutar el código original del punto enlace.

Un ejemplo sencillo de advice puede ser:

```
//utilizando un punto de corte con nombre
–se define el punto de corte con nombre
pointcut setter(Point p1, int newval): target(p1) && args(newval)
      (call(void setX(int) || call(void setY(int)));
–se utiliza el punto de corte creado anteriormente
before(Point p1, int newval): setter(p1, newval) {
      System.out.println("se va a fijar algo en " + p1 +
      " al nuevo valor de " + newval);
  }
//el mismo caso utilizando un punto de corte anónimo
before(Point p1, int newval): target(p1) && args(newval)
      (call(void setX(int)) || call(void setY(int))) {
      System.out.println("se va a fijar algo en " + p1 +
      " al nuevo valor de " + newval);
  }
```

En este ejemplo se muestran dos formas de realizar lo mismo, la primera crea un punto de corte y le da un nombre para, posteriormente, utilizarlo en un advice (o en varios); la segunda utiliza un punto de corte de forma anónima (lo crea dentro del advice, con lo que no se puede utilizar en más advice). Lo que realiza este punto de corte es seleccionar las invocaciones a los métodos `setX` y `setY`, que reciben como parámetro un único `int`, y cuyo objeto es una instancia de la clase `Point`. Lo que hace el advice es fijar que "antes" –*before*– de que se produzca el punto de corte se ejecute el código (en este caso simplemente muestra por pantalla una frase, pero podría ser la invocación a un método, o cualquier cosa que se necesitase).

El código que hemos visto es la forma tradicional de declarar *pointcuts* y *advice* en AspectJ; a partir de la versión 5 existe otra forma de realizarlo mediante *annotations*.

```
public aspect Authorization {
    pointcut restrictedOperation():
     execution(*IRestrictedAccess+.*(..));
```

```
    before() : restrictedOperation() {
        // verificar que el llamante está autorizado
        // para realizar esta operación
    }
}
```

Este código es la forma tradicional de definir los aspectos, mediante las extensiones del lenguaje. A continuación ponemos el mismo aspecto mediante *annotations*.

```
@Aspect
public class Authorization {

  @Pointcut("execution(* IRestrictedAccess+.*(..))")
   void restrictedOperation() {}

  @Before("restrictedOperation()")
   public void verifyAccess() {
        // verificar que el llamante está autorizado
        // para realizar esta operación
   }
}
```

Ambas formas de definir los aspectos son semánticamente iguales y ambas son entendidas por el compilador, tejedor, etc.

5.1.1.4 Tejido en AspectJ 5

El tejedor de AspectJ 5 recibe ficheros *.class* como entrada y genera ficheros *.class* como salida. El proceso de tejido puede tener lugar en tres momentos diferentes [AspectJb]:

- Tejido en tiempo de compilación. Es la manera más sencilla, y es la usada tradicionalmente. El compilador (*ajc*) recibe el código fuente de la aplicación y genera los ficheros *.class* directamente. La invocación al tejedor se realiza de forma interna y transparente al usuario. Los aspectos pueden encontrarse en código fuente o pueden estar previamente compilados.
- Tejido en tiempo de post–compilación[5]. Se utiliza para tejer ficheros que se encuentren en formato *.class* o *.jar*. El fichero original es modificado, generándose un nuevo fichero. Al igual que en el caso anterior las aspectos pueden encontrarse en código fuente o en binario.
- Tejido en tiempo de carga. Es simplemente tejido binario (como el caso anterior) pero pospuesto al momento en que la clase es cargada por el *class loader* y definida para la JVM. El fichero original no es modificado.

El tejido en tiempo de carga es el resultado de la unión de AspectJ y AspectWerkz, soportando el tipo de tejido presente en el último y manteniendo su potencia y sus limitaciones.

Aunque el tejido propiamente dicho se realiza en tiempo de carga la especificación de los puntos de corte y advice está fijada previamente a la carga. Además una vez

[5] También conocido como tejido binario.

cargadas las clases y, por lo tanto, tejidas junto a los aspectos, no se pueden modificar, con lo que no seria posible añadir en ejecución un nuevo aspecto, modificar uno existente o eliminarlo de la ejecución [AOSDEurope].

Una ventaja que presenta el tejido en tiempo de carga sobre los otros dos es que los ficheros originales de la aplicación permanecen intactos, por lo que pueden ser utilizados por aplicaciones que no necesiten los aspectos, o incluso en otras aplicaciones que necesiten tejer otros aspectos.

5.1.1.5 Conclusiones

Este sistema presenta un conjunto de puntos de enlace muy rico, permitiendo adaptar las aplicaciones de una forma muy potente. Al realizar un tejido estático el rendimiento es elevado. En el caso del tejido en tiempo de carga lo único que se penaliza es el tiempo consumido en la carga de las clases debido a que es en ese momento cuando se hace el tejido, pero una vez tejidos los aspectos con la aplicación el rendimiento es el mismo que en los casos anteriores.

Este sistema se ha convertido en el estándar de facto y cuenta con el apoyo de grandes compañías y entidades ([PARC],[DARPA], IBM, diversas universidades, etc.) y una gran base de usuarios.

Los principales inconvenientes que presenta este sistema son la dependencia del lenguaje Java, lo que implica que programas escritos en otros lenguajes no pueden ser adaptados mediante este sistema, y su naturaleza estática.

Aún en el caso del tejido en tiempo de carga, los aspectos y las reglas de tejido tienen que haber sido definidos previamente, con lo que el sistema no puede adaptarse en ejecución a un cambio o requisito no previsto durante la implementación. En caso de que ocurriese esto, es necesario parar la aplicación y volver a tejerla (ya sea en tiempo de compilación o de carga) junto a los nuevos aspectos que cumplan con los nuevos requisitos.

5.1.2 Weave.NET

Este sistema, que se ha desarrollando en el Trinity College Dublín [Weave.NET], pretende conseguir tres objetivos fundamentales:

- Independencia del lenguaje.
- Tejido en tiempo de carga.
- Posibilidad de añadir funcionalidad a un aspecto como si fuese un componente normal.

Para desarrollar el sistema se utiliza la plataforma .NET. En la plataforma .NET cualquier programa escrito en cualquier lenguaje de programación es traducido a código intermedio (IL), que es el que se ejecuta por parte de la máquina virtual.

Este sistema realiza el tejido a nivel de código IL. Al trabajar con el código intermedio de forma directa se obtiene el primero de los objetivos (independencia del lenguaje), ya que cualquier programa escrito en cualquier lenguaje pueda ser adaptado por

aspectos codificados en cualquier otro lenguaje. Debido a esto se consigue otra ventaja que es la no necesidad de disponer del código fuente de los componentes para poder ser modificados.

Para modificar el código intermedio se utiliza una herramienta llamada CLIFileReader [CLIFileReader] la cual permite procesar ficheros que están en binario (IL). Posteriormente, y mediante reflexión, se puede modificar el código original.

Los aspectos son tratados como componentes normales sin ninguna distinción con otro tipo de componente, esto implica que un aspecto puede ser adaptado por otro aspecto.

El conjunto de puntos de enlace que ofrece este sistema es el mismo que el de AspectJ, ofreciendo así la misma funcionalidad que este sistema pero añadiéndole la ventaja de la independencia del lenguaje.

Los puntos de corte se definen de forma independiente a los aspectos. Mediante un fichero XML se especifica la forma en que deben modificarse los componentes por parte de los aspectos (el tejido). El diseño realizado permite utilizar los mismos puntos de corte que en AspectJ (aunque la implementación realizada no soporta varios de ellos, como los accesos de lectura o escritura a campos, inicialización o manejo de excepciones).

El separar el código de los aspectos de la definición de los puntos de corte implica un menor acoplamiento y una mayor posibilidad de reutilización de código debido a que un mismo aspecto se podrá utilizar para modificar el comportamiento de distintos componentes, sin tener que modificar su implementación, simplemente utilizando distintos ficheros XML donde se definan los puntos de corte.

Aunque en la especificación del sistema hay tres tipos de *advice* (*after, before y around*) en la implementación existente sólo se pueden utilizar *after* y *before*.

El proceso de tejido se realiza en tiempo de carga. El tejedor recibe un ensamblado (*.NET assembly*) con el componente original, un fichero XML en el que se especifican los puntos de corte y un ensamblado con el aspecto (o los aspectos). El resultado es un nuevo ensamblado con los aspectos tejidos en base a los puntos de corte del fichero XML.

En general este sistema presenta ciertas ventajas sobre AspectJ ya que es independiente del lenguaje, pudiendo expresar tanto los componentes como los aspectos en cualquier lenguaje compatible con la plataforma .NET. Otra ventaja que presenta es el tratamiento que se hace de los aspectos como si fuesen componentes normales, lo que les permite ser modificados por otros aspectos a su vez. La especificación de los puntos de corte se realiza mediante un fichero XML lo que evita tener que realizar extensiones a los lenguajes (como es el caso de AspectJ) y facilita la reusabilidad de los aspectos.

Este sistema no presenta características dinámicas, ya que el tejido se realiza en tiempo de carga y no puede ser modificado posteriormente, por lo que no se podrían añadir nuevos aspectos a una aplicación que estuviese funcionando, ni tampoco se podría “destejer” un aspecto cuando ya no fuese necesario.

5.1.3 Sourceweave.Net

Este sistema también se ha desarrollado en el Trinity College Dublín [SourceWeave.NET], y es muy similar al anterior. Persigue los mismos objetivos que Weave.NET y la principal diferencia es que trabaja sobre el código fuente en vez de código intermedio.

El objetivo de trabajar con código fuente es facilitar el proceso de depuración, ya que el resultado del proceso de tejido será un programa completo con la funcionalidad del componente base más la de los aspectos.

El conjunto de puntos de enlace que soporta está basado en los de AspectJ, ofreciendo la misma funcionalidad con la ventaja de la independencia del lenguaje.

Los puntos de corte se definen en un fichero XML externo al aspecto. En el fichero se especifica cómo debe adaptarse el componente base mediante los aspectos. El conjunto de puntos de corte que soporta es similar al de AspectJ.

El proceso de tejido se realiza en tiempo de precompilación. Es decir el tejido es un paso previo a la compilación. El tejedor recibe el código fuente del componente base, el fichero XML donde se especifica cómo se debe hacer el tejido, y el código fuente de los aspectos. El resultado es código fuente en el que se encuentra tejido el código del componente base junto al código de los aspectos.

Para poder realizar el tejido de código fuente SourceWeave.NET utiliza CodeDOM[CodeDOM]. CodeDOM permite representar el código de los programas mediante un grafo de objetos de forma abstracta; también permite generar código fuente a partir de la representación del grafo. Para poderlo utilizar los lenguajes tienen que ser compatibles con CodeDOM.

El tejedor obtiene mediante un analizador (*parser*) y un componente AST (*Abstract Syntax Tree*) un grafo que representa al código del componente base. Este grafo es modificado, siguiendo las directrices contenidas en el fichero XML, añadiendo los elementos necesarios para representar el código de los aspectos. Una vez terminado este proceso se vuelve a generar código fuente a partir del grafo. Este código, ya entrelazado, contiene la funcionalidad básica más la de los aspectos y es compilado a un ejecutable.

El tejedor también añade comentarios al código con el fin de conseguir facilitar el proceso de depuración de la aplicación tejida. Los comentarios hacen referencia a lo que está ocurriendo (una llamada al *advice*) y la razón de que se inyectase esa llamada, referenciada al fichero XML.

Este sistema ofrece una potencia similar a AspectJ ofreciendo la posibilidad de utilizar distintos lenguajes para los componentes y los aspectos (con la restricción de que deben ser lenguajes compatibles con CodeDOM). Al trabajar directamente sobre el código fuente se obtiene una mejora en el proceso de depuración, pero a la vez se contrae un requerimiento que puede ser una desventaja y es la necesidad de disponer del código fuente de la aplicación a modificar.

El hecho de trabajar con una librería propietaria de Microsoft (CodeDOM), que no es propia del estándar CLI, implica problemas de portabilidad para otras implementaciones de la plataforma .Net (veremos varias implementaciones en 6.5.3).

El sistema realiza el tejido en un momento anterior a la compilación y no ofrece posibilidades dinámicas. En caso de que hubiese que modificar la aplicación o los aspectos es necesario volver a realizar el proceso de tejido–compilación. Tampoco es posible añadir o eliminar aspectos en ejecución como respuesta a un cambio de requisitos o a la aparición de otros nuevos.

5.1.4 Dotspect (.SPECT)

Este sistema [DotSPECT] está en una fase beta, la primera versión publicada fue en Enero de 2006. Es un proyecto de código abierto, ofrecido bajo la licencia BSD [BSDLicense].

Es un sistema implementado sobre la plataforma .Net. El tejido se realiza de forma estática sobre código IL. Al trabajar sobre código intermedio se beneficia de la posibilidad de utilizar cualquier lenguaje que sea soportado por la plataforma para implementar los aspectos y la aplicación base.

La forma de especificar los aspectos es mediante un lenguaje similar a AspectJ. En el fichero de un aspecto (que tiene extensión *.aspect*) existe una cabecera que indica el lenguaje en el que está implementado el código del aspecto (*advice).* Tras esta cabecera se encuentra la definición de los puntos de corte (expresados en el lenguaje creado al efecto, similar al AspectJ) y también el código del aspecto en el lenguaje especificado (en la actualidad hay soporte para C# y VB.Net).

Actualmente, el sistema sólo soporta como puntos de enlace la ejecución y la invocación de métodos y *advice* de tipo *before y after* (no soporta *around).*

El tejedor recibe por un lado un ensamblado con la aplicación base ya compilada. Por otro lado recibe un fichero *.aspect* en el que vienen definidos los puntos de corte y el código del aspecto (en lenguaje fuente). A partir de aquí realiza los siguientes pasos:

- El tejedor obtiene un AST que representa a la aplicación base. Para realizar esto hace uso de la herramienta de manipulación de código intermedio RAIL [RAIL].
- Se genera el código intermedio correspondiente al código fuente de los *advice* que están implementados en el fichero. Para realizar esta generación hacen uso de la utilidad CodeDOM [CodeDOM] vista anteriormente.
- Se procesan las definiciones de los puntos de corte, para ello se utiliza el *parser* genérico Gold [Gold].
- Mediante un *Visitor* [Gamma94] se recorre el árbol anteriormente creado comprobando en cada nodo si se verifica por algún punto de corte, en caso de que así fuera se añade el código del *advice* (unas cabeceras predefinidas más el código generado anteriormente a partir de los fuentes).
- Como último paso se genera de nuevo (mediante RAIL) el ensamblado con las modificaciones realizadas.

Este sistema se encuentra en una fase muy temprana y, aunque los objetivos son muy ambiciosos, la implementación existente puede tomarse como una prueba de viabilidad. En todo caso el trabajo parece abandonado desde Febrero de 2006, momento en el que el autor presenta la primera versión como implementación de su *Master Thesis* y comenta en su página que se dedicará a otros proyectos.

El sistema presenta un lenguaje similar a AspectJ mediante el que se pueden definir los puntos de corte. Al implementarse sobre la plataforma .NET se consigue una independencia del lenguaje, tanto para el código base (puede ser cualquier lenguaje compatible con el *CLI)* como para el código de los aspectos (que tiene que ser un lenguaje compatible con CodeDOM).

El hecho de que vengan definidos en el mismo fichero los puntos de corte junto al código de los aspectos introduce un grado de dependencia entre éstos y la aplicación base, lo que dificultará su reutilización con otras aplicaciones base o en otros sistemas.

El hecho de trabajar con CodeDOM que es una librería propietaria de Microsoft, no incluida en el estándar CLI, implica problemas de portabilidad para otras implementaciones de la plataforma .Net que no disponen de ella.

5.1.5 AspectDNG

AspectDNG [AspectDNG] es un sistema de código abierto que ofrece POA sobre la plataforma .NET obteniendo de esta forma un sistema independiente del lenguaje. El proceso de tejido es estático. Después de publicar la primera versión operativa de un prototipo en Septiembre de 2006, el autor ha abandonado el proyecto, ofreciéndolo a quién quiera continuarlo, por lo que no es seguro que se continúe con su desarrollo.

El conjunto de puntos de enlace que soporta se limita a invocaciones y ejecuciones de métodos y al acceso de lectura o escritura de campos. El único tipo de *advice* que soporta es el *around.* También permite la inserción de elementos (tipos, campos, etc.) en una clase.

El tejido se realiza en el código intermedio IL, mediante instrumentación de código. Hace uso de Cecil [Cecil] para desensamblar el código de la aplicación base y de los aspectos y volver a ensamblar el código ya enlazado.

La especificación de los puntos de enlace puede hacerse mediante un fichero XML externo o mediante atributos personalizados (*Custom Attributes)* de la plataforma .Net. Semánticamente son equivalentes y, de hecho, se pueden utilizar de forma conjunta, especificando unos puntos de corte mediante los atributos personalizados y otros mediante el fichero XML.

Dentro de la especificación de los puntos de corte la manera de expresar los elementos a capturar se puede hacer mediante XPath o mediante Expresiones Regulares de forma indistinta. XPath ofrece mayor potencia que las expresiones regulares, pero es más complicado de utilizar al requerir un mayor conocimiento del modelo de objetos empleado en el sistema.

El proceso de tejido comienza con el desensamblado de la aplicación base y del aspecto generando, a partir del código intermedio, un árbol que representa a este código (un árbol para el código base y otro para el aspecto) siguiendo una versión reducida del modelo de objetos de Cecil.

En las primeras versiones de AspectDNG esta representación en árbol se volcaba a ficheros XML, en un formato que internamente llamaban ILML. Sobre estos ficheros se ejecutaban las consultas XPath o se validaban las expresiones regulares para determinar qué puntos de enlace se debían enlazar con qué aspectos. Por cuestiones de rendimiento abandonaron esta técnica y en las siguientes versiones trabajan directamente en memoria simulando la ejecución de consultas contra el árbol de objetos.

Una vez determinados los puntos de enlace y aspectos que deben tejerse, se procede a fusionar el árbol de la manera adecuada, dando lugar a un nuevo árbol que tiene la funcionalidad completa. A partir de este árbol se genera el ensamblado ya tejido (esto lo hace Cecil).

Este proyecto presenta un conjunto de puntos de enlace no muy amplio. Además sólo existe la posibilidad de crear *advice* de tipo *around*, pese a ello extender la funcionalidad a otros puntos de enlace o tipos de *advice* no es difícil con la arquitectura presentada.

El sistema es independiente del lenguaje por lo que los aspectos y la aplicación base pueden estar escritos en cualquier lenguaje (con la única restricción de tener que ser lenguajes compatibles con la plataforma .NET) y, al trabajar sobre código intermedio, no es necesario disponer del código fuente de la aplicación base.

El poder expresar los puntos de corte mediante un fichero externo permite un alto grado de reutilización de código ya que no hay ninguna dependencia entre la aplicación base y el aspecto (al contrario que en otros sistemas que en la definición del aspecto hacen referencia a la aplicación base a la que desean modificar). De igual forma el que el sistema no imponga ninguna restricción a la aplicación base ni al aspecto, facilita esta reutilización de código ya que no hay ninguna dependencia respecto al sistema (no tienen que utilizar ninguna librería especial ni hacen uso de extensiones del lenguaje).

5.1.6 Aportaciones y Carencias de los Sistemas Estudiados

Todos los sistemas vistos en esta sección son estáticos (AspectJ, en su versión 5 soporta cierto grado de dinamismo, pero es un dinamismo de instanciación retardada de aspectos, no de adaptación ante nuevos aspectos), por lo que no cumplen con el dinamismo exigido en esta Tesis (1.2.1). Aún así se han querido mostrar aquí ya sea por la importancia que tienen, como es el caso de AspectJ, o por las aportaciones que realizan en otras áreas, en el caso del resto de sistemas.

Respecto al requisito de la independencia del lenguaje (2.1.2) AspectJ es dependiente de Java, y de hecho, ha realizado una extensión al lenguaje, con lo que es necesario el uso de compiladores especiales que soporten dichas extensiones. No soporta la utilización de otros lenguajes de programación. El resto de sistemas están basados en la plataforma .NET de Microsoft, que es independiente del lenguaje, por lo que los sistemas se benefician de ello. La independencia de DotSPECT respecto al lenguaje se ve

reducida por la inclusión en el mismo fichero del código de los puntos de corte (en un lenguaje definido al efecto) y el de los aspectos, por lo que es necesario contar con un preprocesador que sea capaz de separar ambas partes. En la actualidad el sistema soporta sólo dos lenguajes.

El conjunto de puntos de enlace que soporta AspectJ es considerado el estándar, por lo que cumple perfectamente con el requisito 2.1.4. Weave.Net y SourceWeave.Net tienen el objetivo de soportar el mismo conjunto de puntos de enlace que AspectJ, por lo que también cumplen el requisito (si bien hay que tener en cuenta que las implementaciones existentes de ambos sistemas distan mucho de ofrecer todos los puntos de enlace teóricamente soportados). El resto de sistemas presentan un conjunto de puntos de enlace mucho más pobre, limitando igualmente las formas de adaptar cada punto de enlace.

Los entornos sobre las que se han implementado todos los sistemas vistos son independientes de la plataforma sobre la que se ejecuten, por lo que el requisito de independencia de la plataforma (2.2.1) debería cumplirse. Sin embargo, dotSPECT y SourceWeb.NET hacen uso de una serie de librerías no estándar de la plataforma .NET que sólo están disponibles en la implementación realizada por Microsoft, incurriendo en una dependencia del sistema operativo.

La separación del código de los aspectos y de los puntos de corte (requisito 2.4.3.2) no se produce ni en AspectJ ni en DotSPECT, por lo que la reutilización de código en estos sistemas no es inmediata. Además AspectJ ha realizado extensiones al lenguaje, con lo que el aprovechamiento de código fuera de su propio sistema es aún más complicado.

5.2 Sistemas Dinámicos

A continuación vamos a estudiar una serie de sistemas dinámicos, centrándonos en los más difundidos y en aquellos que aportan mejores prestaciones.

5.2.1 Aspectwerkz

AspectWerkz [AspectWerkz] es un proyecto de código abierto con licencia GNU LGPL [GNULGPL]. A principios de 2005 este proyecto se unió al proyecto AspectJ con el fin de dotar a éste último con las características dinámicas del primero. Es un sistema basado completamente en Java.

El sistema permite realizar tejido estático, en tiempo de carga y en ejecución. Aquí vamos a centrarnos únicamente en la forma dinámica de tejido.

El conjunto de puntos de enlace que soporta es el mismo que el de AspectJ, al igual que el conjunto de tipos de *advice* (*before, after y around*), por lo que ofrece la misma potencia que AspectJ pero de una forma dinámica (parte de esta potencia está incluida en AspectJ a partir de su versión 5, como resultado de la unión con este proyecto).

Existen dos formas de definir los puntos de corte:

- Mediante un fichero XML externo al código de los aspectos, en el que se definen los puntos de corte y el código al que se refieren.
- Mediante anotaciones de Java. Aquí se permiten las anotaciones de Java 5 y las anotaciones como comentarios de Javadoc (para versiones anteriores de la máquina virtual).

Ambas formas de definir los puntos de corte son semánticamente equivalentes y se puede traducir fácilmente de la una a la otra. La utilización del fichero XML externo al aspecto presenta una ventaja respecto a las anotaciones, que es una menor dependencia, al no haber nada en el código del aspecto que se refiera al código base, por lo que su posible reutilización para ser tejido con otra aplicación base, o para ser usado en otro sistema de POA sería más sencilla.

La manera de realizar el tejido es mediante instrumentación de código en tiempo de carga [Vasseur04] (por lo que el tejido se realiza sobre *bytecode* de Java). Para ello, es necesario sustituir el cargador de clases (*classloader*), o ampliar su funcionalidad, para que se encargue de añadir el código necesario en los lugares adecuados. Para realizar esta sustitución del cargador de clases existen diversos mecanismos, dependiendo de la versión de la máquina virtual que se emplee o de las distintas implementaciones de la misma que se usen.

Una vez que se ha sustituido el cargador de clases, cada vez que se debe cargar una clase en la máquina virtual es procesada por el cargador modificado y, mediante instrumentación de código, se reescribe de tal forma que se le añada el código necesario en las sombras de los puntos de enlace (*JoinPoint Shadow)* seleccionados en función de los puntos de corte definidos. Este código modificado es el que se pasa a la máquina virtual para que sea utilizado en vez del original.

AspectWerkz soporta dos formas de desplegar un aspecto. La primera consiste en que el aspecto esté activo desde el momento en que se carga la clase a la que afecta. La segunda forma, llamada despliegue en caliente –*hot deployment*–, consiste en que en el momento de la carga de la clase lo que se hace es prepararla para que el aspecto pueda ser activado o desactivado posteriormente en ejecución. También es posible reutilizar el tejido realizado para un aspecto A por otro aspecto B. Lo que no es posible, es desplegar un aspecto sobre una clase que no haya sido preparada previamente, ya que las clases se cargan únicamente la primera vez que se utilizan quedando almacenadas en memoria para su uso posterior, y una vez cargadas no es posible acceder a su código para modificarlo [Vasseur04].

Los aspectos son clases normales de Java que no tienen que cumplir ningún "protocolo" especial (por ejemplo heredar de alguna clase, implementar alguna interfaz, etc.). La única restricción que tienen es que deben tener constructor sin parámetros (o no tenerlo, en cuyo caso se utiliza el constructor por defecto estándar de Java) o un constructor con un parámetro de tipo `AspectContext`.

Este sistema presenta una serie de ventajas como son su amplio conjunto de puntos de enlace (que es equiparable al de AspectJ) y su eficiencia [Boner04]. La posibilidad de definir los puntos de corte en un fichero XML externo al aspecto es una ventaja a la hora de reutilizar el código de los aspectos, ya sea en otra aplicación o en otro sistema

de POA, al no tener ninguna referencia en el código del aspecto a la aplicación base, con lo que no existe dependencia.

El sistema también presenta algunas limitaciones como el hecho de estar basado en Java, lo que implica que, tanto el código de la aplicación base como el de los aspectos, debe estar implementado en este lenguaje. Esto limita su uso en sistemas ya implementados [Li04] (cualquier aplicación existente implementada en otro lenguaje no es susceptible de ser utilizada en este sistema, o en caso de necesitar hacerlo habría que traducirla a Java, para lo que sería necesario disponer de su código fuente).

Otra limitación es la obligación de que los aspectos tienen que cumplir con una signatura predefinida en los constructores, lo que resta posibilidades de reutilización directa en otros sistemas [Rajan05]. Aunque el sistema presenta características dinámicas, mediante las que se puede activar y desactivar un aspecto en ejecución, éstas se ven limitadas por el hecho de tener que haber sido previstas en el diseño con el fin de que en el momento de la carga de la clase se haya preparado para esta activación [Belapurkar04]. Esto impide que el sistema pueda adaptarse a requerimientos no previstos en el momento del diseño.

5.2.2 PROSE

El sistema PROSE (PROgramable extenSions of sErvices) [Popovici01] [Popovici02], ofrece POA de forma dinámica, permitiendo adaptar en tiempo de ejecución una aplicación sin necesidad de haber definido los aspectos en el momento del diseño.

Su plataforma está implementada en Java, siendo el lenguaje de codificación de los aspectos y de las aplicaciones Java. Provee al usuario con un subconjunto de las características esenciales de la POA [Popovici02].

Los aspectos en PROSE deben heredar de una clase abstracta llamada DefaultAspect. Cada aspecto contiene una serie de objetos de *crosscut*, que es como llaman los autores a la unión de un punto de corte y su *advice* asociado (en terminología AspectJ). El conjunto de puntos de enlace que soporta actualmente PROSE incluye la ejecución de métodos, su redefinición, lectura o escritura de campos y tratamiento de excepciones.

En la actualidad existen tres diferentes implementaciones del motor de ejecución de PROSE.

En la primera para implementar el sistema se ha utilizado la interfaz de depuración de la máquina virtual de Java, *JVM Debugger Interface* (JVMDI) y se ha creado un plug–in (añadido) para la JVM de tal forma que soporte el concepto de aspecto de forma directa. Este plug–in recibe el nombre de interfaz de aspectos de la JVM (*JVM Aspect Interface*, JVMAI) que es el que permite al usuario el tejido dinámico de aspectos.

Con posterioridad [Popovici03] y con la intención de ofrecer mejor rendimiento en ejecución, se ha modificado el compilador "justo a tiempo" –*just in time*– (JIT) de la máquina virtual de investigación Jikes de IBM (IBM Jikes Research Virtual Machine) [IBM2003] [Jikes], haciendo que esta JVM soporte de forma directa la POA, y mediante una API, *Application Program Interface* (interfaz de programación de aplicaciones), se ofrecen puntos de enlace a la capa superior.

La versión actual [Nicoara05] es también una modificación de la maquina virtual Jikes. En este caso se ha modificado la máquina virtual para permitir la sustitución de código en ejecución. Cuando se va a ejecutar un punto de enlace se comprueba si se ve afectado por un punto de corte, en caso afirmativo se modifica el código del punto de enlace (se añaden las llamadas necesarias para ejecutar el advice) y es este código modificado el que se envía al compilador *JIT*. De este modo, sólo son tejidos aquellos puntos de enlace que han sido seleccionados. El tratamiento de puntos de enlace de excepciones se ha implementado mediante otra modificación de la máquina virtual. En este caso se ha modificado el gestor de excepciones en tiempo de ejecución de tal manera que se encargue de realizar las llamadas necesarias a los *advice* afectados según la definición de los puntos de corte.

Una limitación que presenta [Popovici02] es la imposibilidad de añadir nuevos miembros a una clase en el código original. Al tener que utilizar Java como lenguaje base se tiene dependencia del lenguaje, lo que imposibilita su uso para programas no escritos en Java. En principio, se podría utilizar cualquier lenguaje para implementar los aspectos, pero el lenguaje debería hacer uso de las librerías del sistema y ser compatible con la plataforma.

Otra limitación que presenta el sistema es el reducido conjunto de puntos de enlace que soporta. Por ejemplo, no soporta invocaciones a métodos. Además el *around*, aunque está soportado por la última versión, no permite la ejecución del código original (el *proceed()* en AspectJ).

Las dos últimas versiones obtienen buenos resultados en cuanto a tiempos de ejecución, pero siguen teniendo las mismas limitaciones que la versión anterior (dependencia del lenguaje y puntos de enlace limitados) y le añaden una nueva: la necesidad de utilizar una JVM modificada, no una estándar. También reconocen los autores que el modelo de puntos de enlace que soporta tiene limitaciones y por ejemplo no permite añadir *advice* (código) a niveles de objeto. Aunque se permite que varios aspectos modifiquen el mismo punto de enlace, no se permite que los aspectos interactúen entre sí, es decir, un aspecto no puede ser modificado por otro aspecto.

5.2.3 CLAW

El sistema CLAW (*Cross–Language Load–Time Aspect Weaving*, tejido de aspectos en tiempo de carga de lenguajes cruzados) [Lam02], inicialmente llamado RAW (*Runtime Aspect Weaver*, tejedor de aspectos en tiempo de ejecución), está implementado sobre la plataforma .NET. El trabajo se realiza a nivel de código intermedio (IL), obteniendo con ello un gran beneficio: la independencia del lenguaje.

En la plataforma .NET cualquier programa escrito en cualquier lenguaje de programación es traducido al código intermedio (IL), que es el que se ejecuta por parte del CLR (Commom Language Runtime). Al introducir los aspectos a este nivel se consigue que cualquier programa escrito en cualquier lenguaje pueda ser adaptado por aspectos escritos en otros lenguajes, ya que al final ambos son compilados a IL y es en este nivel donde son tejidos.

Como se puede ver por el nombre inicial (RAW) y por el definitivo (CLAW) en un principio se argumentaba que este sistema ofrecía tejido en tiempo de ejecución para, posteriormente afirmar que el tejido es en tiempo de carga.

En verdad en este sistema, aunque los aspectos se tejen en tiempo de ejecución, tienen que estar definidos previamente (en tiempo de diseño) lo que implica que el sistema no puede adaptarse ante nuevos requerimientos no previstos en el diseño.

La forma de trabajar de este sistema es utilizar la API de *profiling* (perfilar) que provee .NET. En concreto las dos siguientes interfaces: `IcorProfilerCallback` e `IcorProfilerCallbackInfo`. Estas interfaces funcionan mediante un sistema basado en eventos. Cuando el sistema de aspectos empieza a ejecutarse informa al motor de ejecución de los eventos que quiere monitorizar (los eventos más importantes a monitorizar serian la carga de módulos, carga de "*assemblies*" –ficheros ensamblados ejecutables– o la compilación justo a tiempo –*just in time*– JIT).

En tiempo de ejecución el motor de ejecución llamaría al sistema cuando se produjesen esos eventos, y éste se encargaría (mediante reflexión) de examinar el IL existente, adaptarlo según sea necesario, y escribir el nuevo IL adaptado en memoria para que el compilador JIT se encargue de compilarlo y ejecutarlo.

El principal inconveniente de esta forma de trabajar es que el sistema tiene que estar en modo *profiling*, lo que implica una penalización en el rendimiento. Para obtener la reflexión necesaria el autor ha utilizado las interfaces de metadatos no gestionados (*Unmanaged Metadata Interfaces*) las cuales son un conjunto de interfaces COM que están accesibles desde fuera del entorno .NET lo que hace que no sea portable al ser tecnología propia de Microsoft.

5.2.4 AOPEngine.Net

Este sistema [Frei04] fue desarrollado como un prototipo por el mismo centro de investigación que desarrolla PROSE, intentando obtener un sistema similar con la ventaja añadida de la independencia del lenguaje al trabajar sobre la plataforma .NET. El prototipo se publicó en marzo de 2004 y desde entonces el trabajo parece abandonado. Para desarrollar el sistema se basaron en las ideas de PROSE y las de CLAW (vistos anteriormente).

Los autores afirman que existen sistemas estáticos y sistemas dinámicos que ofrecen POA. Entre estos últimos se puede distinguir los que realizan el tejido en tiempo de carga y los que lo realizan en tiempo de ejecución. La mayoría de los sistemas dinámicos existentes realizan el tejido en tiempo de carga y, aunque en algunos es posible activar y desactivar los aspectos en tiempo de ejecución, no es posible añadir un aspecto en un lugar que no hubiese sido previsto y tejido durante la carga. La única solución para poder realizar esto, es introducir *hooks* en todos los posibles puntos de enlace de la aplicación, para que en un futuro pudiesen ser usados por un hipotético nuevo aspecto. Los autores afirman que esta solución es inviable por el coste en tiempo que supone.

El sistema propuesto no introduce código alguno en la aplicación base hasta que no sea necesario (se introduce en el momento que un aspecto solicita ser tejido), por lo

que no se incurre en penalizaciones en el rendimiento por "posibles" aspectos no implementados.

La arquitectura del sistema se basa en el uso del depurador y del *profiler* de .Net y la existencia de un AOPController (controlador de aspectos). Los aspectos, que vienen expresados en forma de dll en la que está incluido el código del *advice* así como la definición de los puntos de corte, son procesados por el controlador que es el que interactúa con el depurador y el *profiler*.

Cuando una aplicación está ejecutándose el sistema informa al *profiler* de la ocurrencia de ciertos eventos solicitados. En este caso, el evento que es informado es la compilación *JIT,* que se lleva a cabo la primera vez que se va a ejecutar un método. El *profiler* recibe la notificación de que se va a proceder a compilar un método y se examina si ese método verifica alguno de los puntos de corte solicitados. En caso afirmativo, a través del depurador y del *profiler,* se genera código adicional para el método (se teje) que es el que se pasará al compilador para que lo compile y se continúe la ejecución de la aplicación, pero con la funcionalidad tejida.

De este modo la aplicación no es aumentada mientras no sea necesario. La máquina virtual debe ejecutarse en modo depuración y con el *profiler* activado, lo que impone una penalización en el rendimiento, pero los autores argumentan que es menor que la que se incurre en el caso de inyección masiva de *hooks.* Los autores han realizado [Frei04] una comparativa de rendimiento con el sistema JAsCo.Net [Verspecht03] resultando claramente más veloz en el caso de que no haya aspectos enlazados (aunque en el caso de JAsCo.Net la aplicación ya estaba preparada para los futuros aspectos), pero en el caso de la existencia de *advice* de tipo *before y around* es más lento, por lo que no puede inferirse que realmente el sistema ofrezca un mejor rendimiento.

Cuando un aspecto solicita ser eliminado (destejido) de la aplicación, el sistema marca el método como que debe ser recompilado. Al realizarse esta operación se vuelve al código inicial sin el tejido del aspecto.

Este sistema presenta un enfoque dinámico y multilenguaje. El conjunto de puntos de enlace que soporta es muy reducido, pudiendo únicamente capturar las ejecuciones de métodos (puntos de enlace tales como el acceso a campos o el tratamiento de excepciones no pueden ser controlados por este sistema) lo que reduce mucho la potencia del sistema.

Al igual que ocurría en el caso de CLAW, puesto que está basado en las mismas API, se hace uso de interfaces de metadatos no gestionados (*Unmanaged Metadata Interfaces*) las cuales son un conjunto de interfaces COM que están accesibles desde fuera del entorno .NET, lo que hace que no sea portable, al ser tecnología propia de Microsoft.

El hecho de que el código de los aspectos y de los puntos de enlace se encuentre unido restringe las posibilidades de reutilización de los mismos.

5.2.5 LOOM.Net (Rapier–LOOM.Net)

Este sistema, desarrollado en el Hasso Plattner Institute, ofrecía en principio un tejido estático de aspectos [Schult02] sobre la plataforma .Net de Microsoft pero, con posterioridad [Schult02b], [Schult03] y [Schultz03b] ha sido desarrollado como un entorno dinámico llamado Rapier–LOOM.Net.

Rapier se basa en el uso de los *custom attributes* (atributos personalizados) de la plataforma .Net para indicar las reglas de tejido al tejedor. Éste, mediante reflexión y acceso a los metadatos de la plataforma, los evalúa en ejecución realizando el tejido pertinente. El proceso de tejido se realiza sobre código intermedio por lo que se consigue independencia del lenguaje, al igual que en los casos anteriormente presentados.

Los aspectos deben derivar de la clase Loom.Aspect. Respecto a las clases base deben cumplir una serie de requisitos para poder ser utilizadas en el sistema: sus métodos deben ser virtuales o deben definirse mediante una *interface*. Este requisito impone otra restricción al sistema como es la imposibilidad de utilizar el operador `new` (o el operador o instrucción equivalente en el lenguaje en el que se haya implementado la aplicación que se esté adaptando), y usar en su lugar la llamada al método `Loom.Weaver.CreateInstance`, para crear instancias de las clases que pueden ser tejidas.

El tejido se realiza en el momento de instanciación de la clase, quedando la clase tejida hasta la finalización de la ejecución del programa, es decir, no se puede eliminar en ejecución un aspecto que ya esté tejido.

Un aspecto debe indicar, mediante los atributos personalizados, a qué punto de enlace de la aplicación base debe ser asociado. Es decir, debe especificar el punto de corte. En la implementación actual sólo se permiten las invocaciones a métodos o constructores, con los tiempos típicos *before, after y around* (en este caso llamado *instead*).

El sistema presenta numerosas limitaciones. En primer lugar es necesario disponer del código fuente de la aplicación base para adaptarla a los requisitos impuestos (los métodos de las clases a tejer deben ser virtuales o definirse mediante una *interface* y no se puede utilizar el operador `new` para la creación de instancias de estas clases). Esto tiene implicaciones negativas en las posibilidades de reutilización del código, ya que es necesario modificar la aplicación para que funcione en este sistema invalidándola así para poder ser utilizada directamente, sin cambios, en otros sistemas o sin el uso de aspectos.

Otra limitación es el reducido conjunto de puntos de enlace que soporta. En la actualidad sólo soporta la invocación a métodos, pero no soporta ejecución de métodos, acceso a campos, etc. También es una limitación el hecho de que, aunque se permite que más de un aspecto modifique el mismo punto de enlace de forma simultánea, no existe una forma de definir su orden de ejecución (viene fijado por el orden de creación lo que no es suficiente).

De igual modo, la especificación de las reglas de tejido mediante *custom attributes* en el código de los aspectos crea un acoplamiento del código dificultando la reutilización del mismo.

Por último, aunque el tejido se realiza en ejecución, consiste únicamente en la posibilidad de tejer (o añadir) los aspectos en el momento de instanciación de las clases. No se permite el tejido de aspectos no contemplados en tiempo de diseño o la eliminación de un aspecto ya tejido, es pues una característica limitada que no permite un verdadero dinamismo (entendido como adaptación a nuevos requisitos no previstos en el diseño).

En Junio de 2007 se ha publicado la versión nueva versión de la herramienta (2.0) que soluciona algunas de las limitaciones existentes según los autores, pero centran su trabajo en ofrecer un mejor rendimiento más que en ofrecer mejor funcionalidad.

5.2.6 Proyecto AORTA

El sistema AORTA (*Aspect Oriented RunTime Architecture*, arquitectura orientada a aspectos en tiempo de ejecución) [AORTA] pretende ofrecer un entorno con programación orientada a aspectos en tiempo de ejecución. Está siendo desarrollado por el mismo grupo que ha desarrollado CAESAR [Mezini03] (que es una extensión a Java para ofrecer orientación a aspectos en el mismo sentido que AspectJ).

El proyecto AORTA ha desembocado en tres posibles sistemas, cada uno más sofisticado que el anterior. El primer sistema, conocido como Axon [Axon][Haupt02] fue desarrollado sobre la máquina virtual de Java, JVM, haciendo uso de la interfaz de depuración (JVMDI). Cuando se añade un aspecto al sistema se avisa al depurador para que lance un evento en el caso de que se alcance el punto de enlace, el evento es recibido por el gestor que deriva la ejecución al código del aspecto. Según los autores este sistema era una prueba de concepto y presentaba diferentes limitaciones tanto en su rendimiento como en el conjunto de puntos de enlace, o tipos de *advice* soportado (por ejemplo el sistema no soporta el tipo *around).*

Posteriormente desarrollaron el sistema RuByCoM [Danker04] que hace uso del HotSwap de la máquina virtual de Java, mediante el que pueden redefinir clases en tiempo de ejecución. El sistema, haciendo uso de la librería BCEL [BCEL], instrumenta el código de la aplicación base, añadiéndole la funcionalidad del aspecto y sustituye, mediante el HotSwap, la clase original por la clase con el código añadido. Este sistema presenta un conjunto de puntos de enlace más amplio que el anterior y también soporta *advices* de tipo *around.*

El último sistema implementado se llama SteamLoom [Bockisch04] [Haupt06], y ha sido implementado como una modificación de la máquina virtual Jikes [Jikes] [IBM2003]. Han extendido la máquina virtual con la librería BAT [BAT] para realizar instrumentación de código de forma dinámica.

Mediante las modificaciones realizadas en la máquina virtual, en el momento de la carga de clases se realiza una primera instrumentación de código. Posteriormente se detecta la ejecución de los diversos puntos de enlace donde se haya solicitado añadir funcionalidad por parte de los aspectos. En ese momento se obtiene el código original del método, se instrumenta de acuerdo a lo solicitado, y se vuelve a compilar para que la ejecución se realice sobre el código modificado. El sistema impone una restricción consistente en que los aspectos sean instancias de la clase Aspect.

SteamLoom presenta un conjunto de puntos de enlace rico, similar en potencia al que puede tener AspectJ. El sistema permite la activación, desactivación y despliegue de aspectos en ejecución.

El sistema presenta ciertas limitaciones que chocan con los requisitos establecidos. En primer lugar está basado en Java, por lo que tanto las aplicaciones base como los aspectos tienen que ser implementados en este lenguaje. El hecho de utilizar una máquina virtual modificada específicamente para el sistema también representa problemas de dependencia de la plataforma. De igual modo, el sistema no contempla un mecanismo para la gestión del orden de ejecución de distintos aspectos cuando se encuentren enlazados en el mismo punto de enlace [Haupt04] y, en el caso de *advice* de tipo *around,* sólo se permite uno (no puede haber varios *advice* de este tipo activos a la vez) en cada punto de enlace. Además, el obligar a que los aspectos sean instancias de la clase Aspect tiene inconvenientes tanto para su reutilización en otros sistemas como problemas de implementación en un lenguaje como Java, que no permite herencia múltiple [Rajan06].

5.2.7 Reflex

Reflex se define como un sistema versátil para obtener POA multilenguaje [Tanter05]. Inicialmente apareció como un sistema que ofrecía reflexión computacional en Java [Tanter01]. El sistema fue evolucionando y los autores se dieron cuenta de las conexiones existentes entre la reflexión y la POA y debido a esto propusieron [Tanter04] un núcleo para la POA.

Lo que ofrece Reflex es un conjunto de facilidades para implementar lenguajes orientados a aspectos (posiblemente de dominio específico), de tal manera que se puedan componer independientemente del lenguaje en que estén definidos. Se basa en la plataforma Java y en el uso de reflexión e instrumentación de código.

La arquitectura del sistema consta de tres capas:

- Capa de transformación, en la que se añade reflexión estructural y de comportamiento a Java [Tanter03]
- Capa de composición, en la que se produce el tejido (composición) de los aspectos [Tanter06a].
- Capa de lenguaje, mediante la cual se pueden crear nuevos lenguajes de dominio específico que pueden ser embebidos dentro del código de Java [Tanter06b]. Para esto se basan en MetaBorg [MetaBorg]. MetaBorg es un sistema que permite embeber lenguajes de dominio específico en lenguajes de propósito general traduciendo posteriormente las instrucciones embebidas a llamadas a una API del lenguaje anfitrión.

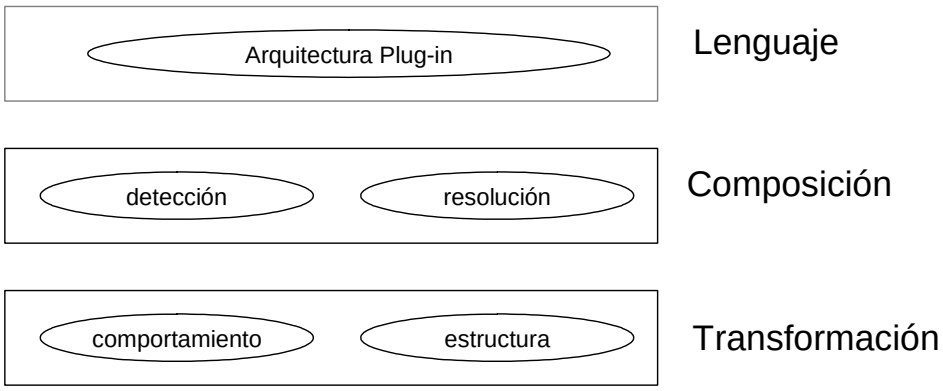

Figura 12: Arquitectura de Reflex

El sistema está implementado como una librería de clases Java para conseguir portabilidad.

En la capa de transformación, mediante instrumentación de código, se añade el comportamiento reflectivo y una serie de puntos donde se podrá enlazar comportamiento (*hooks* en la nomenclatura Reflex). Realmente lo que se realiza en esta capa es la inclusión de un MOP[6] *–Meta Object Protocol–*.

En la capa de composición se detectan y resuelven los puntos de corte (llamados *links*), enlazando los *hooks* introducidos en la capa anterior con el código del aspecto.

La capa de lenguaje inicialmente estaba concebida como componentes añadidos (*plug–in*) al sistema, de tal forma que soporten un lenguaje de aspectos y se encarguen de generar la configuración necesaria para Reflex. En la actualidad se basan en MetaBorg para permitir lenguajes embebidos en Java. De esta forma los aspectos puedan programarse en Java con código embebido, y el sistema se encarga de transformar esto al lenguaje Java con llamadas a la API de Reflex que pueden ser procesadas por la capa de composición para realizar el tejido.

El tejido se realiza en tiempo de carga, por lo que el código añadido permanece durante toda la ejecución de la aplicación. Lo que si que se puede hacer es activar o desactivar en ejecución los *hooks* inyectados, con lo que se tiene la posibilidad de añadir o eliminar comportamiento en ejecución.

Aunque el sistema permite múltiples lenguajes de aspectos, todos ellos estarán embebidos en Java, y la aplicación base debe estar implementada en Java, por lo que no existe realmente una independencia del lenguaje (entendida como que la aplicación y los aspectos pueden ser implementados en cualquier lenguaje).

Además, el código de los aspectos es muy dependiente del sistema, debido al uso de lenguajes específicos cuyo código irá embebido en Java, por lo que su reusabilidad fuera de este entorno es baja.

[6] Veremos más adelante el uso de los MOP en los capítulos dedicados a la reflexión.

5.2.8 Microdyner (μDyner)

Este sistema, presentado en [Segura–Devillechaise03] y [Segura–Devillechaise03b], surge en respuesta a la necesidad que encuentran los autores de poder implementar distintas políticas de "precarga en memoria" –*cache prefetching*– en los servidores Web de forma dinámica. Los autores han identificado la POA como una magnífica herramienta para poder implementar diversas políticas que puedan ir cambiando para adaptarse a requerimientos que varían en el tiempo.

La razón para crear un nuevo sistema es que la gran mayoría de los sistemas no ofrecen POA dinámica sino estática, y un servidor Web no puede detener su ejecución para responder a cambios en el entorno, con lo que no es válido su uso. Otra razón es que los sistemas que ofrecen POA de forma dinámica no lo hacen para el lenguaje C, que es en el que están escritos la mayoría de los servidores Web, con lo que tampoco son aplicables.

La forma de trabajar con este sistema implica que en el programa fuente (el que va a ser adaptado) es necesario indicar qué partes pueden ser adaptadas (es decir, en el diseño e implementación del programa a ser adaptado hay que tener en cuenta la posibilidad de una futura adaptación). Sólo las funciones y variables globales pueden ser candidatas a ser adaptadas. El programa pasa por un preprocesador antes de pasar por el compilador.

Los aspectos se escriben en un lenguaje específico del sistema definido a tal efecto, son compilados mediante μDyner y posteriormente tejidos en tiempo de ejecución.

El tejido se realiza directamente sobre la imagen ejecutable del programa base generando código ejecutable. El sistema permite realizar el tejido o el destejido (eliminación de un aspecto) en muy poco tiempo, por lo que el funcionamiento del servidor Web no se ve interrumpido.

En resumen, este sistema permite crear aspectos únicamente para programas escritos en el lenguaje C, que pueden ser tejidos y destejidos en tiempo de ejecución, presentando los inconvenientes de tener que definir en el programa a adaptar qué es lo que puede ser adaptado y qué no, utilizar un lenguaje específico y sólo poder adaptar funciones y variables globales. Además al realizar el tejido o destejido directamente sobre el código ejecutable se incurre en una dependencia de la plataforma (actualmente el sistema está implementado para un Intel Pentium).

En la actualidad este proyecto ha evolucionado y ahora se llama Arachne [Arachne] siguiendo la misma filosofía aquí presentada.

5.2.9 EAOP

El nombre de este sistema viene de *Event–Based AOP* (POA basada en eventos) [Dounce01]. La propuesta que se hace en [Dounce02] y, posteriormente, en [Dounce04] es un modelo teórico mediante el que se puede conseguir POA basada en eventos y una herramienta que lo implementa en Java.

La EAOP tiene los siguientes conceptos:

- Eventos: que se generan durante la ejecución del programa con el propósito de designar "puntos de interés" –*"points of interest"* o puntos de ejecución.
- *Crosscuts*: que son secuencias de eventos y relaciones entre ellos.
- Monitorización en ejecución de los eventos.

Las principales características que presenta son:

- Posibilidad de combinar aspectos de forma explícita.
- Posibilidad de crear aspectos que adapten a otros aspectos.
- Gestión dinámica de aspectos.

Inicialmente el modelo pretende ser un banco de pruebas para definir lenguajes orientados a aspectos (cualquier lenguaje). Se basa en la monitorización de eventos de ejecución.

Tiene dos lenguajes, el de aspectos, y otro para definir los puntos de ejecución (que son como los *pointcuts* de AspectJ). En principio se puede elegir cualquier lenguaje para que actúe como lenguaje de aspectos pero, una vez seleccionado uno, todos los aspectos tienen que estar en ese lenguaje.

El sistema permite instanciación y composición dinámica de aspectos, pero los aspectos se deben definir en diseño.

El sistema, mediante un preprocesador, procesa el código fuente del programa base y modifica el código mediante *method wrapping* insertando llamadas a sus librerías que son las que permiten gestionar los aspectos.

Los principales inconvenientes de este sistema son la dependencia del lenguaje, puesto que pese a que el sistema pretende ser independiente, en cuanto se ha elegido un lenguaje es necesario seguir utilizándolo, la necesidad de disponer del código fuente para poder adaptar a un programa, y la necesidad de haber definido los aspectos en el diseño, aunque sean instanciados posteriormente. Otra limitación muy importante es el muy restringido conjunto de puntos de enlace que soporta (llamada y retorno de un método o de un constructor), lo que limita mucho los aspectos a implementar. Tampoco permite concurrencia en la ejecución de aspectos.

Como implementación de esta aproximación los autores presentaron Arachne [Dounce05] que es una extensión al lenguaje C sobre plataformas Linux y es una extensión y modificación del proyecto microDyner (visto anteriormente).

5.2.10 Aportaciones y Carencias de los Sistemas Estudiados

Los sistemas estudiados en esta sección ofrecen muy distintos grados de dinamismo. Unos sistemas ofrecen la posibilidad de adaptar una aplicación de forma completamente dinámica, activando y desactivando aspectos en tiempo de ejecución sin que esos aspectos tengan que haber sido previstos durante el desarrollo del sistema tal y como se plantea en los requisitos de esta Tesis (2.1.1) (PROSE, AOPEngine.NET). Otros sistemas restringen esta posibilidad permitiendo activar y desactivar únicamente aspec-

tos que hayan sido definidos durante el desarrollo, por lo que no pueden adaptarse a cambios no previstos inicialmente (CLAW, AspectWerkz, MicroDyner, EAOP). Otros sistemas restringen aún más esta posibilidad no permitiendo la desactivación de aspectos, únicamente su activación (Rapier LOOM.Net).

De entre los sistemas estudiados sólo aquellos que están basados en la plataforma .NET (CLAW, AOPEngine.Net y Loom.Net) son independientes del lenguaje (requisito 2.1.2). Reflex es un sistema multilenguaje, pero los lenguajes que se pueden emplear deben ser definidos de forma embebida dentro de Java, por lo que no es realmente independiente del lenguaje. El resto de sistemas está diseñado para soportar un único lenguaje de programación, en el caso de MicroDyner el lenguaje de la aplicación es C y el de los aspectos es un lenguaje definido al efecto, mientras que en el resto de sistemas el lenguaje es Java. EAOP es independiente del lenguaje, pero en el momento en el que se elige un lenguaje para implementar un componente (ya sea la aplicación o los aspectos que la adaptan) todos los demás componentes tienen que implementarse en el mismo lenguaje, por lo que no es independencia real.

En general el conjunto de puntos de enlace que soportan los sistemas dinámicos es muy reducido en comparación con los sistemas estáticos. Únicamente AspectWerkz y SteamLoom (del proyecto AORTA) soportan un conjunto similar al de AspectJ. PROSE soporta casi todos los puntos de enlace requeridos, a excepción de la invocación a métodos, pero con la arquitectura utilizada esto puede ser resuelto. El resto de sistemas limitan en mayor o menor medida los puntos de enlace disponibles. Esto viene dado muchas veces por limitaciones de la implementación realizada. Por ejemplo tanto CLAW como AOP.Engine.NET se basan en el uso del *profiler* de .NET para ser informados del alcance de puntos de enlace durante la ejecución de la aplicación, pero éste sólo informa de invocaciones a métodos, por lo que el acceso a un campo no es soportado por el sistema como punto de enlace.

De todos los sistemas vistos AspectWerkz, PROSE, Rapier LOOM.Net, SteamLoom y Reflex son realmente independientes de la plataforma (requisito 2.2.1). CLAW y AOP Engine.Net hacen uso de una característica de la plataforma .NET que está disponible únicamente en las implementaciones realizadas en sistemas operativos Windows, por lo que pierden esta independencia. MicroDyner realiza el tejido de aspectos modificando el fichero ejecutable de la aplicación, por lo que es totalmente dependiente de la plataforma.

PROSE y SteamLoom han realizado sus implementaciones modificando la máquina virtual de Java, por lo que su portabilidad se ve limitada al no utilizar un sistema estándar (requisito 2.2.4).

Tanto MicroDyner, como Rapier LOOM.NET necesitan disponer del código de la aplicación a adaptar para poder realizar el tejido. Esto es un inconveniente muy grave pues impide su utilización con aplicaciones de las que no se disponga de este código (por ejemplo aplicaciones de terceros, COTS, etc.) que es uno de los requisitos establecidos (requisito 2.3.1).

5.3 Servidores de Aplicaciones

A continuación vamos a comentar sistemas que ofrecen soporte a la POA en el ámbito de los Servidores de Aplicaciones.

5.3.1 JBoss AOP

JBoss AOP [JBossAOP] es una parte del proyecto JBoss [JBoss] que es parte de RedHat. JBoss AOP se ofrece con una serie de licencias de código abierto de tal forma que el usuario puede escoger la licencia que más le convenga de acuerdo a sus fines.

El sistema es un marco de trabajo que permite la POA sobre Java en el entorno del servidor de aplicaciones de JBoss. No se ha implementado ninguna extensión al lenguaje, en su lugar se utilizan librerías y, para la definición de los puntos de corte, ficheros XML o anotaciones de Java. El conjunto de puntos de enlace que soporta es muy rico, similar al de AspectJ, lo que dota al sistema de una gran potencia.

Los puntos de corte se pueden especificar por medio de ficheros XML externos a los aspectos, o mediante anotaciones Java. La forma de expresarlos es muy similar a la de AspectJ.

Existen dos tipos de aspectos distintos en JBoss AOP. Por un lado están los interceptores –*interceptors*–, que definen un único *advice*, y por otro están los aspectos –*aspects*– que pueden tener un número arbitrario de *advice*. Los interceptores deben heredar obligatoriamente de la interfaz `Interceptor`, los aspectos no tienen ninguna restricción al respecto.

El único tipo de *advice* que soporta JBoss AOP es el *around*. Esto se debe a la forma en que se ha implementado el tejido, basado en envoltorios –*wrappers*–. El proceso de tejido se realiza basándose en los programas ya compilados a *bytecode* en tiempo de carga. Las clases se transforman mediante el empleo de Javassist [Javassist] antes de ser pasadas al cargador de clases. Durante este proceso, cada clase que se carga se comprueba con los puntos de corte definidos y, en caso de que se verifique algún punto de enlace, se instrumenta su código añadiendo las partes necesarias para poder realizar las llamadas a los *advice* que se vean afectados. Este código, ya modificado, es el que se pasa al cargador de clases dejando intacta la clase original.

Es posible activar y desactivar *advice* en tiempo de ejecución, pero para ello es necesario que las clases base a las que afectan hayan sido modificadas previamente.

El sistema presenta un conjunto de puntos de enlace muy rico, y una gran expresividad para definir los puntos de corte. Al no hacer ninguna extensión al lenguaje tanto las clases base como las de los aspectos son independientes de esta plataforma y pueden ser usadas directamente en otros sistemas. La posibilidad de definir los puntos de corte mediante un fichero XML externo (además de la posibilidad de definirlos mediante anotaciones) se traduce en una ventaja a la hora de poder reutilizar los aspectos en diferentes aplicaciones sin tener que modificarlos (simplemente es necesario adaptar los ficheros XML correspondientes). El sistema proporciona además una serie de aspectos estándar ya implementados (caché, comunicación asíncrona, seguridad, etc.) que pueden

ser utilizados directamente por el usuario simplemente definiendo los puntos de corte que apliquen a su sistema.

Al trabajar directamente sobre *bytecode,* y no imponer ninguna condición al código base, no es necesario disponer del código fuente de las aplicaciones, pero, al estar basado en Java, sólo aplicaciones implementadas en Java podrán ser utilizadas. Otra limitación que presenta es la imposibilidad de utilizar otros tipos de *advice* distintos del *around.*

Es posible activar (y desactivar) aspectos en tiempo de ejecución siempre y cuando la clase base a la que modifican haya sido previamente manipulada en el momento de la carga. Esto limita el uso a puntos de enlace ya establecidos en el momento inicial, no pudiendo agregarse otros nuevos (no previstos anteriormente) una vez que la aplicación está ejecutándose.

5.3.2 Spring AOP

Spring AOP [SpringAOP] es un módulo del Framework J2EE Spring [Spring][Johnson04][Johnson05], que utiliza clases Java basadas en JavaBeans [Sun96]. Al ser un Framework las entidades típicas del DSOA vienen como implementaciones de interfaces definidas en el mismo.

Los aspectos, que reciben el nombre de *advisors*, son clases que están compuestas a su vez por instancias de otras clases que representan a los *pointcut* y a los *advice*. La forma de representar estas relaciones es mediante un fichero XML externo donde se especifican las reglas para proceder al enlace.

Spring AOP no hace ninguna extensión al lenguaje, por lo que todos los elementos creados son clases Java normales, en concreto JavaBeans. Esto facilita su reutilización en cualquier otro sistema.

El conjunto de puntos de enlace que soporta se reduce a la invocación a métodos. Esto se debe a que Spring AOP se basa en el uso de los *dynamic proxies* [Sun99] para implementar las invocaciones a los *advice* que se encuentran en los aspectos. Sería posible implementar el acceso a campos como otro punto de acceso, pero los autores argumentan que va en contra de la Orientación a Objetos al romper el encapsulamiento y, además, iría contra el planteamiento del Framework de trabajar únicamente a través de interfaces. No hay restricciones respecto a los tipos de *advice* contemplados en el sistema (*after, around, before*).

Para realizar la adaptación de las aplicaciones no es necesario modificar las clases ni los métodos. Lo que hace el sistema es procesar el fichero XML de configuración y, a partir de él, crea los Proxy adecuados para los puntos de enlace (invocaciones a métodos) solicitados. Cuando se alcanza en la ejecución de la aplicación uno de estos puntos de enlace el Proxy redirige la llamada al *advice*, con lo que se produce la adaptación. Todo este proceso se realiza de forma dinámica, durante la ejecución del sistema.

Este sistema es muy sencillo de implementar y el hecho de no realizar extensiones al lenguaje, ni imponer más restricciones que la implementación de ciertas interfaces, hace que el código generado para él pueda ser reutilizado fácilmente en otros siste-

mas. Como inconvenientes, cabe citar que al estar basado en J2EE se limita su uso al lenguaje Java, y que el conjunto de puntos de enlace que soporta se limita únicamente a las invocaciones a métodos, lo cual lo hace muy pobre en comparación con otros sistemas.

5.3.3 Aportaciones y Carencias de los Sistemas Estudiados

Los dos sistemas vistos en esta sección ofrecen características dinámicas. JBoss AOP permite activar y desactivar aspectos en ejecución, pero únicamente permite activar aspectos en aquellas clases que en el momento de comenzar la ejecución se sabía que iban a ser adaptadas, es decir, si una clase no tiene aspectos en el momento de realizar la carga de la clase posteriormente no los podrá tener. Spring AOP no presenta esta limitación, permitiendo un dinamismo completo (requisito 2.1.1).

Ambos sistemas están basados en J2EE, por lo que son completamente dependientes del lenguaje Java, teniendo que implementar los aspectos y las aplicaciones en él. Esto incumple el requisito 2.1.2, independencia del lenguaje.

Tanto JBoss AOP como Spring AOP están basados en Java, funcionando de forma directa sobre su plataforma, por lo que son independientes de la plataforma (requisito 2.2.1).

El conjunto de puntos de enlace que soporta JBoss AOP es muy rico, si bien únicamente soporta la adaptación del tipo *around*, debido a la forma de realizar el tejido basada en *wrappers*. Spring AOP soporta únicamente la invocación a métodos, en parte por su estructura, basada en *proxies* y en parte por decisión de sus desarrolladores que argumentan que otros tipos chocan con el paradigma de la Orientación a Objetos. Esto es una limitación importante en un sistema AOP de propósito general (requisito 2.1.4).

La forma de definir los puntos de corte en ambos sistemas se basa en el empleo de ficheros XML mediante los que se definen las reglas de tejido. Esto permite la separación completa de los puntos de corte respecto al código de los aspectos (requisito 2.4.3.2), lo que redunda en una mejor reutilización de código.

5.4 Sistemas Basados en Componentes

A continuación vamos a estudiar sistemas que ofrecen soporte a la POA junto al software basado en componentes.

5.4.1 Jasco y Jasco.Net

JAsCo (*Java Aspect Components*) [Suvee03] es un sistema que trata de unir el diseño basado en componentes [Brown98] y la programación orientada a aspectos.

Las principales características de JAsCo son la alta reutilización que se puede hacer de los aspectos, las posibilidades de composición que se pueden realizar entre aspectos, y el alto grado de dinamismo, que permite desplegar y eliminar aspectos en tiempo de ejecución, lo cual es una cualidad necesaria en un sistema que está diseñado buscando dar soporte en el campo de los Servicios Web. JAsCo es una extensión al len-

guaje Java que introduce tres importantes entidades adicionales al lenguaje: *aspect bean, hook y connector*.

Un *aspect bean*, que es una extensión de un Java Bean estándar, está concebido para ser reutilizable en distintos contextos de ejecución, por lo que no presenta información alguna sobre la aplicación base. Los *aspect beans* están compuestos de uno o más *hooks*, los cuales definen la forma de enlazarse (al estilo de los puntos de corte de AspectJ) y el comportamiento frente a los distintos tipos de *advice*. Cada *hook* consta de dos partes:

- Cuándo se activa (*pointcut*). Se especifica la signatura de los métodos que provocarán su activación. Los nombres de estos métodos son abstractos, están definidos en los conectores y no se corresponden con los nombres reales de la aplicación, pero sí tienen la misma signatura. Esto es así con el fin de independizar los aspectos de la aplicación base.
- Qué hará (*advice*). Especifica el comportamiento, el código, para los distintos tipos de *advice* (*before, after o around*).

El lenguaje de definición de aspectos permite no sólo la combinación de los aspectos con el código base sino que permite también la combinación de los aspectos entre sí, al ser entidades del lenguaje, y se pueden relacionar mediante herencia, agregación, etc.

Un conector –*connector*– define la unión entre un *hook* y uno o varios puntos del código base (los puntos de enlace). También establece las relaciones que existen entre varios *hooks* asociados al mismo punto (precedencias, combinaciones, etc.). Se hace uso de la reflexión para conocer el contexto en el que se ubica el conector, con el fin de conocer los *hooks* asociados, comprobar si un conector está o no activado, decidir que *hooks* se deben aplicar, etc.

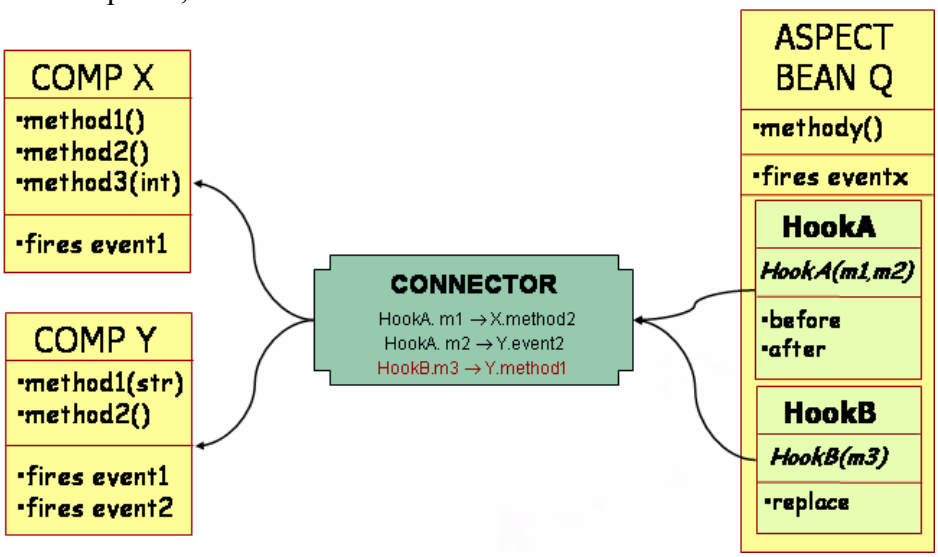

Figura 13: Modelo de componentes de JAsCo [JAsCo06]

Los puntos de enlace que soporta el sistema son:

- Invocación a un método (sólo métodos públicos).

- Ejecución de un método (sólo métodos públicos).
- Activación de un evento de Java Bean.
- Acceso de lectura/escritura a propiedades de los Java Bean.

En las primeras versiones del sistema la forma escogida por JAsCo para realizar el tejido de los aspectos consistía en un preprocesado de los ficheros ya compilados (en *bytecode*), es decir instrumentación de código. Este procesado previo a la ejecución adaptaba todos los componentes de la aplicación base, mediante la inserción de llamadas a funciones externas (*traps*), de tal forma que las ejecuciones de sus métodos (sólo los públicos) se desviase al registro de conectores de JAsCo, que es el que se encarga de verificar si hay *hooks* que deben ejecutarse, su orden, etc. Se puede ver en la Figura 14 el modelo de ejecución.

En versiones posteriores, y debido a la penalización en el rendimiento que supone el adaptar absolutamente todos los componentes con llamadas externas, se añadió una nueva forma de realizar el tejido. En este caso la inserción de las llamadas a funciones externas (los *traps)* se realiza en tiempo de ejecución bajo demanda. Para ello utilizan el HotSwap de Java y la máquina virtual debe estar en modo de depuración.

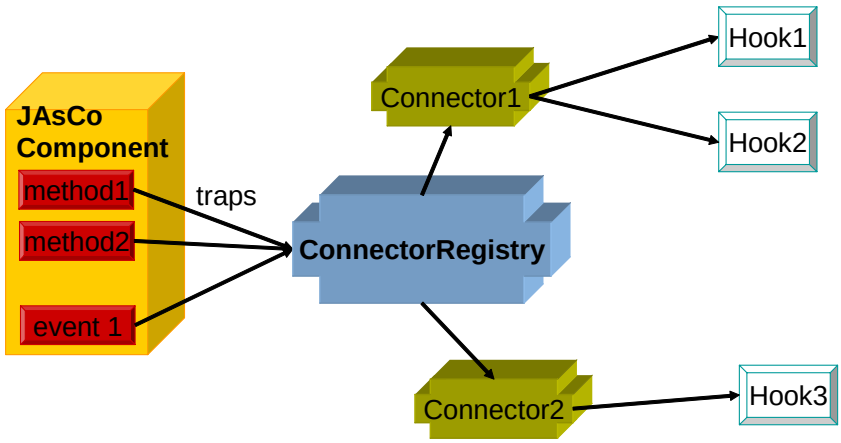

Figura 14: Modelo de ejecución de JAsCo [JAsCo06]

Pese a haber una mejora en el rendimiento al no insertar llamadas en todos los componentes, el tener que ejecutar la máquina virtual en modo depuración sigue penalizándolo. Por ello han desarrollado una nueva forma de realizar el tejido de aspectos [Vanderperren04][DeFraine05]. Es lo que los autores llaman verdadero tejido en tiempo de ejecución. En vez de insertar las llamadas a funciones externas en ejecución, lo que hace el sistema en el momento de la instrumentación de código, es insertar el código de los *hooks* (en *bytecode)* en los lugares necesarios, de tal manera que no se produce una sobrecarga del tiempo de ejecución con llamadas al registro de conectores. El código se teje y "desteje" bajo demanda, en tiempo de ejecución.

Para poder realizar esta instrumentación de código los autores hacen uso del Java Programming Language Instrumentation Services (JPLIS). Esta librería permite realizar instrumentación de código en tiempo de ejecución, sustituyendo unos *bytecodes* por otros, con lo que, cuando es necesario, se sustituye el código original de un método por el código del método tejido junto a los *hooks* que lo afecten. De igual forma, cuando un método ya no se ve afectado por ningún aspecto, se vuelve a cambiar su código para

dejarlo en su estado original. Debido a que la máquina virtual no permite ningún cambio en el esquema en ejecución, JAsCo introduce en el momento de la carga un método vacío por cada método definido en el componente, para después poder añadirle código.

En la actualidad (Diciembre 2006), las tres formas de tejido son compatibles y pueden usarse simultáneamente.

El sistema presenta un conjunto de puntos de enlace no muy amplio, pero suficiente para la mayoría de las aplicaciones. Al ser una extensión del lenguaje Java se limita su uso a sistemas desarrollados en este lenguaje. Pese a que uno de los objetivos principales del sistema es la reutilización de los aspectos, ésta sólo se consigue dentro del sistema, puesto que los aspectos se definen mediante el uso de extensiones del lenguaje Java, lo que los inhabilita para ser usados en otros sistemas que no sean JAsCo.

Aunque las nuevas formas de tejido utilizadas por el sistema consiguen unos mejores rendimientos en ejecución que la original, el hecho de tener que utilizar la máquina virtual en modo de depuración (en un caso), o el de realizar una instrumentación de código complicada durante la ejecución (en el segundo caso) siguen suponiendo una penalización en el rendimiento.

JAsCo.Net [Verspecht03] es un intento de realizar una implementación de JAsCo para la plataforma .NET. Actualmente existe una implementación Beta sobre la que no hay documentación y que es un subconjunto de la implementación JAscO original.

JAsCo.NET hace uso de extensiones al lenguaje C# para definir los aspectos, aunque se podrían añadir extensiones a otros lenguajes compatibles con .NET lo que permitiría implementar los aspectos en estos lenguajes.

La arquitectura de JAsCo.NET es la misma que la arquitectura que presentaba JAsCo en sus primeras versiones. El tejido se realiza en una etapa anterior a la ejecución de la aplicación, y consiste en la instrumentación de código para insertar las llamadas a funciones externas (*traps)* en todos los métodos, con el fin de derivar la ejecución hacia el registro de conectores. Para realizar la instrumentación de código hacen uso de la herramienta Absil [Absil].

Para procesar los aspectos y los conectores se hace uso del *parser* genérico Gold [Gold]. Mediante este *parser* se construyen los compiladores de aspectos y de conectores que sirven para compilar y generar el código correspondiente a cada uno de ellos.

Una ventaja que presenta JAsCo.NET sobre JAsCo es que, al trabajar sobre la plataforma .NET, en código intermedio, se puede utilizar cualquier lenguaje para implementar la aplicación base, ya que sólo se trabaja sobre este código compilado al lenguaje intermedio IL. También se podrían utilizar extensiones a cualquier lenguaje soportado por .NET para definir los aspectos, con su compilador correspondiente que soporte las extensiones añadidas (actualmente sólo se ha realizado para C#).

Los mismos inconvenientes de penalización en el rendimiento están presentes en este sistema, al tener que añadir código en todos los métodos posibles. Además la implementación del sistema no es completa, con limitaciones tanto en la definición de los aspectos como en los conectores [CCosta05]

5.4.2 CAM/DAOP

Este sistema fue presentado inicialmente en [Pinto01] y [Pinto02] (en un principio bajo el nombre de DAOF). Las siglas del sistema se corresponden con *Component–Aspect Model/Dynamic Aspect–Oriented Platform*, que indican que el sistema es un modelo basado en componentes y aspectos, y una plataforma dinámica que lo soporta.

El sistema consta de tres componentes principales:

- CAM: un modelo, especificado mediante UML, que sirve para diseñar aplicaciones basadas en componentes y aspectos.
- DAOP–ADL: un lenguaje, basado en XML, que define la arquitectura del sistema, especificando los componentes, los aspectos y las reglas de composición entre ellos.
- DAOP: la plataforma dinámica que implementa el modelo y realiza la composición de forma completamente dinámica. Esta plataforma es la que ofrece los servicios necesarios como la creación de una instancia de un componente, el envío de mensajes, etc.

El conjunto de puntos de enlace que soporta el sistema está basado en el Desarrollo de Software Basado en Componentes (*Component Based software Development*) [Brown98] [Szyperski02], entendiendo los componentes como "cajas negras" que se intercambian mensajes y lanzan eventos. No se permite el acceso al comportamiento interno de los componentes, pues esto se consideraría una violación del encapsulamiento en el software basado en componentes. Por lo tanto, los puntos de enlace que soporta son la creación y destrucción de instancias de componentes, el envío y la recepción de mensajes y el lanzamiento de excepciones y eventos. El sistema permite utilizar los tiempos *before y after* para la creación y destrucción de instancias y para el envío y recepción de mensajes, el tiempo *after* para el lanzamiento de excepciones, y el tiempo *around* para el lanzamiento de eventos.

Un aspecto es una clase Java normal, con dos restricciones: 1) tiene que definir una serie de variables que se utilizarán para mantener referencias a la plataforma DAOP y 2) que el constructor tiene que tener dos parámetros mediante los que recibe los valores para esas variables. Un aspecto debe implementar uno o varios *advice*.

Los *advices* son métodos dentro de un aspecto que tienen que tener una signatura de tipo `eval<JoinPoint>` siendo *JoinPoint* el nombre del punto de enlace específico (por ejemplo `evalBEFORE_NEW` se refiere a "antes de" la creación de una instancia de un componente).

La forma de definir la composición entre los componentes y los aspectos es mediante un fichero en DAOP–ADL que define la arquitectura de la aplicación y la relación entre los componentes y los aspectos, es decir, los puntos de corte. Este fichero es procesado por la plataforma DAOP, y en base a su contenido decide qué puntos de enlace han sido seleccionados.

El funcionamiento del sistema se basa en el uso que hacen de la plataforma los componentes. Cuando un componente necesita crear o eliminar una instancia de otro componente o enviar un mensaje o un evento se lo solicita a la plataforma, que es la que

ofrece los servicios adecuados para ello. Estas acciones son los puntos de enlace de la aplicación. En el momento que la plataforma recibe la solicitud comprueba si se ha definido algún aspecto para el punto de enlace en cuestión (mediante los puntos de corte procesados) y en el caso de ser así, ejecuta el *advice* correspondiente del aspecto.

Si durante la ejecución de la aplicación es necesario cambiar puntos de corte de algún aspecto que ya existe, basta con modificar el fichero de definición y añadir las modificaciones correspondientes al mismo, que serán procesadas de forma automática por la plataforma DAOP. Si es necesario añadir algún nuevo aspecto no contemplado previamente se añadirá su especificación a la descripción de la arquitectura de la aplicación, indicando las interfaces que implementa, sus clases, propiedades, etc. Esta información será procesada por la plataforma DAOP que la tendrá en cuenta de forma inmediata.

La implementación actual del sistema se ha realizado sobre Java por lo que sólo aplicaciones escritas en Java podrán beneficiarse del mismo, pero el modelo CAM que han definido los autores puede implementarse sobre cualquier plataforma [Fuentes03], eliminando esta limitación.

Este sistema es apropiado para aplicaciones distribuidas basadas en componentes, el sistema permite realizar el diseño de una aplicación mediante CAM, traducirlo a DAOP–ADL y ejecutarlo sobre DAOP. El hecho de tener totalmente separadas la implementación de los aspectos de los puntos de corte es una ventaja a la hora de la posible reutilización del código ya que los aspectos no tienen que tener información del contexto. El proceso de adaptación se realiza de forma realmente dinámica (no en tiempo de carga), ni necesita ningún tipo de instrumentación de los componentes antes de realizar el tejido.

El sistema no es apto para aplicaciones que no se presten al desarrollo de software basado en componentes, o aquellas aplicaciones que ya estén implementadas, ya que para poder funcionar en el sistema tienen que hacer uso de los servicios que ofrece la plataforma DAOP, tales como la creación de componentes o el envío de mensajes.

El conjunto de puntos de enlace que soporta el sistema es no invasivo, por decisión de los autores, impidiendo por tanto el acceso a la parte privada de la implementación.

5.4.3 Aportaciones y Carencias de los Sistemas Estudiados

Los dos sistemas vistos en esta sección ofrecen características dinámicas. Tanto JAsCo como CAM/DAOP permiten activar y desactivar aspectos en tiempo de ejecución sin que sea necesario tener un conocimiento previo de ellos (requisito 2.1.1).

JAsCo está basado en Java, por lo que es dependiente del lenguaje. Además utiliza un superconjunto de Java con lo que esta dependencia se ve acentuada. La versión basada en .NET en principio podría ser independiente del lenguaje, pero al realizar extensiones a aquellos lenguajes que se vayan a utilizar se crea una dependencia. La definición de CAM/DAOP es independiente del lenguaje. La implementación actual, realizada sobre la plataforma Java, no lo es por las dependencias que impone la plataforma

pero implementaciones en otras plataformas, como podría ser .NET, permitirían la independencia del lenguaje (requisito 2.1.2).

JAsCo está basado en la máquina virtual de Java, y JAsCo.NET en la máquina virtual de .NET, siendo ambas independientes de la plataforma, por lo que ambos sistemas también lo son (requisito 2.2.1). CAM/DAOP no está basado en un entorno concreto, pudiendo ser implementado en diversos entornos. La implementación existente se ha realizado sobre Java, que es independiente de la plataforma por lo que la implementación también lo es.

El conjunto de puntos de enlace que soportan ambos sistemas viene limitado por el hecho de ser sistemas que soportan el desarrollo de software basado en componentes. Los autores de ambos sistemas argumentan que un componente debe comportarse como una caja negra que ofrece una serie de interfaces y requiere otras, pero no se debe acceder a su parte privada. Es decir no permiten un modelo de puntos de enlace invasivo. Por esto el conjunto de puntos de enlace soportado no cumple el requisito 2.1.4.

JAsCo define los puntos de corte del sistema mediante extensiones realizadas al lenguaje Java. El código del aspecto va unido al de los puntos de corte, creando un acoplamiento y dificultando su posible reutilización dentro del sistema. CAM/DAOP expresa los puntos de corte mediante un fichero XML externo al código de los aspectos por lo que no se crean dependencias en el código y se facilita la posible reutilización de los aspectos dentro del sistema.

5.5 Otros Sistemas

Existen otros muchos sistemas que ofrecen en mayor o menor grado soporte a la POA. El grado de desarrollo de estos sistemas varía mucho, y va desde sistemas que tienen una implementación desarrollada a sistemas que sólo cuentan con una propuesta teórica (muchas veces poco o nada documentada). A continuación vamos a mencionar sólo algunos de estos sistemas.

SetPoint [SetPoint][Altman04][Altman05] es un sistema novedoso que se basa en el concepto de Web Semántica [W3CSW] para crear el concepto del punto de corte semántico –*Semantic Pointcut*–. El sistema se está implementando sobre la plataforma .NET. El funcionamiento se basa en etiquetar el código base de tal forma que los aspectos se enlacen con la aplicación mediante predicados lógicos. Estas marcas reciben el nombre de Descriptores Semánticos (SetPoint) y cumplen el mismo rol que los puntos de enlace (en la plataforma .NET estos descriptores se corresponden con los atributos personalizados, *custom attributes*). Estos descriptores identifican la semántica del código al que etiquetan (por ejemplo identifican que un método es un *setter* o un *getter*, etc.)

Los puntos de corte pasan a ser predicados lógicos que trabajan sobre esos descriptores. En vez de identificar puntos de enlace específicos a través de su nombre (aproximación general de la mayoría de los sistemas existentes) lo que hace es referirse a los puntos etiquetados de una forma determinada (por ejemplo aquellos métodos etiquetados como *setters*).

Este proyecto se encuentra en una fase muy temprana, no existe mucha documentación al respecto y no es posible probar un prototipo, por lo que es difícil de anali-

zar. Una limitación que se observa en el sistema es la necesidad de etiquetar el código base con los *SetPoint.*

Otro sistema basado en la plataforma .NET es AOP.NET [Schmied02] que fue desarrollado por el departamento de SE de Siemens [SiemensSE]. Hace uso de la API de *profiling* no gestionada, y de forma reflectiva modifican las tablas de metadatos sonde se encuentran las definiciones de métodos para crear proxys que intercepten el código y lo redirijan a los aspectos que lo hayan solicitado. El conjunto de puntos de enlace se limita a la invocación a métodos, y sólo contempla los tipos *before y after*, no dando la opción de utilizar el tipo *around.* El hacer uso de la API de *profiling* presenta una serie de inconvenientes como son el rendimiento y la pérdida de portabilidad tal y como vimos en el sistema CLAW (5.2.3).

EOS [Rajan03] es una extensión al lenguaje C#, del estilo de AspectJ, cuya principal innovación consiste en el concepto de aspecto a nivel de instancia. Los aspectos se definen y enlazan a clases en el momento de la compilación, pero en el momento de la ejecución se puede decidir a qué instancias se aplica el aspecto. Para poder utilizar este sistema es necesario utilizar un compilador propio, que actualmente se encuentra en fase beta, lo que es una limitación. Otra limitación es el hecho de que sólo funcione con C#, y al haberle realizado extensiones, el código implementado para este sistema es difícilmente reutilizable en otros sistemas (que no serán capaces de procesar las extensiones añadidas).

5.6 Conclusiones

El Desarrollo de Software Orientado a Aspectos es un paradigma joven aún y existen pocos sistemas que ofrecen soporte a la Programación Orientada a Aspectos que puedan considerarse maduros, siendo la mayoría propuestas de investigación.

AspectJ es el sistema más extendido de los que ofrecen soporte a la POA. Su influencia se nota en que se ha establecido como patrón de comparación de los sistemas que la ofrecen y, además, los conceptos y terminología en él empleados se encuentran presentes en la gran mayoría de sistemas desarrollados.

La mayoría de sistemas existentes se han implementado para dar soporte a programas realizados en lenguaje Java, lo que crea una dependencia del lenguaje imposibilitando la utilización del sistema por parte de programas implementados en otros lenguajes. En los últimos tiempos han surgido diversos proyectos que pretenden ofrecer una independencia del lenguaje (requisito 2.1.2), basándose la mayoría de ellos en la plataforma .NET.

La mayoría de sistemas se implementan sobre máquinas virtuales (Java y .NET principalmente) lo que les posibilita trabajar sobre el código intermedio propio de la máquina virtual en vez de sobre el código fuente original, esto posibilita que estos sistemas no necesiten disponer del código fuente de las aplicaciones para poder adaptarlas (requisito 2.3.1). Aún así existen algunos sistemas que imponen restricciones o condiciones a las aplicaciones a ser adaptadas limitando su posible utilización.

Inicialmente los sistemas desarrollados ofrecían un soporte estático a la POA, es decir, el tejido se realizaba en el momento del desarrollo del sistema, quedando fijado y

no pudiendo alterarse durante la ejecución. Aunque esto fue un gran avance respecto a las técnicas anteriores, se detectó que en ocasiones es necesario disponer de la posibilidad de realizar el tejido en tiempo de ejecución para dar respuesta a cambios en el entorno de ejecución o a la aparición de nuevos requerimientos no previstos durante el desarrollo y de esta necesidad surgen los sistemas con soporte dinámico a la POA (requisito 2.1.1). Existen sistemas con diverso grado de dinamismo, unos permiten activar o desactivar únicamente aspectos que hubiesen sido definidos en la etapa de desarrollo, mientras que otros permiten realizarlo sobre aspectos que no se conocían en ese momento (éstos son los sistemas realmente dinámicos).

En general los sistemas dinámicos presentan muchas restricciones respecto a los estáticos limitando el número de puntos de enlace disponibles o la forma de realizar las adaptaciones.

Hasta donde conoce el autor, en la actualidad no existe ningún sistema que cumpla todos los requisitos fijados en el capítulo 2. Los sistemas más extendidos y maduros son estáticos con lo que se invalida uno de los principales requisitos fijados (2.1.1). La mayoría de los sistemas existentes, tanto estáticos como dinámicos, están basados en el lenguaje Java, introduciendo una dependencia del lenguaje (requisito 2.1.2).

Muchos de los sistemas que argumentan ser dinámicos lo son únicamente en parte, necesitando conocer durante la fase de desarrollo los aspectos que van a adaptar a una aplicación durante su ejecución, por lo que no presentan un verdadero dinamismo que les permita adaptarse a cambios en el entorno o nuevos requisitos no previstos.

Los sistemas que cumplen estas condiciones (realmente dinámicos e independientes del lenguaje) presentan diversas restricciones como pueden ser el conjunto de puntos de enlace limitado o la imposición de condiciones a las aplicaciones a adaptar, lo que implica en muchas ocasiones tener que disponer de su código fuente para poder modificarlas con el fin de satisfacer dichas condiciones.

Otros sistemas utilizan extensiones a lenguajes (como puede ser Java) para definir los aspectos, lo que imposibilita la reutilización del código en otros sistemas. Además muchos sistemas entremezclan el código de los aspectos con el de los puntos de corte, creando así dependencias entre ellos y dificultando la reutilización de código ya sea en el mismo sistema o en otro sistema.

CAPÍTULO 6
MÁQUINAS ABSTRACTAS

En este capítulo se definen los conceptos de máquina abstracta y máquina virtual que estarán presentes a lo largo de toda la memoria. Analizaremos la evolución histórica que han tenido así como el objetivo principal buscado en cada una de las etapas. Seguidamente veremos los distintos usos que se le ha dado al concepto de máquina abstracta. A continuación se mostrarán las aportaciones que el uso de máquinas abstractas tiene para nuestro sistema, en relación a los requisitos marcados.

Para finalizar el capítulo, se analizarán una serie de sistemas existentes que implementan máquinas abstractas que pueden servir para cumplir los objetivos fijados, destacando sus logros y carencias, en función de los objetivos. Una vez descritos estos sistemas identificaremos qué requisitos han sido cumplidos por los casos prácticos existentes y cómo adaptarlos a nuestros objetivos, así como sus insuficiencias y las necesidades surgidas para superarlas.

6.1 Procesadores Computacionales

Un programa es un conjunto ordenado de instrucciones que se dan al ordenador indicándole las operaciones o tareas que se desea que realice [Cueva94]. Estos programas van a ser ejecutados, animados o interpretados por un procesador computacional.

Un procesador computacional ejecuta las instrucciones propias de un programa que accederán, examinando o modificando, a los datos pertenecientes a dicho programa. La implementación de un procesador computacional puede ser física (*hardware*) o lógica (*software*) mediante el desarrollo de otro programa.

6.1.1 Procesadores Implementados Físicamente

Un procesador físico es un intérprete de programas que ha sido desarrollado de forma física (comúnmente como un circuito integrado). Los procesadores más extendidos son los procesadores digitales síncronos. Están formados por una unidad de control, una memoria y una unidad aritmético lógica, todas ellas interconectadas entre sí [Mandado73].

Un computador es un procesador digital síncrono cuya unidad de control es un sistema secuencial síncrono que recibe desde el exterior (el programador) una secuencia

de instrucciones que le indican las micro operaciones que debe realizar [Mandado73]. La secuencia de ejecución de estas operaciones es definida en la memoria describiendo un programa, pero la semántica de cada instrucción y el conjunto global existente para el procesador es invariable.

La ventaja de este tipo de procesadores frente a los lógicos es su velocidad al haber sido desarrollados físicamente. Su principal inconveniente, como decíamos en el párrafo anterior, es su inflexibilidad.

6.1.2 Procesadores Implementados Lógicamente

Un procesador software es un programa que interpreta a su vez programas de otro procesador [Cueva98]. Es aquel programa capaz de interpretar o emular el funcionamiento de un determinado procesador.

La modificación de un programa que emule a un procesador es mucho más sencilla que la modificación física de un procesador *hardware*. Esto hace que los emuladores software se utilicen, entre otras cosas, como herramientas de simulación encaminadas a la implementación física del procesador que emulan.

La principal desventaja de este tipo de procesadores frente a los procesadores *hardware* o físicos es la velocidad de ejecución. Puesto que los procesadores lógicos establecen un nivel más de computación (son ejecutados o interpretados por otro procesador), es inevitable que requieran un mayor número de computaciones para interpretar un programa que su versión hardware. Esta sobrecarga computacional se aprecia gráficamente en la Figura 15:

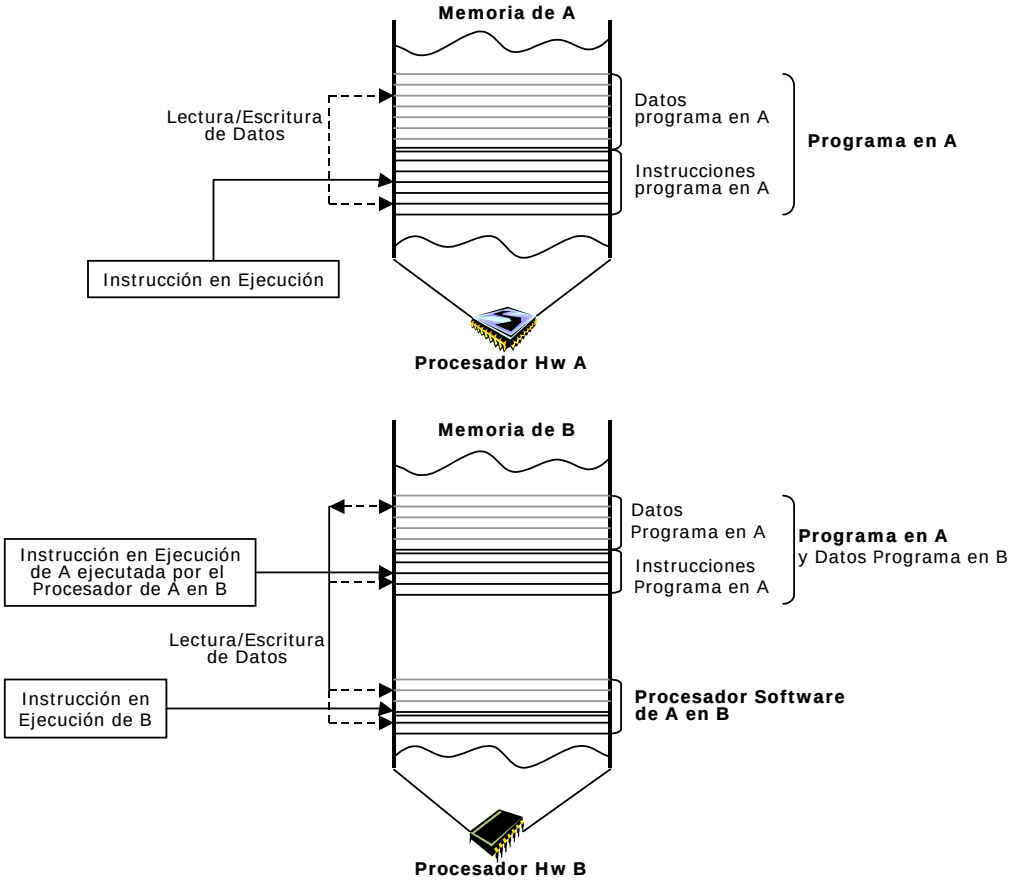

Figura 15: Ejecución de un procesador lógico frente a un procesador físico.

En la parte superior de la figura se muestra cómo el procesador físico **A** va interpretando las distintas instrucciones máquina. La interpretación de las instrucciones implica la lectura y/o escritura de los datos.

En el caso de interpretar a nivel *software* el programa, el procesador es a su vez un programa en otro procesador físico (procesador **B** en la Figura 15). Vemos cómo existe una nueva capa de computación frente al ejemplo anterior. Esto hace que se requieran más computaciones[7] o cálculos que en el primer caso.

En el segundo caso el procesador **A** es más flexible que en el primero. La modificación, eliminación o adición de una instrucción de dicho procesador, se consigue con la modificación del programa emulador. Sin embargo, el mismo proceso en el primer ejemplo, requiere la modificación del procesador a nivel físico.

6.2 Procesadores Lógicos y Máquinas Abstractas

Una máquina abstracta es el diseño de un procesador computacional sin intención de que éste sea desarrollado de forma física [Howe99]. Apoyándose en dicho diseño del procesador computacional, se especifica formalmente la semántica del juego de

[7] Mayor número de computaciones no implica siempre mayor tiempo de ejecución. Esta propiedad se cumpliría si la velocidad de procesamiento del computador **B** fuese igual a la del computador **A**.

instrucciones de dicha máquina en función de la modificación del estado de la máquina abstracta.

Un procesador computacional desarrollado físicamente es también una máquina abstracta con una determinada implementación [Álvarez98]. Existen multitud de ejemplos de emuladores de microprocesadores físicos desarrollados como programas sobre otro procesador. Sin embargo, el nombre de máquina abstracta es empleado mayoritariamente para aquellos procesadores que no tienen una implementación física.

Un intérprete es un programa que ejecuta las instrucciones de un lenguaje que encuentra en un archivo fuente [Cueva98]. Su objetivo principal es animar la secuencia de operaciones que un programador ha especificado, en función de la descripción semántica de las instrucciones del lenguaje utilizado. Un procesador computacional – implementado física o lógicamente – es un intérprete del lenguaje que procese.

Se define máquina virtual como un intérprete de una máquina abstracta[8] [Howe99]. Una máquina virtual es por tanto un intérprete definido sobre una máquina abstracta. De esta forma, la máquina abstracta es utilizada en la descripción semántica de las instrucciones de dicho intérprete.

6.3 Utilización del Concepto de Máquina Abstracta

En este punto realizaremos un estudio de las distintas aplicaciones prácticas encontradas al concepto de máquina abstracta. Especificaremos una clasificación por funcionalidades y una descripción colectiva de lo conseguido con cada utilización. En un punto posterior (6.5 Panorámica de Utilización de Máquinas Abstractas) analizaremos casos particulares de sistemas existentes, destacando sus aportaciones y carencias en función de los requisitos establecidos en el capítulo 3.

6.3.1 Procesadores de Lenguajes

En la implementación de compiladores se ha utilizado el concepto de máquina abstracta para simplificar su diseño [Cueva94]. El proceso de compilación toma un lenguaje de alto nivel y genera un código intermedio. Este código intermedio es propio de una máquina abstracta.

Se diseña una máquina abstracta lo más general posible, de forma que se pueda traducir de ésta a cualquier máquina real existente. Para generar el código binario de una máquina real, tan sólo hay que traducir el código de la máquina abstracta a la máquina física elegida, independientemente del lenguaje de alto nivel del que haya sido compilado previamente.

[8] Aunque el concepto de máquina abstracta y máquina virtual no son exactamente idénticos, son comúnmente intercambiados.

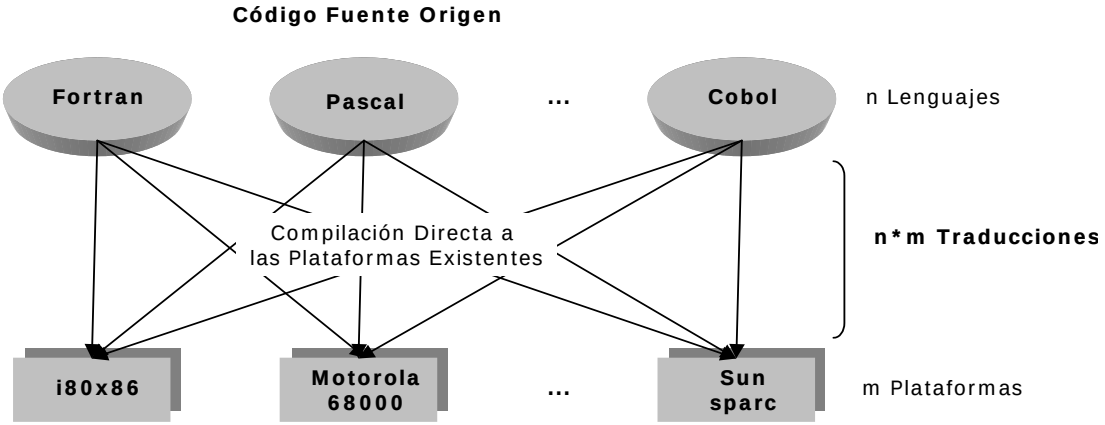

Figura 16: Compilación directa de n lenguajes a m plataformas.

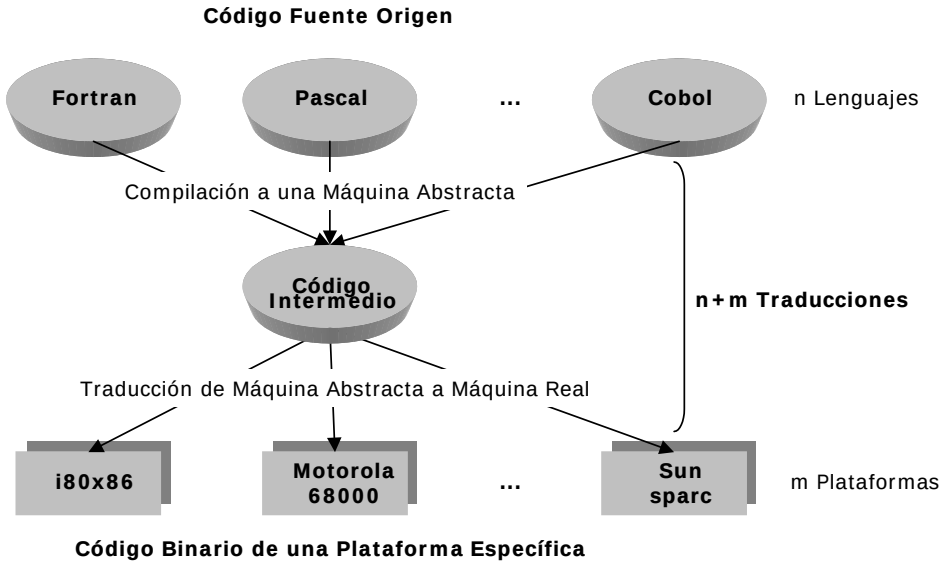

Figura 17: Compilación de lenguajes pasando por la generación de código intermedio.

En la Figura 16 y en la Figura 17 se observa cómo el número de traducciones y compilaciones se reduce cuando tenemos varios lenguajes fuente y varias máquinas destino existentes[9].

El proyecto UNCOL (*Universal Computer Oriented Language*) proponía un lenguaje intermedio universal para el diseño de compiladores [Steel60]. El objetivo de este proyecto era especificar una máquina abstracta universal para que los compiladores generasen código intermedio a una plataforma abierta.

ANDF (*Architecture Neutral Distribution Format)* [Macrakis93] tuvo como objetivo un híbrido entre la simplificación de compiladores y la portabilidad del código (punto 6.3.2). Un compilador podría generar código para la especificación de la máquina ANDF siendo este código portable a distintas plataformas.

[9] En concreto, para n lenguajes y m plataformas, se reduce el número de traducciones para n y m mayores que dos.

Este código ANDF no era interpretado por un procesador software sino que era traducido (o instalado) a código binario de una plataforma específica. De esta forma se conseguía lo propuesto con UNCOL: la distribución de un código de una plataforma independiente.

Esta práctica también ha sido adoptada por varias compañías que desarrollan diversos tipos de compiladores. Un ejemplo de la utilización de esta práctica son los productos de Borland/Inprise [Borland]. Inicialmente esta compañía seleccionó un mismo "*back end*" para todos sus compiladores. Se especifica un formato binario para una máquina compartida por todas sus herramientas (archivos de extensión obj). Módulos de aplicaciones desarrolladas en distintos lenguajes como C++ y Delphi pueden enlazarse para generar una aplicación, siempre que hayan sido compiladas a una misma plataforma [Trados96].

El siguiente paso fue la especificación de una plataforma, o máquina abstracta, independiente del sistema operativo que permita desarrollar aplicaciones en distintos lenguajes y sistemas operativos. Este proyecto bautizado como "Proyecto Kylix" especifica un modelo de componentes CLX (*Component Library Cross–Platform*) que permite desarrollar aplicaciones en C++ Builder y Delphi para los sistemas operativos Win32 y Linux [Kozak2000].

Una aproximación muy similar se ha implementado en la plataforma .NET de *Microsoft* [ECMA335], donde todos los lenguajes de esta plataforma se ejecutan en una máquina virtual común llamada *CLR* (*Common Language Runtime*). Cualquier lenguaje implementado sobre dicha plataforma se compila a un lenguaje intermedio común o CIL (*Common Intermediate Language*), que es ejecutado por la máquina virtual. Gracias a esto poseen, entre otras propiedades, interoperabilidad, de manera que se puede usar cualquier componente desarrollado en un lenguaje desde el código de cualquier otro. Esta capacidad también permite a los lenguajes implementados en este entorno tener acceso a la misma librería de clases base común o BCL (*Base Class Library*), proporcionándoles multitud de funcionalidades (comunicaciones, etc.) comunes a cualquiera de ellos. Esto permite que una vez que se sabe usar cierta librería con uno de los lenguajes, se puede usar de la misma forma desde cualquiera de ellos. Se puede ver un esquema de la plataforma en la Figura 18.

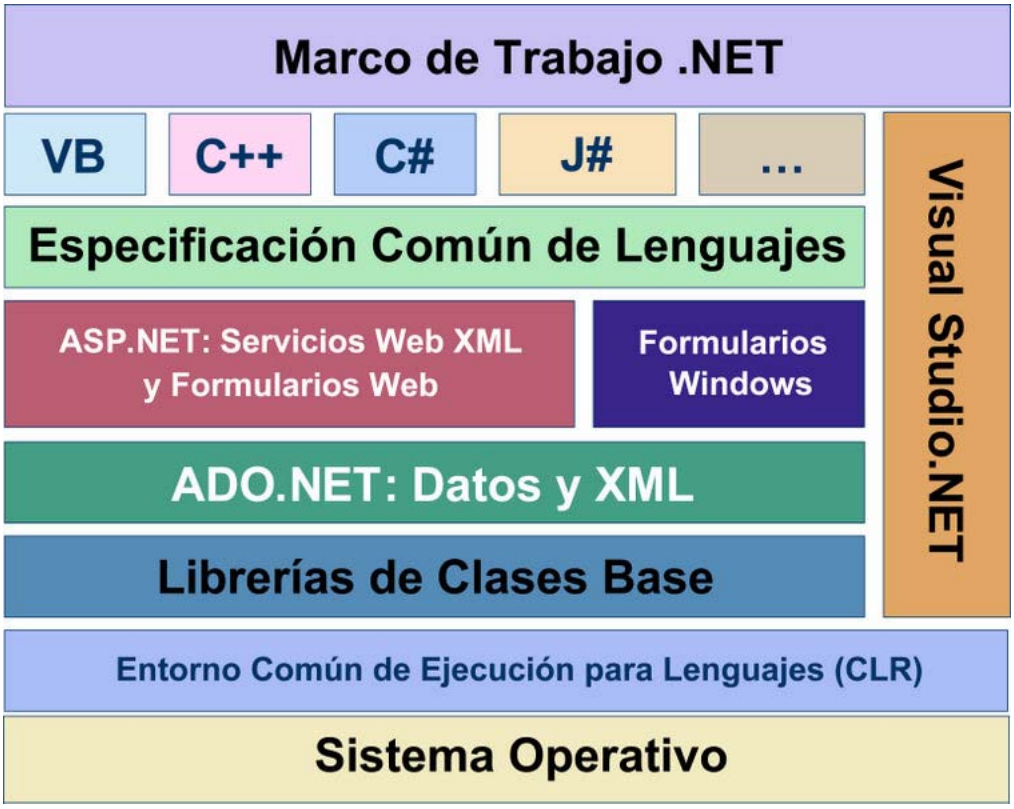

Figura 18: Esquema de la estructura del *CLR*

6.3.1.1 Entornos de Programación Multilenguaje

Apoyándose de forma directa en los conceptos de máquinas abstractas y máquinas virtuales, se han desarrollado entornos integrados de desarrollo de aplicaciones multilenguaje.

POPLOG es un entorno de programación multilenguaje enfocado al desarrollo de aplicaciones de inteligencia artificial [Smith92]. Utiliza compiladores incrementales de Common Lisp, Pop–11, Prolog y Standard ML. La capacidad de interacción entre los lenguajes reside en la traducción a una máquina abstracta de alto nivel (PVM, *Poplog Virtual Machine*) y la independencia de la plataforma física utilizada se obtiene gracias a la compilación a una máquina de bajo nivel (PIM, *Poplog Implementation Machine*) y su posterior conversión a una plataforma física –como veíamos en la Figura 17.

6.3.2 Portabilidad del Código

Aunque todos los procesadores *hardware* son realmente implementaciones físicas de una máquina abstracta, es común utilizar el concepto de máquina abstracta para designar la especificación de una plataforma cuyo objetivo final no es su implementación en silicio.

Probablemente la característica más explotada en la utilización de máquinas abstractas proviene de su ausencia de implementación física. El hecho de que sea un procesador *software* el que interpreta las instrucciones de la máquina, da lugar a una independencia de la plataforma física real utilizada. Una vez codificado un programa para una

máquina abstracta, su ejecución podrá realizarse en cualquier plataforma[10] que posea un procesador lógico capaz de interpretar sus instrucciones, como se muestra en la Figura 19.

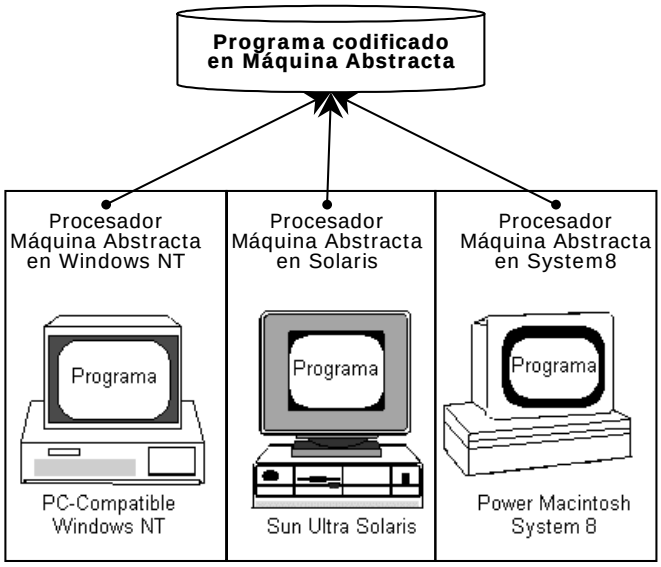

Figura 19: Ejecución de un programa portable sobre varias plataformas.

Al igual que un compilador de un lenguaje de alto nivel genera código para una máquina específica, la compilación a una máquina abstracta hace que ese programa generado sea independiente de la plataforma que lo ejecute. Dicho programa podrá ser ejecutado por cualquier procesador –ya sea hardware o software – que implemente la especificación computacional de un procesador de la máquina abstracta.

Cada procesador computacional de la máquina abstracta podrá estar implementado acorde a las necesidades y características del sistema real. En la Figura 20 se identifican distintos grados de implementación del procesador de la máquina. En el primer ejemplo se implementa la máquina físicamente obteniendo velocidad de ejecución del programa al tener un único nivel de interpretación.

Si se implementa un procesador lógico sobre una plataforma distinta, obtenemos la portabilidad mencionada en este punto perdiendo velocidad de ejecución frente al caso anterior. Cada plataforma física que emule la máquina abstracta, implementará de forma distinta este procesador en función de sus recursos. En el ejemplo mostrado en la Figura 19, el procesador implementado sobre el PC con Windows NT podrá haber sido desarrollado de forma distinta a la implementación sobre el Macintosh con System8.

En la Figura 20 se muestra otro ejemplo en el que el emulador de la máquina abstracta es desarrollado sobre otro procesador lógico. Seguimos teniendo portabilidad del código y obtenemos una flexibilidad en el propio emulador de la máquina abstracta. Se puede ver este caso como un procesador (de **A** sobre **B**) de un procesador de la máquina abstracta (sobre **A**). Volvemos a ganar en flexibilidad a costa de aumentar en número de computaciones necesarias en la ejecución de un programa de la máquina abstracta.

[10] Entendiendo por plataforma la combinación de microprocesador y sistema operativo.

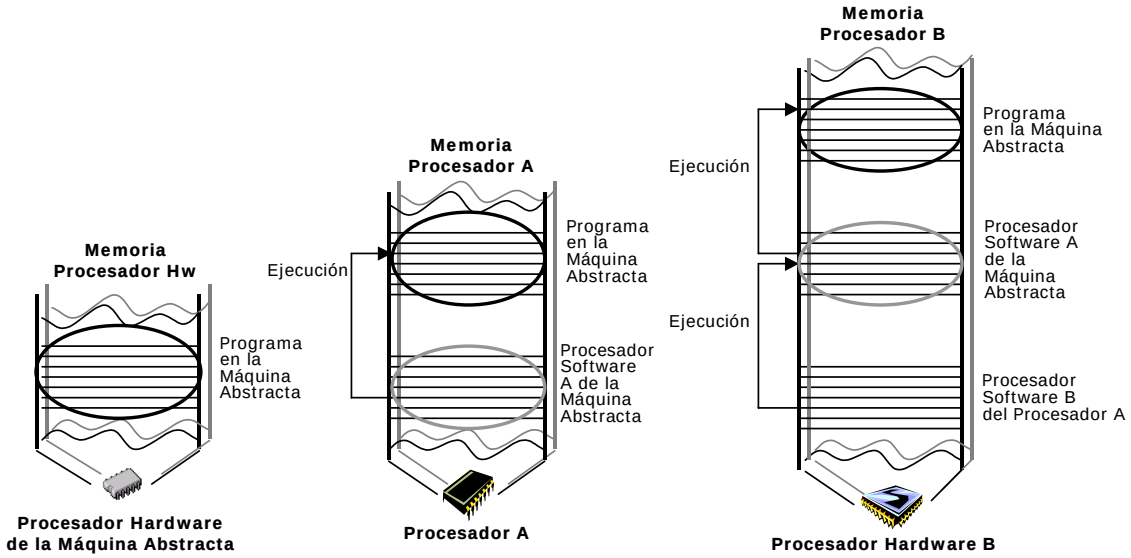

Figura 20: Distintos niveles de implementación de un procesador.

Existen multitud de casos prácticos que utilizan el concepto de máquina abstracta para conseguir programas portables a distintas plataformas. En el punto 6.5 estudiaremos un conjunto de éstos.

6.3.3 Sistemas Interactivos con Abstracciones Orientadas a Objetos

Con la utilización de los lenguajes orientados a objetos en la década de los 80, el nivel de abstracción en la programación de aplicaciones se elevó considerablemente [Booch94]. Sin embargo, los microprocesadores existentes se seguían basando en la ejecución de código no estructurado y los intentos de desarrollarlos con este nuevo paradigma finalizaban en fracaso por la falta de eficiencia obtenida causada por la complejidad de la implementación [Colwel88]. Posteriormente se llevaron a cabo estudios que concluyen que, haciendo uso de optimizaciones de compilación e interpretación se obtienen buenos rendimientos y, por tanto, no es rentable el desarrollo de plataformas físicas orientadas a objetos [Hölzle95].

Para desarrollar entornos de programación y sistemas que aportasen directamente la abstracción de orientación a objetos, y que fuesen totalmente interactivos, se utilizó el concepto de máquina abstracta –orientada a objetos. Los sistemas, al igual los lenguajes de programación, ofrecían la posibilidad de crear y manipular una serie de objetos. Estos objetos residían en la memoria de una máquina abstracta. Las diferencias de trabajar con una máquina abstracta en lugar de compilar a la plataforma nativa son las propias de la dualidad compilador/intérprete:

	Compilador	Intérprete
Tiempo de Ejecución	–	+
Tiempo de Compilación	+	–
Portabilidad de Código	–	+
Interacción entre Aplicaciones	–	+
Flexibilidad	–	+

Como vimos en los dos puntos anteriores, el utilizar un procesador lógico genera un mayor número de computaciones en su interpretación perdiendo tiempo de ejecución y ganando en portabilidad código. Si la plataforma que interpretamos es de un nivel más cercano al lenguaje (es orientada a objetos), los tiempos de compilación se reducirán al no producirse cambio de paradigma.

Como se muestra en Figura 21, el procesamiento lógico de los programas hace que todos ellos compartan una zona de memoria asociada a un proceso –el intérprete. La interacción entre aplicaciones es homogénea, y por tanto más sencilla de implementar que en el caso de que se cree un proceso distinto para la ejecución de cada aplicación.

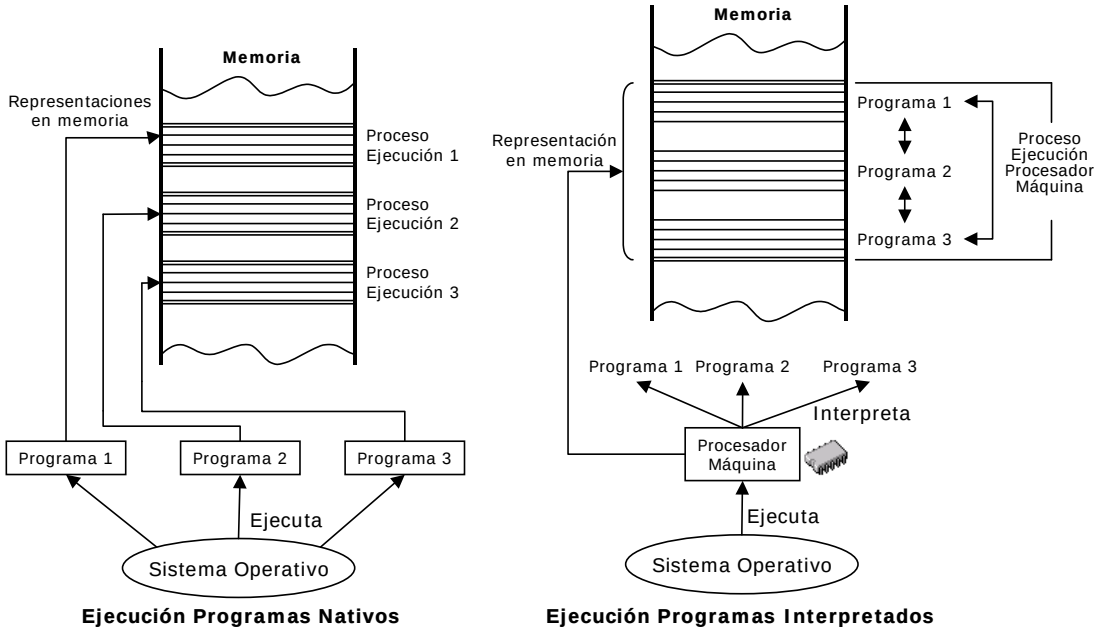

Figura 21: Diferencia entre la ejecución de programas nativos frente interpretados.

El hecho de que las aplicaciones puedan interactuar fácilmente y que se pueda acceder y modificar los distintos objetos existentes, aumenta la flexibilidad global del sistema, pudiendo representar sus funcionalidades mediante objetos y métodos modificables.

Los sistemas desarrollados sobre una máquina abstracta orientada a objetos facilitan al usuario:

- Utilización del sistema con una abstracción más natural a la forma de pensar del ser humano [Booch94].
- Auto documentación del sistema y de las aplicaciones. El acceso a cualquier objeto permite conocer la estructura y comportamiento de éste en cualquier momento.
- Programación interactiva y continua. Una vez que el usuario entra en el sistema, accede a un mundo interactivo de objetos [Smith95]. Puede hacer uso de cualquier objeto existente. Si necesita desarrollar una nueva funcionalidad va creando nuevos objetos, definiendo su estructura y comportamiento, comprobando su correcta funcionalidad y utilizando cualquier otro objeto exis-

tente de una forma totalmente interactiva. A partir de ese momento el sistema posee una nueva funcionalidad y un mayor número de objetos.

Dentro de este tipo de sistemas se pueden nombrar a los clásicos Smalltalk [Mevel87] y Self [Ungar87] que serán tratados con mayor profundidad más adelante.

6.3.4 Distribución e Interoperabilidad de Aplicaciones

Esta ventaja en la utilización de una plataforma abstracta es una ampliación de la característica de portabilidad de código comentada previamente en el punto 6.3.2. El hecho de tener una aplicación codificada sobre una máquina abstracta implica que ésta podrá ejecutarse en cualquier plataforma que implemente esta máquina virtual. Además de esta portabilidad de código, la independencia de la plataforma física puede ofrecer dos ventajas adicionales:

Una aplicación (su código) podrá ser distribuida a lo largo de una red de computadores. Los distintos módulos codificados para la máquina abstracta pueden descargarse y ejecutarse en cualquier plataforma física que implemente la máquina virtual.

Las aplicaciones pueden interoperar entre sí (con envío y recepción de datos) de forma independiente a la plataforma física sobre la que estén ejecutándose. La representación de la información nativa de la máquina abstracta será interpretada por la máquina virtual de cada plataforma.

Aunque estos beneficios en la utilización de una máquina abstracta ya estaban presentes en Smalltalk [Goldberg83] y Self [Smith95], su mayor auge tuvo lugar con la aparición de la plataforma virtual de Java [Kramer96] –que estudiaremos en mayor profundidad en el punto 6.5.2. Esta plataforma impulsó el desarrollo de aplicaciones distribuidas, especialmente a través de Internet. Posteriormente, y siguiendo una filosofía muy similar ha surgido la plataforma .NET, que veremos en 6.5.3.

6.3.4.1 Distribución de aplicaciones

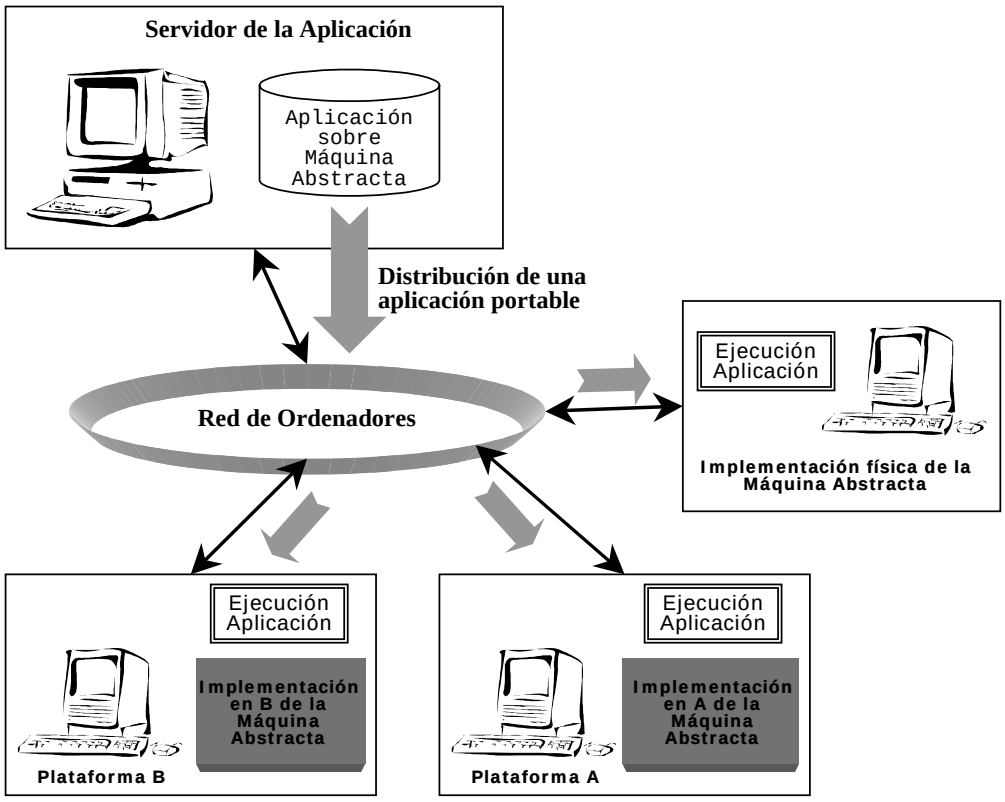

Figura 22: Distribución de aplicaciones portables.

En la Figura 22 se muestra un escenario de distribución de una aplicación desarrollada sobre una máquina abstracta. Un ordenador servidor de la aplicación, conectado mediante una red de ordenadores a un conjunto de clientes, posee el código del programa a distribuir. Mediante un determinado protocolo de comunicaciones, cada cliente demanda la aplicación del servidor, la obtiene en su ubicación y la ejecuta gracias a su implementación de la máquina virtual. Esta ejecución será computacionalmente similar en todos los clientes, con el aspecto propio de la plataforma física en la que es interpretada –como mostrábamos en la Figura 19 de la página 82.

Un caso de uso típico del escenario mostrado es una descarga de *applets* en Internet. El servidor de aplicaciones es un servidor Web. El código de la aplicación es código Java restringido, denominado *applet* [Kramer96]. El cliente, mediante su navegador Web, se conecta al servidor utilizando el protocolo HTTP [Beners96] o HTTPS [Freier96] y descarga el código Java en su navegador. La aplicación es interpretada por la máquina virtual de Java [Sun95] implementada en el navegador, de forma independiente a la plataforma y navegador utilizados por el cliente.

6.3.4.2 Interoperabilidad de aplicaciones

Uno de los mayores problemas en la intercomunicación de aplicaciones distribuidas físicamente es la representación de la información enviada. Si desarrollamos dos aplicaciones nativas sobre dos plataformas distintas y hacemos que intercambien información, deberemos establecer previamente la representación de la información utiliza-

da. Por ejemplo, una variable entera en el lenguaje de programación C [Ritchie78] puede tener una longitud y representación binaria distinta en cada una de las plataformas.

A la complejidad de definir la representación de la información y traducir ésta a su representación nativa, se le une la tarea de definir el protocolo de comunicación: el modo en el que las aplicaciones deben dialogar para intercambiar dicha información.

Existen especificaciones estándar definidas para interconectar aplicaciones nativas sobre distintas plataformas. Un ejemplo es el complejo protocolo GIOP (*General Inter–ORB Protocol*) [OMG95] definido en CORBA [Baker97]. Establece el protocolo y representación de información necesarios para interconectar cualquier aplicación nativa a través de la arquitectura de objetos distribuidos CORBA. Este tipo de *middleware* proporciona un elevado nivel de abstracción, facilitando el desarrollo de aplicaciones distribuidas, pero conlleva una serie de inconvenientes:

- Requiere una elevada cantidad de código adicional para implementar el protocolo y la traducción de la información enviada por la red. Este código recibe el nombre de ORB (*Object Request Broker*).
- Aumenta el volumen de información enviada a través de la red de ordenadores al implementar un protocolo de propósito general que proporcione un mayor nivel de abstracción.

Si las aplicaciones se desarrollan sobre una misma máquina abstracta, el envío de la información se puede realizar directamente en el formato nativo de ésta, puesto que existe una máquina virtual en toda plataforma donde se ejecuten las aplicaciones. La traducción de la información de la máquina a la plataforma física la lleva a cabo en el intérprete de la máquina virtual. El resultado es poder interconectar aplicaciones codificadas en una máquina abstracta y ejecutadas en plataformas, y por lo tanto dispositivos, totalmente dispares.

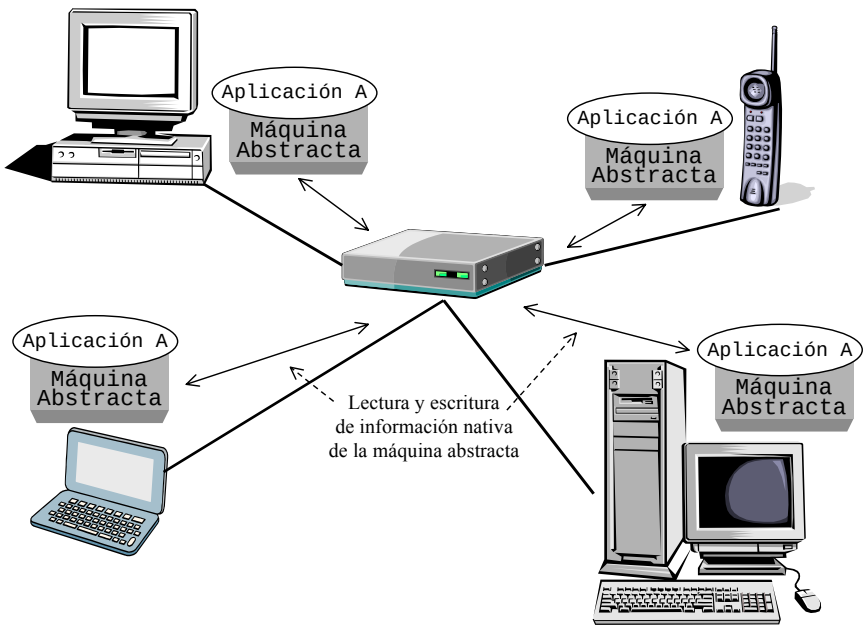

Figura 23: Interoperabilidad nativa de aplicaciones sobre distintas plataformas.

En el lenguaje de programación Java [Gosling96], podemos enviar los datos en su representación nativa. No estamos restringidos al envío y recepción de tipos simples de datos sino que es posible también enviar objetos, convirtiéndolos a una secuencia de bytes (esto es conocido como serialización –*serialization*) [Campione99].

Mediante el paquete de clases ofrecidas en `java.net`, es posible implementar un protocolo propio de un tipo de aplicaciones sobre TCP/IP o UDP/IP [Raman98]. Si deseamos obtener un mayor nivel de abstracción para el desarrollo de aplicaciones distribuidas, al igual que teníamos con CORBA, podemos utilizar RMI (*Remote Method Invocation*) sin sobrecargar tanto la red de comunicaciones [Sun97e]. RMI desarrolla un protocolo de interconexión de aplicaciones Java –JRMP, *Java Remote Method Protocol*– que permite, entre otras cosas, invocar a métodos de objetos ubicados en otras máquinas virtuales.

La potencia de las aplicaciones distribuidas desarrolladas sobre la plataforma Java cobra significado cuando unimos las dos características comentadas en este punto: aplicaciones cuyo código es distribuido a través de la red, capaces de interoperar entre sí, sin llevar a cabo una conversión de datos, de forma independiente a la plataforma física en la que se estén ejecutando.

6.3.5 Diseño y Coexistencia de Sistemas Operativos

La utilización del concepto de máquina abstracta ha estado presente también en el desarrollo de sistemas operativos. Podemos clasificar su utilización en función del objetivo buscado, de la siguiente forma:

- Desarrollo de sistemas operativos distribuidos y multiplataforma.
- Ejecución de aplicaciones desarrolladas sobre cualquier sistema operativo.

Existen sistemas operativos, como el VM/ESA de IBM [IBM2000c], que utilizan la conjunción de ambas funcionalidades en la utilización de una máquina abstracta.

6.3.5.1 Diseño de sistemas operativos distribuidos y multiplataforma

Estos sistemas operativos aprovechan todas las ventajas de la utilización de máquinas abstractas comentadas en los puntos anteriores para desarrollar un sistema operativo distribuido y multiplataforma. Sobre la descripción de una máquina abstracta, se implementa un intérprete de la máquina virtual en toda aquella plataforma en la que vaya a desarrollarse el sistema operativo. En el código de la máquina, se desarrollan servicios propios del sistema operativo que permitan interactuar con el sistema y elevar el nivel de abstracción; lo conseguido finalmente es:

- Una aplicación para este sistema operativo es portable a cualquier plataforma.
- Las aplicaciones no se limitan a utilizar los servicios de la máquina, sino que podrán codificarse en un mayor nivel de abstracción: el ofrecido por los servicios sistema operativo.
- El propio sistema operativo es portable, puesto que ha sido desarrollado sobre la máquina abstracta. No es necesario pues, codificar cada servicio para cada plataforma.

- La interoperabilidad de las aplicaciones es uniforme, al estar utilizando únicamente el modelo de computación de la máquina abstracta. En otros sistemas operativos es necesario especificar la interfaz exacta de acceso a sus servicios.
- Las aplicaciones desarrolladas sobre este sistema operativo pueden ser distribuidas físicamente por el sistema operativo, ya que todas serán ejecutadas por un intérprete de la misma máquina abstracta.
- En la comunicación de aplicaciones ejecutándose en computadores distribuidos físicamente, no es necesario establecer traducciones de datos. La interacción es directa, al ejecutarse todas las aplicaciones sobre la misma máquina abstracta –en distintas máquinas virtuales o intérpretes.

Existen distintos sistemas operativos desarrollados sobre una máquina abstracta ya sean comerciales, de investigación o didácticos.

6.3.5.2 Coexistencia de sistemas operativos

En este apartado veremos la utilización del concepto de máquina virtual desde un punto de vista distinto. Este concepto puede definirse como un acceso uniforme a los recursos de una plataforma física (de una máquina real). Es una interfaz de interacción con una plataforma física que puede ser dividida en un conjunto de máquinas virtuales.

La partición de los recursos físicos de una plataforma, mediante un acceso independiente, permite la ejecución de distintos sistemas operativos inmersos en el operativo que se encuentre en ejecución. Dentro de un sistema operativo, se desarrollan tantas máquinas virtuales como sistemas inmersos deseemos tener. La ejecución de una aplicación desarrollada para un sistema operativo distinto al activo, se producirá sobre la máquina virtual implementada para el sistema operativo "huésped". Esta ejecución utilizará los recursos asignados a su máquina virtual.

La ventaja obtenida se resume en la posibilidad de ejecutar aplicaciones desarrolladas sobre cualquier sistema operativo sin tener que reiniciar el sistema, es decir, sin necesidad de cambiar el sistema operativo existente en memoria.

El principal inconveniente frente a los sistemas descritos en el punto anterior, "Diseño de sistemas operativos distribuidos y multiplataforma", es la carencia de interacción entre las aplicaciones ejecutadas sobre distintos sistemas operativos: no es posible comunicar las aplicaciones "huésped" entre sí, ni con las aplicaciones del propio operativo.

El primer sistema operativo que utilizó de esta forma una máquina virtual fue el producto OS/2 de IBM. Se declaró como el sucesor de IBM del sistema operativo DOS, desarrollado para microprocesadores Intel 80286. Este sistema era capaz de ejecutar en un microprocesador Intel 80386 varias aplicaciones MS–DOS, Windows y las propias aplicaciones gráficas nativas, haciendo uso de la implementación de distintas máquinas virtuales.

IBM abandonó el proyecto iniciado con su sistema operativo OS/2. Sin embargo, parte de sus características fueron adoptadas en el desarrollo de su operativo VM/ESA desarrollado para servidores IBM S/390 [IBM2000c]. Este sistema permite ejecutar

aplicaciones desarrolladas para otros sistemas operativos, utilizando una máquina virtual como modo de acceso a los recursos físicos. El empleo de una máquina virtual es aprovechado además para la portabilidad e interoperabilidad del código: cualquier grupo de aplicaciones desarrolladas sobre esta máquina abstracta puede interoperar entre sí, de forma independiente al tipo de servidor sobre el que se estén ejecutando.

Otro producto que utiliza el concepto de máquina virtual para multiplexar el acceso a una plataforma física es *VMware Virtual Platform* [Jones99][VMWARE]. Ejecutándose en Windows o en Linux permite lanzar aplicaciones codificadas para Windows, MS–DOS, Linux, FreeBDS y Solaris 7 para Intel entre otros.

6.4 Aportación de la Utilización de Máquinas Abstractas

Hemos estudiado el concepto de máquina abstracta y las ventajas que aporta la utilización de las mismas. De todas ellas, podemos indicar que las aportaciones que son significativas para logar los requisitos impuestos a nuestro sistema (descritos en el capítulo 2), son:

- Independencia del lenguaje de programación (6.1.1).
- Portabilidad de su código (6.3.2).
- Independencia de la plataforma (6.3.2).
- Distribución de aplicaciones (6.3.4.1).
- Interoperabilidad nativa de aplicaciones distribuidas (6.3.4.2).

En la siguiente sección se estudiarán una serie de sistemas existentes que, haciendo uso de máquinas abstractas, consiguen estos beneficios (total o parcialmente).

6.5 Panorámica de Utilización de Máquinas Abstractas

En las secciones anteriores se han introducido los conceptos de máquina abstracta y máquina virtual así como las distintas posibilidades prácticas que podían aportar a un sistema informático. Del abanico global de ventajas que ofrece su utilización, subrayamos un conjunto de ellas como deseables para nuestro sistema.

En esta sección presentaremos un estudio de distintas implementaciones realizadas, que puedan servir para cumplimentar los requisitos que se han establecido. En concreto, analizaremos casos prácticos de la utilización de máquinas abstractas como instrumento para conseguir plataformas independientes del lenguaje, sistema operativo y microprocesador. Los sistemas estudiados identifican una máquina abstracta como base computacional y, sobre ésta, desarrollan el conjunto de su plataforma, de modo independiente al entorno computacional real existente.

Una vez descritos y analizados los distintos sistemas, identificaremos qué requisitos han sido cumplidos por los casos prácticos existentes y cómo adaptarlos a nuestros objetivos, así como sus insuficiencias y las necesidades surgidas para superarlas

6.5.1 Smalltalk–80

El sistema Smalltalk–80 tiene sus raíces en el centro de investigación de Xerox, Palo Alto. Empezó en la década de los setenta, pasando por la implementación de tres sistemas principales: Smalltalk 72, 76 y 80; el número corresponde al año en el que fueron diseñados [Krasner83].

Los esfuerzos del grupo investigador estaban orientados a la obtención de un sistema que fuese manejable por personas no informáticas. Para llegar a esto, se apostó por un sistema basado en gráficos, interfaces interactivas y visuales, y una mejora en la abstracción y flexibilidad a la hora de programar. La abstracción utilizada, más cercana al humano, era la orientación a objetos, y, respecto a la flexibilidad, se podía acceder cómodamente y en tiempo de ejecución a las clases y objetos que existían en el sistema [Mevel87].

El sistema Smalltalk–80 está dividido básicamente en dos grandes componentes [Goldberg83]:

1. La imagen virtual: Colección de objetos, instancias de clases, que proporcionan estructuras básicas de datos y control, primitivas de texto y gráficos, compiladores y manejo básico de la interfaz de usuario.
2. La máquina virtual: Intérprete de la imagen virtual y de cualquier aplicación del usuario. Dividida en:
    - El gestor de memoria: Se encarga de gestionar los distintos objetos en memoria, sus relaciones y su ciclo de vida. Para ello implementa un recolector de basura de objetos.
    - El intérprete de instrucciones: Analiza y ejecuta las instrucciones en tiempo de ejecución. Las operaciones que se utilizan son un conjunto de primitivas que operan directamente sobre el sistema.

Aunque exista una especificación formal de la máquina abstracta [Goldberg83], con la existencia de ésta no se buscó directamente una plataforma independiente, sino aprovechar determinadas características propias de un lenguaje interpretado[11]: flexibilidad del sistema y un nivel de abstracción base adecuado –orientación a objetos.

Todas las aplicaciones constituyentes del sistema de Smalltalk–80 están escritas en el propio lenguaje Smalltalk y, al ser éste interpretado, se puede acceder dinámicamente a todos los objetos existentes en tiempo de ejecución (como señalábamos en el punto 6.3.3). El mejor ejemplo es el *Browser* del sistema [Mevel87], mostrado en la Figura 24: es una aplicación que recorre todas las clases (los objetos derivados de `Class`) del sistema (del diccionario `Smalltalk`) y nos visualiza la información de éstas:

- Categoría a la que pertenece la clase.
- Un texto que describe la funcionalidad de ésta.

[11] Smalltalk no es interpretado directamente. Pasa primero por una fase de compilación a un código binario en la que se detecta una serie de errores. Este código binario será posteriormente interpretado por el simulador o procesador software de la máquina abstracta.

- Sus métodos.
- Sus atributos.
- Código que define el comportamiento de cada método.

Esta información es modificable en todo momento; además tenemos la documentación real, puesto que se genera dinámicamente, de todas las clases, objetos, métodos y atributos existentes en el sistema.

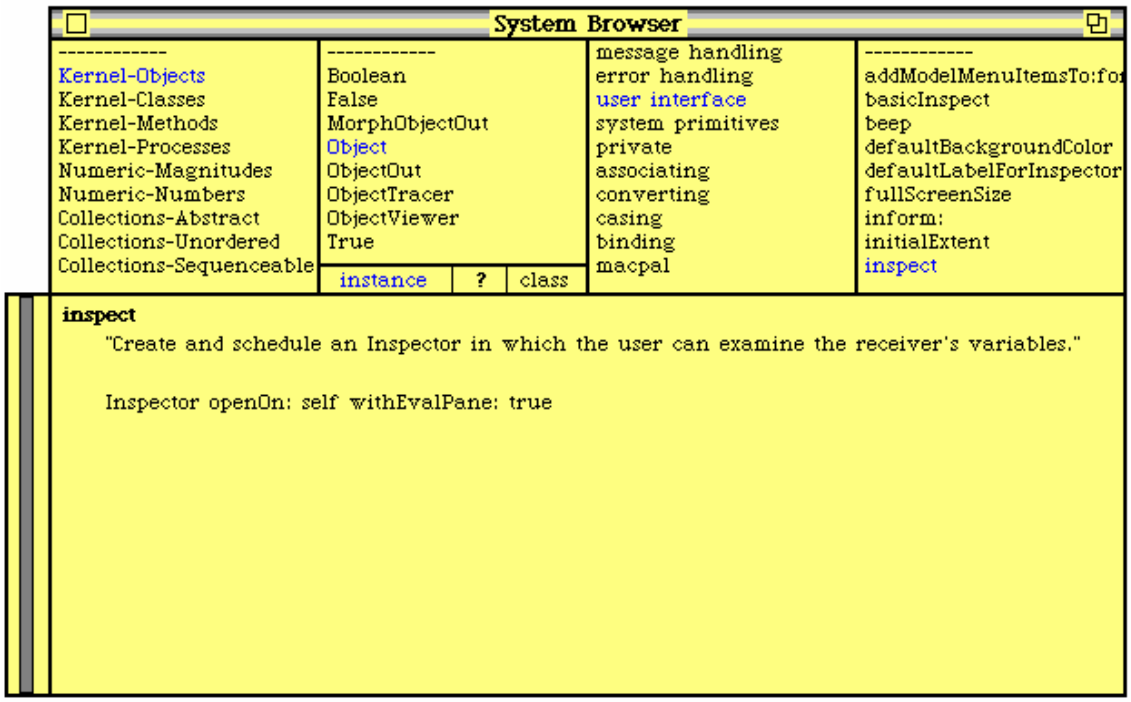

Figura 24: Acceso al método "inspect" del grupo "user interface", propio del objeto "Object" perteneciente al grupo "Kernel–Objects".

El aplicar estos criterios con todos los programas del sistema hace que la programación, depuración y análisis de éstos sea muy sencilla. La consecución de dichos objetivos es obtenida gracias a la figura de una máquina abstracta, encargada de ejecutar el sistema.

Smalltalk–80 es pues, un sistema de computación que consigue, mediante una máquina virtual, una integración entre todas sus aplicaciones, una independencia de la plataforma utilizada y un nivel de abstracción orientado a objetos para su modelo de computación base.

6.5.2 Java

En 1991, un grupo de ingenieros trabajaba en el proyecto Green: un sistema para interconectar cualquier aparato electrónico. Se buscaba poder programar cualquier aparato mediante un lenguaje sencillo, un intérprete reducido, y que el código fuese totalmente portable. Especificaron una máquina abstracta con un código binario en bytes y, como lenguaje de programación de ésta, intentaron utilizar C++ [Stroustrup98]. Se dieron cuenta de su complejidad y dependencia de la plataforma de ejecución y lo redujeron al lenguaje Oak, renombrándolo posteriormente a Java [Gosling96].

Ninguna empresa de electrodomésticos se interesó en el producto y en 1994 el proyecto había fracasado. En 1995, con el extensivo uso de Internet, desarrollaron en Java un navegador HTTP [Beners96] capaz de ejecutar aplicaciones Java en el cliente, descargadas previamente del servidor –denominándose éste HotJava [Kramer96]. A raíz de esta implementación, Netscape introdujo en su *Navigator* el emulador de la máquina abstracta permitiendo añadir computación, mediante *applets*, a las páginas estáticas HTML utilizadas en Internet [Beners93]; Java resultó mundialmente conocido.

Un programa en el lenguaje Java se compila para ser ejecutado sobre una plataforma independiente [Kramer96]. Esta plataforma de ejecución independiente está formada básicamente por:

- La máquina virtual de Java (*Java Virtual Machine*) [Sun95].
- La interfaz de programación de aplicaciones en Java o *core* API (*Application Programming Interface*).

Esta dualidad, que consigue la independencia de una plataforma y el mayor nivel de abstracción en la programación de aplicaciones, es igual que la que hemos identificado en Smalltalk–80: imagen y máquina virtual. El API es un conjunto de clases que están compiladas en el formato de código binario de la máquina abstracta [Sun95]. Estas clases son utilizadas por el programador para realizar aplicaciones de una forma más sencilla.

La máquina virtual de Java es el procesador del código binario de esta plataforma. El soporte a la orientación a objetos está definido en su código binario aunque no se define su arquitectura[12]. Por lo tanto, la implementación del intérprete y la representación de los objetos en memoria quedan a disposición del diseñador del simulador, que podrá utilizar las ventajas propias de la plataforma real.

La creación de la plataforma de Java se debe a la necesidad existente de abrir el espacio computacional de una aplicación. Las redes de computadores interconectan distintos tipos de ordenadores y dispositivos entre sí. Se han creado múltiples aplicaciones, protocolos, *middlewares* y arquitecturas cliente servidor para resolver el problema de lo que se conoce como programación distribuida [Orfali96].

Una red de ordenadores tiene interconectados distintos tipos de dispositivos y ordenadores, con distintas arquitecturas hardware y sistemas operativos. Java define una máquina abstracta para conseguir implementar aplicaciones distribuidas que se ejecuten en todas las plataformas existentes en la red (plataforma independiente). De esta forma, cualquier elemento conectado a la red, que posea un procesador de la máquina virtual y el API correspondiente, será capaz de procesar una aplicación –o una parte de ésta – implementada en Java (característica estudiada previamente en el punto 6.3.4).

Para conseguir este tipo de programación distribuida, la especificación de la máquina abstracta de Java se ha realizado siguiendo fundamentalmente tres criterios [Venners98]:

[12] Si bien no define exactamente su arquitectura, supone la existencia de determinados elementos de ésta como una pila [Sun95].

1. Búsqueda de una plataforma independiente.
2. Movilidad del código a lo largo de la red.
3. Especificación de un mecanismo de seguridad robusto.

La característica de definir una plataforma independiente está implícita en la especificación de una máquina abstracta. Sin embargo, para conseguir la movilidad del código a través de las redes de computadores, es necesario tener en cuenta otra cuestión: la especificación de la máquina ha de permitir obtener código de una aplicación a partir de una máquina remota (punto **6.3.4.1**).

La distribución del código de una aplicación Java es directamente soportada por la funcionalidad de los "*class loaders"* [Gosling96]; se permite definir la forma en la que se obtiene el software –en este caso las clases –, pudiendo éste estar localizado en máquinas remotas. De esta forma la distribución de software es automática, puesto que se puede centralizar todo el código en una sola máquina y ser cargado desde los propios clientes al principio de cada ejecución.

Adicionalmente, cabe mencionar la importancia que se le ha dado a la seguridad y robustez de la plataforma. Las redes representan un instrumento para aquellos programadores que deseen destruir información, escamotear recursos o simplemente molestar. Un ejemplo de este tipo de aplicaciones puede ser un virus distribuido, que se ejecute en nuestra máquina una vez demandado a través de la red.

El primer módulo en la definición de la plataforma enfocado hacia su robustez, es el "*code verifier*" [Sun95]. Una vez que el código ha sido cargado, entra en fase de verificación: la máquina se asegura de que la clase esté correctamente codificada y de que no viole la integridad de la máquina abstracta [Venners98].

El sistema de seguridad en la ejecución de código, proporcionado por la máquina virtual, es ofrecido por el "*security manager*"; una aplicación cliente puede definir un *security manager* de forma que se limite, por la propia máquina virtual, el acceso a todos los recursos que se estime oportuno[13]. Se pueden definir así los límites del software obtenido; verbigracia: en la ejecución de un *applet* descargado de un servidor Web, no se le permitirá borrar archivos del disco duro.

[13] Este concepto ha sido definido como caja de arena o "*sandbox*". Debido a lo estricto que resultaba para el desarrollo de *applets*, en Java2 se ha hecho más flexible mediante la utilización de archivos de políticas en el "*Java Runtime Environment*" del cliente [Dageforde2000].

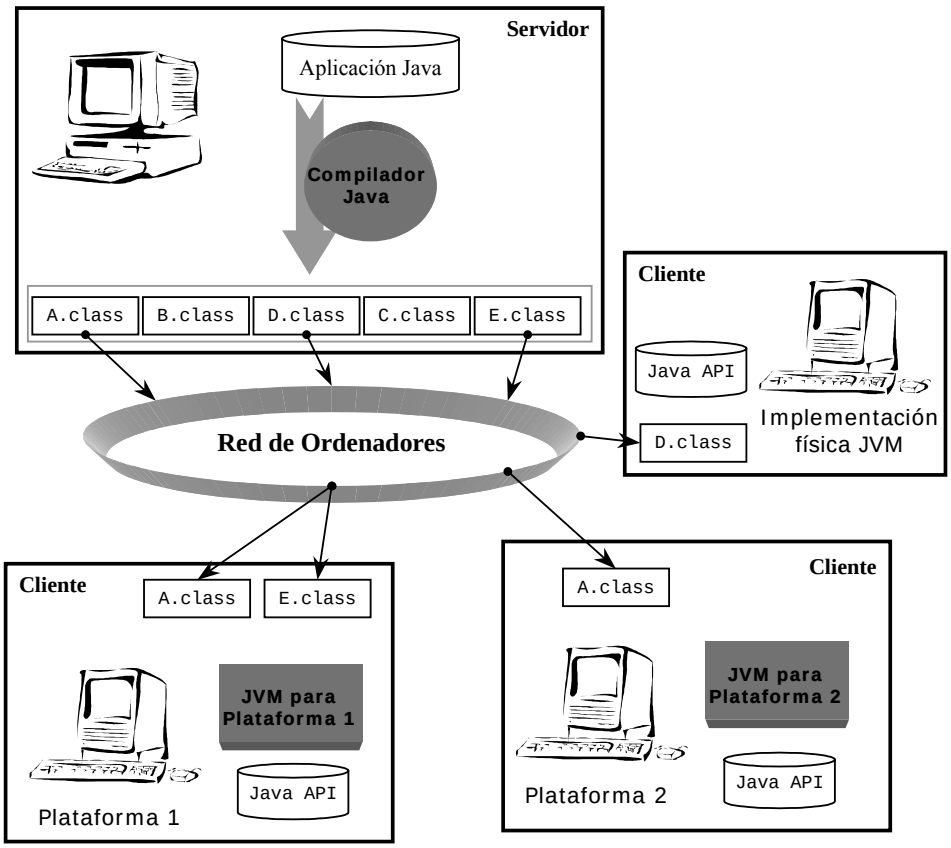

Figura 25: Ejemplo de entorno de programación distribuida en Java.

En la Figura 25 se aprecia cómo se enlazan los distintos conceptos mencionados. En un determinado equipo servidor, se diseña una aplicación y se compila al código especificado como binario de la máquina abstracta. Este código es independiente de la plataforma en la que fue compilado (característica de la utilización de máquinas abstractas, estudiada en 6.3.2), pudiéndose interpretar en cualquier arquitectura que posea dicha máquina virtual.

Parte de esta aplicación es demandada por una máquina remota. La porción del software solicitado es obtenida a través de la red de comunicaciones, y es interpretado por la implementación del procesador en el entorno cliente. La forma en la que la aplicación cliente solicita el código a la máquina remota es definida en su *class loader* (un ejemplo típico es un *applet* que se ejecuta en el intérprete de un *Web browser*, demandando el código del servidor HTTP).

El código obtenido puede ser "controlado" por un *security manager* definido en la aplicación cliente. De esta forma la máquina virtual comprueba los accesos a recursos no permitidos, lanzando una excepción en tiempo de ejecución si se produjese alguno [Gosling96]. El resultado es una plataforma independiente, capaz de distribuir de forma sencilla sus aplicaciones en una red de computadores y con un mecanismo de control de código correcto y seguro.

Aunque en la especificación de la máquina virtual de Java [Sun95] no se identifica una implementación, el comportamiento de ésta se describe en términos de subsistemas, zonas de memoria, tipos de datos e instrucciones. En la Figura 26 se muestran los subsistemas y zonas de memoria nombrados en la especificación.

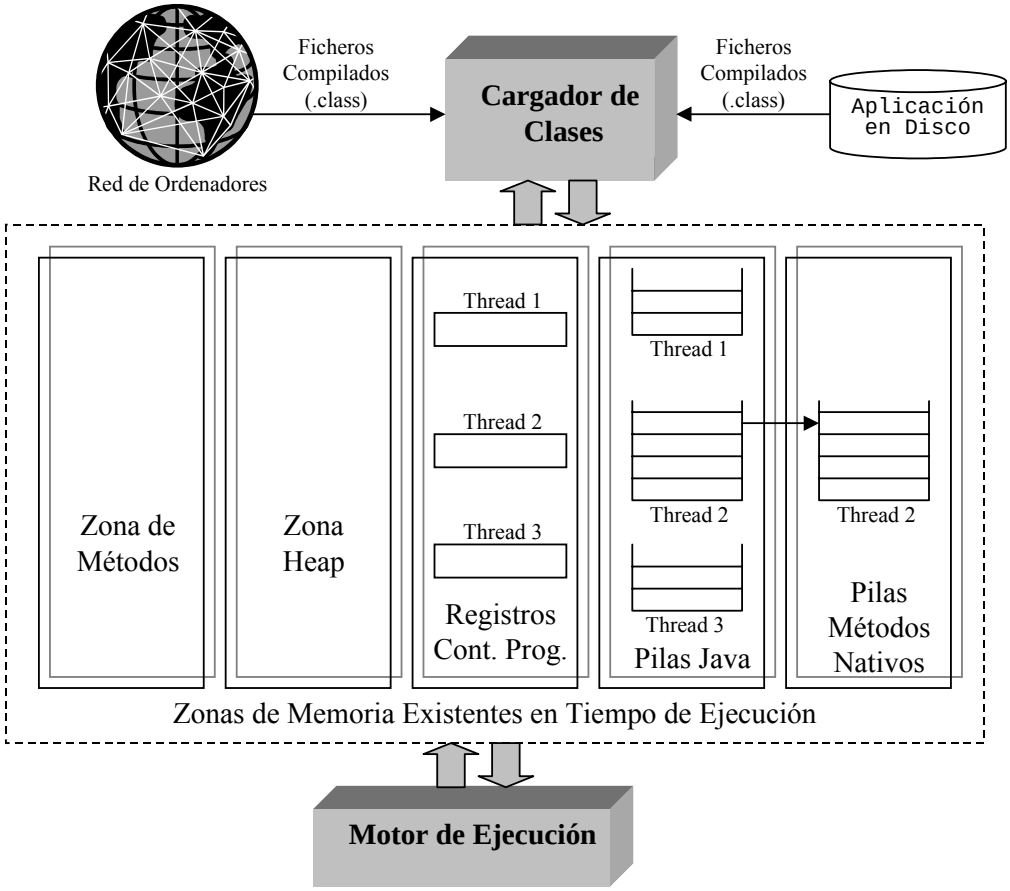

Figura 26: Arquitectura interna de la máquina virtual de Java.

La máquina virtual tiene un subsistema de carga de clases (*class loader*): mecanismo utilizado para cargar en memoria tipos –clases e *interfaces* [Gosling96]. La máquina también tiene un motor de ejecución (*execution engine*): mecanismo encargado de ejecutar las instrucciones existentes en los métodos de las clases cargadas [Venners98].

La máquina virtual identifica básicamente dos zonas de memoria necesarias para ejecutar un programa:

1. Las inherentes a la ejecución de la máquina virtual: una zona por cada ejecución de la máquina.
2. Las inherentes a los hilos (*threads*) de ejecución dentro de una máquina: una zona por cada hilo existente en la máquina.

En el primer grupo tenemos el área de métodos y el área *heap*. En la zona de métodos se introduce básicamente la información y los datos propios de las clases de la aplicación. En la zona *heap* se representan los distintos objetos existentes en tiempo de ejecución.

Por cada hilo en ejecución, se crea una zona de pila, un registro contador de programa y una pila de métodos nativos –métodos codificados en la plataforma real mediante el uso de una interfaz de invocaciones: JNI (*Java Native Interface*) [Sun97c]. En cada hilo se va incrementando el contador de programa por cada ejecución, se van api-

lando y desapilando contextos o marcos en su pila, y se crea una pila de método nativo si se ejecuta un método de este tipo.

Con respecto a la implementación de la máquina virtual, comentaremos que Javasoft y Sun Microelectronics™ han desarrollado la familia de procesadores JavaChip™ [Kramer96]: picoJava™, microJava™ y ultraJava™ [Sun97]. Estos microprocesadores son implementaciones físicas de intérpretes de la máquina abstracta de Java, optimizados para las demandas de esta plataforma como son la multitarea y la recolección de basura.

6.5.3 .Net

En 1998 un equipo de trabajo de Microsoft comenzó a trabajar en un proyecto que denominaron Next Generation Windows Services (NGWS). Este equipo se fusionó con el grupo encargado de desarrollar la versión 7 del Visual Studio con el objetivo de desarrollar un entorno de ejecución común para todos los lenguajes incluidos en él de forma que permitiese a terceras empresas crear lenguajes adaptados al entorno. Finalmente, en el 2000 Microsoft dio a conocer todo este trabajo que denominaron Microsoft.NET

La plataforma .NET ha sido diseñada para cumplir los siguientes objetivos [MicrosoftNET]:

- Proporcionar un entorno de programación orientado a objetos consistente, independientemente de dónde se ejecute.
- Proporcionar un entorno de ejecución de código que minimice el proceso del despliegue del software y los conflictos de versiones.
- Proporcionar un entrono de ejecución que promueva la ejecución segura de código, independientemente de si el código ha sido creado por un desconocido o por un tercero de confianza.
- Proporcionar un entorno de ejecución que elimine los problemas de rendimiento de los entornos de *script* o los interpretados.
- Hacer que el trabajo del desarrollador sea homogéneo a través de una amplia variedad de aplicaciones, como pueden ser aplicaciones Windows, o aplicaciones WEB.
- Construir todas las comunicaciones basadas en estándares de la industria, de tal forma que se pueda asegurar que el código basado en .NET se pueda integrar con cualquier otro código.

La plataforma .NET tiene dos componentes principales: el *Common Language Runtime* (CLR, nombre que recibe la implementación realizada por Microsoft) y la librería de clases (BCL). El CLR es la base sobre la que se construye la plataforma .NET. Se puede ver el CLR como un agente que gestiona el código en tiempo de ejecución, proporcionando servicios de base tales como gestión de memoria o gestión de hilos, a la vez que hace cumplir seguridad de tipos y otras formas de verificación de código que promueven la seguridad y la robustez. El código que se ejecuta a través del CLR es conocido como código gestionado *–managed code–*, mientras que el código que se ejecuta fuera de él es conocido como código no gestionado *–unmanaged code–*. Con el fin de que la plataforma soporte cualquier lenguaje de programación (con ciertas restricciones), todos los programas deben ser compilados a un lenguaje intermedio que es el que

será reconocido por el CLR y podrá ser ejecutado por él. Este es uno de los puntos clave de la plataforma, pues así aplicaciones escritas en distintos lenguajes podrán interactuar de forma directa y, desde un lenguaje de programación, se podrá hacer uso de componentes desarrollados en cualquier otro lenguaje de programación.

La librería de clases (BCL), el otro componente básico, es una extensa colección de tipos orientados a objetos que pueden ser reutilizados (desde cualquier lenguaje de programación) para desarrollar cualquier tipo de aplicación, desde una aplicación de consola, una aplicación con interfaz gráfica o una aplicación basada en servicios Web XML. La librería de clases está implementada y compilada al lenguaje intermedio como parte del estándar de la plataforma, por lo que cualquier lenguaje de programación que necesite utilizarla puede hacerlo libremente, sin tener que implementarla para su propio lenguaje.

En la Figura 27 se puede ver la forma de trabajar de la plataforma .NET. Por compatibilidad con aplicaciones ya desarrolladas, o nuevas aplicaciones que se desarrollen de forma no compatible con .NET, se pueden ejecutar aplicaciones no gestionadas que pueden invocar a componentes, o aplicaciones, gestionadas por el CLR.

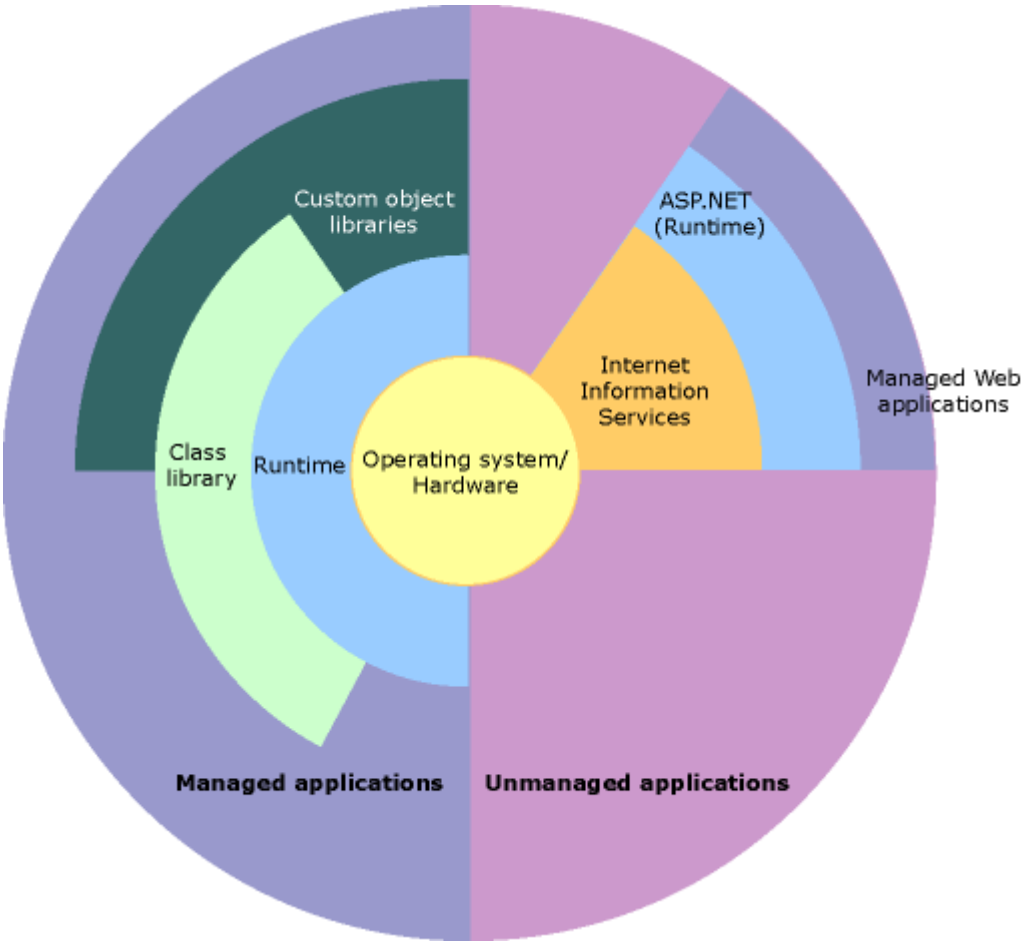

Figura 27: Funcionamiento de la plataforma .NET

A continuación, vamos a mostrar las características más reseñables del estándar (CLI) y veremos diversas implementaciones existentes del mismo.

6.5.3.1 Common Language Infrastructure (CLI)

A partir del año 2000, Microsoft trabajó junto a otras empresas (como Intel y Hewlett–Packard) con el fin de definir un estándar abierto para la plataforma .NET. Este estándar es conocido como *Common Language Infrastructure* (CLI) y fue ratificado por ECMA – *European Computer Manufacturers Association*– con el ECMA–335 [ECMA335] en diciembre de 2001 y posteriormente por ISO/IEC (*International Organisation for Standardization/International Electrotechnical comisión)* con el ISO/IEC 23271:2006[ISO23271] en abril de 2003.

El CLI es una especificación y no una implementación (de hecho existen varias implementaciones de la especificación, como veremos más adelante). Es una especificación para el código ejecutable y el entorno de ejecución sobre el cual correrá. Ésta define un entorno que permite la existencia de múltiples lenguajes de alto nivel que pueden ejecutarse sobre diversas plataformas sin tener que recompilarse.

La especificación del CLI se basa principalmente en cuatro aspectos:

- El sistema común de tipos o *Common Type System* (CTS). El CTS proporciona un amplio sistema de tipos que permite soportar los tipos y operaciones que se encuentran en muchos lenguajes de programación. De hecho se ha definido con la pretensión de soportar la implementación completa de un gran número de lenguajes de programación.
- Metadatos. El CLI utiliza los metadatos para describir y referenciar a los tipos definidos por el CTS. Los metadatos se almacenan de forma independiente al lenguaje por lo que proporcionan un mecanismo para el intercambio de información entre distintas herramientas que manipulan programas (compiladores, depuradores) y entre estas herramientas y el entorno de ejecución.
- La especificación de lenguaje común o *Common Language Specification* (CLS). El CLS es un conjunto de reglas básicas que cualquier lenguaje que quiera usarse en el CLI debería cumplir con el fin de poder interrelacionar con otros lenguajes que las cumplan. El CLS especifica un subconjunto del CTS y una serie de convenciones sobre su uso. Mediante esto se pretende que cualquier librería de clases que sólo ofrezca aspectos (clases, interfaces, métodos, etc.) soportados por el CLS y haciendo uso de sus convenciones pueda ser usada por cualquier programa escrito en cualquier otro lenguaje que cumpla el CLS.
- El Sistema Virtual de Ejecución o *Virtual Execution System* (VES). El VES implementa y hace cumplir el modelo de CTS en ejecución. Es responsable de la carga y ejecución de los programas escritos para el CLI, proporciona los servicios necesarios para ejecutar código y datos gestionados –*managed*– (entendiendo por ello que es el CLI el que controla el uso de la memoria que la aplicación realiza y la seguridad de los hilos de ejecución) usando los metadatos para conectar en el momento de su ejecución módulos generados independientemente.

Todos los lenguajes compatibles con el CLI se compilan al *Common Intermediate Language*[14]*(CIL)*, que es el conjunto de instrucciones soportado por el VES [ECMA335]. El CIL es el lenguaje de más bajo nivel –inteligible para el hombre– del CLI. Es un lenguaje orientado a objetos y está basado en pila. Los programas escritos en distintos lenguajes son compilados a CIL para ser ensamblados en *bytecode* que es ejecutado por el VES.

El código intermedio, ya transformado en *bytecode*, es encapsulado en *assemblies*– ensamblados– que, en el caso de las implementaciones sobre Windows, se corresponden con ficheros PE –*Portable Executable*– (es decir .exe o .dll), veremos su formato más adelante. Un *assembly* está compuesto por uno o más ficheros y es auto descriptivo, al contener los metadatos necesarios para ello.

A la hora de ejecutar un ensamblado por parte del VES, éste toma el código en CIL del ensamblado y lo traduce a código nativo de la plataforma donde está ejecutándose .NET (puede ser Windows, Linux, etc.) con el fin de que sea esta traducción la que se ejecute de forma optimizada (existen implementaciones de la plataforma .NET que no traducen a código nativo sino que realizan una interpretación).

En el momento de la ejecución es cuando la seguridad toma un papel muy importante. La ejecución de código malicioso o erróneo puede causar problemas de confidencialidad, integridad o accesibilidad. Con este fin el VES está diseñado para proteger al sistema de estos problemas; para ello hace uso del concepto de caja de arena –*sandbox*– donde se ejecutará el código. En el contexto de .NET esto es llamado Dominio de la Aplicación. Antes de ejecutar el código se realizan una serie de comprobaciones y validaciones. Se describen las fases brevemente a continuación [Murison05]:

- Carga del ensamblado: se hace una verificación inicial, se verifica que se corresponde con el formato PE/COFF y se inspeccionan los metadatos contenidos en él.
- Gestor de políticas: se aplican las políticas de seguridad en función de los metadatos del componente, y de la situación del entorno de ejecución. Mediante estas políticas se asignan los derechos que debe tener el código.
- Carga de clases: se cargan las clases individualmente, siendo analizadas cada una.
- Compilador y Verificador *Just In Time*: antes de realizar la compilación a código nativo se realizan una serie de comprobaciones de seguridad, como las comprobaciones de tipos y otras como accesos a recursos no permitidos, etc. La compilación tiene lugar bajo demanda, de tal manera que sólo los métodos que son usados son compilados.
- Gestor de código nativo: el código es ejecutado a la vez que se hacen cumplir las políticas de seguridad (restricciones en el acceso a recursos, etc.).

El *Code Access Security –CAS–* es uno de los pilares de .NET [LaMachia02]. Los mecanismos de seguridad de software tradicionales se centran en la identidad del usuario que ejecuta el software en vez de en la identidad del código que se ejecuta, con

[14] También conocido como IL– *Intermediate Language*– o MSIL –*Microsoft Intermediate Language*.

lo que un código no fiable ejecutado por un usuario en quien se confía obtiene los permisos del usuario y puede producir daños. CAS soluciona este problema asignando los permisos al código, basándose en la *evidencia* presentada por el código y las políticas de seguridad que estén en vigor en el entorno de ejecución. Esto no elimina la existencia de permisos basados en el usuario, lo complementa. Un código sólo podrá recibir un subconjunto de los permisos que tenga el usuario que está ejecutando dicho código, por ejemplo nunca podrá recibir permiso para acceder a un dispositivo al que no tenga acceso el usuario que lo está ejecutando. CAS sirve a tres propósitos:

- Autenticar la identidad del código. Se debe poder verificar desde dónde se está ejecutando el código así como quién lo originó. Para ello es necesario que el código esté firmado digitalmente. Esta información es lo que se denomina *evidencia*.
- Autorizar al código a acceder a los recursos. Los *permisos* para acceder a recursos como ficheros, dispositivos, red, etc. se conceden en base a la *evidencia* presentada y a la *política* vigente en el sistema. Por ejemplo esto sirve para no permitir el acceso a ciertos recursos locales, como el disco duro, a ensamblados descargados de Internet. Nunca se pueden dar permisos a un código que no tenga el usuario que lo está ejecutando.
- Hacer cumplir los permisos otorgados. No tendría sentido conceder o denegar permisos de acceso a recursos si después no se hacen cumplir.

Por todo lo visto algunas de las ventajas propias de desarrollar aplicaciones basadas en la arquitectura .NET son [Johnston2000]:

- Utilización del mismo código base .Net puesto que su codificación es independiente de la máquina hardware (se compila una única vez y se puede ejecutar en cualquier plataforma donde haya una implementación del CLI). Esto facilita el despliegue de las aplicaciones.
- La depuración de aplicaciones se realiza de forma indiferente al lenguaje de implementación utilizado. Se pueden depurar aplicaciones, incluso si están codificadas en distintos lenguajes de programación.
- Los desarrolladores pueden reutilizar cualquier clase, mediante herencia o composición, indistintamente del lenguaje empleado.
- Unificación de tipos de datos y manejo de errores (excepciones).
- La máquina virtual ejecuta código con un sistema propio de seguridad, llamado *Code Access Security CAS.*
- Se puede examinar cualquier código contenido en una clase, obtener sus métodos, mirar si el código está firmado, e incluso conocer su árbol de jerarquía[15].

Basándose en esta especificación se han desarrollado diversas implementaciones de la máquina abstracta, a continuación veremos algunas de ellas.

[15] Estas facilidades se obtienen al tener una máquina abstracta dotada de introspección. Estudiaremos este concepto más adelante.

6.5.3.2 CLR

El CLR (que se corresponde a *Common Language Runtime)* es una implementación comercial de la plataforma .Net realizada por Microsoft [Burton02]. A día de hoy es la implementación más extendida y utilizada ya que se incorpora al Sistema Operativo Windows. Esta implementación sólo funciona sobre plataforma Windows.

El CLR es la máquina virtual que proporciona un motor de ejecución de código, independiente del sistema en el que fue desarrollado, y compilado desde cualquier lenguaje de programación.

En el CLR se ha implementado una compilación *Just In Time* del programa en IL a código nativo, optimizado según el entorno, que es ejecutado por la máquina en la que se encuentra el CLR. Mediante esta técnica se consigue un rendimiento mucho mayor que en la interpretación [Anthony05].

Un método es compilado siempre antes de su primer uso, y todos esos métodos compilados son guardados en una caché para su reutilización futura, variando el tamaño de esta caché en función del comportamiento del programa.

Realmente, en el CLR se han implementado dos compiladores JIT para la generación de código, así como una estrategia de compilación independiente no relacionada que puede o no usarse para un código concreto [Manzoor06]:

- Econo–JIT: permite compilar programas muy rápidamente pero no optimiza el código que genera, permitiendo procesar el programa rápidamente a costa de una ejecución de su código más lenta. La aplicación más típica de este compilador es la ejecución de *scripts*.
- JIT estándar: Este compilador procesa el código más lentamente, pero sí optimiza el código generado, consiguiendo mejor rendimiento en tiempo de ejecución. Éste es el que el CLR usará para ejecutar la mayoría de código.
- Compilación en tiempo de instalación: Esta técnica permite al CLR compilar la aplicación a código nativo cuando el programa se instala. La instalación del programa será más lenta, pero estará más adaptada a las características particulares de la máquina donde se va a ejecutar y su velocidad se verá teóricamente aumentada gracias a ello.

Sobre la base de la plataforma (CLR) y gracias a sus características, se ofrece un marco de trabajo base, válido para cualquier plataforma, que puede ser utilizado desde cualquier lenguaje de programación. Es conocido como *Base Class Library* (BCL*)* y proporciona clases que encapsulan una serie de funciones comunes como son la lectura y escritura de ficheros, manipulación de documentaos XML, etc. Estas clases son utilizadas por el programador para realizar aplicaciones de una forma más sencilla. Esta librería puede ser utilizada por cualquier lenguaje que cumpla con el CLS.

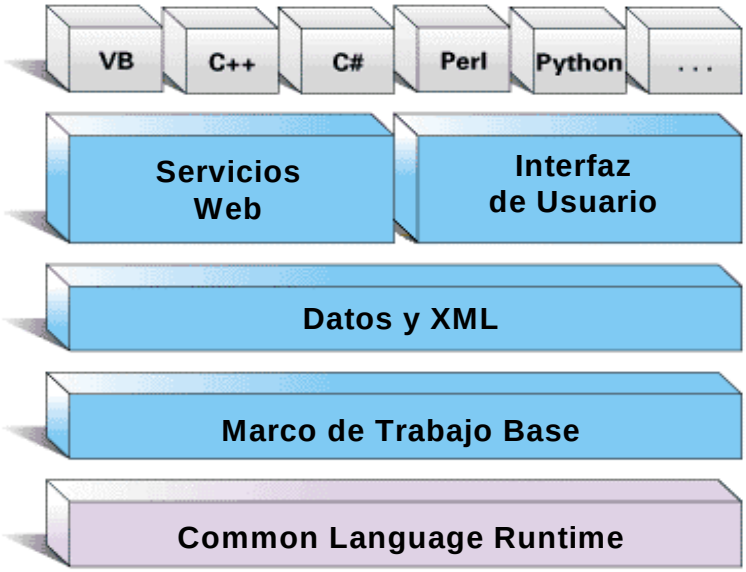

Figura 28: Arquitectura de la plataforma .NET.

Microsoft no se ha limitado a implementar únicamente el CLI sino que ha añadido una serie de componentes y funcionalidades que extienden el estándar, por ejemplo Windows.Forms [WindowsForms] o Passport .NET (que ha pasado a llamarse Windows Live ID). Es decir el CLR, junto a la BCL, es un súper–conjunto de la implementación del CLI. En la Figura 29 podemos observar qué componentes forman parte del estándar y qué componentes son añadidos (tanto para el CLR como para el SSCLI que veremos a continuación)

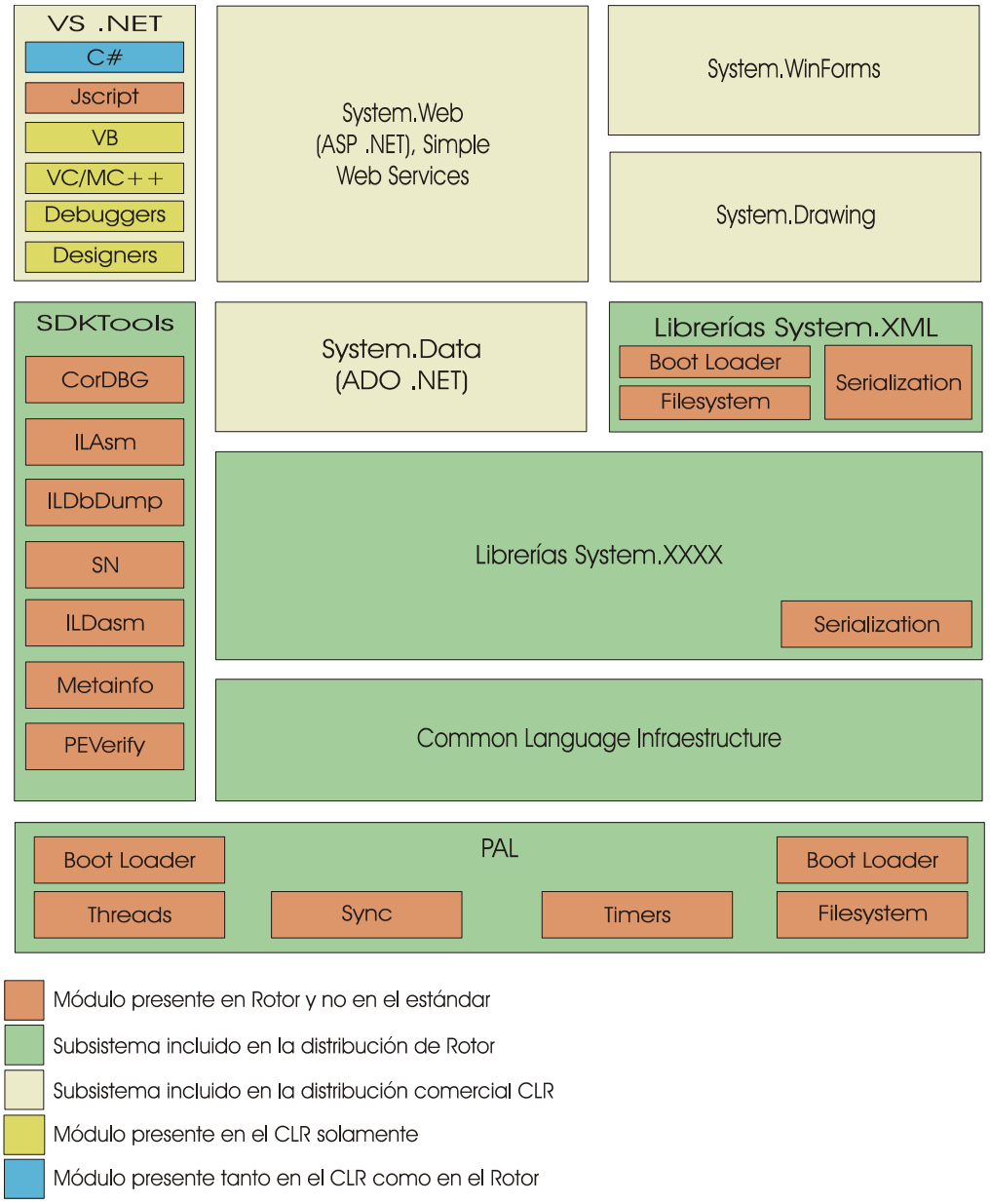

Figura 29: componentes del CLR y del SSCLI

6.5.3.3 SSCLI – Rotor

El *Shared Source Common Language Infrastructure (SSCLI)*, también conocido como *Rotor*, es una implementación del CLI realizada por Microsoft bajo una licencia especial, mediante la cual el código está disponible gratuitamente, y se puede modificar y redistribuir, con la condición de no ser usado comercialmente. De hecho, puede decirse que el *SSCLI* es una versión "recortada" de la versión comercial, el CLR. La política de desarrollar y hacer pública una versión paralela de una herramienta comercial de forma gratuita tiene los siguientes propósitos:

- Permitir comprobar la portabilidad del estándar CLI.
- Permitir estudiar y conocer el entorno de Microsoft comercial CLR, dado el alto grado de similitud entre ambos entornos.
- Favorecer los proyectos de investigación académicos sobre el CLI.

- Establecer una guía para entender el estándar CLI, y además poder servir como modelo para otras implementaciones del estándar que se quieran desarrollar.

El SSCLI viene preparado para poder ser portado a diversas plataformas. La versión 1.0 de SSCLI está preparada para ejecutarse sobre entornos *Windows*, *FreeBSD* y *Mac OS X*, aunque la versión 2.0 sólo es capaz de ejecutarse sobre *Windows XP SP2.*

En principio, para poder portar el SSCLI a una nueva plataforma lo único que habría que modificar es la PAL (*Platform Adaptation Layer* o capa de adaptación de plataforma). El propósito principal de la capa PAL es aislar a la implementación del resto de las capas de los detalles concretos de los diferentes sistemas operativos sobre los que puede ejecutarse el SSCLI. Al ser un producto *Microsoft* y empezar siendo un desarrollo para el sistema *Win32*, la PAL presenta un subconjunto de llamadas al *API* de *Win32*. En este subconjunto no se incluye la totalidad del *API* de *Win32*, sólo aquellas partes que son usadas por el CLI.

6.5.3.4 Proyecto Mono

Mono [Mono] es un proyecto inicialmente impulsado por Miguel de Icaza en *Ximian*, empresa que posteriormente fue adquirida por *Novell* [Novell] la cual sigue patrocinando el proyecto. El propósito principal es crear un conjunto de herramientas compatibles con los estándares .NET de Microsoft (que como ya se ha dicho es a su vez una implementación del estándar CLI [ECMA335]), siendo *Mono* una implementación del mismo estándar y sus tecnologías asociadas. *Mono* es a su vez portable y está disponible en un número mayor de plataformas como *Linux*, *FreeBSD*, *UNIX*, *MacOS X*, *Solaris* y *Windows*.

El código de *Mono* está disponible para desarrolladores de una forma más sencilla que el de .NET. El compilador de *C#* y otras herramientas tienen una licencia *GPL* (*GNU General Public License*) [GNUGPL], mientras que las librerías en tiempo de ejecución se encuentran licenciadas mediante la licencia *LGPL* (*GNU Lesser General Public License*) [GNULGPL] y las librerías de clases se encuentran dentro de la licencia *MIT* [MITLicense]. Todas estas licencias convierten a *Mono* en un proyecto esencialmente de código abierto.

Además de esto, la máquina virtual de *Mono* incluye motores de compilación JIT (*Just In Time*) para diferentes procesadores (*SPARC*, *ARM*, *PowerPC*, *x86* en 32 *bits* y también para algunas plataformas de 64 *bits*). Mientras la plataforma es capaz de convertir el código IL a código nativo para estas plataformas usando el compilador integrado, para otras plataformas diferentes se requerirá el uso de intérpretes.

Existe una versión, llamada Mint, que viene de *Mono Interpreter*, que es un motor de ejecución del código de la plataforma que en vez de realizar compilación a código nativo realiza una interpretación. La utilidad de este sistema es su facilidad para ser portado a otras plataformas, con mucho menos esfuerzo que el crear un compilador para la plataforma sobre la que se ejecute.

6.5.3.5 Proyecto DotGNU – Portable .Net

El proyecto *DotGNU* [DotGNU] es una alternativa a la plataforma .NET de Microsoft, creado con la intención de evitar el monopolio de esta tecnología. Es una proyecto de software libre bajo licencia GNU GPL [GNUGPL] Este proyecto consiste en un conjunto de herramientas *software,* que ejecutan aplicaciones diseñadas para el estándar *CLI* sobre diferentes sistemas operativos y plataformas, con herramientas diversas, así como la implementación propia del estándar *CLI,* denominada *Portable .NET* [Pnet].

Este entorno de desarrollo fue inicialmente propuesto para *GNU/Linux*, pero actualmente existen versiones para múltiples sistemas operativos como *Windows*, *NetBSD*, *FreeBSD*, *Solaris*, y *MacOS X* entre otros, asegurando así su portabilidad. Adicionalmente, soporta varias arquitecturas como *x86*, *PPC*, *ARM*, *Sparc*, *s390*, *Alpha*, *ia–64*, y *PARISC*.

Una de las principales motivaciones del proyecto *DotGNU* es la total compatibilidad tanto con los mencionados estándares ECMA correspondientes al lenguaje *C#* [ECMA334] y al CLI [ECMA335], como con la implementación comercial del CLI de *Microsoft* (el CLR), de forma que una aplicación desarrollada para uno de los sistemas funcione correctamente en el otro y viceversa.

6.5.3.6 Formato de fichero PE– Portable Executable

Tal y como comentamos anteriormente, Microsoft ha extendido el formato de fichero PE/COFF (*Portable Executable and Common Object File Format*) [PE/COFF] para poder utilizarlo con el CLR. De esta forma, cuando un fichero es cargado por el sistema operativo, éste detecta que es un fichero que debe ser ejecutado por el CLR, y traspasa el control a este último, el cual se encarga de cargar las secciones correspondientes y procesarlas.

Figura 30: Estructura de un fichero ejecutable .NET

En la Figura 30 se puede ver la estructura general de un fichero ejecutable. Se puede observar que inicialmente existe una cabecera PE/COFF que es procesada por el sistema operativo. Con la información aquí contenida el sistema operativo traspasa el control al CLR el cual carga la "cabecera CLR" y posteriormente los "datos CLR" (metadatos, código IL, etc.).

El compilador genera las siguientes secciones en un fichero PE:

1. *.text*: esta sección es de sólo lectura y contiene la cabecera del CLR así como los metadatos, el código IL, la información para el control de excepciones gestionadas por el CLR, es decir, los correspondientes al código gestionado –*managed*– y los recursos gestionados.
2. *.sdata*: sección de lectura y escritura que contiene datos.
3. *.reloc*: sección de sólo lectura que contiene relocalizaciones. Realmente la única misión de esta sección en relocalizar la dirección a la cual el *Entry-Point*, punto de entrada de la aplicación, salta.
4. *.rscr*: sección de sólo lectura que contiene recursos no gestionados (generalmente el icono principal del ejecutable).

La cabecera CLR contiene información sobre lo que contiene el fichero (código gestionado, código no gestionado, recursos), qué versiones del CLR precisa para funcionar, offset y tamaño de cada parte de información contenida en el fichero, y el *EntryPointToken* (en el caso de ser un fichero .exe) que indica cual es el método que será el punto de entrada de la aplicación, si el método se encuentra en el mismo módulo, o el fichero donde se encuentra el módulo que lo contiene, de tratarse de un assembly multimódulo.

Los metadatos son, por definición datos que describen datos. En el contexto del CLR, metadatos significa un sistema de descriptores de todos los elementos que son declarados o referenciados en el módulo. Debido a que el modelo de programación es orientado a objetos los elementos representados en los metadatos son clases y sus miembros, con sus correspondientes atributos, propiedades y relaciones.

Estructuralmente los metadatos están organizados como una base de datos relacional normalizada.

El CLR ofrece una serie de librerías que permiten acceder a los metadatos así como al código IL de un ensamblado (es decir ofrece introspección).

6.5.4 Aportaciones y Carencias de los Sistemas Estudiados

En todos los sistemas estudiados, se ha obtenido un sistema de computación multiplataforma (requisito 2.2.1). Si bien en Smalltalk no era el objetivo principal, la independencia del sistema operativo y procesador hardware ha sido buscada en el diseño de todas ellas.

Las máquinas abstractas de Smalltalk y Java fueron desarrolladas para dar un soporte portable a un lenguaje de programación. Su arquitectura está pensada especialmente para trabajar con un determinado lenguaje. Sin embargo, .NET no es dependiente de un determinado lenguaje de programación (aunque el lenguaje C# [ECMA334] sí

que fue diseñado específicamente para esta plataforma). De esta forma, .NET es la única plataforma, de las vistas, pensada de acuerdo con el requisito 2.1.2.

Si bien, cuando se diseñaron las plataformas estudiadas se trataba de obtener una serie de resultados con su utilización, ninguna se desarrolló para solucionar un tipo específico de problema. Las máquinas virtuales aquí presentadas no resuelven un problema *ad hoc* por lo que sirven para resolver cualquier problema (lo que posibilitará el cumplimiento del requisito 2.1.3).

El tamaño de todas las plataformas estudiadas es realmente elevado. Implementar una máquina virtual sobre una nueva plataforma, o en un sistema empotrado es costoso; una ratificación de esto es la distribución de la máquina virtual de Java por Javasoft mediante un *plug–in*: las diversas modificaciones de la JVM y su complejidad hace que se ralentice la implementación de ésta en los navegadores de Internet, y por ello se distribuye con el sistema un *plug–in* [Javasoft99].

6.6 Conclusiones

Hemos visto cómo, inicialmente, la utilización de una plataforma virtual, basada en una máquina abstracta, buscaba la portabilidad de su código binario. Los sistemas desarrollados conseguían este objetivo para un determinado lenguaje; otras plataformas más modernas, que gozan de gran éxito comercial, también han sido desarrolladas para dar soporte principalmente a un lenguaje de programación (como es el caso de *Java Virtual Machine*). Existen otras que son independientes del lenguaje, como la plataforma .NET, ofreciendo a los desarrolladores la posibilidad de beneficiarse de las ventajas del uso de una máquina virtual sin verse restringidos al empleo de un lenguaje de programación determinado (requisito 2.1.2).

Muchas de las plataformas existentes, a la hora de ejecutar un programa, compilan el código fuente a un código intermedio, propio de la máquina abstracta, que es el que realmente será ejecutado. De esta manera, si el sistema permite modificar esa representación del programa en código intermedio (es decir modificar el programa), no es necesario disponer del código fuente original para poder adaptarlo (requisito 2.3.1 y 2.3.2) y todo el proceso se realiza de forma independiente al lenguaje de programación en el que se codificase inicialmente (requisito 2.1.2)

La portabilidad del código ofrecida por una máquina virtual es explotada en entornos distribuidos de computación. Si el código de una plataforma puede ser ejecutado en cualquier sistema (requisito 2.2.1), éste puede ser fácilmente distribuido a través de una red de ordenadores. Además, al existir un único modelo de computación y representación de datos, las aplicaciones distribuidas en un entorno heterogéneo de computación pueden intercambiar datos de forma nativa –sin necesidad de utilizar una interfaz de representación común.

Hemos estudiado plataformas de diseño avanzadas, con las ventajas ya mencionadas. Estos sistemas ofrecen programación portable, distribución de código, interacción de aplicaciones distribuidas y un modelo de computación único, basado en el paradigma de la programación orientada a objetos. La implementación de aplicaciones es más homogénea y sencilla, para entornos heterogéneos distribuidos. Cabe destacar el

proyecto .NET de Microsoft (6.5.3), que además de las ventajas mencionadas incluye la independencia del lenguaje.

Aunque ninguna máquina virtual existente ofrece solución a todos los requisitos planteados por nuestro sistema, existen algunas que pueden ofrecer soporte para, sobre ellas, crear la solución que satisfaga todos los requisitos fijados. A modo de conclusión señalaremos que implementar una máquina virtual de estas características es complejo y costoso. Si para desarrollar nuestro sistema optásemos por modificar alguna de las plataformas existentes para ampliar su funcionalidad y conseguir los objetivos marcados estaríamos incurriendo en un sobrecoste excesivo. El hecho de tener que portar la máquina modificada a cualquier plataforma donde se necesite ejecutar y, ante cualquier modificación de la especificación original del estándar, tener que realizar las modificaciones necesarias desaconsejan esta posibilidad (que no respetaría el requisito 2.2.4).

CAPÍTULO 7
REFLEXIÓN COMPUTACIONAL

A lo largo de este capítulo, introduciremos los conceptos propios de una técnica utilizada para conseguir un elevado grado de flexibilidad en determinados sistemas: la reflexión[16] computacional. Haremos un estudio genérico de las posibilidades de los sistemas reflectivos y estableceremos distintas clasificaciones en función de distintos criterios.

En el siguiente capítulo, todos los términos definidos y estudiados aquí serán utilizados para analizar los distintos sistemas dotados de reflexión.

7.1 Conceptos de Reflexión

7.1.1 Reflexión

Reflexión o reflexión (*reflection*) es la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo, así como ajustar su comportamiento en función de ciertas condiciones [Maes87].

El dominio computacional de un sistema reflectivo –el conjunto de información que computa– añade al dominio de un sistema convencional la estructura y comportamiento de sí mismo. Se trata pues de un sistema capaz de acceder, analizar y modificarse a sí mismo, como si de una serie de estructuras de datos propias de un programa de usuario se tratase.

7.1.2 Sistema Base

Aquel sistema de computación que es dominio de otro sistema computacional distinto, denominado metasistema [Golm97]. El sistema base es el motor de computación (intérprete, *software* o *hardware*) de un programa de usuario.

[16] Traducción de *reflection***.** Muchos autores la han traducido también como reflectividad. Nosotros utilizaremos indistintamente uno u otro a lo largo de esta memoria.

7.1.3 Metasistema

Aquel sistema computacional que posee por dominio de computación a otro sistema computacional denominado sistema base. Un metasistema es por lo tanto un intérprete de otro intérprete.

7.1.4 Cosificación

Cosificación o concretización (*reification)* es la capacidad de un sistema para acceder al estado de computación de una aplicación, como si se tratase de un conjunto de datos propios de una aplicación de usuario [Smith82].

La cosificación es la posibilidad de acceso desde un sistema base a su metasistema, en el que se puede manipular el sistema base como si de datos se tratase; es el salto del entorno de computación de usuario al entorno de computación del intérprete de la aplicación de usuario.

Si podemos cosificar[17] un sistema, podremos acceder y modificar su estructura y comportamiento y, por lo tanto, podremos añadir a su dominio computacional su propia semántica; estaremos trabajando pues, con un sistema reflectivo.

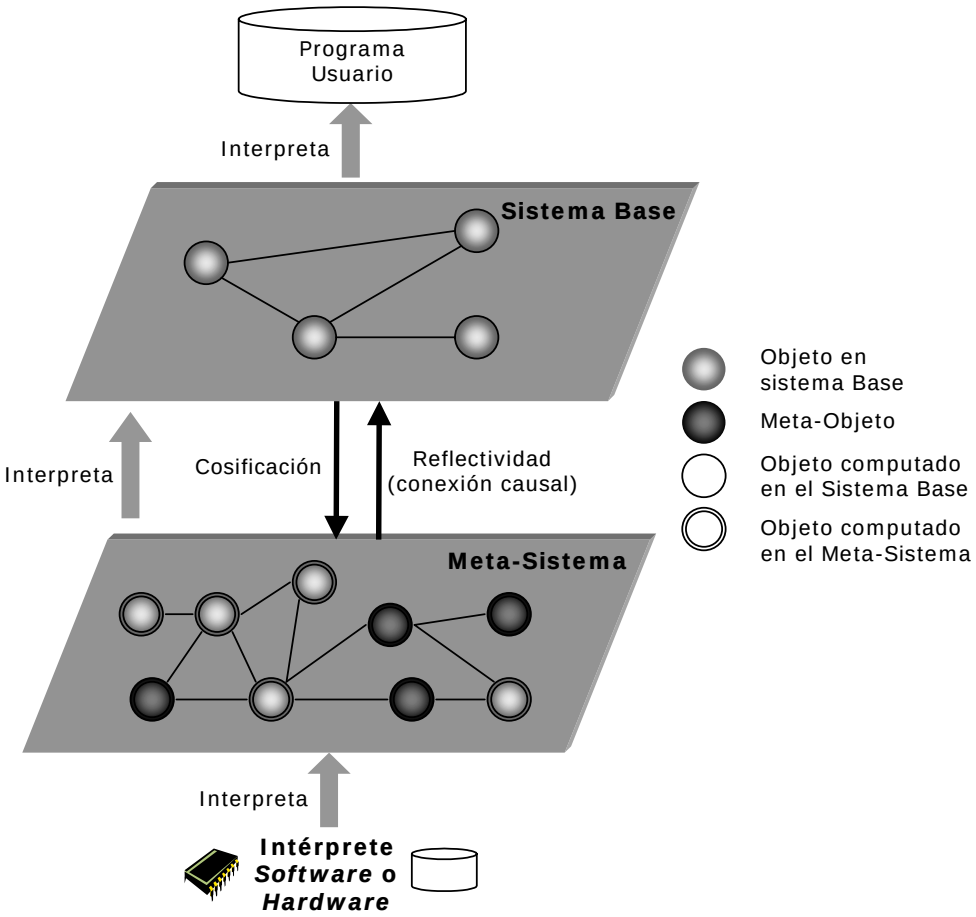

Figura 31: Entorno de computación reflectivo.

[17] Cosificar o concretizar: transformar una representación, idea o pensamiento en cosa.

7.1.5 Conexión Causal

Un sistema es causalmente conexo si su dominio computacional alberga el entorno computacional de otro sistema (permite la cosificación del sistema base) y, al modificar la representación de éste, los cambios realizados causan un efecto en su propia computación (la del sistema base) [Maes87b].

Un sistema reflectivo es aquél capaz de ofrecer cosificación completa de su sistema base, e implementa un mecanismo de conexión causal. La forma en la que los distintos sistemas reflectivos desarrollan esta conexión varía significativamente.

7.1.6 Metaobjeto

Identificando la orientación a objetos como paradigma del sistema reflectivo, puede definirse un metaobjeto como un objeto del metasistema que contiene información y comportamiento propio del sistema base [Kiczales91]. La abstracción del metaobjeto no es necesariamente la de un objeto propio del sistema base, sino que puede representar cualquier elemento que forme parte de éste –por ejemplo, el mecanismo del paso de mensajes.

7.1.7 Reflexión Completa

La conexión causal no es suficiente para un sistema reflectivo íntegro. Cuando la cosificación de un sistema es total, desde el metasistema se puede acceder a cualquier elemento del sistema base y la conexión causal se produce para cualquier elemento modificado, entonces ese sistema posee reflexión completa (*completeness*) [Blair97]. En este caso, el metasistema debe ofrecer una representación íntegra del sistema base (todo podrá ser modificado).

Como veremos más adelante en el estudio de sistemas reales, la mayoría de los sistemas existentes limitan a priori el conjunto de características del sistema base que pueden ser modificados mediante el mecanismo de reflexión implementado.

7.2 Reflexión como una Torre de Intérpretes

Para explicar el concepto de reflexión computacional, así como para posteriormente implementar prototipos de demostración de su utilidad, Smith identificó el concepto de reflexión computacional como una torre de intérpretes [Smith82].

Diseñó un entorno de trabajo en el que todo intérprete de un lenguaje era a su vez interpretado por otro intérprete, estableciendo una torre de interpretaciones. El programa de usuario se ejecuta en un nivel de interpretación 0; cobra vida gracias a un intérprete que lo anima desde el nivel computacional 1. En tiempo de ejecución el sistema podrá cosificarse, pasando el intérprete del nivel 1 a ser computado por otro intérprete –la reflexión computacional implica una computación de la computación. En este caso (Figura 32.b) un nuevo intérprete', desde el nivel 2, pasa a computar el intérprete inicial. El dominio computacional del intérprete' –mostrado en la Figura 32 por un doble recuadro – estará formado por la aplicación de usuario (nivel 0) y la interpretación directa de éste (nivel 1). En esta situación se podrá acceder tanto a la aplicación de usuario (por ejemplo, para modificar sus objetos) como a la forma en la que éste es interpretado

(por ejemplo, para modificar la semántica del paso de mensajes). El reflejo de dichos cambios en el nivel 1 es lo que se denomina reflexión o reflexión.

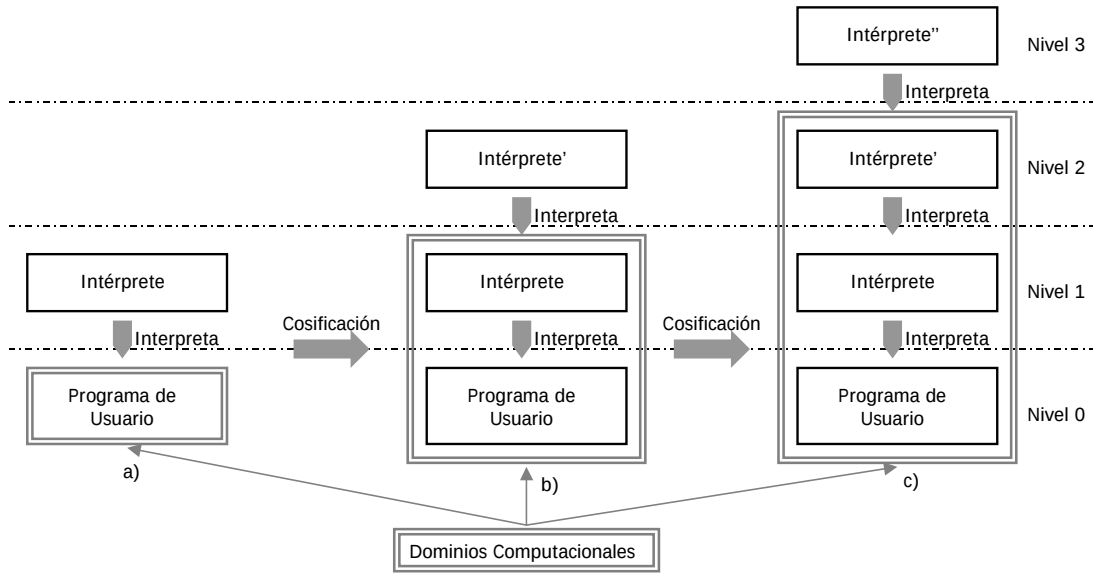

Figura 32: Torre de intérpretes definida por Smith.

La operación de cosificación se puede aplicar tantas veces como deseemos. Si el intérprete' procesa al intérprete del nivel 1, ¿podemos cosificar este contexto para obtener un nuevo nivel de computación? Sí (Figura 32.c); se crearía un nuevo intérprete'' que procesaría los niveles 0, 1 y 2, ubicándose éste en el nivel computacional 3. En este caso reflejaríamos la forma en la que se refleja la aplicación de usuario (tendríamos una meta–meta–computación).

Un ejemplo clásico es la depuración de un sistema (*debug*). Si estamos ejecutando un sistema y deseamos depurar mediante una traza todo su conjunto, podremos cosificar éste, para generar información relativa a su ejecución desde su intérprete. Todos los pasos de su interpretación podrán ser trazados. Si deseamos depurar el intérprete de la aplicación en lugar de la propia aplicación, podremos establecer una nueva cosificación aplicando el mismo sistema [Douence99].

Este entorno de trabajo fue definido por Smith como una torre de intérpretes [Smith82]. Definió el concepto de reflexión, y propuso la semántica de ésta para los lenguajes de programación, sin llevar a cabo una implementación. La primera implementación de este sistema fue realizada mediante la modificación del lenguaje Lisp [Steele90], denominándolo 3–Lisp [Rivières84].

La torre infinita de intérpretes siempre tiene un nivel máximo de computación. Siempre existirá un intérprete que no sea interpretado a su vez por otro procesador: un intérprete *hardware* o microprocesador. Un microprocesador es un intérprete de un lenguaje de bajo nivel, implementado físicamente.

En la Figura 33 mostramos un ejemplo real de una torre de intérpretes. Se trata de la implementación de un intérprete de un lenguaje de alto nivel, en el lenguaje de programación Java [Gosling96].

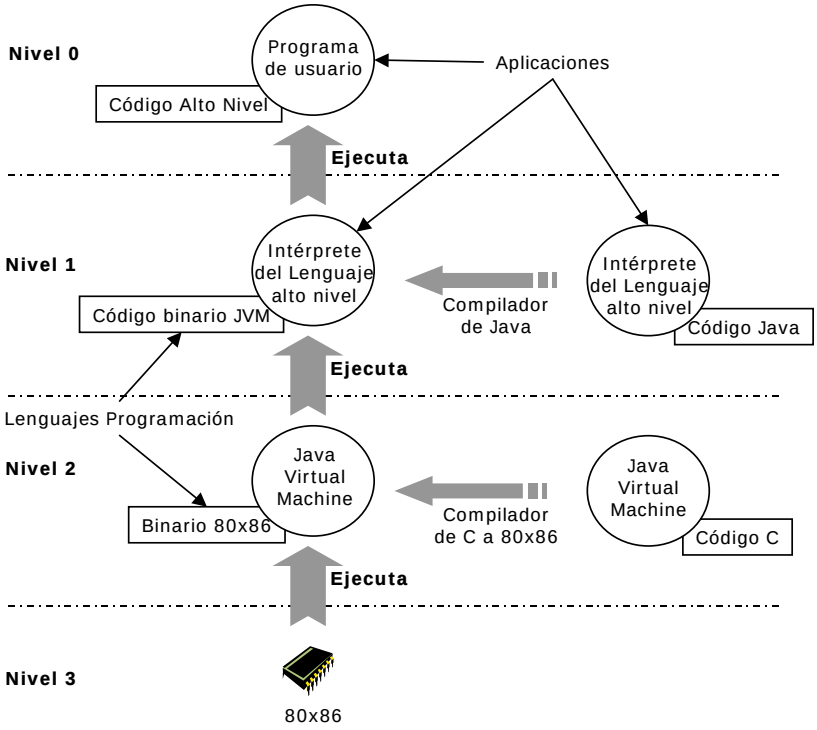

Figura 33: Ejemplo de torre de intérpretes en la implementación de un intérprete en Java.

Java es un lenguaje que se compila a un código binario de una máquina abstracta denominada "*Java Virtual Machine*" [Lindholm96]. Para interpretar este código – previamente compilado – podemos utilizar un emulador de la máquina virtual programado en C y compilado, por ejemplo, para un i80X86. Finalmente el código binario de este microprocesador es interpretado una implementación física de éste. En el ejemplo expuesto tenemos un total de cuatro niveles computacionales en la torre de intérpretes.

Si pensamos en el comportamiento o la semántica de un programa como una función de nivel n, ésta podrá ser vista realmente como una estructura de datos desde el nivel n+1. En el ejemplo, el intérprete del lenguaje de alto nivel define la semántica de dicho lenguaje, y la máquina virtual de Java define la semántica del intérprete del lenguaje de alto nivel. Por lo tanto, el intérprete de Java podrá modificar la semántica computacional del lenguaje de alto nivel, obteniendo así reflexión computacional.

Cada vez que en la torre de intérpretes nos movamos en un sentido de niveles ascendente, podemos identificar esta operación como cosificación[18] (*reification*). Un movimiento en el sentido contrario se identificará como reflexión (*reflection*) [Smith82].

7.3 Clasificaciones de Reflexión

Pueden establecerse distintas clasificaciones en el tipo de reflexión desarrollado por un sistema, en función de diversos criterios. En este punto identificaremos algunos de esos criterios, clasificaremos los sistemas reflectivos basándonos en el criterio selec-

[18] Ciertamente hacemos datos (o cosa) un comportamiento.

cionado, y describiremos cada uno de los tipos. Puede verse una clasificación más exhaustiva en [Ortin02b].

Para cada clasificación de sistemas reflectivos identificada, mencionaremos algún ejemplo existente para facilitar la comprensión de dicha división.

7.3.1 Qué se Refleja

Hemos definido reflexión en 7.1 como "la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo". El grado de información que un sistema posee acerca de sí mismo –aquello susceptible de ser cosificado y reflejado – establece la siguiente clasificación.

7.3.1.1 Introspección

La introspección es la capacidad de un sistema para poder inspeccionar u observar, pero no modificar, los objetos de un sistema [Foote92]. En este tipo de reflexión, se facilita al programador el acceder al estado del sistema en tiempo de ejecución: el sistema ofrece la capacidad de conocerse a sí mismo.

Esta característica se ha utilizado de forma pragmática en numerosos sistemas. A modo de ejemplo, la plataforma Java [Kramer96] ofrece introspección mediante su paquete `java.reflect` [Sun97d]. Gracias a esta capacidad, en Java es posible almacenar un objeto en disco sin necesidad de implementar un solo método: el sistema accede de forma introspectiva al objeto, analiza sus atributos y los convierte en una secuencia de bytes para su posterior envío a un flujo[19] [Eckel2000]. Sobre este mismo mecanismo Java implementa además su sistema de componentes JavaBeans [Sun96]: dado un objeto, en tiempo de ejecución, se determinará su conjunto de propiedades y operaciones.

Otro ejemplo de introspección, en este caso para código nativo compilado, es la adición de información en tiempo de ejecución al estándar ISO/ANSI C++ (RTTI, *Run-Time Type Information*) [Kalev98]. Este mecanismo introspectivo permite conocer la clase de la que es instancia un objeto, para poder ahormar éste de forma segura respecto al tipo [Cardelli97].

7.3.1.2 Reflexión estructural

Se refleja el estado estructural del sistema en tiempo de ejecución –elementos tales como las clases, el árbol de herencia, la estructura de los objetos y los tipos del lenguaje – permitiendo tanto su observación como su manipulación [Ferber88].

Mediante reflexión estructural, se puede acceder al estado de la ejecución de una aplicación desde el sistema base. Podrá conocerse su estado, acceder las distintas partes del mismo, y modificarlo si se estima oportuno. De esta manera, una vez reanudada la ejecución del sistema base (después de producirse la reflexión), los resultados pueden ser distintos a los que se hubieren obtenido si la modificación de su estado no se hubiera llevado a cabo.

[19] Este proceso se conoce como "serialización" (*serialization*).

Ejemplos de sistemas de naturaleza estructuralmente reflectiva son Smalltalk–80 [Goldberg83] y Self [Ungar87]. La programación sobre estas plataformas se centra en ir creando los objetos mediante sucesivas modificaciones de su estructura. No existe diferencia entre tiempo de diseño y tiempo de ejecución; al poder acceder a toda la estructura del sistema, el programador se encuentra siempre en tiempo de ejecución modificando dicha estructura.

7.3.1.3 Reflexión computacional

También denominada reflexión de comportamiento (*behavioural reflection*). Se refleja el comportamiento exhibido por un sistema computacional, de forma que éste pueda computarse a sí mismo mediante un mecanismo de conexión causal (punto 7.1) [Maes87]. Un sistema dotado de reflexión computacional puede modificar su propia semántica; el propio comportamiento del sistema podrá cosificarse para su posterior manipulación.

Un ejemplo de abstracción de reflexión computacional es la adición de las "*Proxy Classes*" [Sun99] a la plataforma Java2. Apoyándose en el paquete de introspección (`java.reflect`), se ofrece un mecanismo para modificar el paso de mensajes: puede manipularse dinámicamente la forma en la que un objeto interpreta la recepción de un mensaje. De esta forma la semántica del paso de mensajes puede ser modificada.

Otros ejemplos reducidos de reflexión computacional en la plataforma Java son la posibilidad de sustituir el modo en el que las clases se cargan en memoria y el grado de seguridad de ejecución de la máquina abstracta. Gracias a las clases `java.lang.ClassLoader` y `java.lang.SecurityManager` [Gosling96], se puede modificar en el sistema la semántica de la obtención del código fuente (de este modo se consigue cargar un *applet* de un servidor *Web*) y el modo en el que una aplicación puede acceder a su sistema nativo (el sistema de seguridad frente a virus, en la ejecución de *applets*, conocido como *sandbox* [Orfali98]).

7.3.1.4 Reflexión lingüística

Un lenguaje de programación posee unas especificaciones léxicas [Cueva93], sintácticas [Cueva95] y semánticas [Cueva95b]. Un programa correcto es aquél que cumple las tres especificaciones descritas. La semántica del lenguaje de programación identifica el significado de cualquier programa codificado en dicho lenguaje, es decir, cuál será su comportamiento al ejecutarse.

La reflexión computacional de un sistema nos permite cosificar y reflejar su semántica; la reflexión lingüística [Ortín2001] nos permite modificar cualquier aspecto del lenguaje de programación utilizado: mediante el lenguaje del sistema base se podrá modificar el propio lenguaje (por ejemplo, añadir operadores, construcciones sintácticas o nuevas instrucciones).

Un ejemplo de sistema dotado de reflexión lingüística es OpenJava [Chiba98]. Ampliando el lenguaje de acceso al sistema, Java, se añade una sintaxis y semántica para aplicar patrones de diseño [GOF94], de forma directa, por el lenguaje de programación –amolda el lenguaje a los patrones *Adapter* y *Visitor* [Tatsubori98].

7.3.2 Cuándo se Produce el Reflejo

En función de la clasificación anterior, un sistema determina lo que puede ser cosificado para su posterior manipulación. En este punto clasificaremos los sistemas reflectivos en función del momento en el que se puede llevar a cabo dicha cosificación.

7.3.2.1 Reflexión en tiempo de compilación

El acceso desde el sistema base al metasistema se realiza en el momento en el que el código fuente está siendo compilado, modificándose el proceso de compilación y por lo tanto las características del lenguaje procesado [Golm97].

Tomando Java como lenguaje de programación, OpenJava es una modificación de éste que le otorga capacidad de reflejarse en tiempo de compilación [Chiba98]. En lugar de modificar la máquina virtual de la plataforma para que ésta posea mayor flexibilidad, se rectifica el compilador del lenguaje con una técnica de macros que expande las construcciones del lenguaje. De esta forma, la eficiencia perdida por la flexibilidad del sistema queda latente en tiempo de compilación y no en tiempo de ejecución.

Lo que pueda ser modificado en el sistema está en función de la clasificación 7.3.1. En el ejemplo de OpenJava se pueden modificar todos los elementos; verbigracia, la invocación de métodos, el acceso a los atributos, operadores del lenguaje o sus tipos. El hecho de que el acceso al metasistema se produzca en tiempo de compilación implica que una aplicación tiene que prever su flexibilidad antes de ser ejecutada; una vez que esté corriendo, no podrá accederse a su metasistema de una forma no contemplada en su código fuente.

7.3.2.2 Reflexión en tiempo de ejecución

En este tipo de sistemas, el acceso del metasistema desde el sistema base, su manipulación y el reflejo producido por un mecanismo de conexión causal, se lleva a cabo en tiempo de ejecución [Golm97]. En este caso, una aplicación tendrá la capacidad de ser flexible de forma dinámica, es decir, podrá adaptarse a eventos no previstos[20] cuando esté ejecutándose.

El sistema MetaXa [Golm98], previamente denominado MetaJava, modifica la máquina virtual de la plataforma Java para poder ofrecer reflexión en tiempo de ejecución [Kleinöder96]. A las diferentes primitivas semánticas de la máquina virtual se pueden asociar metaobjetos que deroguen el funcionamiento de dichas primitivas: el metaobjeto define la nueva semántica. Ejemplos clásicos de utilización de esta flexibilidad son la modificación del paso de mensajes para implementar un sistema de depuración, auditoría o restricciones de sistemas en tiempo real [Golm97b].

[20] No previstos en tiempo de compilación –cuando la aplicación esté siendo implementada.

7.3.3 Desde Dónde se Puede Modificar el Sistema

El acceso reflectivo a un sistema se puede llevar a cabo desde distintos procesos. La siguiente clasificación no es excluyente: un sistema puede estar incluido en ambas clasificaciones.

7.3.3.1 Reflexión con acceso interno

Los sistemas con acceso interno (*inward*) a su metasistema permiten modificar una aplicación desde la definición de éste (su codificación) [Foote92]. Esta característica define aquellos sistemas que permiten modificar una aplicación desde sí misma.

La mayoría de los sistemas ofrecen esta posibilidad. En MetaXa [Kleinöder96], sólo la propia aplicación puede modificar su comportamiento.

7.3.3.2 Reflexión con acceso externo

El acceso externo de un sistema (*outward*) permite la modificación de una aplicación mediante otro proceso distinto al que será modificado [Foote92]. En un sistema con esta característica, cualquier proceso puede modificar al resto. La ventaja es la flexibilidad obtenida; el principal inconveniente, la necesidad de establecer un mecanismo de seguridad en el acceso entre procesos (7.4).

El sistema Smalltalk–80 ofrece un mecanismo de reflexión estructural con acceso externo [Mevel87]: el programador puede acceder, y modificar en su estructura, a cualquier aplicación o proceso del sistema.

7.4 Hamiltonians Versus Jeffersonians

Durante el desarrollo del sistema democrático americano, existían dos escuelas de pensamiento distintas: los *Hamiltonians* y los *Jeffersonians*. Los *Jeffersonians* pensaban que cualquier persona debería tener poder para llevar a cabo decisiones como la elección del presidente de los Estados Unidos. Por el contrario, los *Hamiltonians* eran de la opinión de que sólo una elite educada para ello, y no todo el pueblo, debería influir en tales decisiones. Esta diferencia de pensamiento ha sido utilizada para implementar distintas políticas en sistemas computacionales reflectivos [Foote92].

La vertiente Jeffersoniana otorga al programador de un sistema reflectivo la posibilidad de poder modificar cualquier elemento de éste. No restringe el abanico de posibilidades en el acceso al metasistema. El hecho de permitir modificar cualquier característica del sistema, en cualquier momento y por cualquier usuario, puede desembocar en una semántica del sistema prácticamente incomprensible. Un ejemplo de esto, es un programa escrito en C++ con uso intensivo de sobrecarga de operadores (reflexión del lenguaje): el resultado puede ser un código fuente prácticamente ininteligible.

Muchos autores abogan por el establecimiento de un mecanismo de seguridad que restrinja el modo en el una aplicación pueda modificar el sistema [Foote90]. Un ejemplo clásico es establecer una lista de control de acceso en el que sólo los administradores del sistema puedan cambiar la semántica global. Por el contrario, el resto de usuarios podrán modificar únicamente la semántica de sus aplicaciones.

Como mencionamos en el capítulo 2, esta Tesis trata de buscar un entorno que permita una gran flexibilidad en la interacción entre los aspectos y la aplicación base. Como veremos más adelante, identificamos la reflexión como un mecanismo para conseguirlo, necesitando incluso la reflexión externa. No es nuestro objetivo la implementación de un sistema de seguridad encargado de controlar el nivel de reflexión permitido. Por esta razón, nos centraremos en la vertiente Jeffersoniana, teniendo siempre en cuenta los riesgos que ésta puede llegar a suponer.

CAPÍTULO 8
SISTEMAS REFLECTIVOS EXISTENTES

En la sección 7.3 clasificábamos la reflexión en función de distintos criterios. Utilizando dicha clasificación y los conceptos definidos anteriormente en el mencionado capítulo, estableceremos un estudio de un conjunto de sistemas existentes que utilizan de algún modo la reflexión.

Agruparemos los sistemas en función de ciertas características comunes, especificaremos brevemente su funcionamiento, y, en el ámbito de grupo, detallaremos los beneficios aportados y carencias existentes de los sistemas estudiados. Nos centraremos únicamente en sistemas que puedan tener aplicación para resolver el conjunto de requisitos definidos. A modo de conclusión, y apoyándonos en los requisitos definidos en el capítulo 2, analizaremos las principales carencias del conjunto de sistemas vistos.

8.1 Sistemas Dotados de Introspección

Anteriormente, en 7.3.1.1, definíamos introspección como la capacidad de acceder a la representación de un sistema en tiempo de ejecución. Mediante introspección únicamente se puede conocer el sistema, sin permitir la modificación de éste y sin producirse, por lo tanto, su reflejo mediante un sistema de conexión causal. Por esto determinados autores definen introspección como reflexión estructural "de sólo lectura" [Foote90].

La introspección es probablemente el tipo de reflexión más utilizado actualmente en los sistemas comerciales. Se ha desarrollado en áreas como la especificación de componentes, arquitecturas de objetos distribuidos y programación orientada a objetos.

8.1.1 Plataforma Java

Java define para su plataforma [Kramer96] una interfaz de desarrollo de aplicaciones (API, *Application Programming Interface*) que proporciona introspección, denominada *The Java Reflection API* [Sun97d]. Este API permite acceder en tiempo de ejecución a la representación de las clases, interfaces y objetos existentes en la máquina virtual. Las posibilidades que ofrece al programador son:

- Determinar la clase de la que un objeto es instancia.

- Obtener información sobre los modificadores de una clase [Gosling96], sus métodos, campos, constructores y clases base.
- Encontrar las declaraciones de métodos y constantes pertenecientes a un *interface*.
- Crear una instancia de una clase totalmente desconocida en tiempo de compilación.
- Obtener y asignar el valor de un atributo de un objeto en tiempo de ejecución, sin necesidad de conocer éste en tiempo de compilación.
- Invocar un método de un objeto en tiempo de ejecución, aun siendo éste desconocido en tiempo de compilación.
- Crear dinámicamente un *array*[21] y modificar los valores de sus elementos.

Sobre este API de introspección se define el paquete `java.beans`, que ofrece un conjunto de clases e *interfaces* para desarrollar la especificación de componentes software de la plataforma Java: los *JavaBeans* [Sun96].

Cabe preguntarse por qué es necesaria la introspección para el desarrollo de un sistema de componentes. Un componente está constituido por métodos, propiedades y eventos. Sin embargo, una aplicación que vaya a utilizar un conjunto de componentes no puede saber a priori la estructura que éstos van a tener. Para conocer el conjunto de estas tres características, de forma dinámica, se utiliza la introspección: permite acceder a los métodos de instancia públicos y conocer su signatura.

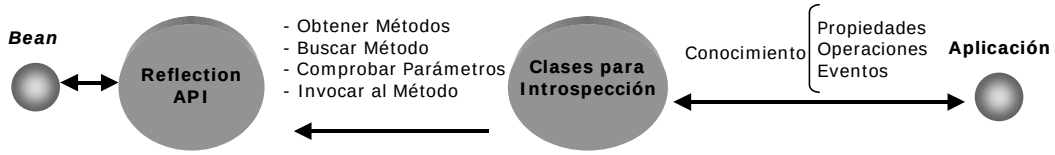

Figura 34: Utilización de introspección en el desarrollo de un sistema de componentes.

El paquete de introspección identifica una forma segura de acceder al API de reflexión, y facilita el conocimiento y acceso de métodos, propiedades y eventos de un componente en tiempo de ejecución. En la Figura 34 se aprecia cómo una aplicación solicita el valor de una propiedad de un *Bean.* Mediante el paquete de introspección se busca el método apropiado y se invoca, devolviendo el valor resultante de su ejecución.

Otro ejemplo de utilización de la introspección en la plataforma Java es la posibilidad de convertir cualquier objeto a una secuencia de bytes[22], para posteriormente poder almacenarlo en disco o transmitirlo a través de una red de computadores. Un objeto en Java que herede del *interface* `java.io.Serializable`, sin necesidad de implementar ningún método, podrá ser convertido a una secuencia de bytes. ¿Cómo es posible conocer el estado del objeto en tiempo de ejecución? Mediante el acceso introspectivo a éste [Eckel2000].

[21] En Java los *array*s son objetos primitivos.

[22] También denominado "serialización" (*serialization*).

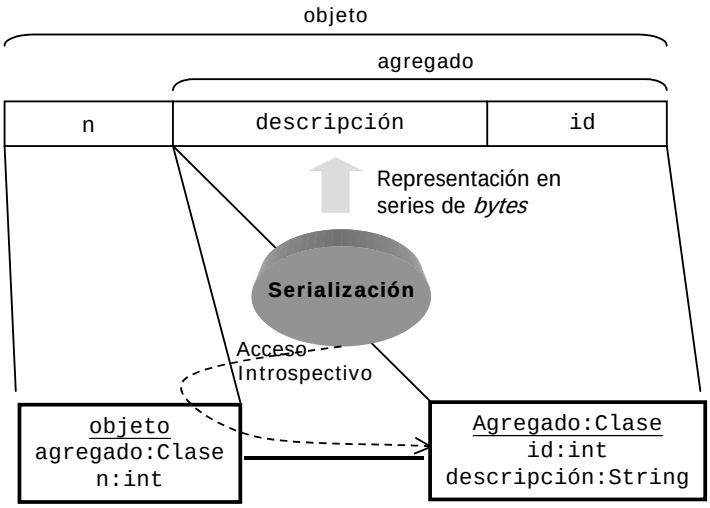

Figura 35: Conversión de un objeto a una secuencia de bytes, mediante un acceso introspectivo.

Mediante el paquete `java.reflect` se va consultando el valor de todos los atributos del objeto y convirtiendo éstos a bytes. Si un atributo de un objeto es a su vez otro objeto, se repite el proceso de forma recursiva. Del mismo modo es posible crear un objeto a raíz de un conjunto de bytes –siguiendo el proceso inverso.

8.1.2 Plataforma *.NET*

La plataforma .NET [MicrosoftNET] es un entorno de desarrollo basado en el estándar *CLI* del *ECMA* [ECMA335], usando una máquina virtual que en su versión comercial se denomina *CLR* [Burton02]. Una descripción más detallada de esta plataforma se puede ver en el punto 6.5.3, por lo que ahora sólo se hará mención a las capacidades introspectivas que ofrecen una serie de paquetes de la librería estándar que incorpora. En general puede afirmarse que las capacidades introspectivas proporcionadas por esta plataforma son muy similares en cuanto a funcionalidad y uso a las que ofrece Java, teniendo básicamente las mismas aplicaciones prácticas que se han visto anteriormente. No obstante, esta plataforma posee alguna característica adicional, como la programación generativa, que será descrita posteriormente, y que la dota de mayor flexibilidad.

Al igual que la plataforma Java, las clases del *API* estándar que proporcionan introspección permiten a cualquier lenguaje incorporado en la plataforma consultar el estado del sistema. Esto es así gracias a que en esta plataforma los lenguajes son interoperables, es decir, se permite que el código desarrollado en un lenguaje X pueda ser usado sin limitaciones desde otro código desarrollado en un lenguaje Y, ocurriendo lo mismo con cualquier clase incorporada en dicho *API* estándar. Las aplicaciones de la introspección en .NET son muy similares a las que podemos tener en Java, de manera que es posible encontrar todos los tipos definidos en un ensamblado o *assembly* (entidad que es usada en esta plataforma para contener tipos) o bien obtener información acerca de cualquier miembro de cualquier clase.

8.1.2.1 Programación Generativa en .Net

.Net cuenta con una capacidad reflectiva adicional a la introspección: La programación generativa. La programación generativa consiste en la generación dinámica de código fuente bajo demanda de forma automatizada, a través de herramientas como clases genéricas, plantillas, aspectos y generadores de código. Mediante su uso, cualquier programador podrá generar nuevos tipos e instancias de los mismos de forma dinámica a medida que el *software* las vaya necesitando, cargándolas en memoria y empleándolas como tipos "normales" definidos en el programa, con el objeto de aumentar su productividad. El conjunto de clases que permiten ese tipo de operaciones se encuentran en el paquete estándar `System.Reflection.Emit`.

Indudablemente, esta capacidad dota a los lenguajes de la plataforma *.NET* de nuevas funcionalidades, ya que permitiría a un usuario involucrarse en la creación de código, generando nuevas clases y reglas de negocio, o bien construyendo herramientas que hagan este trabajo. No obstante, no puede afirmarse que la plataforma *.NET* ofrezca capacidades de reflexión estructural completas empleando este mecanismo, ya que posee ciertas limitaciones que impiden clasificarlo como tal:

- Sólo permite crear tipos nuevos, aunque pueden derivarse de otros ya existentes.
- No se permite la modificación de tipos existentes, es decir, no es posible añadir nuevos atributos y métodos (ni por supuesto modificarlos o eliminarlos) a tipos ya cargados en memoria. Es más, una vez que un tipo nuevo se crea y se habilita para ser usado como un tipo cualquiera, es imposible alterar la composición de ningún miembro.
- El motivo de la limitación anterior es que los tipos nuevos poseen dos "estados": En construcción (momento durante el cual se pueden alterar los miembros de su estructura) y construidos (momento en el que se ya manejan como tipos estándar y no se pueden modificar). No hay pues una verdadera noción de tipo dinámico.
- No se ofrece conocimiento de código existente.

Por tanto, este mecanismo es limitado y de aplicación restringida, no siendo equiparable con un verdadero soporte de reflexión estructural completo, aunque tiene una mayor potencia que la introspección.

8.1.3 Aportaciones y Carencias de los Sistemas Estudiados

En este punto hemos visto cómo es importante la característica introspectiva en el desarrollo de una plataforma. Desde el punto de vista del programador, es necesaria para realizar operaciones seguras respecto al tipo (RTTI de C++) o para almacenar objetos en disco ("Serialización" de la plataforma Java o .Net). Desde el punto de vista de entornos distribuidos, es necesaria para desarrollar aplicaciones flexibles (Java con RMI y .NET con Remoting).

La utilización de la introspección queda patente con numerosos ejemplos prácticos de sistemas actuales reales. En el desarrollo de aplicaciones de comercio electrónico, es común utilizar la transferencia de información independiente de la plataforma y auto descriptiva en formato XML [W3C98]. Mediante la utilización de una librería –

DOM [W3C98b] o SAX [Megginson2000]– es posible conocer los datos en tiempo de ejecución (introspección), eliminando el acoplamiento que se produce entre el cliente y el servidor en muchos sistemas distribuidos. De esta forma, las aplicaciones ganan en flexibilidad pudiéndose adaptar a los futuros cambios.

En el desarrollo de nuestro sistema, la introspección es una característica fundamental, puesto que se necesita conocer la estructura de la aplicación que se va a adaptar y así poder localizar los posibles puntos de enlace de la aplicación.

8.2 Sistemas Dotados de Reflexión Estructural

Fijándonos en la clasificación realizada anteriormente, en 7.3.1.2 se define un sistema dotado de la capacidad de acceder y modificar su estructura como estructuralmente reflectivo. Estudiaremos casos reales de este tipo de sistemas y analizaremos sus ventajas e inconvenientes.

8.2.1 Smalltalk–80

Es uno de los primeros entornos de programación que se han basado en reflexión estructural. El sistema Smalltalk–80 se puede dividir en dos elementos:

- La imagen virtual, que es una colección de objetos que proporcionan funcionalidades de diversos tipos.
- La máquina virtual, que interpreta la imagen virtual y las aplicaciones de usuario.

En un principio se carga la imagen virtual en memoria y la máquina va interpretando el código. Si deseamos conocer la estructura de alguna de las clases existentes, su descripción, el conjunto de los métodos que posee o incluso la implementación de éstos, podemos utilizar una aplicación desarrollada en el sistema denominada *browser* [Mevel87] (Figura 36). El *browser* es una aplicación diseñada en Smalltalk que accede a todos los objetos clase[23] existentes en el sistema, y muestra su estructura y descripción. De la misma forma, se puede acceder a los métodos de éstas. Este es un caso de utilización de la introspección que ofrece el sistema. Gracias a la introspección, se consigue una documentación dinámicamente actualizada de las clases del sistema.

[23] En Smalltalk–80 todas las clases se identifican como un objeto en tiempo de ejecución. Los objetos que identifican clases son instancias de clases derivadas de la clase *Class*.

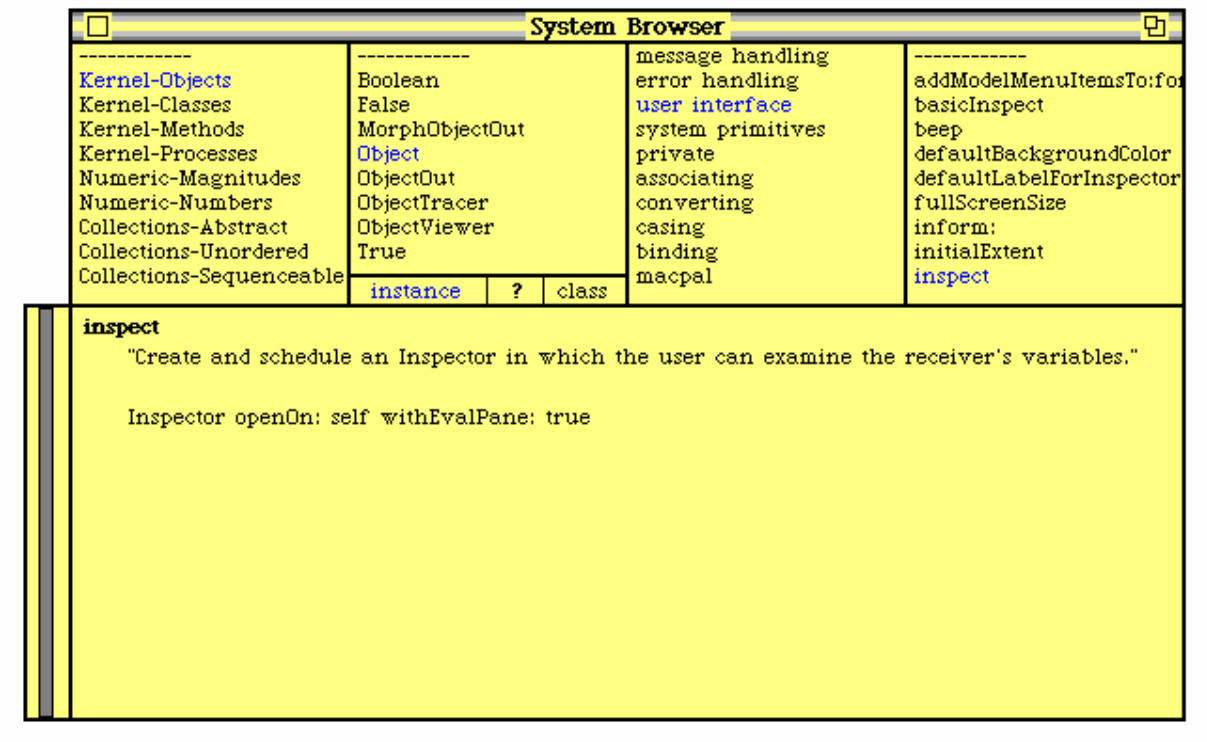

Figura 36: Análisis del método *inspect* del objeto de clase *Object*, con la aplicación *Browser* de Smalltalk–80.

De la misma forma que las clases, haciendo uso de sus características reflectivas, los objetos existentes en tiempo de ejecución pueden ser inspeccionados gracias al mensaje *inspect* [Goldberg89]. Mediante esta aplicación también podremos modificar los valores de los distintos atributos de los objetos.

Vemos en la Figura 37, cómo es posible hacer uso de la reflexión estructural para modificar la estructura de una aplicación en tiempo de ejecución. Una utilidad dada al mensaje *inspect* es la depuración de una aplicación sin necesidad de generar código intruso a la hora de compilar.

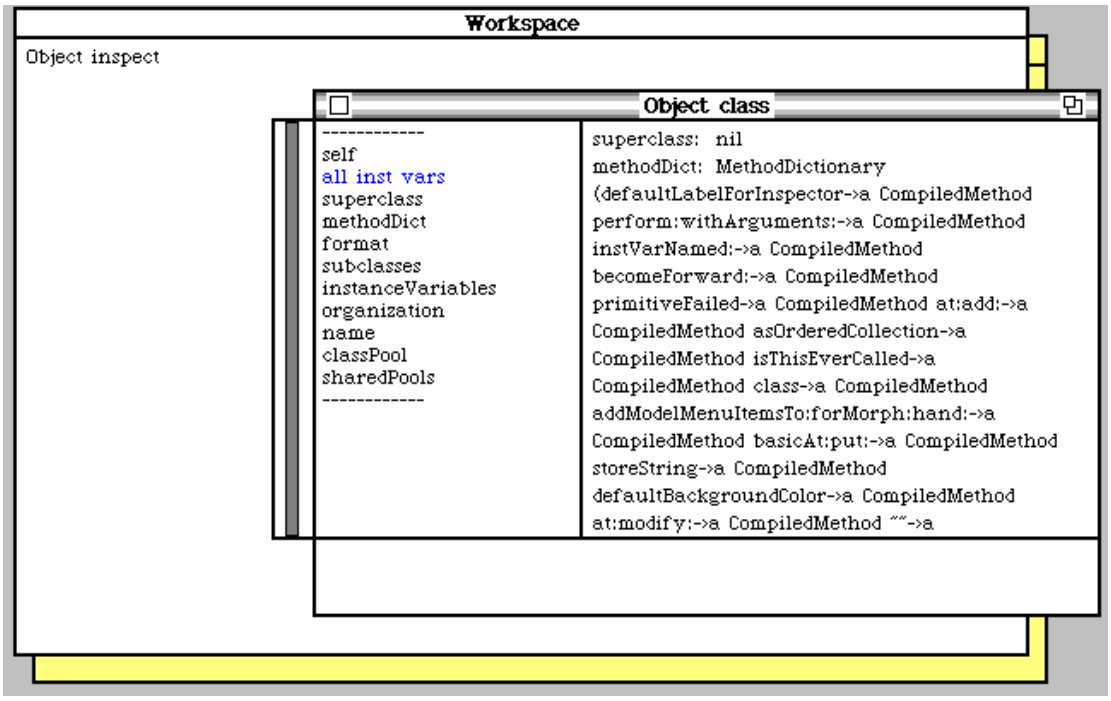

Figura 37: Invocando al método *inspect* del objeto *Object* desde un espacio de trabajo de Smalltalk–80, obtenemos un acceso a las distintas partes de la instancia.

Lo buscado en el diseño de Smalltalk era obtener un sistema fácilmente manejable por personas que no fuesen informáticos. Para ello se identificó el paradigma de la orientación a objetos y la reflexión estructural. Una vez diseñada una aplicación consultando la información necesaria de las clases en el *browser*, se puede depurar ésta accediendo y modificando los estados de los objetos en tiempo de ejecución. Se accede al error de un método de una clase y se modifica su código, todo ello haciendo uso de la reflexión estructural en tiempo de ejecución del sistema.

Hemos visto cómo en Smalltalk–80 se utilizaron de forma básica los conceptos de reflexión estructural e introspección, para obtener un entorno sencillo de manejar y auto documentado.

La semántica del lenguaje Smalltalk viene dada por una serie de primitivas básicas de computación de la máquina virtual, no modificables por el programador; carece por lo tanto de reflexión computacional.

8.2.2 Self, Proyecto Merlin

El sistema Self fue construido como una simplificación del sistema Smalltalk y también estaba constituido por una máquina virtual y un entorno de programación basado en el lenguaje Self [Ungar87]. La reducción principal respecto a Smalltalk se centró en la eliminación del concepto de clase, dejando tan sólo la abstracción del objeto. Este tipo de lenguajes orientados a objetos han sido denominados “basados en prototipos”.

Este sistema, orientado a objetos puro e interpretado, fue utilizado en el estudio de desarrollo de técnicas de optimización propias de los lenguajes orientados a objetos [Chambers91]. Fueron aplicados métodos de compilación continua y compilación adaptable [Hölzle94], métodos que actualmente han sido implantados a plataformas comerciales como Java™ [Sun98].

El proyecto Merlin se creó para hacer que los ordenadores fuesen fácilmente utilizados por los humanos, ofreciendo sistemas de persistencia y distribución implícitos [Assumpcao93]. El lenguaje seleccionado para desarrollar este sistema fue Self.

Para conseguir flexibilidad en el sistema, trataron de ampliar la máquina virtual con un conjunto de primitivas de reflexión (*regions*, *reflectors*, *meta–spaces*) [Cointe92]. La complejidad de ésta creció de tal forma que su portabilidad a distintas plataformas se convirtió en algo inviable [Assumpcao95]. Assumpçao propone en [Assumpcao95] el siguiente conjunto de pasos para construir un sistema portable con capacidades reflectivas:

1. Implementar, sobre cualquier lenguaje de programación, un pequeño intérprete de un subconjunto del lenguaje a interpretar. En su caso un intérprete de "tinySelf". El requisito fundamental es que éste tenga reflexión estructural.
2. Sobre lenguaje (tinySelf), se desarrolla un intérprete del lenguaje buscado (Self), con todas sus características.
3. Implementamos una interfaz de modificación de las operaciones deseadas. La codificación de un intérprete en su propio lenguaje ofrece total flexibilidad: todo su comportamiento se puede modificar –incluso las primitivas computacionales– puesto que tinySelf posee reflexión estructural dinámica. Sólo debemos implementar un protocolo de acceso a la modificación de las operaciones deseadas.

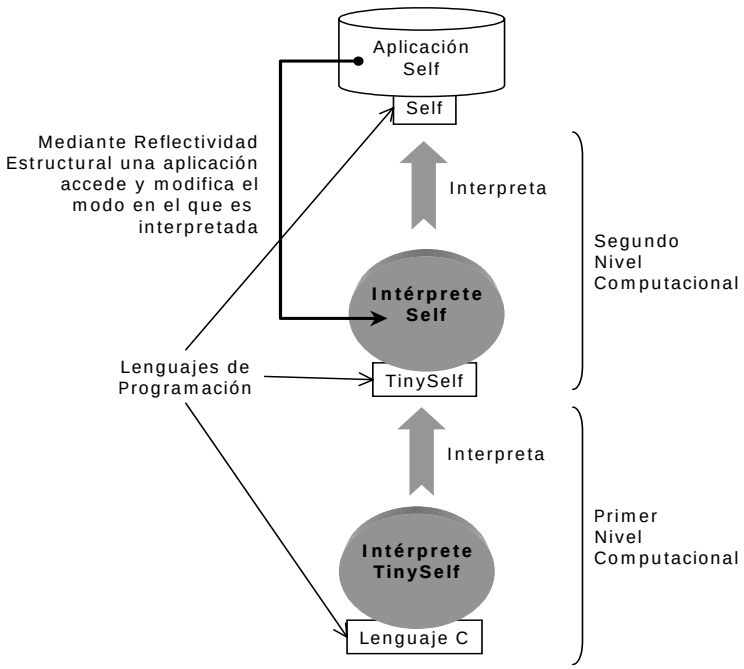

Figura 38: Consiguiendo reflexión mediante la introducción de un nuevo intérprete.

El resultado es un intérprete de Self que permite modificar determinados elementos de su computación, siempre que éstos hayan sido tenidos en cuenta en el desarrollo del intérprete.

El principal problema es la eficiencia del sistema. El desarrollar dos niveles computacionales (intérprete de intérprete) ralentiza la ejecución de la aplicación codificada en Self. Lo que se propone para salvar este obstáculo [Assumpcao95] es implementar un traductor de código tinySelf a otro lenguaje que pueda ser compilado a código nativo. Una vez desarrollado éste, traducir el código propio del intérprete de Self –codificado en el lenguaje tinySelf– a código nativo, reduciendo así un nivel computacional.

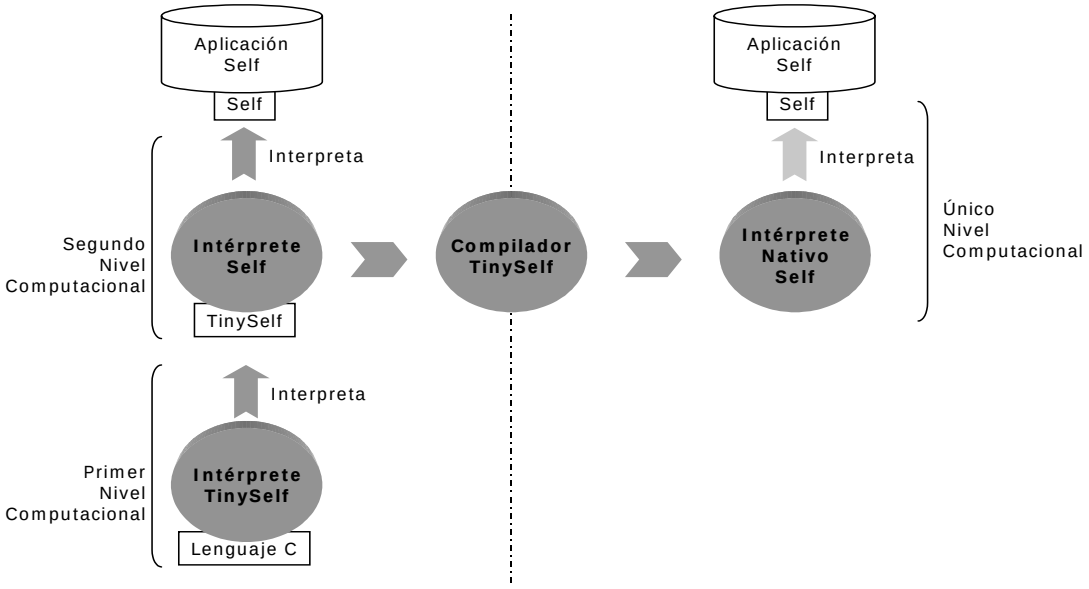

Figura 39: Reducción de un nivel de computación en la torre de intérpretes.

El resultado se muestra en la Figura 39: un único intérprete de Self en lugar de los dos anteriores. Sin embargo, el producto final tendría dos inconvenientes:

1. Una vez que traduzcamos el intérprete, éste no podrá ofrecer la modificación de una característica que no haya sido prevista con anterioridad. Si queremos añadir alguna, deberemos volver al sistema de los dos intérpretes, codificarla, probarla y, cuando no exista ningún error, volver a generar el intérprete nativo.
2. La implementación de un traductor de tinySelf a un lenguaje compilado es una tarea difícil, puesto que el código generado deberá estar dotado de la información necesaria para ofrecer reflexión estructural en tiempo de ejecución. Existen sistemas que implementan estas técnicas como por ejemplo Iguana [Gowing96].
   Al mismo tiempo, la ejecución de un código que ofrece información modificable dinámicamente –sus objetos– ocupa más espacio, y añade una ralentización en sus tiempos de ejecución.

8.2.3 Python

Python es un lenguaje de programación orientado a objetos, portable, interpretado y de propósito general [Rossum01]. Es una plataforma libre de código abierto, perteneciendo el *copyright* de su código a la *Python Software Foundation* (*PSF*) [PSF06]. Su desarrollo comenzó en 1990 en el *CWI* de Ámsterdam. El lenguaje posee módulos, paquetes, clases, excepciones, herencia múltiple, manejo automático de memoria y un

sistema de chequeo de tipos dinámico. Posee también los conceptos de *duck typing* y metaclases. Existen intérpretes del lenguaje en diferentes plataformas como *UNIX*, *Windows*, *Mac,* etc. También podemos destacar múltiples implementaciones de diferentes características de este lenguaje en desarrollo en la actualidad, como *Psyco* [Psyco06] (un compilador *JIT* que convierte código *Python* a código nativo de la máquina virtual de Python especializado de diferentes formas, según los datos manipulados por el programa, para intentar acelerar la ejecución del mismo), *Jython* (un compilador de *Python* a *bytecode Java*) o *IronPython* (que traduce *Python* a *bytecode* de la plataforma *.NET*) entre otras. Otro proyecto interesante en desarrollo es *PyPy* [PyPy06], consistente en reescribir el lenguaje *Python* usando el propio lenguaje con la intención de crear un sistema altamente optimizado con generación estática de código para diferentes plataformas.

Este lenguaje es comúnmente denominado dinámico. Todo código en *Python* se traduce a una representación en forma de *bytecode*, que es ejecutado en la máquina virtual del mismo. Cada vez que un código en *Python* es ejecutado, se genera un archivo de *bytecode* (*pyc*), que permite cargar código dinámicamente de una forma más eficiente, ya que este código no será procesado ni traducido en el momento de la carga. Esto es de especial ayuda cuando se hace uso de un mismo archivo múltiples veces sin que sea sometido a cambios durante la ejecución del programa. Si un archivo de código *Python* es alterado, entonces el archivo de *bytecode* asociado se regenera de nuevo de forma automática.

Una de las características principales de este lenguaje es su extensibilidad. Es posible diseñar y construir fácilmente nuevos módulos que añadan más funciones al mismo, permitiendo diseñar soluciones con este lenguaje adaptadas a múltiples fines.

Otra de las principales ventajas de este lenguaje es su flexibilidad, debido a las operaciones que permite realizar con sus tipos, y cómo pueden ser tratados y modificados (añadir/modificar/eliminar miembros y relaciones ente objetos) en tiempo de ejecución. En concreto, este lenguaje posee reflexión estructural y herencia dinámica, permitiendo pues que cualquier tipo existente pueda ser ampliado o modificado (en su estructura) en tiempo de ejecución y a su vez puedan modificársele sus objetos "padre", cambiando la jerarquía de herencia existente en un momento dado según sea necesario. El modelo de objetos poseído por este lenguaje está basado en el modelo de prototipos (ver 8.2.2 Self, Proyecto Merlin).

Además de reflexión estructural, el intérprete del lenguaje ofrece numerosa información del sistema en tiempo de ejecución mediante introspección. Haciendo uno de la reflexión estructural del lenguaje, se han construido módulos que aumentan las características reflectivas del entorno de programación [Andersen98], ofreciendo un mayor nivel de abstracción basándose en el concepto de metaobjeto [Kiczales91]. Todas las características reflectivas vistas en Python son soportadas gracias a la arquitectura del sistema que se presenta en la figura 5.14 [Jackson05]:

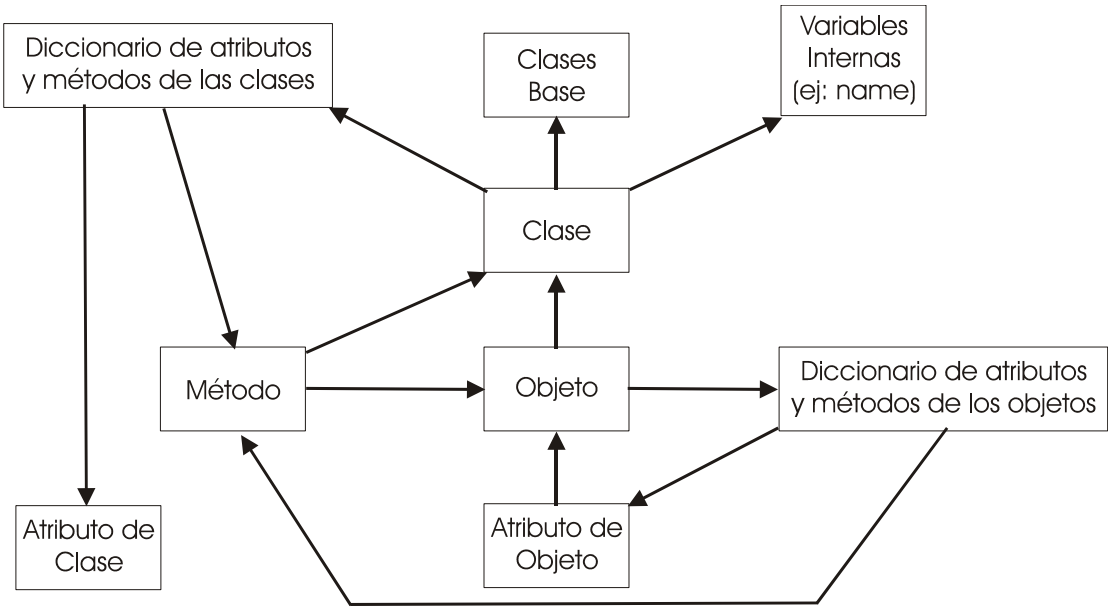

Figura 40: Estructura de clases de la arquitectura del lenguaje *Python*

Describiremos a continuación las características del modelo de objetos y del soporte para reflexión que posee este lenguaje. *Python* es un lenguaje con un sistema de tipos dinámico (soporta el concepto de *duck typing*), por lo que son los valores, y no las variables en sí, los que tienen asociado un tipo. Cuenta con un mecanismo de resolución dinámica de nombres, mediante el cual los nombres de métodos y variables son asociados al contenido al que realmente se refieren en tiempo de ejecución, en lugar de durante la compilación del programa. También soporta el concepto de *closure*.

Sin embargo, una de las características más innovadoras de este lenguaje es su capacidad para soportar reflexión estructural. Aunque *Python* usa efectivamente la palabra reservada `class`, las clases en *Python* no se comportan como las de un lenguaje orientado a objetos basado en clases como *C++*, sino que son objetos que agrupan comportamiento (conocidos como *trait objects*) [Ungar97]. Tanto los objetos como las clases permiten que se les añadan o eliminen atributos o métodos a su definición. Por tanto, *Python* permite la existencia de atributos y métodos de clase y de objeto por separado, pudiendo pues añadir libremente a cada uno de estos elementos cualquier tipo de miembro.

Al conjunto de miembros definidos en el contexto de una clase, cuando ésta se crea o bien asignados posteriormente durante la ejecución, puede accederse a través de la propiedad `__dict__` (que también puede usarse sobre instancias). Podemos también definir mediante el operador de asignación un atributo dentro del contexto de una instancia individual, incluso con el mismo nombre que alguno de su clase, al que entonces ocultará.

Es posible acceder a un atributo de clase desde una instancia de la misma, a través de la propiedad de todo objeto *Python* que permite acceder a su clase (`__class__`).

En *Python* todo tipo no simple es un objeto, incluidas las clases, cuya "clase" se denomina metaclase. El lenguaje soporta introspección de tipos y clases de manera que todo tipo podrá ser leído e inspeccionado, extrayendo sus miembros como un diccionario-

rio. No obstante, ya hemos visto que las capacidades reflectivas de *Python* pueden ir más allá, ya que se permite cambiar la estructura de clases y objetos en tiempo de ejecución (reflexión estructural). En concreto, *Python* ofrece, en tiempo de ejecución, un conjunto de funcionalidades que le permiten [Andersen98]:

- Modificar dinámicamente la clase de un objeto. Todo objeto posee un atributo denominado `__class__` que hace referencia a su clase. La modificación del mismo implica la modificación del tipo del objeto.
- Acceder al árbol de herencia. Toda clase posee un atributo `__bases__` que referencia una colección de sus clases base modificables dinámicamente. *Python* soporta herencia múltiple.
- Acceso a los atributos y métodos de un objeto. Hemos visto como tanto los objetos como las clases poseen un diccionario de sus miembros (`__dict__`) que puede ser consultado y modificado dinámicamente.
- Control de acceso a los atributos. El acceso a los atributos de una clase puede ser modificado con la definición de los métodos `__getattr__` y `__setattr__`.
- Modificación de la estructura de clases y objetos: *Python* permite modificar la estructura de clases y objetos (añadir / modificar / eliminar atributos y métodos) simplemente mediante la asignación al objeto de un valor y un nombre para un atributo, o sólo el nombre para un método. Esto hace que *Python* implemente características de reflexión estructural de una forma muy sencilla.

Por tanto, *Python* permite la herencia dinámica y el cambio de tipo. Sobre la reflexión estructural del lenguaje se han construido módulos que aumentan las características reflectivas del entorno de programación [Andersen98], ofreciendo un mayor nivel de abstracción basándose en el concepto de metaobjeto [Kiczales91].

Como hemos visto las posibilidades que ofrece la reflectividad estructural implementada en este lenguaje, permiten alterar la estructura de clases y objetos (añadiendo y eliminado miembros), cambiar las relaciones existentes entre clases (herencia), etc.

8.2.4 Ruby

Ruby [RubyCentral06b] es un lenguaje dinámico, orientado a objetos puro, implementado mediante un intérprete (aunque existen versiones que se compilan a una máquina virtual como *Gardens point Ruby.Net*, versión beta [Gardens06]) y con soporte para reflexión estructural. Principalmente está enfocado a la sencillez y agilidad de desarrollo. Combina una sintaxis basada en *Ada* y *Perl* con capacidades para trabajar con objetos similares a las poseídas por *Smalltalk*. Actualmente existen implementaciones para *Windows*, *Linux*, *Mac OS X* y la plataforma *.NET* [Gardens06]. Es una plataforma libre, de código abierto con licencia *GPL* (*General Public License*). Recientemente, la popularidad de este lenguaje se ha visto incrementada gracias al *framework Ruby on Rails* [Thomas05].

Cualquier tipo de dato en *Ruby* es un objeto y toda función es un método. Las variables se construyen como referencias a objetos y no son los propios objetos en sí.

Soporta características estándar de lenguajes orientados a objetos (no soporta la herencia múltiple) y también permite la programación procedural (aunque de forma simulada, ya que todo procedimiento pertenecerá a un objeto llamado "*Object*", padre de todos los objetos).

En lo referente a sus capacidades reflectivas, *Ruby* soporta introspección [Ruby-Central06b] y reflexión estructural (aunque de forma no completa, puesto que, aunque se permite modificar la estructura y comportamiento de clases y objetos, no se permite modificar la clase padre de un objeto dado y por tanto no posee herencia dinámica). Este lenguaje también posee algunos elementos que permiten simular funcionalidades de reflexión computacional (de forma limitada).

El modelo computacional de *Ruby* está basado en el modelo de prototipos y utiliza los conceptos de metaclases y de *duck typing*.

La siguiente versión en desarrollo, *Ruby2* [Davis06], parece que no será compatible con la versión anterior y su rendimiento va a tratar de ser mejorado mediante la utilización de una máquina virtual (*Rite* [Davis06]).

A continuación, profundizaremos en la descripción de este lenguaje haciendo especial hincapié en sus capacidades de reflexión. El modelo de objetos empleado por este lenguaje está basado en el modelo de prototipos. Aunque existe la palabra reservada `class`, ésta funciona de manera distinta respecto al modelo de orientación a objetos basado en clases de lenguajes como *C++* [Tldp06]. *Ruby* agrupa el comportamiento compartido por una serie de objetos en una entidad que denomina clase, que contendrá por tanto todos los métodos comunes a los mismos (es decir, las clases funcionan como los *trait objects*).

Sólo a través de los objetos podremos acceder al comportamiento compartido definido en su clase. Como en este lenguaje las clases sólo contendrán el comportamiento compartido por los objetos, son estos últimos los encargados de guardar su estado, a través de las denominadas variables de instancia. Una variable de instancia [Tldp06] tiene un nombre que comienza con *@*, y su ámbito está limitado al objeto al que referencia *self* (equivalente a *this* en *C++*, refiriéndose la propia instancia). En *Ruby* las variables de instancia nunca son públicas y tienen el valor `nil` antes de que se inicialicen. Las variables de instancia en *Ruby* no necesitan declararse, proporcionándose de esta forma una flexibilidad mayor a los objetos, ya que cada una de ellas se añade dinámicamente al objeto la primera vez que se la referencia.

Ruby, aun manteniendo una nomenclatura similar al modelo de orientación a objetos basado en clases tradicional, emplea conceptos del modelo de prototipos. De esta forma, las clases contendrán únicamente el comportamiento compartido de las instancias (serán un *trait object*, como se vio anteriormente), mientras que el estado de cada instancia estará contenido en cada una de ellas individualmente. La modificación de la estructura de una instancia no hará caer al modelo en un estado inconsistente, puesto que la clase no contiene ningún dato acerca del estado y cualquier método añadido a una instancia será propiedad de la misma, sin alterar la estructura del resto. Un objeto puede así redefinir el comportamiento de un método de su clase, haciendo que ese objeto tenga un comportamiento específico, que sólo tendrá efecto para dicho objeto (método *singleton*), sin necesidad de crear una nueva clase para el mismo.

El modelo de herencia de *Ruby* está basado en delegación, aunque no permite la herencia dinámica (no se puede cambiar la clase de un objeto ni el árbol de herencia de las mismas). Tampoco se soporta la herencia múltiple y todo objeto posee como ancestro `Object`. La carencia de herencia múltiple se suple con la introducción del concepto de módulo y de *mixin* (incluido dentro de los módulos).

En lo referente a sus capacidades reflectivas, *Ruby* posee además soporte para introspección dinámica [RubyCentral06b]. Usando este mecanismo, *Ruby* permite examinar dinámicamente todos los objetos de un programa, su jerarquía de clases, atributos y métodos (y la información asociada a ellos).

Ruby ofrecer reflexión estructural, permitiendo añadir nuevos métodos a clases y objetos. También podemos eliminar métodos con primitivas como `remove_method`. En el caso de atributos, podemos eliminarlos mediante `remove_instance_variable` y `remove_class_variable`, según queramos eliminarlos de instancias o clases.

Ruby permite además que el usuario pueda capturar ciertos eventos del sistema mediante el empleo de *hooks*. Estos *hooks* permiten interceptar eventos como la creación de un objeto, lo que hace que podamos redefinir el significado de ciertas primitivas, cambiando efectivamente la semántica de las mismas y simulando de esta forma algunas capacidades de reflexión computacional.

Existen formas adicionales de capturar estos eventos que ocurren dentro de un programa dentro de su ejecución. Mediante el empleo de *callback*s podemos capturar ciertos eventos de forma controlada. Los eventos a capturar son (donde :: indica un método de clase o módulo y # indica un método de instancia):

Evento	**Método *Callback***
Añadir un método de instancia	Module#method_added
Añadir un método singleton	Kernel::singleton_method_added
Crear una subclase para una clase	Class#inherited
Incorporar un módulo a una clase	Module#extend_object

En tiempo de ejecución estos métodos serán llamados por el sistema cuando ocurra el evento especificado. Por defecto estos métodos no ejecutan operación alguna, pero si se define código para los mismos automáticamente pasará a ser ejecutado.

A modo de conclusión, cabe destacar cómo el lenguaje *Ruby* permite al programador contar con un elevado grado de flexibilidad, abarcando tanto introspección como reflexión estructural (limitada, al no tener herencia dinámica) y ciertos elementos de reflexión computacional (de forma simulada con los *hooks*), sobre un modelo computacional basado en prototipos. También soporta la generación dinámica de código que le permite ampliar y modificar las aplicaciones en tiempo de ejecución con código que se genere según el estado de la aplicación, respondiendo dinámicamente a posibles cambios que pudiesen ocurrir.

8.2.5 Aportaciones y Carencias de los Sistemas Estudiados

Los sistemas dotados de reflectividad estructural ofrecen una mayor flexibilidad que los que únicamente ofrecen introspección. En Java, una aplicación puede modificar su flujo en función de su estado –conociéndolo mediante introspección– ; en Smalltalk, además puede modificar su estructura dinámicamente –gracias a la reflectividad estructural. Si analizamos la dicotomía anterior desde el punto de vista de la seguridad, para una plataforma comercial sería peligroso ofrecer reflectividad estructural, puesto que cualquier aplicación podría acceder y modificar la estructura del sistema. Esta discusión se detalla en el punto 7.4.

Los sistemas estudiados ofrecen conocimiento automático de su estructura y su modificación. Una de las características analizadas tanto en el lenguaje Python como en Ruby, es la modificación de la semántica del lenguaje para un objeto o una clase. En estos ejemplos, mediante el uso de la reflexión estructural, es posible modificar el mecanismo de interpretación de la recepción de un mensaje para una clase.

Aunque es cierto que el ofrecer reflexión estructural tiene un sobrecoste en el rendimiento, existen lenguajes que la ofrecen y que gozan de amplia aceptación. De hecho, en el momento de escribir esta Tesis (enero 2007) el lenguaje Python ocupa el octavo lugar en el ranking de lenguajes de programación más usados [TIOBE], con un fuerte incremento en su presencia, y Ruby ocupa el lugar décimo en el mismo ranking habiendo experimentado un gran ascenso durante el último año desde el vigésimo puesto (de hecho ambos lenguajes fueron candidatos a obtener el título honorífico de lenguaje del año 2006 del mismo ranking, obteniéndolo Ruby). Por todo ello parece claro que la flexibilidad que aporta la reflexión estructural es atractiva para los usuarios a pesar el coste en rendimiento que supone soportarla.

8.3 Sistemas Dotados de Reflexión Computacional

8.3.1 Reflexión Computacional Basada en una Máquina Abstracta

A modo de ejemplo vamos a mencionar la plataforma nitrO, creada por Francisco Ortín Soler [Ortin2001], que ofrece un sistema reflectivo no restrictivo. La máquina abstracta implementa una serie de primitivas básicas de reflexión, que sirven como base computacional para el resto del sistema.

Mediante el empleo de las características reflectivas con las que cuenta la plataforma nitrO, se construyó un interprete genérico que permite procesar lenguajes de forma genérica, por lo que cualquier aplicación podrá interactuar con otra usando el modelo computacional de objetos ofrecido por la máquina abstracta, independientemente del lenguaje con la que haya sido creada.

Tanto la estructura de las aplicaciones en tiempo de ejecución como la especificación léxica, sintáctica y semántica de cualquier lenguaje de programación que se use son parámetros configurables dinámicamente, tanto por la propia aplicación como por cualquier otro programa.

En la Figura 43 se puede ver la arquitectura de nitrO. La plataforma consigue una neutralidad respecto al lenguaje, ya que se puede usar cualquier lenguaje y cualquier aplicación es capaz de adaptar a cualquier otra aplicación sin importar el lenguaje

en que estén implementadas. Esto se consigue mediante la existencia de un intérprete genérico que funciona con cualquier lenguaje: sus entradas son la especificación del lenguaje y la aplicación del usuario implementada en ese lenguaje. En tiempo de ejecución una aplicación puede acceder y modificar la especificación de su lenguaje (o el de otra aplicación), así como su propio código, siendo estas modificaciones tenidas en cuenta de forma inmediata por el intérprete.

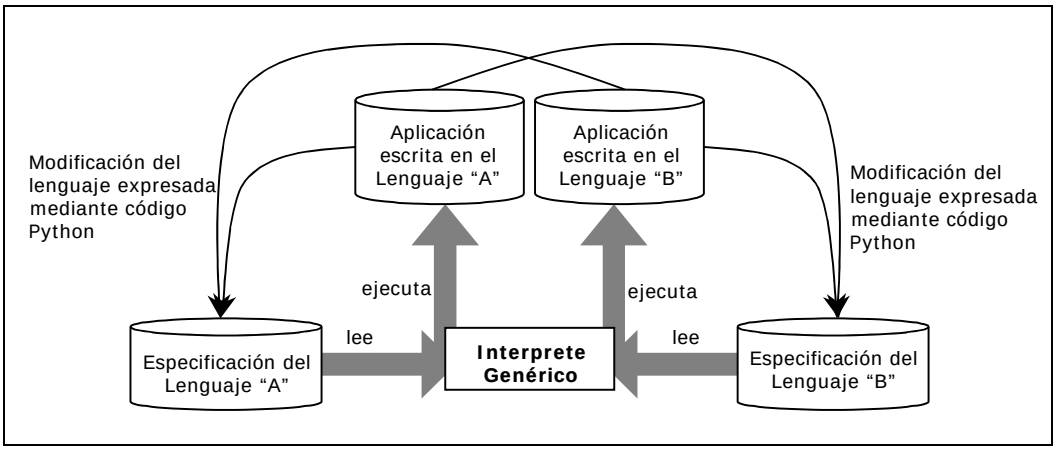

Figura 41: Arquitectura del sistema *nitrO*

El nivel de reflexión no tiene pues ninguna restricción respecto a las características computacionales a configurar, ni respecto al modo de expresar su adaptación, siendo además toda la flexibilidad dinámica: Las aplicaciones no tienen que finalizarse para adaptarlas a nuevos requisitos que surjan en tiempo de desarrollo u otros motivos que aconsejen cambios en el diseño de la aplicación ejecutada.

Aunque las características de esta plataforma son válidas para desarrollar nuestro sistema, presenta una serie de inconvenientes que desaconsejan su utilización. En primer lugar es una plataforma de investigación, que no cuenta con una amplia difusión (que era un requisito del sistema), y que en los prototipos implementados hasta el momento no ofrece buen rendimiento en ejecución. El hecho de tener que suministrar la definición del lenguaje utilizado al interprete puede verse también como un inconveniente.

8.3.2 Reflexión Computacional Basada en Meta–Object Protocols

Los sistemas reflectivos que permiten modificar parte de su semántica en tiempo de ejecución son comúnmente implementados utilizando el concepto de protocolo de metaobjeto (*Meta–Object Protocol*, MOP). Estos MOPs definen un modo de acceso (protocolo) del sistema base al metasistema, permitiendo modificar parte de su propio comportamiento dinámicamente.

En este punto analizaremos brevemente un conjunto de sistemas que utilizan MOPs para modificar su semántica dinámicamente, para posteriormente hacer una síntesis global de sus aportaciones y carencias.

8.3.2.1 Closette

Closette [Kiczales91] es un subconjunto del lenguaje de programación CLOS [Steele90]. Fue creado para implementar un MOP del lenguaje CLOS, permitiendo modificar dinámicamente aspectos semánticos y estructurales de este lenguaje. La implementación se basó en desarrollar un intérprete de este subconjunto de CLOS sobre el propio lenguaje CLOS, capaz de ofrecer un protocolo de acceso a su metasistema.

Existen dos niveles computacionales: el nivel del intérprete de CLOS (metasistema), y el del intérprete de Closette (sistema base) desarrollado sobre el primero. El acceso del sistema base al metasistema se realiza mediante un sistema de macros; el código Closette se expande a código CLOS que, al ser evaluado, puede acceder a su metasistema. La definición de la interfaz de estas macros constituye el MOP del sistema.

El diseño de este MOP para el lenguaje CLOS se centra en el concepto de metaobjeto. Un metaobjeto es cualquier abstracción, estructural o computacional, del metasistema susceptible de ser modificada por su sistema base. Un metaobjeto no es necesariamente una representación de un objeto en su sistema base; puede representar una clase o la semántica de la invocación a un método. El modo en el que se puede llevar a cabo la modificación de los metaobjetos es definido por el MOP.

La implementación del sistema fue llevada a cabo en tres capas, mostradas en la Figura 42:

- La capa de macros; define la forma en la que se va a interpretar el subconjunto de CLOS definido, estableciéndolo mediante traducciones a código CLOS. En el caso de que no existiese un MOP, esta traducción sería la identidad; el código Closette se traduciría a código CLOS sin ningún cambio.
- La capa de "pegamento" (*glue*). Son un conjunto de funciones desarrolladas en CLOS que facilitan el acceso a los objetos del metasistema, para así facilitar la implementación de la traducción de las macros. Como ejemplo, podemos citar la función `find-class` que obtiene el metaobjeto representativo de una clase, cuyo identificador es pasado como parámetro.
- La capa de nivel inferior. Es la representación en CLOS (metasistema) de la estructura y comportamiento del sistema base. Todas aquellas partes del sistema que deseen ser reflectivas, deberán ser implementadas como metaobjetos.

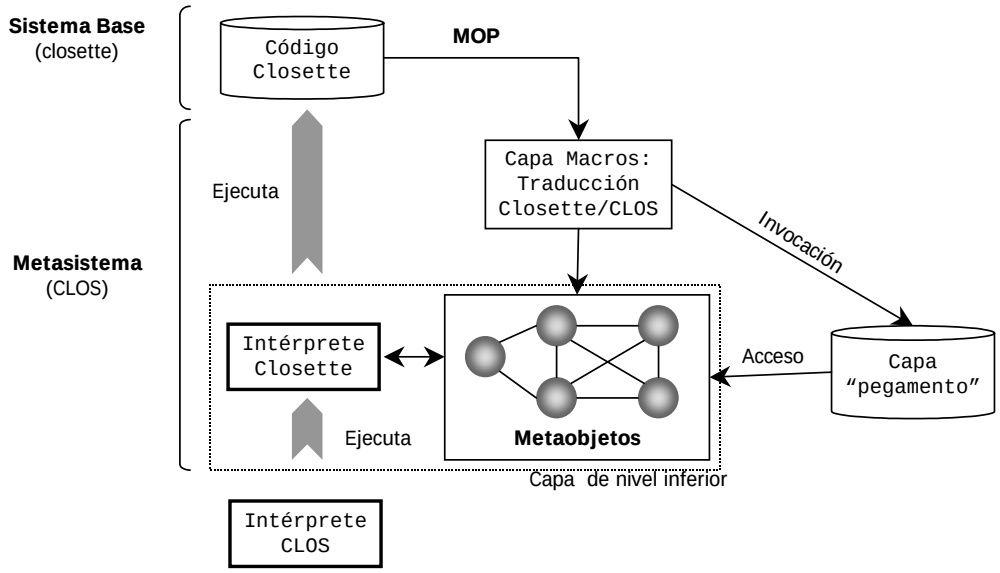

Figura 42: Arquitectura del MOP desarrollado para el lenguaje CLOS.

8.3.2.2 MetaXa[24]

MetaXa es un sistema basado en el lenguaje y plataforma Java que añade a éste características reflectivas en tiempo de ejecución, permitiendo modificar parte de la semántica de la implementación de la máquina virtual [Golm97]. El diseño de la plataforma está orientado a dar soporte a la demanda de creación de aplicaciones flexibles, que puedan adaptarse a requerimientos dinámicos como distribución, seguridad, persistencia, tolerancia a fallos o sincronización de tareas [Kleinöder96].

El modo en el que se deben desarrollar aplicaciones en MetaXa se centra en el concepto de metaprogramación (*meta–programming)* [Maes87]: separación del código funcional del código no funcional. La parte funcional de una aplicación se centra en el modelado del dominio de la aplicación (nivel base), mientras que el código no funcional formaliza la supervisión o modelado de determinados aspectos propios código funcional (metasistema). MetaXa permite separar estos dos niveles de código fuente y establecer entre ellos un mecanismo de conexión causal en tiempo de ejecución.

El sistema de computación de MetaXa se apoya sobre la implementación de objetos funcionales y metaobjetos conectados a los primeros –a un metaobjeto se le puede conectar objetos, referencias y clases; de forma genérica utilizaremos el término objeto. Cuando un metaobjeto está conectado a un objeto sobre el que sucede una acción, el sistema provoca un evento en el metaobjeto indicándole la operación solicitada en el sistema base. La implementación del metasistema permite modificar la semántica de la acción provocadora del evento. La computación del sistema base es suspendida de forma síncrona, hasta que el metasistema finalice la interpretación del evento capturado.

[24] Anteriormente conocido como MetaJava.

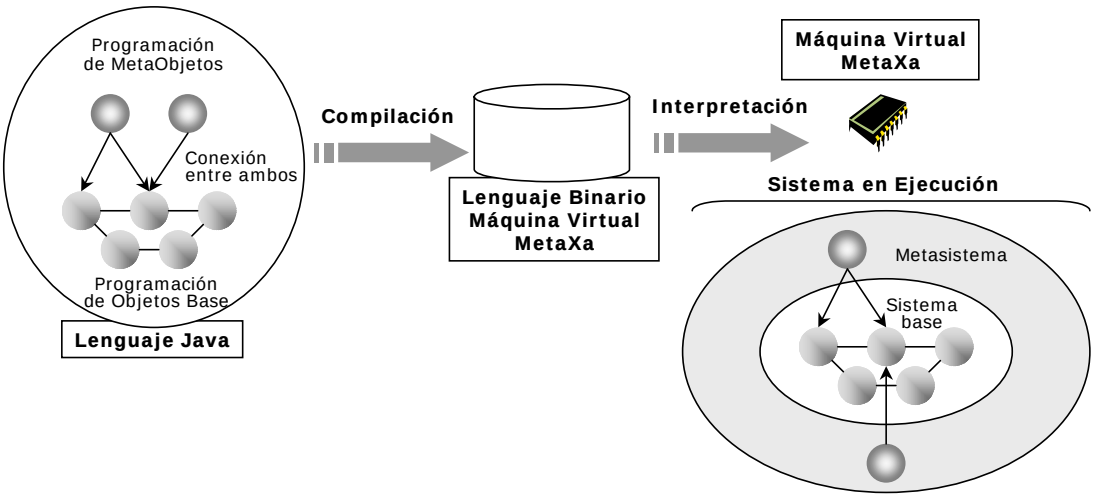

Figura 43: Fases en la creación de una aplicación en MetaXa.

Si un metaobjeto recibe el evento *method–enter*, la acción por omisión será ejecutar el método apropiado. Sin embargo, el metaobjeto puede sobrescribir el método `eventMethodEnter` para modificar el modo en el que se interpreta la recepción de un mensaje.

En MetaXa, para hacer el sistema lo más eficiente posible, inicialmente ningún objeto está conectado a un metaobjeto. En tiempo de ejecución se producen las conexiones apropiadas para modificar los comportamientos solicitados por el programador.

La implementación de la plataforma reflectiva de MetaXa toma la máquina virtual de Java™ [Sun95] y añade sus nuevas características reflectivas mediante la implementación de métodos nativos residentes en una librería de enlace dinámico [Sun97c].

Uno de los principales inconvenientes del diseño abordado es el de la modificación del comportamiento individual de un objeto. Al residir el comportamiento de todo objeto en su clase, ¿qué sucede si deseamos modificar la semántica de una sola instancia de dicha clase? MetaXa crea una nueva clase para el objeto denominada clase sombra (*Shadow Class)* con las siguientes características [Golm97c]:

- Una clase y su clase sombra asociada son iguales para el nivel base.
- La clase sombra difiere de la original en las modificaciones realizadas en el metasistema.
- Los métodos y atributos de clase (`static`) son compartidos por ambas clases.

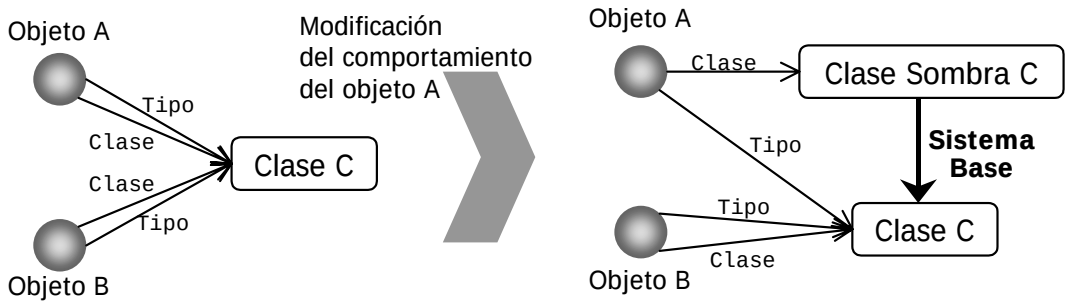

Figura 44: Creación dinámica de una clase sombra en MetaXa para modificar el comportamiento de una instancia de una clase.

Hay que solventar un conjunto de problemas para mantener coherente el sistema en la utilización de las clases sombra [Golm97c]:

1. Consistencia de atributos. La modificación de los atributos de una clase ha de mantenerse coherente con su representación sombra.
2. Identidad de clase. Hay que tener en cuenta que el tipo de una clase y el de su clase sombra han de ser el mismo, aunque tengan distinta identidad.
3. Objetos de la clase. Cuando el sistema utilice el objeto representante de una clase (en Java una clase genera un objeto en tiempo de ejecución) han de tratarse paralelamente el de la clase original y el de su sombra. Un ejemplo puede ser las operaciones `monitorenter` y `monitorexit` de la máquina virtual [Venners98], que tratan los objetos clase como monitores de sincronización; ha de ser el mismo monitor el de la clase sombra que el de la real.
4. Recolector de basura. Las clases sombra deberá limpiarse cuando el objeto no tenga asociado un metaobjeto.
5. Consistencia de código. Tendrá que ser mantenida cuando se produzca la creación de una clase sombra a la hora de estar ejecutándose un método de la clase original.
6. Consumo de memoria. La duplicidad de una clase produce elevados consumos de memoria; deberán idearse técnicas para reducir estos consumos.
7. Herencia. La creación de una clase sombra, cuando la clase inicial es derivada de otra, obliga a la producción de otra clase sombra de la clase base original. Este proceso ha de expandirse de forma recursiva.

La complejidad surgida por el concepto de clase en un sistema reflectivo hace afirmar a los propios autores del sistema, cómo los lenguajes basados en prototipos como Self [Ungar87] o Moostrap solucionan estos problemas de una forma más coherente [Golm97c].

8.3.2.3 Iguana

La mayoría de los sistemas reflectivos, adaptables en tiempo de ejecución y basados en MOPs, son desarrollados mediante la interpretación de código; la ejecución de código nativo no suele ser común en este tipo de entornos. La principal razón es que los intérpretes tienen que construir toda la información propia de la estructura y la semántica de la aplicación a la hora de ejecutar ésta. Si un entorno reflectivo trata de modificar la estructura o comportamiento de un sistema, será más sencillo ofrecer esta información si la ejecución de la aplicación es interpretada.

En el caso de los compiladores, la información relativa al sistema es creada en tiempo de compilación –y generalmente almacenada en la tabla de símbolos [Cueva92b]– para llevar a cabo todo tipo de comprobación de tipos [Cueva95b] y generación de código [Aho90]; una vez compilada la aplicación, dicha información deja de existir. En el caso de un depurador (*debugger*), parte de la información es mantenida en tiempo de ejecución para poder conocer el estado de computación (introspección) y permitir modificar su estructura (reflexión estructural dinámica). El precio a pagar en este caso es un aumento de tamaño de la aplicación, y una ralentización de su ejecución.

El sistema Iguana ofrece reflexión computacional en tiempo de ejecución basada en un MOP, compilando código C++ a la plataforma nativa destino [Gowing96]. En la generación de código, de forma contraria a un depurador, Iguana no genera información de toda la estructura y comportamiento del sistema. Por omisión, compila el código origen C++ a la plataforma destino sin ningún tipo de información dinámica. El programador ha de especificar qué parte del sistema desea que sea adaptable en tiempo de ejecución, de modo que el sistema generará el código oportuno para que sea reflectivo.

Iguana define dos conceptos para especificar el grado de adaptabilidad de una aplicación:

- Categorías de cosificación (*Reification Categories*). Indican al compilador dónde debe producirse la cosificación del sistema. Son elementos susceptibles de ser adaptados en Iguana; ejemplos son clases, métodos, objetos, creación y destrucción de objetos, invocación a métodos o recepción de mensajes, entre otros.
- Definición múltiple de MOPs (*Multiple fine–grained MOPs*). El programador ha de definir la forma en la que el sistema base va a acceder a su información dinámica, es decir se ha de especificar el MOP de acceso al metasistema.

La implementación de Iguana está basada en el desarrollo de un preprocesador que lee el código fuente Iguana –una extensión del C++– y traduce toda la información especifica del MOP, a código C++ con información dinámica adicional adaptable en tiempo de ejecución (el código C++ no reflectivo no sufre proceso de traducción alguno). Una vez que la fase de preproceso haya sido completada, Iguana invocará a un compilador de C++ para generar la aplicación final nativa, adaptable dinámicamente.

8.3.2.4 Cognac

Cognac [Murata94] es un sistema orientado a objetos basado en clases, cuya intención es proporcionar un entorno de programación de sistemas operativos orientados a objetos como Apertos [Yokote92]. El lenguaje de programación de Cognac es intencionalmente similar a Smalltalk–80 [Goldberg89]; para aumentar su eficiencia, se le ha añadido comprobación estática de tipos [Cardelli97].

Los principales objetivos del sistema son:

- Uniformidad y simplicidad. Para el programador sólo debe haber un tipo de objeto concurrente, sin diferenciar ejecución síncrona de asíncrona.
- Eficiencia. Necesaria para desarrollar un sistema operativo.

- Seguridad. Deberá tratarse de minimizar el número de errores en tiempo de ejecución; de ahí la introducción de tipos estáticos al lenguaje.
- Migración. En el sistema los objetos deberán poder moverse de una plataforma física a otra, para seleccionar el entorno de ejecución que más les convenga.
- Metaprogramación. El sistema podrá programarse separando las distintas incumbencias y aspectos de las aplicaciones, diferenciando entre el código funcional del no funcional.

La arquitectura del sistema está compuesta de cinco elementos:

1. El *front–end* del compilador. El código fuente Cognac es traducido a un código intermedio independiente de la plataforma destino seleccionada. El compilador selecciona la información propia de las clases y la almacena, para su posterior uso, en el sistema de clases (quinto elemento).
2. El *back–end* del compilador. Esta parte del compilador toma el código intermedio y lo traduce a código binario propio de la plataforma física utilizada. Crea un conjunto de funciones traducidas de cada una de las rutinas del lenguaje de alto nivel.
3. Librería de soporte dinámico (*Run–time Support Library*). Es el motor principal de ejecución. Envía una petición al sistema de clases para conocer el método apropiado del objeto implícito a invocar; una vez identificado éste en el código nativo, carga la función apropiada y la ejecuta.
4. Intérprete. Aplicación nativa capaz de ejecutar el código intermedio de la aplicación. Será utilizado cuando el sistema requiera reflejarse dinámicamente. La ejecución del código en este modo se ralentiza frente a la ejecución nativa.
5. Sistema de clases. Información dinámica relativa a las clases y métodos del sistema.

El proceso de reflexión dinámica y los papeles de las distintas partes del sistema se muestran en la Figura 45. La ejecución del sistema es controlada por la librería de soporte dinámico, que lee la información relativa al mensaje solicitado, busca éste entre el código nativo y lo ejecuta. Cuando se utiliza un metaobjeto dinámicamente, el motor de ejecución pasa a ser el intérprete, que ejecuta el código intermedio de dicho metaobjeto. El comportamiento del sistema es derogado por la interpretación del metaobjeto creado; éste dicta la nueva semántica del sistema.

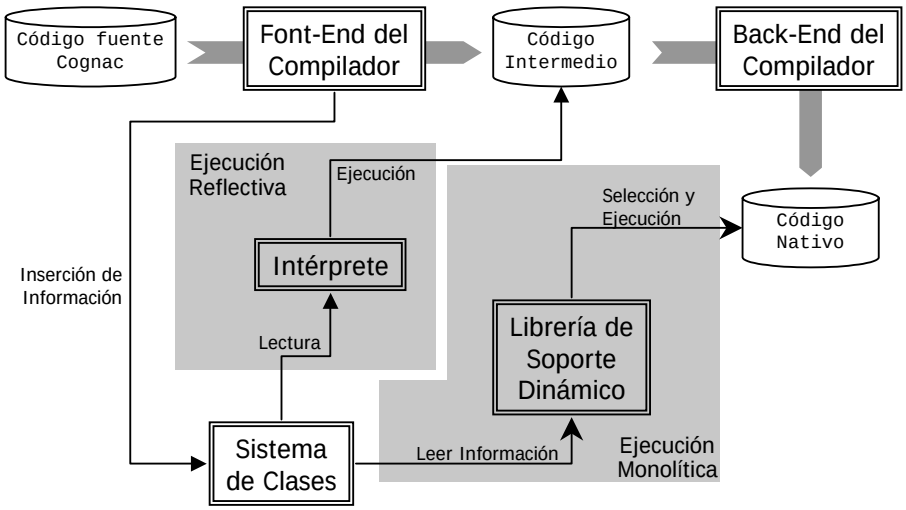

Figura 45: Arquitectura y ejecución de una aplicación en Cognac.

8.3.2.5 Aportaciones y carencias de los sistemas estudiados

En la totalidad de los MOPs estudiados, se permite la modificación dinámica de parte del comportamiento del mismo.

El concepto de MOP establece un modo de acceso del sistema base al metasistema, identifica el comportamiento que puede ser modificado. El establecimiento de este protocolo previamente a la ejecución de la aplicación supone una restricción a priori de la semántica que podrá ser modificada dinámicamente. Los sistemas basados en MOPs otorgan una flexibilidad dinámica de su comportamiento.

La mayor carencia de los sistemas reflectivos es su eficiencia en ejecución. La utilización de intérpretes es más común, pero las aplicaciones finales poseen tiempos de ejecución más elevados que si hubieren sido compiladas a código nativo. A raíz de analizar los sistemas estudiados podemos decir:

1. La implementación de un MOP en un sistema interpretado (por ejemplo, MetaXa) es más sencilla y menos eficiente que el desarrollo de un traductor a un lenguaje compilable, como por ejemplo Iguana. En este caso, el sistema debe poseer información dinámica adicional para poder acceder y modificar ésta en tiempo de ejecución.
   Un ejemplo de la complejidad mencionada es la posibilidad de conocer dinámicamente el tipo de un objeto en C++ (RTTI, *RunTime Type Information*) [Stroustrup98]. Esta característica supone modificar la generación de código, para que todo objeto posea la información propia de su tipo – generalmente la ejecución de una aplicación nativa no necesita esta información y por tanto no se genera.
2. Puesto que los sistemas compilados poseen mayor eficiencia frente a los interpretados, que ofrecen una mayor sencillez a la hora de implementar sistemas flexibles, la unión de las dos técnicas de generación de aplicaciones puede dar lugar a un compromiso eficaz.
   En el caso de Cognac, todo el sistema se ejecuta en tiempo dinámico excepto aquella parte que se identifica como reflectiva; en este momento un intérprete ejecuta el código intermedio que define el nuevo comportamiento. Para el

sistema Iguana todo el código es traducido sin información dinámica, salvo aquél que va a ser adaptado.
3. El desarrollo de un MOP en dos niveles de interpretación (Closette) es más sencillo que si sólo elegimos uno (MetaXa). Si necesitamos modificar el MOP de Closette, deberemos hacerlo sobre el primer intérprete; en el caso de MetaXa, deberemos recodificar la máquina virtual.

8.4 Conclusiones

A lo largo de este capítulo, estudiando distintos tipos de sistemas reflectivos, hemos visto cómo la reflexión es una técnica que puede ser empleada para obtener flexibilidad en un sistema computacional.

Puesto que el concepto de sistema reflectivo puede ser catalogado de diversos modos (clasificación realizada en el capítulo anterior), analizaremos globalmente los sistemas reflectivos teniendo en cuenta dos criterios: cuándo se produce la reflexión y qué se refleja.

8.4.1 Momento en el que se Produce el Reflejo

La reflectividad en tiempo de ejecución otorga un elevado grado de flexibilidad al sistema, puesto que éste puede adaptarse a contextos impredecibles en fase de diseño. Cuando una aplicación necesita poder especificar nuevos requisitos dinámicamente, la reflectividad en tiempo de compilación no es suficiente.

Sin embargo, la reflectividad estática posee una ventaja directa sobre la dinámica: la eficiencia de las aplicaciones en tiempo de ejecución. La adaptabilidad dinámica de un sistema produce una ralentización del mismo en su ejecución.

Desde el punto de vista empírico, podemos ver cómo los sistemas estáticos ofrecen reflectividad del lenguaje de programación, cuando esto no ocurre en ningún sistema dinámico; el lenguaje de programación en estos casos se mantiene inamovible.

Los sistemas que ofrecen POA de forma estática se basan en el uso de la reflectividad estática (en tiempo de compilación) para lograr sus objetivos, por lo que poseen los mismos inconvenientes que ésta para adaptarse a nuevos requisitos dinámicamente (requisito 2.1.1).

8.4.2 Información Reflejada

El primer nivel de información a reflejar es la estructura del sistema en un modo de sólo lectura: introspección (punto 8.1). La característica práctica de este nivel de reflexión queda patente en el número de sistemas comerciales que la utilizan. Permite desarrollar fácilmente sistemas de componentes, de persistencia, comprobaciones de tipo dinámicas, o *middlewares* de distribución.

El segundo grado de información a reflejar es la reflexión estructural, en la que se permite tanto el acceso como la modificación dinámica de la estructura del sistema. A nivel práctico existen muchas posibilidades para este tipo de sistemas, muchas de ellas todavía no explotadas. Ejemplos pueden ser interfaces gráficas adaptables median-

te la incrustación, eliminación y modificación dinámica de la estructura de los objetos gráficos, aplicaciones de bases de datos que trabajen con un abanico de información adaptable en tiempo de ejecución, o la apertura a un nuevo modo de programación adaptable dinámicamente [Golm98] y creación de nuevos patrones de diseño [Ferreira98]. El auge que están tomando en los últimos tiempos lenguajes como Ruby o Python que ofrecen reflexión estructural es un indicador de la utilidad de este tipo de reflexión.

Cuando la semántica del sistema puede modificarse, nos encontramos en el tercer nivel de esta clasificación: reflectividad computacional. Éste ofrece una flexibilidad elevada para todo el sistema en su conjunto. Se ha utilizado en la mayoría de casos a nivel de prototipo, aplicándose a depuradores (*debuggers*), compilación dinámica (JIT, *Just In Time compilation*), desarrollo de aplicaciones en tiempo real, o creación de sistemas de persistencia y distribución.

CAPÍTULO 9
ARQUITECTURA DEL SISTEMA

Hasta el momento hemos descrito todos los objetivos y requisitos a conseguir por el sistema; hemos estudiado distintos sistemas y técnicas existentes en la consecución de éstos, y evaluado las aportaciones y carencias de los mismos. Ahora describiremos el funcionamiento y la arquitectura general del sistema innovador propuesto en esta Tesis, analizando brevemente su funcionamiento, su estructura y los objetivos generales a cumplir por cada uno de sus elementos.

En capítulos posteriores profundizaremos en la descripción de cada uno de los elementos del sistema, así como en la justificación de las técnicas seleccionadas y el cumplimiento de los objetivos marcados.

9.1 Funcionamiento del Sistema

La arquitectura del sistema se muestra en la Figura 46 y en la Figura 47. Analizaremos el funcionamiento temporal del sistema, remarcando cada uno de los elementos clave del mismo, que serán explicados posteriormente.

Como hemos visto anteriormente, mediante el empleo de una **Máquina Abstracta** puede conseguirse independencia del lenguaje o de la plataforma. Estos son algunos de los requisitos planteados en el capítulo 2, por lo que el sistema lo implementaremos sobre una que seleccionaremos posteriormente.

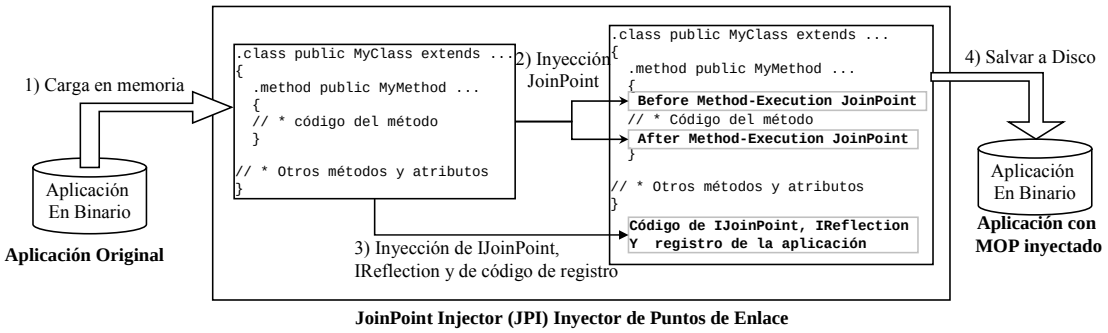

Figura 46: Arquitectura del sistema en tiempo de instrumentación

El funcionamiento temporal del sistema es el siguiente: En primer lugar (Figura 46), cuando una aplicación comienza su ejecución en el sistema, lo cual implica que ya está compilada, tiene que pasar por un proceso previo en el que se analiza su código y es modificada por parte del sistema. El elemento del sistema que se encarga de realizar esta tarea se denomina ***Join Point Injector*** (JPI) (inyector de puntos de enlace). El JPI inserta rutinas de reflexión computacional añadiendo un MOP en el código de la aplicación, además de una serie de clases que permitirán a la aplicación relacionarse con el resto del sistema, como son clases que implementan la parte del **Framework de Aspectos** correspondiente a la aplicación**, Mecanismos de Introspección Intraaplicaciones,** los elementos necesarios para la **Intercomunicación entre procesos** y el código necesario para registrarse en el sistema cuando comience su ejecución. Una vez generado el código definitivo se obtiene un nuevo ejecutable, con la funcionalidad ampliada, que es el que efectivamente se ejecuta en el sistema. La arquitectura del sistema en tiempo de ejecución se muestra en la Figura 47.

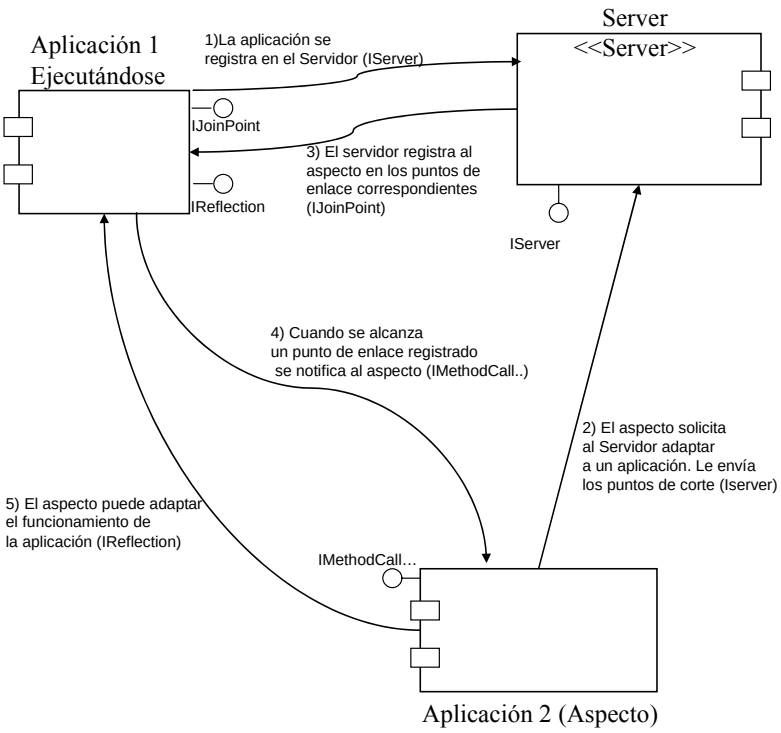

Figura 47: Arquitectura del sistema en tiempo de ejecución

Una vez que la aplicación ha sido modificada puede comenzar su ejecución, y lo primero que hace es registrarse en el **Servidor de Aplicaciones** del sistema (este registro lo hace mediante el código insertado anteriormente) con un GUID (identificador único global, *Globally Unique IDentifier*) que la identifica de forma única en el sistema. Este paso es necesario para que el Servidor tenga conocimiento de la existencia de la aplicación y pueda interactuar con ella.

En el sistema, los aspectos son aplicaciones normales, completamente funcionales y que tienen las mismas características que cualquier otra aplicación.

Cuando un aspecto necesita adaptar el comportamiento de una aplicación que está ejecutándose en el sistema, lo solicita al servidor. Para realizar la solicitud, el aspecto le envía al servidor, además de información sobre sí mismo (que el servidor enviará a la

aplicación a adaptar con el fin de que puedan interactuar entre ellos), un **fichero XML con la especificación de los puntos de corte** deseados y, obviamente, una identificación de la aplicación a la que desea adaptar. El servidor procesa el fichero XML y activa en la aplicación a ser adaptada los puntos de enlace correspondientes, seleccionados por los puntos de corte solicitados por el aspecto, y le transmite la información necesaria para que pueda comunicarse directamente con el aspecto (esto lo hace mediante las clases que implementan el Framework anteriormente inyectadas, y que explicaremos en detalle más adelante).

Cuando en la aplicación que está siendo adaptada se alcanza alguno de los puntos de enlace activados anteriormente, se notifica al aspecto que solicitó la activación que ha ocurrido esto (para ello utiliza las interfaces adecuadas definidas en el Framework de Aspectos, que tienen que ser implementadas por parte de los aspectos). La aplicación envía al aspecto información acerca del punto de enlace alcanzado y una serie de referencias a sí misma que posibilitarán al aspecto acceder a la aplicación y adaptar su comportamiento.

El aspecto, mediante la referencia recibida y las clases inyectadas previamente en la aplicación (que implementan las interfaces del Framework), puede acceder a la aplicación que está adaptando, obtener su estructura, invocar código de la misma, o ejecutar rutinas propias de ésta.

Si un aspecto que está adaptando a una aplicación necesita cambiar en cualquier momento los puntos de enlace seleccionados, puede hacerlo informando de ello al servidor suministrándole un nuevo fichero XML con la especificación de los nuevos puntos de corte deseados. El proceso es idéntico al seguido anteriormente, el servidor se encarga de procesar el fichero y activar y desactivar los puntos de enlace correspondientes en la aplicación que está siendo adaptada. Cuando un aspecto no necesita seguir adaptando a la aplicación, informa de esto al servidor, el cual desactiva en la aplicación los puntos de enlace correspondientes al aspecto, con el fin de que no sea advertido de su ejecución en adelante.

Nótese que el servidor hace de intermediario entre los aspectos y las aplicaciones a adaptar únicamente para activar y desactivar puntos de enlace en las aplicaciones. Una vez realizada esta operación, y cuando se alcanza un punto de enlace activo en una aplicación, es ésta la que de forma directa (no a través del servidor) informa al aspecto y, a partir de ese instante, el aspecto puede actuar, también de forma directa, sobre la aplicación.

Como se puede observar, no es necesario ningún conocimiento previo (durante el diseño) de los aspectos que van a adaptar a una aplicación, ni de las aplicaciones que pueden ser adaptadas.

9.2 Elementos del Sistema

En la sección anterior hemos visto una serie de elementos de los que consta el sistema que enumeraremos y describiremos muy someramente en esta sección.

- Máquina Abstracta: el sistema se implementará sobre una máquina abstracta con el fin de obtener el cumplimiento de requisitos tales como la indepen-

dencia del lenguaje, portabilidad o independencia de la plataforma, de forma directa, beneficiándose de las características de la máquina sin tener que añadir nada por parte del sistema para conseguir este requisito.

- Join Point Injector (JPI): el inyector de puntos de enlace se encarga de añadir código a la aplicación base, es decir, instrumentarla, para añadirle un MOP que ofrezca reflexión computacional al sistema. Además se encarga de inyectar el código necesario para que la aplicación interactué en el sistema.
- Framework de Aspectos: se definen una serie de interfaces y clases que tendrán que implementar los aspectos y las aplicaciones que utilicen el sistema. En el caso de las aplicaciones, esto no implica que tenga que ser el desarrollador el encargado de implementarlos, es el propio sistema, a través del JPI, quien se encarga de añadir las clases necesarias a la aplicación. Es el elemento que va a permitir la interacción entre aplicaciones y aspectos.
- Especificación del lenguaje de definición de puntos de corte: es necesario especificar un lenguaje que permita definir los puntos de corte en el sistema. Esto implica la selección, en primer lugar, de los posibles puntos de enlace que soportará el sistema y, en base a éstos, se creará el lenguaje para seleccionarlos.
- Servidor de Aplicaciones: este servidor se encarga de mantener la relación de las aplicaciones que se están ejecutando en el sistema, ofreciendo a los aspectos la posibilidad de adaptarlas. También recibe las peticiones de adaptación de aplicaciones por parte de los aspectos y activa o desactiva en las aplicaciones los puntos de enlace con el fin de que se produzca la adaptación. Dichas peticiones vienen expresadas mediante un fichero XML en el que se describen los puntos de corte en el lenguaje anteriormente definido.
- Mecanismos de Introspección Intraaplicaciones: se debe dotar al sistema de capacidades reflectivas externas, es decir, que una aplicación pueda acceder mediante reflexión a las características de otra aplicación. Si esta posibilidad no está presente en la máquina virtual sobre la que se implemente el sistema, será necesario implementarla por parte del sistema.
- Intercomunicación de Procesos: el funcionamiento de todo el sistema depende de la capacidad de intercomunicación entre los distintos procesos. Las aplicaciones deben poder comunicarse con el servidor, y a su vez éste debe poder interactuar con las aplicaciones. Los aspectos deben poder comunicarse con el servidor para hacer las solicitudes de adaptación. Las aplicaciones y los aspectos deben poder interactuar entre sí. Por todo ello es necesario disponer de un mecanismo de intercomunicación de procesos que soporte todas las características deseadas. Si la plataforma sobre la que se implemente el sistema carece de un mecanismo que soporte los requisitos necesarios se deberá implementar un sistema *ad hoc*.

En los siguientes puntos iremos describiendo cada uno de estos elementos analizando brevemente su estructura y objetivos generales. En capítulos posteriores se realiza una descripción detallada de cada uno de ellos.

9.2.1 Máquina Virtual

Como hemos visto anteriormente (capítulo 6), mediante el uso de una máquina abstracta pueden lograrse una serie de beneficios para un sistema de computación. Las máquinas abstractas se han utilizado anteriormente para conseguir sistemas indepen-

dientes del lenguaje y de la plataforma donde se ejecute, interoperabilidad nativa entre aplicaciones, portabilidad de código, etc.

Mediante el empleo de una máquina abstracta, sobre la que se implementa el sistema, se consigue satisfacer de forma directa alguno de los requisitos generales del sistema (dependiendo de las características de la máquina seleccionada):

- Independencia del Lenguaje, 2.1.2.
- Independencia de la Plataforma, 2.2.1.
- Adaptación de Código Binario 2.3.1.

Esto impone una serie de requerimientos a la máquina abstracta sobre la que implementemos el sistema, como son la existencia de un lenguaje intermedio o nativo de la máquina (independiente de otros lenguajes de programación), la posibilidad de modificar los programas codificados en este lenguaje y que la máquina se haya diseñado de forma independiente del entorno donde se ejecute.

De igual manera, la máquina virtual sobre la que se implemente el sistema debe ofrecer una serie de características que hagan posible cumplir con el resto de los requerimientos del sistema. Debe ofrecer mecanismos reflectivos suficientes para el sistema o, en caso de no soportarlos directamente, debe ser posible implementar la funcionalidad necesaria para ello, debe soportar la comunicación entre aplicaciones de una forma flexible, potente y segura, etc.

En el capítulo 10 procederemos a analizar los requisitos que debe cumplir la máquina virtual sobre la que se desarrollará el sistema para satisfacer los requerimientos especificados, realizaremos un análisis de alternativas posibles y procederemos a la selección de un entorno sobre el que desarrollaremos el sistema.

9.2.2 Lenguaje de Definición de Puntos de Corte

De acuerdo con el requisito 2.1.4, el conjunto de puntos de enlace que ofrezca el sistema debe ser lo más amplio posible y equiparable a los de los sistemas estáticos (en concreto al conjunto ofrecido por AspectJ, como estándar de facto de sistemas que ofrecen POA).

Para proceder a la selección de los puntos de enlace que soportará el sistema procederemos a estudiar los que soporta AspectJ y comprobar su posible existencia en la plataforma sobre la que implementemos el sistema. Ya que AspectJ está ligado al lenguaje Java, puede contener puntos de enlace que estén asociados a este lenguaje y no tengan sentido en un entorno independiente del lenguaje como tiene que ser el nuestro (requisito 2.1.2). De igual forma, en la plataforma que hayamos seleccionado pueden existir otros posibles puntos de enlace que tengan utilidad para el sistema y AspectJ no contemple por no existir en el lenguaje Java.

Una vez que hayamos seleccionado los puntos de enlace que soportará el sistema, debemos definir un mecanismo para que el usuario pueda especificar aquellos en los que está interesado, es decir, que pueda expresar los puntos de corte. En nuestro sistema el mecanismo para permitir al usuario expresar los puntos de corte consistirá en un lenguaje definido al efecto.

Todo el estudio de los puntos de enlace que ofrecerá el sistema, la selección de un mecanismo para expresar los puntos de corte, así como la definición del lenguaje de especificación de los puntos de corte basado en ese mecanismo, se realiza en el capítulo 11.

9.2.3 Servidor de Aplicaciones

El servidor de aplicaciones es el componente central del sistema, el cual permite la comunicación entre el resto de componentes.

El servidor debe ofrecer a las aplicaciones que se ejecuten en el sistema un mecanismo por el cual puedan: 1) registrarse cuando comiencen su ejecución y estén disponibles para ser adaptadas, y 2) desregistrarse cuando finalicen la ejecución con el fin de notificar al servidor que ya no están disponibles para la adaptación. Así mismo, el servidor debe encargarse de verificar que las aplicaciones que están registradas siguen estando activas, para evitar posibles problemas originados por el no desregistro de la aplicación (ya sea por una finalización anormal, un fallo en la comunicación, etc.).

De igual manera, el servidor debe ofrecer a los aspectos que quieran adaptar el comportamiento de una aplicación un mecanismo para poder hacerlo. Entre las operaciones que deben estar disponibles podríamos citar la obtención de un listado de las aplicaciones disponibles para su adaptación por parte del aspecto, la solicitud de adaptación de una aplicación determinada de acuerdo con los puntos de corte que se especifiquen (mediante un fichero XML externo al aspecto), la modificación de los puntos de corte utilizados en la adaptación de una aplicación, o la solicitud del cese de la adaptación por parte del aspecto de una aplicación determinada.

Todas estas operaciones que ofrece el servidor, tanto a las aplicaciones como a los aspectos, vienen definidas a través de una interfaz (las aplicaciones conocen esta interfaz y tienen el código necesario para trabajar con ella una vez que han sido modificadas por el JPI).

El servidor será capaz de procesar los puntos de corte definidos por parte del aspecto, para lo cual debe implementar un procesador del lenguaje anteriormente definido. El procesador construirá en memoria una representación de los puntos de corte solicitados por el aspecto mediante el patrón de diseño *Interpreter* [Gamma94]. Con esta representación, el sistema es capaz de determinar para cada punto de enlace existente en la aplicación que se quiere adaptar si es seleccionado o no por los puntos de corte especificados.

Otra tarea del servidor es la seguridad del sistema. Si no se implementa un mecanismo de seguridad podría ocurrir que cualquier aplicación fuese adaptada por parte de cualquier aspecto, lo que significaría un gran riesgo para la integridad de las aplicaciones, ya que un aspecto con código malicioso podría adaptar una aplicación accediendo a información a la que no debería acceder, provocando su malfuncionamiento e incluso produciendo la terminación anormal de su ejecución. El servidor debe permitir adaptar una aplicación por parte únicamente de aquellos aspectos que tengan privilegios para ello.

En el capítulo 13 mostraremos en detalle la arquitectura del servidor, los servicios que ofrece y la interfaz que define, así como las características de seguridad que debe contemplar y el mecanismo de comunicación entre procesos que se ha seleccionado para el sistema.

9.2.4 Framework de Aspectos

El Framework de Aspectos está formado por un conjunto de interfaces que definen el sistema y que permiten la interacción entre los distintos elementos del mismo.

Por un lado están las interfaces que deben implementar los aspectos para poder interactuar en el sistema. El objetivo de fijar una interfaz para los aspectos es poder recibir llamadas por parte de la aplicación a la que están adaptando. Mediante estas llamadas el aspecto recibe un conjunto de información que podrá usar para determinar qué hacer en cada caso. Esta información depende del tipo de punto de enlace que sea, por ejemplo en una invocación a un método se adjuntan los parámetros que se utilizan en la invocación, si los hay. Las llamadas por parte de las aplicaciones se producen cuando, durante su ejecución, se alcanza un punto de enlace que ha sido seleccionado por los puntos de corte del aspecto. Es responsabilidad del programador del aspecto el implementar esta interfaz.

Por otro lado, se define una interfaz que debe implementar la aplicación que vaya a ser adaptada. El objetivo de la interfaz es permitir la activación y desactivación de los puntos de enlace que ofrezca la aplicación por parte del Servidor de Aplicaciones. La implementación de esta interfaz no la realiza el programador de la aplicación sino que es inyectada cuando la aplicación ya ha sido compilada (ver punto 9.2.5). De esta manera garantizamos que cualquier aplicación se pueda ejecutar en el sistema, sin necesidad de disponer de su código fuente (requisito 2.3.1) y sin imponer condiciones que limiten la reutilización del código en otros sistemas (requisito 2.3.2).

Reflexión en el Framework

Para dotar al sistema de toda su potencia es necesario que un aspecto pueda acceder en ejecución a la información de la aplicación a la que está modificando, es decir, es necesario que el aspecto pueda tener acceso reflectivo sobre la aplicación (reflexión con acceso externo).

En general, y por motivos de seguridad, la reflexión con acceso externo no se permite en los sistemas existentes, por lo que deberemos implementar un mecanismo para solventar esta limitación en el sistema.

El primer paso es analizar las necesidades reflectivas del sistema, es decir, qué nivel de reflexión será necesario proporcionar por parte del sistema. Una vez determinado esto se tiene que diseñar un mecanismo que proporcione dicho nivel a las aplicaciones que lo soliciten. Para ello se definirá una interfaz que represente las funciones ofertadas por parte de las aplicaciones. Dicha interfaz debe ser implementada por parte de la aplicación que va a ser adaptada. Como entre los requisitos marcados para nuestro sistema (capítulo 2) están el que no sea necesario disponer del código fuente de la aplicación a modificar (requisito 2.3.1) y el no imponer condiciones a las aplicaciones a modificar (requisito 2.3.2), el sistema, de forma automática, debe ser capaz de hacer que

la aplicación implemente la interfaz. Esto se consigue en el paso de instrumentación de código, mediante la adición de las clases necesarias.

La explicación detallada del Framework de Aspectos definido puede verse en el capítulo 14.

9.2.5 Instrumentación de Código para Conseguir un MOP – JoinPoint Injector

Uno de los requisitos básicos del sistema que proponemos es su independencia del lenguaje (requisito 2.1.2), tanto para las aplicaciones que puedan ser adaptadas como para los aspectos que las adapten. Anteriormente hemos comentado que con el uso de una máquina virtual se puede cumplir con este requisito. Si la máquina virtual define un lenguaje intermedio propio, todos los programas en código fuente se compilan a este lenguaje intermedio para poder ser ejecutados por la máquina virtual.

Si la máquina virtual ofrece posibilidades para modificar (instrumentar) los programas que se encuentren en código intermedio, esto nos permite realizar el proceso de tejido sobre este código intermedio de una forma independiente del lenguaje (y además, sin la necesidad de disponer del código fuente del programa). Por todo ello será necesario que la máquina virtual que se utilice para implementar el sistema sobre ella facilite este proceso.

El primer paso para realizar la instrumentación de código es seleccionar las *JoinPoint Shadows* (sombras del punto de enlace) en el código intermedio. Una *joinpoint shadow* es la instrucción o conjunto de instrucciones en lenguaje intermedio que se corresponden con un punto de enlace. Por ejemplo, un punto de enlace expresado de forma abstracta puede ser "invocación a un método", esto en el lenguaje intermedio puede corresponderse con una serie de instrucciones (o conjunto de instrucciones) distintas dependiendo de si el método al que se invoca es de instancia o de clase. Otro ejemplo podría ser el punto de enlace "ejecución de un método", para seleccionar su *joinpoint shadow* habría que localizar los lugares en el código intermedio donde comienza y termina la ejecución de un método (que pudieran ser varios, dependiendo del lenguaje).

Una vez seleccionadas las *joinpoint shadows* es necesario determinar qué código se debe introducir en esos lugares para conseguir ofrecer un MOP de reflexión computacional.

Además de la adición del código para implementar el MOP mencionado, se inyectará en la aplicación otro código cuya finalidad es la de permitir que ésta se pueda integrar en el sistema. Por una parte se añadirán las clases necesarias para implementar el Framework de Aspectos en la parte correspondiente a la aplicación. También se insertará el código necesario para permitir la comunicación de la aplicación con el Servidor y con los posibles aspectos que soliciten adaptarla (este código será la implementación del mecanismo de intercomunicación de procesos 9.2.6).

De igual manera se inserta el código que hace que la aplicación se registre en el Servidor al comenzar su ejecución y se desregistre al finalizarla. Por último, se añadirá el código necesario para que la aplicación ofrezca reflexión intraaplicaciones (ver punto

9.2.4). Dicha reflexión intraaplicaciones es necesaria para permitir que los aspectos que estén adaptando a la aplicación tengan acceso durante la ejecución a la información de la misma, su estructura, e incluso puedan realizar invocaciones de miembros de la aplicación.

Al realizar todas estas adiciones sobre el código de la aplicación ya compilado a lenguaje intermedio no es necesario disponer del código fuente de la aplicación (requisito 2.3.1), ni se le imponen restricciones de ningún tipo que puedan afectar a su implementación.

En el capítulo 12 explicamos el proceso de selección de las *JoinPoint Shadows* y determinamos el código que debe ser inyectado en cada una de ellas. Realizamos un estudio de diferentes herramientas que permiten la manipulación de código y seleccionamos una de ellas en función de las necesidades del sistema. También se especifican las decisiones de implementación que se han tomado para conseguir cumplir los requisitos marcados.

9.2.6 Intercomunicación de Procesos

En nuestro sistema, los aspectos son aplicaciones completas, totalmente funcionales, que interactúan inicialmente con el Servidor de Aplicaciones para solicitar la adaptación de una aplicación y, posteriormente, con la aplicación a la que están adaptando.

La comunicación entre el aspecto y el Servidor de Aplicaciones consiste, básicamente, en la invocación remota de métodos. La comunicación entre los aspectos y las aplicaciones a las que adaptan consiste en invocaciones remotas a métodos y en manipulación remota de objetos (por medio del mecanismo de reflexión intraaplicaciones).

El mecanismo de intercomunicación de procesos que se utilice en el sistema debe cumplir con las necesidades de comunicación anteriormente mencionadas, sin invalidar ninguno de los requisitos generales del sistema (por ejemplo, no se podrá utilizar un sistema dependiente de un lenguaje o de una plataforma ya que invalidaría otros requerimientos impuestos).

El sistema debe soportar características de seguridad de tal forma que la comunicación pueda realizarse de forma cifrada y autenticada. De igual modo debe ser un sistema flexible que permita su utilización a través de distintas plataformas.

En el punto 13.4 se especifican las necesidades de intercomunicación existentes en el sistema y se realiza un análisis de alternativas, seleccionando una de ellas.

CAPÍTULO 10
MÁQUINA VIRTUAL

En el capítulo 9 hemos presentado la arquitectura general del sistema desglosándola en una serie de componentes o elementos que la conforman. En este capítulo vamos a centrarnos en la máquina virtual sobre la que se implementará el sistema.

Anteriormente (ver capítulo 6) hemos identificado el uso de máquinas abstractas como una técnica para conseguir una serie de beneficios, algunos de los cuales son necesarios para nuestro sistema. Analizaremos qué beneficios puede obtener el sistema del uso de una máquina abstracta que le permitan satisfacer los requisitos generales del sistema, qué requisitos se le imponen a la máquina para conseguir esos beneficios y, por último, realizaremos un estudio de diversas alternativas existentes seleccionando una de ellas para la realización de un prototipo que demuestre la viabilidad del sistema propuesto.

10.1 Aportaciones del Uso de una Máquina Abstracta

En el capítulo 6 hemos visto diversos usos que se le han dado a las máquinas abstractas con el fin de conseguir ciertos objetivos. A continuación comentaremos los beneficios significativos para nuestro sistema mediante el uso de una máquina abstracta.

- Independencia de la plataforma[25]: como se comentó anteriormente, una de las características más importantes del uso de una máquina abstracta es la ausencia de una implementación física. Debido a que quien interpreta las instrucciones de la máquina es un procesador *software* se obtiene una independencia de la plataforma física sobre la que esté implementado. Una vez que el programa está compilado para su ejecución por parte de la máquina abstracta, su ejecución podrá realizarse en cualquier plataforma en la que se haya implementado un procesador lógico capaz de interpretar sus instrucciones.
- Independencia del lenguaje: otro de los usos que se han comentado anteriormente es el desarrollo de sistemas multilenguaje. Si las instrucciones de la máquina abstracta se representan mediante un lenguaje común y ese lenguaje común es capaz de representar las abstracciones de todos los lenguajes fuen-

[25] Entendiendo como plataforma al conjunto de microprocesador y sistema operativo.

te, sin estar ligado a ninguno de ellos, es posible construir compiladores de estos lenguajes que traduzcan los programas codificados en ellos a instrucciones de la máquina abstracta. Una vez que los programas están expresados en instrucciones en código nativo de la máquina abstracta se podrán ejecutar en cualquier implementación de la misma, sin necesidad de ningún otro tipo de condicionante. Si es posible acceder a la representación de los programas (su código) en este lenguaje nativo de la máquina y modificar esa representación, estaremos modificando el programa sin tener que conocer el lenguaje original en el que fueron codificados. Con esto podemos modificar programas escritos en cualquier lenguaje.

- Adaptación de código binario: como hecho derivado de la existencia de un lenguaje nativo de la máquina abstracta, todos los programas codificados en cualquier lenguaje se traducen a él como paso previo a su ejecución. Si es posible trabajar con los programas, es decir, modificarlos, ya traducidos a este lenguaje, no se necesita disponer de su código fuente para poder adaptarlos.
- Interoperabilidad nativa entre aplicaciones: como se comentó en el punto 6.3.4.2 uno de los principales problemas en la intercomunicación de aplicaciones distribuidas es la representación de la información enviada (un mismo tipo de dato, en un mismo lenguaje, puede representarse de formas distintas en cada plataforma donde se compile). Una vez que los programas, codificados originariamente en lenguajes distintos, han sido traducidos a las instrucciones de la máquina abstracta, la comunicación entre ellos es mucho más sencilla, ya que comparten el mismo sistema de tipos, las mismas representaciones en memoria para los distintos tipos de dato, etc. pese a que no se ejecuten sobre la misma plataforma.

10.2 Requisitos Impuestos a la Máquina Abstracta

Analizando los requisitos necesarios impuestos a la máquina abstracta para poder desarrollar el sistema propuesto podemos enumerarlos del siguiente modo:

- Independencia de la plataforma: la máquina abstracta sobre la que se vaya a implementar nuestro sistema debe haber sido diseñada sin dependencias de plataforma alguna (tanto *hardware* como sistema operativo), de tal manera que se puedan realizar implementaciones de la misma en cualquier plataforma. De esta forma, el propio sistema a implementar obtendrá el mismo beneficio (puesto que se ejecuta en la implementación de la máquina abstracta), siendo uno de los requerimientos generales impuestos al sistema (requisito 2.2.1).
- Lenguaje propio de la máquina abstracta: la máquina abstracta debe contar con un lenguaje intermedio, al cual se traducen los programas codificados en lenguaje fuente como paso previo a poder ser ejecutados por la máquina. Esto garantiza que el mismo programa se podrá ejecutar en cualquier plataforma donde haya una implementación de la máquina abstracta.
- El lenguaje propio de la máquina abstracta debe ser neutral: el lenguaje no debe ser dependiente de un lenguaje de programación, es decir no debe haber sido diseñado para dar soporte a uno determinado, lo que dificultaría, e incluso imposibilitaría, la traducción de ciertos lenguajes a dicho lenguaje de la máquina. Si la máquina cumple este requisito, se podrían utilizar en ella

programas codificados en cualquier lenguaje (con la condición de que exista un compilador de ese lenguaje al lenguaje de la máquina abstracta).

- Posibilidad de acceder a la codificación de los programas en el lenguaje de máquina abstracta: al poseer un lenguaje nativo la máquina abstracta y poder trabajar sobre él (acceder al código como si fuesen datos y poder manipularlo), se consigue que el sistema pueda trabajar de una forma independiente del lenguaje en el que se hayan implementado las aplicaciones y los aspectos. Las aplicaciones y los aspectos pueden estar implementados en cualquier lenguaje, pero son traducidos al lenguaje nativo para poder ser ejecutados, con lo que si el sistema puede acceder a la representación del programa una vez traducido, todo el trabajo se podrá realizar sobre este lenguaje intermedio de la máquina, es decir, de forma independiente del lenguaje original en el que se hubiese codificado la aplicación.
  De igual modo, no es necesario el código fuente de la aplicación a adaptar, ya que con su representación en código intermedio es suficiente. Además, al ejecutarse los programas una vez que han sido traducidos al lenguaje de la máquina, el intercambio de información entre ellos es mucho más sencillo, pues comparten el mismo sistema de tipos.
- Capacidades reflectivas: la máquina abstracta debe contar, al menos, con características de introspección, de tal manera que se pueda acceder a la estructura de un programa. Esto es necesario para que los aspectos puedan obtener información reflectiva de las aplicaciones que están adaptando.
- Interacción directa entre aplicaciones: puesto que los aspectos, que son aplicaciones dentro del sistema, van a adaptar el funcionamiento de las aplicaciones es deseable que puedan interactuar directamente entre sí, sin la necesidad de utilizar una capa intermedia para realizar la comunicación.
- Mecanismo de comunicación entre procesos: puesto que un aspecto vital en el sistema es la comunicación entre los aspectos y la aplicación a la que adaptan, es necesario que la máquina abstracta soporte un mecanismo de comunicación entre procesos que sea independiente del lenguaje, flexible, seguro, distribuido y configurable.
- De propósito general: uno de los requisitos impuestos al sistema es que tiene que ser de propósito general (requisito 2.1.3), es decir, que debe servir para resolver cualquier tipo de problema, no únicamente uno en concreto. Debido a esto, la máquina abstracta sobre la que se implemente el sistema debe ser igualmente de propósito general para no imponer limitaciones al sistema.
- Ampliamente difundido: cuanto mayor sea la difusión de la máquina abstracta que se seleccione mayor será el conjunto de potenciales usuarios del sistema. De igual forma, se supone que existirá más *software* desarrollado para la misma y, por lo tanto, que se pueda beneficiar del sistema (requisito 2.2.2).
- Buen rendimiento: es obvio que si la base sobre la que se desarrolle la plataforma ofrece un buen rendimiento todo el sistema se beneficiará de él (requisito 2.2.3).

Una máquina abstracta que ofrezca estas características será válida para desarrollar el sistema propuesto.

10.3 Elección de una Máquina Abstracta

En los puntos anteriores hemos visto los beneficios que el uso de una máquina abstracta podría aportar al sistema propuesto y los requisitos que debe cumplir la máquina abstracta para aportar esos beneficios. En este punto vamos a proceder a seleccionar una plataforma concreta, justificando la elección en base a los requisitos impuestos anteriormente.

Uno de los requisitos impuestos a la máquina abstracta es el hecho de que debe ser de propósito general, por lo que descartamos inmediatamente cualquier sistema de propósito específico (como puede ser portabilidad de código, interoperabilidad de aplicaciones, etc.). Otro requisito impuesto es la independencia de la plataforma, lo que reduce el abanico de posibles candidatas a máquinas que lo sean, es decir, que hayan sido diseñadas de forma independiente de cualquier *hardware* o sistema operativo, por lo que podrá realizarse una implementación de las mismas en cualquier entorno. Entre las máquinas abstractas independientes de la plataforma cabe destacar a Smalltalk, Java y .NET, todas ellas cuentan con diversas implementaciones en muy diversos entornos.

Si ahora restringimos en base a otro requisito muy importante como es la independencia del lenguaje, el abanico se reduce aún más, quedando Java y .NET como posibilidades, aunque en el caso de Java su diseño se centra en soportar el lenguaje de programación Java, haciendo difícil su uso por parte de otros lenguajes [Meijer01], por lo que cumple este requisito en menor medida (realmente lo cumple de una forma teórica más que práctica).

Por ultimo, únicamente la plataforma .NET cumple el requisito de interacción directa entre aplicaciones. Además .NET cumple el resto de requisitos impuestos, puesto que, como vimos en el capítulo 8, tiene capacidades reflectivas que nos permiten conocer la estructura de las aplicaciones y también permite acceder y realizar manipulaciones sobre los programas codificados en el lenguaje nativo de la máquina. De igual modo, ofrece mecanismos que permiten la comunicación entre procesos de forma independiente al lenguaje y distribuida. La plataforma .NET está ampliamente difundida, ya que una de sus implementaciones, el CLR de Microsoft, se encuentra instalado en la práctica totalidad de los ordenadores personales con sistema operativo Windows y además ofrece un alto rendimiento en ejecución. Es decir, la plataforma .NET cumple con la totalidad de los requisitos impuestos, por lo que es la seleccionada para implementar sobre ella el sistema propuesto en esta Tesis.

Existen diversas implementaciones del estándar CLI [ECMA335] de la plataforma .NET, tal y como vimos en 6.5.3. En general, estas implementaciones no se limitan a cumplir con el estándar, sino que añaden funcionalidades no contempladas en él. Con el objetivo de que nuestro sistema pueda ejecutarse sobre cualquier implementación (de acuerdo al requisito 2.2.4), no debemos hacer uso para la implementación del sistema de ninguna característica no incluida en el estándar.

Existen también diferentes versiones del estándar, añadiendo cada nueva versión funcionalidad a la ya existente, pero conservando la compatibilidad con versiones anteriores. Para poder ser utilizado en cualquier implementación existente el sistema debe ser desarrollado basándose en las características básicas del estándar disponibles en to-

das las versiones. Cumpliendo este requisito, el sistema propuesto podrá ejecutarse sobre cualquier implementación del estándar.

CAPÍTULO 11
LENGUAJE DE DEFINICIÓN DE PUNTOS DE CORTE

En este capítulo procederemos a seleccionar los puntos de enlace que debe soportar el sistema para cumplir con los requisitos impuestos, para lo cual nos basaremos en los sistemas existentes, buscando el conjunto de puntos de enlace mayormente aceptado.

Posteriormente, seleccionaremos un mecanismo por medio del cual podamos expresar los puntos de corte correspondientes y, haciendo uso del mecanismo anteriormente seleccionado, diseñaremos un lenguaje que permita expresar los puntos de corte de acuerdo a los requisitos planteados.

11.1 Elección de los Puntos de Enlace

Como hemos comentado en el punto 4.3, un punto de enlace es un punto de la ejecución de un programa bien definido donde se podrá añadir código. Sólo en los puntos de enlace se permite añadir código (o modificar el comportamiento del programa) por parte de los aspectos.

Como hemos visto en el capítulo 5, existen múltiples sistemas que soportan el Desarrollo de Software Orientado a Aspectos (DSOA), y cada uno de ellos propone o implementa una serie de puntos de enlace distintos, no habiendo unanimidad respecto a cuáles son los que debe soportar un sistema. Sin embargo, el sistema que cuenta con una mayor aceptación es AspectJ (en Diciembre de 2006 figura en el lugar 86 del ranking TIOBE [TIOBE] siendo el único sistema con soporte al DSOA que figura en el mismo). De hecho la mayoría de sistemas intentan implementar un conjunto de puntos de enlace similar al de AspectJ (al menos los sistemas estáticos, los dinámicos en general presentan un conjunto de puntos de enlace mucho más reducido), de tal manera que se ha establecido como patrón de referencia a la hora de comparar unos sistemas con otros.

Es por esto que vamos a basarnos en los sistemas estudiados, especialmente en AspectJ, para definir los puntos de enlace que son deseables que tenga un sistema, y comprobaremos la viabilidad de su existencia en la plataforma .NET, que es sobre la que implementaremos nuestro sistema (como hemos visto en el capítulo 10), y en concreto en su lenguaje intermedio *CIL*.

El conjunto de puntos enlace que soporta AspectJ es el siguiente [AspectJc]:

- Invocación a un método.
- Invocación a un constructor.
- Ejecución de un método.
- Ejecución de un constructor.
- Inicialización de objetos que se crean con el constructor.
- Preinicialización de atributos (asignación en la declaración) realizada antes de la invocación al constructor del padre (super).
- Inicialización de bloque estático.
- Acceso de lectura a campo.
- Acceso de escritura a campo.
- Cuando una excepción `IOException` (o un subtipo de la misma) se trata en un bloque catch.
- Ejecución de cualquiera de los *advice* inyectados.

De estos puntos de enlace hay tres que no tienen sentido en nuestro sistema. Tanto la inicialización de bloque estático como la preinicialización de atributos son propias del lenguaje Java, por lo que no tienen sentido en un entorno independiente del lenguaje como el propuesto (requisito 2.1.2). La ejecución de un *advice* inyectado tampoco tiene sentido en nuestro sistema, ya que en el sistema propuesto los aspectos son aplicaciones normales (requisito 2.4.2), por lo que pueden ser adaptados como cualquier otra aplicación, con lo que no es necesario este punto de enlace ya que se puede conseguir la misma funcionalidad de forma más sencilla. De igual modo tampoco es necesario contemplar la inicialización de objetos que se crean con el constructor.

En resto de puntos de enlace sí tienen sentido en nuestro sistema por lo que los adoptamos. El punto de enlace de tratamiento de una excepción no se limita a un tipo concreto de excepción, tal y como ocurría en AspectJ, sino que puede aplicarse a cualquier tipo de excepción, ampliando de esta forma su potencia.

Además se han añadido dos nuevos puntos de enlace que son propios de la plataforma .NET: el acceso de lectura y el de escritura a propiedades –*properties.*

Existen sistemas que ofrecen conjuntos de puntos de enlace diferentes, pero en general esto es debido a que soportan paradigmas distintos, como puede ser el desarrollo de software por componentes o, a que por ejemplo, se ejecutan en entornos que no soportan la orientación a objetos.

Por todo ello, el conjunto de puntos de enlace que soportará nuestro sistema es el siguiente:

- Invocación a un método.
- Invocación a un constructor.
- Ejecución de un método.
- Ejecución de un constructor.
- Acceso de lectura a campo.
- Acceso de escritura a campo.
- Acceso de lectura a una propiedad.
- Acceso de escritura a una propiedad.
- Tratamiento de una excepción en un bloque catch.

11.2 Elección de los Tipos de Advice

Además de definir los puntos de enlace del sistema, que consiste en especificar el dónde podemos adaptar a una aplicación, es necesario poder decidir el cuándo queremos adaptarla. Esto, en la mayoría de sistemas, viene expresado como el tipo de *advice*. Una vez más AspectJ es el estándar de facto y es el que estableció inicialmente los tipos de *advice* existentes de forma general que consisten en *before, after y around.*

El tipo *before* significa que la adaptación del comportamiento de la aplicación se produce antes de la ejecución del punto de enlace. Por ejemplo, si el punto de enlace es una invocación a un método, la existencia de un *advice* de este tipo implicaría que antes de realizar la invocación se ejecutaría el código del *advice*, dando lugar a la modificación del funcionamiento de la aplicación, y cuando terminase de ejecutarse este código, se ejecutaría realmente la invocación al método.

El tipo *after* significa que la adaptación del comportamiento de la aplicación se produce inmediatamente después de la ejecución del punto de enlace. En AspectJ existen variaciones de este tipo dependiendo de si el retorno se produce de forma normal, o si devuelve un valor determinado o se produjo por el lanzamiento de una excepción.

El tipo *around* significa que la adaptación del comportamiento de la aplicación se produce en lugar de la ejecución del punto de enlace. Es decir, en el mismo caso anterior de un punto de enlace que sea la invocación a un método, la existencia de un *advice* de tipo *around* implica que en lugar de realizar la invocación al método se ejecuta el código del *advice*. En AspectJ en el código del *advice* se puede decidir que una vez terminada su ejecución se realice o no la ejecución del punto de enlace (en este caso la invocación al método).

En general los sistemas estáticos intentan soportar los tres tipos. Los sistemas dinámicos también los soportan aunque de una forma más restringida, sobre todo el tipo *around*, que suele estar disponible para únicamente algún tipo de punto de enlace en concreto, o no estar disponible en modo alguno.

Al igual que ocurre con los puntos de enlace, lo deseable es que el sistema ofrezca el mayor número de posibilidades al desarrollador, por lo que nuestro sistema debe implementar los tres tipos de *advice*, y para seguir la nomenclatura más habitual en los sistemas que ofrecen DSOA los llamaremos *before, after y around* y lo denominaremos como el tiempo del punto de corte.

11.3 Selección del Mecanismo de Expresión de los Puntos de Corte

Una vez establecidos los puntos de enlace que soportará el sistema, así como los tipos de *advice* existentes (el tiempo), es necesario definir un mecanismo mediante el que se puedan expresar los puntos de corte. Para ello debemos tener en cuenta los requisitos fijados en el capítulo 2.

Los requisitos que se imponen al mecanismo de expresión de puntos de corte, en base a los requisitos generales del sistema, son:

- Independencia del lenguaje. El mecanismo no puede ser dependiente del lenguaje ya que en caso de serlo impondría una dependencia al sistema que impediría el cumplimiento del requisito 2.1.2.
- Debe permitir la reutilización de los aspectos. El mecanismo no debe introducir ninguna dependencia en los aspectos de tal manera que dificulte o impida su reutilización, ya sea dentro del propio sistema en otra aplicación, o en otro sistema distinto al aquí planteado.
- Debe contar con una expresividad rica. El sistema debe permitir expresar los puntos de corte de una manera potente pero simple.

Para establecer el mecanismo a utilizar revisaremos qué mecanismos son los que emplean los sistemas existentes, analizando las ventajas e inconvenientes en base a los requisitos impuestos y procederemos a seleccionar el adecuado en base a los mismos.

Existen sistemas que han realizado extensiones al lenguaje sobre el que se implementan los aspectos, por ejemplo AspectJ o Caesar, por lo que el código del aspecto se encuentra mezclado con los puntos de corte. Esto tiene varios inconvenientes como son el acoplamiento que se produce entre el aspecto y la aplicación a la que está adaptando, reduciendo así las posibilidades de reutilización del aspecto en el mismo sistema (requisito 2.4.3.2), el hecho de que al implementar los aspectos en un lenguaje que ya no es el estándar es necesario disponer de un compilador específico, lo que resta portabilidad al sistema, y para utilizar el aspecto en otro sistema sería necesario modificarlo (requisito 2.4.3.1).

Otros sistemas definen los puntos de corte programáticamente mediante librerías de métodos que se utilizan dentro del código del aspecto para definir los puntos de corte en los que está interesado. Ejemplos de este tipo son PROSE o SteamLoom de AORTA. Este mecanismo presenta inconvenientes similares al anterior. Al incluir código específico en el aspecto para definir los puntos de corte estamos creando dependencias entre el aspecto y la aplicación a la que adaptan. De igual forma para poder utilizar esos aspectos en otro sistema sería necesario eliminar todo el código que se encarga de la definición de los aspectos.

Otra posibilidad utilizada por varios sistemas es el empleo de los *custom attributes* en la plataforma .NET (ejemplos de ello son AspectDNG o LOOM.NET) o las *annotations* en el lenguaje Java (con ejemplos como AspectJ, AspectWerkz o JBoss). Con este sistema sigue existiendo una dependencia del aspecto con el contexto aunque ésta es significativamente menor, de tal manera que para reutilizar un aspecto con otra aplicación (dentro del mismo sistema) es necesario modificar el código (los *custom attributes* o las *annotations*) del aspecto.

Por último, otra posibilidad empleada por varios sistemas es la definición de un lenguaje propio para especificar los puntos de corte, los cuales se codificarán en un fichero independiente del código del aspecto. Varios sistemas que utilizan los otros mecanismos para expresar los puntos de corte están migrando hacia este mecanismo o lo soportan indistintamente. Ejemplos de este sistema son Weave.Net, Source.Net, aspectDNG, CLAW, AspectWerkz, JBoss, CAM/DAOP, dotSPECT o Reflex entre otros. La mayoría (no todos) de estos lenguajes están basados en XML por características tales como: la facilidad para definir el lenguaje, su independencia respecto a los posibles lenguajes de implementación utilizados, la posibilidad de verificar su validez, su potencia,

la existencia de múltiples utilidades para procesar ficheros XML de forma directa y por ser legible por un usuario.

El hecho de separar la definición de los puntos de corte del código de los aspectos presenta ventajas claras. En primer lugar el aspecto puede ser completamente independiente de la aplicación a la que está adaptando, por lo que podrá ser reutilizado en otra aplicación de forma directa, sin tener que modificar nada (un ejemplo de esto podría ser un aspecto de traza o de log genérico que funcionaría igual para cualquier aplicación). Como segundo beneficio, derivado del hecho de no entremezclar los puntos de corte junto al código del aspecto, es que el código del aspecto será un código estándar que podrá ser procesado por cualquier compilador del lenguaje en el que esté implementado con lo que su utilización en otra plataforma sería mucho más sencilla (de todos modos, para poder utilizar el aspecto en otra plataforma, habría que tener en cuenta que pueden haberse impuesto restricciones, como el uso de ciertas librerías, formas de trabajo, etc. en la plataforma destino).

Por todo lo mencionado basaremos nuestro mecanismo de definición de puntos de corte en un lenguaje definido al efecto. Los puntos de corte se definirán en un fichero independiente del fichero con el código del aspecto. El lenguaje que definiremos estará basado en XML, y su definición vendrá dada mediante un esquema XML [W3C01], obteniendo así los siguientes beneficios:

- Separación completa entre el código del aspecto y los puntos de corte, facilitando así la reutilización de los aspectos, tanto dentro del mismo sistema como en otros sistemas (requisito 2.4.3).
- Total independencia del lenguaje (relacionado con el requisito 2.1.2). Al definir los puntos de corte en XML no hay ninguna dependencia respecto al lenguaje de implementación de los aspectos.
- La posibilidad de definir un lenguaje que cumpla con el requisito 2.1.5 Expresividad del Lenguaje de Definición de Puntos de Corte.
- La existencia de multitud de herramientas que permiten al usuario crear un fichero XML, de acuerdo a la especificación contenida en el esquema, y validarlo de forma previa a ser introducido en el sistema.

11.4 Lenguaje de Definición de los Puntos de Corte

Como hemos comentado anteriormente, para definir los puntos de corte del sistema emplearemos un lenguaje basado en XML, el cual tiene que cumplir una serie de requisitos que enumeramos a continuación:

- Independencia del lenguaje.
- Separación de la definición de los puntos de corte del código de los aspectos.
- Expresividad rica.

Para definir el lenguaje de especificación de los puntos de corte del sistema vamos a basarnos en la aproximación que utilizaron los autores del proyecto Weave.net (visto en 5.1.2). Tomando como modelo las extensiones realizadas por AspectJ al lenguaje Java crearon un esquema XML que tenía la misma expresividad [Lafferty03]. La metodología seguida consistió en primer lugar en expresar mediante una extensión de BNF (Backus–Naur Form) [Estier02] las extensiones realizadas a Java por parte de As-

pectJ para, posteriormente, traducir de BNF al esquema XML. Con este proceso los autores logran obtener un mecanismo para expresar los puntos de corte de AspectJ mediante un fichero XML externo, con la misma potencia expresiva que el original y con dos ventajas importantes: la independencia del lenguaje y la no necesidad de realizar extensiones al lenguaje en el que se implementen los aspectos [Lafferty03].

Puesto que nuestro sistema no soporta los mismos puntos de enlace que AspectJ (aunque sí tiene muchos en común), no tendrá los mismos puntos de corte. Basándonos en el esquema definido en [Lafferty03] procederemos a eliminar aquellas partes que no son contempladas por nuestro sistema y a añadir aquellas que soporta nuestro sistema, pero no están incluidas en AspectJ. Además de esto, es necesario tener en cuenta que AspectJ está basado en Java, al contrario que nuestro sistema, que no debe estar basado en ningún lenguaje concreto, por lo que cualquier particularidad referida a un lenguaje concreto debe ser evitada.

El esquema a definir permitirá expresar en un único fichero XML todos los puntos de corte existentes entre un aspecto y la aplicación a la que va a adaptar. De este modo toda la información relativa a los puntos de corte estará centralizada en un único punto. En el fichero XML no debe existir ninguna referencia al aspecto. Como veremos más adelante, es el aspecto, en el momento de solicitar adaptar una aplicación, el que selecciona el fichero con los puntos de corte que debe usarse para realizar dicha adaptación. Al no existir ninguna referencia al aspecto es posible reutilizar los ficheros de puntos de corte por parte de otros aspectos.

Dado que las incumbencias ortogonales, por definición, se encuentran diseminadas a lo largo de múltiples módulos y afectan a múltiples puntos de enlace en el sistema, el lenguaje de definición debe de proveer un modo "económico" para seleccionar dichos puntos de enlace [Laddad03] (es decir, un sistema por el que no haya que especificar explícitamente todos y cada uno de los puntos de enlace, sino que puedan especificarse implícitamente). AspectJ utiliza una sintaxis basada en el uso de los comodines (expresiones regulares) para seleccionar puntos de enlace que comparten ciertas características. Nosotros haremos lo mismo, utilizando de forma conjunta expresiones regulares en cadenas de texto (utilizamos el carácter * como indicador del comodín) y elementos XML que representen comodines. En la Figura 48 se puede observar el uso de comodines como elemento XML (en este caso se trata de especificar los tipos de los parámetros en la invocación a un método o constructor, indicando el elemento *param_wildcard* cualquier número de parámetros de cualquier tipo cada uno de ellos).

Al igual que la mayoría de lenguajes, nuestro sistema ofrecerá una serie de operadores para construir expresiones condicionales complejas a partir de otras expresiones condicionales más simples. Estos operadores se representarán por medio de elementos del lenguaje. En concreto el sistema soportará el operador unario de negación (not) y los operadores binarios de disyunción (or) y de conjunción (and) (Figura 49).

A continuación se muestran una serie de figuras que son representaciones gráficas de parte del esquema XML definido.

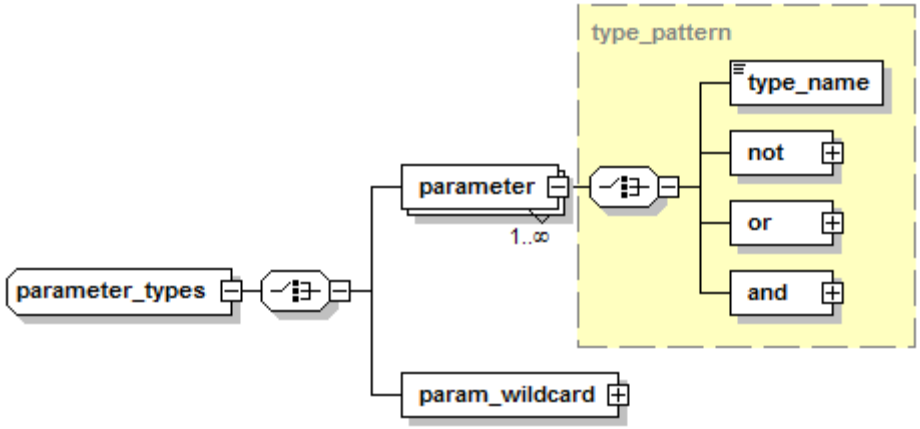

Figura 48: Uso de comodines en el lenguaje

En el siguiente fragmento de código podemos ver la utilización del comodín en una cadena de texto (en él se está seleccionando cualquier identificador cuyo nombre empiece por *get*).

```
<name>
      <identifier_name>get*</identifier_name>
</name>
```

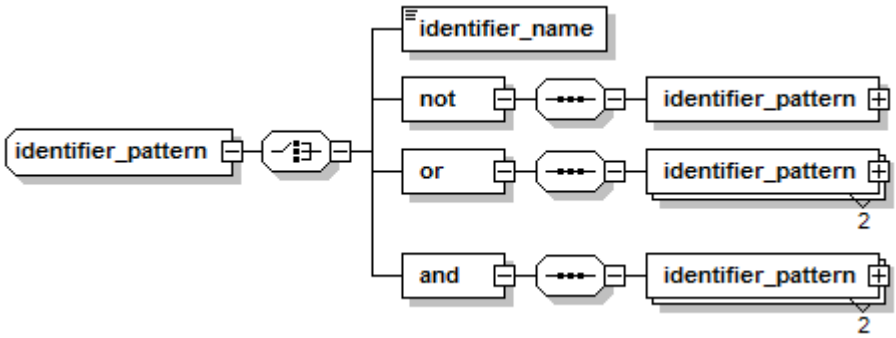

Figura 49: Operadores del lenguaje

El elemento que representa al operador unario de negación seleccionará todos los puntos de enlace (del tipo para el que se está definiendo el punto de corte) excepto aquellos que son seleccionados por el punto de corte. El elemento que representa al operador binario de disyunción selecciona aquellos puntos de enlace que se corresponden con cualquiera de los dos elementos operando sobre los que actúa. El elemento que representa al operador binario de conjunción selecciona aquellos puntos de enlace que se corresponden de forma simultánea con los dos elementos operando sobre los que actúa.

Por ejemplo, en el siguiente fragmento de código se seleccionan todos aquellos identificadores cuyo nombre no comienza ni por get ni por set.

```
<name>
      <not>
            <identifier_pattern>
                  <or>
                        <identifier_pattern>
                              <identifier_name>get*</identifier_name>
                        </identifier_pattern>
                        <identifier_pattern>
                              <identifier_name>set*</identifier_name>
```

```
                        </identifier_pattern>
                    </or>
                </identifier_pattern>
            </not>
</name>
```

La estructura general del lenguaje permite definir un número indeterminado de puntos de corte en el mismo fichero. Cada punto de corte consta de dos elementos primordiales: el tiempo y el tipo de punto de enlace al que se refiere (En la Figura 50 se muestra de forma gráfica la estructura general del esquema). El tiempo podrá ser cualquier combinación de las tres posibilidades que ofrece el sistema (*before, after y around)*, es decir, un mismo punto de corte puede aplicarse de forma simultánea en diferentes tiempos. El tipo de punto de enlaces será uno (y sólo uno) de los posibles puntos de enlace existentes en el sistema.

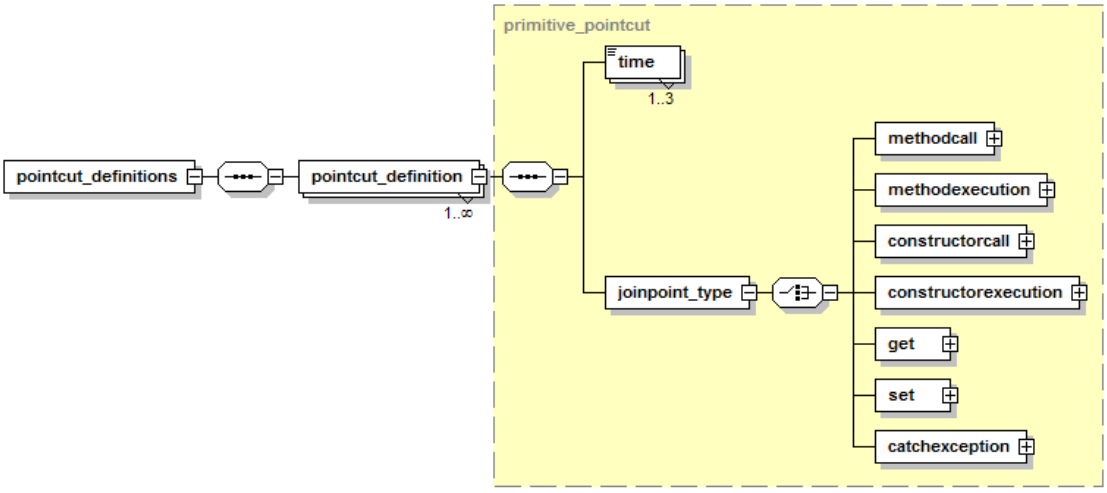

Figura 50: Tipos de puntos de corte

Dependiendo del tipo de punto de enlace al que hace referencia el punto de corte se deberá especificar una información u otra. Esta información servirá para localizar inequívocamente los puntos de enlace que coinciden (se ven seleccionados) por el punto de corte.

Nuestro sistema, tal y como hemos visto anteriormente, se desarrolla sobre la máquina abstracta de .NET y, particularmente, haciendo uso del lenguaje intermedio *CIL* que soporta dicha máquina. En este lenguaje todos los métodos, constructores, campos, etc. tienen una signatura que los identifica de forma única. Utilizaremos esta signatura en los puntos de corte para definir con exactitud los puntos de enlace que se desean seleccionar. Para ello analizamos la signatura de cada uno de los elementos involucrados en los puntos de enlace del sistema (por ejemplo en la ejecución o invocación a un método nos interesa la signatura del método, pero en el acceso de lectura o escritura a un campo nos interesa la signatura del campo), y añadimos al lenguaje los elementos necesarios para representarla.

A modo de ejemplo podemos mencionar que la signatura de un método en *CIL* consta de cuatro elementos:

- Los *Flags* o modificadores: que representan información sobre el método tal como que el método es público, privado, etc.
- El tipo de retorno del método.
- El nombre cualificado, es decir, el nombre completo del método que lo ubica de forma única dentro de la aplicación, compuesto por el nombre del *namespace* más el nombre de la clase y el nombre del método (de esta manera puede haber varios métodos que se llamen igual en distintas clases).
- Los parámetros del método (puede tenerlos o no), indicando el tipo de cada uno de ellos.

Como la signatura de un método consta de esos cuatro elementos, se crean en el lenguaje los elementos necesarios para poder expresar dicha signatura. Se procederá de igual forma con los demás tipos de punto de enlace, creando los elementos necesarios para poder representar las distintas signaturas de los elementos (campos, constructores, etc.). En la Figura 51 podemos ver los elementos que forman parte de la signatura de un método en el lenguaje.

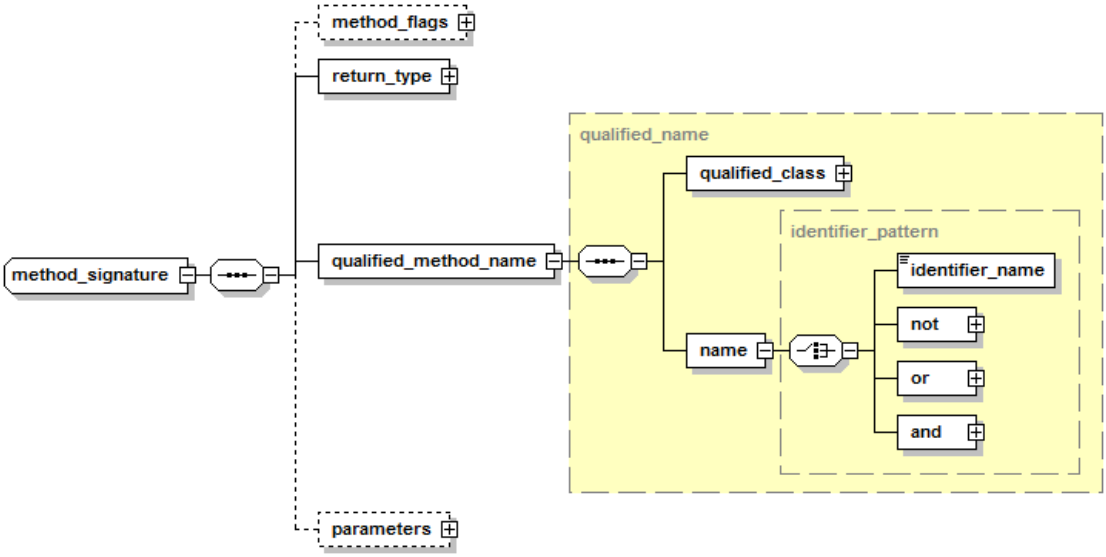

Figura 51: Signatura de un método

En el apéndice a se puede ver en detalle la especificación del lenguaje definido para especificar los puntos de corte.

CAPÍTULO 12
INSTRUMENTACIÓN DE CÓDIGO PARA CONSEGUIR UN MOP

En capítulos anteriores hemos identificado la reflexión como un mecanismo para conseguir la adaptación de aplicaciones y, si la reflexión tiene lugar en tiempo de ejecución, lo que se logra es adaptación dinámica de aplicaciones.

En este capítulo veremos las aportaciones que puede suponer el uso de la reflexión para conseguir los objetivos del sistema propuesto. Identificaremos la instrumentación de código como un mecanismo que nos permitirá conseguir el nivel de reflexión necesario en la plataforma .NET (que es sobre la que desarrollamos el sistema) y decidiremos cuándo realizar la instrumentación, a qué instrucciones afecta y qué es lo que se va a añadir en concreto.

También veremos qué otro código se debe inyectar en la aplicación para que pueda interactuar con el resto del sistema, cumpliendo con los requisitos que habíamos establecido para el sistema de no imponer ninguna condición a las aplicaciones que vayan a ser adaptadas (requisito 2.3.2) y de no ser necesario disponer del código fuente de la aplicación (requisito 2.3.1). Todas estas tareas de instrumentación de código las lleva a cabo un elemento del sistema que hemos llamado *JoinPoint Injector* (JPI).

12.1 Aportaciones de la Instrumentación de Código

Diversos autores han identificado la reflexión como una técnica adecuada para conseguir aplicaciones adaptables que pueden cumplir el paradigma del DSOA [Sullivan01][Bencomo05][Tanter04][Ortin04][Chitchyan04]. Estos autores identifican la reflexión estructural y la computacional como necesarias para conseguir los objetivos. La plataforma .NET, sobre la que desarrollamos nuestro sistema, ofrece únicamente características reflexivas de introspección, por lo que por sí misma no es suficiente para lograr los objetivos perseguidos.

Para solucionar este problema, existe la posibilidad de modificar la máquina virtual añadiéndole soporte nativo para la reflexión estructural [Ortin05] [Redondo06] [Redondo07], pero esta aproximación choca con los requisitos impuestos en este sistema (2.2.4 Portabilidad del Sistema – Sistema Estándar) por lo que queda descartada.

Otra posibilidad es hacer uso de algún lenguaje de programación que soporte este tipo de reflexión (como podrían ser Python o Ruby, vistos en el punto 8.2), pero esto

implicaría una dependencia del lenguaje, que invalidaría el requisito 2.1.2 (Independencia del Lenguaje), por lo que también queda descartada como opción.

Mediante la instrumentación de código y, gracias a la introspección que ofrece la plataforma .NET, podemos trabajar con las instrucciones de un programa como si fuesen datos. Podemos modificar estos datos y posteriormente volver a generar una representación que sea comprendida por la máquina virtual, es decir, código. Mediante este proceso podemos modificar la estructura del programa y, añadiendo las instrucciones adecuadas, podemos conseguir que en ejecución se tenga una simulación de un MOP que permita modificar la semántica de la invocación a métodos, su ejecución, el acceso a atributos, tratamiento de excepciones, etc. Es decir, mediante la instrumentación de código, haciendo uso de la introspección, podemos obtener la simulación de un MOP que ofrezca un cierto grado de reflexión computacional, dotando al sistema de las capacidades necesarias para satisfacer los requisitos impuestos en el capítulo 2.

12.2 Elección del Momento de la Inyección

Existen varias posibilidades para elegir el momento en el que se debe realizar la instrumentación de código en el entorno de la máquina abstracta de .NET, que es sobre la que vamos a implementar nuestro sistema. A continuación vamos a comentar diversos momentos en los que puede efectuarse la inyección de código, explicando las ventajas e inconvenientes que presentan con respecto a nuestro sistema y por último seleccionaremos uno de ellos.

Una primera posibilidad que utilizan algunos sistemas es la modificación de código en su lenguaje fuente. Esta posibilidad no es apta en nuestro sistema puesto que iría contra varios de los requisitos impuestos como son la independencia del lenguaje (para que el sistema fuese independiente deberíamos ser capaces de modificar programas escritos en cualquier lenguaje, lo que en la práctica es imposible) o la adaptación de código binario.

Otra posibilidad es realizar la instrumentación de código en un momento post compilación, pero previo a la ejecución de la aplicación. Al hacer la adaptación una vez que el programa ha sido compilado eliminamos los inconvenientes anteriores, ya que trabajamos directamente con el código intermedio, independientemente del lenguaje original en el que se hubiese codificado la aplicación y sin la necesidad de disponer el código fuente original. Como inconvenientes de esta posibilidad podemos mencionar que el usuario debe encargarse de lanzar este proceso de transformación (aunque podría realizarse de forma automática inmediatamente después de la compilación), y que el fichero original podría quedar modificado (en el caso de que se sobrescriba) o existirán dos versiones del mismo programa, una adaptada y la otra no (en el caso de que no se sobrescriba el fichero sino que se cree uno nuevo).

Otros sistemas realizan la adaptación en el momento de la carga de las clases en memoria. Al igual que en el caso anterior se trabaja sobre el programa ya compilado a código intermedio, con lo que se obtienen los mismos beneficios. El hecho de realizar la modificación en el momento de la carga tiene como beneficio que el fichero original no es modificado, ya que la instrumentación de código se realiza en memoria, con lo que puede seguir siendo utilizado en su forma original. Como contrapartida, el realizar la adaptación en el momento de la carga de las clases implica que dicho proceso se ve ra-

lentizado, ya que además de realizar las tareas habituales del proceso ahora debe realizar la instrumentación de código de forma simultánea (y lo debe realizar cada vez que se carguen las clases).

Una última posibilidad es realizar la instrumentación durante la ejecución del código del programa. Cuando se va a ejecutar una parte del programa, y se va a invocar al compilador *JIT* para que genere el código correspondiente, el sistema recibe aviso de esta ocurrencia y, en caso de que tenga que ser adaptado, modifica el código que debe compilarse de tal manera que el código generado se corresponde con la adaptación deseada. Al realizar todo el trabajo en memoria el fichero original no es modificado en ningún momento pero, para que en la plataforma .NET el sistema sea advertido de que se va a proceder a realizar una compilación de código, la máquina virtual debe estar ejecutándose en modo depuración o *profiling* con lo que el rendimiento de la máquina virtual se ve fuertemente penalizado (se realice o no adaptación). Además, el proceso de adaptación tiene lugar durante la ejecución de la aplicación, dando lugar a una ejecución más lenta. Otro inconveniente que presenta esta opción es que el conjunto de puntos de enlace que podemos tratar se ve reducido, ya que el sistema no puede controlar eventos como el acceso a un campo, puesto que no es advertido de su ocurrencia. Otra carencia de esta posibilidad es que es necesario que la aplicación sea ampliada con un conjunto de clases que servirán para implementar el *Framework*, y estas clases deben estar desde el primer momento en la aplicación, ya que lo primero que tiene que hacer la aplicación es registrarse en el servidor de aplicaciones.

También se podía implementar esta última solución modificando la implementación de la máquina virtual, de tal manera que cuando se invocase al compilador se hiciesen previamente las adaptaciones necesarias o fuese el propio compilador *JIT* (modificado) el que se encargase de realizarlas. Esta opción queda descartada por el hecho de que trabajaríamos con una máquina virtual no estándar, lo que iría en contra del cumplimiento del requisito 2.2.4, como ya hemos mencionado.

Nuestro sistema va a realizar la adaptación o inyección de código como un proceso post compilación. De esta manera la ejecución de la aplicación no se ve penalizada por el tiempo empleado en la inyección de código. Este proceso puede ser totalmente automatizado. Cuando una aplicación se ejecuta en el sistema, como primer paso es procesada por el JPI, el cual genera una segunda aplicación (con las modificaciones necesarias) que es la que se ejecuta realmente.

12.3 Identificación de las Joinpoint Shadows

Tal y como hemos comentado anteriormente, el sistema se va a desarrollar sobre la plataforma .NET, y la instrumentación de código la haremos sobre código IL, por lo que será en el IL donde debemos identificar las *joinpoint shadows.*

Una *joinpoint shadow* es la sombra que deja un punto de enlace a bajo nivel (en nuestro sistema esto se corresponde con código CIL). Existen puntos de enlace cuya sombra se corresponde con una instrucción y es sencilla de localizar. Por ejemplo, un punto de enlace de tipo invocación a método, a bajo nivel se corresponde con una instrucción de llamada al método (`call, callvirt` y otras). En cambio, existen puntos de enlace cuya sombra no se corresponde con una única instrucción o, incluso, no se corresponde con ninguna instrucción sino con una zona del código. Por ejemplo, un

punto de enlace de tipo ejecución de un método, en código IL se corresponde con la primera instrucción de su código y con la instrucción para abandonar la ejecución, `ret` en IL, que puede aparecer varias veces en el código, o con el hecho de alcanzar el final del código del método (la última instrucción en orden secuencial).

Utilizaremos una herramienta para poder acceder reflectivamente al código de la aplicación y obtener sus instrucciones (se puede encontrar una explicación detallada en el punto 12.4). Cada instrucción viene representada por dos objetos, uno representa su código de operación y otro su operando (por ejemplo, en una instrucción de invocación a un método el código de operación indica que es una llamada, de un tipo específico, y el operando contendrá la signatura del método al que se invoca).

Con esta estructura tenemos tres posibilidades para localizar las sombras de los puntos de enlace:

- Existen sombras que no se corresponden con una instrucción concreta, sino con una zona del código. Por ejemplo en la ejecución de un método el comienzo de la sombra viene marcado por la primera instrucción que se encuentre en el código (sea la que sea), o en el punto de enlace de tratamiento de una excepción el comienzo viene marcado por la primera instrucción del bloque del gestor de la instrucción (el *handler*).
- Otras sombras pueden ser localizadas por el código de operación (o su nombre equivalente). Por ejemplo, en la ejecución de un método la sombra de finalización de la ejecución se corresponde con la instrucción `ret`, o con la última instrucción en orden secuencial del código del método en el caso de que no sea `ret`.
- Otras sombras las podemos localizar por el operando de la instrucción. Por ejemplo, las invocaciones a métodos tienen como operando uno del tipo `InlineMethod`. Obviamente, también es posible localizar estas sombras por todos los posibles códigos de operación que se corresponden con invocaciones pero, en este caso, es más sencillo hacerlo por el tipo de operando.

Cabe destacar que en código IL las propiedades–*properties*– se transforman en dos métodos llamados get_XX y set_XX, donde XX es el nombre de la propiedad, por lo que el tratamiento de las propiedades en código IL es equivalente al de los métodos. A continuación vamos a ir viendo las sombras de cada uno de los puntos de enlace existentes en nuestro sistema.

12.3.1 Ejecución de un Método o Constructor

Los constructores son métodos normales pero que en su definición en IL reciben el nombre `.ctor` o `.cctor`, por lo demás se comportan exactamente igual que cualquier otro método, por lo que la localización de las sombras es idéntica en ambos casos.

Existen dos sombras asociadas con la ejecución de un método o constructor, la primera es el punto de entrada del método, es decir, su primera instrucción. Si necesitamos realizar instrumentación del tipo "antes de la ejecución de un método" sería en este punto, justo antes de la primera instrucción del método, donde debería realizarse, aña-

diendo el código necesario, de tal forma que la ejecución comience por el código añadido en vez del original.

La otra sombra asociada a la ejecución es la salida del método, es decir, su finalización. De un método se puede salir ordenadamente de dos formas: 1) alcanzando la última instrucción del código, con lo que se abandona la ejecución por no haber nada más que ejecutar o 2) por encontrar la instrucción `ret`, que implica el abandonar la ejecución del método. En un método puede haber varias instrucciones `ret`. Si necesitásemos realizar instrumentación del tipo "después de la ejecución de un método" sería justo antes de la instrucción `ret` donde se debería añadir el código correspondiente para que se ejecutase antes de la instrucción o, en el caso de no haber instrucción `ret`, si no que se abandona la ejecución por haber llegado a la última instrucción, será después de esta última instrucción donde deba ir el código añadido.

12.3.2 Invocación a un Método o Constructor

Como hemos comentado en el punto anterior, los constructores son métodos con un nombre especial, por lo que se puede detectar de la misma manera la sombra de la invocación a un método que la de la invocación a un constructor.

Existen numerosas instrucciones de invocación en IL, pero todas ellas comparten en común que su operando es del tipo `InlineMethod`, es decir, representa la signatura del método o constructor al que invocan, y éstas son las únicas instrucciones que tienen ese operando, por lo que podemos utilizar el operando para localizar el punto exacto donde se realiza la invocación.

En IL la invocación a un método o constructor se produce mediante una única instrucción, con lo que ésa será la sombra asociada al punto de enlace correspondiente. De todos modos, será necesario tener en cuenta que, si un método o constructor recibe parámetros, éstos se encontrarán cargados en la pila justo antes de producirse la invocación. Esto hay que tenerlo en cuenta a la hora de añadir el código correspondiente a la instrumentación, puesto que será necesario conocer el valor de esos parámetros durante la ejecución.

La adición de código se realizará justo antes o justo después (dependiendo del tipo de instrumentación que estemos haciendo) de la instrucción de invocación, pero, como veremos más adelante, incluso en el caso de instrumentar para el tipo de punto de enlace "después de la invocación a un método" será necesario haber introducido código antes de la instrucción de invocación con el fin de obtener cierta información como los parámetros que se le pasan al método.

12.3.3 Acceso a un Campo

En IL el acceso a un campo, ya sea de lectura o escritura, se produce a través de la pila. En el caso de acceso de lectura, el campo (su valor) se carga en la pila para ser accedido posteriormente, en el caso de acceso para escritura el valor a escribir ya se encuentra en la pila y se almacena en el campo.

Las instrucciones que se corresponden con el acceso de lectura a un campo son `ldsfld` y `ldfld` (dependiendo de si el campo es estático o no). Estas instrucciones

serán pues las sombras del punto de enlace y las adiciones de código se deben realizar justo antes o justo después de la instrucción, dependiendo del tiempo para el que estemos instrumentando el punto de enlace.

De igual forma, las instrucciones que se corresponden con el acceso de escritura a un campo son `stsfld` y `stfld` (dependiendo de si el campo es estático o no). Estas instrucciones serán pues las sombras del punto de enlace y las adiciones de código se deben realizar justo antes o justo después de la instrucción, dependiendo del tiempo para el que estemos instrumentando el punto de enlace.

12.3.4 Tratamiento de una Excepción

En IL existen dos formas de expresar el control y manejo de excepciones. La primera forma, basada en el uso de las etiquetas del código, consiste en una instrucción que indica: 1) el conjunto de instrucciones que forman parte del bloque *try* mediante la etiqueta de la primera y última instrucción de dicho bloque, 2) el tipo de excepción que trata y 3) el conjunto de instrucciones que forman parte del manejador (*handler*) de la excepción (el bloque *catch*) mediante la etiqueta de la primera y última instrucción del bloque.

Otra forma de expresar lo mismo es indicando los bloques directamente en el ámbito del código mediante el uso de sentencias de bloques *try* y *catch* en el código al que afectan. Esta forma de expresarlo está orientada a la legibilidad por parte del desarrollador, pero es menos potente.

A continuación mostramos las dos formas de expresar el mismo tratamiento de excepciones, la primera se basa en el uso de etiquetas:

```
BeginTry:
   // Código protegido
   .....
   leave KeepGoing
BeginHandler:
   // The exception handler
   .....
   leave KeepGoing
KeepGoing:
   .....
   ret
   .try BeginTry to BeginHandler catch [mscorlib]System.Exception
      handler BeginHandler to KeepGoing
```

Esta segunda forma de representarlo es equivalente a la anterior, pero para un desarrollador es más fácil de entender por tener representados los bloques en los lugares adecuados del método, y no al final del mismo.

```
.try {
   // Código protegido
   .....
   leave KeepGoing
}
catch [mscorlib]System.Exception {
   // The exception handler
   .....
   leave KeepGoing
}
```

```
   .....
KeepGoing:
   .....
```

Las sombras originadas por este punto de enlace son dos. La primera es el punto de entrada del bloque *catch*, es decir la primera instrucción. Si se necesitase realizar instrumentación sería en este punto donde se añadiría código de tal manera que se ejecutase antes de la primera instrucción del bloque.

La otra sombra se corresponde con el punto de salida del bloque, y se identifica por la instrucción de salida, `leave` o su variante corta `leave.s`, o por la llegada a la última instrucción del bloque sin haber encontrado la instrucción de salida. En ambos casos se procedería de igual forma que en la ejecución de métodos.

12.4 Herramientas Utilizadas

Para realizar la instrumentación de código existen multitud de herramientas que permiten extraer el código de un programa ya compilado, así como más información asociada al mismo (obtenida de los metadatos contenidos en el ensamblado). Una vez extraído el código la mayoría de los sistemas ofrecen una representación interna del programa (normalmente mediante un AST –*Abstract Syntax Tree*–) que puede ser modificada y posteriormente, a partir de esta representación modificada, se puede generar otro fichero con el código intermedio modificado.

Hemos valorado la utilización de diversas herramientas, entre ellas CLIFileReader [CLIFileReader], RAIL [Rail], CECIL [Cecil] o Absil [Absil] para realizar la instrumentación de código. Finalmente la herramienta seleccionada ha sido ILReader [ILReader], la cual permite desensamblar un programa y acceder de forma directa, instrucción por instrucción, al código IL que lo forma, así como a la metainformación asociada.

Debido a que todo el proceso lo hacemos a bajo nivel (al nivel de instrucciones de IL), no necesitamos crear una representación en memoria del programa para luego poder modificarla, simplemente con poder acceder al código de cada una de las instrucciones y su metainformación asociada es suficiente para poder identificar las *joinpoint shadows* y, al trabajar en un momento anterior a la ejecución de la aplicación, podemos generar un fichero con el código IL original más el código añadido por el proceso de instrumentación que será procesado por el ensamblador generando un nuevo ejecutable, ya modificado.

12.5 Inyección del MOP

La forma de trabajar del JPI consiste en acceder al fichero a instrumentar, obtener toda su información reflectiva (clases, interfaces, métodos, atributos, etc.) de forma organizada y jerarquizada y poder acceder a cada instrucción del código IL. Toda esta información se va volcando a un fichero de forma secuencial y, cada vez que se localiza una *joinpoint shadow*, se inyecta el código correspondiente para implementar el MOP (se genera el código y se escribe en el fichero). Una vez creado el fichero completo, con todo el código añadido, se ensambla mediante el ensamblador disponible en la plataforma y se obtiene un nuevo fichero ejecutable con el código modificado, que es el que

se ejecutará en nuestro sistema en sustitución del original. Este proceso está representado en la Figura 46.

Como ya hemos comentado anteriormente, no es necesario crear en memoria ninguna estructura que represente al programa a modificar, es suficiente con poder acceder de forma reflectiva a cada uno de los tipos contenidos en el programa (clases, interfaces, etc.) y a su implementación (sus instrucciones).

El sistema irá descendiendo a través de la jerarquía de tipos existentes en el programa hasta llegar a la implementación de los métodos (instrucciones en IL), que es donde se podrán localizar las sombras (*shadow*) de los puntos de enlace. Una vez detectada una sombra, se procede a añadir el código necesario para implementar el MOP. En el siguiente pseudocódigo se puede ver de forma general el código que se debe añadir en una sombra para soportar el tiempo *before* de un punto de enlace cualquiera.

```
//comienza código inyectado
Comprobar si hay aspectos que hayan solicitado el punto de
         enlace en ese tiempo
SI HAY
      Realizar las tareas propias del punto de enlace (recopilar la
            información necesaria)
      Para cada aspecto que lo haya solicitado
            Invocar al aspecto, pasándole la información necesaria
NO HAY
      Se continúa
//finaliza código inyectado
Sombra del punto de enlace
```

La primera tarea que efectúa el código inyectado es comprobar si algún aspecto ha solicitado ser informado de la ejecución de ese punto de enlace en ese tiempo concreto (el tiempo puede ser *before, after* o *around* dependiendo de los casos). Para realizar esta comprobación se hace uso de las funciones definidas en la interfaz `IJoinPoint`, perteneciente al Framework (que veremos en detalle en el capítulo 14).

Si no existen aspectos que hayan solicitado ser informados de la ejecución del punto de enlace la ejecución continúa de forma normal. Si existen aspectos que hayan solicitado ser informados se llevan a cabo dos acciones: 1) se recopila toda la información reflectiva necesaria para enviársela al aspecto (esta información varía de unos puntos de enlace a otros) y 2) para cada aspecto que lo haya solicitado, se le informa de la ejecución del punto de enlace enviándole la información recopilada. Para poder informar al aspecto de la ejecución del punto de enlace se hace uso de la interfaz de aspectos correspondiente (IMethodCall, IPropertyFieldAccess, IMethodExceution, etc. los veremos en detalle en el capítulo 14), definido en el Framework y que los aspectos deben implementar.

La información reflectiva que se recopila tiene una parte común, que consiste en la información sobre el punto de enlace (de qué tipo es, el tiempo –*before, after o around*–, dónde se ha ubicado –*namespace*, clase, etc.–) y una parte que varía de un punto de enlace a otro. Por ejemplo, en la invocación a un método es necesario conocer los parámetros –el tipo, nombre y valor de cada uno– que se le pasan (si es que se le pasa alguno) y, en el acceso de escritura a un campo, es necesario conocer cuál es el valor que se va a asignar (o se ha asignado, dependiendo del tiempo si es antes o después– *before o after*) al campo.

A continuación y a modo de ejemplo vamos a mostrar el pseudocódigo que se debe inyectar en un punto de enlace de tipo "invocación a un método" en los tres tiempos posibles (*before, after y around)*.

El JPI analiza instrucción por instrucción en busca de *joinpoint shadows* y cuando localiza una es cuando inyecta el código correspondiente. En el siguiente fragmento de código podemos ver en negrita la detección de una *joinpoint shadow*.

```
....
ldstr "cadena de prueba"
call  void [mscorlib]System.Console::WriteLine(string )
....
```

Al localizar una instrucción que se corresponde con una sombra el JPI añade el código correspondiente al punto de enlace en cuestión. Por ejemplo, en el siguiente pseudocódigo podemos ver (a muy alto nivel) la inyección del tiempo *before* para una invocación a un método.

```
....
//comienza el código inyectado
1 Comprueba la existencia de aspectos
2 Si no hay aspectos salta a Final_inyección
3 Descarga el contenido de la pila a un array
4 Construye un array con los parámetros del método (nombre, tipo y
    valor)
5 Reconstruye la pila
6 Para cada aspecto
7     Invoca al aspecto (IMethodCall.exec) pasándole la información
8 Final_inyección:
//final de código inyectado
call  tipo_de_retorno método(parámetros…)
....
```

A continuación explicaremos de forma breve el pseudocódigo presentado. El primer paso consiste en comprobar si existen aspectos que hayan solicitado ser informados de la ejecución de este punto de enlace. Para ello se utiliza el método `getAspects`, de la interfaz `IJoinPoint`, que devuelve un array ordenado con los aspectos que han solicitado ser informados de la ejecución del punto de enlace. Caso de que no haya aspectos se salta directamente al final del código inyectado.

En el paso 3 se recoge el contenido de la pila –*stack*– y se lleva a un array para almacenarlo temporalmente. Esto lo hacemos para preservar su contenido a través de las operaciones que realicemos (el proceso de lectura de la pila es destructivo, puesto que no se puede consultar el elemento que se encuentra en la cima de la pila sin extraerlo de la misma, ni se puede consultar otro elemento distinto del que se encuentra en la cima). En el paso 4 se construye un array con los parámetros del método, incluyendo para cada uno su tipo, nombre y el valor que se pasa. Este array se enviará al aspecto cuando sea invocado junto con el resto de la información necesaria.

El paso 5 consiste en reconstruir la pila para que cuando se ejecute la instrucción original encuentre los mismos datos que había inicialmente. Para realizar esto se utiliza el array que se rellenó en el paso 3.

El paso 6 es un bucle, que se ejecuta tantas veces como aspectos haya, que engloba al paso 7, la invocación al aspecto. En este paso se invoca al aspecto por medio

del método `exec` de la interfaz IMethodCall, que tiene que haber implementado el aspecto. En esta llamada se envía toda la información posible acerca del punto de enlace: su identificación exacta (tipo, *namespace*, clase, método, tipo de retorno, etc.), su tiempo (*before* en este caso), y los parámetros que recibirá el método, y que habíamos copiado, junto a información que los describe (tipo y nombre), en un array creado al efecto en el paso 4.

Una vez se ha invocado a todos los aspectos que lo hubiesen solicitado, la ejecución continúa con la invocación al método de forma normal.

A continuación vamos a mostrar, en pseudocódigo, la inyección en el punto de enlace "invocación a método" como en el caso anterior, pero en el tiempo *after*. En este caso es necesario inyectar código antes y después de la propia invocación al método.

```
....
//comienza el código inyectado parte 1
1 Comprueba la existencia de aspectos
2 Si no hay aspectos salta a Final_inyección
3 Descarga el contenido de la pila a un array
4 Reconstruye la pila
5 Final_inyección:
//final de código inyectado parte 1
call  tipo_de_retorno método(parámetros…)
//comienza el código inyectado parte 2
6 Comprueba la existencia de aspectos
7 Si no hay aspectos salta a Final_inyección2
8 Si existe se recupera el valor de retorno (sin eliminar de la pila)
9 Construye el array con los parámetros del método (nombre, tipo y
      valor)
10 Para cada aspecto
11    Invoca al aspecto (IMethodCall.exec) pasándole la información
12 Final_inyección2:
....
```

Es conveniente hacer notar que se ha inyectado código antes de la llamada y después de la misma. Esto es así porque la parte previa a la llamada sirve para recuperar información que se va a utilizar en la parte que se ejecuta tras la llamada (y no está disponible una vez se ha realizado ésta).

Los pasos del 1 al 5 son muy similares a los pasos del caso anterior (tiempo *before*). Como se puede ver, la diferencia consiste en que en este caso no se ejecuta bucle alguno llamando a los aspectos, y tampoco se crea el array con los parámetros. Lo único que se ha hecho ha sido recabar información que será utilizada posteriormente.

El paso 6, que ya se realiza después de haber hecho la invocación al método, comprueba si hay aspectos para, en el caso de que no los hubiese, paso 7, saltar al final del código inyectado, y continuar con la ejecución normal del programa.

El paso 8, al que se llega únicamente si hay aspectos, recupera de la pila el valor devuelto por la invocación al método, caso de que lo haya. Se debe recuperar sin eliminarlo de la pila, para que la ejecución posterior del resto del programa no se vea alterada. Este valor se enviará también al aspecto cuando se le invoque.

El paso 9 consiste en construir el array de parámetros, de forma similar a como se hizo en el caso del tiempo *before*. La razón de hacer en este momento la captura de esta información es que los parámetros pueden haber sido modificados por el método al que se invocó (en el caso de parámetros pasados por referencia), por lo que es necesario obtener sus valores después de la ejecución del método y no antes.

El paso 10, junto con el 11, es el bucle de invocación a los aspectos que hayan solicitado ser informados de la ejecución del punto de enlace. En este caso se envía la misma información que en el caso anterior más lo que haya retornado el método que se ejecutó (si es que devolvió algo).

Una vez realizada la invocación a todos los aspectos que lo hubiesen solicitado, la ejecución de la aplicación continúa de la forma original.

Para mostrar un caso de cada tiempo, a continuación vamos a mostrar el pseudocódigo del mismo punto de enlace en el tiempo *around*.

```
....
//comienza el código inyectado
1 Comprueba la existencia de aspectos
2 Si no hay aspectos salta a Final_inyección
3 Descarga el contenido de la pila a un array
4 Construye un array con los parámetros del método (nombre, tipo y
    valor)
5 Invoca al aspecto (IMethodCall.exec) pasándole la información
6 Salta a parte2_inyección
7 Final_inyección:
//final de código inyectado
call  tipo_de_retorno método(parámetros…)
8 salta a Final_inyección2
9 parte2_inyección:
10 Meter en la pila lo que haya devuelto el aspecto en el formato
      adecuado
11 Final_inyección2:
....
```

Como se puede ver, la parte que se inyecta antes de la invocación al método es muy similar al código inyectado en el caso del tiempo *before*, las diferencias consisten en que la pila no se reconstruye, ya que no se va a realizar la llamada al método y cuando se continúe la ejecución la pila debe estar limpia. La otra diferencia se encuentra en la llamada al aspecto: en este caso sólo puede haber un aspecto, por lo que no se realiza ningún bucle.

Si en la parte inicial se realiza la llamada al aspecto, en la parte final es necesario recuperar lo que haya devuelto el aspecto e introducirlo en la pila (con el formato adecuado a lo que se espera), para que la ejecución de la aplicación continúe con el valor devuelto por el aspecto en vez del que hubiese devuelto el método en caso de que se le hubiese invocado.

En el pseudocódigo que se ha mostrado no se contempla la posibilidad de ejecutar el método una vez que hay algún aspecto del tipo *around*, es decir, no se contempla realizar algo similar al `proceed` de AspectJ. Para poder implementarlo sería necesario cambiar levemente el código a inyectar haciendo que el aspecto indicase si se debe proceder a la ejecución del punto de enlace o no y, en base a lo que hubiese indicado se

procedería de una forma u otra. No se ha mostrado aquí con el fin de lograr una mayor claridad del ejemplo. Por el mismo motivo el pseudocódigo mostrado tampoco contempla la posibilidad de que haya más de un aspecto en el mismo punto de enlace para el tiempo *around*.

Los pseudocódigos que hemos visto muestran las instrucciones que se han de insertar en el caso de inyectar únicamente el tiempo correspondiente. En caso de inyectar para el mismo punto de enlace código para los tres tiempos es necesario hacerlo en el orden adecuado, y se podría reutilizar la información recabada en unos para los otros. A continuación se muestra cómo quedaría el mismo punto de enlace con el código inyectado de los tiempos *before, after y around* de forma simultánea.

```
....
Código del tiempo Around, parte 1
Código del tiempo Before
Código del tiempo After, parte 1
call //Invocación al método
Código del tiempo After, parte 2
Código del tiempo Around, parte 2
....
```

12.6 Inyección de Código Adicional y Otras Acciones

Si la inyección de código por parte del JPI se limitase a la adición del MOP reflectivo, la aplicación no sabría cómo interactuar con el resto del sistema, por lo tanto es necesario que se añada más código que permita a la aplicación interactuar con él.

Además, de forma simultánea al proceso de detección de sombras e inyección de código, se construye una tabla con todos los puntos de enlace detectados e inyectados. La tabla contiene información acerca del tipo de punto de enlace, su ubicación, nombre, tiempo, etc. Esta tabla se usa posteriormente durante el proceso de selección de los puntos de enlace a partir de los puntos de corte suministrados por un aspecto. Cuando un aspecto solicita realizar la adaptación de una aplicación, suministra uno o varios puntos de corte, los cuales se validan con los puntos de enlace contenidos en la tabla (que son los existentes en la aplicación), comprobando si afectan a cada uno de ellos. La utilización en ejecución de esta tabla la realiza la implementación de la interfaz `IJoinPoint`, contenido en el Framework, que se añade a la aplicación durante su proceso de inyección.

12.6.1 Comentarios al Código

Durante el proceso de inyección de código se incluyen comentarios descriptivos de las acciones que realiza cada parte del código inyectado. Esto se realiza debido a que el código inyectado puede dificultar la comprensión del código global de la aplicación, por parte de un desarrollador, a la hora de realizar una depuración del programa tejido, ya que es código que no ha sido generado por él. De esta manera, al hacer una depuración en código intermedio, las partes inyectadas contienen explicaciones de las acciones que llevan a cabo, facilitando así la comprensión de las mismas por parte del desarrollador.

12.6.2 Remoting y Código de Registro

Puesto que la aplicación debe establecer comunicación bidireccional, tanto con el Servidor de Aplicaciones como con los aspectos que estén interesados en adaptar su comportamiento, es necesario que implemente mecanismos de comunicación que lo permitan.

Tal y como veremos en el punto 13.4, el mecanismo de comunicación entre procesos que utilizaremos en el sistema es Remoting. Para que una aplicación pueda hacer uso de Remoting es necesario incluir al menos una librería en la aplicación (`System.Runtime.Remoting`), por lo esta librería debe ser añadida al código del programa (lo que se añade es una referencia a la librería, no la librería en sí).

Nada más comenzar la ejecución de la aplicación, ésta debe registrarse en el Servidor (que veremos en detalle en el capítulo 13), el cual actúa como un servidor de Remoting. Para poder actuar como un cliente de Remoting es necesario crear un canal de comunicación y obtener un objeto publicado previamente por el servidor. Una vez que se ha obtenido el objeto del Servidor, a través del cual podemos invocar a los métodos definidos en la interfaz `IServer` (ver 13.2), la aplicación debe registrarse en el Servidor, enviando su identificador único, así como una referencia a sí misma de tal manera que el Servidor pueda invocarla posteriormente (para, por ejemplo, activar o desactivar puntos de enlace en función de los puntos de corte solicitados por un aspecto).

Para que la aplicación pueda recibir invocaciones a través del canal que ha abierto, es decir, que pueda actuar también como servidor de Remoting recibiendo peticiones de un cliente, es necesario que el canal de comunicación se haya abierto en modo cliente y servidor. En nuestro caso la aplicación no va a publicar nada de forma general al exterior, sino que enviará el objeto a utilizar por otras aplicaciones a través del canal Remoting actuando como cliente, para que posteriormente estas aplicaciones que recibieron el objeto (y sólo éstas) puedan invocarlo.

Por lo tanto, todo este código (el de la parte cliente y el de la parte servidor) se añadirá justo delante de la primera instrucción de la aplicación, de tal manera que sea lo primero en ejecutarse cuando la aplicación comience su funcionamiento. De esta manera, lo que hace la aplicación en primer lugar es abrir los canales de comunicación, permitiéndole realizar invocaciones y que otras aplicaciones la invoquen a ella y realiza el registro de la aplicación en el Servidor. Una vez hecho esto continúa con la ejecución normal de la aplicación.

12.6.3 Interfaces

La aplicación a ser adaptada debe ofrecer la posibilidad de acceder a las implementaciones de dos interfaces (`IJoinPoint` e `IReflection`) que son la parte del Framework que debe implementar. Estas implementaciones se encuentran en la librería Interfaces, proporcionada por el sistema, que también debe añadirse a la aplicación (al igual que en el caso de Remoting, lo que se añade es una referencia a la misma, de tal forma que al realizar el ensamblado del fichero generado se enlace el código y pueda hacerse uso de lo implementado).

La interfaz `IJoinPoint` (su implementación) es utilizada en el código añadido en los puntos de enlace para comprobar si algún aspecto ha solicitado ser informado de la ejecución del mismo. Además, es utilizada por el Servidor para registrar y desregistrar aspectos en los puntos de enlace de la aplicación.

La interfaz `IReflection` permite que desde otra aplicación (el aspecto) se acceda a la estructura de la aplicación para conocerla (introspección) e invocar a sus miembros (métodos, atributos, etc.).

Una descripción detallada de ambas interfaces puede verse en el capítulo 14.

12.7 Inyección Selectiva

Según hemos planteado hasta el momento, cuando se instrumenta un programa se localizan todos sus puntos de enlace, en concreto sus sombras en IL, y se añade el código necesario para ofrecer el MOP que permitirá la adaptación dinámica en cada uno de esos puntos. Gracias a esto, no es necesario conocer las posibles adaptaciones que se le van a realizar a un programa antes de que comience la ejecución del mismo. El programa podrá ser adaptado durante su ejecución, sin necesidad de ser detenido, por cualquier aspecto (que tenga los permisos adecuados) en cualquier punto de enlace de la aplicación. Es decir, se consigue con ello una verdadera adaptación dinámica de la aplicación.

Existen circunstancias en las que conocemos dónde se va a producir la adaptación antes de la ejecución de la aplicación pero aún así se necesita el dinamismo. Una posibilidad es que no se conozca la adaptación exacta que se va a realizar, ya que si se conociese se utilizaría tejido estático, más eficiente. Otra posibilidad es que la adaptación precise ser dinámica, ya sea porque precise activarse o desactivarse dinámicamente, o porque se pueda necesitar cambiar de forma dinámica. Un ejemplo de lo último puede ser un servidor Web en el que es necesario cambiar la política de precarga en memoria de forma dinámica y ajustada a las particularidades del contexto, no conocidas de antemano [Segura–Devillechaise03].

Ante una situación de estas características podría ser beneficioso no inyectar el MOP completo en todos los puntos de enlace de la aplicación sino únicamente en aquellos puntos donde se fuese a necesitar. Con esto, el sistema se ahorraría el coste de soportar el MOP (por pequeño que sea) en puntos de enlace que no necesitan ser adaptados. Esto se puede considerar como un modo particular de optimización. Sólo se utilizaría para optimizar zonas de código que sabemos con certeza que no queremos adaptar dinámicamente.

Debido a esto, hemos diseñado un mecanismo para realizar una inyección selectiva del MOP, permitiendo al usuario del sistema controlar qué puntos de enlace y en qué tiempo deben ser inyectados (aumentados con el MOP) y qué puntos no. El usuario debe indicarle al JPI (*JoinPoint Injector*) los puntos de enlace en los que está interesado y los tiempos concretos. Si no le indica nada el sistema asume que debe inyectar el MOP para todos los puntos de enlace que se detecten en el programa.

Puesto que necesitamos un mecanismo para seleccionar puntos de enlace y en el sistema ya se cuenta con uno, el lenguaje de definición de puntos de corte, lo lógico es

utilizar el mismo lenguaje para que el usuario pueda seleccionar los puntos de enlace que quiere que sean inyectados con el MOP.

Por lo tanto, el funcionamiento del JPI será el siguiente: por defecto, al instrumentar una aplicación se localizarán todos los posibles puntos de enlace y en cada uno de ellos se inyectará el código adecuado para implementar el MOP. En caso de que el usuario especifique, mediante un fichero XML, una serie de puntos de corte, que a su vez seleccionan o afectan a una serie de puntos de enlace, el sistema detectará todos los puntos de enlace de la aplicación y para cada uno de ellos comprobará si se ve afectado (es seleccionado) por alguno de los puntos de corte existentes. En caso afirmativo procederá a realizar la inyección de código en el punto de enlace, y en caso negativo (el punto de enlace no es seleccionado por ninguno de los puntos de corte suministrados por el usuario) no se realiza inyección de código alguno en el punto de enlace.

Obviamente, esto sólo tiene utilidad en los casos mencionados en los que se tiene certeza de que no va a ser necesario adaptar a la aplicación de forma dinámica en cualquier punto de enlace, sino sólo en una serie de ellos que son conocidos.

En el siguiente punto comentaremos cómo se realiza el procesado del fichero de puntos de corte.

12.8 Procesado del Fichero XML de Puntos de Corte

Como ya hemos comentado anteriormente, los puntos de corte se expresan en un lenguaje basado en XML, por lo tanto, es necesario diseñar un intérprete de este lenguaje para poder procesar los puntos de corte y determinar a qué puntos de enlace afectan.

Haciendo uso del patrón *Interpreter* [Gamma94] definimos una representación de la gramática del lenguaje y un intérprete que use dicha representación para interpretar las sentencias del mismo. En concreto, lo que nos interesa es determinar si para un determinado punto de enlace alguno de los puntos de corte expresados en el archivo lo satisface, es decir, lo selecciona.

El proceso a seguir es el siguiente:

1. Inicialmente se procesa el fichero XML recorriendo cada uno de sus elementos de forma recursiva descendente y creando los elementos correspondientes que representan un árbol sintáctico abstracto (AST) de las sentencias contenidas en el fichero.
2. Una vez construido el árbol se invoca a la operación que interpreta el árbol. En este caso la operación se denomina `toBeInjected` y recibe una serie de parámetros que identifican a un punto de enlace (*namespace*, clase, nombre, tipo del punto de enlace –*methodcall, constructorexecution,* etc.–, tiempo, etc.). Esta operación determina si alguna de las sentencias contenidas en el árbol satisface el punto de enlace o no.

En el JPI se realiza el primer paso, la construcción del árbol, al comienzo del procesamiento de la aplicación y, posteriormente, cada vez que se detecta un posible punto de enlace (mediante su *joinpoint shadow*), se invoca a `toBeInjected`, pasándole la información necesaria para definir completamente el punto de enlace, con el fin

de determinar si es necesario realizar la inyección de código o no, y en base al resultado se procede en un sentido u otro.

Este mismo intérprete se utiliza también para activar y desactivar los puntos de enlace en base a las solicitudes realizadas por los aspectos. En este caso, el primer paso, la construcción del AST, se realiza en el Servidor (ver capítulo 13), y el segundo paso, la comprobación de qué puntos de enlace son seleccionados, se realiza por parte de la implementación del interfaz `IJoinPoint` (ver 14.2) que se ha añadido a la aplicación.

Para poder realizar esto existe otra operación que interpreta el árbol creado, denominada `inject`, la cual recibe una lista con todos los puntos de enlace existentes en la aplicación, un valor que le indica si se debe activar o desactivar, y una referencia al aspecto que ha solicitado la adaptación. Esta operación comprueba, para cada punto de enlace contenido en la lista, si alguna de las sentencias contenidas en el árbol lo satisface y, en caso de ser así, y dependiendo de si se ha solicitado activar o desactivar, se procede a añadir o eliminar la referencia al aspecto en el punto de enlace concreto de la lista recibida.

En el capítulo 15 se puede ver un diseño más detallado de la implementación realizada.

CAPÍTULO 13
SERVIDOR DE APLICACIONES

En este capítulo presentamos el Servidor del sistema, que es el punto central del mismo y permite que las aplicaciones se registren para poder ser adaptadas, y los aspectos las adapten.

En las secciones siguientes mostraremos las funciones del Servidor, la forma de hacerlas accesibles hacia los demás actores del sistema, cuestiones de seguridad que debe contemplar y el mecanismo de comunicación entre procesos que se usará por parte de todos los elementos del sistema.

13.1 Funciones del Servidor

El Servidor es el centro neurálgico del sistema, a través del cual se comunicarán el resto de actores (aplicaciones y aspectos). En la Figura 52 se muestran las funciones del Servidor,

Tal y como comentamos en capítulo 9, el funcionamiento del sistema implica que, cuando una aplicación comienza su ejecución, debe avisar de ello al Servidor (se registra) para que éste tenga conocimiento de la aplicación y permita que se la adapte. De igual modo, cuando una aplicación finaliza su ejecución debe informar también de ello al Servidor (se da de baja), con el fin de que tenga conocimiento de que la aplicación deja de estar disponible para ser adaptada.

El Servidor mantiene una lista actualizada de las aplicaciones que se están ejecutando en el sistema y que, por lo tanto, pueden ser adaptadas por parte de los aspectos que lo soliciten y tengan permiso para hacerlo.

Cuando un aspecto necesita adaptar el comportamiento de una aplicación debe solicitárselo al Servidor, para ello le indica a quién quiere adaptar (la aplicación) y en qué puntos desea hacerlo (los puntos de enlace, que son seleccionados a través de los puntos de corte que vienen expresados en un fichero XML, que se adjunta a la solicitud). El Servidor realiza las comprobaciones de seguridad necesarias para verificar que el aspecto puede hacer lo que solicita y, en caso afirmativo, procede a realizar lo solicitado. El Servidor procesa el fichero XML recibido y crea una representación en memoria de los puntos de corte en él contenidos (ver el punto 12.8 para una explicación detallada del proceso). Esta representación en memoria se envía a la aplicación, a través de la implementación de la interfaz `IJoinPoint` inyectada en ella (ver el punto 14.2

para encontrar una explicación de la interfaz), y es la propia aplicación la que registra al aspecto en los puntos de enlace que son seleccionados por los puntos de corte que recibe. La posterior comunicación entre la aplicación y el aspecto tiene lugar sin la mediación del Servidor de aplicaciones.

Cuando un aspecto desea eliminar alguno (o todos) de los puntos de enlace en los que está registrado en una aplicación a la que está adaptando, se lo solicita al Servidor del mismo modo que en el proceso de registro, le indica la aplicación sobre la que desea realizar la acción y los puntos de corte que seleccionarán los puntos de enlace de los cuales quiere desregistrarse (mediante un fichero XML). El proceso es el mismo que en el caso anterior, el Servidor procesa el fichero y crea una representación en memoria del mismo, esta representación en memoria se envía a la aplicación que es la que se encarga de eliminar el aspecto de los puntos de enlace seleccionados.

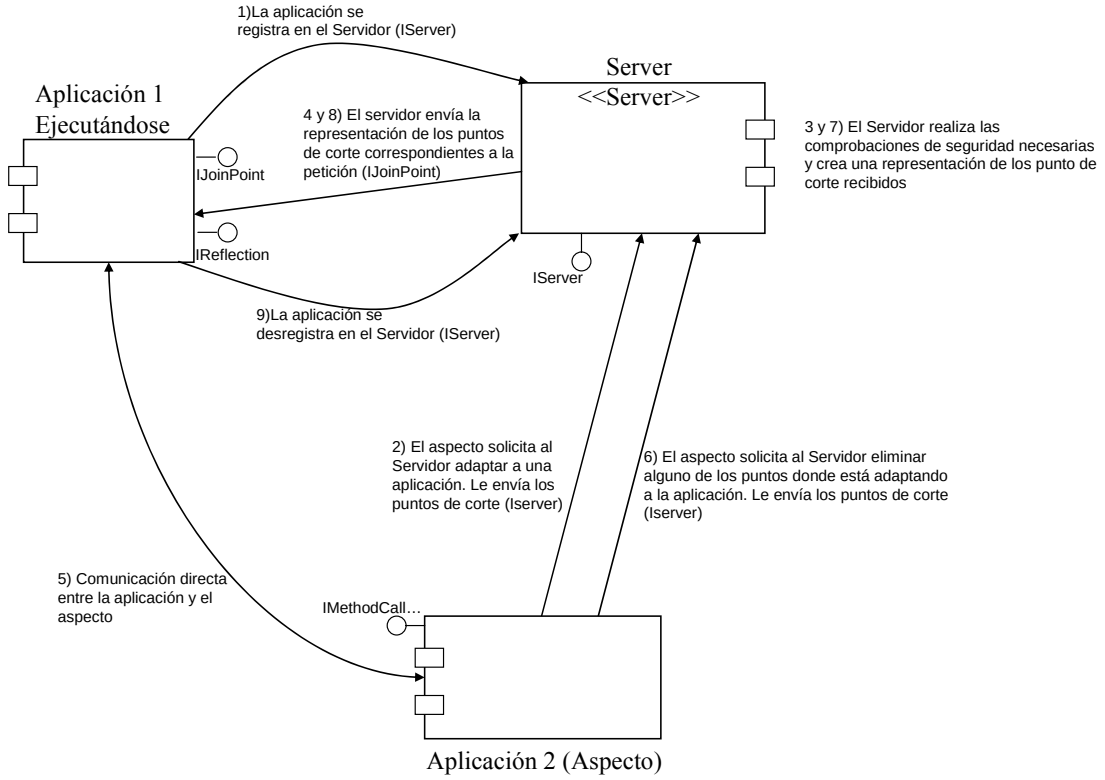

Figura 52: Funciones del Servidor

Por lo tanto, como funciones principales el Servidor debe permitir que las aplicaciones se den de alta y de baja en el sistema, debe llevar un registro de las aplicaciones que están disponibles para ser adaptadas, y debe permitir que los aspectos soliciten adaptar una aplicación, modificar la adaptación que están haciendo y también que dejen de adaptarla.

13.2 Interfaz IServer

Para hacer posible la comunicación entre el Servidor y las aplicaciones por un lado, y entre el Servidor y los aspectos por otro, se ha definido una interfaz que debe ser implementada por el Servidor de tal manera que se haga posible la comunicación.

Esta interfaz, llamada `IServer`, define las operaciones que tiene que ofrecer el Servidor del sistema, de tal manera que el código que se ha inyectado en la aplicación con el fin de que se registre en el Servidor al comenzar la aplicación y se desregistre al finalizarla (este código se inyectó por parte del JPI de forma simultánea a la inyección del MOP, ver 12.6) haga uso de las operaciones aquí definidas. De igual forma, es responsabilidad del desarrollador de los aspectos el hacer uso de las operaciones definidas en el interfaz con el fin de adaptar una aplicación en el sistema.

A continuación vamos a ver las operaciones básicas que define la interfaz, comentándolas brevemente:

- `recordApplication`. Este método sirve para que una aplicación se pueda registrar en el Servidor, de forma que, en adelante, pueda ser adaptada por los posibles aspectos que lo soliciten. El Servidor añade la aplicación a la lista de aplicaciones que están disponibles para ser adaptadas. El método recibe dos parámetros, uno que es un identificador global único de la aplicación (GUID *Globally Unique Identifier*) y otro que es una referencia a la propia aplicación. A través de esta referencia el Servidor podrá invocar posteriormente a los métodos de la aplicación contenidos en la implementación de la interfaz `IjoinPoint` (ver el punto 14.2), con el fin de añadir o eliminar aspectos de los puntos de enlace de la aplicación.
- `deleteApplication`. Este método sirve para que una aplicación se dé de baja del Servidor. El Servidor elimina la aplicación de la lista de aplicaciones que están disponibles para ser adaptadas. El método recibe un parámetro con el identificador global único de la aplicación.
- `recordAspect`. Mediante este método un aspecto solicita adaptar a una aplicación. Recibe cuatro parámetros, el primero es el identificador de la aplicación a la que desea adaptar, el segundo es el fichero XML con los puntos de corte que solicita, el tercero es una referencia al propio aspecto, lo que permitirá que el Servidor invoque al aspecto, y que además, le transmita esta referencia a la aplicación, para que ésta pueda invocar a los métodos del aspecto cuando se alcancen los puntos de enlace solicitados. Por último, recibe un cuarto parámetro que indica la prioridad del aspecto respecto a otros posibles aspectos que soliciten adaptar a la aplicación en los mismos puntos de enlace. Mediante esta prioridad se puede coordinar la ejecución de múltiples aspectos sobre el mismo punto de enlace (requisito 2.4.1.1).
- `deleteAspect`. Mediante este método un aspecto solicita dejar de adaptar a una aplicación de forma total o parcial. El método recibe tres parámetros, el primero es el identificador de la aplicación sobre la que se desea actuar (en este caso dejar de adaptar), el segundo parámetro es un fichero XML con los puntos de corte que seleccionan los puntos de enlace de los que el aspecto quiere dejar de ser informado de su ejecución. El tercer parámetro es una referencia a sí mismo. Esta referencia se necesita para que la aplicación pueda localizar y eliminar el registro del aspecto en los distintos puntos de enlace. Los puntos de corte pueden seleccionar todos los puntos de enlace donde se hubiese registrado el aspecto, con lo que éste dejaría de adaptar por completo a la aplicación, o por el contrario pueden seleccionar un subconjunto de los puntos de enlace donde se hubiese registrado el aspecto, en cuyo caso el aspecto seguiría adaptando a la aplicación en los puntos de enlace no seleccionados.

- `getApplications`. Este método devuelve una lista de las aplicaciones que están disponibles en el Servidor para ser adaptadas en ese momento.

13.3 Seguridad

La seguridad es una cuestión muy importante en el Servidor, a través de él se obtiene acceso a las aplicaciones que se están ejecutando en el sistema, y se puede llegar a modificar su funcionamiento. Podría ocurrir que una aplicación fuese adaptada por un aspecto con código malicioso que podría provocar el malfuncionamiento de la aplicación o realizar el robo de información. Por todo ello es necesario implementar un mecanismo de seguridad en el sistema que valide qué aspecto puede modificar a qué aplicación.

Puesto que, como hemos comentado, el Servidor ocupa el lugar central del sistema y de él dependen los aspectos para poder modificar el comportamiento de las aplicaciones, es el lugar adecuado para implementar la seguridad del sistema.

Existen múltiples mecanismos de seguridad que se podrían implementar en el Servidor, entre ellos las listas de control de acceso, la gestión de permisos asociados a los propietarios de los programas, etc. La implementación de cualquiera de estos mecanismos no entra dentro de los objetivos de esta Tesis, por lo que no vamos a profundizar más en su estudio.

13.4 Intercomunicación entre Procesos

Como ya hemos comentado anteriormente, para el funcionamiento del sistema es vital que haya una intercomunicación entre procesos. Por un lado, las aplicaciones se comunican con el Servidor para darse de alta y de baja en el sistema. El Servidor, a su vez, se comunica con las aplicaciones para activar y desactivar puntos de enlace en los que está interesado un aspecto. Los aspectos se comunican con el Servidor para solicitar adaptaciones a las aplicaciones. Las aplicaciones se comunican con los aspectos para notificar la ejecución de puntos de enlace en los que están interesados, y los aspectos se comunican con las aplicaciones para hacer uso de la implementación de la interfaz `IReflection`, y poder así acceder a su estructura, invocar métodos, etc. y, en definitiva, modificar el funcionamiento de la aplicación.

A continuación vamos a ver los requisitos que debe cumplir el mecanismo de comunicación entre procesos que usará el sistema, elegiremos un mecanismo en base a los requisitos impuestos y comentaremos sus características brevemente.

13.4.1 Requisitos Impuestos al Mecanismo de Comunicación

Para cumplir con los requisitos generales del sistema es necesario que el mecanismo de comunicación entre procesos cumpla con una serie de requisitos, que especificamos a continuación:

- Independiente del lenguaje. Puesto que un requisito del sistema es la independencia del lenguaje (requisito 2.1.2), los diferentes componentes del mismo pueden estar implementados en cualquier lenguaje, y aún así deben

poder establecer comunicación directa entre ellos. Por todo esto el mecanismo a utilizar debe ser independiente del lenguaje.

- Independiente de la plataforma. Otro requisito general del sistema es la independencia de la plataforma (requisito 2.2.1), entendiendo como plataforma el conjunto de *hardware* y sistema operativo. Este mismo requisito debe ser verificado por el mecanismo de comunicación de procesos para no introducir ninguna restricción a este nivel en el sistema, lo que invalidaría el requisito general.
- Distribuido. El sistema debe permitir la comunicación entre procesos que se encuentren distribuidos en distintas máquinas, de tal forma que la aplicación a adaptar, el Servidor y los aspectos puedan estar cada uno en una máquina distinta. De igual modo, debe permitir la comunicación entre procesos que se encuentren en la misma máquina, pero en distintos dominios de aplicación.
- Independiente del protocolo. Con el objetivo de no imponer ninguna restricción al sistema es necesario que el mecanismo de comunicación no imponga el uso de un protocolo específico, lo que podría limitar la utilidad del sistema a aquellos entornos donde existiese el protocolo especificado. Además, es frecuente que, por cuestiones de seguridad, ciertos protocolos no estén permitidos en las redes de las empresas, con lo que si nuestro sistema no pudiese hacer uso de alguno de los protocolos permitidos no tendría posibilidad de ser usado en esas empresas.
- Seguro. El mecanismo de comunicación debe ofrecer características de seguridad como la transferencia cifrada de las comunicaciones o la autenticación de los procesos que se comuniquen.
- Invocación remota de objetos. Para poder implementar nuestro sistema es necesario que los distintos actores del mismo no sólo se puedan enviar información entre sí, sino que puedan invocar a objetos remotos de una forma directa.

El cumplimiento de estos requisitos garantiza la posibilidad de conseguir los requisitos generales del sistema, establecidos en el capítulo 2. A continuación vamos a proceder a seleccionar el mecanismo de comunicación que va a utilizar nuestro sistema.

13.4.2 Elección del Mecanismo de Comunicación

Como hemos visto en el punto anterior el mecanismo de comunicación entre procesos que se emplee en nuestro sistema debe cumplir una serie de requisitos, y en base a ellos procederemos a su elección.

Existen dos posibilidades básicas sobre el mecanismo de comunicación de procesos. Una posibilidad es la de implementar un mecanismo *ad hoc* que cumpla con los requisitos impuestos, y la otra posibilidad es utilizar un mecanismo ya implementado que los satisfaga. La opción de desarrollar un mecanismo nuevo es muy costosa y además iría contra los requerimientos generales del sistema de la utilización de una plataforma estándar (requisito 2.2.4) y que sea un sistema ampliamente difundido (requisito 2.2.2), por lo que sólo debería contemplarse esta posibilidad en el caso de que no existiese un mecanismo ya definido que satisfaga los requisitos exigidos.

Puesto que nuestro sistema se va a ejecutar sobre la plataforma .NET el primer candidato que debemos tener en cuenta es .NET Remoting [MicrosoftRemoting], que es

el mecanismo de comunicación entre procesos nativo de la plataforma .NET. Debemos comprobar si cumple con todos los requerimientos impuestos y caso de ser así lo adoptaremos como el mecanismo del sistema.

.NET Remoting es nativo de la plataforma .NET y forma parte del estándar [ECMA335], y esto significa que cualquier lenguaje que se pueda ejecutar en la plataforma puede hacer uso de él, por lo que el requisito de la independencia del lenguaje es satisfecho. De igual modo, al ejecutarse dentro del .NET *Framework*, el sistema está disponible en cualquier plataforma donde esté disponible la máquina virtual, cumpliendo así el requisito de la independencia de la plataforma.

El sistema permite la comunicación entre objetos pertenecientes a dominios de aplicación o a procesos distintos, se encuentren en la misma máquina o en distintas máquinas distribuidas. Por lo tanto, el requisito impuesto de que fuese un sistema distribuido es cumplido completamente.

Otra particularidad de .NET Remoting es su flexibilidad en cuanto a la elección del protocolo de transporte, al formato de serialización, etc. Permite la utilización de cualquier protocolo. El sistema trae implementados por defecto una serie de ellos (como pueden ser TCP, HTTP o Named_Pipes), pero permite la implementación de cualquier otro protocolo y su utilización por parte de .NET Remoting. Es decir, el sistema es independiente del protocolo, que era otro de los requisitos impuestos.

Al poder seleccionar cualquier protocolo para el transporte, la comunicación puede realizarse mediante un protocolo seguro y que permita la autenticación de los procesos que intervienen en la comunicación, por lo que las características de seguridad impuestas pueden ser satisfechas con la elección del protocolo adecuado.

Como ya hemos comentado anteriormente, la invocación a objetos remotos es una de las características de .NET Remoting, por lo que el último requisito impuesto también lo cumple.

Puesto que .NET Remoting cumple con todos los requerimientos impuestos, es el mecanismo de comunicación entre procesos recomendado en la plataforma .NET y ofrece un buen rendimiento en ejecución (varía en función del protocolo de transporte que se utilice), es el mecanismo de comunicación entre procesos que utilizaremos en nuestro sistema. En el siguiente punto daremos una visión muy general de la arquitectura y características de .NET Remoting que pueden ser útiles para nuestro sistema.

13.4.3 .NET Remoting

.NET Remoting es el mecanismo de comunicación entre procesos especificado en el estándar de .NET. Está pensado para permitir la comunicación entre procesos en el entorno de la máquina virtual de .NET, pero también permite la comunicación con procesos externos que no se encuentren en una máquina virtual de .NET.

A continuación vamos a ir comentando brevemente los aspectos más relevantes para nuestro sistema de la tecnología .NET Remoting.

Arquitectura

.NET Remoting se puede utilizar para permitir que diferentes aplicaciones se comuniquen entre sí, sin importar que estas aplicaciones residan en la misma máquina, o en diferentes máquinas en la misma red local o incluso distribuidas en distintas redes a través de Internet. La infraestructura de Remoting abstrae la complejidad de la comunicación entre procesos.

En cierto sentido Remoting es similar a los Servicios Web [W3CWS], pero es más extensible. Mientras los Servicios Web permiten un acceso de alto nivel a los objetos vía SOAP [W3CSOAP], Remoting permite un control de bajo nivel sobre la comunicación. Remoting permite a los desarrolladores la elección del protocolo y el formato para el transporte de mensajes e incluso construir sus propios protocolos o formatos.

Si ambas partes de la relación están configurados correctamente, un cliente se limita a crear una nueva instancia de la clase del servidor. El sistema de interacción remota crea un objeto proxy local que representa a la clase (tiene su misma estructura, con sus métodos, propiedades, etc.) y devuelve al objeto del cliente una referencia al objeto Proxy creado. Otra posibilidad es crear una copia local completa del objeto, en vez del proxy, en cuyo caso las invocaciones a métodos del objeto tienen lugar en esa copia local, ejecutándose en el propio ámbito del cliente.

Cuando un cliente llama a un método lo hace a través del proxy (Figura 53), la infraestructura de interacción remota controla la llamada, comprueba el tipo de información y dirige la llamada por el canal hacia el proceso del servidor. Un canal a la escucha detecta la solicitud y la reenvía al sistema de interacción remota del servidor, que a su vez busca (o crea, si es necesario) y llama al objeto solicitado. A continuación, el proceso se invierte: el sistema de interacción remota del servidor incluye la respuesta en un mensaje que el canal del servidor envía al canal del cliente. Por último, el sistema de interacción remota del cliente devuelve el resultado de la llamada al objeto del cliente a través del objeto proxy.

En la Figura 53 podemos ver el funcionamiento general de una invocación a un método de un objeto remoto. El cliente se comunica con un objeto proxy local para solicitar la operación sobre el objeto remoto (para el cliente es transparente el hecho de que se trate de un proxy y no del objeto remoto realmente). Las credenciales de autenticación (por ejemplo, nombres de usuario, contraseñas, certificados, etc.) pueden establecerse mediante el proxy del objeto remoto. La llamada a un método pasa por una cadena de receptores (el desarrollador puede implementar sus propios receptores personalizados, como por ejemplo para realizar el cifrado de los datos) hasta un receptor de transporte que se encarga de enviar los datos por la red. En el servidor, la llamada pasa por la misma canalización (en el sentido inverso), hasta llegar al objeto que debe atender la petición. La respuesta del método invocado sigue el mismo camino en el sentido inverso (del servidor al cliente) para llegar al cliente que invocó al método.

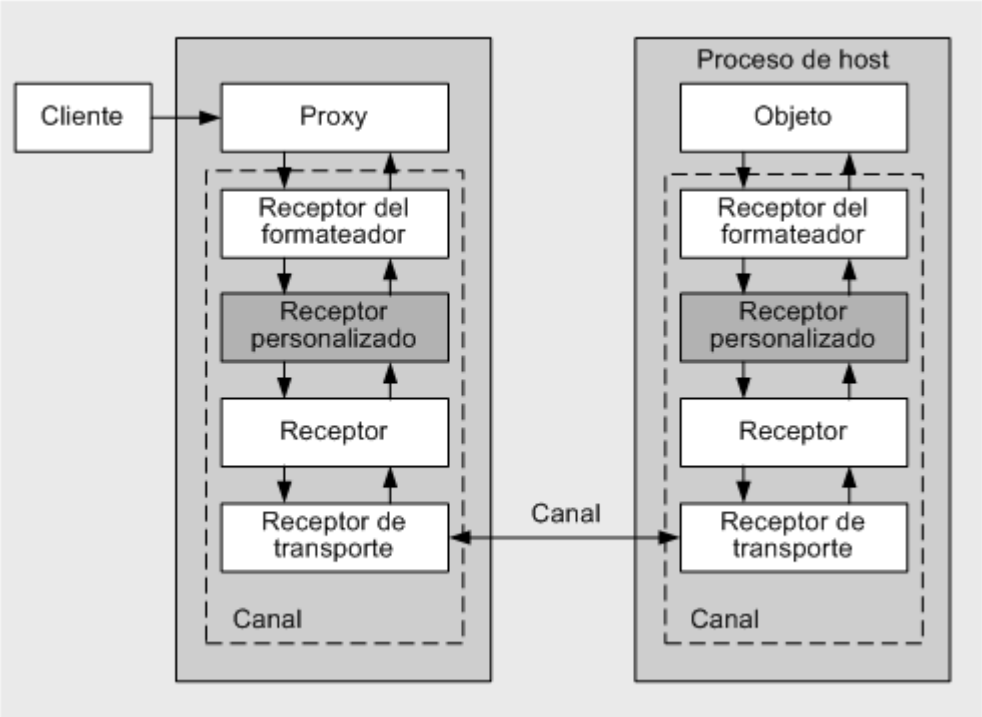

Figura 53: Arquitectura Remoting

.NET Remoting utiliza receptores de canales de transporte, receptores de canales personalizados y receptores de canales de formateador cuando un cliente invoca una llamada a un método en un objeto remoto.

Los receptores de canales de transporte transmiten llamadas a métodos por la red entre el cliente y el servidor. La implementación de .NET de Microsoft proporciona los canales TCP y HTTP ya implementados (otras implementaciones, como Mono, tienen otros canales implementados) aunque la arquitectura es totalmente extensible y permite al desarrollador utilizar sus propias implementaciones personalizadas.

Los receptores de canales personalizados pueden utilizarse en distintas ubicaciones de la canalización de receptores de canales para modificar los mensajes enviados entre el cliente y el servidor. Un receptor de canales que proporciona cifrado y descifrado constituye un ejemplo típico de receptor de canales personalizado.

Los receptores del formateador reciben llamadas de método y las serializan en un flujo que se puede enviar por la red. La implementación de .NET de Microsoft proporciona dos receptores de formateador: a) el Formateador SOAP que empaqueta las llamadas de método en forma de mensaje SOAP, y b) el formateador binario que empaqueta las llamadas de método en forma de un flujo binario serializado y ofrece un rendimiento muy alto en comparación con el formateador SOAP. Se pueden implementar otros tipos de formateadotes personalizados.

Canales

Los canales son objetos que transportan mensajes de una aplicación a otra a través de los límites de interacción remota, tanto de un dominio de aplicación a otro, como de un proceso a otro o de un equipo a otro. Un canal puede escuchar los mensajes entrantes en un extremo, enviar los mensajes salientes a otro extremo o ambas acciones (es

decir, puede ser de entrada, de salida o de entrada y salida). Se pueden construir canales personalizados para que soporten cualquier protocolo, de hecho, no es necesario que en el otro extremo del canal esté una máquina .NET, puede estar cualquier sistema que entienda el protocolo usado.

Como hemos visto, en el cliente los mensajes se pasan a la cadena de receptores de canal del cliente, el primer receptor de canal suele ser un receptor de formateador que serializa el mensaje en una secuencia que, a continuación, pasa por la cadena de receptores de canal hasta el receptor de transporte del cliente. El receptor de transporte del cliente escribe entonces la secuencia en la conexión.

En el servidor, el receptor de transporte del servidor lee las solicitudes de la conexión y pasa la secuencia de solicitud a la cadena de receptores de canal del servidor. El receptor de formateador del servidor situado al final de esta cadena deserializa la solicitud en un mensaje. A continuación, lo pasa a la infraestructura de interacción remota que lo transmite al objeto del servidor al que iba dirigido.

La cuestión de seleccionar el canal sobre el que desarrollar una aplicación no es trivial, ya que pueden afectar a cuestiones tales como la eficiencia, la seguridad (permiten o no encriptación de la información, permiten la autenticación, etc.), o la posibilidad de uso en redes que puedan tener ciertos protocolos restringidos.

Como hemos visto, se pueden implementar canales que soporten cualquier tipo de protocolo. Los más comunes son los canales que soportan los protocolos TCP y HTTP, pero existen otros muchos. En la actualidad hay muchos proyectos, generalmente de código abierto, con lo que están disponibles para ser usados libremente, que se dedican a implementar canales de acuerdo a otros protocolos. Algunos ejemplos pueden ser:

- Named_pipes: utiliza *named pipes* para establecer las comunicaciones. Su principal ventaja respecto a TCP y sobre todo HTTP es el rendimiento pero, como contrapartida, sólo puede utilizarse dentro de la misma máquina (no permite la comunicación entre procesos ubicados en máquinas distintas).
- Remoting.Corba [RemotingCORBA], es un proyecto de código abierto que pretende integrar la comunicación con sistemas CORBA/IIOP en Remoting. La principal ventaja es la posibilidad de interactuar entre objetos .NET, CORBA y J2EE.
- IIOP.NET [IIOP.NET], es un proyecto similar al anterior que persigue los mismos objetivos.
- IPC: soporte para el protocolo IPC (*interproccess comunication)* propio del sistema operativo Windows (por lo que sólo está disponible en las implementaciones de la máquina virtual que trabajen sobre Windows). Está disponible en la versión 2.0 del Framework de Microsoft.

Configuración

Como hemos comentado anteriormente, .NET Remoting es un sistema flexible que permite al usuario la configuración de los diversos elementos que va a utilizar. Para configurarlo se puede optar por realizarlo de forma programática o por medio de ficheros de configuración.

En ambos casos es necesario indicar la información necesaria para poder establecer la comunicación. Por ejemplo, es necesario establecer los canales de comunicación, esto incluye indicar el protocolo y el puerto, el tipo de formateador y, en el caso de que estemos configurando un servidor, indicar los objetos que desea publicar y con qué nombre (que lo identifique de forma única en la red, es decir un URI), en el caso de los clientes debe indicar a qué URI desea conectarse para obtener los objetos remotos, etc.

La ventaja principal del uso de archivos de configuración, con respecto al uso de la configuración programática, es que se separa la información de configuración del código del programa, para así poder realizar cambios modificando simplemente el archivo de configuración, sin necesidad de editar y volver a compilar el código fuente de programa. Tanto los objetos de servidor como los de cliente de servicios remotos .NET pueden utilizar estos archivos de configuración.

Metadatos

Tanto en el paso por valor como en el paso por referencia, cuando se obtiene un objeto remoto el proceso que se sigue implica una serialización en el proceso remoto (servidor), la transmisión a través del canal de esta información serializada, y la posterior reconstrucción (deserialización) en el proceso local (cliente), ya sea para crear una copia del objeto remoto (paso por valor) o para crear un proxy que lo represente (paso por referencia).

En cualquier caso, para que el proceso local pueda reconstruir el objeto a partir de la información serializada que ha recibido, es necesario que disponga de información suficiente para hacerlo. Parte de esta información son los metadatos del objeto presentes en los ensamblados de .NET. El cliente necesita tener acceso a estos metadatos para poder acceder al objeto, esto puede hacerlo de tres formas distintas:

- A través del ensamblado del objeto del servidor: el objeto de servidor puede crear un ensamblado con los metadatos necesarios y distribuirlo entre los clientes. El objeto de cliente debe hacer referencia al ensamblado mientras se compila. Este método resulta muy útil en escenarios cerrados en los que tanto el servidor como el cliente son componentes administrados.
- Los objetos de servicios remotos pueden proporcionar un archivo WSDL [W3CWSDL] que describe el objeto y sus métodos. Los clientes capaces de leer y generar solicitudes de SOAP correspondientes al archivo WSDL pueden llamar a este objeto y establecer comunicaciones con él utilizando SOAP. Los objetos de servidor de los servicios remotos .NET pueden emplear la herramienta soapsuds.exe que se distribuye con el SDK de .NET para generar archivos WSDL en forma de metadatos. Este método resulta útil en aquellos casos en los que una organización desea ofrecer un servicio público al que tenga acceso y que pueda utilizar cualquier cliente.
- Los clientes de .NET pueden crear archivos fuente o un ensamblado que contenga sólo metadatos, no código, descargando del servidor el esquema XML (generado en el mismo) con la utilidad soapsuds. Los archivos fuente se pueden compilar en la aplicación del cliente. Este método resulta útil en las aplicaciones de varios niveles, en las que los objetos de un nivel tratan de obtener acceso a los objetos remotos de otro.

Paso por valor y paso por referencia

Existen una serie de objetos que no son utilizables de forma remota, en concreto la mayor parte de las clases de la Biblioteca de Clases Base (BCL) de .NET no son utilizables de forma remota.

A los objetos que son utilizables de forma remota se puede acceder de dos formas distintas, mediante un proxy local que facilita el acceso al objeto remoto, o mediante una copia local del objeto remoto. Es decir, existen objetos de cálculo por referencia y objetos de cálculo por valor.

Los objetos de cálculo por valor (*Marshal–By–Value* MBV) se transmiten de la siguiente manera: el sistema de interacción remota realiza una copia completa de estos objetos y se la pasa al dominio de aplicación que hace la llamada. Una vez que la copia está en el dominio de aplicación del llamador, las llamadas al objeto se dirigen directamente a esa copia. Un objeto es de tipo MBV cuando es marcado con el atributo `[Serializable]` o implementa la interfaz `ISerializable`.

Los objetos de cálculo por referencia (*Marshal–By–Reference* MBR) funcionan de la siguiente forma: cuando un cliente crea una instancia de un objeto MBR en su propio dominio de aplicación, la infraestructura de .NET Remoting crea un objeto proxy en el dominio de aplicación del llamador que represente al objeto MBR y devuelve al llamador una referencia a dicho objeto proxy. A partir de ese momento, el cliente realiza las llamadas en el proxy. La interacción remota calcula las referencias de esas llamadas, las devuelve al dominio de aplicación de origen e invoca la llamada en el objeto propiamente dicho. Un objeto es de tipo MBR cuando extiende la clase `MarshalByRefObject`.

En nuestro sistema vamos a necesitar principalmente utilizar el paso de objetos por referencia debido a que si necesitamos modificar algún objeto recibido es necesario que la modificación tenga lugar en el objeto remoto, ya que una modificación en la copia local no adaptaría el comportamiento de la aplicación; de igual modo si necesitásemos invocar a algún método es necesario que la ejecución del método tenga lugar en el origen y no en una copia (por ejemplo, si tuviésemos que mostrar un mensaje por consola es necesario que se ejecute en la aplicación remota – que se está adaptando–, y no en la aplicación local –el aspecto–).

CAPÍTULO 14
FRAMEWORK DE ASPECTOS

En este capítulo definimos el Framework de Aspectos en el que se basa el sistema. Presentamos su estructura general, así como las diferentes interfaces que define, explicando detalladamente su funcionalidad.

14.1 Estructura del Framework

El Framework aquí presentado se basa en la definición de una serie de interfaces, que tendrán que ser implementadas por los distintos actores del sistema (aplicaciones y aspectos, el Servidor hace uso de las implementaciones que hagan ellos).

La razón de necesitar estas interfaces es la de establecer una serie de contratos que deben cumplir las aplicaciones y los aspectos para poder formar parte del sistema y garantizar la posibilidad de comunicación entre ellos.

Una aplicación debe permitir que el Servidor active o desactive sus puntos de enlace en base a las peticiones que recibe por parte de un aspecto. La forma de realizar esto es mediante una serie de métodos que utilizará el Servidor. Para garantizar la existencia de esos métodos y su correcto funcionamiento (al menos en lo que respecta a su tipo) se obliga a la aplicación a implementar una interfaz creada al efecto.

De igual modo, cuando una aplicación alcanza la ejecución de un punto de enlace ésta debe notificar el evento a aquellos aspectos que hayan seleccionado dicho punto de enlace. Para ello, mediante el código inyectado para implementar el MOP reflectivo, se realiza una llamada a un método de cada aspecto. La manera de poder hacer esto, garantizando que dicho método existe en el aspecto, es obligar a que el aspecto implemente una interfaz, que también conocerá la aplicación mediante el código añadido.

Existe además otra razón por la que es necesario establecer las interfaces. Nuestro sistema se basa en las relaciones entre procesos: la aplicación se registra en el servidor, éste activa los puntos de enlace en la aplicación, los aspectos solicitan al servidor adaptar una aplicación, las aplicaciones avisan a los aspectos cuando su ejecución alcanza un punto de enlace seleccionado, etc. Para poder llevar a cabo estas tareas es necesario que exista un mecanismo de comunicación entre procesos que permita esto.

El sistema, tal y como hemos visto en 13.4, utiliza Remoting como mecanismo para establecer las comunicaciones entre procesos. En Remoting un servidor publica u oferta un servicio al exterior (al resto de aplicaciones), el servicio realmente es un objeto. El cliente obtiene un Proxy local del objeto y puede interactuar con el objeto original a través del Proxy local. Para obtener el objeto (o el Proxy que lo representa) el proceso que se sigue es el siguiente: 1) el objeto es serializado en el servidor, 2) viaja de forma serializada hasta el cliente y 3) se crea un Proxy local en el cliente para interactuar con el objeto del servidor, para lo cual es necesario poder deserializar lo recibido. Las peticiones que haga el cliente al Proxy creado (invocaciones a métodos, etc.) serán transferidas, de forma transparente para el desarrollador, al objeto del servidor, que es quien las ejecuta, devolviendo el resultado, si es que hay uno, al cliente a través del Proxy.

Para poder deserializar un objeto es necesario conocer su estructura, es decir, su tipo y, por lo tanto, es necesario que el tipo sea conocido por la aplicación cliente. Para lograr esto existen varias posibilidades y una de ellas es el uso de interfaces. Se define una interfaz al que se tiene que ajustar el objeto publicado por el servidor, dicho interfaz debe ser conocido también por el cliente, y de esta forma al recibir el objeto serializado puede proceder a su deserialización.

En concreto el Framework consta de los siguientes componentes:

- `IJoinPoint`. Interfaz que debe implementar la aplicación que puede ser adaptada. Su funcionalidad principal es la de permitir la activación y desactivación de los puntos de enlace de la aplicación.
- `IReflection`. Interfaz que debe implementar la aplicación que puede ser adaptada. Su funcionalidad consiste en ofrecer reflexión intraaplicaciones, es decir, el acceso reflectivo desde una aplicación a otra.
- `IMethodCall`. Interfaz que deben implementar los aspectos que van a adaptar a una aplicación en los puntos de enlace de tipo invocación a método o constructor o ejecución de método o constructor.
- `IPropertyFieldAccess.` Interfaz que deben implementar los aspectos que van a adaptar a una aplicación en los puntos de enlace de tipo acceso de lectura o escritura a un campo o una propiedad.
- `ICatchException`. Interfaz que deben implementar los aspectos que van a adaptar a una aplicación en los puntos de enlace de tipo tratamiento de excepción.
- `IServer`. Interfaz que debe implementar el Servidor y que permite que los aspectos y las aplicaciones entren en el sistema.

En la Figura 54 se pueden observar los distintos componentes existentes en el sistema con las interfaces que implementan cada uno de ellos.

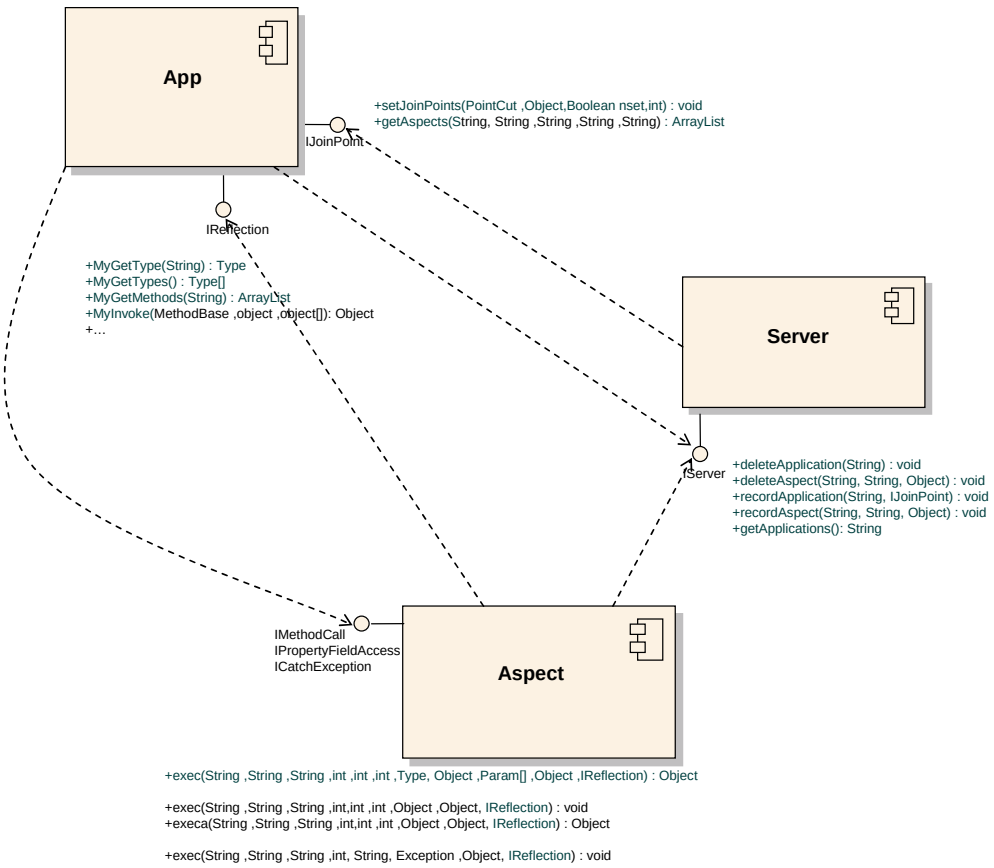

Figura 54: Diagrama de componentes e interfaces

A continuación vamos a ver detalladamente cada uno de estos componentes (a excepción del `IServer`, el cual se explicó detalladamente en el punto 13.2 por lo que no volveremos a explicarlo aquí.).

14.2 IJoinPoint

Esta interfaz contiene las definiciones para los métodos relacionados con los puntos de enlace que debe implementar una aplicación para que pueda ser adaptada. En concreto son dos los métodos que define:

- `SetJoinPoints`. Este método será utilizado por parte del Servidor para activar o desactivar puntos de enlace. El método recibe cuatro parámetros: los puntos de corte que solicitó el aspecto, una referencia al aspecto, un valor booleano que indica si se solicita activar o desactivar y la prioridad del aspecto en el caso de coincidencia de varios aspectos en el mismo punto de enlace. Los puntos de corte se reciben como una representación mediante un modelo de objetos del fichero que el aspecto envió al Servidor (ver 12.8). El funcionamiento consiste en que para cada punto de enlace existente en la aplicación se comprueba si alguno de los puntos de corte suministrados le afecta y, caso de ser así y, si está activando los puntos de enlace (tercer parámetro), se almacena la referencia al aspecto y la prioridad en una tabla, si se están desactivando puntos de enlace se elimina la referencia del aspecto de la tabla.

- `GetAspects`. Este método se utiliza por parte del código inyectado que implementa el MOP reflectivo para obtener los aspectos que han solicitado ser notificados de la ejecución de un punto de enlace. El método recibe una serie de parámetros que identifican un punto de enlace (su tipo, *namespace*, clase, nombre, tiempo, etc.) y devuelve una lista con las referencias de todos los aspectos que hayan activado ese punto de enlace. La lista devuelta estará ordenada en base a la prioridad de ejecución de cada aspecto (recibida en el momento del registro).

Esta interfaz debe ser implementada por la aplicación que vaya a ser adaptada, pero como deben cumplirse los requisitos de no imposición de condiciones a la aplicación (requisito 2.3.2) y la no necesidad de disponer de su código fuente (requisito 2.3.1) es el sistema quien debe aportar esta implementación.

Mediante el JPI, durante el proceso de inyección del MOP reflectivo, se añade el código necesario a la aplicación para que contenga una implementación de esta interfaz y pueda ser utilizada desde el exterior (ver 12.6).

14.3 IReflection

Esta interfaz contiene las definiciones de los métodos que permitirán acceder reflectivamente a la aplicación que lo implemente desde otra aplicación externa.

.NET ofrece mecanismos de introspección, pero sólo están accesibles en el dominio de la propia aplicación, es decir, no se puede acceder desde una aplicación a la información de otra. Lo que va a realizar la implementación de esta interfaz es ofrecer una serie de métodos que obtendrán la información de la aplicación en la que se ejecutan y la devolverán a la aplicación que los invocó. Se ha seleccionado un subconjunto de las características introspectivas de .NET para ser ofrecidas a través de la interfaz.

Realmente lo que hace esta interfaz (su implementación) es actuar de intermediaria entre una aplicación remota y el *namespace* `System.Reflection`. A la funcionalidad de este *namespace* se puede acceder únicamente desde la propia aplicación. Es decir, ofrece información de la propia aplicación que lo invoca, por lo que si necesitamos obtener información de otra aplicación distinta tendrá que ser esa aplicación la que realice efectivamente las invocaciones para obtenerla.

En concreto hay dos funcionalidades básicas que debe contemplar la interfaz. Por un lado es necesario que ofrezca una serie de métodos que permitan conocer la estructura de la aplicación, es decir, los tipos existentes en la aplicación, las clases, los miembros (campos, métodos, propiedades etc.) de esas clases, etc. La otra funcionalidad básica que debe ofrecer es la posibilidad de invocar a miembros de la aplicación remota (los miembros invocables pueden ser métodos, campos y propiedades).

Dentro del *namespace* `System.Reflection` se han seleccionado los métodos que son necesarios para el sistema y se ha creado un método en la interfaz por cada uno de ellos. El método de nombrado que se ha seguido consiste en anteponer "`My`" al nombre original del método, de tal forma que sean fácilmente reconocibles por parte del desarrollador familiarizado con el *namespace* original. Los métodos que define esta interfaz son los siguientes:

- `MyGetAssembly`. Devuelve el ensamblado desde el que se inició la aplicación.
- `MyGetNamespaces`. Devuelve una lista con todos los *namespaces* de los que está compuesta la aplicación.
- `MyGetTypes`. Devuelve una lista con todos los *types* que contiene la aplicación. A partir de esos tipos se podrá ir descendiendo hasta acceder a las clases, sus métodos, parámetros de estos, etc.
- `MyGetClasses`. Devuelve una lista con todos los nombres de todas las clases contenidas en la aplicación.
- `MyGetFields`. Devuelve una lista con todos los campos de la clase que recibe como parámetro.
- `MyGetMethods`. Devuelve una lista con todos métodos de la clase que recibe como parámetro.
- `MyGetProperties`. Devuelve una lista con todas las propiedades de la clase que recibe como parámetro.
- `MyGetType`. Este método recibe una cadena de caracteres con el nombre del tipo solicitado y lo devuelve, caso de encontrarlo.
- `MyInvoke` (para métodos). Este método recibe tres parámetros: el primero es información del método al que se va a invocar, el segundo una referencia a la instancia en la que se va a ejecutar el método, y el tercero una lista con los parámetros que se quieren enviar al método que se invoca. Devuelve lo que devuelva la ejecución del método.
- `MyInvoke` (para constructores). Este método es similar al anterior, pero sólo recibe dos parámetros: la información referente al constructor, y la lista de parámetros que se le quieren pasar. Devuelve la instancia que haya creado la ejecución del constructor.
- `MyInvokeMemberField`. Este método permite invocar a un campo, ya sea para realizar un acceso de lectura o escritura (dependiendo de los parámetros recibidos). Si es de lectura devuelve el valor leído.
- `MyInvokeMemberProperty`. Este método permite invocar a una propiedad, ya sea para realizar un acceso de lectura o escritura (dependiendo de los parámetros recibidos). Si es de lectura devuelve el valor leído.

Al igual que en el caso anterior, esta interfaz debe ser implementada por la aplicación que vaya a ser adaptada, pero como deben cumplirse los requisitos de no imposición de condiciones a la aplicación (requisito 2.3.2) y la no necesidad de disponer de su código fuente (requisito 2.3.1) es el sistema quien debe aportar esta implementación.

Mediante el JPI, durante el proceso de inyección del MOP reflexivo, se añade el código necesario a la aplicación para que contenga una implementación del mismo y pueda ser utilizada desde el exterior (ver 12.6).

14.4 Interfaces de Aspectos

Cuando una aplicación alcanza durante su ejecución un punto de enlace para el cual se ha registrado un aspecto, debe informarle de esto. La manera de informar es mediante la invocación a un método del aspecto, pasándole como parámetros toda la información que pueda necesitar (esta información varía de un tipo de punto de enlace a

otro). Esta información incluye la identificación exacta del punto de enlace cuya ejecución produjo la llamada así como información reflectiva sobre el punto de enlace, que podrá ser utilizada por el aspecto para su funcionamiento.

La invocación al aspecto forma parte del código inyectado en la aplicación como parte del MOP y, para poder realizar dicha inyección, es necesario conocer el método al que se va a invocar, es decir, es necesario definir una interfaz que especifique los métodos existentes y su signatura. Puesto que cada tipo de punto de enlace se identifica de forma única y lleva asociado distinta información reflectiva, se ha definido un conjunto de interfaces, agrupando por puntos de enlace que comparten el mismo tipo de información.

Cada una de las interfaces define un método llamado `exec` que recibirá la llamada de la aplicación cuando se alcance el punto de enlace. De igual modo, definen otro método, denominado `ping`, cuya única misión es que se pueda comprobar desde el proceso llamante si el aspecto sigue activo. Si un aspecto desea adaptar el comportamiento de una aplicación, debe implementar las interfaces que se correspondan con los puntos de enlace en los que esté interesado.

Si un aspecto está interesado (se ha registrado) en varios puntos de enlace del mismo tipo, cuando en la aplicación se alcanza la ejecución de cualquiera de ellos se invoca al mismo método del aspecto (el `exec`), el cual, mediante la información que recibe como parámetros, puede saber desde qué punto de enlace se le ha invocado, y en función de esto realizar unas acciones u otras.

A continuación vamos a explicar brevemente las diferentes interfaces de aspectos que existen en el sistema.

IMethodCall

Esta interfaz se corresponde con los puntos de enlace invocación o ejecución a un método o constructor. Los parámetros que recibe el método `exec` son los siguientes:

- Namespace. Indica el *namespace* al cual pertenece el método o constructor.
- Class. Indica la clase a la que pertenece el método o constructor.
- Member. Nombre del método a ejecutar o invocar.
- Type. Indica el tipo de miembro al que se refiere, en este caso puede ser "method" para el caso de que sea un método, o "constructor" para el caso de que sea un constructor.
- Jp. Indica el tipo de punto de enlace al que se refiere, en este caso puede ser *Call* o *Execution*, dependiendo de si es una invocación o una ejecución respectivamente.
- Time. Indica el tiempo al cual se refiere el punto de enlace. Los posibles valores son *Before, After y Around*.
- ResultType. Tipo de retorno del método, en el caso de que devuelva algo (si no devuelve nada, el tipo sería `void`).
- ResultVal. Indica el valor que ha devuelto la ejecución o la invocación al método o constructor, en el caso de que devuelva algo. Esto sólo tiene sentido en el tiempo *after*.

- Params. Array que contiene un registro por cada parámetro que recibe el método o constructor, conteniendo el nombre del parámetro, su tipo y su valor. En caso de que no tenga parámetros el campo vendría vacío.
- Object_This. Referencia al objeto `this` desde donde se originó la llamada (desde el punto de enlace). Gracias a esta referencia y a la implementación de `IReflection` que hace la aplicación se puede adaptar ésta. En caso de que fuese desde un lugar estático este campo vendría vacío (`null`).
- IR. Referencia a la implementación de la interfaz `IReflection` por parte de la aplicación. Mediante esta referencia el aspecto podrá invocar a los métodos definidos en la interfaz, que están implementados en la aplicación, accediendo a la estructura de ésta o invocando a sus miembros (para invocar métodos, acceder a campos, etc.).

El método devuelve un `object` que, en el tiempo *around* para la invocación a métodos y constructores, contendrá lo que se quiere devolver en sustitución de lo que habría devuelto la ejecución normal de la invocación.

IPropertyFieldAccess

Esta interfaz se corresponde con los puntos de enlace acceso de lectura o escritura a un campo o a una propiedad. Los parámetros que recibe el método `exec` son los siguientes:

- Namespace. Indica el *namespace* al cual pertenece el campo o propiedad.
- Class. Indica la clase a la que pertenece el campo o propiedad.
- Member. Nombre del campo o propiedad a la que se accede.
- Type. Indica el tipo de miembro al que se refiere, en este caso puede ser "Property" para el caso de que sea una propiedad, o "Field" para el caso de que sea un campo.
- Jp. Indica el tipo de punto de enlace al que se refiere, en este caso puede ser *reference* o *set*, dependiendo de si es un acceso de lectura o escritura.
- Time. Indica el tiempo al cual se refiere el punto de enlace. Los posibles valores son *Before, After* (no puede ser *Around*, para este tiempo hay otro método específico).
- Val. Indica el valor de la propiedad o campo en ese momento. Si se trata de un acceso de lectura el valor se corresponde con el del campo o propiedad, pero si es un acceso de escritura se corresponde con el valor que se le va a asignar (o se le ha asignado, dependiendo del tiempo).
- Object_This. Referencia al objeto `this` desde donde se originó la llamada (desde el punto de enlace). En caso de que fuese desde un lugar estático este campo vendría vacío (`null`).
- IR. Referencia a la implementación de la interfaz `IReflection` por parte de la aplicación. Mediante esta referencia el aspecto podrá invocar a los métodos definidos en la interfaz, que están implementados en la aplicación, accediendo a la estructura de ésta o invocando a sus miembros.

El método no devuelve nada. En esta interfaz existe otro método llamado `execa`, que es específico para el tiempo *Around*. Tiene los mismos parámetros que el método `exec` pero con las diferencias de que el parámetro *time* tiene que valer obligato-

riamente *Around* (realmente este parámetro no haría falta incluirlo pero se ha hecho para mantener la misma signatura que el método `exec`), y que devuelve un `object`. Dicho objeto es el que se asignará al campo o propiedad en el caso de un acceso de escritura, o en el caso de acceso de lectura, se introducirá en la pila en sustitución del que se hubiese introducido si se hubiese realizado la ejecución normal del punto de enlace (sin aspecto).

ICatchException

Esta interfaz se corresponde con el punto de enlace tratamiento de una excepción en un bloque *catch*. Los parámetros que recibe el método `exec` son los siguientes:

- Namespace. Indica el *namespace* al cual pertenece el método donde se encuentra el código de tratamiento de la excepción.
- Class. Indica la clase a la cual pertenece el método donde se encuentra el código de tratamiento de la excepción.
- Member. Nombre del método donde se encuentra el código de tratamiento de la excepción.
- Time. Indica el tiempo al cual se refiere el punto de enlace. Los posibles valores son *Before, After o Around*.
- Exception_type. Tipo de la excepción que se captura para ser tratada en el bloque *catch*.
- Exception_val. La excepción que se va a tratar.
- Object_This. Referencia al objeto `this` desde donde se originó la llamada (desde el punto de enlace). En caso de que fuese desde un lugar estático este campo vendría vacío (`null`).
- IR. Referencia a la implementación de la interfaz `IReflection` por parte de la aplicación. Mediante esta referencia el aspecto podrá invocar a los métodos definidos en la interfaz, que están implementados en la aplicación, accediendo a la estructura de ésta o invocando a sus miembros.

CAPÍTULO 15
IMPLEMENTACIÓN DE UN PROTOTIPO

En este capítulo se mostrará el diseño de un prototipo del sistema presentado, realizado como prueba de viabilidad. Para su desarrollo hemos utilizado el lenguaje de programación C# [ECMA334].

La arquitectura del sistema es la mostrada en el capítulo 9, donde se explica detalladamente, identificando los distintos componentes del mismo y su funcionalidad.

15.1 Visión General

En el diagrama de la Figura 55 podemos ver los diferentes componentes que actúan en el sistema. Se puede observar que hay tres nodos. Uno donde se ejecuta la aplicación que puede ser adaptada, otro donde se ejecuta el Servidor del sistema, y un tercero donde se ejecuta el aspecto que adapta a la aplicación.

Se pueden ver una serie de componentes que se han implementado como parte del sistema:

- PointCut: intérprete del lenguaje de puntos de corte.
- XML2PointCut: componente que procesa un fichero XML donde vienen definidos los puntos de corte y crea una estructura PointCut para que sea interpretado.
- Interfaces: librería que contiene las definiciones de las interfaces del Framework del sistema, así como las implementaciones de las interfaces `IReflection` e `IJoinPoint`.
- JPI (*JoinPoint Injector*): aplicación que se encarga de instrumentar las aplicaciones en el momento de su entrada en el sistema para que puedan ser adaptadas.
- Server: Servidor de aplicaciones del sistema.

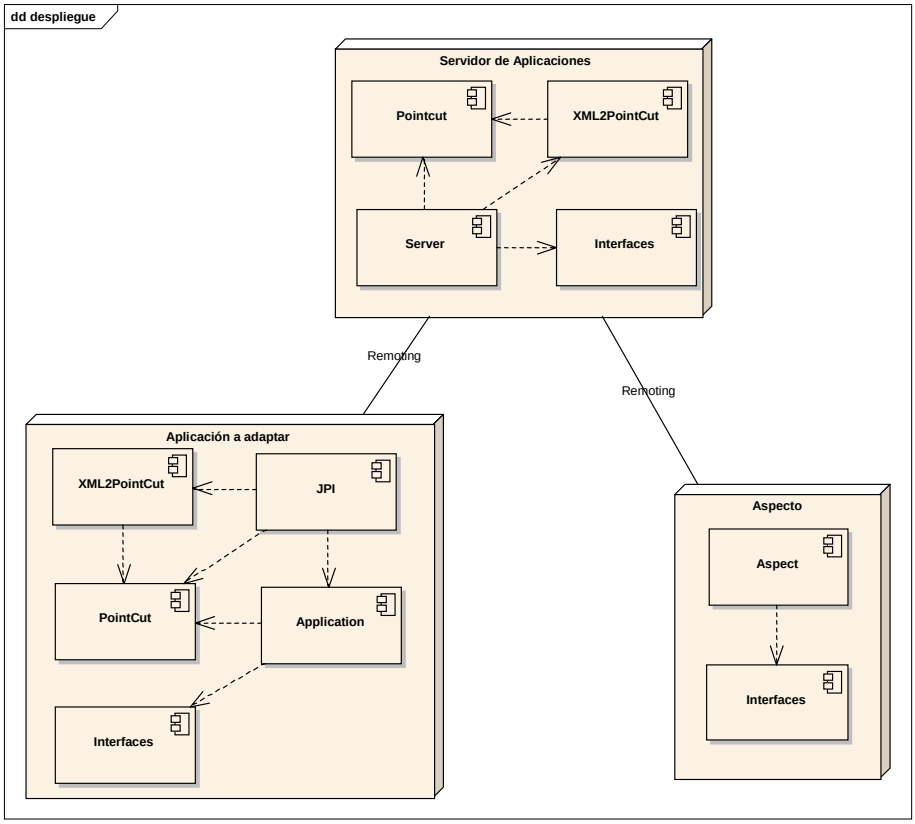

Figura 55: Diagrama de componentes del Sistema

A continuación vamos a describir someramente las decisiones de implementación tomadas para cada elemento del sistema.

15.2 PointCut

Como explicamos en el capítulo 11, los puntos de corte se definen en el sistema mediante un fichero en XML, que tiene que ser validado con el esquema XML definido. Para procesar los ficheros de este lenguaje hemos creado un intérprete que analizará el fichero y será capaz de realizar dos acciones. La primera es responder a la pregunta ¿es necesario inyectar código en un punto de enlace determinado?. La segunda es, habiendo recibido una tabla *Hash* con todos los puntos de enlace de una aplicación y una referencia a un aspecto, determinar cuáles de los puntos de enlace recibidos son seleccionados por los puntos de corte, y añadir el aspecto (registrarlo) en los puntos de enlace que hayan sido seleccionados.

Para realizar el intérprete hemos hecho uso principalmente del patrón *Interpreter* [Gamma94], e internamente de los patrones *Composite*, *Pattern* y *State* [Gamma94]. Mediante este diseño representamos los puntos de corte de un fichero determinado.

En la Figura 56 se puede ver la primera parte del diagrama de clases de este componente.

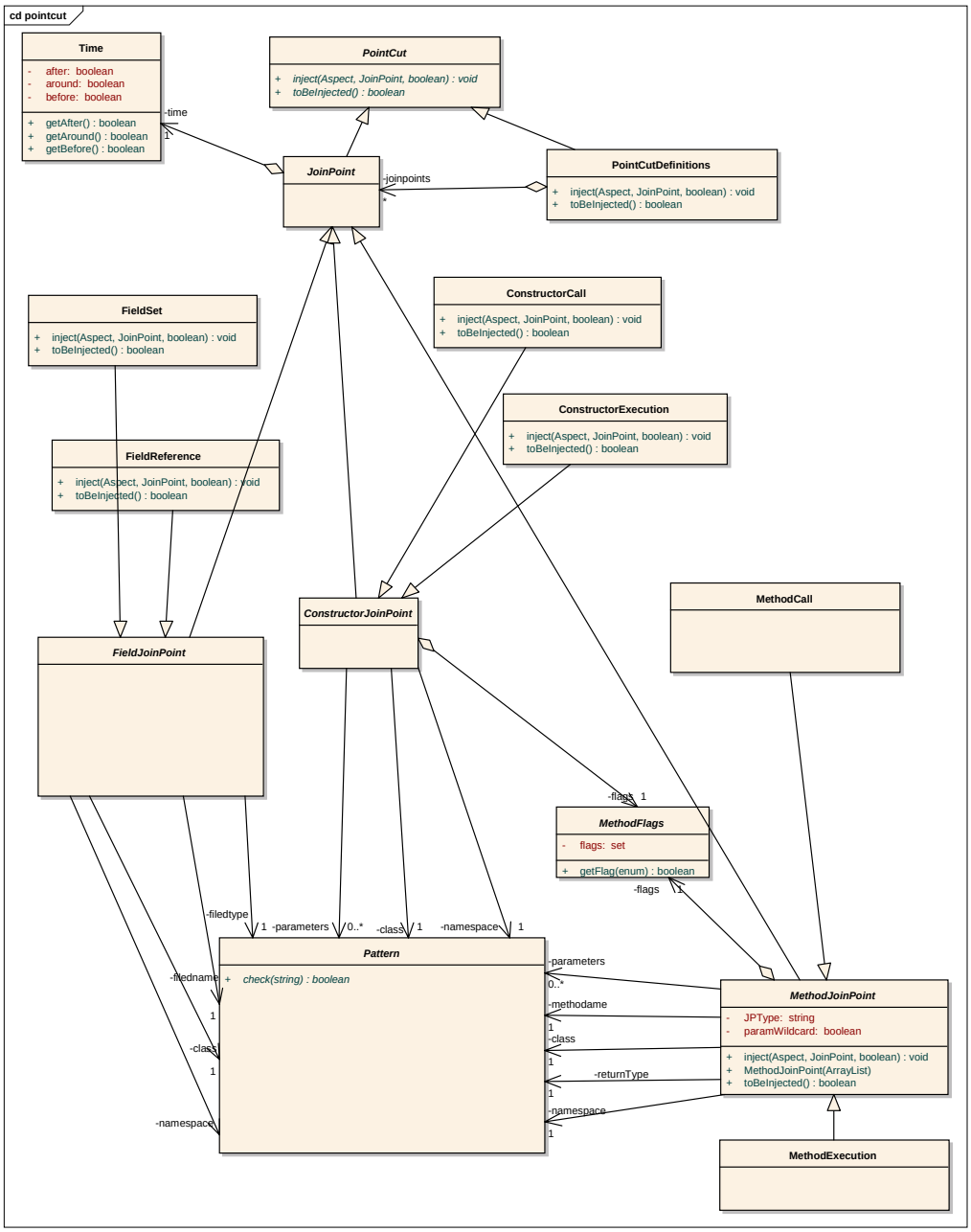

Figura 56: Diagrama de clases de PointCut, parte a

En la Figura 57 podemos ver el diagrama con la implementación de Pattern.

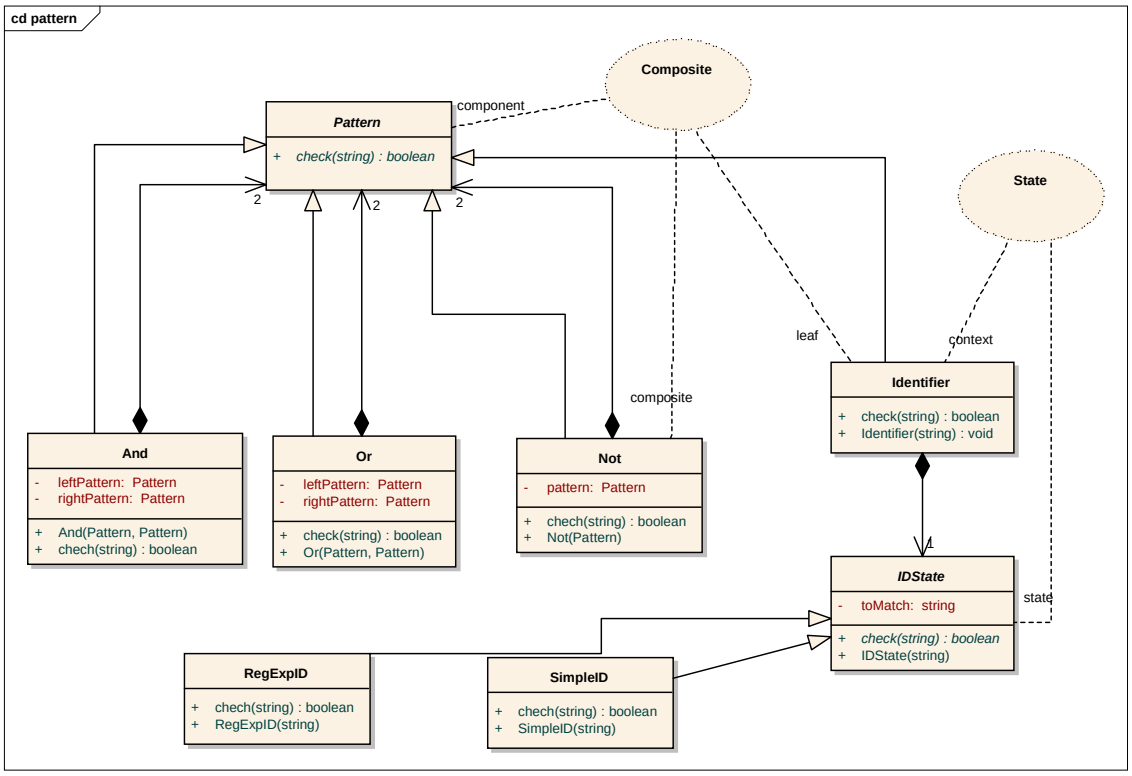

Figura 57: Diagrama de clases de PointCut, parte b

Este componente se presenta como una librería del sistema y es utilizada tanto por el JPI como por el Servidor.

15.3 XML2PointCut

Como hemos comentado, los puntos de corte vienen expresados mediante un fichero XML que debe ser válido respecto al esquema definido en el sistema. Hemos creado un modelo que permite representar los puntos de corte e interpretarlos. En este componente lo que hacemos es leer el fichero XML, validarlo y, en base a lo que contenga, crear el modelo de objetos pointcut correspondiente.

Para procesar el fichero XML hemos hecho uso de una de las múltiples herramientas que permiten, a partir del esquema XML , generar código que es capaz de cargar en memoria el fichero, creando una estructura de árbol de elementos, ofreciendo una interfaz para trabajar con el código cargado.

El proceso seguido consiste en validar y cargar en memoria el fichero, y realizar un recorrido recursivo descendente desde el elemento de nivel superior, creando los elementos necesarios para formar un intérprete *PointCut* con la estructura vista anteriormente. El resultado es un modelo de objetos pointcut que representa a los puntos de corte expresados en el fichero.

15.4 Interfaces

Esta librería contiene las definiciones de las interfaces del Framework. En concreto:

- `IJoinPoint`. Interfaz que implementa la aplicación a adaptar (de forma automática a través de la instrumentación de código.
- `IReflection`. Interfaz que implementa la aplicación a adaptar (de forma automática a través de la instrumentación de código)
- `IServer`. Interfaz que implementa el servidor.
- `IMethodCall`. Interfaz que deben implementar los aspectos interesados en puntos de enlace de tipo invocación o ejecución de método o constructor.
- `IPropertyFieldAccess`. Interfaz que deben implementar los aspectos interesados en puntos de enlace acceso de lectura o escritura a campos o propiedades.

Esta librería es utilizada por todos los módulos del sistema para poder establecer la comunicación por medio de Remoting. Como comentamos en 13.4.3, para poder establecer una comunicación mediante Remoting y enviar un objeto de un proceso a otro, es necesario serializar el objeto en el origen, transmitirlo y deserializarlo en el destino. Para poder realizar el proceso de deserialización es necesario que el proceso que recibe el objeto tenga conocimiento de la estructura del elemento a deserializar, es decir, tiene que conocer sus metadatos. Uno de los métodos posibles para lograr esto es crear un ensamblado con la definición de la interfaz que cumple el objeto a deserializar y es éste el mecanismo elegido.

Además de las definiciones de las interfaces del sistema, en la librería se han incluido las implementaciones de `IJoinPoint` y de `IReflection`. Ambas implementaciones son incluidas en la aplicación en el momento de su instrumentación, de esta manera no es necesario que la aplicación las contenga anteriormente con lo que no se impone ninguna restricción. Ambas implementaciones hacen uso del patrón de diseño *Singleton* [Gamma94].

15.5 JPI

Este componente se encarga de instrumentar el código de la aplicación. Para obtener el código de la aplicación hacemos uso de la herramienta ILReader [ILReader] que, junto a las características introspectivas de la plataforma .NET, nos permite conocer la estructura de la aplicación y acceder a cada línea de código de la misma.

Una vez que tenemos acceso al código de la aplicación procedemos a volcar en un fichero una versión modificada según corresponda para ofrecer el MOP de reflexión computacional, así como la funcionalidad necesaria para que la aplicación se registre y desregistre en el Servidor, y tenga las interfaces `IJoinPoint` e `IReflection`.

Cuando se ha terminado de procesar el código de la aplicación y se ha obtenido un nuevo código adaptado al sistema, con la funcionalidad original más la añadida, es necesario procesar este nuevo código con el ensamblador del sistema para obtener un ejecutable, que es el que se ejecutará realmente en el sistema.

El volcado comienza con las referencias a librerías externas utilizadas en la aplicación, para luego pasar a declarar todas las clases presentes y por último especificar la implementación de las clases de la aplicación. Es en esta especificación de las clases donde se va a realizar la modificación de código para ofrecer el MOP reflectivo. En

concreto será dentro del código de los métodos de las clases donde se vaya a añadir código.

La forma de trabajar es ir analizando instrucción por instrucción de forma secuencial para comprobar si es una sombra de un punto de enlace (*joinpoint shadow)*. La forma de detectar cada una de las sombras existentes en el sistema se explicó detalladamente en el punto 12.3 por lo que no se repetirá aquí.

Cada instrucción tiene tres partes diferenciadas:

- Su offset, que identifica su etiqueta (*label*), y que sirve para identificar cada sentencia de código de forma única dentro del método de tal forma que se puedan realizar saltos a ella mediante las instrucciones de salto existentes en IL.
- Su código de operación, que se corresponde con un nombre de operación. Identifica la operación en concreto (invocación a un método, carga de un campo en la pila, etc.).
- Operandos de la instrucción. Dependiendo de la instrucción a la que haga referencia puede representar valores muy dispares. Por ejemplo en una invocación a un método sería la signatura del método al que se invoca. En una carga de un campo en la pila, sería el nombre del campo. En un salto sería el nombre de la etiqueta de la instrucción a la que se salta.

Es decir, una instrucción en IL tiene la siguiente forma

```
Etiqueta: nombre_operación operandos
```

Como vimos en el punto 12.3 existen sombras que no se corresponden con una instrucción sino que se corresponden con una zona del código. Un ejemplo podría ser la ejecución de un método, cuya sombra de comienzo es el punto anterior a la primera instrucción del método. En casos como éste la inyección de código para los tiempos *before* y *around* debe realizarse en ese punto exacto, delante de la primera instrucción.

Otras sombras se corresponden con una instrucción (por ejemplo el acceso de lectura a un campo), y en estos casos se debe realizar la inyección entre la etiqueta y el nombre de la operación tiempo *before*, y las primeras partes de los tiempos *around* y *after*. Y justo después de los operandos para las segundas partes de los tiempos *around* y *after*. En 12.5 se puede ver el pseudocódigo del código que se debe inyectar en cada tiempo (*after, before y around),* para un punto de enlace de ejemplo.

La razón de inyectar código entre la etiqueta y la propia operación es garantizar que siempre que se alcance este punto, ya sea por la ejecución secuencial de operaciones o porque se haya llegado a él a través de un salto desde otra zona del código, se ejecute el código inyectado (si se inyectase antes de la etiqueta, cuando se accediese al punto mediante un salto no se ejecutaría).

Además de la inyección de código para ofrecer el MOP, durante el procesamiento del código de la aplicación se realizan otras inyecciones de código en la misma.

En primer lugar es necesario añadir referencias a las librerías externas que se van a utilizar. En concreto, es necesario añadir referencias a la librería del sistema (Interfaces) y a `System.Runtime.Remoting`. La referencia a la librería de Remoting sólo se debe añadir en el caso de que la aplicación no la tuviese ya referenciada, y sirve para poder hacer uso de Remoting, necesario para que la aplicación pueda relacionarse con el resto del sistema. La librería Interfaces es propia del sistema y contiene la definición de las interfaces del Framework, así como la implementación de las interfaces `IJoinPoint` e `IReflection`.

Otra acción realizada por el JPI es preparar todas las clases de la aplicación para que puedan ser serializadas y transmitidas a través de Remoting. En el punto 15.7 se explica detalladamente este proceso.

También es necesario añadir el código necesario para que la aplicación se registre en el Servidor. Este proceso de registro debe ser lo primero que ejecute la aplicación, por lo que se insertará al comienzo del método que figure como `entrypoint` de la aplicación. El código que se inyecta consiste en la apertura de los canales de Remoting para poder conectarse al Servidor, y la invocación al método `IServer.recordApplication` que registra la aplicación en el servidor, pasándole como parámetros una identificación única de la aplicación y una referencia a la implementación de `IjoinPoint` que será usada por el Servidor para registrar/desregistrar aspectos.

Por último, es necesario que cuando la aplicación abandone la ejecución se desregistre del Servidor. Para ello se añade código en la finalización del método marcado como `entrypoint` de tal manera que cuando se alcance ese punto, se invoque al método del Servidor `IServer.deleteApplication` pasándole como parámetro la identificación de la aplicación para que proceda a eliminar la aplicación de la lista de aplicaciones disponibles. La finalización del método se localiza mediante la instrucción `ret` (que puede aparecer varias veces en el código del método) o por alcanzar la última instrucción del método.

La aplicación, cuando esté ejecutándose, debe tener conocimiento de los puntos de enlace que ofrece, para así poder proceder a su activación y desactivación, y también debe poder registrar los aspectos que han solicitado ser notificados ante la ejecución de un punto de enlace. Para lograr esto se ha creado una estructura que almacena los puntos de enlace y referencias a los aspectos registrados asociados a ellos.

Esta estructura consiste en un conjunto de tablas *Hash*, ordenadas jerarquizadamente en forma de árbol. Cada una de las "ramas" del árbol identifica un punto de enlace existente en la aplicación, siendo sus hojas (los nodos terminales) referencias a los aspectos que hayan solicitado ser notificados de la ejecución del punto de enlace concreto.

En la Figura 58 podemos ver esta organización. El tipo de JoinPoint puede ser uno de los existentes en el sistema (MethodCall, ConstructorExecution, FieldReference, etc.). Mediante el *namespace*, la clase y el miembro identificamos a quién se está haciendo referencia. El miembro se refiere al nombre del método, constructor, campo, etc. El tiempo es uno de los establecidos en 11.2, es decir, *before, after* y *around*. Los

aspectos serán las referencias a los aspectos que hayan solicitado registrarse en el punto de enlace.

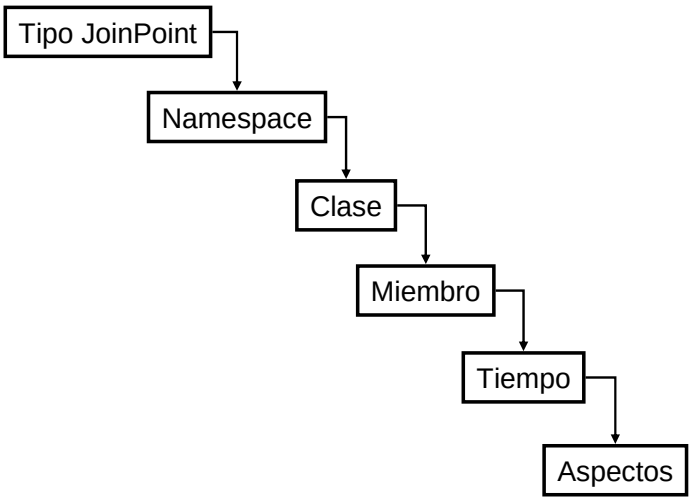

Figura 58: Jerarquía de la tabla *Hash* de puntos de enlace

Durante el proceso de inyección de puntos de enlace se crean las diversas ramas de este árbol, en base al código inyectado. Al finalizar el proceso de inyectado el conjunto de tablas es serializado y almacenado en fichero, de tal forma que esté disponible para que la aplicación durante la ejecución pueda deserializarlo y tener conocimiento de los puntos de enlace inyectados.

El principal objetivo de este prototipo es ratificar la consecución de los objetivos impuestos a la Tesis, utilizando las técnicas descritas a lo largo de la misma. Es por ello, que la implementación del prototipo no ofrece aquellas funcionalidades que no aportan un elemento innovador al sistema, sino que implican una mera extensión de otras funcionalidades más importantes que sí han sido desarrolladas. En concreto, respecto al JPI, no se han implementado las siguientes partes del sistema:

- No se han implementado los puntos de enlace de tipo manejo de excepciones. No se realiza inyección de código alguna para este caso.
- La identificación de un punto de enlace de tipo invocación o ejecución de método o constructor no tiene en cuenta sus parámetros ni el tipo de retorno. Esto implica que no se puede restringir la identificación de los puntos de enlace en base a ellos. Tanto el lenguaje como los componentes para procesarlo sí que lo tienen en cuenta. Es decir, donde no se ha implementado ha sido únicamente en la inyección de código.

15.6 Servidor

El Servidor implementa la interfaz `IServer` y la ofrece vía Remoting, con el fin de que las aplicaciones puedan registrarse y desregistrarse en él, y que los aspectos puedan solicitar adaptaciones de las aplicaciones registradas.

Como se comentó en 13.3 el Servidor debe ser el lugar donde se implemente la seguridad del sistema, pero tanto la elección del mecanismo de seguridad, como su propia implementación están fuera del alcance de esta Tesis, por lo que no se ha contemplado en la implementación actual.

El Servidor tiene que mantener información sobre las aplicaciones que están ejecutándose en el sistema, es decir, disponibles para ser adaptadas. Para ello se ha utilizado una tabla Hash, en la que la clave se corresponde con el GUID que identifica de forma única a cada aplicación, y el valor se corresponde con la referencia a la propia aplicación, recibida en el momento del registro y que será usada para proceder a registrar y desregistrar aspectos.

Cuando un aspecto solicita adaptar una aplicación envía un fichero XML con la especificación de los puntos de corte que solicita. El Servidor, haciendo uso de XML2PointCut, crea una estructura PointCut que representa los puntos de corte del fichero y es capaz de interpretarlos. A continuación haciendo uso de la referencia a la aplicación a adaptar procede a solicitar el registro del aspecto, mediante la implementación de la interfaz `IJoinPoint`.

15.7 Limitaciones de la Implementación

Tal y como se comento en el punto13.4 el sistema se basa en el uso de Remoting para realizar las comunicaciones entre procesos. Esto tiene una serie de implicaciones para el sistema.

Paso por valor y paso por referencia

Existen dos formas de pasar objetos a través de Remoting: paso por valor (*Marshal By Value* MBV) y paso por referencia (*Marshal By Reference* MBR). Puesto que el paso por valor crea una copia del objeto cualquier modificación que se haga sobre la copia no tendrá efecto sobre el original. En nuestro sistema necesitamos poder realizar modificaciones sobre los elementos originales, no sobre copias, para que así el comportamiento de la aplicación varíe. Es decir en nuestro sistema el paso debe realizarse por referencia, para que las modificaciones que se hagan sobre cualquier objeto de una aplicación por parte de un aspecto que la adapta tengan efecto real en la aplicación.

El paso de objetos a través de Remoting consiste en un proceso de serialización en el origen y deserialización en el destino. Para que un objeto sea serializable es necesario marcarlo como tal para que a la hora de compilarlo se genere la información necesaria para poder realizar el proceso. Existen dos formas de marcar a las clases para que sean serializables. La primera consiste en asignarles el atributo [Serializable] (también es válido que implementen la interfaz ISerializable), pero en este caso los objetos se pasan por valor. La otra forma consiste en que la clase herede de `MarshalByRefObject` y en este caso el paso se realiza por referencia.

Durante el proceso de inyección de código es necesario preparar todas las clases de la aplicación para que puedan serializarse. Marcarlas con el atributo [Serializable] es posible en todos los casos pero haría que el paso fuese por valor, lo que no sirve a nuestras necesidades. Hacer que las clases hereden de `MarshalByRefObject` es la solución pero no siempre es posible aplicarla. En el lenguaje intermedio de la plataforma

.NET (IL) no existe la herencia múltiple por lo que, si existe una clase que ya hereda de otra, no es posible hacer que herede de la clase que nos interesa. Existen soluciones para resolver este problema, muchas de ellas comerciales.

En nuestro prototipo aplicamos la serialización por referencia en todas las clases en las que se pueda realizar directamente, es decir, en todas las que puedan heredar de `MarshalByRefObject`.

Paso de Threads

En Remoting no está soportado el paso de *threads* de un proceso a otro, generando una excepción. Por esto no haremos instrumentación de código de elementos que pertenezcan directa o indirectamente (mediante herencia) al *namespace* `System.Threading`. Es decir, si hay una invocación a un método de un elemento perteneciente a ese *namespace* no se realiza la instrumentación de código para ese punto de enlace.

CAPÍTULO 16
ÁMBITO DE APLICACIÓN

El Desarrollo de Software Orientado a Aspectos (DSOA) es un paradigma emergente que empieza a gozar de cierta popularidad, apareciendo cada vez más herramientas que lo soportan. Se han realizado estudios [Loughran06] que analizan las incumbencias más importantes en las que el DSOA puede aportar mejoras. Ejemplos pueden ser la persistencia, la seguridad, movilidad o la coordinación, entre otros.

En general, en cualquier problema donde sea beneficioso el empleo del DSOA también lo sería el empleo de su vertiente dinámica, pero el soportar el dinamismo tiene un coste en el rendimiento de los sistemas, por lo que si el rendimiento es un punto más importante en el sistema que la dinamicidad puede ser recomendable utilizar sistemas dinámicos únicamente en el caso de que el problema a resolver así lo requiera.

A continuación vamos a ver algunos ejemplos donde puede ser beneficioso el empleo de un sistema dinámico con las características del diseñado en esta Tesis. No mencionaremos ejemplos donde no se necesiten las características dinámicas, puesto que en esos casos podría ser más adecuado el uso de sistemas estáticos en aras de un mejor rendimiento.

16.1 Aplicaciones que no Pueden Detenerse

Existen multitud de aplicaciones que tienen que estar funcionando 24 horas al día los 365 días del año, por lo que no pueden detenerse para ser modificadas. Ejemplos de éstas pueden ser: aplicaciones de entidades financieras, sistemas de monitorización en tiempo real, sistemas empotrados, servidores Web, etc. La forma de realizar actualizaciones en estas aplicaciones suele pasar por replicar el sistema (con el consiguiente coste económico) o por detener la aplicación (lo que lleva a un cese del servicio lo que suele equivaler a un coste económico, por ejemplo en un portal Web que no pueda recibir visitas durante cierto tiempo).

En casos así es donde cobra especial importancia la posibilidad de que las adaptaciones se puedan llevar a cabo de forma dinámica.

16.1.1 Adaptación Dinámica de Aplicaciones

La adaptación dinámica de estas aplicaciones permitiría cambiar su comportamiento sin la necesidad de detener y volver a iniciar su ejecución, lo que supondría un claro beneficio.

El patrón Strategy [Gamma94] ayuda a diseñar aplicaciones adaptables, que respondan ante diversas situaciones previstas durante el desarrollo, sin embargo, no es posible prever por anticipado todas las formas en las que una aplicación puede ser adaptada [Bergel05].

Es aquí donde la programación orientada a aspectos dinámica tiene utilidad. Gracias a ella es posible modificar el comportamiento de un programa mientras éste sigue ejecutándose. A continuación mostramos una serie de ejemplos de casos en los que la aplicación de la POA dinámica puede ser beneficiosa.

Sistema de precarga en un Web caché

Debido al gran tráfico que origina el protocolo HTTP en Internet el Web caché es esencial para reducir tiempos de acceso, consumo de ancho de banda, etc. Si se dispusiese de un mecanismo de precarga en el Web caché, de tal forma que se anticipase a las peticiones de los usuarios, el beneficio sería aún mayor [Jacobson98]. Para conseguir esto, la precarga debe ajustarse al usuario y a la aplicación que esté sirviendo el servidor Web en cada momento, por lo que es necesario cambiar la política de precarga de forma dinámica para adecuarse a los cambios en el entorno.

Como hemos comentado anteriormente, el prever todos los posibles escenarios en los que pueda ejecutarse el sistema, contemplando distintos perfiles de usuarios y de aplicaciones, es del todo imposible, por lo que tampoco es posible construir un sistema con una serie de políticas precargadas que se adecuen a todos los escenarios y que se fuesen activando o desactivando según fuese necesario. Es decir, para obtener un sistema optimizado en todo momento, es necesario que se puedan crear políticas y que éstas puedan añadirse al sistema de caché en sustitución de las antiguas, todo esto sin detener la ejecución del sistema.

La incumbencia de la precarga es transversal a la estructura de la caché, por lo que resolver el problema mediante POA es una solución natural y efectiva, para ello es necesario disponer de un sistema dinámico, con el fin de poder añadir y eliminar políticas en tiempo de ejecución sin tener que detener el servidor y sin tener que haberlas previsto durante el desarrollo [Segura–Devillechaise03].

Persistencia

Diversos autores han identificado la POA como una técnica adecuada para manejar la incumbencia de la persistencia [Loughran06][Rashid01][Rashid03][Soares02]. Considerando la persistencia como una incumbencia típica en la mayoría de las aplicaciones, la separación de ésta del código principal del sistema permite el desarrollo de programas sin tener en cuenta sus requisitos persistentes, añadiendo y adaptando éstos en fases posteriores.

En general, se ha utilizado la POA estática para separar el aspecto de persistencia de la funcionalidad de la aplicación, pero una adaptación dinámica de los distintos parámetros relacionados con la persistencia de un sistema, así como su asignación y eliminación en tiempo de ejecución, son relevantes en sistemas adaptables y adaptativos a contextos surgidos en tiempo de programación [Lopez06].

Por todo ello el uso de la POA dinámica es una técnica muy adecuada para manejar la incumbencia de la persistencia en las aplicaciones.

Monitorización de aplicaciones

Cuando se desarrolla e implementa una aplicación es imposible decir a priori qué se va a necesitar monitorizar en un futuro ni cuándo se va a necesitar realizar la monitorización.

Preparar la aplicación para poder monitorizar todas las posibilidades, además de resultar imposible en aplicaciones de cierta envergadura, implica una carga de trabajo que normalmente no se puede asumir.

De igual modo, monitorizar de forma constante una aplicación en previsión de poder necesitar la monitorización en algún momento dado conlleva una penalización continua en el rendimiento de la aplicación que no es admisible en la mayoría de las circunstancias.

Mediante el uso de la POA dinámica se puede adaptar la aplicación en ejecución, monitorizando lo que se necesite únicamente en el momento en que se necesite, y pudiendo eliminar esa monitorización cuando ya no sea necesaria. De este modo no es necesario preparar la aplicación para todas las posibilidades, ni tiene que estar ejecutándose de forma monitorizada cuando no se necesite.

16.1.2 Corrección de Errores

En sistemas que deben permanecer funcionando de forma continua la existencia de un error (o de una ineficiencia) puede suponer un problema de difícil solución, ya que aunque dispongamos del código fuente para localizar y solucionar el error (lo que no siempre es así, ya que el error puede deberse a una librería desarrollada por terceros, por ejemplo), la sustitución de la aplicación defectuosa por su versión corregida implicaría un cese de su actividad, lo que en ocasiones no es admisible (por ejemplo el servidor de un portal Web, el sistema informático de un banco, etc.).

Mediante la POA dinámica es posible solucionar este problema sin necesidad de paralizar la actividad del sistema.

Como primer paso, es posible utilizar la POA dinámica para detectar dónde se produce el error (o un funcionamiento no eficiente) en condiciones reales de funcionamiento. Muchas veces, un error no se ha detectado durante las pruebas en desarrollo por no ser capaces de prever las condiciones en las que se ejecutará realmente el sistema. Por ejemplo, en un sistema con pocos clientes puede ser soportable el abrir una conexión con una base de datos para cada petición que haga un cliente, pero en el caso de que los clientes fueran cientos de miles no se podría soportar el establecer una nueva

conexión a la base de datos por cada petición ya que representaría un coste enorme en el tiempo de ejecución (abrir una conexión es costoso en tiempo), y en los recursos consumidos en el propio servidor de bases de datos. Una opción mejor sería la existencia de un pool de conexiones a la base de datos. El poder realizar una traza en el entorno de producción, sin tener que detener el sistema, nos posibilita detectar el qué está ocurriendo y dónde (en qué parte del código).

Una vez detectado el problema, se puede proceder a su solución mediante la sustitución dinámica de las partes incorrectas de la aplicación, sin necesidad de detenerla, con lo que se mantiene el servicio de forma ininterrumpida y se consigue solucionar el problema. En el ejemplo anterior de las conexiones a la base de datos se podría sustituir la creación, por parte de la aplicación, de una nueva conexión a la base de datos por una petición a un pool de conexiones (que habría que implementar) para que nos provea con una conexión ya abierta.

En el caso de no disponer del código fuente de la aplicación cobra más fuerza la necesidad de observar el comportamiento de la aplicación en condiciones reales de carga, puesto que no se puede analizar el código en busca de errores.

16.2 Desarrollo de un Sistema Autonómico

Los sistemas autonómicos [Autonomic] son sistemas que son capaces de realizar mantenimiento de sí mismos. La forma de realizar esto consiste, generalmente, en monitorizar una serie de variables del entorno donde se están ejecutando y, cuando detectan que algo no se está comportando de la forma deseada o el entorno ha cambiado de alguna forma tal que implica que el sistema no se está ejecutando de forma óptima, se reconfiguran a sí mismos para corregir el problema detectado.

IBM [Horn03] establece que para que un software pueda considerarse autonómico tiene que cumplir cuatro características:

- autoconfigurarse: el sistema se adapta dinámicamente a cambios en el entorno;
- autosanarse: el sistema descubre, diagnostica y reacciona ante problemas (errores);
- autooptimizarse: el sistema monitoriza y afina su funcionamiento;
- autoprotegerse: el sistema anticipa, detecta y se protege ante ataques.

En general, la mayoría de los sistemas denominados autonómicos sólo cumplen un subconjunto de estas características.

Existen diversos estudios que identifican la POA dinámica como un mecanismo para conseguir construir un sistema autonómico [Dantas03][Greenwood04][Yang02]. En estos estudios se identifican una serie de ventajas que aporta el uso de la POA de forma dinámica y una serie de características que debe cumplir el sistema orientado a aspectos para que pueda utilizarse.

Como hemos visto, uno de los requisitos básicos para que un sistema se considere autonómico es su capacidad de adaptarse dinámicamente ante cambios en el entorno.

La POA en su vertiente dinámica es una opción natural para lograr esto, puesto que es ese precisamente su fin: “adaptar una aplicación de forma dinámica”.

Otro aspecto a considerar es la posibilidad de eliminar aspectos/adaptaciones que ya no se necesiten, con el fin de que el rendimiento del sistema no se vea penalizado de forma innecesaria.

La mayoría de las adaptaciones que realizan los sistemas autonómicos tienen que ver con incumbencias transversales como pueden ser la seguridad, la persistencia o el caché. La POA separa perfectamente estas incumbencias, permitiendo modificarlas de forma individual y sencilla.

Una característica deseable en un sistema que ofrezca soporte a la POA para construir un sistema autonómico es la existencia de “aspectos de aspectos” [Greenwood04]. Debido a la naturaleza dinámica de estos sistemas puede ser necesario realizar cambios no previstos en el desarrollo del sistema, y esto también puede ser cierto para los aspectos que estén modificando el sistema en un momento dado, por lo tanto es deseable que se puedan adaptar los propios aspectos que están adaptando a la aplicación.

Nuestro sistema cumple con todas estas características mencionadas por lo que podría emplearse para implementar un sistema autonómico.

16.3 Adaptación y Localización Dinámica de Reglas de Negocio

Cuando se crea un sistema de información las reglas de negocio se introducen en el código, normalmente de forma fragmentada a lo largo de diferentes partes o módulos del sistema. Habitualmente los sistemas sufren cambios a lo largo del tiempo, ya sea para ampliar su funcionalidad, corregir errores o adaptarse a nuevos requerimientos. Con estos cambios es fácil que las reglas de negocio queden ocultas entre las sucesivas modificaciones de código (y muchas veces a esto se suma el hecho de que no existe una documentación actualizada).

Si se necesita modificar una regla de negocio o verificar que la implementación realizada se corresponde con lo deseado es necesario localizar todo el código que se refiere a ella, que normalmente estará disperso.

Conseguir localizar las reglas de negocio en el código de una aplicación no es sencillo. El uso de un sistema con soporte a la POA de forma dinámica puede ayudar a localizar estas reglas. Durante la ejecución del programa podemos activar aspectos que identifiquen dónde se encuentra la ejecución del programa y qué está haciendo exactamente. Si estos aspectos los activamos inmediatamente antes de lanzar el proceso cuyas reglas de negocio queremos localizar en el código del programa, podremos obtener información de qué partes de la aplicación se han ejecutado y de qué forma.

16.4 Mantenimiento del Software

En muchos casos, cuando se entrega al cliente una aplicación software y éste comienza a usarla surgen incidencias por parte del cliente acerca del funcionamiento. En estos casos puede ocurrir que existan fallos en el programa o que los fallos sean por parte de los usuarios en el manejo de la aplicación.

Ante esta situación, aparecen una serie de preguntas que hay que responder como ¿Qué estaba haciendo exactamente el cliente?, ¿Cuáles eran las variables del entorno?, ¿Cómo respondió el programa?, etc. Para poder contestar a estas preguntas lo más usual es modificar la aplicación para añadir código que realice una traza y así poder obtener información precisa para responderlas. Pero surgen otras cuestiones que hay que resolver ¿En qué módulos es necesario poner la traza?, ¿hasta qué nivel de detalle necesitamos?, etc. Estas preguntas no son fáciles de contestar. Examinar una traza de una aplicación no es una tarea sencilla, si el nivel de detalle es muy alto la cantidad de información generada hará difícil su tratamiento, si el nivel es bajo puede no indicar el problema. Además, añadir código de traza a toda la aplicación es un proceso costoso.

Mediante la POA se puede añadir la funcionalidad de traza a la aplicación de una manera sencilla, seleccionando los módulos en los que estemos interesados mediante puntos de corte, con lo que se facilitan estas tareas. Si este proceso se realiza de forma estática implica modificar la aplicación base, generar los ejecutables, volver a instalársela al cliente y ejecutarla para observar los resultados. Este proceso (modificar–generar–instalar) debería repetirse hasta que se hayan solucionado los problemas, lo que ocasiona molestias al cliente.

Si hiciésemos uso de la POA de forma dinámica se podría añadir la funcionalidad de traza sin que el cliente tuviese que hacer nada, pudiendo activarla y desactivarla en los distintos módulos según fuese necesario, simplificando así el proceso.

CAPÍTULO 17
VALIDACIÓN

En este capítulo procederemos a realizar una evaluación cualitativa y una evaluación cuantitativa del sistema presentado en esta Tesis.

Para realizar la evaluación cualitativa haremos uso de los requisitos establecidos con anterioridad en el capítulo 2. Todos los sistemas presentados y estudiados en esta memoria han sido evaluados en función de estos criterios. Para llevar a cabo la valoración de un modo comparativo, evaluaremos de forma paralela un grupo de sistemas reales, adicionalmente al presentado.

Por último realizaremos una evaluación cuantitativa del coste que tiene el sistema tanto en tiempo de ejecución como en consumo de memoria.

17.1 Evaluación Cualitativa

Para evaluar el conjunto de sistemas, estableceremos una clasificación de los requisitos y estudiaremos éstos tanto en función en esta categorización, como de un modo global (absoluto). Una vez cuantificadas y representadas las distintas evaluaciones, analizaremos los beneficios aportados por nuestro sistema en comparación con el resto de los estudiados.

17.1.1 Comparativa de Sistemas

A la hora de comparar nuestro sistema con otros existentes, hemos seleccionado, del conjunto global de sistemas estudiados en esta Tesis, los más representativos y próximos a satisfacer los requisitos estipulados. De esta forma, nos centraremos en:

- AspectJ. Sistema más extendido y con mayor base de usuarios, en su última versión ofrece características dinámicas. [AspectJ]
- PROSE. Sistema que ofrece soporte a la Programación Orientada a Aspectos de forma dinámica, existen distintas versiones del mismo, nosotros haremos uso para la comparación de la más avanzada [Nicoara05].
- JBoss AOP. Framework que soporta la POA en el entorno del servidor de aplicaciones JBoss. [JBossAOP]
- Rapier–LOOM.Net. Sistema que ofrece soporte a la POA de forma dinámica e independiente del lenguaje, basado en la plataforma .NET. [Schult02b]

- AOP Engine.NET. Sistema que ofrece soporte a la POA de forma dinámica e independiente del lenguaje. [Frei04]
- JAsCo. Sistema que soporta POA de forma dinámica. Está orientado a dar soporte al software basado en Componentes y específicamente al campo de los Servicios Web. [Suvee03]
- READY AOP. Es el nombre con el que nos referiremos al sistema propuesto en esta Tesis. Las siglas del nombre se corresponden con *REAlly DYnamic AOP*.

17.1.2 Criterios de Evaluación

La evaluación de los distintos sistemas es básicamente cualitativa. El objetivo principal de esta Tesis es la consecución de un sistema que ofrezca soporte al Desarrollo de Software Orientado a Aspectos de forma dinámica e independiente del lenguaje. El conjunto de criterios es básicamente el descrito en el capítulo 2. Los clasificaremos y enumeraremos de la siguiente forma:

A. Criterios generales
    1. Sistema completamente dinámico. La adaptación de las aplicaciones debe tener lugar en tiempo de ejecución y sin que sea necesario un conocimiento previo de la adaptación a realizar. Además debe permitir la desactivación en ejecución de aspectos que ya no sean necesarios.
    2. Independencia del lenguaje de programación. Debe ser indiferente el lenguaje de programación en el que se implemente la aplicación y los aspectos que la adapten.
    3. Interacción entre distintos lenguajes de programación. Las aplicaciones y los diferentes aspectos que las adapten pueden estar implementados en distintos lenguajes de programación.
    4. Sistema de propósito general. El sistema ha de estar diseñado para poder ser utilizado en cualquier tipo de problema.
    5. Amplio conjunto de puntos de enlace. El conjunto de puntos de enlace que soporta el sistema ha de ser todo lo amplio posible.
    6. Formas de adaptar un punto de enlace. Evaluaremos los posibles tiempos o formas de adaptar un punto de enlace que soporte el sistema.
    7. Expresividad del lenguaje de puntos de corte. El lenguaje de puntos de corte debe permitir la selección de los puntos de enlace de una manera sencilla a la vez que potente.
B. Criterios de la plataforma
    1. Independencia de la plataforma. El sistema podrá ser implementado sobre cualquier plataforma.
    2. Plataforma comercial ampliamente difundida. El sistema debe funcionar sobre una plataforma comercial con amplia aceptación.
    3. Plataforma con buen rendimiento. El rendimiento de la plataforma sobre la que se implementa el sistema debe ser alto.
    4. Sistema estándar. El sistema debe estar implementado sobre una plataforma estándar, sin modificaciones.
C. Criterios correspondientes a las aplicaciones

1. No necesidad del código fuente de las aplicaciones. Para realizar las adaptaciones de las aplicaciones no es necesario disponer de su código fuente.
2. No imposición de condiciones a las aplicaciones. Cualquier aplicación debe poder hacer uso del sistema, sin restricciones ni modificaciones.

D. Criterios correspondientes a los aspectos
   1. Adaptación simultánea de aplicaciones. Una aplicación puede ser adaptada de forma simultánea por más de un aspecto.
   2. Mecanismo de coordinación de múltiples aspectos. Cuando varios aspectos adaptan en el mismo punto de enlace y de forma simultánea a una aplicación el usuario podrá elegir el orden de ejecución de los mismos.
   3. Aspectos como aplicaciones. Los aspectos deben ser aplicaciones normales del sistema, pudiendo beneficiarse de las características del mismo de igual forma que las aplicaciones.
   4. Uso de un lenguaje estándar. El sistema hará uso de lenguajes estándar ya existentes, sin necesidad de realizar extensiones.
   5. Separación del código de los aspectos del de los puntos de corte. El código de los aspectos deberá encontrarse separado del de la definición de los puntos de corte para conseguir una mejor reutilización del código.

17.1.3 Evaluación

La evaluación del sistema será equiponderada respecto a los criterios descritos. La consecución plena de un objetivo será ponderada con la unidad; su carencia, con un valor nulo; los factores intermedios serán cuantificados en la mayoría de los casos cualitativamente –para conocer una justificación somera de la evaluación, consúltese el punto 17.1.4.

Criterios Generales

Criterio	AspectJ	PROSE	JBoss AOP	Rapier LOOM.Net	AOP Engine.Net	JAsCo	READY AOP
A.1	0,2	1	0,5	0,2	1	1	1
A.2	0	0	0	1	1	0	1
A.3	0	0	0	1	1	0	1
A.4	1	1	0,5	1	1	0,5	1
A.5	1	0,5	1	0	0	0,5	1
A.6	1	0,5	0,3	1	0,4	1	1
A.7	1	1	1	1	1	1	1
Total	4,2	4	3,3	5,2	5,4	4	7

Representación parcial de los criterios generales del sistema:

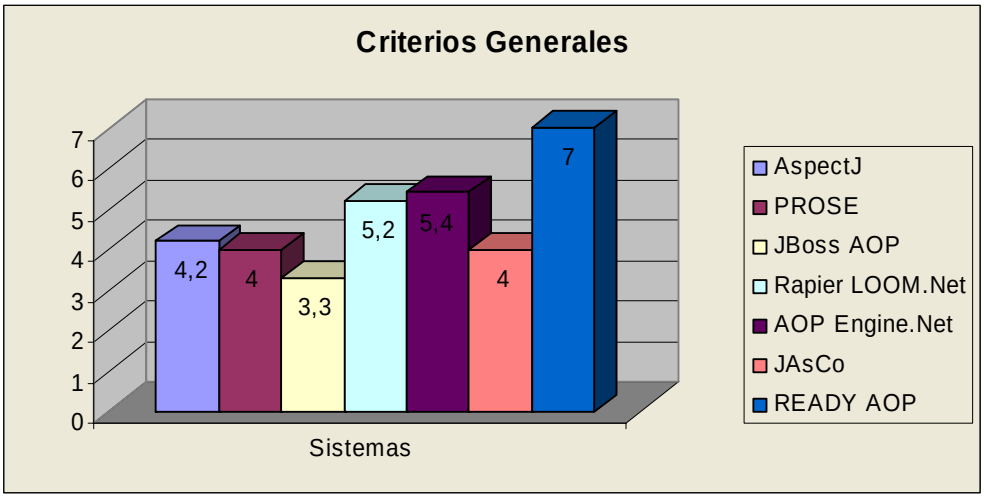

Figura 59: Evaluación de los criterios generales del sistema.

Criterios referidos a la plataforma

Criterio	AspectJ	PROSE	JBoss AOP	Rapier LOOM.Net	AOP Engine.Net	JAsCo	READY AOP
B.1	1	1	1	1	0	1	1
B.2	1	1	1	1	1	1	1
B.3	1	1	1	1	1	1	1
B.4	1	0	1	1	1	1	1
Total	4	3	4	4	3	4	4

Representación parcial de los criterios referidos a la plataforma sobre la que se construye el sistema:

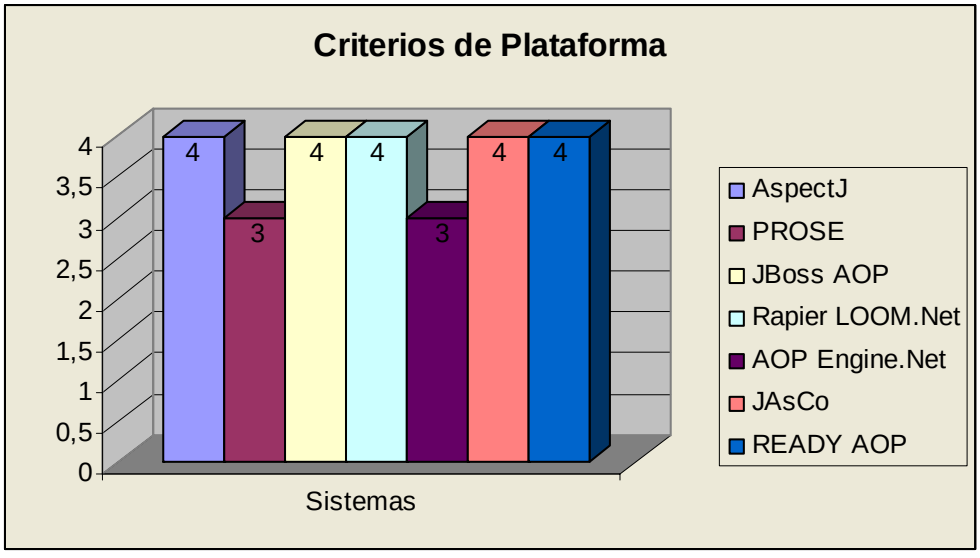

Figura 60: Evaluación de los criterios de la plataforma utilizada.

Criterios correspondientes a las aplicaciones

Criterio	AspectJ	PROSE	JBoss AOP	Rapier LOOM.Net	AOP Engine.Net	JAsCo	READY AOP
C.1	1	1	1	0	1	1	1
C.2	1	1	1	0	1	1	1
Total	2	2	2	0	2	2	2

Representación parcial de los criterios concernientes a las aplicaciones:

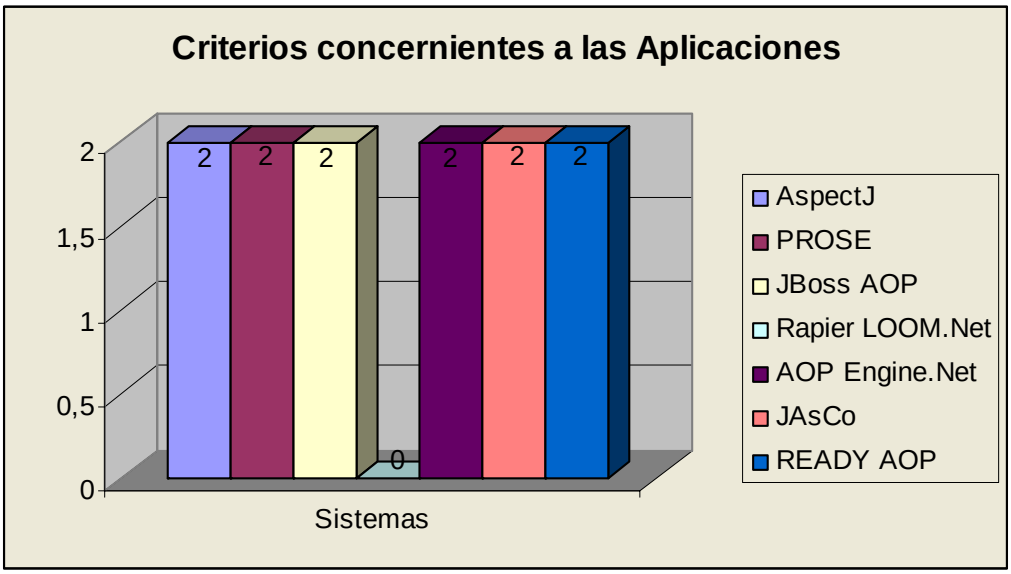

Figura 61: Evaluación de los criterios concernientes a las aplicaciones.

Criterios concernientes a los Aspectos

Criterio	AspectJ	PROSE	JBoss AOP	Rapier LOOM.Net	AOP Engine.Net	JAsCo	READY AOP
D.1	1	1	1	1	0	1	1
D.2	1	1	1	0,3	0	1	1
D.3	0	0	0	0	0	1	1
D.4	0	1	1	1	1	0	1
D.5	0	0	1	0	0	0	1
Total	2	3	4	2,3	1	3	5

Representación parcial de los criterios concernientes a los Aspectos en el sistema:

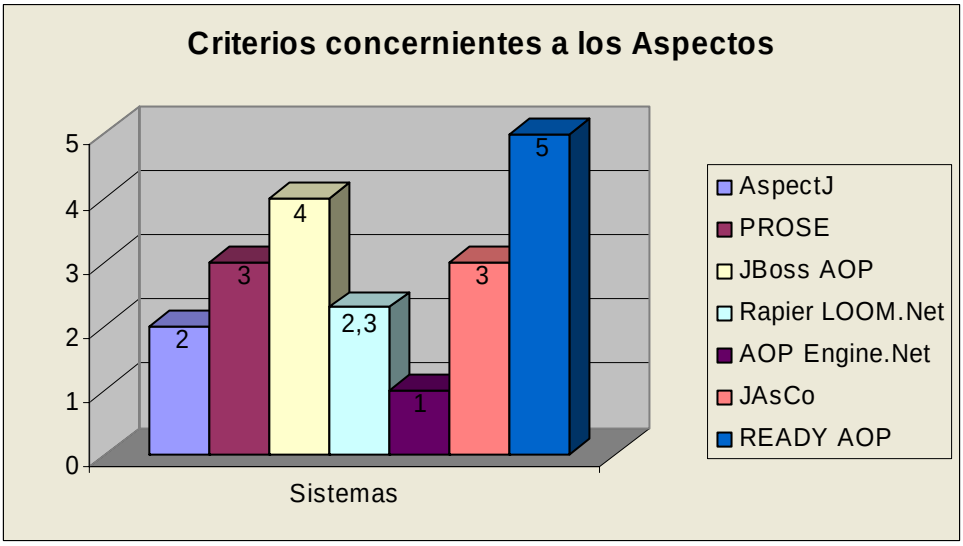

Figura 62: Evaluación de los criterios concernientes a los aspectos.

Evaluación global de todos los criterios

Criterio	AspectJ	PROSE	JBoss AOP	Rapier LOOM.Net	AOP Engine.Net	JAsCo	READY AOP
A	4,2	4	3,3	5,2	5,4	4	7
B	4	3	4	4	3	4	4
C	2	2	2	0	2	2	2
D	2	3	4	2,3	1	3	5
Total	12,2	12	13,3	11,5	11,4	13	18

Representación global de la evaluación de todos los criterios definidos, equiponderando todas las mediciones:

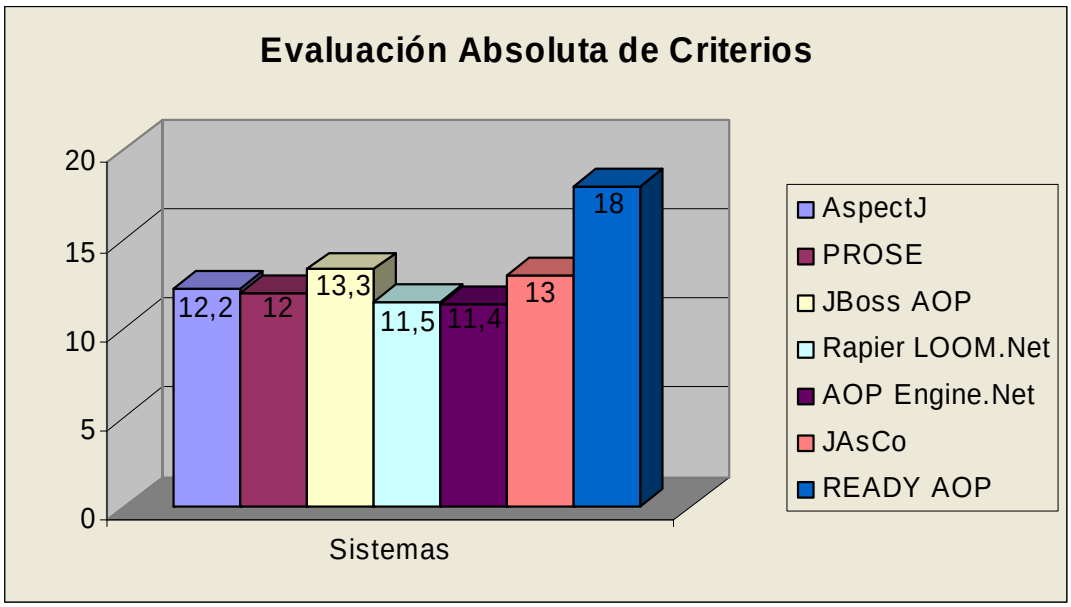

Figura 63: Evaluación de los criterios establecidos, de un modo absoluto.

17.1.4 Justificación de las Evaluaciones

A continuación describiremos, de forma somera, la evaluación de los distintos criterios seleccionados previamente para cada uno de los sistemas estudiados. En caso de desear una ampliación de las justificaciones, puede consultarse el capítulo 5 donde se evalúan en profundidad los sistemas comparados.

A. Criterios generales del sistema
   1. AspectJ y Rapier LOOM.Net sólo permiten adaptar a una aplicación mediante aspectos que ya existiesen en el momento de la carga de clases, aunque se puede diferir su activación, por lo que no permiten adaptar a una aplicación mediante aspectos que hayan surgido con posterioridad al comienzo de la ejecución. Tampoco permiten desactivar o eliminar aspectos que hayan sido tejidos con la aplicación, lo que impide la adaptación dinámica. JBoss permite adaptar a una aplicación con aspectos no previstos inicialmente únicamente en aquellas clases que ya estuviesen siendo adaptadas, por lo que si surgiese la necesidad de adaptar una clase que no estuviese siendo adaptada desde el comienzo de la ejecución del programa no sería posible realizarlo. El resto de sistemas soportan una verdadera adaptación dinámica.

2. Únicamente Rapier LOOM.Net, AOP Engine.Net y READY AOP son independientes del lenguaje, por estar construidos sobre una plataforma que lo es. El resto de sistemas trabajan únicamente con el lenguaje Java (ya sea el lenguaje puro o con ampliaciones del estándar, lo que limita aún más la independencia), por lo que tanto las aplicaciones como los aspectos deben ser implementados en ese lenguaje.
3. Derivado del punto anterior, la interacción entre distintos lenguajes de programación sólo es posible en los mismos tres sistemas que están basados en la plataforma independiente del lenguaje .NET.
4. Aunque en principio todos los sistemas evaluados son de propósito general, JBoss esta integrado en un servidor de aplicaciones, por lo que sólo es útil en aplicaciones que puedan ser servidas por un servidor. De igual modo JAsCo esta basado en componentes y orientado específicamente a los servicios Web, por lo que va dirigido a aplicaciones que encajen en esta filosofía.
5. Rapier LOOM.Net soporta únicamente las invocaciones a métodos como puntos de enlace, mientras que AOP Engine.Net soporta únicamente las ejecuciones de métodos. PROSE no soporta invocaciones a métodos, pero soporta el resto de puntos de enlace estándar, mientras que JAsCo carece de tratamiento de excepciones. El resto de sistemas soportan todos los puntos de enlace estándar.
6. JBoss AOP sólo permite el *around* como forma de adaptar a una aplicación, mientras que PROSE, aunque permite los tres tipos normales, presenta limitaciones en el caso del *around.* AOP Engine.Net no soporta el *around*. Los demás sistemas soportan los tiempos estándar: *before, after y around*.
7. Todos los sistemas presentan mecanismos de definición de puntos de corte con gran potencia y flexibilidad.

B. Criterios de la plataforma
   1. Todos los sistemas evaluados han sido desarrollados sobre implementaciones de las máquinas abstractas de Java o de .NET que, como vimos en el capítulo 6, eran independientes de la plataforma, por lo que en principio heredan esta característica. La única excepción es AOP Engine.Net, que hace uso de características propias de la implementación de la máquina de .NET por parte de Microsoft y que sólo pueden estar disponibles en entornos con un sistema operativo Microsoft Windows, por lo que deja de ser independiente.
   2. Todos los sistemas se han desarrollado sobre plataformas que gozan de gran aceptación comercial (Java y .NET).
   3. Las plataformas sobre las que se han implementado todos los sistemas cuentan con implementaciones que ofrecen un buen rendimiento.
   4. PROSE ha modificado la implementación de la máquina de Java para realizar su sistema, por lo que no es un sistema estándar. El resto de sistemas trabajan sobre plataformas estándar sin necesidad de modificarlas para su funcionamiento.

C. Criterios correspondientes a las aplicaciones
   1. Rapier LOOM.Net, pese a realizar el proceso de tejido en código intermedio, lo que en principio no haría necesario disponer del código fuente de la aplicación a adaptar, impone una serie de condiciones a

las aplicaciones que, en la práctica, implican la necesidad de disponer del código fuente de la aplicación para adaptarla a las condiciones impuestas. El resto de sistemas no necesitan disponer del código fuente de las aplicaciones para poder adaptarlas.

2. Como se ha comentado en el punto anterior Rapier LOOM.Net impone condiciones a las aplicaciones a adaptar. El resto de sistemas no impone ninguna condición a las aplicaciones para que puedan ser adaptadas.

D. Criterios correspondientes a los aspectos

1. Todos los sistemas permiten la adaptación simultánea de aplicaciones por parte de más de un aspecto a excepción de AOP Engine.Net.
2. AOP Engine.Net no ofrece ningún mecanismo para coordinar la ejecución de aspectos en el caso de que adapten a una aplicación en el mismo punto de enlace. Rapier LOOM.Net establece el orden de ejecución dependiendo del orden de creación de los aspectos, lo que no deja la posibilidad al desarrollador de seleccionar el orden de ejecución y en el caso de que los aspectos surjan en distintos momentos puede no ser posible conseguir el orden de ejecución deseado. Los restantes sistemas presentan mecanismos de coordinación de aspectos.
3. Sólo JAsCo y READY AOP soportan los aspectos como aplicaciones, posibilitando así que puedan beneficiarse de la POA. En el resto de sistemas los aspectos no son aplicaciones sino que son clases, o ficheros dll que se utilizan por parte de los tejedores, pero que no pueden ejecutarse de forma independiente.
4. AspectJ y JAsCo han realizado extensiones al lenguaje Java, por lo que no hacen uso de un lenguaje estándar. El resto de sistemas hacen uso de lenguajes estándar, sin modificaciones.
5. AspectJ y JAsCo definen los puntos de corte en el código de los aspectos mediante las extensiones realizadas al lenguaje (en el caso de AspectJ también es posible realizarlo mediante las *annotations* de Java, pero presentan el mismo inconveniente). PROSE y AOP Engine.Net especifican los puntos de corte mediante invocaciones a ciertos métodos definidos por el sistema en el código de los aspectos. Rapier LOOM.Net utiliza los *custom attributes* de la plataforma .NET para definir los puntos de corte en el código de los aspectos. Por lo tanto, todos estos sistemas presentan acoplamiento entre el código de los aspectos y la definición de los puntos de corte. Sólo JBoss AOP y READY AOP, que utilizan un fichero XML externo para definir los puntos de corte (JBoss también soporta el uso de *annotations* pero su utilización es una desventaja), presentan separación entre los puntos de corte y el código de los aspectos.

17.1.5 Conclusiones

17.1.5.1 Evaluación de los criterios generales

Comenzando con los resultados de la evaluación de los criterios generales, observamos en la Figura 59 cómo los sistemas basados en la plataforma Java se ven más desfavorecidos que los sistemas basados en la plataforma .NET. Esto se debe a las características de independencia del lenguaje propias de la plataforma .NET que, en el

caso de la máquina de Java, aunque de forma teórica existe cierta independencia del lenguaje, en la realidad no es tal. Estos sistemas se ven restringidos al uso del lenguaje Java tanto en las aplicaciones a adaptar como en los aspectos que las adapten.

Los sistemas basados en .NET reciben como beneficio la posibilidad de hacer un sistema independiente del lenguaje, siendo éste uno de los principales requisitos del sistema diseñado. READY AOP ha sido diseñado teniendo en cuenta que debe ofrecer un amplio conjunto de puntos de enlace y un completo dinamismo.

17.1.5.2 Evaluación de los criterios correspondientes a la plataforma

En general, los sistemas evaluados cumplen con todos los criterios fijados con respecto a la plataforma sobre la que implementan el sistema, tal y como podemos ver en la Figura 60. Esto es así debido a que las dos plataformas que utilizan los sistemas evaluados (Java y .NET) son independientes de la plataforma sobre la que se ejecutan (entendiendo como plataforma la combinación del *Hardware* y del Sistema Operativo), además, en la actualidad son las dos plataformas más difundidas, encontrándose presentes en la mayoría de instalaciones existentes y ofreciendo un buen rendimiento.

Los dos sistemas que no obtienen el total de puntos son PROSE, por haber realizado modificaciones a la máquina virtual de Java, con lo que ha dejado de ser estándar, y AOP Engine.Net, que hace uso de características presentes en una única implementación de la plataforma .NET y que son dependientes del sistema operativo Windows.

17.1.5.3 Evaluación de los criterios correspondientes a las aplicaciones

En la Figura 61 podemos ver que todos los sistemas evaluados, con la excepción de Rapier, obtienen la máxima puntuación en relación a los criterios establecidos con respecto a los aspectos en el sistema.

Todos los sistemas trabajan con una representación en código intermedio de las aplicaciones, no necesitando, por ello, el acceso al código fuente de la aplicación. De igual manera, ninguno de los sistemas impone condiciones a las aplicaciones que pueden ser adaptadas dentro de los mismos. La excepción es el sistema Rapier, que impone ciertas condiciones a las aplicaciones a adaptar. En concreto, impone condiciones sobre las clases, debiendo sus métodos ser virtuales o haber sido definidos mediante un *interface*, lo que implica que no se pueda utilizar el operador `new` (o su equivalente en el lenguaje en que esté programada la aplicación) debiendo realizar una llamada a una librería suministrada por el sistema. Esto implica que es necesario acceder al código fuente para proceder a realizar las modificaciones en la aplicación que la permitan ser adaptada en el sistema.

17.1.5.4 Evaluación de los criterios correspondientes a los aspectos

En esta evaluación tenemos en cuenta los criterios que se refieren a los aspectos en el sistema, como son las posibilidades de adaptar de forma simultánea una aplicación por parte de varios aspectos y, en caso de soportarlo, el grado de control que se brinda al usuario o las posibilidades de reutilización de los aspectos, ya sea en el propio sistema o en otros sistemas. En la Figura 62 podemos ver la representación grafica de las valoraciones obtenidas por cada sistema.

Se puede observar que todos los sistemas permiten la adaptación simultánea de una aplicación por parte de varios aspectos en el mismo punto de enlace, a excepción de AOP Engine.Net. Únicamente Rapier LOOM.Net no presenta un mecanismo que permita al usuario coordinar la ejecución de múltiples aspectos cuando éstos afectan al mismo punto de enlace. Rapier LOOM.Net fija esta coordinación dependiendo del orden de creación de los aspectos, lo que no es suficiente.

Respecto a la posible reutilización de los aspectos, vemos que los sistemas que han realizado extensiones al lenguaje (JasCo y AspectJ) se ven penalizados, pues su código no puede usarse fuera del sistema, por no ser un lenguaje estándar. El resto de sistemas utilizan lenguajes estándar, por lo que en cualquier entorno donde haya un compilador del lenguaje se podrán utilizar.

Únicamente JBoss y READY AOP separan la definición de los puntos de corte del código de los aspectos, facilitando así la reutilización tanto de los propios aspectos, como de los puntos de corte. El resto de sistemas mezclan en el mismo código del aspecto las definiciones de los puntos de corte, con lo que no es posible la reutilización directa de los aspectos sin tener que modificar el código.

Entre todos los sistemas evaluados, sólo JAsCo y READY AOP ofrecen la posibilidad de que los aspectos sean aplicaciones dentro del sistema, con lo que también podrían beneficiarse de la POA en su implementación, es decir, pueden ser adaptados por otros aspectos.

17.1.5.5 Evaluación absoluta

En la Figura 63 podemos ver la evaluación absoluta de los sistemas, equiponderando todas las puntuaciones. Se puede comprobar que las oscilaciones de puntuación de los sistemas tomados como comparación no son muy acusadas, manteniéndose todos los sistemas en una horquilla pequeña. Esto indica que no hay un sistema claramente superior al resto y que las carencias en determinados aspectos se ven compensadas con las virtudes en otros. La excepción a esto es READY AOP, que obtiene la máxima puntuación en base a los criterios establecidos.

En general, existen muchos más sistemas desarrollados en entornos Java que en otros entornos y, en todo caso, tienen un grado de madurez superior. En la evaluación realizada, todos estos sistemas basados en Java son penalizados por su dependencia del lenguaje, pero al realizar el cómputo conjunto de todos los criterios no son los peor calificados. Esto se debe a su mayor grado de desarrollo y madurez, que les hace ser mejor evaluados en otros criterios, compensando así la puntuación absoluta. Los sistemas evaluados basados en .NET son aproximaciones académicas, por lo que su grado de madurez y base de usuarios es más limitada.

AOP Engine.Net es un sistema independiente del lenguaje que ofrece un alto grado de dinamismo. Pese a ello es el peor valorado de todos los sistemas estudiados. Su principal inconveniente es estar basado en un sistema no estándar que además limita de forma muy importante el conjunto de puntos de enlace que puede soportar el sistema. Además de esto y debido a su implementación, no permite la adaptación simultánea por parte de varios aspectos en el mismo punto de enlace.

Rapier LOOM.Net, pese a ser independiente del lenguaje, tiene una valoración muy similar al anterior y esto se debe principalmente a las imposiciones realizadas a las aplicaciones que pueden ser adaptadas en el sistema, que implican disponer del código fuente de las aplicaciones, a su bajo grado de dinamismo y a su muy reducido conjunto de puntos de enlace.

PROSE es un sistema basado en Java que ofrece un alto grado de dinamismo. Los autores han fijado la eficiencia (no evaluado como criterio) como un objetivo importante en su desarrollo, penalizando requisitos como el conjunto de puntos de enlace que soporta o la utilización de sistemas estándar (que sí han sido evaluados).

AspectJ, que es el sistema más ampliamente difundido de los evaluados junto a Jboss AOP, ha sido diseñado como un sistema basado en Java y que hace extensiones al propio lenguaje. Estas características penalizan su puntuación por la dependencia del lenguaje y la nula reutilización que se puede hacer de los aspectos. El bajo grado de dinamismo ofrecido (el dinamismo de AspectJ se ha añadido en la versión 5, como resultado de la unión del proyecto AspectWerkz) también limita su puntuación.

Jboss AOP es un sistema integrado en el servidor de aplicaciones Jboss que ofrece cierto grado de dinamismo y unas buenas características generales. El hecho de ser dependiente del lenguaje Java y sus limitaciones respecto a la adaptación dinámica es lo que más le ha penalizado en esta evaluación.

JAsCo es un sistema basado en el lenguaje Java, pero al igual que AspectJ ha realizado extensiones al lenguaje, viéndose penalizado igualmente por ello. Cabe mencionar que, junto a READY AOP, es el único sistema de los evaluados que permite adaptar aspectos.

READY AOP ha sido diseñado de forma que satisfaga todos los requisitos establecidos en el capítulo 2 que, básicamente, se corresponden con los criterios evaluados aquí. El sistema se ha diseñado sobre la máquina abstracta de .NET, obteniendo gracias a ello los beneficios de independencia del lenguaje y de la plataforma. La no utilización de ninguna característica externa al estándar garantiza estos beneficios. Mediante el uso de reflectividad se consigue dotar al sistema de una adaptación dinámica completa, así como la no necesidad de imponer condición alguna a las aplicaciones ni acceder a su código fuente. La forma de interactuar entre los distintos elementos del sistema ha sido definida de forma que se garantice una posible reutilización de código.

17.2 Evaluación Cuantitativa

Para realizar una evaluación cuantitativa del sistema propuesto se han realizado una serie de mediciones, tanto de tiempo de ejecución como de consumo de memoria, con el fin de conocer el coste real que tiene la implementación del sistema realizada.

Para todas las pruebas se ha utilizado un PC Pentium IV con una velocidad de 2GHz, y 1GB de memoria RAM, con sistema operativo Windows XP Profesional, Versión 2002 y Service Pack 2. La implementación de .NET utilizada ha sido la versión 1.1.4322.2032 del Framework .NET de Microsoft.

Se han realizado mediciones de lo siguiente:

- Remoting. Se han tomado medidas del coste en tiempo de ejecución que supone el empleo de Remoting, tanto en el registro del canal de comunicación como en la propia comunicación entre dos procesos.
- Coste del registro y desregistro de una aplicación en el Servidor.
- Coste por primitiva (*joinpoint*).
- Escenario real. Diversas mediciones para una aplicación y escenario reales.

A continuación se muestran los resultados obtenidos y su explicación.

Medición de Remoting

Como hemos comentado en 13.4.2 el mecanismo de comunicación entre procesos utilizado en el sistema es Remoting, por lo que es interesante conocer qué coste en el rendimiento implica su utilización.

Para realizar estas mediciones se han creado dos aplicaciones al efecto. Una primera aplicación que actúa de Servidor, el cual publica un objeto por Remoting y ofrece un método al que se puede invocar de forma remota (el Servidor implementa la interfaz `IPropertyFieldAccess`, para simular adecuadamente las invocaciones que tienen lugar en el sistema). Este método no realiza ninguna acción, simplemente devuelve el control al invocador. La segunda aplicación realiza las funciones de Cliente en la comunicación y la primera tarea que lleva a cabo es obtener el objeto publicado por el Servidor y a continuación realiza invocaciones al método ofertado.

Se han realizado pruebas con los protocolos TCP y *named pipes* para observar su distinto comportamiento. Para obtener unos tiempos fiables se han realizado diversas mediciones obteniendo la media aritmética de ellas que es la que se presenta aquí. Para obtener una mayor precisión se han realizado iteraciones de una, diez, cien, diez mil y un millón de llamadas desde el Cliente al Servidor. En la siguiente tabla pueden verse los tiempos medios obtenidos para el protocolo TCP, en milisegundos.

Tabla 1. Tiempos medios con el protocolo TCP

Nº Llamadas (TCP)	1	10	100	10.000	1.000.000
Tiempo de registro (ms)	176	177	153	155	152
Tiempo de llamada (ms)	83	11	4,65	2,03	1,81

En la tabla 2 pueden verse los resultados de la mismas mediciones para el protocolo *named pipes*.

Tabla 2. Tiempos medios con el protocolo *named pipes*

Nº Llamadas (Named Pipes)	1	10	100	10.000	1.000.000
Tiempo de registro (ms)	102	106	103	108	109
Tiempo de llamada (ms)	16	3,1	0,94	0,71	0,65

Como puede observarse el uso de un protocolo u otro tiene una importancia sustancial en el rendimiento, por lo que es un parámetro a tener en cuenta a la hora de implementar el sistema. Gracias a las características de Remoting, la configuración del protocolo a emplear puede realizarse por medio de un fichero externo a la aplicación (ver 13.4.3 para más información), por lo que no es necesario fijar un protocolo para el sistema, sino que se puede adaptar a las necesidades del entorno en el que se ejecute.

Puesto que el protocolo *named pipes* ha demostrado ser más eficiente que el TCP, será éste el que utilizaremos para el resto de mediciones de aplicaciones.

También se puede observar que el tiempo empleado en cada llamada desciende cuanto mayor es el número de ellas realizadas. Esto se debe a las optimizaciones realizadas por el compilador *JIT* en ejecución. Esta optimización no se puede llevar a cabo de una manera tan completa cuando las invocaciones no se realizan al mismo objeto o al mismo método o cuando los datos que se envían como parámetros varían, por lo que el tiempo realmente consumido por cada invocación no puede calcularse con exactitud a priori.

Coste del registro y desregistro de la aplicación

Para realizar esta medición hemos creado una aplicación que no realiza más acción que la creación de un objeto diez millones de veces (con el fin de que el tiempo de ejecución fuese mayor que un milisegundo, que es la precisión máxima que podemos obtener en las mediciones). Se ha medido su tiempo de ejecución normal, y se ha medido su ejecución con el código añadido para proceder a realizar un registro y un desregistro en el Servidor.

El coste medio en tiempo de ejecución medido del registro y desregistro en el Servidor es de 121 milisegundos. El código añadido consiste en la obtención de un objeto del servidor y en la realización de dos invocaciones, una para registrarse y otra para desregistrarse, por lo que el tiempo consumido es el que se podía esperar.

Coste por tipo de punto de enlace

Cada punto de enlace inyectado en el código de la aplicación tiene un coste en el rendimiento, tanto si hay aspectos que hayan solicitado ser advertidos de su ejecución como si no los hay.

Se han realizado mediciones del tiempo de ejecución consumido por el código inyectado en el caso de que no haya aspectos registrados. Esto lo hacemos así para diferenciar entre el tiempo consumido por la verificación que realiza el código con el fin de comprobar si hay algún aspecto registrado en el punto de enlace al que deba invocarse, del consumido por la invocación al aspecto en el caso de que deba realizarse. A continuación se presentan los tiempos medios medidos en milisegundos.

Tabla 3. Tiempos medios por punto de enlace (ms)

Tipo	MC	ME	FS	FR
Tiempo	0,0031	0,0046	0,0047	0,0056

Las siglas MC se corresponden con el punto de enlace invocación a un método o constructor, ME se corresponde con la ejecución de un método o constructor. FS y FR se corresponden con el acceso de escritura y lectura, respectivamente, a un campo. El acceso a propiedades ha sido implementado como invocaciones a métodos, por lo que son los tiempos de la invocación a métodos los que deben aplicarse en ese caso.

Escenario real

Para realizar mediciones sobre una aplicación real se ha seleccionado una que viene incluida como ejemplo en el SDK de la implementación de .NET por parte de

Microsoft: WordCount (la cual se encuentra en el subdirectorio Samples\Applications\ WordCount dentro del SDK). Está disponible con el código en versión C# y Visual Basic.NET.

Para realizar las pruebas hemos modificado la aplicación, en su versión C#, añadiéndole la funcionalidad necesaria para que tome datos del tiempo de ejecución. La aplicación completa, ya modificada, consta de 649 líneas de código en C# que se corresponden con 2.123 líneas de código en IL.

Esta aplicación realiza un análisis de un fichero de texto cuyo nombre recibe como parámetro y devuelve como resultado el número de líneas, palabras, caracteres y Bytes que contiene el fichero. Las pruebas se han realizado sobre un único fichero de texto con 23.266 líneas, 404.834 palabras y 2.605.474 Bytes. Inicialmente se han realizado mediciones de los siguientes casos:

- La aplicación original sin ninguna modificación.
- La aplicación únicamente ampliada con el código de registro y desregistro.
- La aplicación ampliada con todos los puntos de enlace posibles, pero sin que ningún aspecto se hubiese registrado en ellos.
- La aplicación ampliada con todos los puntos de enlace posibles y con aspectos registrados en todos los puntos de enlace inyectados.

Para cada caso se ha medido el tiempo de ejecución, su consumo de memoria, el número de líneas de código IL, el número de ellas que son comentarios en el caso de código inyectado y, en el caso que proceda, el número de comprobaciones e invocaciones realizadas.

Los datos correspondientes a comprobaciones se refieren a las veces que se ejecuta el código de comprobación de la existencia de aspectos registrados inyectado en los puntos de enlace. Las invocaciones se refieren al número de veces que la comprobación anterior resulta cierta y se invoca al aspecto correspondiente.

Los resultados obtenidos se pueden observar en la siguiente tabla (se muestran entre parénTesis los incrementos porcentuales respecto al original):

Tabla 4. Resultados obtenidos con máxima adaptación posible

	Original	Sólo registro	Todo, sin aspectos	Todo, con aspectos
Tiempo	3.140 ms	3.218 ms (2,4%)	10.827 ms (244%)	38 minutos (72.511%)
Memoria (Bytes)	6.944.416	11.091.068 (59,7%)	13.135.872 (89%)	13.480.486 (94%)
Líneas de código	2.123	2.178	16.528	16.528
Comentarios	462	484	951	951
Comprobaciones	N/A	N/A	1.665.994	1.665.994
Invocaciones	N/A	N/A	0	1.665.994

En la fila correspondiente al consumo de memoria se puede observar que hay un gran incremento entre la aplicación original y las aplicaciones inyectadas, incluso en el caso de inyección únicamente del registro que implica un incremento de 55 líneas de código. Esto se debe a que además del código propio del registro se han añadido enlaces a las librerías de Remoting y a la librería del sistema `Interfaces`.

La tercera columna de la tabla (aplicación inyectada por completo, sin aspectos registrados) representa el coste que supone inyectar código en todos los posibles puntos de enlace, cuando no existen aspectos registrados para ellos. Entre los puntos de enlace inyectados en este caso se encuentran los relacionados con los objetos propios de la aplicación y con objetos ajenos a ella, como los pertenecientes a la *BCL* (por ejemplo, cualquier acceso a la consola o a un método de la clase `String`).

La medición realizada sobre una aplicación con aspectos registrados en todos los puntos de enlace posibles no es un escenario realista de ejecución de la aplicación. Uno típico podría ser el de un aspecto que muestre al usuario el número de palabras que va encontrando a la vez que procesa el fichero. Hemos realizado sobre la aplicación distintas mediciones basadas en supuestos más reales. En concreto se han realizado las siguientes mediciones:

- La aplicación original modificada de forma manual en C# para que muestre el número de palabras encontradas de forma simultánea a la ejecución.
- La aplicación original inyectada únicamente con los puntos de enlace que se corresponden con elementos de la propia aplicación, es decir, no se contemplan invocaciones a elementos ajenos a la aplicación como pueden ser los contenidos en la *BCL*.
- La aplicación original inyectada con los mismos puntos de enlace del caso anterior y un aspecto registrado en los puntos de enlace que se corresponden con la asignación de valores al campo que contiene el número de palabras encontradas. El aspecto no realiza acción alguna en este caso.
- Las mismas condiciones que en el caso anterior, pero el aspecto muestra por pantalla el número de palabras encontradas de forma simultánea al procesado del fichero.

En la siguiente tabla pueden verse los resultados obtenidos:

Tabla 5. Resultados obtenidos con adaptación en escenario realista

	Original	Original +mostrar	Inyectada modificada	Aspecto sin mostrar	Aspecto mostrando
Tiempo	3.140 ms	1:27 (2.670%)	5.562ms (77%)	8:01 (15.218%)	9:20 (17.734%)
Memoria (Bytes)	6.944.416	7.092.482 (2%)	11.890.072 (71,2%)	11.902.272 (71,3%)	11.902.272 (71,3%)
Líneas de código	2.123	2.123	16.546	16.528	16.528
Comentarios	462	462	951	951	951
Comprobaciones	N/A	N/A	404.836	404.836	404.836
Invocaciones	N/A	N/A	N/A	404.836	404.836

Como se puede observar, el mostrar por pantalla de forma concurrente al procesado del fichero la información del número de palabras encontradas hasta ese momento tiene un coste muy alto en tiempo de ejecución. La aplicación original pasa de consumir poco más de tres segundos (3.140 milisegundos) a consumir un minuto y veintisiete segundos. Un incremento de la misma magnitud se puede ver entre el aspecto que muestra por pantalla esa información y el que no lo muestra (columnas quinta y cuarta respectivamente).

Conclusiones

A la hora de evaluar el rendimiento de una aplicación en el sistema es importante poder diferenciar entre los tiempos de los distintos elementos que se ejecutan. En este caso existe una serie de elementos que tienen influencia en el tiempo de ejecución:

- La aplicación original.
- El código de registro y desregistro.
- El código de verificación de aspectos registrados.
- La invocación a los aspectos registrados.
- La ejecución de los aspectos.

En la cuarta columna de la tabla 5 se muestra la ejecución de la aplicación en un supuesto realista. El tiempo total empleado, ocho minutos un segundo, se puede desglosar en diferentes partes que son la ejecución del código de la aplicación original –3.140 ms–, el tiempo de registro y desregistro, que en las mediciones obtenidas para este ejemplo ha resultado de 121ms de media, el código de la verificación de aspectos registrados, que se puede obtener de la tercera columna de la misma tabla restando del tiempo total –5.562ms– los tiempos de la aplicación original y del registro resultando 2.301ms y, por último, la invocación y ejecución de los aspectos.

El tiempo consumido por el registro y desregistro de la aplicación en el Servidor es independiente de la naturaleza de la aplicación a adaptar y de la adaptación realizada, siendo de 121 milisegundos de media.

La verificación de aspectos registrados en la aplicación es dependiente de la adaptación realizada y de la aplicación. Hemos realizado mediciones de un escenario realista en el que el incremento del tiempo de ejecución achacable a la verificación de aspectos se sitúa en el 73,2% del tiempo de ejecución de la aplicación original (1.301ms con respecto a los 3.140ms originales). También hemos realizado mediciones en escenarios no realistas, por exceso, en los que el incremento debido a la verificación de aspectos se sitúa en el 239% (basándose en los datos de la tercera columna de la tabla 4).

Del tiempo total (8:01) de ejecución de la aplicación en un posible escenario real se desprende que el 0,6% del tiempo se corresponde con la ejecución de la aplicación original (3.140ms) y el 0,02% con el registro y desregistro (que es estable e independiente de la aplicación y consume 121ms). También se puede asumir que el 0,47% del tiempo empleado se corresponde con la ejecución del código de verificación de los puntos de enlace (2.301ms). Por último, el restante 98,8% se corresponde con la ejecución de la invocación remota y la ejecución del aspecto (que en este caso no hacía nada más que devolver el control tras recibir la información).

Teniendo en cuenta el tiempo empleado para realizar las invocaciones y las ejecuciones de los aspectos (475.438ms, obtenidos de restar del tiempo total –8:01 o lo que es lo mismo 481.000ms– los tiempos correspondientes al resto de apartados), y el número de invocaciones y ejecuciones de aspecto realizadas –404.836–, se obtiene que el tiempo medio consumido en realizar una invocación, ejecutar el aspecto y devolver el control es de 1,17ms.

Se puede observar que el punto que mayor incidencia tiene en el tiempo de ejecución de la aplicación es la comunicación de procesos. Esto entra dentro de lo esperado, puesto que el mecanismo de comunicación empleado requiere la serialización y deserialización de los parámetros que se envían en las invocaciones a métodos, lo que es costoso en tiempo. Como se vio anteriormente, la elección de un protocolo u otro para la comunicación entre procesos tiene una importancia muy alta en su rendimiento.

Una mejora en el rendimiento en el mecanismo de comunicación de procesos implicaría una mejora directa en el rendimiento de todo el sistema.

Por todo esto podemos concluir que el rendimiento está muy ligado al conjunto de puntos de enlace inyectados en la aplicación y, sobre todo, al conjunto de puntos de corte que se hayan activado, existiendo un compromiso entre dinamicidad y rendimiento.

CAPÍTULO 18
CONCLUSIONES Y TRABAJO FUTURO

A lo largo de esta Tesis hemos analizado las distintas alternativas para crear un sistema que ofrezca, como principal objetivo, un *soporte dinámico a la separación de aspectos de forma independiente del lenguaje y de la plataforma*. El desglose completo de los objetivos del sistema se encuentra en el capítulo 2.

Comenzamos estudiando en el capítulo 3 el problema de la Separación de Incumbencias e identificamos en el capítulo 4 el Desarrollo de Software Orientado a Aspectos (DSOA) como una aproximación adecuada para resolver dicho problema. Tras estudiar exhaustivamente un conjunto representativo de sistemas reales que ofrecen DSOA (capítulo 5), tanto del ámbito empresarial como del académico, concluimos que ninguno de los sistemas estudiados satisface los requisitos planteados.

Algunos de los sistemas que ofrecen DSOA más empleados son estáticos lo que no satisface los objetivos marcados. Entre los sistemas dinámicos de uso empresarial, la mayoría presentan un dinamismo limitado y en general son dependientes del lenguaje y/o de la plataforma. Aunque existen algunos sistemas, mayormente de ámbito académico, que presentan un grado de dinamismo aceptable, suelen ser dependientes del lenguaje, y aquellos que no lo son presentan otras limitaciones que los invalidan para cumplir los objetivos marcados.

En el capítulo 6 estudiamos uno de los mecanismos más utilizados para conseguir sistemas que ofrezcan DSOA independiente de la plataforma, el empleo de máquinas abstractas como base sobre la que construir el sistema, y vimos las aportaciones que se obtienen de su empleo, así como las características que debe reunir una máquina abstracta para que sea posible implementar sobre ella un sistema que cumpla con todos los objetivos fijados.

En los siguientes capítulos analizamos una de las técnicas más empleadas para conseguir sistemas adaptables, y en concreto, sistemas que ofrezcan DSOA dinámico: la reflexión. Estudiamos los distintos grados de reflexión, fijando categorías, y establecimos los beneficios que de su utilización obtendría el sistema a desarrollar así como el grado de reflexión necesario para conseguir dichos objetivos. Realizamos un análisis de distintos sistemas existentes con el fin de evaluar sus características reflectivas y comprobar su posible adecuación a los requisitos generales del sistema propuesto.

En el capítulo 9 presentamos la arquitectura del sistema propuesto, que desarrollamos de forma detallada en los siguientes capítulos. En el capítulo 15 analizamos el diseño de un prototipo del sistema propuesto que se ha realizado para verificar la viabilidad del mismo. En el capítulo 16 mostramos posibles aplicaciones del sistema propuesto y en el capítulo 17 procedimos a certificar la validez de la propuesta realizada para satisfacer los requisitos generales del sistema.

En este capítulo, estudiaremos cómo el diseño de nuestro sistema ha superado todas las limitaciones puntualizadas, satisfaciendo los requisitos preestablecidos a lo largo de esta Tesis –capítulo 2–. Analizaremos las principales ventajas aportadas, y estableceremos las futuras líneas de investigación a seguir.

18.1 Sistema Diseñado

El sistema diseñado (capítulo 9) utiliza como base una máquina abstracta y sobre ella se han creado dos aplicaciones y se ha definido un lenguaje de especificación de puntos de corte y un Framework de Aspectos que deben implementar las aplicaciones y los aspectos que vayan a ejecutarse en el sistema.

Como máquina abstracta sobre la que implementar el sistema se ha seleccionado la plataforma .NET debido, principalmente, a sus características de independencia del lenguaje y de la plataforma, así como a la posibilidad de interacción directa entre aplicaciones. Gracias a estas características, el sistema obtenido es independiente del lenguaje y de la plataforma, permitiendo que las aplicaciones y los aspectos puedan estar codificados en cualquier lenguaje e interactúen entre ellos de forma directa, sin la necesidad de una capa intermedia.

Las aplicaciones creadas son el *JoinPoint Injector* (JPI), que se encarga de instrumentar el código de las aplicaciones que van a ejecutarse en el sistema con el fin de que puedan ser adaptadas en su comportamiento, y el Servidor de Aplicaciones, que hace de intermediario en la adaptación de las aplicaciones por parte de los aspectos.

Haciendo uso de las capacidades reflectivas de la plataforma .NET es posible instrumentar el código de las aplicaciones a ejecutar en el sistema, añadiéndolas un MOP que ofrezca reflexión computacional de forma dinámica, posibilitando el que puedan ser adaptadas en ejecución. El componente que se encarga de esta instrumentación es el JPI. La instrumentación se realiza como paso previo a la entrada de una aplicación en el sistema y tiene lugar sobre el ejecutable de la aplicación, obteniendo la representación del programa en el código intermedio nativo de la máquina abstracta, esta representación es la que se modifica y aumenta para adaptar la aplicación, generando un nuevo ejecutable que es el que realmente se ejecuta en el sistema. Al realizar el proceso sobre el código intermedio obtenido a partir del ejecutable, se obtiene la independencia del lenguaje para el sistema, y no es necesario disponer del código fuente de una aplicación para poder adaptarla.

La otra aplicación del sistema, el Servidor de Aplicaciones, se encarga de permitir la adaptación de las aplicaciones por parte de los aspectos. A diferencia de la mayoría de los sistemas existentes en los que los aspectos no son aplicaciones normales, sino que son un conjunto de métodos que se añaden al código de la aplicación para dar lugar a la aplicación adaptada, en nuestro sistema los aspectos son aplicaciones ordinarias

que, como tales, pueden ejecutarse de forma individual e independiente de las aplicaciones.

Son los propios aspectos los que solicitan al Servidor el adaptar el comportamiento de una aplicación y le suministran el conjunto de puntos de corte en los que están interesados. El Servidor procesa los puntos de corte identificando los puntos de enlace de la aplicación seleccionados por ellos y activando estos últimos con el fin de que el aspecto sea notificado cuando la ejecución de la aplicación los alcance. Cuando la ejecución de la aplicación alcanza un punto de enlace para el que se han registrado aspectos la aplicación notifica a todos los que se hayan registrado de este hecho. El orden de notificación viene determinado por la precedencia que se haya asignado a cada aspecto, siendo esta precedencia configurable por el usuario.

Se ha definido un lenguaje basado en XML para la especificación de los puntos de corte en el sistema. Los puntos de corte se especifican en un fichero independiente del aspecto, en el lenguaje definido. Esto implica un desacoplamiento entre el código del aspecto y los puntos de corte, facilitando la reutilización de código. El hecho de utilizar XML permite mantener la independencia del lenguaje de los puntos de corte con respecto al de implementación de los aspectos.

Con el fin de posibilitar la comunicación entre los aspectos, aplicaciones y Servidor se ha definido un Framework en el sistema que especifica un conjunto de interfaces que deben implementar los distintos elementos para formar parte del sistema.

El Servidor implementa una interfaz que posibilita a las aplicaciones registrarse y desregistrarse del sistema, y a los aspectos solicitar adaptaciones de aplicaciones.

Las aplicaciones deben implementar dos de estas interfaces. Por un lado una interfaz que posibilita al Servidor la activación y desactivación de los puntos de enlace de la aplicación, y por otro una interfaz que ofrece reflexión intraaplicaciones, permitiendo que un aspecto obtenga información reflectiva de la aplicación a la que está adaptando y pueda invocar a sus miembros (ejecución de métodos, acceso a campos, etc.). Ambas implementaciones están contenidas en una librería del sistema que se añade a las aplicaciones en el momento de realizar su instrumentación, de esta manera no es necesario que el desarrollador se responsabilice de la inclusión de las mismas en la aplicación y no se impone ninguna restricción a las aplicaciones a adaptar.

Los aspectos deben implementar otra interfaz que permite a las aplicaciones notificar el alcance, durante la ejecución de la aplicación, de un punto de enlace en el que se haya registrado el aspecto. Es el punto de entrada a partir del cual pueden proceder a modificar el comportamiento de la aplicación. Esta implementación es responsabilidad del desarrollador del aspecto.

18.2 Principales Ventajas Aportadas

Detalladamente, el conjunto de ventajas aportadas por nuestro sistema es la consecución de los requisitos establecidos inicialmente en el capítulo 2. Agrupando éstos y destacando únicamente los más representativos, podemos concluir las siguientes aportaciones.

18.2.1 Sistema Realmente Dinámico

El sistema diseñado permite a los usuarios la activación y desactivación en tiempo de ejecución, sin ninguna restricción, de aspectos que adapten a una aplicación, ya sea que hubiesen sido previstos en la fase de diseño o hayan surgido posteriormente con motivo de un nuevo requerimiento o ante cambios en el entorno de ejecución.

La mayoría de los sistemas que ofrecen soporte al DSOA de forma dinámica lo hacen con restricciones como puede ser la necesidad de tener conocimiento del aspecto en la fase de desarrollo, aunque su activación pueda producirse en ejecución. Esto limita mucho las posibilidades reales de esos sistemas.

El sistema aquí presentado ofrece la posibilidad de adaptar a una aplicación con aspectos no contemplados durante la fase de desarrollo, de forma que el sistema podrá adaptarse a cambios en el entorno de ejecución o a requerimientos que hayan aparecido con posterioridad a la puesta en ejecución de la aplicación, sin tener que detener la misma.

18.2.2 Independencia del Lenguaje

Independientemente del lenguaje en el que estén escritas las aplicaciones que actúen dentro del sistema, al ser compiladas son traducidas a una representación en código intermedio nativo de la máquina abstracta sobre la que se ha desarrollado el sistema. Como todo el proceso de modificación de las aplicaciones se realiza sobre la representación de las mismas en ese código nativo, se obtiene un sistema independiente del lenguaje, que puede trabajar con cualquier aplicación escrita en cualquier lenguaje.

La mayoría de los sistemas existentes que ofrecen soporte al DSOA (al menos los más difundidos y con mayor base de usuarios) se han implementado sobre la plataforma Java, siendo dependientes del lenguaje (en muchos casos utilizan Java estándar, pero en otros utilizan Java con extensiones añadidas por el sistema).

La independencia del lenguaje que presenta el sistema propuesto, junto a la no imposición de ninguna restricción a las aplicaciones a adaptar, añade una nueva ventaja al sistema propuesto como es la no necesidad de disponer del código fuente de una aplicación para poder adaptarla. Con la representación en código intermedio (el ejecutable) es suficiente.

Respecto a los aspectos tampoco existe ninguna limitación referente al lenguaje en el que deben implementarse ya que se benefician del sistema común de tipos que ofrece la máquina abstracta sobre la que se implementa el sistema, permitiendo la interacción directa, al compartir los tipos, entre procesos escritos en distintos lenguajes.

18.2.3 Independencia de la Plataforma

El sistema se ha diseñado sobre una máquina abstracta independiente de la plataforma, de tal forma que pueden existir máquinas virtuales que la implementen en cualquier entorno (*Hardware* y sistema operativo). Puesto que el sistema hace uso únicamente de características propias de la definición estándar de la máquina abstracta, evitando hacer uso de características particulares de cualquier implementación o externas a

la propia máquina abstracta, podrá utilizarse sobre cualquier máquina virtual que implemente el estándar, obteniendo de esta forma la misma independencia de la plataforma.

18.2.4 Amplio Conjunto de Puntos de Enlace

El sistema dinámico diseñado, a diferencia de la mayoría de sistemas dinámicos existentes, ofrece un conjunto de puntos de enlace muy rico, similar a los ofrecidos por los sistemas estáticos. El sistema cuenta con todos los tipos de puntos de enlace existentes en los sistemas estudiados que pueden aplicarse al lenguaje intermedio de la máquina abstracta sobre la que se implementa (en concreto soporta todos los puntos de enlace que soporta AspectJ e, incluso, alguno de ellos de una forma más amplia, sin restricciones). Obviamente, y por cuestiones de independencia del lenguaje, se han eliminado aquellos puntos de enlace que existen únicamente en algún lenguaje de programación concreto (como pueda ser Java).

De esta manera, se ofrece al desarrollador un mayor abanico de alternativas, facilitándole su tarea y posibilitando un mejor uso del DSOA.

18.2.5 Alta Reutilización del Código de los Aspectos

Debido a dos características impuestas por los requisitos del sistema como son el no realizar ninguna extensión a ningún lenguaje para definir los aspectos y la separación física existente entre el código de los aspectos y los puntos de corte que los relacionan con las aplicaciones, se obtiene como beneficio una alta posibilidad de reutilización del código de los aspectos.

En primer lugar un aspecto genérico puede ser reutilizado dentro del sistema con otra aplicación sin necesidad de realizar ningún cambio en su código, simplemente sería necesario definir los nuevos puntos de corte que relacionen este aspecto con la aplicación que se desea adaptar. Un ejemplo de este tipo de aspectos puede ser una traza.

De igual modo, un aspecto implementado en el sistema, al estar codificado en un lenguaje estándar sin ninguna extensión, podrá ser utilizado en otro sistema (que admita el lenguaje empleado) de una forma sencilla.

18.2.6 Sistema Distribuido

Aunque éste no era un requisito planteado en el capítulo 2, el diseño del sistema se ha realizado pensando en la creación de un sistema distribuido, en el que los distintos actores del mismo puedan estar de forma indistinta en el mismo equipo o en equipos distintos distribuidos a través de la red.

De esta manera, se ofrecen alternativas como la existencia de un único Servidor del sistema en una red, o la existencia de varios de ellos e incluso uno en cada equipo que ejecute aplicaciones dentro del sistema. Gracias a esto, se podrían emplear Servidores redundantes o realizar balanceos de carga a través de la red.

Igualmente es posible distribuir entre distintos equipos la ejecución de la incumbencia principal de la aplicación (representada por la propia aplicación) y la de las incumbencias ortogonales a ella (representadas por los aspectos).

18.2.7 Uso Simultáneo con Otros Sistemas

Otra ventaja añadida del sistema realizado, pese a no ser un requisito planteado en el capítulo 2, es la posibilidad de compatibilizar de forma simultánea su uso con el de otros sistemas que ofrezcan soporte al DSOA.

Debido a ciertas características impuestas al sistema como son el no realizar ninguna extensión al lenguaje en el que se definen los aspectos, es decir, utilizar un lenguaje estándar, el hecho de no imponer ninguna restricción a las aplicaciones a adaptar y el trabajar directamente sobre el código intermedio para realizar la adaptación es posible adaptar una aplicación mediante el uso del sistema aquí propuesto de forma simultánea a una adaptación por parte de otro sistema.

De este modo es posible diferenciar entre adaptaciones conocidas durante el diseño, que podrán ser implementadas mediante un sistema que ofrezca DSOA de forma estática, y por lo tanto más eficiente, y las adaptaciones desconocidas en tiempo de diseño que requerirán una adaptación dinámica. Gracias a esto, la ejecución de la aplicación resultante será más rápida que si utilizásemos únicamente la adaptación dinámica para todos los casos ya fuese necesario o no el dinamismo.

18.3 Futuras Líneas de Investigación y Trabajo

El trabajo de investigación realizado en esta Tesis abre nuevas líneas de investigación futuras, además de existir puntos en los que ampliar y mejorar las implementaciones asociadas al sistema presentado. Podemos identificar las siguientes líneas de trabajo inminente:

18.3.1 Ampliación de la Expresividad del Lenguaje de Definición de Puntos de Corte

En el sistema hemos definido la forma de seleccionar puntos de enlace basándonos únicamente en su tipo, pero AspectJ y otros sistemas soportan además otras formas de seleccionar los puntos de enlace como son por el flujo de ejecución (`cflow`) o mediante puntos de corte de tipo léxico (`within`). La implementación de estas posibilidades ampliaría la potencia del sistema.

Para soportar la selección condicionada al flujo de ejecución, además de modificar la definición del lenguaje de puntos de corte, es necesario conocer en ejecución el flujo seguido por la aplicación, para ello es necesario inyectar código en la aplicación que se encargue de almacenar el recorrido que sigue la aplicación. Cuando se alcanza un punto de enlace se comprueba el recorrido que ha seguido la aplicación hasta llegar a ese punto, de tal forma que se pueda decidir en ese momento si el punto de enlace es seleccionado o no por el punto de corte. Esto, obviamente, presenta un coste de rendimiento, que habría que valorar.

La selección del tipo `within` o `withincode` es sencilla de implementar con la estructura actual, además no requiere comprobaciones más que en el momento en el que se registra un aspecto, por lo que no tendría un impacto negativo en el rendimiento de las aplicaciones.

18.3.2 Ampliación y Modificación de la Implementación

Siguiendo con el sistema presentado en esta Tesis, es interesante emplear trabajo de implementación para obtener un sistema que, manteniendo todas las ventajas obtenidas, ofrezca una mejor eficiencia y con el desarrollo de una interfaz gráfica de usuario que facilite su uso.

18.3.2.1 Ampliación de la implementación del JPI

En la actualidad, la implementación del JPI realizada como prototipo para demostrar la viabilidad de la propuesta no contempla todos los puntos de enlace definidos, habiendo suprimido el tratamiento del punto de enlace "tratamiento de una excepción".

En 12.3 estudiamos la detección de las sombras de los puntos de enlace, incluidas las del punto de enlace "tratamiento de una excepción" y en 14.4 definimos la interfaz adecuada que deben implementar los aspectos para adaptar este punto de enlace, en el capítulo 11 estudiamos las sentencias necesarias para expresar los puntos de corte referentes a este punto de enlace y el procesador de ficheros de puntos de corte implementado lo contempla, por lo que lo único que queda pendiente de implementarse es la propia inyección de código.

18.3.2.2 Inyección de código optimizado

En la implementación del JPI no se han tenido en cuenta aspectos de rendimiento y, cuando se ha tenido que decidir entre velocidad de ejecución y ocupación de memoria se ha optado siempre por una mayor velocidad de ejecución, inyectando código repetido, que podría haber sido unificado mediante métodos disminuyendo las necesidades de memoria pero aumentando el tiempo de ejecución.

En los casos de inyección en un punto de enlace del código para tratar un único tiempo el código está optimizado, pero en el caso de coincidencias de varios tiempos en el mismo punto de enlace no lo está, siendo posible obtener un mejor rendimiento en ejecución mediante un código más optimizado.

En general, se ha optado por la eficiencia en el rendimiento, aunque esto supusiese un aumento en las necesidades de memoria de la aplicación. Esto puede no ser adecuado en ciertas aplicaciones para las que las necesidades de memoria sean un factor crítico, por lo que sería deseable que el JPI ofreciese la posibilidad de optimizar en función del rendimiento o en función del uso de memoria, según lo solicitase el usuario.

18.3.2.3 Ampliación del Servidor

La implementación actual del Servidor se limita a cumplir la implementación de la interfaz `IServer`, que es necesaria para que las aplicaciones se puedan registrar en el sistema y los aspectos puedan adaptarlas. Tal y como explicamos en 13.3, el Servidor

debe incorporar un mecanismo de seguridad que permita controlar qué aspectos tienen derechos para adaptar el comportamiento de qué aplicaciones. Por ello, la elección de un mecanismo de seguridad apropiado y su implementación en el Servidor es uno de los puntos donde se puede enfocar trabajo de implementación.

Otra posibilidad de trabajo es la creación de un entorno gráfico desde el que se pueda controlar la actividad del Servidor, ofreciendo utilidades para observar las aplicaciones disponibles en el sistema, las adaptaciones que están en vigor de aspectos sobre aplicaciones, gestión de permisos, visualización de estadísticas, etc.

18.3.2.4 Herramienta para la creación de ficheros de puntos de corte

En la implementación actual del sistema no se cuenta con ninguna ayuda para la creación de los ficheros de puntos de corte. El usuario debe conocer la especificación del lenguaje y crear el fichero adecuado a sus intereses. Además, no cuenta con una herramienta específica que le permita conocer los puntos de enlace existentes en la aplicación que desea adaptar.

Es interesante la creación de una utilidad que muestre de una forma gráfica y fácilmente comprensible los puntos de enlace existentes en una aplicación, y que permita al usuario seleccionarlos igualmente de forma gráfica, encargándose la aplicación de generar el fichero con los puntos de corte correspondientes en el lenguaje definido. De esta forma no sería necesario conocer por parte del desarrollador de aspectos la especificación del lenguaje de definición de puntos de corte, simplificando la utilización del sistema y minimizando posibles errores.

18.3.3 Creación de una Biblioteca de Aspectos

Existen una serie de incumbencias típicas que son utilizadas de forma recurrente en la mayoría de las aplicaciones, ejemplos de éstas pueden ser la persistencia, gestión de trazas, gestión de transacciones, manejo de excepciones, etc. De hecho, algunos sistemas ofrecen aspectos ya implementados que tratan estas incumbencias (por ejemplo [JBossAOP] en su biblioteca de aspectos) y existen proyectos para crear estándares de tal forma que los aspectos puedan ser reutilizados entre distintos sistemas [AOPAlliance].

Sería interesante contar con una biblioteca de aspectos de estas características, de tal forma que puedan ser usados de forma directa o con mínimas modificaciones (para adaptarlo a un comportamiento particular deseado) en el sistema. El lenguaje en el que estuviesen implementados dichos aspectos sería indiferente, pues podrían adaptar a cualquier aplicación indistintamente del lenguaje en el que estuviese implementada. Podrían implementarse en varios lenguajes y así el desarrollador seleccionaría aquél que esté implementado en el lenguaje en que se sienta más cómodo.

APÉNDICE A
LENGUAJE XML DE DEFINICIÓN DE PUNTOS DE CORTE

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="urn:gramaticapointcuts-schema"
targetNamespace="urn:gramaticapointcuts-schema"
elementFormDefault="qualified">
  <xs:element name="pointcut_definitions">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="pointcut_definition"
type="primitive_pointcut" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="empty">
    <xs:complexContent>
      <xs:restriction base="xs:anyType"/>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="identifier_pattern">
    <xs:choice>
      <xs:element name="identifier_name" type="identifier_name"/>
      <xs:element name="not">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="identifier_pattern"
type="identifier_pattern"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="or">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="identifier_pattern"
type="identifier_pattern" minOccurs="2" maxOccurs="2"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="and">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="identifier_pattern"
type="identifier_pattern" minOccurs="2" maxOccurs="2"/>
          </xs:sequence>
```

```
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="type_pattern">
    <xs:choice>
      <xs:element name="type_name" type="type_name"/>
      <xs:element name="not">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="type_pattern" type="type_pattern"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="or">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="type_pattern" type="type_pattern"
minOccurs="2" maxOccurs="2"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="and">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="type_pattern" type="type_pattern"
minOccurs="2" maxOccurs="2"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
  <xs:simpleType name="identifier_name">
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z$_\*][a-zA-Z$_0-9\*]*"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="type_name">
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-zA-Z$_\*][a-zA-Z$_0-9\*]*)([\.|\.\.][a-
zA-Z$_\*][a-zA-Z$_0-9\*]*)*([\+])?([\[][\]])*"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="qualified_name">
    <xs:sequence>
      <xs:element name="qualified_class" type="qualified_type_name"/>
      <xs:element name="name" type="identifier_pattern"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="qualified_type_name">
    <xs:sequence>
      <xs:element name="namespace" type="type_pattern"/>
      <xs:element name="class" type="identifier_pattern"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="compound_name">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="simple_name" type="identifier_pattern"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="time_modifier">
```

```
    <xs:restriction base="xs:string">
      <xs:enumeration value="before"/>
      <xs:enumeration value="after"/>
      <xs:enumeration value="around"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="method_flag">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="privatescope" type="empty"/>
      <xs:element name="private" type="empty"/>
      <xs:element name="famandassem" type="empty"/>
      <xs:element name="assembly" type="empty"/>
      <xs:element name="family" type="empty"/>
      <xs:element name="famorassem" type="empty"/>
      <xs:element name="public" type="empty"/>
      <xs:element name="static" type="empty"/>
      <xs:element name="final" type="empty"/>
      <xs:element name="virtual" type="empty"/>
      <xs:element name="hidebysig" type="empty"/>
      <xs:element name="newslot" type="empty"/>
      <xs:element name="abstract" type="empty"/>
      <xs:element name="specialname" type="empty"/>
      <xs:element name="pinvokeimpl" type="empty"/>
      <xs:element name="unmanagedexp" type="empty"/>
      <xs:element name="rtspecialname" type="empty"/>
      <xs:element name="reqsecobj" type="empty"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="field_flag">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="privatescope" type="empty"/>
      <xs:element name="private" type="empty"/>
      <xs:element name="famandassem" type="empty"/>
      <xs:element name="assembly" type="empty"/>
      <xs:element name="family" type="empty"/>
      <xs:element name="famorassem" type="empty"/>
      <xs:element name="public" type="empty"/>
      <xs:element name="static" type="empty"/>
      <xs:element name="initonly" type="empty"/>
      <xs:element name="literal" type="empty"/>
      <xs:element name="notserialized" type="empty"/>
      <xs:element name="specialname" type="empty"/>
      <xs:element name="pinvokeimpl" type="empty"/>
      <xs:element name="marshal" type="empty"/>
      <xs:element name="rtspecialname" type="empty"/>
      <xs:element name="reqsecobj" type="empty"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="constructor_signature_def">
    <xs:sequence>
      <xs:element name="constructor_signature"
type="constructor_signature"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="constructor_signature">
    <xs:sequence>
      <xs:element name="method_flags" type="method_flag"
minOccurs="0"/>
      <xs:element name="qualified_class_name"
type="qualified_type_name"/>
      <xs:element name="parameters" type="parameter_types"
```

```
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="method_signature_def">
    <xs:sequence>
      <xs:element name="method_signature" type="method_signature"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="method_signature">
    <xs:sequence>
      <xs:element name="method_flags" type="method_flag"
minOccurs="0"/>
      <xs:element name="return_type" type="type_pattern"/>
      <xs:element name="qualified_method_name" type="qualified_name"/>
      <xs:element name="parameters" type="parameter_types"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="field_signature">
    <xs:sequence>
      <xs:element name="field_flags" type="field_flag" minOccurs="0"/>
      <xs:element name="field_type" type="type_pattern"/>
      <xs:element name="qualified_field_name" type="qualified_name"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="primitive_pointcut">
    <xs:sequence>
      <xs:element name="time" type="time_modifier" maxOccurs="3"/>
      <xs:element name="joinpoint_type">
        <xs:complexType>
          <xs:choice>
            <xs:element name="methodcall"
type="method_signature_def"/>
            <xs:element name="methodexecution"
type="method_signature_def"/>
            <xs:element name="constructorcall"
type="constructor_signature_def"/>
            <xs:element name="constructorexecution"
type="constructor_signature_def"/>
            <xs:element name="get" type="field_signature"/>
            <xs:element name="set" type="field_signature"/>
          </xs:choice>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="parameter_types">
    <xs:choice>
      <xs:element name="parameter" type="type_pattern"
maxOccurs="unbounded"/>
      <xs:element name="param_wildcard" type="empty"/>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```

APÉNDICE B
REFERENCIAS BIBLIOGRÁFICAS

[Absil]	Abstract IL http://research.microsoft.com/projects/ilx/absil.aspx
[Aho90]	A. V. Aho. "Compiladores: Principios, Técnicas y Herramientas". Addison–Wesley Iberoamericana. 1990.
[Aksit92]	Aksit M., Bergmans L. and Vural S. An object–oriented language–database integration model: The composition filters approach. ECOOP '92. LNCS 615, Springer–Verlag.
[Aksit94]	Mehmet Aksit, Jan Bosch, William van der Sterren, Lodewijk Bergmans. Real Time Specification Inheritance Anomalies and Real–Time filters. In Mario Tokoro and Remo Pereschi, editors. European Conference on Object–Oriented Programming ECOOP 94. pp 386–407. Bologna, Italia. Julio de 1994.
[Altman04]	R.Altman and A.Cyment. SetPoint: a semantic. approach for the pointcut resolution in AOP. Msc. Thesis, Universidad de Buenos Aires. 2004
[Altman05]	Ruben Altman, Alan Cyment, and Nicolas Kicillof. On the need for setpoints. 2nd European Interactive Workshop on Aspects in Software (EIWAS'05), Septiembre de 2005.
[Álvarez98]	Darío Álvarez Gutiérrez. "Persistencia Completa para un Sistema Operativo Orientado a Objetos usando una Máquina Abstracta con Arquitectura Reflectiva". Tesis Doctoral. Departamento de Informática. Universidad de Oviedo. Marzo de 1998.
[Andersen98]	Anders Andersen. "A note on reflection in Python 1.5" Distributed Multimedia Research Group Report. MPG–98–05, Lancaster University (Reino Unido). Marzo de 1998.
[Anthony05]	D. Anthony, M. Leung, W. Srisa–an. To JIT or not to JIT: The Effect of Code Pitching on the Performance of .NET Framework. UNION Agency – Science Press, Plzen, Czech Republic (2005)
[AOM03]	Workshop: "AOM: Aspect–Oriented Modeling with UML" at AOSD2003 conference.
[AOPAlliance]	AOP Alliance (Java/J2EE AOP standards) http://aopalliance.sourceforge.net/
[AORE3]	Workshop: "Early Aspects 2003: Aspect–Oriented Requirements Engineering And Architecture Design" at AOSD2003 conference.
[AORTA]	AORTA (Aspect–Oriented Run–Time Architecture) http://www.st.informatik.tu–

	darmstadt.de/static/pages/projects/AORTA/AORTA.jsp
[AOSDEurope]	Survey of Aspect–oriented Languages and Execution Models. ID: AOSD–Europe–VUB–01. AOSD – Europe. Mayo de 2005
[Arachne]	Arachne AOP http://www.emn.fr/x–info/arachne/index.html
[Archer01]	Archer, T., 2001. Inside C#, Microsoft Press.
[AspectBench]	AspectBench Compiler, abc. http://www.aspectbench.org/
[AspectC++]	AspectC++ homepage http://www.aspectc.org
[AspectDNG]	AspectDNG homepage http://aspectdng.tigris.org
[AspectJ]	AspectJ homepage. http://eclipse.org/aspectj
[AspectJb]	The AspectJ 5 Development Kit Developer's Notebook. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj–home/doc/ajdk15notebook/index.html
[AspectJc]	AspectJ Quick Reference. http://www.eclipse.org/aspectj/doc/released/quick5.pdf
[AspectR]	AspectR for Ruby homepage http://aspectr.sourcforge.net
[AspectS]	AspectS for Squeak/Smalltalk homepage http://www.prakinf.tu–ilmenau.de/~hirsch/Projects/Squeak/AspectS/
[AspectWerkz]	AspectWerkz homepage http://aspectwerkz.codehaus.org
[Assumpcao93]	Jecel Assumpcao Jr. "O Sistema Orientado a Objetos Merlin em Máquinas Paralelas". Anais do V SBAC–PAD. Florianópolis (Brasil). Septiembre de 1993.
[Assumpcao95]	Jecel Assumpcao Jr. y Sergio Takeo Kufuji. "Bootstrapping the Object–Oriented Operating System Merlin: Just add Reflection". Meta'95 Workshop on Advances in Metaobject Protocols and Reflection. ECOOP. 1995.
[Autonomic]	Autonomic Computing. IBM. http://www–3.ibm.com/autonomic/index.shtml
[Axon]	Axon homepage http://www.st.informatik.tu–darmstadt.de/static/pages/projects/AORTA/Axon.jsp
[Baker97]	Seán Baker. CORBA Distributed Objects. Addison–Wesley, ACM Press. ISBN 0–201–92475–7. 1997.
[Baker02]	Jason Baker, Wilson Hsieh. Runtime aspect weaving through metaprogramming. AOSD 2002 conference Proceedings, Pages: 86 – 95
[BAT]	BAT library homepage. http://www.st.informatik.tu–darmstadt.de/static/pages/projects/BAT/BAT2.html
[BCEL]	Byte Code Engineering Library (BCEL) Home Page. http://jakarta.apache.org/bcel/
[Belapurkar04]	Belapurkar, Abhijit. Use AOP to maintain legacy Java applications. Techniques for dealing with complex and unfamiliar Java code. http://www–128.ibm.com/developerworks/java/library/j–aopsc2.html
[Bencomo05]	N. Bencomo, G. Blair, G. Coulson, P. Grace, A. Rashid. Reflection and AOP meet again: Runtime Reflective Mechanisms for Dynamic Aspects. 1st International Workshop on Aspect–Oriented Middleware Development (AOMD05) Proceedings
[Beners96]	Tim Beners–Lee, R. Fielding y H. Frystyk. "Hypertext Transfer Protocol – HTTP 1.0". HTTP Working Group. Enero de 1996.
[Bergel05]	Alexandre Bergel and Stéphane Ducasse, Scoped and Dynamic

	Aspects with Classboxes, In RSTI – L'Objet (programmation par aspects), Volume 11, Number 3, pp. 53—68, 2005
[Bergmans94]	L. Bergmans. "Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object–Oriented Programs". Ph. D. Dissertation. University of Twente. 1994
[Blair97]	Gordon S. Blair, Geoff Coulson. "The Case for Reflective Middle–ware". Proceedings of the 3 rd Cabernet Plenarg Workshop. Rennes (Francia). Abril de 1997.
[Bockisch04]	Christoph Bockisch, Michael Haupt, Mira Mezini and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points in Proceedings of International Conference on Aspect–Oriented Software Development (AOSD'04). Pages 83–92
[Böllert99]	Böllert, K. On Weaving Aspects. In: European Conference on Object–Oriented Programming (ECOOP) Workshop on Aspect Oriented Programming. 1999.
[Boner04]	Jonas Poner. AspectWerkz: Dynamic AOP for Java, AOSD 2004, http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf Marzo de 2004
[Booch94]	Grady Booch. "Análisis y diseño orientado a objetos con aplicaciones". Editorial Addison–Wesley / Díaz de Santos. 1994.
[Borland]	http://www.borland.com/
[Brown98]	A.W. Brown, K.C. Wallnau. "The Current State of CBSE", IEEE Software, Septiembre/Octubre 1998.
[BSDLicense]	http://opensource.org/licenses/bsd–license.php
[Burton02]	K. Burton. .NET Common Language Runtime Unleashed. ISBN: 0672321246 (2002)
[Campione99]	Mary Campione, Kathy Walrath. "The Java Tutorial. Second Edition. Object Oriented Programming for Internet". The Java Series. Sun Microsystems. 1999.
[Cardelli97]	Luca Cardelli. "Type Systems". Handbook of Computer Science and Engineering, Chapter 103. CRC Press. 1997.
[CCosta05]	C. Costa, J. Pérez, J.A. Carsí. Estudio e implementación de un modelo de arquitecturas orientado a aspectos y basado en componentes sobre tecnología .NET. DSIC–II/11/05 http://www.dsic.upv.es/docs/bib–dig/informes/etd–09212005–113129/DSIC–II–11–05.TechReport.ccosta.pdf
[Cecil]	Cecil homepage http://www.mono–project.com/Cecil
[Chambers91]	Craig Chambers and David Ungar. "Making Pure Object–Oriented Languages Practical". OOPSLA '91 Conference Proceedings, Phoenix, AZ. Octubre de 1991.
[Chiba98]	Chiba, S., Michiaki, T., 1998. A Yet Another java.lang.Class. In: European Conference on Object–Oriented Programming (ECOOP) Workshop on Reflective Object Oriented Programming and Systems.
[Chitchyan04]	Chitchyan,R., I. Sommerville (2004) AOP and Reflection for Dynamic Hyperslices. Workshop on Reflection, AOP and Meta–Data for Software Evolution (held with ECOOP 2004), Oslo, Norway.
[CLIFileReader]	CLIFileReader homepage http://www.dsg.cs.tcd.ie/index.php?category_id=193

[CodeDOM]	Espacio de nombres System.CodeDOM .Net Framework http://msdn2.microsoft.com/es–es/library/system.codedom(VS.80).aspx
[Cohen04]	T. Cohen, J. Gil. AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework. ECOOP 2004 – Object–Oriented Programming, 3086 pages 219–243, Springer–Verlag.
[Cointe92]	Pierre Cointe, Jacques Malenfant, Chistophe Dony, Philippe Mulet. "Etude de la réflexion de comportement dans le langage Self". Premières Journées Représentation par Objects. La Grande Motte (Francia). 1992.
[Colwel88]	Robert P. Colwel, Edward F. Gehringer, E. Douglas Jensen. "Performance Effects of Architectural Complexity in the Intel 432". ACM Transactions on Computer Systems, Vol. 6, No. 3. Agosto de 1988.
[Composition Filters]	Homepage of Aspect–Oriented Research on Composition Filters, University of Twente, The Netherlands. http://trese.cs.utwente.nl/composition_filters/.
[Cueva91]	Juan Manuel Cueva Lovelle. "Lenguajes, Gramáticas y Autómatas". ISBN: 84–600–7871–X. Diciembre de 1991.
[Cueva92b]	Juan Manuel Cueva Lovelle. "Tablas de Símbolos en Procesadores de Lenguajes". Cuaderno Didáctico número 54. Departamento de Matemáticas. Universidad de Oviedo. 1992.
[Cueva93]	Juan Manuel Cueva Lovelle. "Análisis Léxico en Procesadores de Lenguaje". Cuaderno Didáctico número 48. Departamento de Matemáticas. Universidad de Oviedo. 1993.
[Cueva94]	J. M. Cueva Lovelle, P.A. García Fuente, B. López Pérez, C. Luengo Díez y M. Alonso Requejo. "Introducción a la Programación Estructurada y Orientada a Objetos con Pascal". ISBN: 84–600–8646–1. 1994.
[Cueva95]	Juan Manuel Cueva Lovelle. "Análisis Sintáctico en Procesadores de Lenguaje". Cuaderno Didáctico número 61. Departamento de Matemáticas. Universidad de Oviedo. 1995.
[Cueva95b]	Juan Manuel Cueva Lovelle. "Análisis Semántico en Procesadores de Lenguaje". Cuaderno Didáctico número 62. Departamento de Matemáticas. Universidad de Oviedo. 1995.
[Cueva98]	Juan Manuel Cueva Lovelle. "Conceptos Básicos de Procesadores de Lenguaje". Cuaderno Didáctico número 10. Editorial Servitec.. Diciembre de 1998.
[Czarnecki98]	K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment–Based Component Models. Ph.D. thesis, in preparation, Technische Universität Ilmenau, Germany, 1998.
[Dageforde2000]	Mary Dageforde. "Security in Java2 SDK 1.2. The Java™ Tutorial." Javasoft, Sun Microsystems. Febrero de 2000.
[DAJ]	DAJ: Demeter in AspectJ http://www.ccs.neu.edu/research/demeter/DAJ
[Danker04]	Jens Danker. Aspect Orientation through Runtime Evolution. diploma thesis March 2004 http://www.st.informatik.tu–darmstadt.de/database/theses/thesis/RuByCoM.pdf?id=8

[Dantas03]	Dantas, A., Borba, P. "Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications with Aspects", Proceedings of SugarloafPLoP 2003 Conference, 2003.
[DARPA]	Defense Advanced Research Projects Agency (DARPA) http://www.darpa.mil/
[Demeter]	Adaptative Programming. http://www.ccs.neu.edu/research/demeter/biblio/dem–book.html
[DeFraine05]	De Fraine, B., Vanderperren, W., Suvee, D. and Brichau, J. Jumping Aspects Revisited. In Proceedings of DAW 2005, Chicago, USA, March 2005.
[Diehl00]	Diehl, S., Hartel, P., and Sestoft, P. 2000. Abstract Machines for Programming Language Implementation. Elsevier Future Generation Computer Systems, Vol. 16 (7).
[Dijkstra76]	Dijkstra E.W. A Discipline of Programming. Prentice Hall, 1976.
[DJ]	DJ: Dynamic Structure–Shy Traversals and Visitors in Pure Java http://www.ccs.neu.edu/research/demeter/DJ/
[Dmitriev02]	M. Dmitriev. Application of the HotSwap Technology to Advanced Profiling. In ECOOP 2002 International Conference.
[DotGNU]	DotGNU project – GNU Freedom for the Internet. http://dotgnu.info
[DotSPECT]	DotSPECT/.SPECT homepage http://dotspect.tigris.org/
[Douence99]	Rémi Douence, Mario Südholt. "The next 700 Reflective Object–Oriented Languages". École des mines de Nantes. Dept. Informatique (Francia). Technical report no.: 99–1–INFO. 1999.
[Dounce01]	R. Douence, O. Motelet, M. Südholt "A formal definition of crosscuts", Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns, LNCS Springer Verlag, Septiembre de 2001.
[Dounce02]	R. Douence, M. Südholt "A model and a tool for Event–Based AOP", technical report no. 02/11/INFO, École des Mines de Nantes, at LMO'03, 2nd edition, diciembre de 2002.
[Dounce04]	R. Douence, P. Fradet, M. Südholt "Composition, Reuse and Interaction Analysis of Stateful Aspects", 3rd Int. Conf. on Aspect–Oriented Software Development (AOSD'04), Marzo de 2004.
[Dounce05]	R. Douence, T. Fritz, N. Loriant, J.–M. Menaud, M. Ségura, M. Südholt. "An expressive aspect language for system applications with Arachne", 4th Int. Conf. on Aspect–Oriented Software Development (AOSD'05), Marzo de 2005
[Eckel2000]	Bruce Eckel. "Thinking in Java, second edition". Prentice Hall. ISBN 0–13–027363–5. 2000.
[Eckel2000b]	Bruce Eckel. "Thinking in C++", second edition, volume 1. Prentice Hall, 2000.
[Eclipse]	Eclipse Project. http://www.eclipse.org
[ECMA334]	Ecma International. Standard ECMA–334. C# Language Specification.
[ECMA335]	ECMA. Standard ECMA–335: Common language infrastructure (CLI).
[EMACS]	XEmacs homepage http://www.xemacs.org/
[Elrad01]	Tzilla Elrad , Mehmet Aksits , Gregor Kiczales , Karl Lieberherr ,

	Harold Ossher. Discussing aspects of AOP. Communications of the ACM October 2001 Volume 44 Issue 10
[Estier02]	T. Estier, "What is BNF notation?," vol. 2002, 2002.
[Ferber88]	Jacques Ferber. “Conceptual Reflection and Actor Languages”. Meta–Level Architectures and Reflection. P. Maes and D. Nardi Editors.North–Holland. 1988.
[Foldoc97]	The Free Online Dictionary. “spaghetti code”. http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=spaghetti+code. 1997.
[Foldoc98]	The Free Online Dictionary. “wrapper”. http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=wrapper. 1998
[Foote90]	Brian Foote. “Object–Oriented Reflective Metalevel Architectures: Pyrite or Panacea?” ECOOP/OOPSLA’90 Workshop on Reflection and Metalevel Architectures. Julio de 1990.
[Foote92]	Brian Foote. “Objects, Reflection, and Open Languages”. Workshop on Object–Oriented Reflection and Metalevel Architetures. ECOOP’92. Utrect (Holanda). 1992.
[FORTE]	http://wwws.sun.com/software/sundev/jde/index.html
[Frei04]	Andreas Frei, Patrick Grawehr, Gustavo Alonso.A Dynamic AOP–Engine for .NET. Technical Report 445, Department of Computer Science, ETH Zürich, Marzo de 2004.
[Freier96]	Alan O. Freier, Philip Karlton y Paul C. Kocker. “The SSL Protocol, version 3.0”. Transport Layer Security Working Group. 1996
[Fuentes03]	Fuentes L, Pinto M, Vallecillo A. HowMDA can help designing component– and aspect–based applications. Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Brisbane, Australia, Septiembre de 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 124–135.
[Goldberg83]	Goldberg A. y Robson D. “Smalltalk–80: The language and its Implementation”. Addison–Wesley. 1983.
[Goldberg89]	Goldberg A. y Robson D. “Smalltalk–80: The language”. Addison–Wesley. 1989.
[Gowing96]	Brendan Gowing, Vinny Cahill. “Meta–Object Protocols for C++: The Iguana Approach”. Distributed Systems Group, Department of Computer Science, Trinity College. Dublin (Irlanda). 1996.
[Gamma94]	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design patterns: elements of reusable object–oriented software, Addison–Wesley Longman Publishing Co., Inc., Boston, MA, 1994
[GNUGPL]	http://www.gnu.org/licenses/licenses.es.html#GPL Licencias GNU GPL
[GNULGPL]	http://www.gnu.org/licenses/licenses.es.html#LGPL Licencias GNU LGPL
[Gold]	GOLD Parsing System. A Free, Multi–Programming Language, Parser http://www.devincook.com/goldparser/
[Golm97]	Michael Golm. “Design and Implementation of a Meta Architecture for Java”. Friedrich–Alexander–Universität. Computer Science Department. Erlangen–Nürnberg, Alemania. Enero de 1997.
[Golm97b]	Michael Golm, Jürgen Kleinöder. “Implementing Real–Time Ac-

	tors with MetaJava". ECOOP'97 Workshop on Reflective Real–time Object–Oriented Programming and Systems. Jyväskylä (Finlandia). Junio de 1997.
[Golm97c]	Michael Golm, Jürgen Kleinöder. "MetaJava – A Platform for Adaptable Operating–System Mechanisms". ECOOP'97 Workshop on Reflective Real–time Object–Oriented Programming and Systems. Jyväskylä (Finlandia). Junio de 1997.
[Golm98]	Michael Golm, Jügen Kleinöder. "metaXa and the Future of Reflection". OOPSLA'98 Workshop in Reflective Programming. Vancouver (Canadá). Octubre de 1998.
[Gosling96]	James Gosling, Bill Joy y Guy Seele. The JavaLanguage Specification. Addison–Wesley. 1996.
[Gowing96]	Brendan Gowing, Vinny Cahill. "Meta–Object Protocols for C++: The Iguana Approach". Distributed Systems Group, Department of Computer Science, Trinity College. Dublin (Irlanda). 1996.
[Greenwood04]	Philip Greenwood and Lynne Blair "Using Dynamic Aspect–Oriented Programming to Implement an Autonomic System". Proceedings of the 2004. Dynamic Aspects Workshop (DAW04) Lancaster 2004
[Haupt02]	Michael Haupt, Mira Mezini, Marianno Cilia and Alejandro Buchmann. Towards Event–Based Aspect–Oriented Runtime Environments. Technical Report TUD–ST–2002–01 http://www.st.informatik.tu–darmstadt.de/database/publications/ data /TR–TUD–ST–2002–01.pdf?id=71
[Haupt06]	Michael Haupt . Virtual Machine Support for Aspect–Oriented Programming Languages. Doctoral Dissertation, at Software Technology Group, Darmstadt University of Technology, 2006
[Hölzle94]	Hölzle, U., Ungar, D. A Third–Generation SELF Implementation: Reconciling Responsiveness with Performance. In: Proceedings of the Object–Oriented Programming Languages, Systems and Applications (OOPSLA) Conference. 1994.
[Hölzle95]	Urs Hölzle, David Ungar. "Do Object–Oriented Languages Need Special Hardware Support?" ECOOP' 95 Conference Proceedings, Springer Verlag Lecture Notes on Computer Science 952. Agosto de 1995.
[Horn03]	Horn, P., "Autonomic Computing: IBM's Perspective on the State of Information Technology", 2003.
[Howe95]	Denis Howe. The Free Online Dictionary. "Abstract Machine". www.foldoc.org. Marzo de 1995.
[Howe99]	Denis Howe. The Free Online Dictionary. "Virtual Machine". www.foldoc.org. Marzo de 1999.
[Huggins96]	Jim Huggins. "Abstract State Machines". www.eecs.umich.edu/ gasm/cover.html. Septiembre de 1996.
[Huggins99]	Jim Huggins. "Abstract State Machines: Introduction". www.eecs.umich.edu/gasm/intro.html. Febrero de 1999.
[Hürsch and Lopes95]	Hürsch, W.L., Lopes, C.V., 1995. Separation of Concerns, Technical Report UN–CCS–95–03, Northeastern University, Boston, USA.
[IBM2000a]	"Multi–Dimensional Separation of Concerns". International Business Machines Corporation, IBM Research. 2000.

[IBM2000b]	"HyperJ: Multi–Dimensional Separation of Concerns for Java". International Business Machines Corporation, IBM Research. 2000.
[IBM2000c]	IBM Corporation. "VM/ESA – An S/390 Platform for Business Solutions. VM/ESA Version 2 Release 4 Specification Sheet. 2000.
[IBM2003]	IBM Jikes Research Virtual Machine homepage oss.software.ibm.com/developerworks/oss/jikesrvm/?origin=jikes
[IBMWeb-Sphere]	IBM WebSphere homepage www–306.ibm.com/software/ websphere
[ICSE2000]	Second Workshop on Multi–Dimensional Separation of Concerns in Software Engineering. ICSE'2000. Limerick (Irlanda). Junio de 2000.
[IIOP.NET]	IIOP.Net Project iiop–net.sourceforge.net/
[ILReader]	Lutz Roeder's ILReader. http://www.aisto.com/roeder/dotnet/
[ISO23271]	ISO Standard ISO/IEC 23271:2006: Common Language Infra-structure (CLI)
[Jacobson98]	Q. Jacobson and P. Cao. Potential and limits of Web prefetching between low-bandwidth clients and proxies. In Third International WWW Caching Workshop, 1998.
[JAsCo06]	Aspect Oriented Programming in JAsCo http://ssel.vub.ac.be/jasco/lib/exe/fetch.php?cache=cache&media= documentation%3Ajasco–vub–session1.ppt
[Javassit]	Javassist (Java Programming Assistant) www.csg.is.titech.ac.jp/ ~chiba/javassist/
[JBoss]	JBoss Application Server homepage http://jboss.org
[JBossAOP]	JBoss AOP homepage http://labs.jboss.com/portal/jbossaop/
[Jbuilder]	http://www.borland.com/jbuilder/
[Jikes]	The Jikes Research Virtual Machine. http://jikesrvm.sourceforge.net/.
[Jones99]	Paul Jones. "VMware Virtual Platform for Linux: Beyond Dual–Booting". Internet.com. Mayo de 1999.
[Johnson04]	R.Johnson and J.Ho eller. Expert One–on–One J2EE Development without EJB. Wiley, 2004.
[Johnson05]	Rod Jonson et All. "Spring Framework Reference Documentation. Version 1.2.6". 2005
[Kalev98]	Danny Kalev. "The ANSI/ISO C++ Professional Programmers Handbook". 1998.
[Kiczales91]	Kiczales, G., Rivieres, J., Bobrow, D.G. The Art of Meta–object Protocol. MIT Press. 1991. ISBN 0262610744.
[Kiczales97]	Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J. Aspect Oriented Programming. In: Proceedings of European Conference on Object–Oriented Programming (ECOOP), vol. 1241 of Lecture Notes in Computer Science, Springer Verlag. 1997. http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales –ECOOP97/for–web.pdf
[Kiczales01]	Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. Getting Started with AspectJ, Communications of

	the ACM, Octubre de 2001.
[Kleinöder96]	Kleinöder, J., Golm, M., 1996. MetaJava: An Efficient Run–Time Meta Architecture for Java™. In: European Conference on Object–Oriented Programming (ECOOP) Workshop on Object Orientation in Operating Systems.
[Kozak2000]	Robert Kozak. "The Dish on Kylix. Cross–Platform Controls. From Windows to Linux, and Back". DelphiZine.com. Mayo 2000.
[Kramer96]	Douglas Kramer. "The Java Platform. A White Paper". Sun Micro–Systems JavaSoft. Mayo de 1996.
[Krasner83]	Glenn Krasner. "Smalltalk–80: Bits of History, Words of Advice". Addison–Wesley. 1983.
[Laddad02]	Ramnivas Laddad. "I want my AOP!". January 2002 issue of Java World http://www.javaworld.com/javaworld/jw–01–2002/jw–0118–aspect.html
[Laddad03]	Ramnivas Laddad. AspectJ In Action. Manning Publicactions. 2003. ISBN 1–930110–93–6
[Lafferty03]	Donal Lafferty, Vinny Cahill. W3C XML Schema for AspectJ aspects survey of AspectJ grammar. http://www.dsg.cs.tcd.ie/uploads/category144/32.zip
[Lam02]	John Lam. CLAW, Cross–Language Load–Time Aspect Weaving on Microsoft's Common Language Runtime. In AOSD 2002 conference
[Lam02b]	John Lam homepage http://www.iunknown.com
[LaMachia02]	Brian A. LaMachia, Sebastian Lange, Matthew Lyons, Rudi Martin, and Kevin T. Price. .NET Framework Security. Addison Wesley, 2002.
[Li06]	Jingyue Li, Axel Anders Kvale, and Reidar Conradi: "A Case Study on Building COTS Component–Based System Using Aspect–Oriented Programming", Journal of Information Science and Engineering, 22:375–390, 2006, ISSN 1016–2364, Institute of Information Science, Academia Sinica (selected papers from Software Engineering Track at SAC'2005).
[Lieberherr96]	Karl J. Lieberherr. "Adaptive Object Oriented Software: The Demeter Method". PWS Publishing Company. 1996.
[Lindholm96]	Lindholm, T., and Yellin, F. The Java Virtual Machine Specification. Addison Wesley. 1996.
[Lopes97]	Cristina Videira Lopes, "D: A Language Framework for Distributed Programming", Ph.D. thesis, Northeastern University, Nov. 1997
[Lopez06]	López, Benjamín. Tesis Doctoral. Adaptación dinámica de persistencia de objetos mediante reflectividad computacional. 2006
[Loughran06]	N. Loughran, et al., A domain analysis of key concerns – known and new candidates AOSD–Europe Deliverable D43, AOSD–Europe–KUL–6. Febrero de 2006
[Macrakis93]	Macrakis, S. 1993. From UNCOL to ANDF: Progress in Standard Intermediate Languages. Technical Report, Open Software Foundation Research Institute.
[Maes87]	Maes, P. Computational Reflection. PhD. Thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Belgium. 1987.

[Maes87b]	Maes, P. "Issues in Computational Reflection. Meta–Level Architectures and Reflection". Pattie Maes and D. Nardi Editors. North–Holland. Bruselas, Bélgica. Agosto de 1987.
[Mandado73]	Enrique Mandado. "Sistemas Electrónicos Digitales". Marcombo Boixareu Editores. 1973.
[Manzoor06]	K. Manzoor. The Common Language Runtime (CLR) and Java Runtime Environment (JRE). http://www.codeproject.com/dotnet/clr.asp
[Matthijs97]	Matthijs, F., Joosen, W., Vanhaute, B., Robben, B., Verbaten, P., Aspects should not die. In: European Conference on Object–Oriented Programming (ECOOP) Workshop on Aspect–Oriented Programming. 1997.
[Megginson2000]	David Megginson. "SAX 2.0: The Simple API for XML". http://www.meggison.com.
[Meijer01]	Meijer, K. and Gough, J. Technical Overview of the Common Language Runtime. 2001. http://docs.msdnaa.net/ark/Webfiles/whitepapers.htm.
[MetaBorg]	MetaBorg. http://www.stratego–language.org/Stratego/MetaBorg
[Mevel87]	Mével A. y Guéguen T. "Smalltalk–80". Mac Millan Education, Houndmills, Basingstoke. 1987.
[Mezini03]	M. Mezini and K.Ostermann. Conquering Aspects with Caesar. In (M. Aksit ed.) Proceedings of the 2nd International Conference on Aspect–Oriented Software Development (AOSD), March 17–21, 2003, Boston, USA. ACM Press, pp. 90–100
[MicrosoftNET]	http://msdn.microsoft.com/library/default.asp?url=/library/en–us/cpguide/html/cpovrIntroductionToNETFrameworkSDK.asp
[MicrosoftRemoting]	.NET Remoting web page http://msdn.microsoft.com/webservices/remoting/default.aspx
[MITLicense]	The MIT License http://www.opensource.org/licenses/mit–license.php
[Mono]	The Mono Project http://www.mono–project.com/Main_Page
[Murata94]	Kenichi Murata, R. Nigel Horspool, Yasuhiko Yokote, Erig G. Mannig, Mario Tokoro. "Cognac: a Reflective Object–Oriented Programming System using Dynamic Compilation Techniques". Proceedings of the annual Conference of Japan Society of Software Science and Technology (JSSS'94). Octubre de 1994.
[Murison05]	Murison, Nicholas John. .NET Framework Security http://www.urgusabic.net/docs/dotnet–framework–security.pdf
[Nicoara05]	Angela Nicoara, Gustavo Alonso: Dynamic AOP with PROSE. In: Proceedings of International Workshop on Adaptive and Self–Managing Enterprise Applications (ASMEA 2005) in conjunction with the 17th Conference on Advanced Information Systems Engineering (CAISE 2005), Porto, Portugal, Junio de 2005.
[Novell]	Novell www.novell.com
[NUnit]	NUnit Home page. http://nunit.org/
[O'Brien01]	O'Brien, L., The First Aspect–Oriented Compiler, Software Development Magazine. Septiembre de 2001
[OMG95]	Object Management Group (OMG). "The Common Object Request Broker: Architecture and Specification". Julio de 1995.

[OMG96]	Object Management Group (OMG). "Description of New OMA Reference Model. Draft 1". Mayo de 1996.
[OMG97]	Object Management Group (OMG). "A Discussion of the Object Management Architecture". Enero de 1997.
[OMG98]	Object Management Group (OMG). "CORBA Components. CORBA 3.0 Draft Specification". Noviembre de 1998.
[OOPSLA99]	First Workshop on Multi–Dimensional Separation of Concerns in Object–Oriented Systems. OOPSLA'99. Denver (EE.UU.). Noviembre de 1999.
[Orfali98]	Robert Orfali, Dan Harkey. "Client/Server Programming with Java and CORBA". 2ª Edición. Wiley Editorial. 1998.
[Ortin01]	Ortín, F., and Cueva, J. M. Building a Completely Adaptable Reflective System. European Conference on Object Oriented Programming ECOOP'2001. Workshop on Adaptive Object–Models and Metamodeling Techniques, Budapest, Hungary. Junio de 2001.
[Ortin02]	Francisco Ortín, Juan Manuel Cueva. Implementing a real computational–environment jump in order to develop a runtime–adaptable reflective platform. ACM SIGPLAN Notices, Volume 37, Issue 8. Agosto de 2002
[Ortin02b]	Francisco Ortín Soler. Sistema computacional de programación flexible diseñado sobre una máquina abstracta reflectiva no restrictiva. ISBN: 84–8317–304–2. Tesis Doctoral. Febrero de 2002.
[Ortin03]	Francisco Ortín and Juan Manuel Cueva. Non–Restrictive Computational Reflection. Elsevier Computer Software & Interfaces. Volume 25, Issue 3, Pages 241–251. Junio de 2003.
[Ortin03b]	Francisco Ortín, Belén Martínez, Juan Manuel Cueva. The Reflective nitrO Abstract Machine. ACM SIGPLAN Notices. Junio de 2003.
[Ortin04]	Francisco Ortín Soler, Juan Manuel Cueva Lovelle. Dynamic Adaptation of Application Aspects. Elsevier Journal of Systems and Software, Volume 71, Issue 3. Mayo de 2004.
[Ortin05]	Francisco Ortín Soler, José M. Redondo, Luis Vinuesa, Juan M. Cueva . Adding Structural Reflection to the SSCLI. Journal of .Net Technologies, Volume 3, Number 1–3, pp. 151–162. Mayo de 2005.
[Ossher99]	H. Ossher, P. Tarr. "Multi–Dimensional Separation of Concerns using Hyperspaces". IBM Research Report 21452. Abril de 1999.
[PARC]	Xerox PARC Research http://www.parc.com/
[Parnas72]	Parnas, D. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, Vol. 15, No. 12. 1972.
[PE/COFF]	Microsoft Portable Executable and Common Object File Format Specification http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
[Pinto01]	Pinto, M., Amor, M., Fuentes, L., Troya, J.M., Run–Time Coordination of Components: Design Patterns vs. Component & Aspect based Platforms. In: European Conference on Object–Oriented Programming (ECOOP) Workshop on Advanced Separation of Concerns. 2001.

[Pinto02]	M. Pinto, L. Fuentes, M.E. Fayad, J.M. Troya. Separation of Coordination in a Dynamic Aspect Oriented Framework. AOSD 2002 Proceedings, Pages 134–140
[Pnet]	DotGNU Portable.NET http://dotgnu.info/pnet.html
[Popovici01]	Popovici, A., Gross, T., Alonso, G., 2001. Dynamic Homogenous AOP with PROSE, Technical Report, Department of Computer Science, ETH Zürich, Switzerland.
[Popovici02]	Popovici, A., Gross, T., Alonso, G. Dynamic Weaving for Aspect Oriented Programming. In 1st Intl. Conf. on Aspect–Oriented Software Development, Enschede, The Netherlands Agosto de 2002
[Popovici03]	Popovici, A., Alonso, G., Gross, T. Juis–In–Time Aspects: Efficient Dynamic Weaving for Java. In 2nd Intl. Conf. on Aspect–Oriented Software Development, Boston, USA. Abril de 2003.
[Pryor02]	Jane Pryor, Andres Diaz Pace and Marcelo Campo. Reflecting on Separation of Concerns. Revista de Informática Teórica y Aplicada (RITA) Volume X, No. 2 (2002)
[Psyco06]	Sourceforge.net. Psyco Homepage. http://psyco.sourceforge.net/
[PyPy06]	PyPy Homepage. http://codespeak.net/pypy/index.html
[Rail]	RAIL Runtime Assembly Instrumentation Library. http://rail.dei.uc.pt/
[Rajan03]	Rajan H., Sullivan K., EOS: Instance–Level Aspects for Integrated System Design. In Proc. of Joint European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). Helsinki, Finland, Septiembre de 2003.
[Rajan05]	Hridesh Rajan . Classpects: unifying aspect– and object–oriented language design. In ICSE '05: Proceedings of the 27th international conference on Software engineering 2005, Pages 59–68. ACM Press. ISBN: 1–59593–963–2
[Rajan06]	Hridesh Rajan, Robert Dyer, Youssef Hanna, Harish Narayanappa, "Preserving Separation of Concerns through Compilation", In Software Engineering Properties of Languages and Aspect Technologies (SPLAT 06), A workshop affiliated with AOSD 2006. Bonn, Germany
[Raman98]	L. G. Raman. "OSI Systems and Network Management". IEEE Communications Magazine. Marzo de 1998.
[Rashid01]	Rashid, A. 2001. On to Aspect Persistence. In Proceedings of the Second international Symposium on Generative and Component–Based Software Engineering–Revised Papers (October 09 – 12, 2000). G. Butler and S. Jarzabek, Eds. Lecture Notes In Computer Science, vol. 2177. Springer–Verlag, London, 26–36.
[Rashid03]	Rashid, A. and Chitchyan, R. 2003. Persistence as an aspect. In Proceedings of the 2nd international Conference on Aspect–Oriented Software Development (Boston, Massachusetts, March 17 – 21, 2003). AOSD '03. ACM Press, New York, NY, 120–129.
[Redondo06]	José Redondo López, Francisco Ortín Soler, Juan Cueva Lovelle Diseño de primitivas de reflexión estructural eficientes integradas en SSCLI. JISBD (Jornadas en Ingeniería del Software y Bases de Datos) 06. Sitges, 4–6 de Octubre. Publicado por Cinme, Barcelo-

	na
[Redondo07]	José Manuel Redondo López. Tesis Doctoral. Mejora del rendimiento de las primitivas de reflexión estructural mediante técnicas de compilación JIT. 2007
[Reina00]	Antonia Mª Reina Quintero, Visión General de la Programación Orientada a Aspectos. Informes del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.
[Remoting-CORBA]	http://remoting–corba.sourceforge.net/
[Ritchie78]	D.M. Ritchie y B. W. Kerninghan. "The C Programming Language". Prentice Hall. 1978.
[Rivières84]	J. des Rivières, B. C. Smith. "The Implementation of Procedurally Reflective Languages". Proceedings of ACM Symposium on Lisp and Functional Programming. 1984.
[Rossum01]	Rossum, G., Python Reference Manual. Fred L. Drake Jr. Editor, Relesase 2.1. 2001.
[Ruby]	RubyCentral, The soruce for Ruby. http://www.rubycentral.com/
[Ruby04]	Dave Thomas, with Chad Fowler and Andy Hunt. Programming Ruby. The Pragmatic Programmer's Guide, Second Edition ISBN 0–9745140–5–5. 2004
[Schmied02]	Schmied, F. AOP with .NET. http://wwwse.fhs–hagenberg.ac.at/se/berufspraktika/2002/se99047/contents/
[Schult02]	W. Schult and A. Polze. Aspect–oriented programming with C# and .NET. In International Symposium on Object–oriented Real–time distributed Computing (ISORC), pages 241–248, Crystal City, VA, USA. 2002.
[Schult02b]	Wolfgang Schult and Andreas Polze Dynamic Aspect–Weaving with .NET Workshop zur Beherrschung nicht–funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen, TU Berlin, Germany. Noviembre de 2002.
[Schult03]	Wolfgang Schult, Andreas Polze Speed vs. Memory Usage – An Approach to Deal with Contrary Aspects. The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at the International Conference on Aspect–Oriented Software Development, Boston, Massachusetts. Marzo de 2003.
[Schultz03b]	Wolfgang Schult and Peter Tröger, Loom.NET – an Aspect Weaving Tool, in Workshop on Aspect–Oriented Programming, ECOOP'03, Darmstadt 2003.
[Segura–Devillechaise03]	Marc Ségura–Devillechaise, Jean–Marc Menaud, Gilles Muller, Julia L. Lawall. Web cache prefetching as an aspect: towards a dynamic–weaving based solution. AOSD 2003 Proceedings, Pages: 110 – 119.
[Segura–Devillechaise03b]	Marc Ségura–Devillechaise and Jean–Marc Menaud microDyner: Un noyau efficace pour le tissage dynamique d'aspects sur processus natif en cours d'exécution, in LMO2003, Hermès, Vannes, Feb 2003
[SetPoint]	http://setpoint.codehaus.org
[SiemensSE]	Siemens Software & Engineering

	http://w4.siemens.de/ct/en/technologies/se/index.html
[Smith82]	B. C. Smith. "Reflection and Semantics in a Procedural Language".MIT–LCS–TR–272. Massachusetts Institute of Technology. Cambridge (EE.UU.). 1982.
[Smith92]	Robert Smith, Aaron Sloman y John Gibson. "PROLOG's Two–Level Virtual Machine Support for Interactive Languages". Research Directions in Cognitive Science, v.5. 1992.
[Smith95]	Randall B. Smith, David Ungar. "Programming as an Experience: The Inspiration for Self". Sun Microsystems Laboratories. 1995.
[Soares02]	Soares, S., Laureano, E., and Borba, P. 2002. Implementing distribution and persistence aspects with AspectJ. In Proceedings of the 17th ACM SIGPLAN Conference on Object–Oriented Programming, Systems, Languages, and Applications (Seattle, Washington, USA, November 04 – 08, 2002). OOPSLA '02. ACM Press, New York, NY, 174–190.
[Source-Weave.Net]	SourceWeave.Net homepage http://www.dsg.cs.tcd.ie/?category_id=438
[Spring]	Spring Home Page. http://www.springframework.org/.
[SpringAOP]	Spring AOP (from the Spring reference documentation). http://www.springframework.org/docs/reference/aop.html.
[Steel60]	T. B. Steel Jr. "UNCOL: Universal Computer Oriented Language Revisited". Datamation. Enero–Febrero de 1960.
[Steele90]	Jr. Steele "Common Lisp: The Language". Segunda Edición. Digital Press, 1990.
[Strong58]	Strong, J., Wegstein, J., Tritter, A., Olsztyn, J, Mock, O., and Steel, T. The problem of programming communication with changing machines: a proposed solution. Communications of the ACM, 1(8): 12–18. 1958.
[Stroustrup98]	Bjarne Stroustrup. "The C++ Programming Language". Third Edition. Addison–Wesley. Octubre de 1998.
[Sullivan01]	Gregory T. Sullivan. Aspect–oriented programming using reflection and metaobject protocols. Communications of the ACM. October 2001/ Vol 44 No. 10 Pages: 95 – 97.
[Sun95]	"The Java Virtual Machine Specification". Release 1.0 Beta Draft. Sun Microsystems Computer Corporation. Agosto de 1995.
[Sun96]	"JavaBeans™ 1.0 API Specification". Sun Microsystems Computer Corporation. Diciembre de 1996.
[Sun97c]	"Java™ Native Interface Specification". JavaSoft. Sun Microsystems. Mayo de 1997.
[Sun97d]	"Java Core Reflection. API and Specification". JavaSoft. Enero de 1997.
[Sun97e]	"Java™ Object Serialization Specification". JavaSoft. Sun Microsystems. Febrero de 1997.
[Sun98]	"The Java HotSpot Virtual Machine Architecture". White Paper. Sun Microsystems. 1998.
[Sun99]	"Dynamic Proxy Classes". Sun Microsystems, Inc. 1999.
[Suvee03]	Suvee, D., Vanderperren, W. and Jonckers, V. JAsCo: an Aspect–Oriented approach tailored for Component Based Software Development. In Proceedings of international conference on aspect–

	oriented software development (AOSD), Boston, USA, pp 21–29, ISBN 1–58113–660–9, ACM Press. Marzo de 2003.
[Szyperski02]	Szyperski, C. Component Software. Beyond Object–Oriented Programming. Addison–Wesley / ACM Press, 2002 ISBN 0–201–74572–0
[Tanter01]	Éric Tanter, Noury Bouraqadi, Jacques Noyé. Reflex – Towards an Open Reflective Extension of Java. In: Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'01). 2001
[Tanter03]	Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. Éric Tanter, Jacques Noyé, Denis Caromel, Pierre Cointe. In: Proceedings of the 18th ACM SIGPLAN Conference on Object–Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)
[Tanter04]	Eric Tanter and Jacques Noyé. Motivation and Requirements for a Versatile AOP Kernel. In: European Interactive Workshop on Aspects in Software (EIWAS'04)
[Tanter05]	Éric Tanter and Jacques Noyé. A Versatile Kernel for Multi–Language AOP. In: Proceedings of the ACM International Conference on Generative Programming and Component Engineering (GPCE'05)
[Tanter06a]	Éric Tanter. Aspects of Composition in the Reflex AOP Kernel. In: Proceedings of the 5th International Symposium on Software Composition (SC 2006)
[Tanter06b]	Éric Tanter. An Extensible Kernel Language for AOP. In: Proceedings of the AOSD Workshop on Open and Dynamic Aspect Languages (ODAL 2006)
[Tarr99]	Tarr, P., Ossher, H., Harrison, W., Sutton, S., N Degrees of separation: Multi–Dimensional Separation of Concerns. In: Proceedings of the 1999 International Conference on Software Engineering.
[Tatsubori98]	Michiaki Tatsubori, Shigeru Chiba. “Programming Support of Design Patterns with Compile–time Reflection”. OOPSLA’98 Workshop in Reflective Programming. Vancouver (Candada). Octubre de 1998.
[TIOBE]	TIOBE Software. The Coding Standards Company www.tiobe.com
[TldpRuby]	Guia del usuario de Ruby. http://es.tldp.org/Manuales–LuCAS/doc–guia–usuario–ruby/doc–guia–usuario–ruby–html/
[Trados96]	Alain Trados y Eric Uber. “Using Borland’s Delphi and C++ Together”. A technical Paper for Developers. Borland Online. Marzo de 1996.
[Turing36]	A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. Proceedings of The London Mathematical Society, Series 2, 42. 1936.
[Ungar87]	D. Ungary R. B. Smith. “SELF: The Power of Simplicity”. In OOPSLA’87 Conference Proceedings. Published as SIGPLAN Notices, 22, 12, 227–241. 1987.
[Vanderperren04]	Vanderperren, W. and Suvee, D. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In proceedings of Dynamic Aspects Workshop (DAW), published as RIACS Technical Report 04.01.,

	Lancaster, UK. Marzo de 2004
[Vasseur04]	Vasseur, Alexandre. Dynamic {AOP} and Runtime Weaving for {Java}—{How} does {AspectWerkz} Address It?. In proceedings of DAW: Dynamic Aspects Workshop 2004. http://aosd.net/2004/workshops/daw/Proc–2004–Dynamic–Aspects.pdf
[Venners98]	Bill Venners. "Inside the Java Virtual Machine". Java Masters. McGraw Hill. 1998.
[Verspecht03]	Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V. JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services. In Proceedings of Second Nordic Conference on Web Services NCWS'03, Vaxjo, Sweden. Published in Mathematical modeling in Physics, Engineering and Cognitive Sciences, Vol. 8, pp 125–137, November 2003. .
[W3C98]	"Extensible Markup Language (XML) 1.0". World Wide Web Consortium. Febrero de 1998.
[W3C98b]	"Level 1 Document Object Model Specification". Version 1.0. W3C Working Draft, WD–DOM–19980720. Julio de 1998.
[W3CWSDL]	"Web Services Description Language (WSDL) 1.1" W3C Note 15 March 2001 http://www.w3.org/TR/wsdl
[W3C01]	"XML Schema 1.0" World Wide Web Consortium. Mayo de 2001
[W3CSOAP]	SOAP Version 1.2 W3C Recommendation 24 June 2003 http://www.w3.org/TR/2003/REC–soap12–part0–20030624/
[W3CSW]	W3C Semantic Web http://www.w3.org/2001/sw/
[W3CWS]	W3C Web Services Activity http://www.w3.org/2002/ws/
[VMWARE]	VMWare http://www.vmware.com
[Weave.NET]	Weave.NET homepage http://www.dsg.cs.tcd.ie/index.php?category_id=193
[WindowsForms]	Windows Forms .Net Official Site http://www.windowsforms.net/
[Xdoclet]	Xdoclet attribute oriented programming. http://xdoclet.sourceforge.net/
[Yang02]	Yang, Z., "An Aspect–Oriented Approach to Dynamic Adaptation", WOSS 2002, 2002.
[Yokote92]	Y. Yokote. "The New Mechanism for Object–Oriented System Programming". Proceedings of IMSA'92 International Workshop on Reflection and Meta–level Architecture". 1992.
[Zinki97]	Zinky, J.A., Bakken, D.E., Schantz, R.E., Architectural Support for Quality of Service for CORBA Objects, Theory and Practice of Object Systems, 3 (1). 1997.