UNIVERSIDAD DE OVIEDO

ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA EN INFORMÁTICA DE OVIEDO

TRABAJO FIN DE MÁSTER

DSAW: SISTEMA HOMOGÉNEO DE TEJIDO ESTÁTICO Y DINÁMICO INDEPENDIENTE DE LENGUAJE Y PLATAFORMA

José Manuel Félix Rodríguez

Vº Bº del Director del Proyecto

DIRECTORES:

Luis Vinuesa Martínez

Francisco Ortín Soler

Resumen

El **Desarrollo de Software Orientado a Aspectos** *–Aspect Oriented Software Development-* (AOSD) es una eficiente aproximación al principio de la **separación de incumbencias** *–Separation of Concerns-* (SoC). Un punto clave en este paradigma es el momento en que los componentes y los aspectos son tejidos juntos, formando la aplicación final. Las herramientas de tejido estático realizan esta composición previamente a la ejecución de la aplicación. Este tipo de tejido proporciona un buen rendimiento, pero impide adaptar la aplicación en tiempo de ejecución. En respuesta a esta limitación, las herramientas de tejido dinámico realizan la composición de las aplicaciones en tiempo de ejecución. El principal beneficio del tejido dinámico es la flexibilidad que se consigue al poder adaptar las aplicaciones durante su ejecución, pero su principal desventaja es la penalización en el rendimiento de las mismas.

Varios trabajos de investigación señalan la idoneidad de aproximaciones híbridas, que permitan obtener los beneficios de ambos métodos de tejido en la misma plataforma. De esta forma, se aplicaría el tejido estático cuando fuese posible y el tejido dinámico cuándo fuese necesario, así se proporcionaría un mejor equilibrio entre rendimiento en tiempo de ejecución y las adaptaciones dinámicas necesarias.

Este trabajo presenta DSAW (*Dynamic and Static Aspect Weaver*), un sistema orientado a aspectos que soporta tejido estático y dinámico homogéneamente sobre la plataforma .Net. El diseño de DSAW ha sido realizado siguiendo el principio de la SoC, de esta forma el dinamismo (momento de tejido) de una aplicación no interfiere en el desarrollo orientado a aspectos. Un aspecto puede ser usado para adaptar una aplicación estática o dinámicamente, sin necesidad de modificar su código fuente. Además, DSAW es independiente del lenguaje y de la plataforma, y ni código fuente de los componentes ni de los aspectos es necesario para su adaptación.

El objetivo del sistema es permitir que el tejido de aspectos sea independiente del lenguaje, de la plataforma y del momento de tejido. Usando esta herramienta, los desarrolladores pueden crear aspectos antes de la ejecución de la aplicación y tejerlos estáticamente, cuando la información que necesitan es conocida previamente a la ejecución. O pueden crear los aspectos durante la ejecución de la aplicación y tejerlos dinámicamente cuando la información necesaria no sea conocida antes del arranque de la aplicación.

DSAW es una evolución de un proyecto previo –Ready AOP (*Really Dynamic AOP*). DSAW mantiene los principales beneficios de Ready AOP como su dinamismo y la independencia del lenguaje y de la plataforma junto con un conjunto rico de puntos de enlace; ampliándolos con el tejido estático de forma transparente. Esto permite mejorar el equilibrio entre rendimiento y flexibilidad en el AOSD.

Palabras Clave

Programación Orientada a Aspectos, tejido estático, tejido dinámico, homogeneidad, reflectividad, instrumentación de código, rendimiento, adaptabilidad.

Abstract

Aspect Oriented Software Development (AOSD) is an effective realization of the **Separation of Concerns** (SoC) principle. A key issue of this paradigm is the moment when components and aspects are weaved together, composing the final application. Static weaving tools perform application composition previously to its execution. This kind of weaving reduces dynamic aspectation of running application generating applications with a good runtime performance. In response to this limitation, dynamic aspect weaving tools perform application composition at runtime. The main benefit of dynamic weaving is runtime adaptability; its main drawback is runtime performance.

Existing research has identified the suitability of hybrid approaches, obtaining the benefits of both methods in the same platform. Applying static weaving where possible and dynamic weaving when needed provides a balance between runtime performance and dynamic adaptability.

This research presents DSAW (**Dynamic and Static Aspect Weaver**), an aspect-oriented system that supports both dynamic and static weaving homogeneously over the .Net platform. The design of DSAW has been performed following the SoC principle, so that the dynamism (weaving-time) of an application does not interfere in the aspect-oriented development. An aspect can be used to adapt an application both statically and dynamically, without needing to modify its source code. Moreover, DSAW is language and platform neutral, and source code of neither components nor aspects is required.

The main aim of the system is to achieve language, platform and weaving-time neutrality. Using this tool on one hand, developers can create aspects before the execution of the application and to weave them statically, when the necessary information is known previously at runtime execution. On the other hand, developers can also create aspects at runtime execution and to weave dynamically when the necessary information is only known at runtime.

DSAW is the enhancement of a previous AOSD tool –Ready AOP (**Really Dynamic AOP**). DSAW maintains all the advantages of Ready AOP such as a real separation of concerns at runtime, language and platform neutrality, and a wide join-point set; extending its core to provide the programmer a transparent static and dynamic weaving. This improves the balance between performance and adaptability on AOSD.

Keywords

Aspects Oriented Programming, static weaving, dynamic weaving, homogeneous, reflective, code instrumentation, performance, adaptability.

Índice General

Índice Figuras

Índice Tablas

1 Introducción

Las aplicaciones informáticas están formadas por sus incumbencias o funcionalidades principales, e intentan resolver problemas. Una incumbencia[1] es cualquier parte de interés de un programa o aplicación. Incumbencia es un sinónimo de característica, competencia o comportamiento.

Pero además de estas incumbencias principales, existen otro grupo de incumbencias secundarias que deben ser desarrolladas también. Un ejemplo de esto son las métricas que se deben generar de una aplicación, para ello se necesitan generar ficheros de traza de las operaciones realizadas. Esto implica la necesidad de poner mensajes informativos a lo largo del código de toda la aplicación. Sin embargo, los desarrolladores de la aplicación no deberían de preocuparse por estas métricas, ni por la generación de estos mensajes, ya que no son parte de la aplicación. No representan ni una regla de negocio, ni una entidad, ni una relación. Simplemente es un código ortogonal que requiere su duplicación en los sistemas realizados mediante los métodos tradicionales de **programación orientada a objetos** *-object-oriented programming* (OOP - POO).

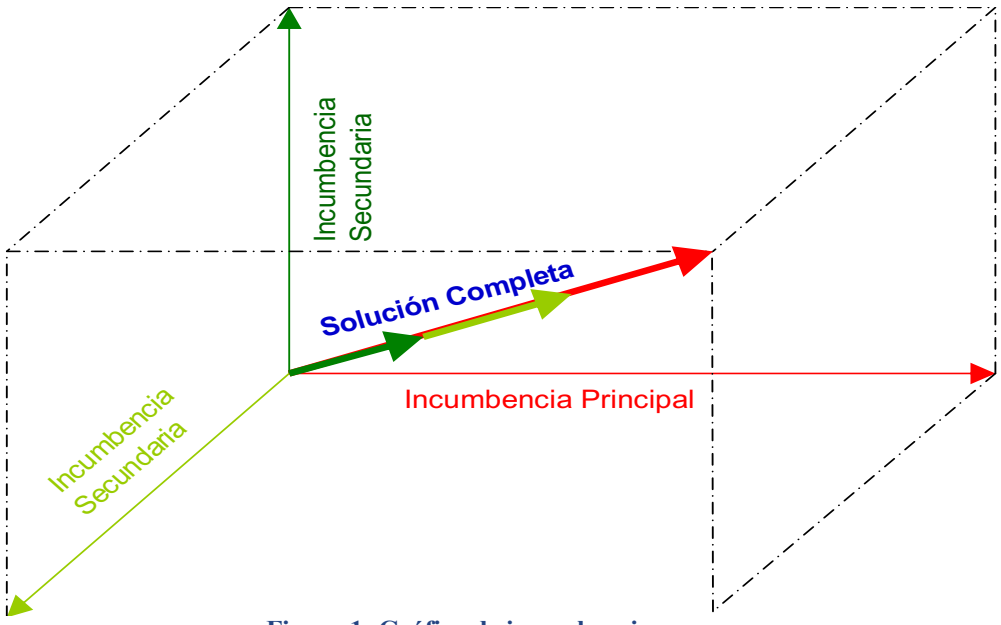


Figura 1: Gráfico de incumbencias.

Las incumbencias secundarias no se pueden expresar fácilmente de una manera modular. Están poco relacionadas con el problema principal que trata de resolver la aplicación, y se encuentran **desperdigadas** y **enredadas** –*scattered* y *tangled*- a lo largo de toda la aplicación junto con otras incumbencias. Aún así, estas incumbencias son muy importantes para el correcto funcionamiento de la aplicación ya que se encargan de tareas como autentificación, **registro** –*logging*-, persistencia, rendimiento y traceado [Ortin04a] entre otras.

Estas incumbencias secundarias son ortogonales a las funcionalidades principales, y pueden ocasionar importantes problemas para el mantenimiento y comprensión de las aplicaciones, o para la reutilización de su código.

[1] Obligación o cargo de hacer algo.

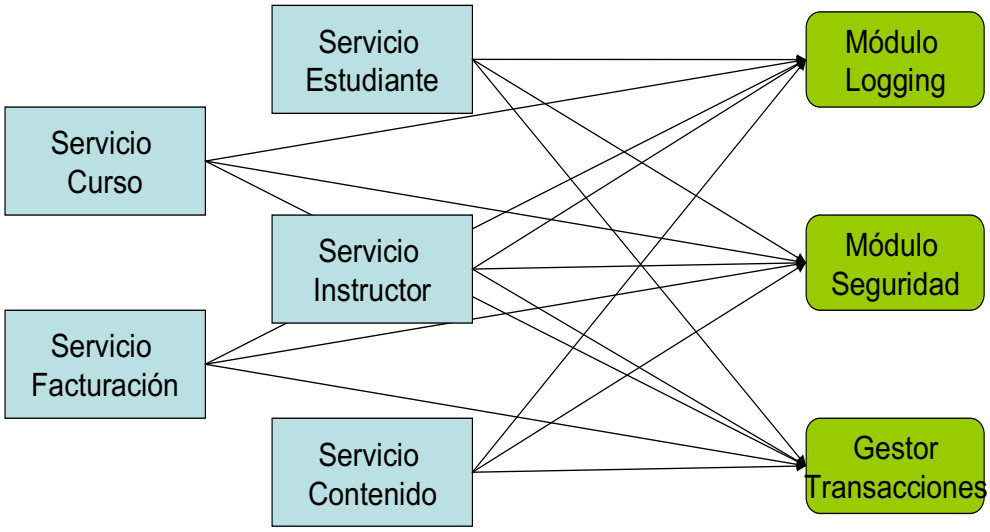


Figura 2: Llamadas desperdigadas *scattered* a módulos [Walls07].

En el ámbito de la Ingeniería del software, muchas investigaciones han llegado a la conclusión que la mejor forma de crear aplicaciones manejables es identificar sus incumbencias. Esta identificación es conocida como el principio de la **Separación de Incumbencias** -*Separation of Concerns*– (SoC) [Parnas72] [Dijkstra76] [Hürsch95]. Este principio [Parnas72] propone que el mejor modo de gestionar la complejidad en el desarrollo del software es a través de la realización de módulos que se solapen entre sí de la forma más mínima, éstos ocultan las decisiones que toman de las tomadas por los otros módulos de la aplicación. De esta forma, se podrían resolver los problemas expuestos anteriormente.

A lo largo de los años, se han desarrollado aproximaciones y técnicas para conseguir la SoC, que han ido evolucionando con el objetivo de hacer frente a las características cambiantes, ámbito y complejidad de las aplicaciones software. Algunas de las técnicas existentes son:

- Separación Multidimensional de Incumbencias [Tarr99].
- Filtros de Composición [Bergmans94].
- Programación Adaptable [Lieberherr96].
- Modelado de Roles [Krinstensen96].
- Programación Orientada a Sujetos [Harrison93].
- Desarrollo Software Orientado a Aspectos (AOSD) [Kiczales97].

El **Desarrollo de Software Orientado a Aspectos** –*Aspect Oriented Software Development*- (AOSD - DSOA) o **Programación Orientada a Aspectos** –*Aspects Oriented Programming*- (AOP - POA) [Kiczales97] es una aproximación concreta a la SoC.

El AOSD ofrece soporte directo para modularizar las diferentes incumbencias que se entremezclan o enmarañan a través de las funcionalidades principales de las aplicaciones. Separando el código de las funcionalidades principales del código de los aspectos entrecruzados, se consigue que el código fuente de la aplicación no esté enmarañado, siendo más fácil depurarla, mantenerla y modificarla [Parnas72].

La AOP extiende el modelo tradicional de la OOP para mejorar la reutilización de código a través de las diferentes jerarquías de objetos. El concepto básico de la AOP es el **aspecto** [Kizcales97], esto es un comportamiento común que típicamente está diseminado a través de métodos, clases, jerarquías de objetos, o incluso modelos enteros de objetos.

Este paradigma utiliza **aspectos** *–aspects-* para modularizar las incumbencias secundarias Su objetivo es construir aplicaciones adaptadas con las incumbencias que no son las funcionalidades principales. Para ello, los aspectos se desarrollan de forma independiente a las funcionalidades principales de la aplicación. Posteriormente las funcionalidades principales y los aspectos formarán la aplicación final.

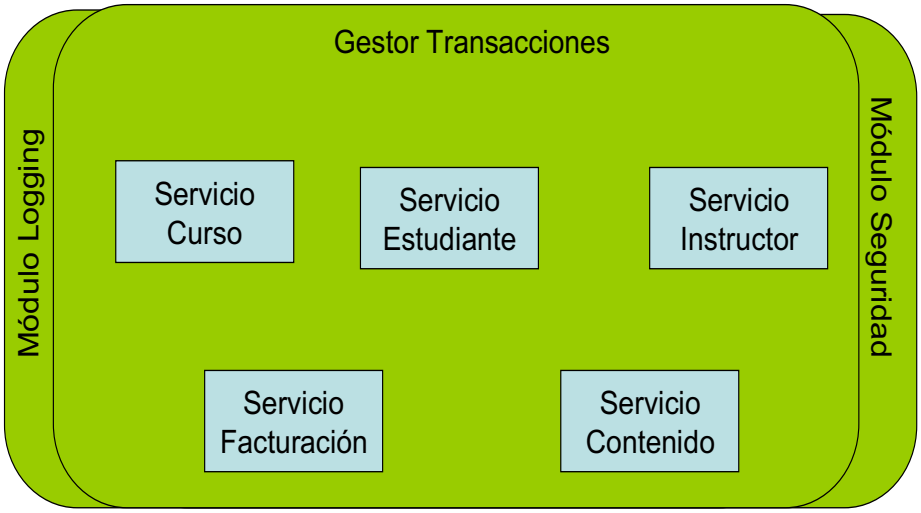


Figura 3: Separación de incumbencias usando AOP [Walls07].

El proceso de integrar los aspectos dentro del código de la aplicación principal se llama **tejido** *–weaving-* o inyección, y la herramienta que realiza este proceso se llama **tejedor de aspectos** *–aspect weaver*. Este tejido puede ser realizado estáticamente si se hace antes de la ejecución de la aplicación, o dinámicamente en tiempo de ejecución.

El tejido estático de aspectos es idóneo en aplicaciones donde el desarrollador conoce la información necesaria para configurar el aspecto antes de ejecutar la aplicación. Se puede realizar durante la compilación de la aplicación, en una fase post-compilación o durante la carga de la aplicación. El sistema de tejido estático de referencia es AspectJ [Kizcales01], que es considerado un estándar de facto por la comunidad de usuarios.

Muchos entornos de programación que ofrecen AOSD emplean tejedores estáticos. Una vez que la aplicación final ha sido compilada, tejida y se está ejecutando, no es posible añadir nuevos aspectos, o eliminar aspectos tejidos previamente durante la ejecución. Esto es suficiente en muchos casos, sin embargo en determinados escenarios es necesario poder adaptar una aplicación durante su ejecución en respuesta a cambios del entorno, o la aparición de nuevos requisitos durante la ejecución de la aplicación, no siendo posible algunas veces detenerlas. Ejemplos de esto son: aplicaciones para adaptar sus competencias específicas en respuesta a cambios en tiempo de ejecución

[Popovici01], aplicaciones para la gestión de requerimientos de calidad del servicio en sistemas distribuidos CORBA [Zinki97], sistemas de **pre-carga en memoria caché** –*cache prefetching*– en un servidor Web [Segura-Deville03] o sistemas con distribución de incumbencias basadas en el equilibrio de la carga [Matthijs97].

Ante estas necesidades, se han desarrollado los sistemas de tejido dinámico, los cuales son poderosas herramientas para crear software adaptable en tiempo de ejecución. Estos sistemas permiten cambiar o modificar una aplicación ejecutándose para poder adaptarse a cambios emergentes o posibles fallos sin detenerse. Ejemplos de estos sistemas son PROSE [Popovici01], JBoss AOP [JBossAOP08], Rapier-LOOM.Net [Schult03] o Wool [Sato03]. Este dinamismo también tiene varias limitaciones que lo hacen no atractivo en algunos casos, como por ejemplo producir una penalización considerable del rendimiento [Haupt04].

Aunque el tejido estático ofrece innegables beneficios de rendimiento, también implica algunas limitaciones. Si el proceso de tejido es sólo estático, el AOSD no se acomoda especialmente bien para el desarrollo rápido de aplicaciones. Si se toma como ejemplo los aspectos de *logging* o de pruebas, es necesario compilar, tejer, desplegar, ejecutar y depurar la aplicación. Además el contexto en tiempo de ejecución debe ser reproducido, y toda la información generada por los aspectos debe ser analizada. Si algún error ocurriese en tiempo de ejecución, la aplicación debería de ser modificada, re-compilada, re-tejida y re-ejecutada [Böllert99] [Eaddy07a].

El tejido dinámico proporciona una mayor flexibilidad en el desarrollo de software. Mientras se están desarrollando aplicaciones orientadas a aspectos, el mecanismo de adaptación dinámica es preferible porque facilita el desarrollo incremental, y además hace más fácil las tareas de depuración de la aplicación [Ortin02a] [Eaddy05].

En caso de usar un tejedor dinámico, los aspectos podrían ser añadidos en los puntos de ejecución exactos indicados por el usuario, y posteriormente eliminados. Además, la ejecución de la aplicación no se pararía si se desease modificar un aspecto, y la depuración sería más fácil ya que esta aproximación de tejido es no-invásiva. Cuándo la aplicación se fuese a desplegar, los aspectos que no necesitasen ser adaptados en tiempo de ejecución deberían ser tejidos estáticamente por razones de rendimiento.

1.1 Antecedentes. Necesidad De los Sistemas Mixtos de Configuración

Algunas investigaciones pioneras que sugirieron las necesidades de usar sistemas mixtos (dinámicos y estáticos) para adaptar las aplicaciones, no trabajaban con aspectos, eran trabajos sobre implementaciones de **ORB** –*object request broker*- donde el número de parámetros a configurar es muy alto.

El proyecto ZEN [Klefstad02], realizado por el grupo DOC [DOC08], es un **ORB** basado en Java, y que se aplica sobre sistemas distribuidos, en tiempo real y embebidos –*distributed, real-time and embedded*- (DRE) principalmente. En este proyecto se resalta la importancia de utilizar configuraciones estáticas y dinámicas para configurar algunos de los múltiples parámetros de un ORB. Por ejemplo, la carencia de información a priori de la carga de trabajo que soporta un servidor hace imposible

configurar este parámetro estáticamente. En cambio existen entornos embebidos donde los recursos disponibles están muy limitados, por lo tanto la sobrecarga que implica el dinamismo no puede ser absorbida por el entorno, o peor aún no se proporciona soporte para utilizar configuraciones dinámicas.

En investigaciones posteriores de este grupo [Gorappa05] se han utilizado componentes y aspectos tejidos estáticamente para la configuración de RTZen (una versión posterior de ZEN). Los componentes son usados como módulos software que encapsulan funcionalidades o características del núcleo de CORBA en tiempo de ejecución. Esos componentes son clases o paquetes que pueden integrarse en cualquier momento en el ORB. Este diseño mediante componentes permite tener varias implementaciones y varias versiones de una característica, y también permite extender fácilmente el framework de RTZen. Los aspectos son usados para la implementación de algunas funcionalidades (incumbencias) que se encuentran desperdigadas por todo el núcleo del ORB, como la gestión de memoria en tiempo real. Usando la AOP se consigue maximizar la modularidad del ORB.

Aunque los trabajos desarrollados por este grupo no son sistemas AOP, y sus esfuerzos por el momento se han centrado en la parte estática, se debe reconocer que sus ideas han influenciado otras investigaciones posteriores [Gilani04b] que si son sistemas AOP mixtos.

1.2 Motivación

El tejido estático proporciona aplicaciones adaptadas con buen rendimiento pero no permite adaptar las aplicaciones posteriormente durante su ejecución. Este tejido puede ser utilizado en muchas aplicaciones, pero existen algunas aplicaciones en las que sus características intrínsecas impiden su utilización.

Existen varios motivos que impiden el uso del tejido estático para la adaptación de todas las aplicaciones. Uno de ellos es la ausencia de información previa a la ejecución de la aplicación, como por ejemplo la carga de trabajo de un servidor, que impide a los desarrolladores la creación de lo aspectos necesarios [Matthijs97]. Además, puede existir la necesidad de reparar fallos o actualizar el sistema durante la ejecución [Gilani07b]. Esto se podría resolver parando la aplicación, y aplicándole los aspectos estáticos necesarios, para posteriormente volver a arrancarla, pero existen aplicaciones que no pueden ser paradas [Popovici03], como aquellas que pueden poner en peligro vidas humanas o deben dar un servicio 24x7, o en las que el tiempo consumido parando, compilando y desplegando la aplicación [Cech06] hacen inaplicable esta técnica.

Otro motivo más para no usar el tejido estático puede ser la existencia de un número excesivo de escenarios para la ejecución de la aplicación que impide diseñarla inicialmente para adaptarse a todos ellos [Schröder06].

La solución podría ser tejer dinámicamente los aspectos durante la ejecución de la aplicación. De esta forma, se podrían adaptar las aplicaciones durante la ejecución de la aplicación consiguiendo una mayor flexibilidad. Pero esto genera una penalización considerable en el rendimiento [Haupt04]. Otro inconveniente de los sistemas de tejido dinámicos es que en algunos entornos con recursos reducidos, como sistemas embebidos o dispositivos móviles, la sobrecarga de recursos que implica el dinamismo [Schöder06] [Popovici01], o la inexistencia de funcionalidades que lo soporte

[Klefstad02] pueden hacer inaplicable esta solución. Otra limitación de los tejedores dinámicos existentes es que disponen de un conjunto de **puntos de unión** –*join point*- [Kizcales97] más reducido que los estáticos, porque la adaptación en tiempo de ejecución es más complicada de implementar [Blackstock04].

El principal beneficio del tejido estático es el rendimiento en tiempo de ejecución. Como la combinación de los componentes y los aspectos se realiza previamente a la ejecución de la aplicación, implica un bajo coste en el rendimiento en comparación a los entornos tradicionales de desarrollo orientados a objetos [Böllert99] [Haupt04]. En contraste, las herramientas de tejido dinámico tienen como principal ventaja su flexibilidad para adaptarse a los cambios, pero esto implica una penalización en el rendimiento [Haupt04].

Debido a esta situación donde tanto los sistemas de tejido estático como los de tejido dinámico tienen puntos débiles, investigaciones previas han identificado la conveniencia de integrar en el mismo entorno de desarrollo el tejido estático y dinámico [Blackstock04] [Böllert99] [Gilani07b]. De esta forma, los beneficios de ambos tipos de tejido se podrían obtener en el proceso de desarrollo del software. Esta aproximación puede considerarse el resultado de aplicar la SoC a la incumbencia del momento de tejido (dinamismo) sobre AOSD. Este proceso se debería realizar transparentemente, reduciendo el impacto de cambiar el momento de tejido de las incumbencias en la aplicación.

Otro punto conflictivo en el desarrollo de estos sistemas de tejido –tanto estático como dinámico- es su dependencia del lenguaje. Algunos sistemas sólo permiten escribir aplicaciones o aspectos en un lenguaje, como los sistemas AspectJ [Kiczales01], JBoss AOP [JBossAOP08] o PROSE [Nicoara05] quiénes sólo pueden ser usados con Java. Otros necesitan añadir meta-información como anotaciones o atributos junto con el código de los aspectos que van adaptar la aplicación, o acceder al código fuente de la aplicación para cambiarla con objeto de adaptarse a los requisitos del tejedor como Gripper-LOOM.Net [Schult08], Rapier-LOOM.Net [Schult03] o Wicca [Eaddy07c]. Otros utilizan una extensión del lenguaje, como AspectC++ [Aspectc08] que es una extensión de C++, o JAsCo [JAsCo08] y AspectJ que son ampliaciones de Java, para la definición de los aspectos produciendo una dependencia entre el aspecto y el sistema. Todo ello impide hacer independientes las aplicaciones y los aspectos del sistema de tejido.

Por otro lado existen algunos sistemas como PROSE [Nicoara05], que modifican la plataforma sobre la que se implementa el sistema, o utilizan librerías no estándares de ella como AOPEngine.Net [Frei04] o Wicca [Eaddy07c]. Esto hace dependiente el sistema de la plataforma, lo cual imposibilita aplicar el sistema en otras plataformas basadas en el mismo estándar.

Debido a los problemas y los beneficios que presentan tanto los sistemas que son solamente estáticos como los sistemas que son solamente dinámicos, consideramos que es necesario un sistema que permita tejer aspectos estática y dinámicamente. Con este sistema los desarrolladores podrían desarrollar aspectos los cuales se podrían inyectar independientemente del momento de tejido, obteniendo la aplicación óptima dependiendo de la información conocida por los desarrolladores en cada momento. Esto mejorará el equilibrio entre el rendimiento en tiempo de ejecución y la flexibilidad para la adaptación dinámica de las aplicaciones. Este sistema permitirá a los desarrolladores utilizar tanta información a priori como sea posible en cada momento, de esta forma se

conseguiría el objetivo de evitar dinamismo cuando sea posible y ellos quieran [Schröder06].

Otro beneficio del uso de este sistema, sería en el desarrollo de software, los desarrolladores podrían utilizar tejedores dinámicos no invásivos para el desarrollo ágil [Vinuesa04] [Eaddy05] cuando fuese necesario. Una vez que la aplicación desarrollada fuese probada, se podría aplicar tejido estático donde fuese posible para obtener un mejor rendimiento en tiempo de ejecución.

También una característica clave en el sistema planteado es la homogeneidad del tipo de tejido, estático y dinámico. Los aspectos deben comportarse de la misma forma tanto si son tejidos estática como dinámicamente [Gilani04b]. Además de este cambio en el tejido debe ser transparente para el desarrollador.

Además, este sistema debe ser independiente del lenguaje, así se podrá usar cualquier lenguaje para crear los aspectos y las aplicaciones de forma que sean totalmente independientes. Además de ello, el sistema debe ser independiente de la plataforma que se utilice para poder utilizarlo en cualquier plataforma que cumpla las mismas especificaciones.

1.3 Finalidad del Proyecto

En este trabajo de investigación se ha diseñado e implementado una plataforma orientada a aspectos llamada DSAW (*Dynamic and Static Aspect Weaver*). Esta plataforma soporta tejido estático y dinámico de forma homogénea. DSAW ofrece el mejor rendimiento proporcionado por el tejido estático, y el desarrollo ágil e interactivo al usar tejido dinámico de una forma transparente. Además, DSAW es independiente del lenguaje y de la plataforma, y posee también un conjunto de puntos de enlace más amplio que otros sistemas de tejido dinámicos.

Dependiendo de la información conocida en cada momento, el desarrollador diseñará, implementará e inyectará los aspectos. De esta forma, este sistema proporciona un equilibrio óptimo entre rendimiento y flexibilidad, reduciendo el consumo de recursos necesario.

DSAW es una mejora de un sistema AOSD de tejido dinámico llamado Ready AOP (*Really Dynamic AOP*) [Vinuesa07] [Vinuesa04]. Ready AOP es un sistema independiente del lenguaje y de la plataforma, que ofrece una separación real de aspectos en tiempo de ejecución, a diferencia de otros sistemas dinámicos donde los aspectos deben ser definidos previamente a la ejecución de la aplicación como Rapier-LOOM.Net [Schult03].

Este sistema permite tejer los aspectos estática y dinámicamente de forma homogénea, independientemente del lenguaje de desarrollo y de la plataforma. Se puede usar cualquier lenguaje de la plataforma .Net para la realización de las aplicaciones o los aspectos, a diferencia de otros sistemas como AspectJ [Kiczales01] que sólo pueden utilizar el lenguaje Java. Además los lenguajes proporcionados por esta plataforma son de propósito general y pueden aplicarse a un gran ámbito de aplicaciones.

Además, el código fuente de las aplicaciones no es necesario para su adaptación ya que se va a trabajar a nivel de código intermedio. Esto proporciona independencia

del lenguaje, permitiendo adaptar cualquier aplicación implementada en cualquier lenguaje que sea soportado por la máquina virtual para la construcción de las aplicaciones o los aspectos. Además, el sistema no impone ninguna restricción a las aplicaciones para poder ser adaptadas, por lo que se pueden adaptar hasta aplicaciones realizadas por terceros de las que no se dispone de sus fuentes [Popovici03]. En otros sistemas AOSD, como Gripper-LOOM.Net [Schult08], Rapier-LOOM.Net [Schult03] o Wicca [Eaddy07c], es necesario acceder al código fuente para indicar qué parte de la aplicación se va *"aspectizar"* mediante anotaciones o meta-información, e incluso puede ser necesario cambiar el código fuente de la aplicación para adaptarse a las restricciones impuestas por el sistema de tejido.

Tampoco se impone ningún tipo de restricción al código fuente de los aspectos que van adaptar las aplicaciones. La implementación de una utilidad para la generación de ficheros ejecutables que actúen como un *stub* o **despachador** *–dispatcher-* a una librería (generada con cualquier lenguaje soportado por el sistema) donde se encuentran los aspectos a inyectar, evita imponer restricciones en signaturas, funciones a implementar, etc. Usando esta utilidad se pueden crear *stubs* sin conocer el código fuente de los aspectos, operando a nivel de código intermedio.

El sistema se ha diseñado sobre la plataforma estándar .Net [ECMA08], la cual no ha sido modificada. El sistema puede ser implementado sobre cualquier plataforma que cumpla ese estándar. Es independiente de la plataforma, a diferencia de otros sistemas como AOPEngine.Net [Frei04] o PROSE [Nicoara05].

2 Objetivos del Trabajo

La clasificación más común de las herramientas AOSD está basada en el momento de tejido de los aspectos. Las herramientas de tejido estático lo realizan previamente a la ejecución de la aplicación. El sistema más conocido de tejido estático es AspectJ [Kiczales01]. En cambio, las herramientas dinámicas realizan el tejido en tiempo de ejecución. Hay varias herramientas que ofrecen algún tipo de adaptación dinámica de aspectos, siendo PROSE [Popovici01] [Nicoara05] y JBoss AOP [JBossAOP08] dos ejemplos bien conocidos de este tipo de tejido.

En el capítulo anterior se han comentado las ventajas e inconvenientes de cada tipo de tejido, y se han enumerado las razones que hacen posible que un sistema de tejido mixto o híbrido solucione estos inconvenientes. Por lo tanto, el primer objetivo que se persigue con la realización de este trabajo es la realización de una plataforma de tejido mixta.

Objetivo 1. Realización de un sistema de tejido no invásivo que permita realizar tejido estático y dinámico de aspectos. Así los desarrolladores podrán usar el tejido estático dónde sea posible, y el tejido dinámico cuándo sea necesario [Gilani07b] [Böllert99] [Schröder06]. Como consecuencia de esto, se facilitará el equilibrio entre rendimiento y flexibilidad. Los desarrolladores podrán cambiar el momento de inyección de un aspecto para ganar rendimiento o flexibilidad fácil y rápidamente.

Objetivo 2. Combinación de adaptación dinámica y estática. Las aplicaciones no sólo podrán ser *"aspectizadas"* dinámica o estáticamente, también se podrán combinar los dos tipos de tejido en la misma aplicación.

Objetivo 3. Ambos tipos de tejido, estático y dinámico, deben de ser soportados por el sistema de un modo homogéneo. Una herramienta que soporte ambas técnicas de tejido debe permitir definir aspectos independientemente de cuando vayan a ser tejidos. Los aspectos se tienen que poder inyectar estática o dinámicamente sin ninguna modificación sobre su código.

Objetivo 4. Separación de la incumbencia del momento de tejido (dinamismo). El sistema propuesto se beneficiará de disponer de funcionalidades de tejido estático y dinámico en el mismo sistema. El resultado será la separación de la incumbencia del momento de tejido del aspecto en el proceso AOSD [Gilani07b]. Los desarrolladores podrán posponer la decisión de si un aspecto es inyectado estática o dinámicamente a las últimas etapas del despliegue.

Objetivo 5. Un sistema de tejido dinámico totalmente adaptable. Muchos sistemas dinámicos AOSD no son totalmente adaptables en tiempo de ejecución. La mayoría de ellos requieren la especificación estática de qué aspectos van a ser tejidos en tiempo de ejecución. Otros ofrecen adición dinámica de nuevos aspectos pero no soportan el borrado dinámico. Mediante el sistema propuesto los aspectos no necesitarán ser especificados antes de ejecutar la aplicación. Se podrán crear nuevos aspectos y añadirlos en tiempo de ejecución, y borrar aspectos dinámicos tejidos previamente.

Objetivo 6. Proporcionar un conjunto rico de puntos de enlace. El conjunto de puntos de enlaces que ofrecen muchos sistemas dinámicos es significativamente más pequeño que en sistemas estáticos [Blackstock04] [Vinuesa04]. El sistema propuesto proporcionará un conjunto amplio de puntos de enlace, esto permitirá adaptar cualquier punto de la aplicación. Se tomará como modelo AspectJ [Kiczales01], ya que este sistema tiene el conjunto más amplio de puntos de enlace y se ha tomado ya como referencia en otras implementaciones. Como el sistema que se va a realizar es homogéneo, los mismos puntos de enlaces estarán disponibles tanto para tejido estático como para tejido dinámico, y deberán ser usados de la misma forma.

Objetivo 7. Independencia del lenguaje. Cualquier aplicación o aspecto escrito con un lenguaje soportado por la plataforma sobre la cual se desarrolle el sistema podrá ser usado por este sistema.

Aparte del dinamismo, un sistema orientado a aspectos se beneficiaría de ser independiente del lenguaje. Esta característica permitiría la adaptación de aplicaciones por aspectos desarrollados en diferentes lenguajes de programación. De esta forma, se promueve la reutilización de aspectos y componentes. Además de ello, si la característica de neutralidad de plataforma también se alcanzase, el sistema orientado a aspectos podría ser ejecutado sobre cualquier plataforma basada en el mismo estándar.

Objetivo 8. Independencia de la plataforma. El sistema se debe diseñar sobre una plataforma estándar no modificada. Se debe poder utilizar sobre cualquier otra plataforma que cumpla el estándar y no se haya modificado.

Objetivo 9. Adaptación de código binario. Una importante característica de los sistemas AOSD es la adaptación de módulos binarios de los cuales no es requerido su código fuente para adaptar la aplicación. El tejido del sistema propuesto se realizará en código intermedio de la máquina virtual. Esto posibilitará la adaptación de aplicaciones de terceros de las que no se disponga de su código fuente [Popovici03].

Objetivo 10. No debe existir ningún acoplamiento entre las aplicaciones y los aspectos que las adaptan. El sistema propuesto no necesitará modificar el código fuente de la aplicación original para que ésta sea consciente de la existencia futura de aspectos estáticos o dinámicos. Se sigue la idea POJO del API de persistencia de Java [JSR-220.08].

Objetivo 11. Reutilización de los componentes y los aspectos. La técnica de tejido no-invásiva y la no necesidad de código fuente proporcionarán un acoplamiento nulo entre los componentes y los aspectos facilitando su reutilización.

Objetivo 12. Adaptación un aspecto con otro aspecto. Los aspectos se pueden considerar componentes en DSAW. Por lo que también se podrán adaptar.

2.1 Posibles Ámbitos de Aplicación

El sistema planteado en este trabajo puede aplicarse en una gran variedad de proyectos, desde grandes aplicaciones que no se pueden parar, hasta aplicaciones embebidas o empotradas con limitadas capacidades. Otro ámbito de aplicación muy interesante sería para facilitar el desarrollo de software de forma rápida y eficiente.

Aplicaciones que no se pueden parar, como aplicaciones 24x7, servidores de aplicaciones, sistemas aerospaciales, sistemas críticos para las personas o sistemas de *middleware* [Matthijs97] [Popovici03]. El coste de parar estas aplicaciones es muy elevado, e incluso la parada o interrupción de su servicio puede poner en peligro vidas humanas. Aplicando una plataforma como la propuesta en este trabajo, se podrían considerar muchos requerimientos de las aplicaciones como aspectos estáticos. Y para aquellos requerimientos que sólo se van a plantear en tiempo de ejecución, como la carga de trabajo de un servidor de aplicaciones, o fallos en tiempo de ejecución que necesiten ser reparados de forma inmediata, se podrían utilizar aspectos inyectados dinámicamente. Si el problema que provocó la necesidad de inyectar ese aspecto dinámico se resolviese, se podría eliminar este aspecto.

Aplicaciones embebidas o empotradas, como sistemas operativos, software de teléfonos móviles, GPS, electrodomésticos, etc. Tienen como características más comunes contemplar un conjunto creciente de funcionalidades pero con grandes restricciones hardware. Esto exige un adecuado equilibrio entre estas funcionalidades y las capacidades hardware y software proporcionadas por los dispositivos [Schöder06]. Sus funcionalidades van a depender del entorno donde se ejecuten en cada momento. Sería muy beneficioso configurar las características más importantes de forma estática, y añadir o eliminar otras características dinámicas dependiendo del entorno de ejecución dónde operé el dispositivo en ese momento.

El desarrollo de aplicaciones orientada a aspectos se puede beneficiar de un sistema mixto de tejido. Así se puede utilizar tejido dinámico mientras se desarrolla la aplicación, ya que facilita el tejido incremental y hace más fácil la depuración del código [Ortin02a]. Se podría utilizar tejido dinámico cuando el sistema está siendo probado siguiendo un esquema de depuración *edit-and-continue* [Eaddy05]. Cuando se finalice la aplicación, los aspectos que no necesiten adaptar la aplicación en tiempo de ejecución se deberían de tejer estáticamente por razones de rendimiento.

Mediante los aspectos inyectados dinámicamente el desarrollador podría reducir el tiempo necesario para ver los cambios en ejecución *–zero turn-around time* (ZTAT). Así se reduciría el tiempo necesario para aplicar los cambios sobre la ejecución de la aplicación, evitando tareas como el preprocesado de los ficheros de código, copiado de ficheros de configuración específicos del entorno, descarga de dependencias, compilado del código fuente, creación de la distribución, etc.

El desarrollador también se beneficiaría de poder preservar el estado de la aplicación que está ejecutando en ese momento. De esta forma cuando fuese necesario depurar la aplicación, no sería necesario continuamente empezar todo el proceso hasta alcanzar el estado sobre el cual se desea hacer las pruebas.

Otro problema que se solucionaría en el desarrollo de software, es cuando se utilizan en las aplicaciones técnicas de cacheado para tener cargado en memoria instancias que son relativamente estáticas. Si éste cacheado se hace siguiendo una política de evaluación **perezosa** *–lazy-,* donde sólo se cachean las instancias que el usuario va solicitando, provoca dos problemas: el tiempo de desarrollo se incrementa, y el usuario final cuando comience a usar la aplicación puede suponer una velocidad de la aplicación mucho menor de la real, debido a que mientras la caché no almacene bastante información su función no está justificada. Para evitar esto se puede aplicar una política de cacheado **devoradora** *–eager-,* donde se cachean todas las instancias que se relacionan con una instancia que ha solicitado el usuario, pero los resultados pueden dar

lugar a un ciclo de desarrollo incluso mayor que aumente aún más el tiempo de desarrollo.

Aplicando el sistema planteado en esta investigación, estos problemas se evitarían, ya que para depurar una aplicación se usarían aspectos inyectados dinámicamente, evitando así el re-despliegue de la aplicación, alcanzar el estado necesario para las pruebas, y el posible cacheado de instancias. Posteriormente, este aspecto se podría inyectar de forma estática para reducir el tiempo de ejecución de la aplicación, en caso de que fuese posible.

Un proyecto que busca reducir el ZTAT es la herramienta JavaRebel [JavaRebel08]. Es un *plugin* sobre la máquina JVM, cuyo objetivo es hacer cambios a las clases de Java en el momento *–on the fly-* evitando el tiempo de re-desplegar y re-iniciar una aplicación.

3 Estado Actual de los Conocimientos Científicos-Técnicos

Este capítulo se divide en tres apartados. Inicialmente se van a comentar varias aproximaciones que se han diseñado siguiendo el principio de la Separación de Incumbencias (SoC). Después se detallará más profundamente una de estas aproximaciones, el Desarrollo de Software Orientado a Aspectos (AOSD). Por último, se comentará el sistema de referencia de tejido de aspectos AspectJ, y alguno de los sistemas de tejido mixtos de aspectos existentes actualmente.

3.1 Separación de Incumbencias (Separation of Concerns)

El principio de la **separación de incumbencias**[2] (SoC) –*Separation of Concern*s- [Parnas72] [Dijkstra76] [Hürsh95] ha sido utilizado para gestionar la complejidad del desarrollo software. Facilita la separación de las funcionalidades principales de una aplicación de otras funcionalidades secundarias de propósito específico. Estas funcionalidades secundarias son ortogonales a las funcionalidades principales, y son las encargadas de realizar tareas como autenticación, gestión de memoria, *logging*, rendimiento, persistencia, etc. La aplicación final será la suma de las funcionalidades principales junto con las funcionalidades de propósito específico.

La SoC es el proceso de dividir la aplicación que se va a implementar en las distintas características independientes entre sí cuya funcionalidad se solape lo mínimo posible. Los principales beneficios [Vinuesa07] de esta aproximación son:

- Un nivel de abstracción más alto, debido a que el desarrollador se puede centrar en incumbencias concretas de forma aislada.
- Una mayor facilidad a la hora de entender la funcionalidad de la aplicación. El código que implementa la funcionalidad 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 de implementación.

Tradicionalmente se ha intentado diseñar las aplicaciones informáticas a través del uso de técnicas como la modularización, el encapsulamiento y el ocultamiento de la

[2] También conocida como “Separación de Competencias”.

información. Pero estas técnicas por sí solas no permiten lograr plenamente los objetivos deseados, ya que existen incumbencias ortogonales entre sí, y se tendrían que diseñar e implementar de forma totalmente independiente evitando el acoplamiento con otras incumbencias. Esto no es posible ya que muchas veces una incumbencia se encuentra desperdigada y entremezclada entre varias incumbencias distintas a lo largo de una aplicación, y no puede ser modularizada.

La SoC es un importante principio de diseño, y está siendo aplicada a otras disciplinas como planificación urbana, arquitectura o diseño de información entre otras. La meta es diseñar sistemas cuyas funciones puedan ser optimizadas independientemente de otras. Así un fallo sobre una función no causará que otras funciones fallen. Esto hace más fácil comprender, diseñar y gestionar sistemas complejos interdependientes. Ejemplos de diseños realizados siguiendo este principio son los pasillos que conectan habitaciones en lugar de tener puertas directas, o las instalaciones eléctricas de los hogares, donde los electrodomésticos están en un circuito y las luces en otro.

Durante los últimos años varias aproximaciones se han realizado para intentar conseguir una SoC óptima, que permita desarrollar sistemas informáticos beneficiándose de todas las ventajas de este principio. A continuación se van a comentar algunas de las más conocidas.

3.1.1 Filtros de Composición

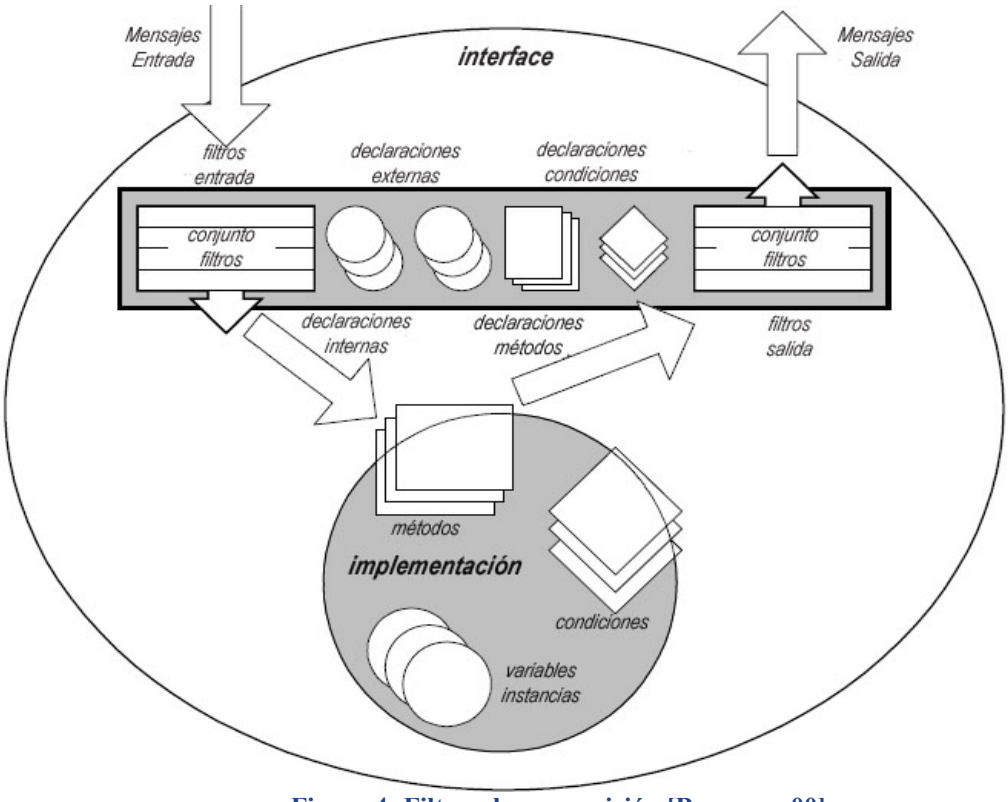


Figura 4: Filtros de composición [Bergmans00].

Un objeto de tipo **filtro de composición** –*composition filters*- [Aksit92] consta de dos partes: una parte interfaz y una parte implementación. La parte de interfaz es la encargada de procesar los mensajes de entrada y salida. Consiste de uno o más filtros de

entrada, uno o más filtros de salida, objetos opcionales externos e internos, y cabeceras de declaración de métodos.

Los filtros son controlados por condiciones. Los nombres de los filtros, las cabeceras de los métodos, y los nombres de las condiciones pueden ser visibles por los clientes del objeto, pero sus implementaciones se encuentran definidas en el interfaz de implementación y son invisibles.

Si un mensaje de entrada pasa los filtros, puede ser delegado a un objeto interno, a un objeto externo, o a un método. Todos los mensajes que son generados por la ejecución de métodos dentro del objeto y son enviados a objetos externos fuera del propio objeto pasan a través de los filtros de salida.

La parte de implementación contiene definiciones de métodos, declaraciones de variables de instancia, definiciones de condiciones y una operación opcional de inicialización. Esta parte está totalmente encapsulada dentro del objeto.

Los filtros aportan al desarrollador la oportunidad de poder atrapar el envío y la 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 implementación 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 son ortogonales entre sí [Pryor02].

3.1.2 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 [IBM2000]. Los principales objetivos de estos sistemas son [Ortin02b] [Ortin03b]:

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 se ha 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] [OOSPLA99]. Un ejemplo es *Hyperspaces* [Ossher99], un sistema de separación multidimensional de incumbencias independiente del lenguaje, creado por IBM, así como la herramienta Hyper/J [HyperJ08] que da soporte a *Hyperspaces* en el lenguaje de programación Java.

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.

3.1.3 Programación Orientada a Sujetos

Se han identificado tres problemas en los proyectos realizados usando Programación Orientada a Objetos (OOP) [Clarke99]:

- Los modelos son a menudo grandes y monolíticos.
- Los diseños son difíciles de reutilizar.
- Falta de alineación entre los requisitos y el código, con el diseño que se encuentra atrapado en el medio.

La **Programación Orientada a Sujetos** –*Subject-Oriented Programming*-(SOP) [Harrison93] es un método para la composición de aplicaciones software. Permite la construcción de sistemas orientados a objetos como una composición de sujetos. Extiende los sistemas componiéndolos con nuevos sujetos, e integrando otros sistemas los cuales se compondrán usando otros sujetos adaptadores en caso de que sean necesarios.

En la SOP, un sujeto es una colección de clases o fragmentos de clases cuya jerarquía de clases modela su dominio desde un punto de vista particular. Un sujeto puede ser una aplicación completa en si misma, o puede ser un fragmento incompleto que debe ser compuesto con otros sujetos para producir una aplicación completa. La composición de sujetos combina jerarquías de clases para producir nuevos sujetos que incorporan funcionalidades de sujetos existentes.

La flexibilidad de la composición de sujetos introduce nuevas oportunidades para desarrollar y modularizar programas orientados a objetos. La SOP implica dividir un sistema en sujetos, y escribir reglas para componerlos correctamente. Complementa la OOP, resolviendo problemas que ocurren cuando la OOP es usada para desarrollar sistemas grandes, o conjuntos de aplicaciones integradas o interconectadas.

3.1.4 Modelado de Roles

Los roles y los modelos de roles son mecanismos de abstracción y descomposición. Las clases estipulan las capacidades de objetos individuales, mientras la noción de rol se centra sobre la posición y las responsabilidades de un elemento dentro de todas las partes de un sistema o subsistema.

Un modelo de roles identifica una estructura arquetípica de elementos (objetos), y la describe como una estructura equivalente de roles. Los modelos de roles capturan como los objetos interaccionan entre sí en colaboraciones.

En el modelo conceptual de objetos y roles inicialmente propuesto [Kristensen96], el objeto al cual un rol está destinado es el objeto intrínseco, tiene miembros intrínsicos (datos y métodos). Los roles añaden miembros extrínsecos (datos y métodos), y proporcionan perspectivas que pueden ser usados por otros objetos como un modo selectivo de conocer y acceder el objeto.

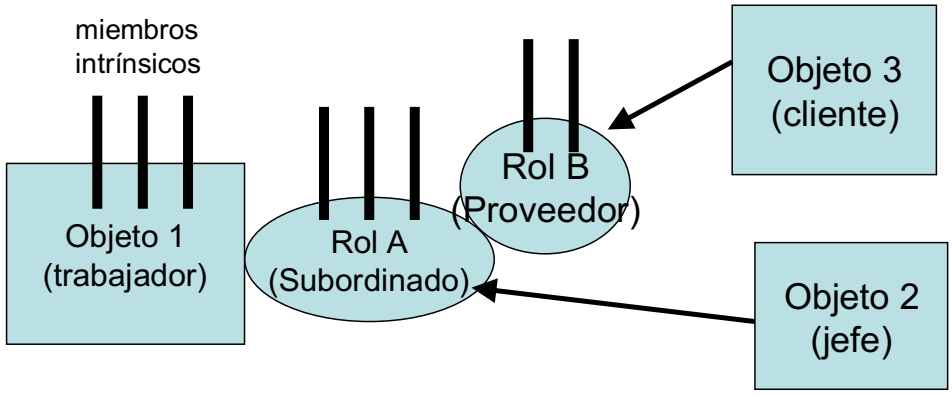


Figura 5: Objeto y sus roles: miembros intrínsicos y extrínsecos [Kristensen96b].

Los roles proporcionan las siguientes propiedades:

- Abstracción. Los roles pueden ser clasificados en jerarquías.
- Agregación/Composición. Los roles pueden ser compuestos de otros roles, con visibilidad variable.

- Dependencia. Un rol puede no existir sin el objeto.
- Dinamismo. Un rol puede ser añadido o borrado durante el ciclo de vida de un objeto.
- Identidad. El rol y el objeto tienen la misma identidad.
- Jerarquía. Un rol para una clase es también un rol para cualquier subclase de la clase. Y un super-rol es un rol de una clase, si un sub-rol de este rol es un rol de esa clase.
- Localidad. Un rol sólo tiene un significado en un modelo de roles.
- Multiplicidad. Varias instancias del mismo rol pueden existir para el mismo objeto al mismo tiempo.
- Visibilidad. El acceso al objeto está restringido por rol.

3.1.5 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 la estructura de los objetos [Lieberherr96]. De este modo, la programación adaptable eleva el nivel de la programación orientada a objetos (OOP). El principal caso práctico de la programación adaptable es el proyecto Demeter [Demeter08], el cual se basa en el desarrollo de software mediante un método adaptable.

El software adaptable [Lieberherr96] es una extensión del software orientado a objetos. Se basa en que las relaciones entre funciones y datos son más flexibles, las funciones y los datos están ligeramente unidos. Adaptable significa que el software de forma heurística se cambia así mismo para manejar una clase de cambios de requisitos relacionados que deben modificar la estructura del objeto. Un programa adaptable permite expresar la intención sin dar detalles acerca de la estructura del objeto.

Los beneficios del software adaptable son:

- Programas más cortos y un nivel más alto de abstracción.
- Bibliotecas de software reutilizable.
- Posibilidad para planificar los cambios mediante aprendizaje.
- Posibilidad para construir tecnología de objetos familiar.
- Evita el riesgo.
- Mínima dependencia de un lenguaje orientado a objetos particular.

El software adaptable es una evolución natural de la OOP desde que cada programa orientado a objetos es un programa adaptable. A pesar de que la OOP proporciona mejores capacidades para adaptar programas que la programación tradicional, todavía es una disciplina rígida y dura para evolucionar aplicaciones.

3.1.6 Programación Orientada a Aspectos

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**–*Aspect Oriented Programming*– (AOP) surge en 1997 [Kiczales97], y desde entonces ha suscitado mucho interés por las posibilidades que ofrece. En la AOP 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 6 se puede ver el esquema de funcionamiento de una aplicación orientada a aspectos.

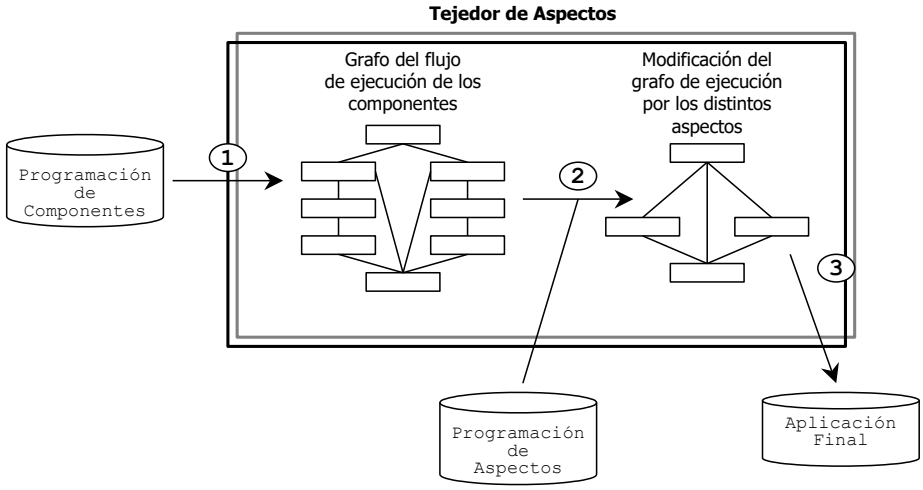


Figura 6: Esquema de la Programación Orientada a Aspectos [Vinuesa07].

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) [Ortin07].
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).

Se está realizando muchas investigaciones sobre esta aproximación, y han aparecido varios sistemas que la soportan, siendo el más importante hasta el momento

AspectJ [AspectJ08], el cual ha sido utilizado en grandes proyectos con buenos resultados. También se ha incorporado a varios servidores de aplicaciones como Jboss AOP [JBossAOP08], o al framework de desarrollo Spring [SpringAOP08].

La AOP está siendo usada también para el **Software Autónomo** *–Autonomic Software-* [Autonomic08], cuyas aplicaciones deben auto configurarse, repararse, optimizarse y protegerse a sí mismas. Como ejemplos de investigaciones en este campo se puede resaltar: la actualización dinámica de sistemas software [Cech06], el remplazado de código [Nicoara08] o la autocuración de sistemas [Griffith05].

3.2 Desarrollo de Software Orientado a Aspectos

El término **Programación Orientada a Aspectos** *–Aspect Oriented Programming–* (AOP - POA) es propuesto por Gregor Kiczales [Kiczales97] en 1997, aunque el equipo Demeter [Demeter08] había estado trabajando sobre este campo desde mucho antes. La definición más temprana de **aspecto** también la dio este grupo en 1995 [Demeter08]. 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–* (AOSD - DSOA), que es el término empleado actualmente, quedando la AOP como un subconjunto de la misma.

3.2.1 Definición de Aspecto

Puesto que la AOP 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.

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 7 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 [Laddad03].

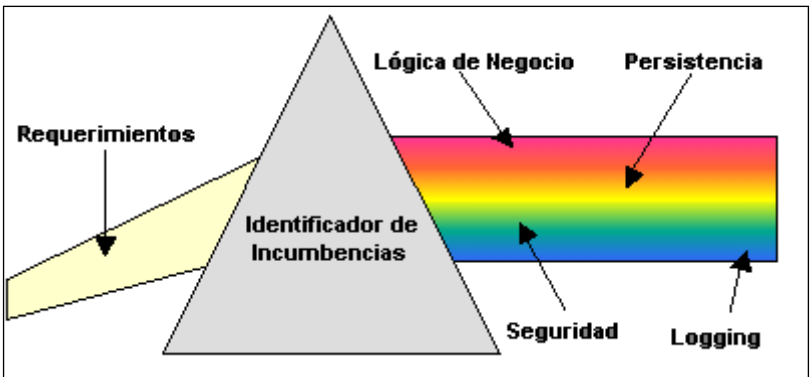


Figura 7: Requerimientos del sistema según AOP [Laddad03].

La estructura de un programa orientado a aspectos se muestra en la Figura 8. 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, **registro** *–logging-*, gestión de memoria, etc.

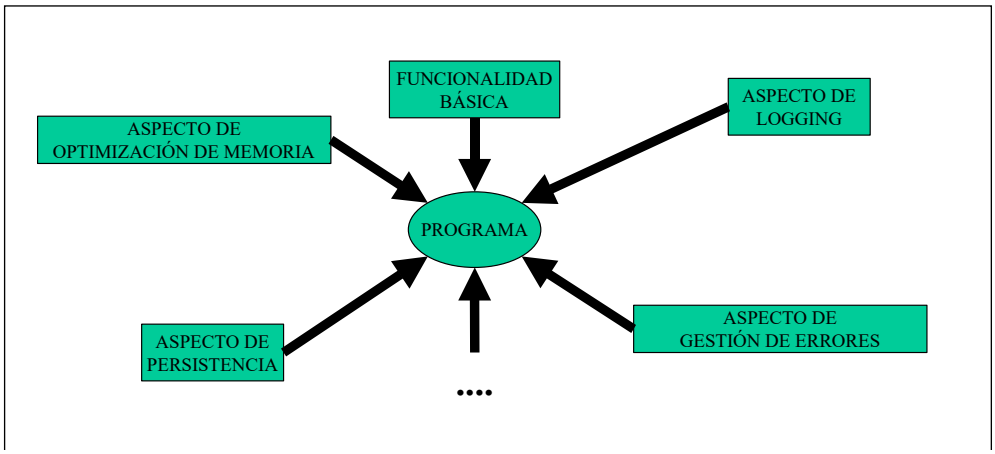


Figura 8: Estructura de un programa orientado a aspectos [Vinuesa07].

En la parte izquierda de la Figura 9 se puede ver la forma que tiene el código de un programa siguiendo una metodología tradicional. Se puede observar que el código está entremezclado siendo difícil de entender, depurar y modificar. En cambio siguiendo la AOP, 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.

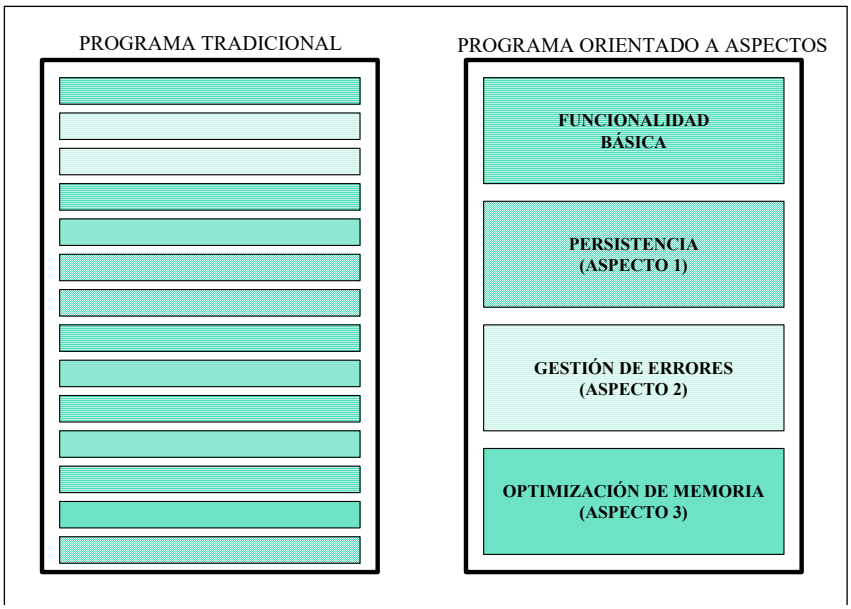


Figura 9: Código programa tradicional vs. AOP [Vinuesa07].

3.2.2 Cómo Funciona la AOP

Básicamente en el funcionamiento de la AOP se pueden identificar tres pasos (Figura 10):

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.*

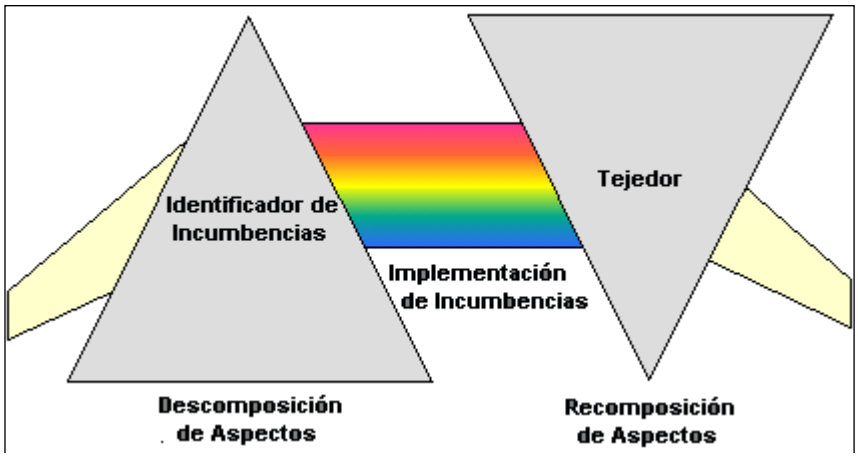


Figura 10: Pasos de la AOP [Laddad03]

Los lenguajes orientados a aspectos definen una nueva unidad de programación software, el **aspecto**, para encapsular las funcionalidades que están **diseminadas**

(desperdigadas) y **enmarañadas** (enredadas) –*scattered* 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** –*join point*.

Los puntos de enlace son un 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 inyectar, 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 [AspectJ08] 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 11, 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 AOP 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.

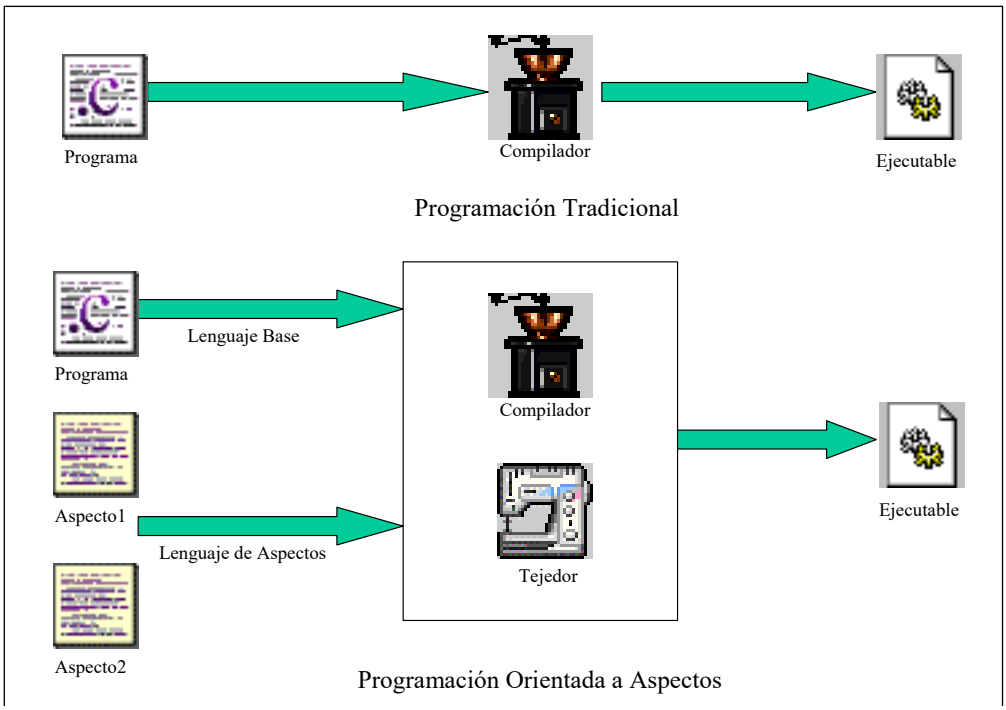


Figura 11: Implementación tradicional vs. AOP [Vinuesa07].

3.2.3 Terminología

Ya se han mencionado algunos de los términos que se emplean en la AOP, 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** *-join point-*: 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 AOP 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).
- **Punto de corte** *–pointcut-*: es un conjuntos 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.).

3.2.4 Tipos de AOP

3.2.4.1 Clasificaciones de AOP

Se han realizado varios trabajos clasificando los sistemas AOP. Una de las primeras clasificaciones analizó los mecanismos que utilizaba cada sistema específico para soportar un determinado tipo de **incumbencia cruzada** *–crosscutting concerns* [Masuhara03a]. Otros han clasificado los sistemas AOP dinámicos por conceptos como *momento del tejido* y *tipo de tejido* [Chitchyan04].

El momento del tejido [Chitchyan04] para sistemas dinámicos se divide en:

- Tiempo de carga. Las clases son modificadas cuando se cargan en la máquina virtual.
- Compilación JIT. La modificación de las clases toma lugar cuando el compilador JIT traduce el código intermedio a código nativo.
- *Proxies* dinámicos. Es una facilidad específica del lenguaje Java. Se usa para explotar la invocación reflectiva de *advices*.

El tipo de tejido es cómo el código de la aplicación se modifica, para posteriormente soportar la invocación en tiempo de ejecución de las funcionalidades de los *advices*:

- Inserción **total** *–total-* de **enganches** *–hooks*. Se realiza sobre toda la aplicación. Estos *hooks* son los responsables de saltar a la infraestructura AOP, que será la encargada de ejecutar los *advices*.
- Inserción **real** *–actual*. Sólo se insertan *hooks* sobre las localizaciones explícitamente especificadas de código.
- Inserción **completa** *–collected*. Evita la inserción de *hooks,* y en su lugar teje directamente la invocación al código de los *advices*.

Durante el congreso AOSD-Europe en 2005 se realizaron dos encuestas para clasificar los sistemas AOP Una de ellas analizo el *middleware* basado en aspectos [Loughran05], y la otra los lenguajes AOP y los modelos de ejecución[3].

Recientemente se ha realizado un análisis [Haupt06] de los parámetros que se deberían de tener en cuenta para clasificar los sistemas AOP dinámicos. Para ello se basaron en las investigaciones enumeradas anteriormente. A partir de estas investigaciones se realizó una clasificación agrupando los sistemas en familias -algunos sistemas AOP pueden ser clasificados en varias.

Las familias encontradas fueron:

[3] No disponible consultar [Haupt06].

- Compilador soporta estáticamente AOP. Los sistemas no soportan tejido dinámico, y se basan en un lenguaje de aspectos dedicado que es procesado por un compilador que soporta de forma estática AOP.
- Compilador soporta dinámicamente AOP totalmente a nivel de aplicación. Estos sistemas se basan en extensiones del lenguaje y dependen del compilador pero soportan tejido dinámico. El tejido dinámico es implementado totalmente a nivel de aplicación y no usan las facilidades del entorno de ejecución.
- Compilador soporta dinámicamente AOP con soporte del entorno. A diferencia de la familia anterior, estos sistemas emplean las facilidades del entorno de ejecución.
- Configuración dirigida dinámica AOP con soporte de entorno. La configuración puede ser realizada mediante ficheros XML o mediante anotaciones de meta-información.
- Framework dinámico para AOP con soporte del entorno. Las definiciones de los aspectos toman lugar proporcionando clases que extienden el framework de clases adecuadamente. Internamente existen mecanismos específicos proporcionados por el entorno de ejecución que son usados para implementar el comportamiento orientado a aspectos.
- Meta programación para AOP. Estos sistemas explotan las capacidades de meta programación del lenguaje de programación usado para soportar los aspectos [Redondo08]. Se dividen en tres subgrupos: modelo introspectivo basado en reflectividad estructural, modelo completo reflectivo soportando reflectividad del comportamiento, y entorno totalmente reflectivo que sólo permite operar a nivel "meta" para implementar AOP [Ortin04b].

Otra clasificación que se puede utilizar [Vinuesa07] es relativa al dominio de los lenguajes de aspectos, éstos pueden ser de propósito general o de propósito específico. Dentro de los propósitos generales se puede destacar AspectJ que puede ser utilizado con cualquier clase de aspecto. Mientras que los de propósito específico sólo soportan algunos tipos de aspectos, y proporcionan una abstracción mayor.

Para finalizar, la clasificación más utilizada para dividir los sistemas AOP existentes es en función del tipo de tejido, estático o dinámico. Esta clasificación debería modificarse para dar cabida a los sistemas de tejido mixtos, los cuales combinan tejido estático y dinámico.

3.2.4.2 Tejido Estático Vs. Dinámico

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

- Un pre-procesador que haga las sustituciones pertinentes en el código.
- Un post-procesador que modifique archivos binarios.
- Un compilador que soporte la *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 post-procesamiento 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.

Si se utiliza como criterio el *momento* en el que se realiza el tejido se distingue 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.

Este trabajo de investigación se enfoca hacia los sistemas mixtos de tejido, los cuales combinan características de tejido estático y dinámico. Estos sistemas pueden utilizarse para realizar tanto tejido estático como dinámico dependiendo de las necesidades de los desarrolladores.

3.2.4.2.1 Tejido Estático

El tejido estático consiste en la modificación en tiempo de compilación del código fuente o en una fase previa a la ejecución, insertando llamadas a las rutinas específicas de los aspectos. Los lugares donde estas llamadas se pueden insertar son los **puntos de enlace** *–join points* (definidos por cada sistema).

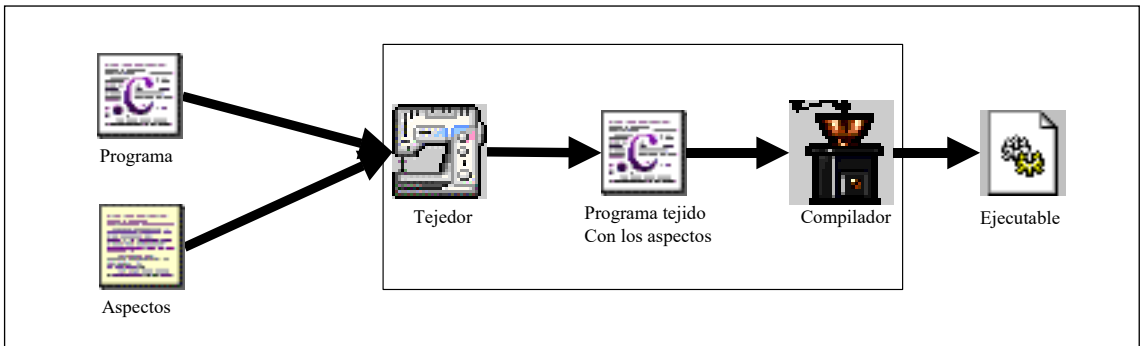


Figura 12: Tejido estático [Vinuesa07].

En la Figura 12 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 y las instrucciones que recibe el tejedor sobre *dónde y cómo* insertar el código de los aspectos en el código principal (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:

- Inexistencia o carencia de información antes de la ejecución de la aplicación. Los desarrolladores necesitan información para implementar las incumbencias mediante aspectos. Pero muchas veces esa información sólo está disponible durante la ejecución de la aplicación, por ejemplo la carga de trabajo de un servidor de aplicaciones [Gilani04a]. La adaptación de la aplicación por estos aspectos debe ser realizada dinámicamente en tiempo de ejecución.
- Imposibilidad de reparar fallos o realizar actualizaciones durante la 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 compilador y del tejedor), y debe ser re-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 [Popovici03].
- Excesivo número de escenarios de ejecución. En el mundo del software de sistemas embebidos (por ejemplo dispositivos móviles o PDAs) se podría considerar un sistema que contemple todos los requisitos de los posibles escenarios de uso. Pero esto no es realista, y además su desarrollo provocaría un consumo excesivo de recursos. Por ello, se deben realizar sistemas que proporcionen las funcionalidades necesarias mínimas y nada más. En tiempo de ejecución se debería poder añadir o eliminar funcionalidades dependiendo del escenario donde trabaje la aplicación en ese momento. [Schröder06]
- Incremento de costes. Si se añaden funcionalidades software, las cuales pueden ser utilizadas o no dependiendo del uso del sistema, se necesitarían más recursos para que operase el sistema y éste sería mucho más complejo. Los costes de fabricación se harían mayores, y existirían mayores probabilidades de que se produzcan fallos ante el aumento de la complejidad. [Schröder06]
- Aumento de los costes de depuración. La depuración de una aplicación que ya haya sido compilada, y ya se le hayan añadido los aspectos es una tarea compleja. El código que se debe analizar es el resultante del proceso de tejido, y es distinto del código original que no está entremezclado con él de los aspectos, el cual estará diseminado por toda la aplicación [Eaddy07a] [Vinuesa04]. Por otro lado, las herramientas de depuración a día de hoy no permiten analizar el código generado correctamente.

La principal ventaja que ofrece este tipo de tejido es evitar que el uso de la AOP 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 su dependencia del lenguaje; esto implica que el sistema sólo sirve para un lenguaje específico, no pudiendo crear distintos aspectos en distintos lenguajes. Por ejemplo, AspectJ [AspectJ08] sólo puede usarse con el lenguaje Java.

3.2.4.2.2 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 o en una fase previa a la ejecución de la aplicación. Si se quiere enriquecer la aplicación con un nuevo aspecto, o incluso eliminar uno de los aspectos actualmente tejidos, el programa debe ser recompilado y reiniciado de nuevo.

Aunque no todas las aplicaciones necesitan ser adaptadas mediante aspectos en tiempo de ejecución, hay aplicaciones específicas 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]. Técnicas parecidas 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-Deville03], y en distribución de incumbencias basada en el equilibrio de carga [Matthijs97]. Otro ejemplo de uso de la AOP de forma dinámica es lo que recientemente se ha llamado **software autónomo** –*autonomic software/computing*- software capaz de auto repararse, gestionarse, optimizarse o recuperarse [Autonomic08].

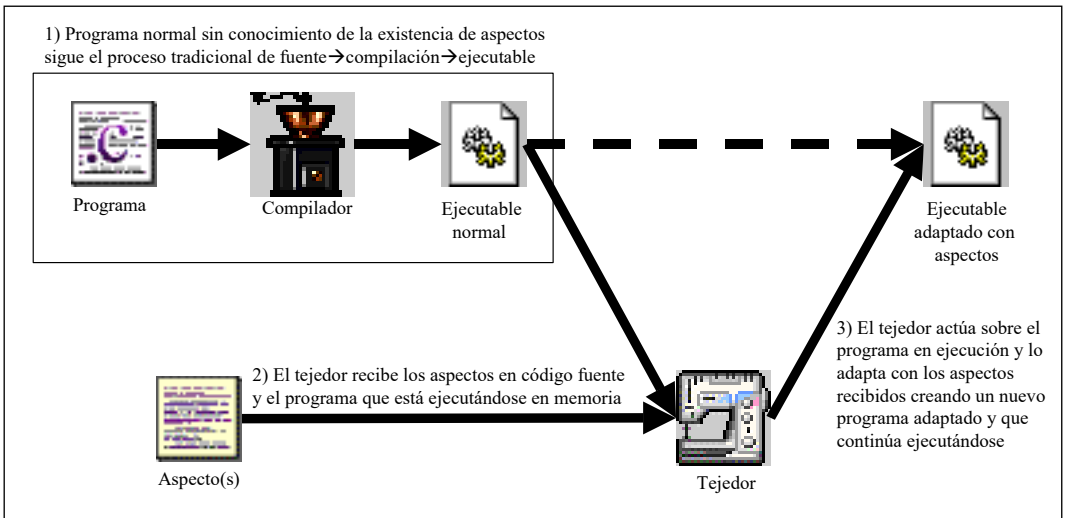


Figura 13: Tejido dinámico [Vinuesa07].

En la Figura 13 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 (éste 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].

Los sistemas de tejido dinámico 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] [Kiczales92][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.

Las principales desventajas del tejido dinámico respecto al estático son:

- Penalización en el rendimiento de las aplicaciones. Reducción del rendimiento de las aplicaciones adaptadas. [Böllert99] [Haupt04]
- Sobrecarga de recursos. En algunos entornos con recursos reducidos como sistemas embebidos o dispositivos móviles, la sobrecarga de recursos que implica el dinamismo [Schröder06] [Popovici01] hacen inaplicable este tipo de tejido.
- Inexistencia de funcionalidades que soporten el dinamismo. [Klefstad02] En algunos entornos no se proporcionan funcionalidades complejas como el dinamismo, esto hace no aplicable este tipo de tejido en las aplicaciones que se ejecuten en esos entornos.
- Reducido conjunto de puntos de enlace. La adaptación en tiempo de ejecución es más compleja para su implementación [Blackstock04].

3.3 Sistemas AOP

Existen varios trabajos de investigación que clasifican los sistemas AOP existentes [Masuhara03a] [Chitchyan04] [Loughran05] [Haupt06] [Vinuesa07], pero ninguno se ha centrado en sistemas de tejido mixtos. En este trabajo se realizará un estudio de algunos de los sistemas AOP mixtos existentes actualmente. Además, como el sistema AspectJ [AspectJ08] es usado como referencia para la implementación de muchos sistemas AOP, y el sistema propuesto en este trabajo también lo usa como referencia para algunas de sus características como el conjunto de puntos de corte, será comentado también.

3.3.1 Sistemas Estáticos

Los sistemas estáticos son los que primero se desarrollaron. El tejido puede ser realizado en tiempo de compilación, en una fase posterior a la compilación y anterior a la ejecución de la aplicación, o en tiempo de carga de las clases en memoria. En tiempo de compilación se necesitaría trabajar con el código fuente de la aplicación, y en los otros casos no es necesario el código fuente.

3.3.1.1 AspectJ

AspectJ [AspectJ08] fue el primer sistema comercial que ofreció AOP, y es el más extendido en la actualidad, siendo un modelo a seguir y de comparación para el resto de sistemas que pretenden ofrecer AOP. Fue desarrollado 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 [Eclipse08].

AspectJ es una especificación de lenguaje y también una implementación de lenguaje que soporta AOP. 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 forma total o parcial. La implementación de la herramienta por parte del equipo *"oficial"*[4] de AspectJ ofrece herramientas para compilar, depurar y documentar código.

Las construcciones de AspectJ extienden el lenguaje Java, 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).

Los conceptos que AspectJ ha añadido a Java son:

- **Puntos de enlace** -*Joinpoints*.
- **Puntos de corte** -*Pointcut designators* o *Pointcuts*.
- Código del aspecto –*Advice*.

Estos conceptos han sido adoptados por la mayoría de los sistemas que implementan orientación a aspectos.

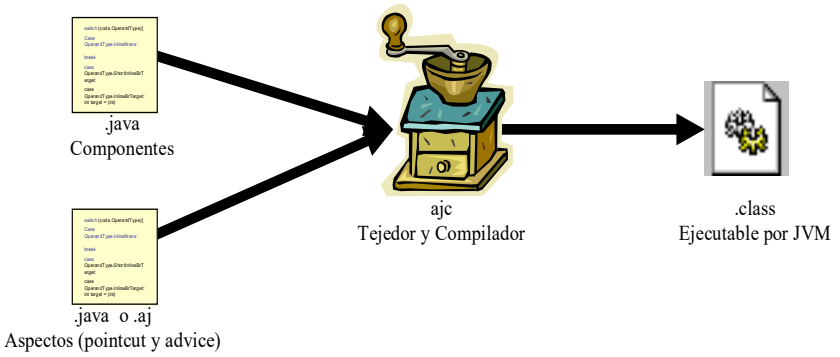


Figura 14: Proceso de compilación en AspectJ [Vinuesa07].

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

La forma de trabajar de AspectJ se puede ver en la Figura 14. 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 *advices*) 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.

También se puede usar el compilador estándar de Java, actuando en este caso el compilador de AspectJ como un mero pre-compilador 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 un principio era así como funcionaba).

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.

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

3.3.1.1.1 Puntos de Enlace

Al ser AspectJ el primer sistema que ofreció orientación a aspectos y por ser actualmente el patrón a seguir por 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:

- 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.
- Pre-inicialización 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 *advices* inyectados.

Éstos 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.

3.3.1.1.2 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`, pre-inicialización 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 *advices* 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.

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.

Los puntos de corte se definen y son posteriormente usados en la declaración de los *advices*.

3.3.1.1.3 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 ejecutar el punto de enlace (una invocación a un método, un acceso a un campo, etc.).
- `after`, el código se ejecuta después.
- `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.
- `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.
- `proceed`, (sólo accesible en los de tipo *around*) que permite ejecutar el código original del punto enlace.

Este sistema presenta un conjunto de puntos de enlace muy rico, permitiendo adaptar las aplicaciones de una forma muy potente. 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 la necesidad de disponer del código fuente del componente a adaptar.

3.3.1.1.4 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 [AspectJ08]:

- 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 [AspectWerkz08], soportando el tipo de tejido presente en este 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 *advices* está fijada previamente a la carga. Además una vez 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 [Haupt06].

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.

3.3.2 Sistemas Mixtos

En la actualidad existen pocos sistemas AOP que permitan tejido mixto. Normalmente los sistemas AOP existentes utilizan solamente uno de los dos tipos de tejido, estático o dinámico.

Algunos tejedores estáticos a veces pueden atribuirse capacidades dinámicas. Pero éstas se refieren únicamente al código que es tejido estáticamente para tomar decisiones en tiempo de ejecución con algunas instrucciones complejas AOP, como por ejemplo `whithin`, `cflow`, o `pertarget`. Un ejemplo de sistema AOP que pudiera considerarse mixto es AspectJ, sin embargo en una clasificación reciente [Haupt06] se explica claramente sus capacidades reales. El sistema inyecta código estáticamente para poder tomar decisiones en tiempo de ejecución sobre instrucciones complejas como las referidas anteriormente.

Los sistemas AOP mixtos que se han buscado para este trabajo son aquellos donde el tejedor realiza (des)tejido de aspectos, tanto previamente a la ejecución de la aplicación como en tiempo de ejecución.

[5] También conocido como tejido binario.

3.3.2.1 LOOM.Net

El Grupo de Sistemas Operativos y Middleware [LOOM.Net08] ha desarrollado dos tejedores, uno dinámico y otro estático, dentro de su proyecto LOOM.Net [Schult03] sobre el mismo núcleo central. Inicialmente crearon un sistema de tejido dinámico –Rapier-LOOM.Net [Köhne05]- sobre la plataforma .Net de Microsoft. Pero ante los problemas de rendimiento, y las restricciones impuestas a las aplicaciones para su adaptación dinámicamente, desarrollaron un sistema de tejido estático - Gripper-LOOM.Net [Schult08].

Los tejedores desarrollados se basan en el uso de los **atributos personalizados** –*custom attibutes*- de la plataforma .Net para indicar las reglas de tejido al tejedor. Éstos, mediante reflectividad y acceso a los metadatos de la aplicación, son evaluados en tiempo de ejecución o previamente al arranque de la aplicación realizando el tejido necesario. El proceso de tejido se realiza sobre código intermedio en tiempo de carga con lo que se consigue independencia del lenguaje.

Los aspectos deben derivar de la clase *Loom.Aspect*. Respecto a las clases base de la aplicación original deben cumplirse una serie de requisitos para poder ser utilizadas en el sistema, sus métodos deben ser virtuales o deben definirse mediante una *interface*. Además existe otra restricción en el sistema dinámico como es la imposibilidad de usar el operador *new* (o el operador o instrucción equivalente en el lenguaje que se haya implementado la aplicación que se está adaptando), se debe usar en su lugar la llamada al método *Loom.Weaver.Create*, para crear instancias de las clases que van a ser tejidas.

```
Base b = Loom.Weaver.Create<Base>(ta);
B.Hello(name);
Console.ReadLine();
```

Tabla 1: Instanciación de clases para el sistema dinámico de LOOM.Net.

```
Base b = new Base();
B.Hello(name);
Console.ReadLine();
```

Tabla 2: Instaciación de clases para el sistema estático de LOOM.Net.

El tejido mediante el sistema dinámico se realiza en el momento de la instanciación de la clase, quedando la clase tejida hasta la finalización de la ejecución del programa, es decir no se puede eliminar un aspecto que se tejió dinámicamente. El tejido estático se realiza en una etapa post-compilación (previa a la ejecución de la aplicación) donde la aplicación se entremezcla con todos los aspectos estáticos.

Un aspecto debe indicar, mediante los atributos personalizados, a que punto de enlace de la aplicación base debe ser añadida. Es decir, debe especificar el punto de corte. No se proporcionan el mismo conjunto de puntos de enlace que AspectJ, aún así el conjunto permite adaptar aplicaciones en muchos partes de ella, como por ejemplo creación de la clase, llamadas a métodos, acceso a propiedades, y finalización de la clase a adaptar. Además de los puntos de corte típicos, se han añadido otros nuevos como *Include*, *IncludeAll*, etc.

El sistema desarrollado presenta numerosas limitaciones:

- Sistema no homogéneo. El tejido estático y el dinámico se utilizan de distinta manera. La sintaxis del lenguaje de puntos de corte no es la misma en ambos tejedores.
- Necesidad de cambiar el código fuente de la aplicación. Es necesario adaptar la aplicación a los requisitos impuestos como que los métodos deben definirse como virtuales, o definirse mediante una *interface*. Además para el tejido dinámico no se puede utilizar el operador `new` y se debe utilizar `Loom.WeaverCreate`, en el estático no hace falta. El código base de una aplicación que va a ser tejido estática o dinámicamente es distinto. Esto tiene implicaciones negativas en la posibilidad de reutilización de 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 [Schult03] [Köhne05].
- Reducido conjunto de puntos de enlace. Aunque en las últimas versiones del sistema se han ampliado los puntos de enlace, todavía no se contemplan los mismos que en AspectJ [Schult03] [Köhne05].
- Acoplamiento de código. La utilización de *custom attributes* para especificar las reglas de tejido en el código de los aspectos, crea un acoplamiento de código dificultando la reutilización de éstos [Schult08].
- Inyección de nuevos aspectos dinámicamente. Aunque el tejido se realice dinámicamente, no se permite el tejido de aspectos no contemplados en tiempo de diseño. Tampoco se permite, la eliminación de un aspecto tejido dinámicamente [Frei04]. No existe un verdadero dinamismo (entendido éste como la adaptación a nuevos requisitos no previstos inicialmente en el diseño).

3.3.2.2 Wicca

Otro sistema mixto de tejido de aspectos es Wicca [Wicca08] [Eaddy07a]. Se ha desarrollado en la Universidad de Columbia, utilizando la plataforma .Net y haciendo uso del framework Phoenix –una infraestructura *back-end* de compilación [Phoenix08]. Soporta tejido dinámico para permitir hacer cambios automáticos a programas C#. También soporta *Edit-and-Continue*[6] para permitir hacer modificaciones manuales, editando el código fuente directamente de un programa escrito en C#, VB.NET o JScript.NET. Wicca automáticamente compila, teje y actualiza el programa en el momento.

Wicca puede realizar tejido estático mediante instrumentación de código, y tejido dinámico usando la API de depuración de CLR. Para el tejido estático se puede usar Phx.Morph o wcs. Phx.Morph es un tejedor estático de *byte-codes* (instrumenta código binario) para aplicaciones .Net construido sobre Microsoft Phoenix. Wcs es un tejedor de código fuente y un compilador, es similar a AspectJ pero en lugar de soportar Java, soporta C#. Puede ser usado en línea de comandos, o mediante un API. Los aspectos son definidos mediante atributos o meta-datos de tipo *advice* en el código fuente.

El tejido dinámico es implementado como un *plugin* (en versión alpha) de mdbg (Microsoft Manager Debugger), y utiliza la herramienta AssemblyDiff también

[6] Característica para ahorro de tiempo que permite hacer cambios sobre el código fuente mientras el programa está en modo depuración.

desarrollada por ellos. AssemblyDiff permite realizar actualización dinámica de aplicaciones, para ello utiliza Microsoft Debugger Edit-and-Continue API. La herramienta produce un fichero de *byte-codes,* y otro de metadatos el cual es consumido por la API.

El tejido dinámico de *byte-codes* es realizado mediante Phx.Morph, quién proporciona un API para soportar tejido en tiempo de carga y en ejecución. Este tejedor tejé el ensamblado como en el caso de tejido estático, y produce un ensamblado temporal. Para ello utiliza la herramienta AssemblyDiff.

Wicca soporta también tejido dinámico usando puntos de ruptura. Para ello, Phx.Morph tejé el ensamblado como en el caso estático, pero en lugar de inyectar llamadas a los *advices*, llama a una función *callback* implementada por Wicca. Cuando Wicca recibe el *callback*, lo marca como un punto de ruptura usando Microsoft Debugger Breakpoint API. Cuando un punto de ruptura es alcanzado, Wicca obtiene los parámetros de contexto usando Micrsoft Debugger API, y entonces ejecuta el código del *advice* en el espacio de proceso del programa en ejecución usando Microsoft Debugger Func-Eval API.

Algunas de las limitaciones presentes en este sistema son:

- La implementación del sistema no es homogénea. Se utilizan dos herramientas con diferentes capacidades para realizar el tejido, Phx.Morph y wcs. Para el tejido en tiempo de compilación se puede usar cualquiera de las dos, pero en tiempo de carga o en ejecución sólo se puede usar Phx.Morph. Además el tejido estático es más expresivo que el dinámico.
- Se necesita acceder al código fuente. Se deben hacer anotaciones en el código fuente. Esto impide tejer aspectos dinámicamente que no se hubiesen considerado antes de estas anotaciones, y hace dependiente a las aplicaciones del sistema de tejido.
- Dependiente de la plataforma. Wicca hace uso de la API de depuración específica de una implementación de la plataforma .Net (la CLR), perdiendo la neutralidad de la plataforma.
- Rendimiento pobre. Su tejido dinámico tiene problemas de rendimiento porque deshabilita la optimización JIT y habilita el soporte *Edit-and-Continue* de la CLR [Eaddy07c], las aplicaciones se deben ejecutar en modo depuración.
- Deshabilitar los aspectos dinámicos. Impide deshabilitar o destejer aspectos tejidos dinámicamente en tiempo de ejecución.

3.3.2.3 AspectC++

Otra interesante iniciativa es *"una familia de tejedores dinámicos"* [Gilani04] realizada por el Grupo de Sistemas Distribuidos y Sistemas Operativos [DCS408]. Se ha realizado usando AspectC++ [Aspectc08], y está fuertemente inspirada en los trabajos realizados sobre el ORB ZEN [Klefstad02].

Esta iniciativa se dirige principalmente al software de sistemas, y específicamente a los sistemas embebidos. Tradicionalmente en el desarrollo de los sistemas tejedores se han implementado herramientas que disponen de casi todas las características AOP, pero existen entornos como los sistemas embebidos donde los

recursos disponibles son escasos, e incluso algunas veces funcionalidades pesadas como el dinamismo no son soportadas.

Esta propuesta se basa en la creación de tejedores dinámicos, pero con la particularidad de que se realizarán artesanalmente o "a mano" para los requisitos particulares de la aplicación específica que se está tratando en cada momento. Su idea principal es realizar una familia de tejedores dinámicos de aspectos. En función de la aplicación que se desea tejer, se crea un tejedor dinámico ad-hoc que se adapta apropiadamente a sus requisitos de tejido. Para la realización de este tejedor se tendrán en cuenta características como: sí los aspectos son conocidos, el orden de interacción de los aspectos, sí los puntos de enlace son conocidos, sí los puntos de enlace son filtrados, características AOP soportadas etc. De esta forma, se puede reducir los recursos necesarios por el sistema de tejedor y ahorrar infraestructura, acomodándose al entorno dónde se va a ejecutar la aplicación adaptada.

Para diseñar estos sistemas tejedores, se basan en que un sistema tejedor dinámico debe ser capaz de hacer uso del tejido estático y el dinámico de acuerdo a los requisitos específicos de la aplicación, y a consideraciones de coste. El diseño de un tejedor particular se debe basar en la familia de sistemas tejedores. Esta familia ha sido definida jerárquicamente por este grupo de investigación, clasificando los tejedores dependiendo de las características AOP implementadas por cada miembro de esa familia.

AspectC++ [Aspectc08] es una extensión para AOP del lenguaje de programación C++ influenciada por AspectJ. Permite escribir aspectos estáticos y dinámicos. Se ha implementado como un pre-procesador de C++, que permite transformar de AspectC++ a C++. Después un compilador convencional de C++ es usado para generar las librerías compartidas.

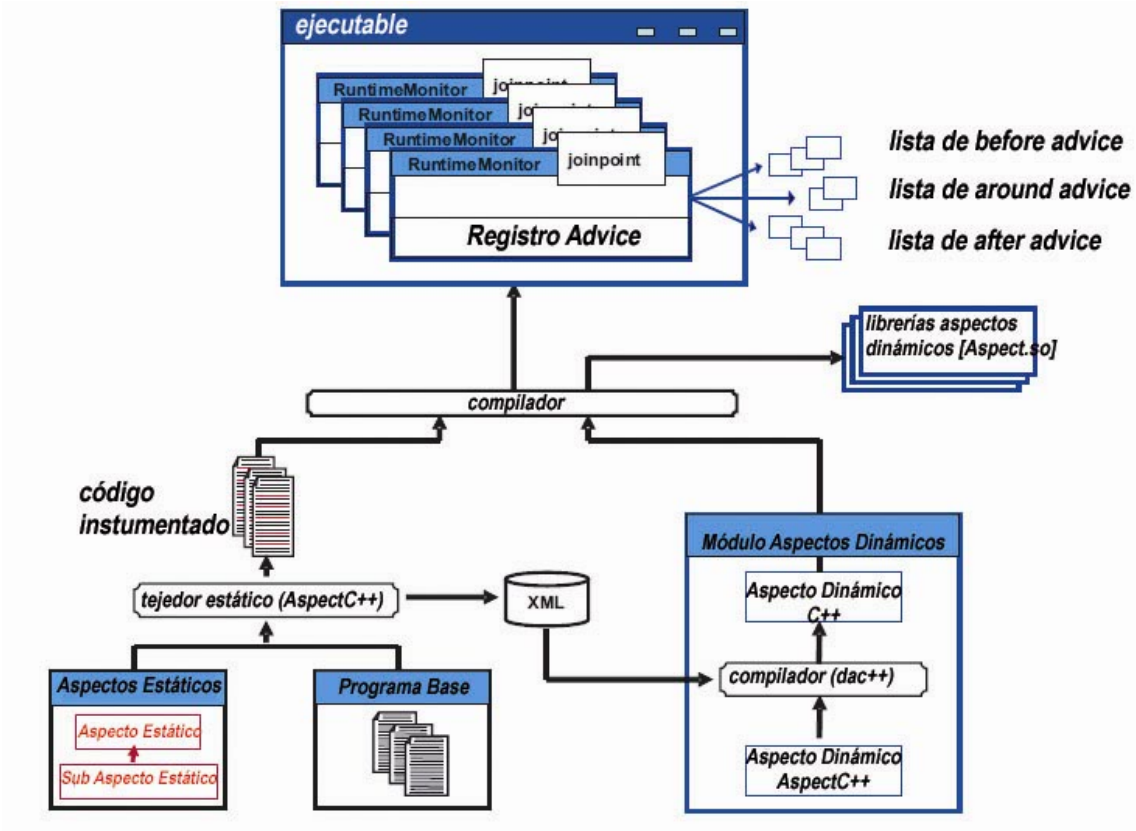


Figura 15: Arquitectura de la "Familia Basada en Tejedores Dinámicos de Aspectos" [Gilani07b].

El tejedor estático AspectC++ genera un informe XML de los puntos de enlace afectados por los aspectos inyectados [Lohmann04]. Esta información es usada por el **Monitor en Tiempo de Ejecución** *–Run-Time Monitor-* para crear la estructura de datos para cada punto de enlace, donde potencialmente se pueda inyectar dinámicamente un aspecto. La estructura de datos es una lista usada para mantener el código de los aspectos registrados dinámicamente junto con la referencia a los puntos de enlace donde se han inyectado. Tan pronto como algún aspecto dinámico es registrado en el *Run-Time Monitor*, la lista de puntos de enlace es recorrida para encontrar los que están afectados por este aspecto. Un puntero al código del aspecto (implementado como un método C++) es añadido a la lista de cada punto de enlace afectado. Cuando se alcanza un punto de enlace en un hilo de control, el monitor es invocado y el aspecto registrado es ejecutado.

En la implementación actual, los aspectos tanto estáticos como dinámicos son escritos usando AspectC++. Los aspectos son librerías compartidas, pueden ser unidas estáticamente con el código del componente, o cargadas en tiempo de ejecución por medio del **cargador** dinámico de aspectos -*loader*. Las expresiones de puntos de corte para describir los puntos de unión de interés se hacen mediante emparejamientos de cadenas y caracteres comodín. Un fichero XML contiene las signaturas de los puntos de corte y sus identificadores únicos.

Algunas limitaciones presentes en este sistema son:

- Problemática específica. Se enfoca casi en su totalidad al ámbito de los sistemas embebidos. Es muy difícil extender el sistema diseñado a otro tipo de aplicaciones.
- Dependiente del lenguaje. Sólo se puede usar el lenguaje de programación C++ para desarrollar las aplicaciones.
- Dependiente de una extensión del lenguaje para definir los aspectos. Los aspectos se deben definir de forma obligatoria usando la extensión AspectC++.

3.3.2.4 AOP.NET

AOP.NET, también conocido como NAop, es otra propuesta de sistema de tejido mixto -implementada como un prototipo muy limitado [Blackstock04]. Esta propuesta ha sido desarrollada por la Universidad de la Columbia Británica.

Su diseño utiliza la decoración de componentes basada en un *proxy*. El tejedor usa la clase *proxy* en lugar de cualquier clase del componente. El *proxy* es capaz de adaptar el comportamiento de la clase decorada. Dependiendo de los puntos de corte, el *proxy* delega su funcionalidad sobre la clase original o llama a los aspectos registrados.

Este *proxy* es usado tanto en escenarios estáticos como dinámicos. El tejedor estático realiza este proceso previamente a la ejecución de la aplicación; el tejedor dinámico lo hace en tiempo de ejecución.

Algunas de las limitaciones que presenta este sistema son:

- Prototipo muy limitado. El prototipo implementado sólo permitía tejer un componente y un aspecto [Blackstock04]. Aunque se informaba de que se realizarían ampliaciones del sistema, hasta la fecha no han aparecido.
- Limitado conjunto de puntos de enlace. [Blackstock04]
- Componentes auto-referenciados. Si un componente se referencia a sí mismo, se estará accediendo al componente sin aspecto en vez del componente con aspecto [Blackstock04].
- Utiliza anotaciones para definir aspectos. Rechaza los ficheros XML para evitar código entremezclado [Blackstock04]. A día de hoy la gran mayoría de los sistemas AOSD han optado por la utilización de ficheros XML, o proporcionan esta opción junto con otras.

4 Sistema Propuesto

DSAW (*Dynamic and Static Aspect Weaver*) [Vinuesa08] es una plataforma orientada a aspectos, de tejido homogéneo estático y dinámico. Su principal objetivo es ser independiente del lenguaje, de la plataforma, y del momento de tejido. Esta plataforma no sólo permite tejer aspectos estática y dinámicamente, sino que también se puede usar la misma implementación de los componentes y de los aspectos independientemente del escenario de tejido elegido.

La aplicación original no necesita ser modificada para ser inyectada con los aspectos, y tampoco los aspectos deben ser modificados si se cambiase su momento de inyección pasando de estático a dinámico, o viceversa. De este modo, se puede usar la orientación a aspectos para realizar prototipado rápido de aplicaciones (usando tejido dinámico), y finalmente optimizar la aplicación resultante (usando tejido estático) sin realizar ningún cambio en su código fuente. El desarrollador será el encargado de equilibrar apropiadamente la relación entre rendimiento y flexibilidad, de acuerdo a los requisitos necesarios de la aplicación [Gilani07b].

Una de las características que ha sido considerada en el diseño de DSAW es ofrecer un conjunto de puntos de enlace amplio y versátil que permiten adaptar cualquier punto significativo de una aplicación. El conjunto ofrecido por DSAW es similar al ofrecido por AspectJ [Kiczales01].

El diseño del sistema ha sido realizado sobre el estándar de referencia de la máquina virtual .Net [Ecma08], sin modificar, ni extender la semántica de la plataforma. Este hecho garantiza la completa independencia de la plataforma, permitiendo el despliegue de este sistema sobre cualquier implementación .Net (por ejemplo Mono, SSCLI y DotGNU).

DSAW realiza la adaptación del software a nivel de Lenguaje Intermedio (IL) de la máquina virtual –ficheros ejecutables y librerías. Esto significa que el tejedor no requiere el código fuente de las aplicaciones, y es independiente del lenguaje.

Por último, los puntos de corte son especificados por medio de ficheros XML, que especifican el mapeo entre los puntos de corte y los *advices*. La gramática XSD de estos documentos XML es una evolución de un diseño usado sobre la plataforma Weave.Net [Lafferty03] [Weave08]. Se ha mantenido el esquema original ya que proporcionaba un conjunto de puntos de corte similar al proporcionado por AspectJ, y se ha ampliado para permitir expresar puntos de corte para ser usados con tejido dinámico. También se ha adaptado para indicar puntos de corte para tejido estático mediante la definición de *advices*.

La separación entre los puntos de corte y los *advices* del código de los aspectos mejora la reutilización de estos. De hecho, los aspectos pueden ser tratados como componentes. Los aspectos pueden ser adaptados por otros aspectos, estática o dinámicamente, independientemente de su lenguaje de programación.

4.1 Ready AOP

DSAW es una ampliación de un prototipo de investigación AOSD desarrollado en la Universidad de Oviedo llamado Ready (*Really Dynamic*) AOP [Vinuesa07] [Vinuesa04]. Ready AOP es un sistema de tejido de aspectos **realmente dinámico** en tiempo de ejecución, independiente del lenguaje y de la plataforma. Mediante este sistema es posible añadir nuevos aspectos (y borrar los tejidos previamente) a una aplicación que se está ejecutando, incluso si los aspectos han sido desarrollados posteriormente al arranque de la aplicación. No existe acoplamiento estático entre los componentes y los aspectos. Ready AOP se encuentra disponible para su descarga gratuita en *http://www.reflection.uniovi.es*.

El sistema fue implementado sobre la especificación estándar de la plataforma .Net, obteniendo así independencia total del lenguaje, y sin modificar la plataforma. El proceso de tejido es realizado a nivel de lenguaje intermedio (IL) de la máquina virtual, sin usar el código fuente de la aplicación [Ortin05]. Su conjunto de puntos de enlace es similar al ofrecido por AspectJ.

DSAW extiende el núcleo de Ready AOP para proporcionar al desarrollador de forma transparente tejido estático y dinámico, manteniendo todas las ventajas de Ready AOP. De está forma, es posible mezclar aspectos estáticos y dinámicos en la misma aplicación. También es posible cambiar, en cualquier momento del proceso de desarrollo del software, cuando un aspecto debe ser tejido estática o dinámicamente. El desarrollador puede así gestionar el equilibrio entre rendimiento y flexibilidad obteniendo la aplicación más apropiada en cada momento dependiendo de los requisitos, y permitiendo la separación de la incumbencia momento de tejido.

4.1.1 Objetivos de Ready AOP

El objetivo principal perseguido con la realización de Ready AOP era obtener un sistema que ofreciese *soporte para el desarrollo de software orientado a aspectos de forma dinámica y completamente independiente del lenguaje y de la plataform*a.

Otros requisitos de este sistema fueron: que fuese de propósito general, que tuviese un amplio conjunto de puntos de enlace, y un lenguaje de definición de puntos de corte con gran expresividad.

Además en la selección de la plataforma sobre la que se implementó el sistema se buscó que ésta cumpliese los siguientes requisitos: ser de uso comercial, estar ampliamente difundida, y tener un buen rendimiento en tiempo de ejecución. La plataforma finalmente elegida fue .Net.

En relación con las aplicaciones que debían poder ser adaptadas por el sistema diseñado, los requisitos que tenía que cumplir éste fueron: el sistema implementado tenía que trabajar con el código binario de las aplicaciones sin usar sus códigos fuentes, y el sistema de tejido tampoco debía imponer condiciones a las aplicaciones que se iban a adaptar.

En relación con los aspectos, se consideró que los aspectos debían ser tratados como aplicaciones. Se exigió al sistema que permitiese aplicar varios aspectos de forma simultánea, contemplar mecanismos de coordinación entre los aspectos inyectados en el

mismo punto de enlace, facilitar la reutilización de los aspectos, uso de un lenguaje estándar sin restricciones, y separar el código de los aspectos de los puntos de corte.

Todos los requisitos que se exigieron a Ready AOP se han mantenido en el nuevo sistema diseñado e implementado, DSAW.

4.1.2 Funcionamiento de Ready AOP

Cuando una aplicación comienza su ejecución en el sistema Ready AOP, lo cual significa que ya está compilada, tiene que pasar por un proceso previo en el que se analiza su código y parte es modificado por el sistema. El elemento del sistema que se encarga de realizar esta tarea se denomina **Inyector de Puntos de Enlace -***Join Point Injector* (JPI). El JPI inserta rutinas de reflectividad 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, cómo son las clases que implementan la parte del **Framework de Aspectos** correspondiente a la aplicación, los **Mecanismos de Introspección Intraaplicaciones**, los elementos necesarios para la **Intercomunicación entre procesos**, y el código necesario para registrarse y desregistrarse en el sistema cuando comience su ejecución. Una vez generado el código definitivo, se obtiene un nuevo ejecutable, con la funcionalidad ampliada, que es él que realmente se ejecuta en el sistema.

Una vez que la aplicación ha sido modificada puede comenzar su ejecución, y lo primero que debe hacer es registrarse en el **Servidor de Aplicaciones** del sistema con un identificador global que la identifica de forma única dentro del 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 Ready AOP, los aspectos son aplicaciones normales completamente funcionales que, tienen las mismas características que cualquier otra aplicación. Si un aspecto necesita adaptar el comportamiento de una aplicación que está ejecutándose en el sistema, lo solicita al servidor. El aspecto debe enviar al servidor información de sí mismo, un fichero XML con la especificación de los puntos de corte deseados y, obviamente, una identificación de la aplicación a la que se 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 (esto lo hace mediante las clases del Framework de Aspectos anteriormente inyectadas).

Cuando la ejecución de la aplicación que está siendo adaptada alcanza alguno de los puntos de enlace activados anteriormente, se notifica al aspecto que solicitó la activación que esto ha ocurrido (utiliza las interfaces adecuados del Framework de 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 de Aspectos), 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 una aplicación necesita cambiar en cualquier momento los puntos de enlace seleccionados, puede hacerlo suministrándole un nuevo

fichero XML con la especificación de los nuevos puntos de corte deseados al servidor. El servidor actuará de forma idéntica a lo comentado anteriormente.

En el desarrollo de DSAW se ha mantenido este funcionamiento, y se han añadido más funcionalidades para permitir el tejido estático de los aspectos.

4.2 Arquitectura del Sistema DSAW

El sistema DSAW está compuesto de tres elementos principales:

1. Join Point Injector (JPI). El inyector de puntos de enlace se encarga del tejido estático de aspectos, y de añadir código para facilitar la inyección dinámica de aspectos posteriormente. Se encarga de añadir código a la aplicación base, es decir instrumentarla, para añadirle un MOP que ofrezca reflectividad computacional al sistema, y los aspectos estáticos. Además se encarga de añadir el código necesario para que la aplicación interactué dentro del sistema, con el servidor de aplicaciones y con los aspectos inyectados dinámicamente.
2. Servidor de Aplicaciones. Se encarga de coordinar los componentes con los aspectos que son tejidos dinámicamente. Mantiene la relación de las aplicaciones que se están ejecutando en el sistema, ofrece a los aspectos que se van a inyectar dinámicamente la posibilidad de adaptar las aplicaciones del sistema. También recibe las peticiones de adaptación de estos aspectos, las cuales son enviadas hacia las aplicaciones donde los puntos de enlace apropiados seleccionados por los puntos de corte son activados o desactivados, y también envía a las aplicaciones la referencia a los aspectos que las adaptan.
3. Framework de Aspectos. Formado por un conjunto de interfaces encargadas de integrar todos los elementos del sistema en escenarios de tejido dinámicos. Estas interfaces deben ser implementadas por los aspectos que se van a inyectar dinámicamente y los componentes que utilicen el sistema. En el caso de los componentes, esto no implica que tenga que ser el desarrollador el encargado de implementarlas, es el propio sistema, a través del JPI, quien se encarga de añadir las clases necesarias a la aplicación. Estas interfaces también pueden ser usadas por los aspectos que se van a inyectar estáticamente para facilitar su programación.

En este sistema, hay dos etapas en el ciclo de vida de las aplicaciones. La primera es previa a la ejecución de la aplicación. Una vez que la aplicación ha sido compilada, el inyector de puntos de enlace (JPI) manipula el programa, tejiendo los aspectos estáticos, e incluyendo código para hacer posible en tiempo de ejecución de la aplicación el tejido dinámico de aspectos.

La segunda etapa del ciclo de vida de las aplicaciones está referida a su ejecución. La aplicación es iniciada junto con los aspectos tejidos estáticamente. El servidor de aplicaciones proporciona la adaptación dinámica de la aplicación en tiempo de ejecución. Cualquier aspecto, haciendo uso de las interfaces definidas en el framework de aspectos, será capaz de adaptar las aplicaciones que se están ejecutando.

En DSAW, los aspectos inyectados dinámicamente podrían también ser procesados por el inyector para convertirse en aplicaciones totalmente funcionales en este sistema –por ejemplo componentes.

4.2.1 Otros Elementos del Sistema

Otros elementos de los que consta el sistema se describen de forma breve a continuación:

- Máquina Abstracta. El sistema se ha implementado sobre una máquina abstracta, la plataforma .Net. De esta forma, se obtiene independencia del lenguaje, y portabilidad o independencia de la plataforma, de forma directa, beneficiándose de las características de la máquina.
- Especificación del lenguaje de definición de puntos de corte y *advices*. Mediante una gramática XSD se define el formato de los ficheros XML donde vendrán especificados los puntos de corte, y los mapeos entre éstos y los *advices*.
- Mecanismos de Introspección entre Aplicaciones. Se necesitan capacidad reflectivas externas para el caso de los aspectos inyectados dinámicamente. Estos aspectos pueden acceder reflectivamente a las características de la aplicación.
- Intercomunicación entre Procesos. Para los escenarios de tejido dinámico, es necesario que las aplicaciones pueden comunicarse con el servidor de aplicaciones, y con los aspectos inyectados dinámicamente. Estos últimos también deben comunicarse con el servidor de aplicaciones. Para ello se ha usado la API .Net Remoting.

4.3 Tejido Estático e Inyección de Puntos de Enlace

Previamente a la ejecución de la aplicación, ésta es procesada por el **Inyector de Puntos de Enlace** *–join point injector* (JPI). El JPI recibe la aplicación, los aspectos a ser tejidos estáticamente, y el documento XML mapeando los puntos de enlace a los *advices*. El JPI genera una nueva aplicación que incluye la funcionalidad principal más los aspectos que han sido tejidos estáticamente.

Además, el JPI también instrumenta el código de la aplicación con código que permite la adaptación dinámica por los aspectos en tiempo de ejecución. Concretamente, lo que se añade es un *Meta Object Protocol* (MOP) que ofrece reflectividad computacional dinámica [Kizcales92] [Ortin02a] (permite la adaptación dinámica de aplicaciones en ejecución). Este MOP modifica parte de la semántica de la máquina virtual como el mecanismo de paso de mensajes y el acceso a los campos o propiedades. De este modo, se pueden adaptar aplicaciones ejecutándose con aspectos dinámicos. Finalmente, el JPI añade código a la aplicación para otras funcionalidades de la plataforma como el registro de la aplicación al iniciarse en el servidor de aplicaciones, introspección entre aplicaciones y el desregistro de la aplicación a su fin (Figura 16).

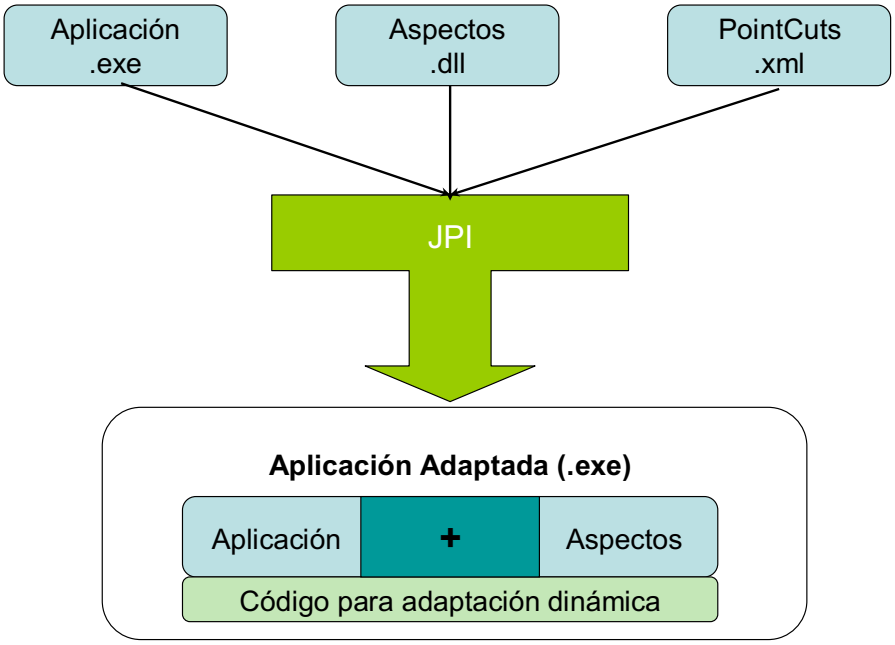


Figura16: Tejido estático e inyección de puntos de enlace.

El JPI realiza las operaciones siguientes:

1. El JPI toma como entrada la aplicación compilada, los aspectos a ser inyectados estáticamente, y el fichero XML especificando el mapeo entre los puntos de enlace y los *advices*.
2. Analizando el código IL, el JPI detecta la sombra de los puntos de enlace de la aplicación (el mapeo entre los puntos de enlace y los puntos en el código del programa donde el compilador realmente opera [Masuhara03b]).
3. Cuando la sombra de un punto de enlace es seleccionada por cualquier punto de corte descrito en el documento XML, una llamada al *advice* que el desarrollador desea tejer es incorporada dentro del código IL.
4. Junto con el código añadido a la aplicación descrito en el punto anterior, un MOP reflectivo es incluido en cada sombra de punto de enlace. Este MOP hará a la aplicación capaz de ser adaptada en tiempo de ejecución por nuevos aspectos durante su ejecución. Con el objetivo de obtener un mejor rendimiento en tiempo de ejecución, la inyección del MOP podría ser evitada en las sombras de algunos puntos de enlace –para evitar esta inserción de código en el fichero XML donde se especifican los puntos de corte se pueden utilizar expresiones regulares, y operadores lógicos para seleccionar o deseleccionar partes del código de la aplicación que se está adaptando. Ésta es una de las posibles optimizaciones que el desarrollador puede realizar para equilibrar rendimiento y adaptabilidad en tiempo de ejecución, ya que se puede realizar una inyección de código en todos los puntos de enlace con lo que la ejecución de la aplicación se ralentizaría debido a las comprobaciones que se deben hacer pero permitiría adaptar cualquier punto de ella dinámicamente, o se podría instrumentar menos puntos de enlace

penalizando menos el rendimiento pero impidiendo adaptar cualquier punto de ella dinámicamente.
5. La aplicación almacenará en una estructura de datos referencias a las sombras de puntos de enlace alteradas. Estas referencias serán usadas para activarlas o desactivarlas en tiempo de ejecución.
6. El JPI también añade código para registrar la aplicación en el servidor de aplicaciones al iniciarse. Además, el JPI añade una rutina para desregistrar la aplicación a su fin, y funcionalidades para la publicación de información reflectiva .Net para permitir a los aspectos en tiempo de ejecución inspeccionar la estructura de la aplicación.
7. Después de analizar la aplicación e inyectar todo el código necesario, la aplicación es compilada de nuevo.
8. Finalmente la aplicación instrumentada y compilada es almacenada a fichero. Se genera un nuevo fichero binario, el fichero binario con la aplicación original no se modifica.

4.3.1 Lectura del Código IL

Para desensamblar la aplicación a adaptar, y acceder de forma directa instrucción por instrucción al código IL que la forma, así como acceder a la meta-información asociada, se ha usado el proyecto ILReader [Luo08].

Ready AOP utilizaba para esta tarea la librería *Reflector for .Net* [Roeder08]. La plataforma sobre la que se había implementado Ready AOP era el Framework .Net 1.5. DSAW se ha implementado sobre la versión 2.0 de la misma plataforma .Net. Con la aparición de la versión 2.0 del Framework .Net, también se desarrolló un nuevo *Reflector* el cual no permite acceder a las instrucciones en código IL de la misma forma. Esto impide generar el nuevo código IL con las instrucciones leídas del ejecutable.

Se valoró la utilización de otras herramientas para la lectura de código IL, entre ellas CLIFileReader[7] , RAIL [Rail08], Absil [Absil08], el citado Reflector.dll [Roeder08] o Cecil [Cecil2008] para realizar la instrumentación de código en la versión 2.0. Finalmente se eligió la herramienta ILReader [Luo08]. Esta herramienta tiene las mismas funcionalidades que la versión anterior de Reflector, pero además proporciona su código fuente pudiéndose integrar dentro del sistema desarrollado de forma rápida, y sin apenas cambios en el código del sistema de partida.

4.4 Ejecución de la Aplicación

La aplicación una vez procesada por el JPI, ya ha sido tejida junto con sus aspectos estáticos. Estos aspectos ya están incluidos en la aplicación para su ejecución. Si nuevos aspectos estáticos deben ser añadidos, o alguno de los existentes debe ser eliminado, el JPI debe reprocesar la aplicación.

Sin embargo, si la aplicación necesita ser adaptada en tiempo de ejecución, este sistema facilitará la adición o borrado dinámico de estos aspectos. Este funcionamiento solamente se puede obtener después de la instrumentación de la aplicación por el JPI, ya

[7] Inaccesible. Consultar http://www.dsg.cs.tcd.ie/dynamic/?category_id=144

que en este paso se añade el código necesario para facilitar la inyección de aspectos dinámicamente.

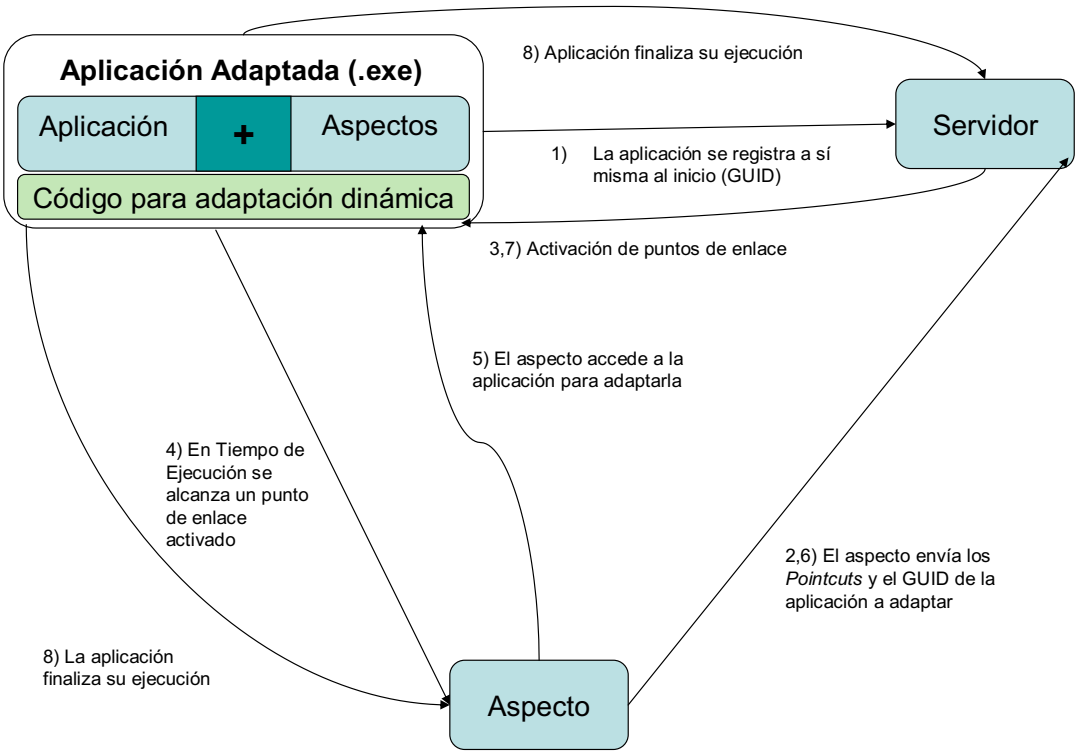


Figura 17: Adaptación dinámica mediante DSAW.

Los pasos que se siguen en tiempo de ejecución para adaptar una aplicación con aspectos inyectados dinámicamente (Figura 17) son los siguientes:

1. Cuando se inicia la aplicación, ésta se registra automáticamente en el servidor de aplicaciones con un GUID (*Global Unique Identifier*) siguiendo la especificación OSF/DCE [Miller92]. Este GUID fue generado previamente por el JPI durante la instrumentación de código, y es usado para identificar la aplicación en el sistema.
2. Una vez que la aplicación está ejecutándose, un aspecto puede ser usado para adaptarla dinámicamente. Para ello, el aspecto llama al servidor de aplicaciones vía .Net Remonting. El aspecto envía la siguiente información: una referencia a sí mismo, un documento XML especificando los puntos de corte, y el GUID de la aplicación a ser adaptada.
3. El servidor procesa el fichero XML creando una estructura con los puntos de corte requeridos por el aspecto. Estos puntos de corte son activados en los puntos de enlace de la aplicación seleccionados por medio del MOP que fue añadido en la fase previa por el JPI. Esta activación implica la llamada exacta al aspecto en tiempo de ejecución, haciendo uso de todos los servicios del Framework de Aspectos. El resultado del proceso es el tejido dinámico. El servidor transmite la información necesaria a la aplicación para que pueda comunicarse con el aspecto inyectado dinámicamente.
4. Cuando la ejecución de la aplicación tejida dinámicamente alcanza un punto de enlace activado, el sistema llama los aspectos dinámicos que han sido previamente tejidos en ese punto de la aplicación. Estos aspectos deberían de

implementar las interfaces específicas descritas en el framework de aspectos [Vinuesa04] para hacer posible que esto ocurra. De este modo, la aplicación enviará al aspecto información relativa al punto de enlace y de la propia aplicación (incluyendo su propia la referencia). El aspecto tiene ahora la oportunidad de adaptar el funcionamiento de la aplicación en tiempo de ejecución.

5. El aspecto puede usar la referencia a la aplicación para acceder a ésta por medio de los servicios reflectivos de publicación .Net añadidos por el JPI. Las operaciones ofrecidas por estos servicios reflectivos son: inspección de la estructura de la aplicación, modificación y acceso de los valores de los campos o propiedades, e invocación a miembros (métodos, constructores, campos o propiedades) de la aplicación.
6. Un aspecto tejido dinámicamente podría tener la necesidad de cambiar los puntos de corte en tiempo de ejecución. Esta operación es también ofrecida por el servidor de aplicaciones. El aspecto debería enviar un nuevo documento XML especificando el nuevo conjunto de puntos de corte. El servidor de aplicaciones analizará entonces el nuevo fichero XML, procesando los nuevos puntos de enlace y mediante el MOP añadido en la fase previa los nuevos puntos de enlace se activarían, y se desactivarían los viejos puntos de enlace de la aplicación en ejecución.
7. Si un aspecto dinámico no quiere adaptar la aplicación más, lo notifica al servidor de aplicaciones. El servidor de aplicaciones indicará al MOP que desactive los puntos de enlace anteriormente activados por el aspecto. Por lo tanto, el aspecto no recibirá notificaciones futuras.
8. Cuando la ejecución de la aplicación finalice, el código añadido por JPI notifica a al servidor de aplicaciones que la aplicación ha finalizado.

El servidor de aplicaciones actúa como un mediador entre los aspectos y las aplicaciones. Esta mediación es sólo realizada cuando los puntos de enlace son activados o desactivados. Una vez que estas operaciones han sido realizadas, la aplicación y el aspecto interaccionan directamente entre ellos. El aspecto puede inspeccionar la aplicación cuando una notificación de un punto de enlace es recibida.

Se podría dar el caso de que este sistema fuese utilizado por *hackers*, para dañar la aplicación original o para realizar operaciones no permitidas, usando aspectos inyectados dinámicamente para añadir código malicioso. Esto se podría evitar implementando mecanismos para validar la identidad y los permisos de los aspectos, que pretenden adaptar las aplicaciones en el servidor de aplicaciones.

Con este diseño, las aplicaciones no necesitan conocer los aspectos que podrían ser tejidos en tiempo de ejecución. Además de esto, los aspectos pueden ser inyectados a cualquier aplicación, o incluso a otros aspectos, sin ninguna dependencia estática. Este funcionamiento reduce el acoplamiento, y promueve la reutilización de aspectos y componentes.

Para la implementación de este sistema se ha utilizado la API de comunicaciones .Net Remoting usando el canal *named_pipes* para comunicar el servidor de aplicaciones, los aspectos inyectados dinámicamente y las aplicaciones. .Net Remoting es un API estándar sobre la plataforma .Net, y es independiente del canal. Se ha usado *named_pipes* por una cuestión de rendimiento, pero los usuarios de este sistema podrían usar cualquier canal que ellos quisiesen modificando sólo los ficheros de configuración. El sistema diseñado en este trabajo podría ser usado incluso en entornos distribuidos.

4.5 Resolución de Conflictos entre los Aspectos

DSAW permite el tejido de muchos aspectos en el mismo punto de enlace de una aplicación. Esta característica implica la necesidad de establecer estrategias de coordinación entre los aspectos [Nagy05]. El programador podrá usar estos mecanismos de coordinación para priorizar la ejecución de los aspectos.

El mecanismo de coordinación ofrecido por DSAW se basa en la clasificación de los aspectos en estáticos y dinámicos, más la utilización de prioridades para ordenar los aspectos. Como DSAW no implementa una técnica para detectar conflictos semánticos en puntos de enlace compartidos [Durr05], el programador debe identificar los requisitos de la resolución de conflictos con un valor para la prioridad. Por último, también se tiene en cuenta cuando el aspecto fue registrado en el servidor de aplicaciones.

Debido a que los aspectos dinámicos están orientados a la adaptación de aplicaciones en tiempo de ejecución ante requerimientos emergentes o posibles errores en tiempo de ejecución, se ha proporcionado a los aspectos dinámicos una mayor precedencia que a los estáticos para poder realizar esta tarea.

Los aspectos estáticos son tejidos siguiendo una estrategia basada en prioridades que el desarrollador de la aplicación podría cambiar mientras la aplicación está siendo desarrollada. Por lo tanto, la ejecución de los aspectos estáticos registrados en el mismo punto de enlace dependerá de su nivel de prioridad. El JPI es el responsable de tener en cuenta está prioridad mientras la aplicación está siendo tejida estáticamente.

En el caso de los aspectos dinámicos, DSAW da la prioridad a éstos en relación con los aspectos estáticos. Los conflictos entre los aspectos tejidos dinámicamente son también resueltos mediante una estrategia basada en la prioridad como los aspectos tejidos estáticamente. La principal diferencia entre los dos tipos de aspectos es que la resolución de los aspectos inyectados dinámicamente es realizada por el desarrollador, mientras que los conflictos de los aspectos estáticos son resueltos por el JPI. Aparte de esto, los aspectos dinámicos pueden modificar su prioridad en tiempo de ejecución, dependiendo de sus requerimientos específicos.

Una restricción que se ha añadido al sistema implementado, es que si un aspecto inyectado dinámicamente se va a ejecutar en el momento *around* sobre un punto de enlace de tipo *Method Call,* o *Field Set*, no se ejecutarán los aspectos que se hubiesen inyectado estáticamente en ese punto de enlace, en caso de que los hubiese. Se considera que cuando se inyecta dinámicamente de esta forma, el objetivo es anular todo código estático de la propia aplicación, y de los posibles aspectos inyectados estáticamente. En caso de que se quisiese ejecutar el código original de la aplicación se podría llamar reflectivamente al método original.

Finalmente, aspectos con el mismo tipo de dinamismo y prioridad son ejecutados siguiendo una política FIFO.

4.6 Rendimiento en Tiempo de Ejecución

La principal desventaja del tejido dinámico es el rendimiento en tiempo de ejecución [Böllert99] [Haupt04]. Una aplicación orientada a aspectos estáticamente tejida puede ser casi tan eficiente como si fuese desarrollada sin una aproximación AOSD [Böllert99]. No obstante, el proceso de adaptar una aplicación en tiempo de ejecución, así como el uso de reflectividad, provoca un grado considerable de sobrecarga durante la ejecución de la aplicación [Popovici03].

En el sistema anteriormente desarrollado, Ready AOP, este decremento en el rendimiento en tiempo de ejecución se producía por dos características críticas:

1. El MOP inyectado por el JPI para activar o desactivar los puntos de enlace.
2. La comunicación entre los diferentes módulos del sistema.

La primera causa significa un chequeo dinámico de la activación de los puntos de enlace. También realiza un conjunto de verificaciones que manipula el estado de la pila, significando éste un coste en el rendimiento. El coste medio en el rendimiento de estas operaciones es de un 70%.

La segunda penalización del rendimiento en tiempo de ejecución es derivada de la comunicación entre los aspectos, las aplicaciones, y el servidor de aplicaciones. Se ha usado la API .Net Remoting para este propósito. El coste de rendimiento es lineal con respecto a la información intercambiada. La comunicación entre los procesos significa que habrá un coste más alto que repercute en el rendimiento en tiempo de ejecución. Está es la razón por la que DSAW es una mejora de Ready AOP, uno de los beneficios del tejido estático es un incremento en el rendimiento de las aplicaciones adaptadas.

Mientras las aplicaciones están siendo desarrolladas, DSAW ofrece la posibilidad de aplicar el tejido dinámico para facilitar la creación de aplicaciones AOSD. Esto facilita el desarrollo incremental y la aplicación de un esquema de depuración ágil *"fix-and-continue"* [Ortin02a] [Dmitriev02]. Antes del despliegue de la aplicación, los aspectos que no necesiten ser tejidos dinámicamente podrían ser inyectados estáticamente en la aplicación final [Ortin03a]. De este modo, el rendimiento de la aplicación en su conjunto mejoraría notablemente. Este es el principal beneficio de separar la incumbencia de momento de tejido en este sistema.

Otra limitación de rendimiento del sistema estaría causada por la ejecución sobre una máquina virtual. Sin embargo, la utilización comercial actualmente de muchas máquinas virtuales basadas en lenguajes intermedios -como por ejemplo Java, Python o C# - muestran la inexactitud de este razonamiento. Ya que debido a técnicas de compilación **JIT** -*just-in-time*-, o generación adaptable de código nativo estas máquinas virtuales han logrado incrementar de forma notable su rendimiento.

4.7 Generación Automática de Stubs para Aspectos

Para que los aspectos puedan ser inyectados dinámicamente necesitan un *stub* o *dispatcher*. Este componente se encargará de instanciar el aspecto, registrar el aspecto

junto con los puntos de corte y el GUID de la aplicación que se necesita adaptar en el servidor de aplicaciones del sistema. Posteriormente cuando la aplicación adaptada alcance un punto de enlace activado por ese aspecto, se establecerá una comunicación a través entre la aplicación y el aspecto inyectado dinámicamente en ese punto usando el *stub* generado como mediador. Finalmente, cuando la aplicación no necesite seguir siendo adaptada por este aspecto, el *dispatcher* se encargará de desregistrar toda la información relativa al aspecto del servidor de aplicaciones.

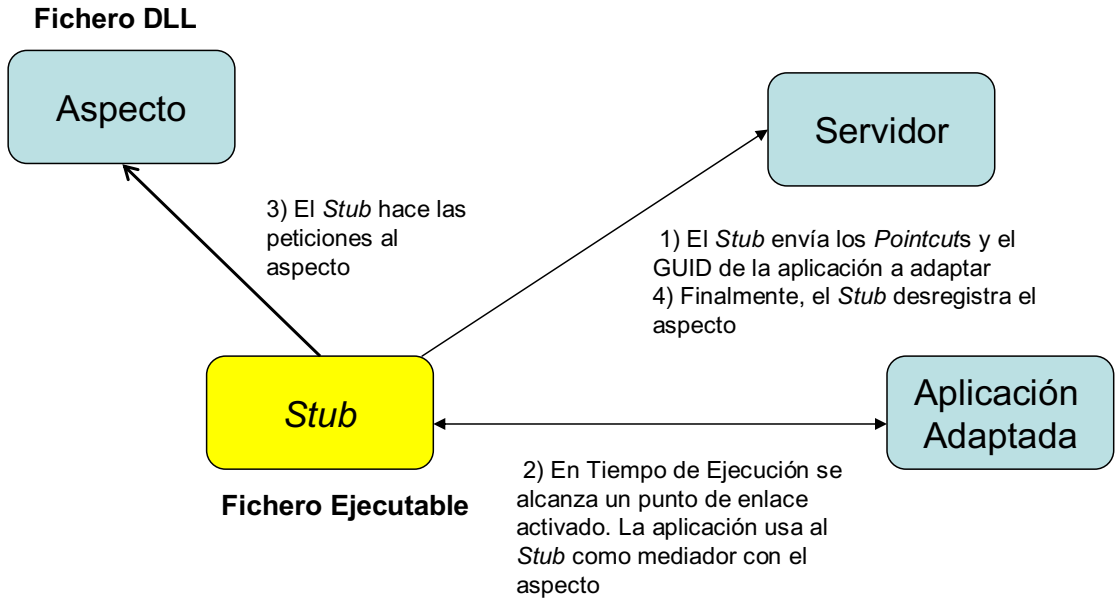


Figura 18: Utilización de un *stub* para comunicarse con un aspecto.

Los desarrolladores puede programar estos *stubs* de forma rápida y fácil, ya que su código es idéntico en todos los casos, utilizando un ejemplo inicial sólo deberían cambiar el nombre de de las clases que encapsulan los aspectos, y los nombres de sus funcionalidades. Pero para conseguir un sistema más transparente se ha realizado una utilidad que permite **generar de despachadores** de forma automática.

Esta utilidad recibe como parámetros una librería (DLL) con los aspectos desarrollados, y genera un fichero ejecutable (*stub*). Este fichero ejecutable se encargará de comunicar los aspectos de la librería con el servidor de aplicaciones, y de actuar como mediador entre la aplicación que se está siendo adaptada y el aspecto.

El despachador generado tiene como parámetros un fichero XML, donde se indicar los puntos de corte donde el desarrollador pretende inyectar el aspecto dinámicamente, y un GUID para indicar la aplicación que se quiere adaptar.

Para la realización de esta utilidad se ha utilizado la librería `System.CodeDom` [CodeDOM08] proporcionada por la plataforma .Net. Usando está librería se ha creado el fichero ejecutable. También se ha usado la librería ILReader [Luo08] para acceder a la librería donde se encuentran los aspectos encapsulados, y leer las signaturas de los métodos que actuarán como aspectos.

Con la realización de esta utilidad se consigue un beneficio adicional, éste es no imponer restricciones a los aspectos. Cualquier librería, desarrollada sobre cualquier lenguaje soportado por el sistema, puede ser procesada por esta utilidad para la generación de un *stub* que permita su uso como aspecto. Además es posible el procesamiento de librerías de las que no se dispone código binario, ya que se trabajara a nivel de código intermedio y por lo tanto no es necesario el código fuente.

4.8 Especificación del Lenguaje de Definición de Puntos de Corte

Entre los requisitos que se impusieron el desarrollo de Ready AOP estaba la definición e implementación de un lenguaje de puntos de cortes que ofreciese una gran expresividad. Además este lenguaje debía permitir la separación total del código de los aspectos y de los puntos de corte.

Para cumplir este requisito se definió una gramática XSD para la creación de ficheros XML que definiesen los puntos de corte solicitados por los aspectos. La gramática XSD utilizada es una evolución de un diseño usado sobre la plataforma Weave.Net [Lafferty03] [Weave08].

La gramática de partida fue evolucionada buscando por un lado facilitar el tejido dinámico de aspectos. Por otro lado se buscó una expresividad que permita seleccionar o deseleccionar cualquier punto de enlace de la aplicación.

Finalmente con la incorporación de aspectos tejidos de forma estática sobre las aplicaciones, algunas veces hasta de forma simultánea con aspectos inyectados dinámicamente, se ha implementado nuevas construcciones que proporcionan esta expresividad. Además, se han definido etiquetas que faciliten las reutilización de puntos de corte entre los dos tipos de tejido y permitan su coordinación.

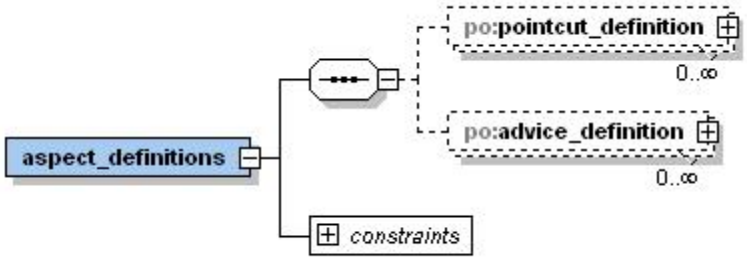


Figura 19: Definición de puntos de corte y de aspectos estáticos.

4.8.1 Definición de un Punto de Corte

Un punto de corte tiene dos elementos principales: el momento (<*time*>) cuando se va a ejecutar y el tipo de punto de corte (<*joinpoint_type*>). Estos dos elementos constituyen su definición (<*pointcut_definition*>). El momento de inyección puede ser: "antes" (*before*), "después" (*after*) o "en vez de" (*around*).

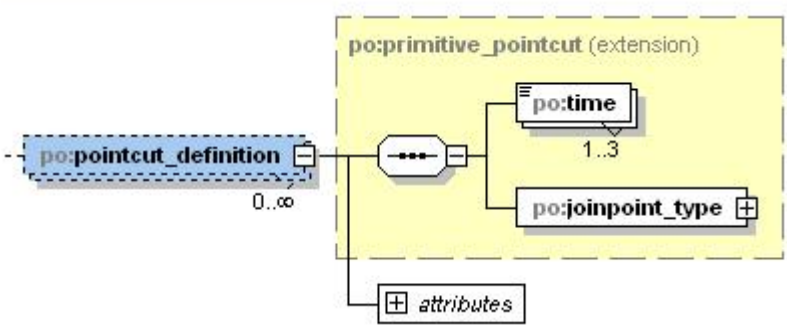


Figura 20: Elementos principales de un punto de corte.

El tipo de un punto de corte depende de la parte de la aplicación que se quiere seleccionar o deseleccionar. Puede ser un método, un constructor, un campo o una propiedad. Para un método o constructor, se pueden seleccionar sus llamadas, o cuando se ejecutan estas llamadas. Para un campo o propiedad se puede indicar si el acceso es de lectura o de escritura.

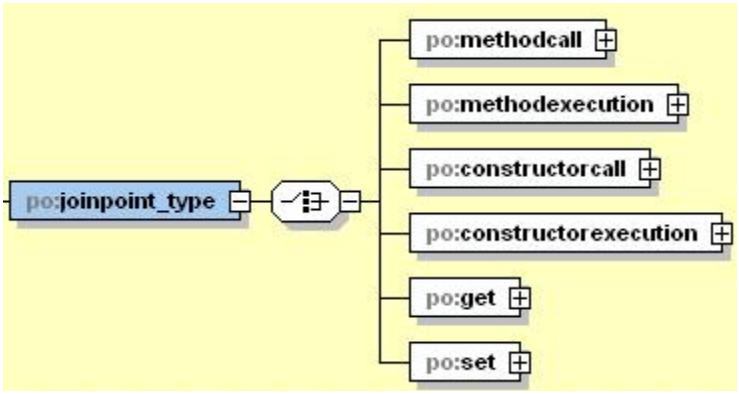


Figura 21: Tipos de puntos de corte.

En caso de que el punto de corte sea para un método o un constructor, se debe definir la signatura del método o constructor con él que se realizarán las comparaciones. Para el caso de un campo, se definirá el tipo y el nombre del campo que se quiere seleccionar.

La signatura de un método se compone de los siguientes elementos: flags del método, tipo de retorno del método, nombre del método cualificado (en caso de que esté cualificado) y sus parámetros. En el caso de definir la signatura de un constructor, no existe valor de retorno, y el nombre del método debe coincidir con el nombre de la clase. El sistema soporta todos los flags (también para campos o propiedades) de la especificación del lenguaje ECMA [Ecma2008] como por ejemplo: `privatescope`, `private`, `assembly`, etc.

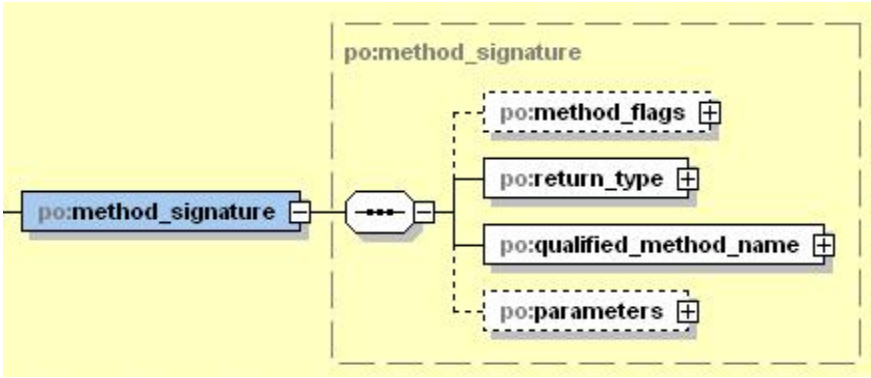


Figura 22: Elementos de la signatura de un método.

Para el caso de un campo o propiedad, sus elementos son: flags del campo o de la propiedad, tipo del campo o de la propiedad, y nombre de la clase en caso de que esté cualificado.

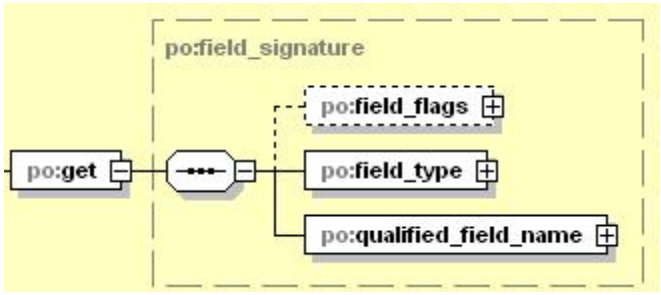


Figura 23: Elementos de la signatura de un campo o propiedad.

En el último nivel de la especificación del punto de corte, se puede usar el nombre del elemento como el nombre del método, nombre del campo, tipo de retorno, etc.; o se puede definir un patrón de búsqueda utilizando los caracteres comodín y los operadores lógicos `AND`, `OR` o `NOT`. De esta forma se seleccionarán todos los métodos que cumplan el patrón, o los que no lo cumplan.

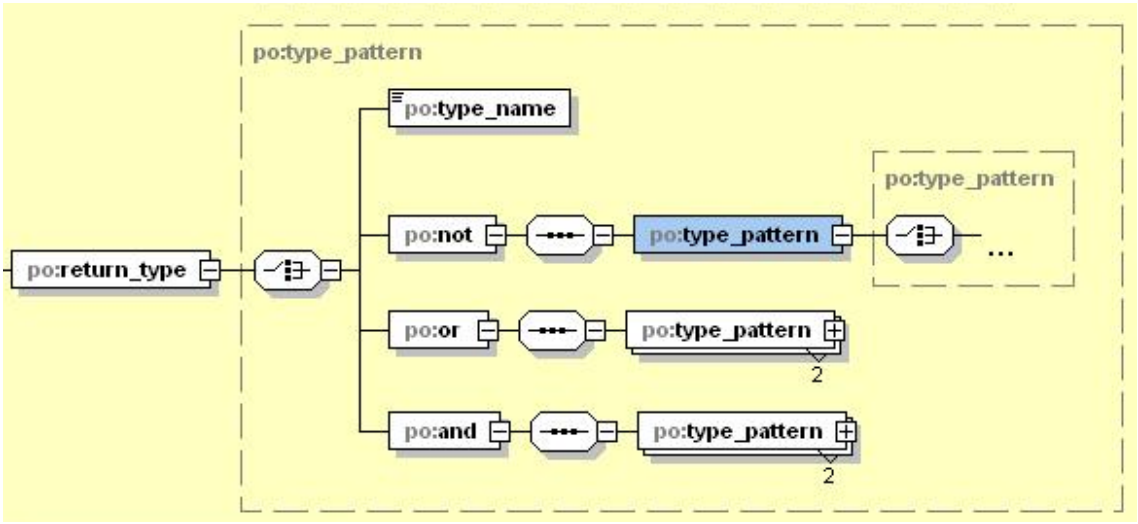


Figura 24: Patrón para la definición de un tipo de retorno.

En el siguiente fragmento de código se puede 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>
```

Tabla 3: Ejemplo de uso del comodín.

A continuación se muestra un ejemplo de un punto de corte para la ejecución de un método, y cuyo momento de inyección es antes, después y en lugar de.

```
<pointcut_definition>
      <time>before</time> <time>after</time><time>around</time>
      <joinpoint_type>
            <methodexecution>
                  <method_signature>
                        <return_type>
                              <type_name>*</type_name>
                        </return_type>
                        <qualified_method_name>
                              <qualified_class>
                                    <namespace>
                                    <type_name>*</type_name>
                                    </namespace>
                                    <class>
                              <identifier_name>*</identifier_name>
                                    </class>
                              </qualified_class>
                              <name>
                              <identifier_name>*</identifier_name>
                              </name>
                        </qualified_method_name>
                  </method_signature>
            </methodexecution>
            </joinpoint_type>
</pointcut_definition>
```

Tabla 4: Ejemplo de definición de un punto de corte.

4.8.2 Definición de un Aspecto Estático

Para definir un aspecto estático se debe indicar dónde está el código del *advice* que se va a inyectar en la aplicación, y el punto o los puntos de corte que van a seleccionar las sombras de los puntos de enlace donde se va a realizar esa inserción. se definen de forma similar a un punto de corte, momento de inyección y lista de puntos de corte (pueden ser 1 o más). Pero además se debe dar un nombre al aspecto e indicar donde se encuentra el código que será inyectado. Todo esto se agrupa mediante el elemento <*advice_definition*>.

Los elementos que se deben definir son los siguientes:

- Nombre del aspecto <*name*>.
- Localización del fichero donde se encuentra el aspecto <*assembly*>. Se puede poner la ruta completa donde se encuentra el fichero, en caso de que esa ruta no coincida con el directorio donde se encuentra el JPI, y el fichero de aspectos no exista donde está el JPI, el fichero con los aspectos estáticos será copiado al directorio donde está JPI.
- Espacio de nombres en caso de que exista y nombre de la clase <*type*>.
- Nombre del método que será insertado como aspecto <*behaviour*>.

- Prioridad del aspecto <*priority*>. Así se puede dar un peso indicando el aspecto estático más prioritario.

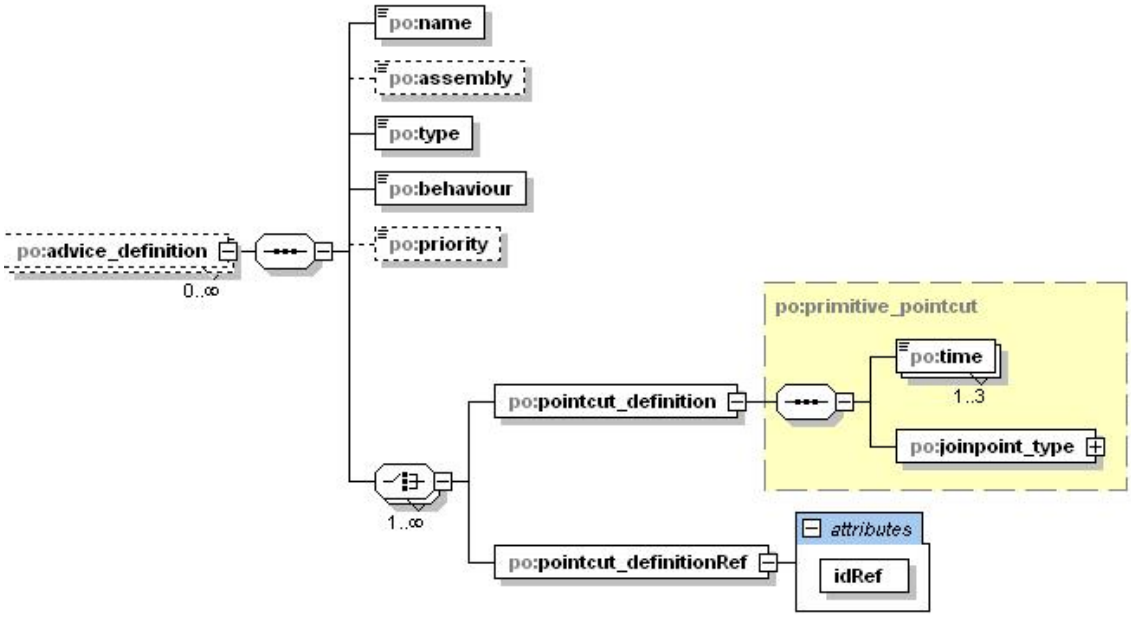


Figura 25: Definición de un aspecto estático.

```
<advice_definition>
      <name>ATMMachineSecurity</name>
      <assembly>Test5StaticAspect.dll</assembly>
      <type>Test5StaticAspect.Test5StaticAspect</type>
      <behaviour>verifySecurity</behaviour>
      <priority>3</priority>
      <pointcut_definition>
            <time>around</time>
            <joinpoint_type>
                  <methodcall>
                        <method_signature>

      <return_type><type_name>*</type_name></return_type>
                                    <qualified_method_name>
                                          <qualified_class>

      <namespace><type_name>*</type_name></namespace>

      <class><identifier_name>*</identifier_name></class>
                                          </qualified_class>

      <name><identifier_name>withdrawMoney</identifier_name></name>
                                    </qualified_method_name>
                        </method_signature>
                  </methodcall>
            </joinpoint_type>
      </pointcut_definition>
</advice_definition>
```

Tabla 5: Ejemplo de definición de un aspecto estático.

4.8.3 Reutilización de los Puntos de Corte

```
<pointcut_definition id="pcPayment">
    …
    …
    …
</pointcut_definition>
<advice_definition>
      <name>StaticAspectLogger</name>
      <assembly>Test6StaticAspect.dll</assembly>
      <type>Test6StaticAspect.Test6StaticAspect</type>
      <behaviour>loggingConcern</behaviour>
      <priority>1</priority>
      <pointcut_definitionRef idRef="pcPayment"/>
</advice_definition>
```

Tabla 6: Ejemplo de reutilización de un punto de corte.

Para la reutilización de los puntos de corte que se hayan definido para la inyección de código dinámicamente en los puntos de enlace, se ha incorporado un atributo opcional *id* al elemento <*pointcut_definition*>. Este atributo permite asignar un identificador único al punto de corte. Sobre este atributo se ha definido una regla de integridad referencial para evitar que dos puntos de corte diferentes tengan el mismo valor.

Posteriormente al definir un aspecto estático se puede utilizar el elemento <*poincut_definitionRef*> con su atributo obligatorio *idRef* para hacer referencia al punto de corte previamente definido. Sobre este atributo se ha definido otra regla de integridad referencial para evitar que se usen valores que no se han definido en la parte de puntos de corte.

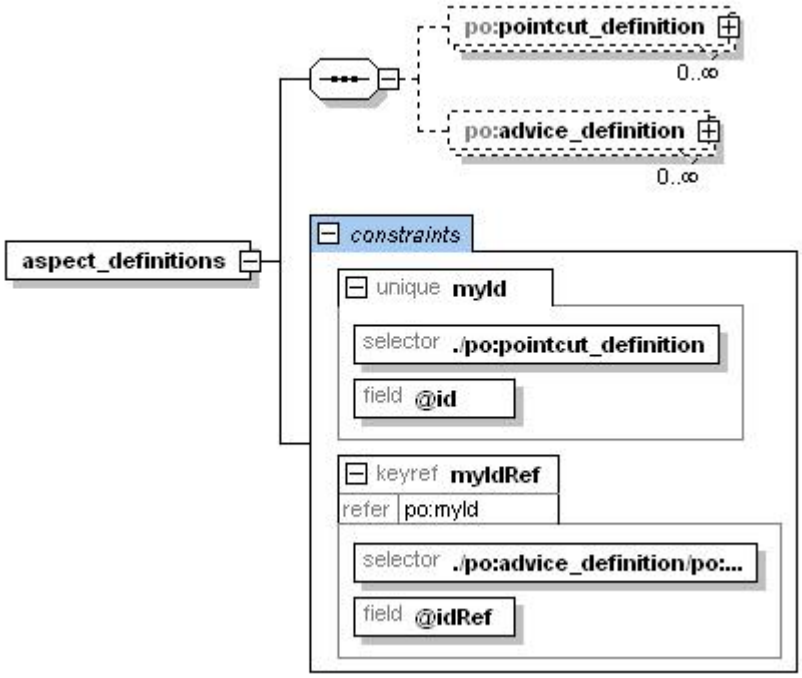


Figura 26: *Constraints* sobre los atributos.

5 Metodología de Trabajo

El objetivo principal de esta investigación es la demostración de que el uso combinado de aspectos tejidos estática y dinámicamente permite obtener aplicaciones con un equilibrio óptimo, entre rendimiento y flexibilidad ante posibles cambios o modificaciones en tiempo de ejecución.

Para demostrar esto se ha implementado una plataforma orientada a aspectos (DSAW) que permite los dos tipos de tejidos, tanto de forma aislada como de forma combinada. El sistema implementado, junto con múltiples ejemplos, se puede descargar de forma gratuita de la dirección Web *http://www.reflection.uniovi.es*.

Una vez que la plataforma ya ha sido implementada, se ha evaluado. Esta evaluación ha tenido dos vertientes, por un lado la comparación entre el tejido estático y el tejido dinámico realizado por la propia plataforma. Por otro lado, se ha comparado DSAW con otras plataformas orientadas a aspectos existentes, tanto a nivel comercial como a nivel académico.

5.1 Benchmarks

Para demostrar que este sistema proporciona un mejor rendimiento que otros, se deberían utilizar *benchmarks* genéricos, así se podrían realizar mediciones que permitiesen comparaciones entre este sistema y otros de forma estándar. En la actualidad estos *benchmarks* genéricos no se encuentran disponibles, y existe un debate de cómo se pueden hacer estas mediciones [Haupt04] [Dufour04].

Como muchos de los sistemas AOP trabajan sobre la plataforma Java, se están reutilizando *benchmarks* de carga tales como SPECjvm98 [Specjvm98], Java Grande [JavaGrande08] o XSLTMark [XSLTMark08]. Estos *benchmarks* no pueden ser aplicados en nuestro sistema en esta fase del proyecto, ya que la herramienta ha sido implementada sobre la plataforma .Net, y no se disponen actualmente de recursos para realizar una migración de estos *benchmarks* a ese entorno.

Otra corriente para la prueba de los sistemas AOP, es la utilización de aspectos complejos, cuidadosamente analizados y diseñados, que juegan un importante papel en la aplicación (*production aspects*). Ejemplos de este tipo de mediciones son la implementación de un sistema de control de Ingeniería (TCS) [Diaz01], la refactorización entera de una implementación de un ORB [Zhang03], o la extensión del procesador Xalan XSLT [Xalan08] con la funcionalidad de traza. En contraste con esta corriente, están las mediciones que se han realizado utilizando casos de prueba simples (*toy examples*). Ante las restricciones de tiempo en este trabajo, se ha optado por realizar un grupo de casos de pruebas simples para realizar las mediciones.

5.2 Evaluación del Tipo de Tejido de DSAW

Para la comparación de los dos tipos de tejidos, que se pueden realizar con DSAW, se va a comprobar la homogeneidad de los aspectos que se inyectan de forma

estática y los que se inyectan de forma dinámica. Como se marcó en los objetivos del capítulo 2 es necesario que no se deban realizar modificaciones para cambiar el momento de tejido de un aspecto, es decir el paso de un aspecto entre los dos tipos de tejido debe ser transparente para el desarrollador.

Otros parámetros que se han utilizado para comparar los dos tipos de tejido son el rendimiento en tiempo de ejecución y el consumo de memoria. Aunque claramente el tejido estático es más eficiente que el dinámico, medir esa diferencia da una referencia valiosa de cómo es posible mejorar una aplicación al cambiar el momento de tejido de un aspecto en aquellos casos que sea posible.

Para evaluar el rendimiento en tiempo de ejecución, se ha instrumentado el código añadiendo contadores para obtener el valor del reloj del procesador. Se ha medido la diferencia del valor entre el inicio y el fin, para así obtener el tiempo total de ejecución de cada aplicación sobre cada punto de enlace y los aspectos inyectados. Se ha tenido en cuenta que el sistema no inyecte aspectos sobre estas funciones de medida.

Para suprimir el coste de generar código nativo por el compilador JIT, al iniciar la aplicación se usa una sola vez la primitiva del punto de enlace a medir. Esta invocación inicial no es tomada en cuenta para la evaluación. Por lo tanto, el método de medición seguido ignora el tiempo requerido para dinámicamente generar código nativo por el compilador JIT sobre la máquina virtual.

Para medir el consumo de memoria de la aplicación (sin aspectos y con aspectos inyectados estáticamente) no existen posibles puntos de controversia. Sin embargo, la medición de la memoria de una aplicación con aspectos inyectados dinámicamente puede dar lugar a opiniones discrepantes. Esto es así, porque los aspectos inyectados dinámicamente son procesos independientes de la aplicación, por lo tanto existen al menos dos procesos independientes. En este trabajo se ha considerado la suma de todos los procesos (dos en las pruebas realizadas, aplicación y aspecto inyectado dinámicamente) para calcular el consumo estimado de memoria. No se ha considerado el servidor de aplicaciones.

5.2.1 Homogeneidad del Tipo de Tejido

Para favorecer la flexibilidad, se debe poder cambiar el momento de tejido de un aspecto sin modificar su código, el tejido estático y dinámico debe ser transparente para el desarrollador.

Para demostrar la consecución de este objetivo de homogeneidad, la evaluación que se ha planteado ha sido realizar un aspecto genérico de traza que explota muchas de las características AOSD posibles en este sistema, como por ejemplo acceso a los parámetros de un método, llamada reflectiva de ese método, traza, modificación del valor de un parámetro, etc. Así se pueden comparar las diferencias entre la implementación del aspecto de forma estática y de forma dinámica.

Además como el sistema implementado también puede inyectar aspectos sobre los accesos a lectura y escritura de un campo o propiedad, se ha realizado otro aspecto genérico para estos casos. Este aspecto genérico accede en modo lectura o escritura al valor de un campo o propiedad.

5.2.2 Rendimiento y Consumo de Memoria del Tipo de Tejido

Las pruebas de rendimiento, que se han realizado con la plataforma desarrollada actualmente, tienen como objetivo medir la diferencia en el rendimiento en tiempo de ejecución cuando se ejecuta un aspecto inyectado de forma estática, y cuando ese mismo aspecto se ha inyectado dinámicamente.

Para comparar esto, se ha realizado una aplicación que simula un cajero automático. Sobre esta aplicación se han creado dos escenarios, por un lado un escenario implica añadir una incumbencia para validar la seguridad antes de retirar el dinero de la cuenta. Por otro lado, se ha creado otro escenario donde se debe grabar mensajes de traza en un fichero antes de retirar el dinero de la cuenta.

Se han creado los siguientes micro *benchmarks* en C#, partiendo del **escenario 1** donde se debe verificar la seguridad antes de sacar el dinero:

- Aplicación cajero sin incumbencias. Saca el dinero sin validar seguridad.
- Aplicación cajero con incumbencia para verificar seguridad. Esta aplicación tiene dentro del código C# la llamada a la función de verificar seguridad.
- Aplicación cajero con un aspecto inyectado estáticamente para verificar seguridad. Este aspecto hace una llamada desde el punto de enlace *Method Call Around* hacia el método para sacar el dinero.
- Aplicación cajero con un aspecto inyectado dinámicamente para verificar la seguridad. Este aspecto hace una llamada desde el punto de *Method Call Around* hacia el método para sacar el dinero.

Para el **escenario 2**, donde se debe grabar mensajes de traza a fichero, se han creado los siguientes micro *benchmarks* en C#:

- Aplicación cajero. Saca el dinero sin grabar información de traza a fichero.
- Aplicación cajero con incumbencia de traza a fichero. Está aplicación tiene dentro del código C# la llamada a la función que graba los mensajes de traza a fichero.
- Aplicación cajero con aspecto inyectado estáticamente de traza a fichero. Este aspecto se añade en el punto de enlace *Method Call Before* y graba mensajes de traza a fichero.
- Aplicación cajero con aspecto inyectado dinámicamente de traza a fichero. Este aspecto se añade en el punto de enlace *Method Call Before* y graba mensajes de traza a fichero.

Cada uno de los micro *benchmarks* de los dos escenarios ha sido ejecutado 10.000 veces. Se ha medido el tiempo de ejecución de todas las iteraciones y el consumo de memoria. Solamente se ha inyectado un aspecto (*Before* o *Around*) en los dos casos de uso, tanto estática como dinámicamente.

5.3 Evaluación de Otras Plataformas

Para la evaluación de la plataforma implementada en relación con otras plataformas de tejido de aspectos, se han seleccionado un conjunto de sistemas representativos orientados a aspectos existentes en al actualidad. Con los sistemas seleccionados se han hecho comparaciones de forma cuantitativa, y comparaciones de forma cualitativa.

5.3.1 Sistemas Seleccionados

Los sistemas seleccionados han sido los siguientes:

- AspectJ 1.6.0. El sistema más extendido y con mayor base de usuarios [AspectJ08] sobre la plataforma JVM. Es una extensión que proporciona orientación a aspectos al lenguaje Java. Es un sistema estático aunque proporciona alguna característica dinámica tal como `cflow` o `within`, su tejedor no es ejecutado en tiempo de ejecución.
- PROSE 1.3.0. PROSE (PROgrammable extenSions sErvices) es un sistema que proporciona AOP de forma dinámica [Nicoara05]. Sólo permite realizar aplicaciones y aspectos en Java. Se ha evaluado su versión para JDK 1.5 sobre Windows XP con notificación de eventos sobre JVMDI/JVMTI.
- JAsCo 0.8.7. Sistema que soporta AOP de forma dinámica. Está orientado a dar soporte al software basado en componentes, y específicamente al campo de los Servidores Web [Suvee03]. Sólo permite realizar aspectos y aplicaciones en Java. La tecnología de JAsCo sobresale proporcionando integración dinámica y borrado de aspecto con una sobrecarga mínima en el rendimiento de la aplicación adaptada. Requiere la máquina virtual de Java 1.5. La plataforma JAsCo proporciona un rendimiento óptimo. La incorporación de software que realiza optimizaciones eficientes, como JUTTA (*Just-in-time combined aspect compilation*) [Wanderperren05] proporcionando un código optimizado de los puntos de enlace y la llamada a los *advices*, y de la librería *Java Programming Language Instrumentation Services API* (JPLIS) para optimizar las intercepciones que se hacen a los métodos, demuestran su efectividad en las mediciones que se han realizado.
- Rapier-LOOM.Net 2.21. Sistema que soporta AOP de forma dinámica [Köhne05]. Desarrollado sobre la plataforma .Net. El proyecto LOOM.Net tiene como objetivo investigar y promocionar el uso de AOP en el contexto de la plataforma .Net.
- Gripper-LOOM.Net 0.92. Sistema que soporta AOP de forma estática [Schult08]. También ha sido desarrollado sobre la plataforma .Net.
- Spring Java 2.5.5. Framework de desarrollo para la plataforma JVM que soporta AOP de forma estática [Spring08]. Corre sobre Java 1.4+. Ofrece un API para dinámicamente añadir y remover aspectos en tiempo de ejecución. Soporta tejido estático en tiempo de compilación y de carga, y tejido dinámico en tiempo de ejecución
- Spring .Net 1.1.2. Framework de desarrollo para la plataforma .Net que soporta AOP de forma estática [SpringNET08]. Está basado en el diseño hecho para la versión Java del framework de Spring. Proporciona una librería predefinida de aspectos fácil de usar con aspectos implementando

transacciones, logging, monitorización de rendimientos, cacheado, etc. Necesita la versión 1.2 del framework de la plataforma .Net.

- JBoss AOP 2.0.0.CR8. Plataforma para desarrollo AOP 100% Java que permite tejer en tiempo de compilación, en tiempo de carga y *hotswap* [JBossAOP08]. Requiere Java 1.5.

En esta evaluación cualitativa cuando se refiere a tejido dinámico, se está refiriendo a **tejido realmente dinámico**. Ésta es una importante característica que debe ser tomada en cuenta cuando se analizan los sistemas de tejido dinámicos [Suveé03]. A diferencia de muchas implementaciones AOP, el destejido o el retejido de aspectos en tiempo de ejecución debería ser posible, incluso en puntos de enlace donde no se tejió nada previamente. Esto significa que no existe acoplamiento real entre los aspectos y los componentes. Un aspecto debe ser capaz de adaptar una aplicación en ejecución, incluso sí éste fue creado posteriormente al inicio de la ejecución de esta aplicación. Implementaciones como Spring Java [SpringAOP08] o Spring Net [SpringNET08] se autodefinen como con tejido totalmente dinámico, pero si se hace un análisis más detallado se comprueba que no cumplen las condiciones necesarias para ser consideradas con tejido realmente dinámico.

5.3.2 Comparación Cuantitativa

Para realizar la comparación cuantitativa se ha realizado una pequeña aplicación con una función que tiene como parámetros tres números enteros, y devuelve un número entero. Su signatura es: `public int addInt(int a, int b, int c).`

Esta función se llama 10.000 veces desde el `Main`. A esta función se le ha inyectado un aspecto estática y dinámicamente en el punto de enlace *Method Call* en los momentos *Before*, *After* y *Around*. La función inyectada no hace nada excepto en el caso de *Around* donde ejecuta reflectivamente el método.

La aplicación ha sido implementada tanto en Java como en C# intentado que las dos implementaciones fuesen lo más parecidas. Para cada uno de los sistemas a comparar, también se ha intentado que la aplicación fuese lo mas parecida posible a las implementaciones de partida. Para el caso de los sistemas basados en Spring esto no se ha logrado, ya que la técnica de inyección de dependencias en la que se basa este framework de desarrollo ha obligado a cambiar el diseño de la aplicación.

Para el sistema PROSE no se ha podido calcular el tiempo de ejecución y su consumo de memoria para el caso de un punto enlace *Method Call* para el momento *Around*. No se ha conseguido con la versión proporcionada en su página Web para la plataforma Windows XP realizar esta funcionalidad.

Para el caso de JBoss AOP, sólo se ha calculado los parámetros de la evaluación cuantitativa para el momento *Around*. En este sistema no existe los momentos *Before* o *After*, aunque en la información publicada en su página Web se anuncia que la próxima versión a sacar del sistema si los soportarán. Este sistema permite tres tipos de tejido (tiempo de compilación, tiempo de carga y *hotswap*), para cada uno de estos tipos de tejido se calculó su rendimiento en tiempo de ejecución y su consumo de memoria.

Las mediciones se han realizado siguiendo los mismos criterios que se han aplicado para las hechas en el apartado anterior.

5.3.3 Comparación Cualitativa

Para la comparación cualitativa se han utilizado como parámetros de la evaluación los requisitos marcados en el trabajo de investigación que dio lugar a Ready AOP [Vinuesa07]. A estos requisitos se han añadido la capacidad de soportar tejido estático, homogeneidad entre tejido estático y dinámico, y permitir tejido simultáneo de aspectos inyectados estática y dinámicamente. También se ha modificado el requisito D.6 a los criterios de los aspectos, este requisito es la no imposición de condiciones a los aspectos, y la no necesidad de su código.

Los requisitos que se habían marcado en el trabajo previo se dividían en cuatro grandes bloques: generales, plataforma, aplicaciones y aspectos. La clasificación de estos requisitos junto con los nuevos parámetros en este trabajo es la siguiente:

A. Criterios Generales.
   1. Sistema estático. Se debe poder adaptar las aplicaciones de forma estática.
   2. 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 no sean necesarios. Debe existir un tejido realmente dinámico.
   3. Tejido estático y dinámico homogéneo. Se debe poder cambiar el momento de tejido de un aspecto sin realizar cambios sobre éste.
   4. Tejido simultaneó de aspectos estáticos y dinámicos. Se debe permitir tejer a la vez aspectos de forma estática y de forma dinámica.
   5. 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.
   6. Interacción entre distintos lenguajes de programación. Las aplicaciones y los diferentes aspectos que las adapten pueden estar implementados en distintos lenguajes de programación.
   7. Sistema de propósito general. El sistema debe estar diseñado para poder ser utilizado en cualquier tipo de problema.
   8. Amplio conjunto de puntos de enlace. El conjunto de puntos de enlace que soporté el sistema debe ser todo lo amplio posible.
   9. Formas de adaptar un punto de enlace. Se evaluarán los posibles tiempos o formas de adaptar un punto de enlace que soporte el sistema.
   10. 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 correspondientes a 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 las aplicaciones. Una aplicación puede ser adaptad de forma simultánea por varios aspectos.
2. Mecanismo de coordinación de múltiples aspectos. Cuando varios aspectos adaptan 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 estándares 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.
6. No imposición de condiciones a los aspectos, y no necesitar su código fuente. Cualquier librería implementada por un lenguaje soportado por el sistema debe poder ser usada por el aspecto. Como una consecuencia de esto, no se necesitaría el código fuente de esa librería para acceder a la funcionalidad de sus aspectos.

Para realizar esta evaluación se han agrupado los dos tejedores, Rapier-LOOM.Net y Gripper-LOOM.Net, bajo el epígrafe LOOM.Net. Aunque no sean homogéneos, se considera que se complementan entre sí, y las características que no sean contemplados por uno pero si por el otro deben valorarse a los dos.

Siguiendo este mismo criterio, también se han agrupado los frameworks de desarrollo de Spring para Java y para .Net bajo el epígrafe Spring. Al igual que en el caso anterior se complementan entre sí.

5.4 Estimación del Consumo de Memoria

Tanto para comparar los dos tipos de tejidos de DSAW entre sí, como para comparar DSAW con los otros sistemas orientados a aspectos, se han realizado mediciones para estimar la memoria empleada por el entorno para soportar el sistema con aspectos inyectados estática o dinámicamente.

Para las aplicaciones que tienen aspectos inyectados dinámicamente, se ha considerado como memoria consumida tanto la memoria de la aplicación, como la memoria de los aspectos inyectados dinámicamente. Por lo tanto, se han sumado todos los consumos, y se ha calculado la media aritmética para calcular el consumo en este caso.

Todos los datos de los *benchmark*s relativos al consumo de memoria han sido obtenidos utilizando el monitor de rendimiento de Windows XP. Se ha medido el tamaño máximo del conjunto de memoria de trabajo usada por el proceso desde su inicio (la propiedad *PeakWorkingSet*). El conjunto de trabajo de un proceso es el

conjunto de páginas de memoria actualmente visibles al proceso en la memoria física RAM. Estas páginas son residentes y disponibles para que una aplicación las use sin disparar un fallo de página. El conjunto de trabajo incluye datos privados y compartidos. Los datos compartidos comprenden las páginas que contienen todas las instrucciones que el proceso ejecuta, incluyendo instrucciones desde los módulos de los procesos y las librerías del sistema.

5.5 Equipamiento

Todos los tests han sido ejecutados sobre un sistema Intel Pentium M Proccesor 1.80 Ghz 594 Mhz, 1.00 Gb de RAM ligeramente cargado corriendo Windows XP Home Edition Versión 2002 SP 2. La implementación de .Net utilizada ha sido la versión 2.0.50727 del Framework .Net de Microsoft. Para evaluar la media de los porcentajes, ratios y órdenes de magnitud, se ha usado la media geométrica.

6 Resultados Obtenidos

6.1 Evaluación del Tipo de Tejido de DSAW

6.1.1 Homogeneidad

Al realizar un aspecto genérico de traza (acceso a los parámetros de un método, llamada reflectiva de ese método, traza etc.) que explote todas las características posibles de este sistema, tanto de forma estática como dinámica el código implementado es el mismo. No se ha necesitado cambiar nada de código para cambiar el momento de tejido de este aspecto.

Tampoco se necesito cambiar nada para el aspecto genérico realizado para acceder en lectura o escritura cobre un campo o propiedad. El código del aspecto fue el mismo tanto al inyectarlo de forma estática como dinámica.

6.1.1.1 Discusión de la Homogeneidad

Aunque no se debe cambiar el código de un aspecto para pasar de inyectarlo de forma estática a dinámica, o viceversa, aún para tejer el aspecto dinámicamente es necesario un programa lanzador genérico. Este programa se ocupa de registrar el aspecto y desregistrarlo en el servidor de aplicaciones, e inicializar el aspecto.

Esta labor puede ser realizada por los desarrolladores fácilmente, ya que partiendo de una clase esqueleto sólo se deben cambiar los nombres relativos a la clase que encapsula el aspecto. Sin embargo, para conseguir una mayor transparencia desde el punto de vista del desarrollador, se ha implementado una utilidad que generar de forma automática esto ficheros *dispatcher* o *stub*. De esta forma el desarrollador no tendría que realizar ninguna operación para pasar cambiar el momento de tejido del aspecto. Estos ficheros actuarían como mediadores entre la aplicación y los aspectos que las adaptan dinámicamente.

6.1.2 Rendimiento y Consumo de Memoria

6.1.2.1 Around (Escenario 1)

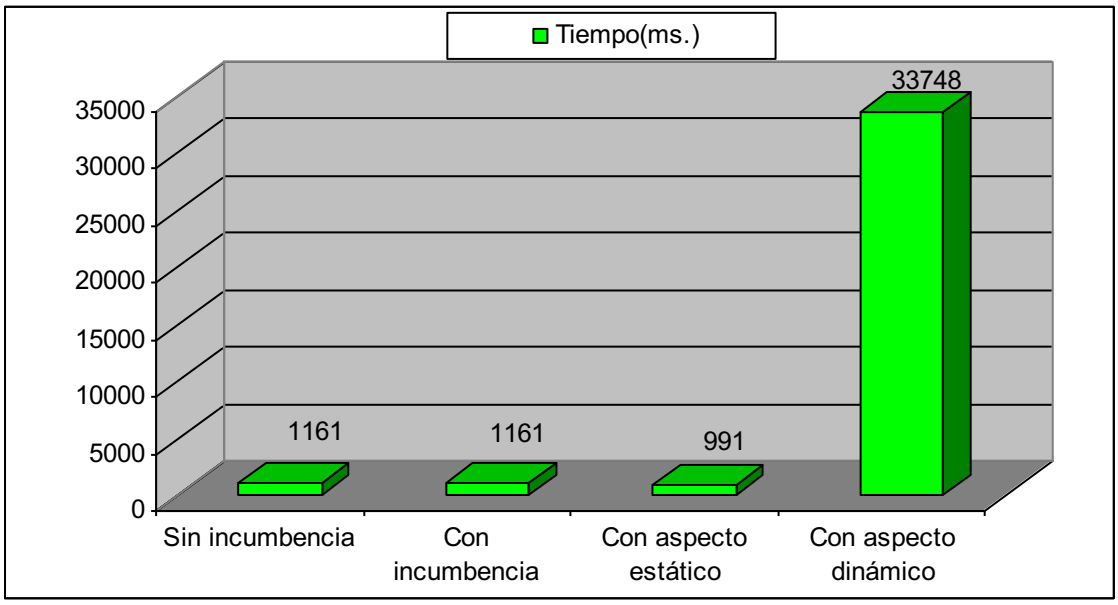


Figura 27: Rendimiento en *Method Call Around* (Escenario 1).

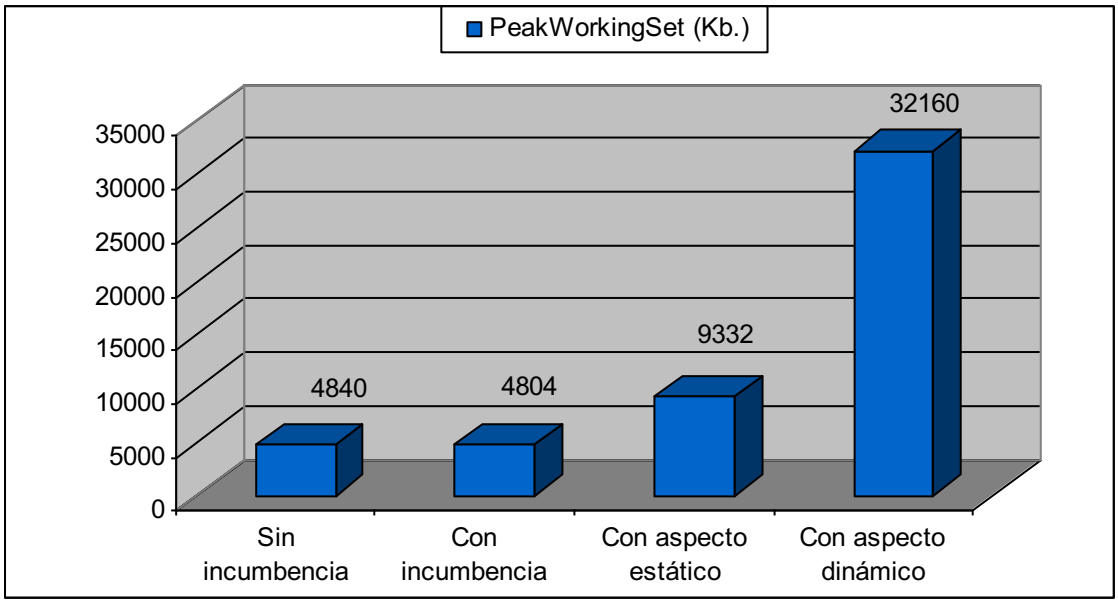


Figura 28: Memoria consumida en *Method Call Around* (Escenario 1).

6.1.2.2 *Before (Escenario 2)*

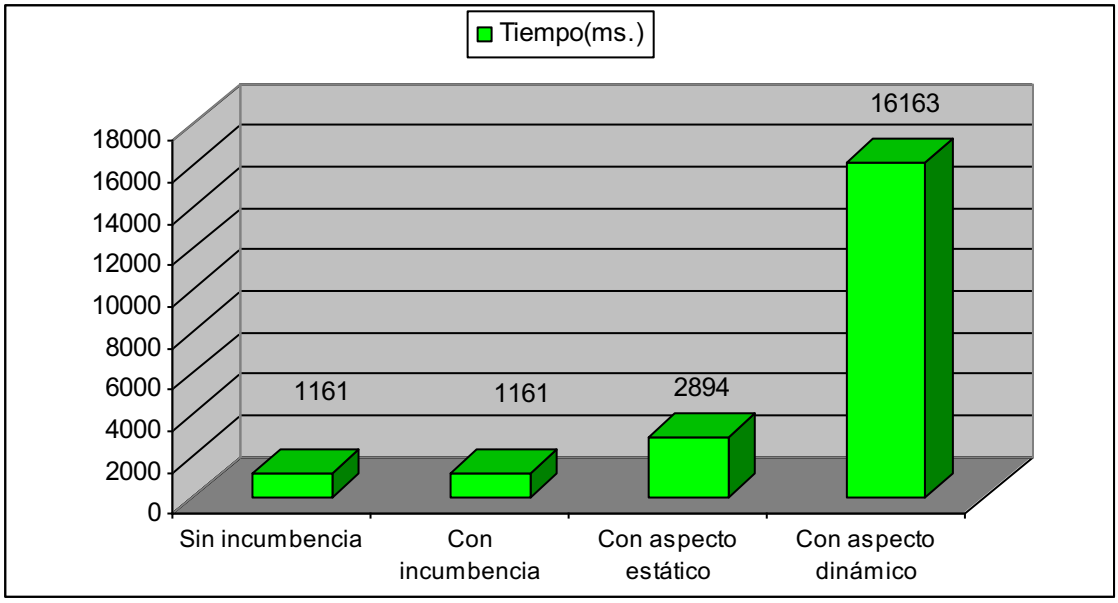


Figura 29: Rendimiento en *Method Call Before* (Escenario 2).

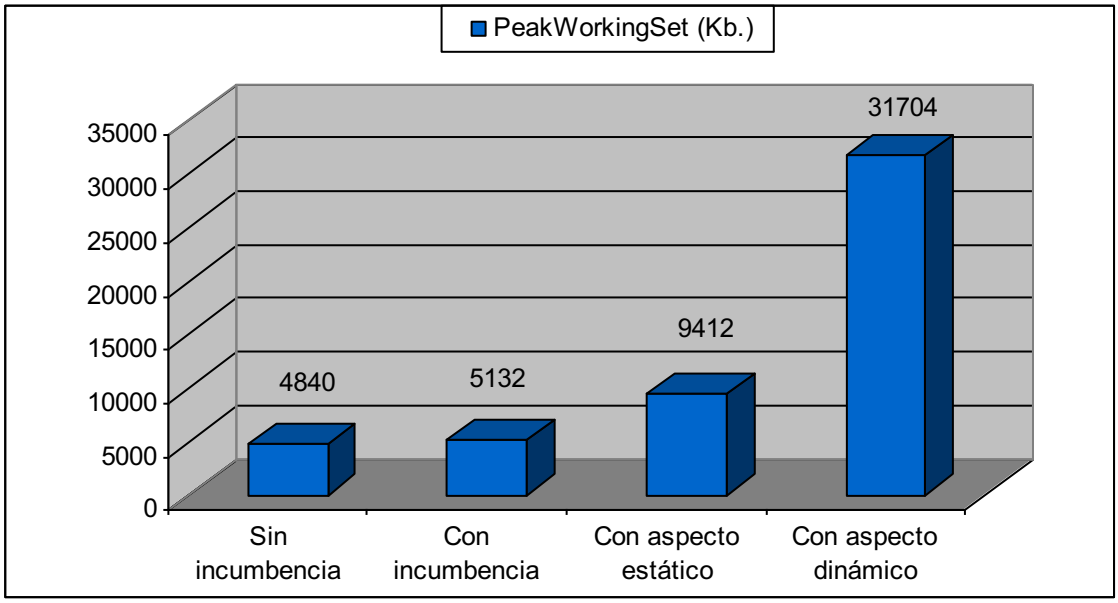


Figura 30: Memoria consumida en *Method Call Before* (Escenario 2).

6.1.2.3 *Interpretación de los Resultados de Rendimiento*

Aunque en los *benchmarks* realizados sólo se ha inyectado un aspecto dinámicamente sobre un punto de enlace, la aplicación resultante ha tenido una penalización considerable, 2900% en el caso de *Around* y 1392% en el caso de *Before*. Sin embargo, la inyección del mismo aspecto de forma estática apenas penaliza la ejecución de la aplicación. Se comprueba así que ejecutar aspectos inyectados dinámicamente sobre una aplicación supone una penalización en el rendimiento muy importante respecto a ejecutar el mismo aspecto inyectado estáticamente.

Se debe remarcar, como los aspectos tejidos dinámicamente en puntos de enlace de tipo *Method Call* en el momento *Around* llamando reflectivamente al método

original (un *proceed*), pueden provocar penalizaciones de casi el doble de tiempo que otros aspectos inyectados dinámicamente sin explotar esta funcionalidad. Debido a esto, este tipo de llamadas a métodos reflectivamente deberían de ser realizadas siempre desde aspectos tejidos estáticamente, y evitar su uso en escenarios dinámicos.

6.1.2.4 Discusión del Consumo de Memoria

Un punto anormal respecto a estas mediciones, es la diferencia existente entre la memoria consumida por una aplicación sin aspectos, y una aplicación con aspectos inyectados estáticamente. Existe un consumo de un 100% más de memoria. Se ha analizado el código, y se ha llegado a la conclusión de que esa memoria se consume para realizar el registro de la aplicación en el servidor de aplicaciones, y de esta forma poder comunicarse con los aspectos dinámicos. Se han hecho mediciones de consumo de la memoria de los tests del Escenario 1 obteniendo los siguientes datos:

- 4.840 kilobytes. Aplicación sin aspectos.
- 5.736 kilobytes. Aplicación con aspectos estáticos pero sin código IL para registrarse en el servidor de aplicaciones.
- 9.340 kilobytes. Aplicación con aspectos estáticos y con código IL para registrarse en el servidor de aplicaciones.

Esto ha proporcionado un punto de mejora del sistema implementado. El código del registro es necesario para permitir la ejecución de aspectos inyectados dinámicamente, pero si al post-procesar una aplicación para usar este sistema no se selecciona ningún punto de enlace para una posible adaptación dinámica posterior, se tendría una aplicación que no necesitaría el servidor de aplicaciones para su ejecución. Por lo tanto en estos casos se ha eliminado la parte de registro, así no sólo se reduce el consumo de memoria sino que el sistema implementado se puede utilizar más fácilmente.

6.2 Evaluación de Otras Plataformas

6.2.1 Evaluación Cuantitativa

6.2.1.1 Rendimiento y Consumo de Memoria

Para realizar la evaluación del sistema DSAW comparándolo con otras plataformas, se ha tomado como sistema de referencia AspectJ. Se han asignado a los valores obtenidos de los parámetros de medición (tiempo de ejecución y rendimiento) de esta plataforma un valor de 1.

Los valores obtenidos para los otros sistemas se han comparado mediante la inversa con los valores de referencia, en este caso los de AspectJ. Un valor superior a 1 indica que el sistema se comporta mejor que AspectJ respecto a este parámetro, y un valor inferior indica que el sistema se comporta peor respecto AspectJ.

6.2.1.1.1 Method Call Before

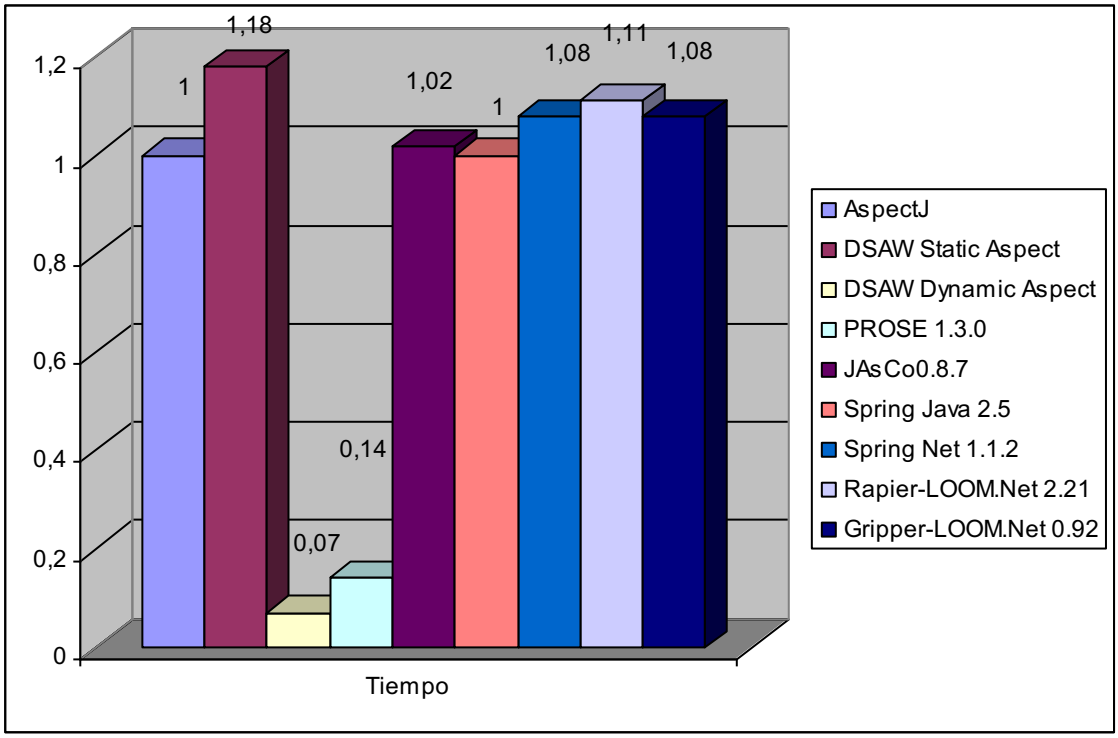


Figura 31: Inversa del rendimiento en *Method Call Before*.

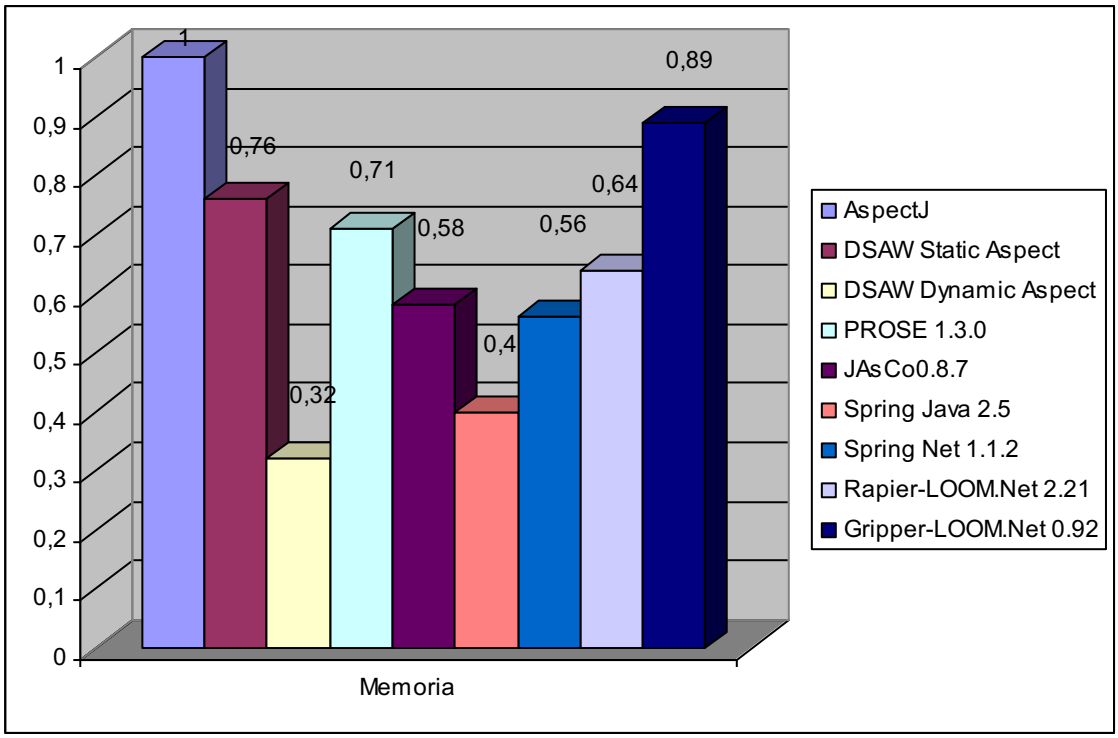


Figura 32: Inversa del consumo de memoria en *Method Call Before*.

6.2.1.1.2 Method Call After

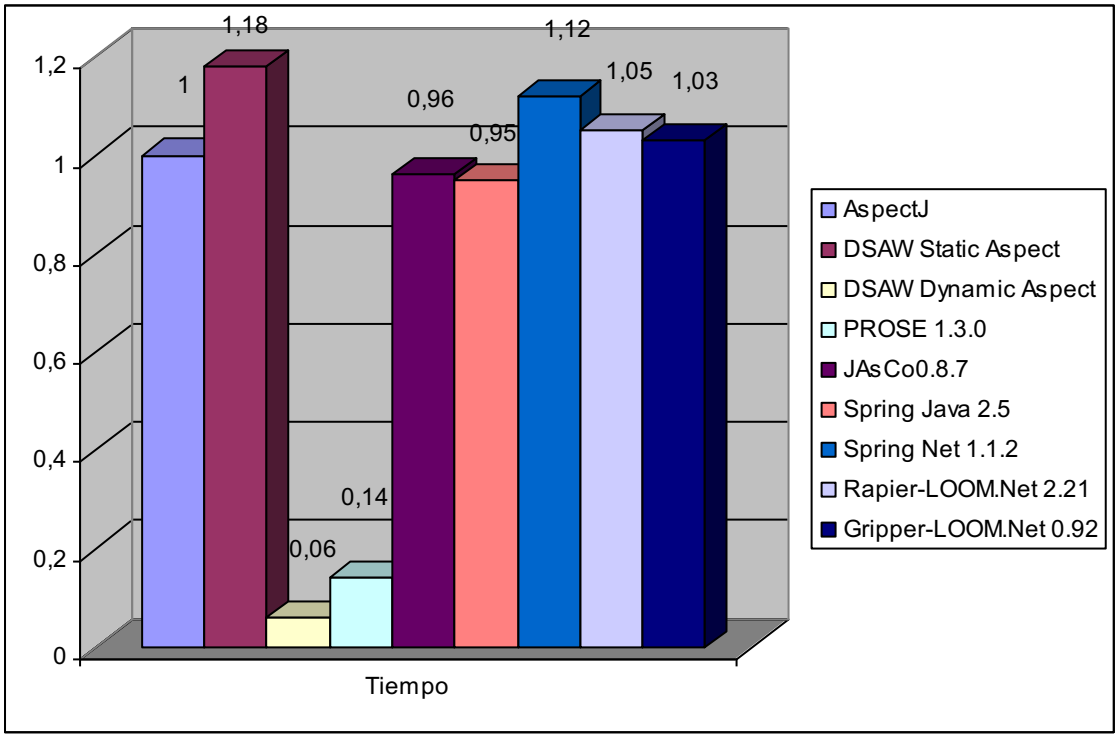


Figura 33: Inversa del rendimiento en *Method Call After*.

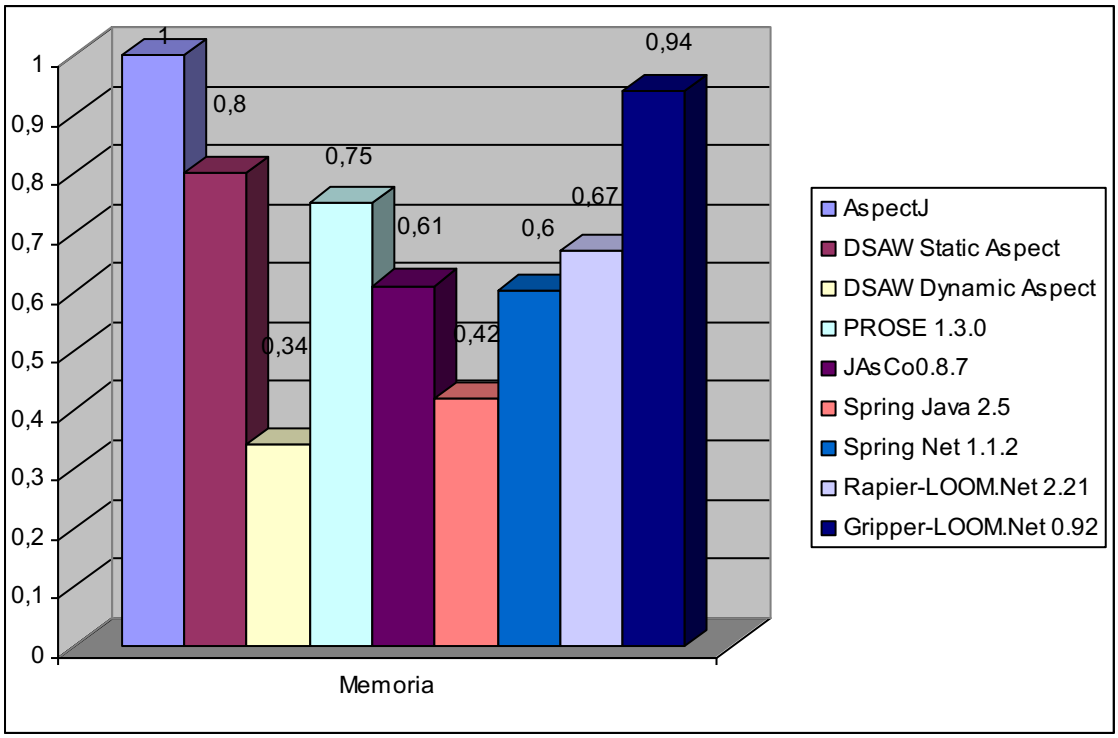


Figura 34: Inversa de la memoria consumida en *Method Call After*.

6.2.1.1.3 Method Call Around

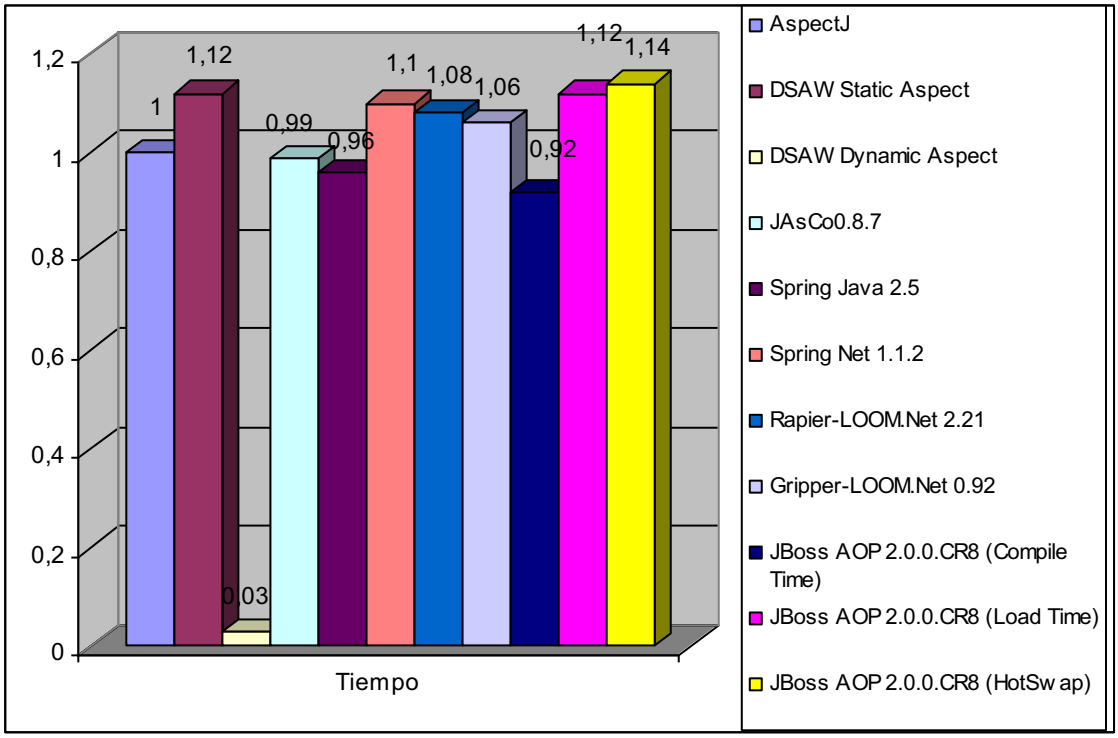


Figura 35: Inversa del rendimiento en *Method Call Around*.

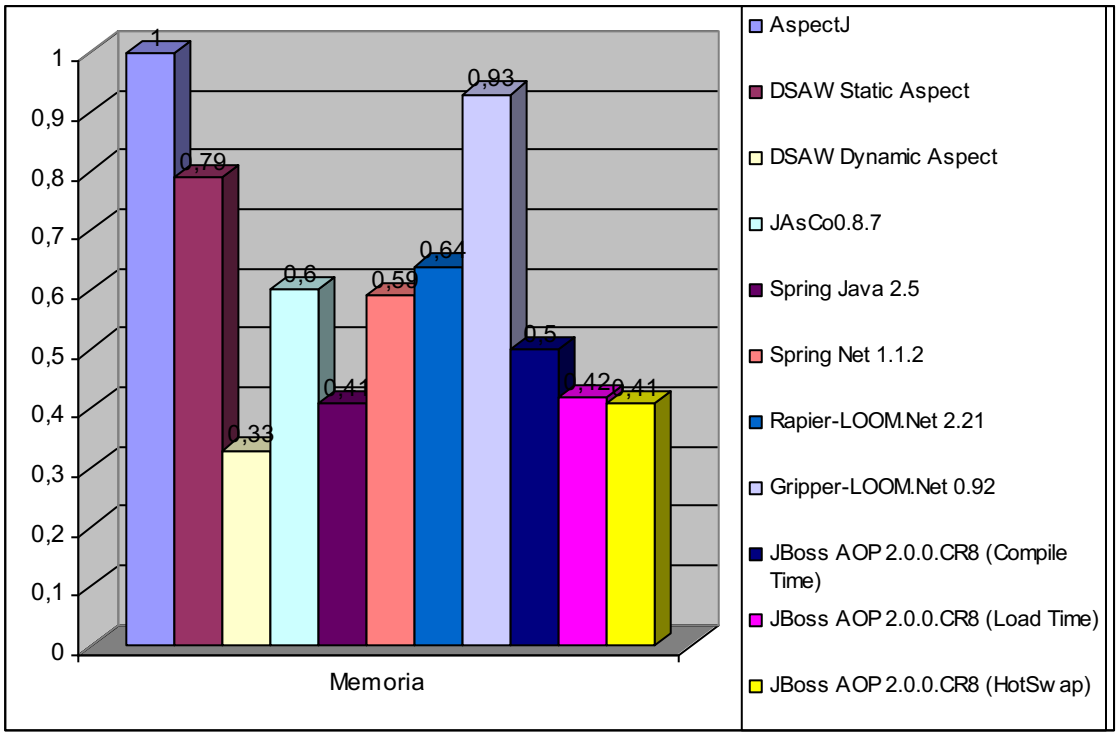


Figura 36: Inversa de memoria consumida en *Method Call Around*.

6.2.1.2 Discusión de Rendimiento y Consumo de Memoria

6.2.1.2.1 Tejido Estático vs. Dinámico

Viendo los gráficos anteriores con los datos de rendimiento, se puede ver como DSAW tiene un mejor rendimiento cuando trabaja con tejido estático. Este hecho puede también ser contrastado si se analizan los datos de las otras plataformas que realizan adaptación **realmente dinámica**. Todos ellas (DSAW Dynamic, PROSE, JAsCo y JBoss) obtienen peor rendimiento en tiempo de ejecución que AspectJ; mientras que las plataformas que no proporcionan ese grado de dinamismo (DSAW Static, Spring y Rapier-LOOM.Net) obtienen un mejor rendimiento. Se debe remarcar también que el tejedor dinámico de DSAW consume menos memoria que su tejedor estático. Esto es así porque los aspectos son cargados *just-in-time*.

6.2.1.2.2 Tejido Estático

DSAW ofrece un rendimiento óptimo en tiempo de ejecución para aspectos inyectados estáticamente en comparación con los otros sistemas. Su rendimiento es un poco mejor que el sistema de referencia, AspectJ. Su consumo de memoria es peor, pero como se ha comentado en la evaluación del tipo de tejido del apartado anterior, esto se debe a la memoria consumida para cargar la parte de registro en el servidor de aplicaciones. Esta parte se podría eliminar cuando sólo existe tejido estático mejorando este parámetro notablemente.

Comparando el rendimiento de DSAW con los otros sistemas estáticos (Spring Java 2.5, Spring Net 1.1.2, Gripper-LOM.Net 0.92 y JBoss AOP 2.0.0.CR8 con inyección en tiempo compilación o de carga), su rendimiento en tiempo de ejecución es bastante similar a ellos, y su consumo de memoria también.

El buen rendimiento en tiempo de ejecución, su consumo de memoria y el amplio conjunto de puntos de enlace que ofrece está implementación del tejedor estático DSAW, proporcionan un tejedor muy competitivo.

6.2.1.3 Tejido Dinámico

El rendimiento de DSAW cuando se utiliza con aspectos inyectados dinámicamente no es eficiente. El tejedor dinámico de DSAW obtiene el peor rendimiento en tiempo de ejecución, siendo el siguiente tejedor con peor rendimiento PROSE. Estas medidas difieren del rendimiento obtenido por el tejedor más reciente de JAsCo (sólo un 2% más lento que AspectJ), y el tejedor de JBoss (13% más lento que AspectJ).

Esta diferencia tan significativa es explicada por el mecanismo usado para interconectar los aspectos y los componentes. A diferencia de DSAW, los tres tejedores comentados localizan aplicaciones y aspectos en el mismo proceso. Para interconectarlos, PROSE usa el API de notificación de JVMDI/JVMTI; JBoss emplea el API de Java Dynamic Proxy Classes; y JAsCo utiliza los nuevos agentes Java para instrumentar programas, sumados junto con el paquete `java.lang.insturment` de la versión 1.5. Sin embargo, DSAW crea un proceso para cada aplicación y cada aspecto, y utiliza el API de .Net Remoting para interconectar los diferentes procesos. Aunque esta aproximación implica una incuestionable penalización en tiempo de ejecución, también proporciona dos beneficios. El primero es la robustez. Cuando un aspecto “casca” o se “cuelga” en tiempo de ejecución –cosa bastante común si un programador sigue una metodología de desarrollo basada en AOSD y *edit-and-continue*-, el resto de la aplicación podría continuar ejecutándose, debido a que están corriendo en procesos diferentes. El segundo beneficio es que DSAW es una plataforma

AOSD distribuida, porque los canales de .Net Remoting han sido diseñados para entornos distribuidos.

Después de este análisis, se deduce que el enfoque de DSAW para tejido dinámico es apropiado para escenarios *edit-and-continue*, pero es necesario enfocar el trabajo futuro incluyendo un enfoque similar al implementado por JAsCo. De este modo, .Net Remoting podría ser usado en tiempo de desarrollo, y una técnica de adaptación dinámica *in process* sería usada para las aplicaciones desplegadas que necesitasen de forma obligatoria tejido dinámico.

6.2.2 Evaluación Cualitativa

6.2.2.1 Datos y Gráficos Obtenidos

Criterios Generales							
Criterio	DSAW	Aspecto	PROSE	JAsCo	Spring	LOOM.Net	JBoss AOP
A.1	1	1	0	0	1	1	1
A.2	1	0	1	1	0,4	0,2	0,5
A.3	1	0	0	0	0,6	0,4	0,6
A.4	1	0	0	0	0	0	0
A.5	1	0	0	0	0,5	1	0
A.6	1	0	0	0	0,6	1	0
A.7	1	1	1	0,5	1	1	1
A.8	1	1	0,5	0,5	0,5	0,5	1
A.9	1	1	0,5	1	1	1	0,5
A.10	1	1	1	1	1	1	1
Sistemas	10	5	4	4	6,6	7,1	5,6

Tabla 7: Datos de los criterios generales.

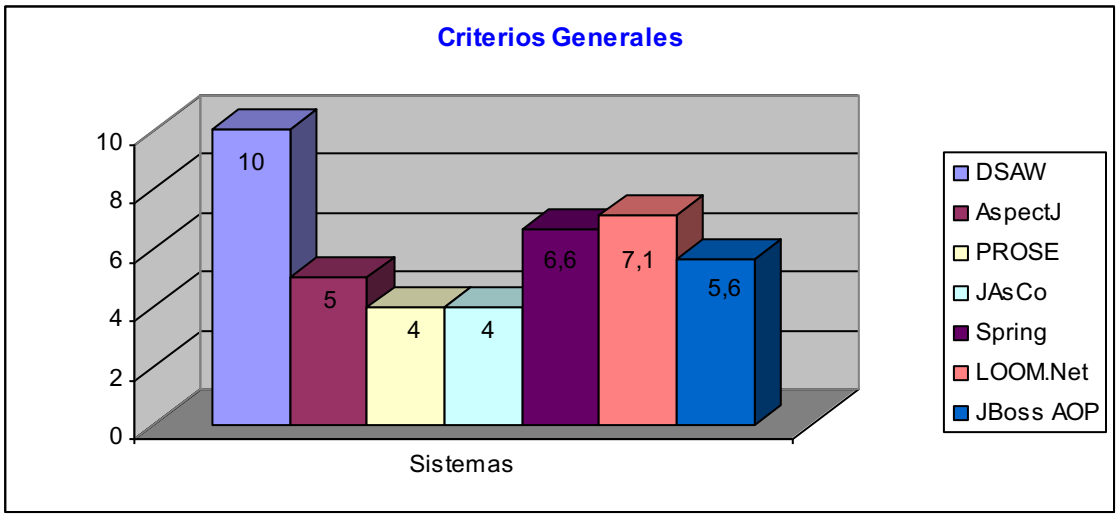


Figura 37: Criterios generales.

Criterios de la Plataforma							
Criterio	DSAW	AspectJ	PROSE	JAsCo	Spring	LOOM.Net	JBoss AOP

B.1	1	1	1	1	1	1	1
B.2	1	1	1	1	1	1	1
B.3	1	1	1	1	1	1	1
B.4	1	1	0	1	1	1	1
Sistemas	4	4	3	4	4	4	4

Tabla 8: Datos de los criterios de la plataforma.

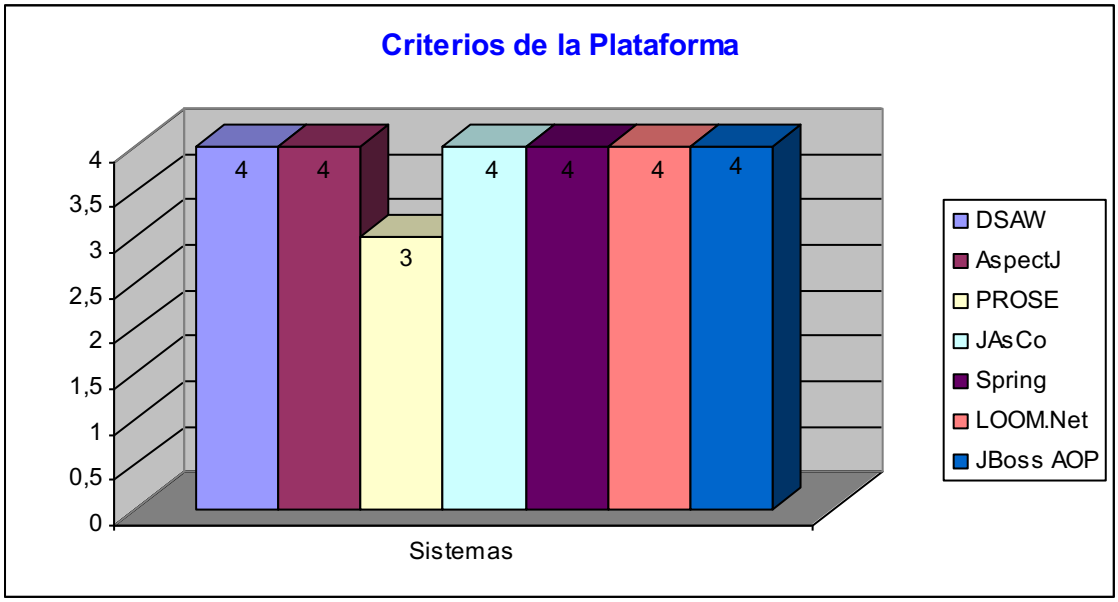


Figura 38: Criterios de la plataforma.

Criterios de las Aplicaciones							
Criterio	DSAW	AspectJ	PROSE	JAsCo	Spring	LOOM.Net	JBoss AOP
C.1	1	1	0,8	1	0	0	1
C.2	1	1	1	1	1	0	1
Sistemas	2	2	1	2	1	0	2

Tabla 9: Datos de los criterios de las aplicaciones.

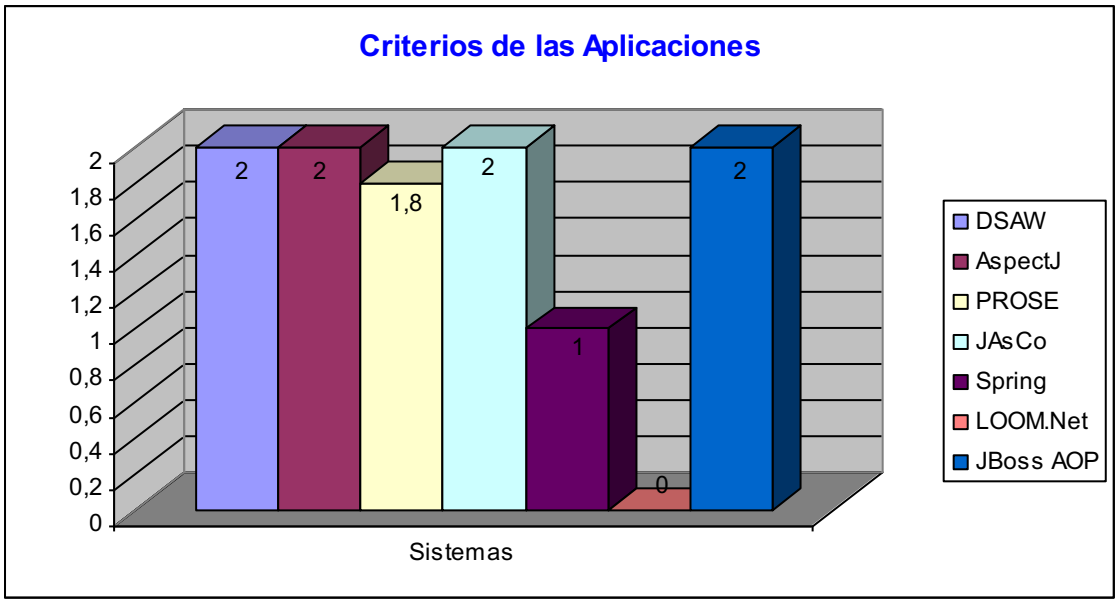


Figura 39: Criterios de las aplicaciones.

Criterios de los Aspectos							
Criterio	DSAW	AspectJ	PROSE	JAsCo	Spring	LOOM.Net	JBoss AOP
D.1	1	1	1	1	1	1	1
D.2	1	1	1	1	1	0,3	1
D.3	1	0	0	1	0	0	0
D.4	1	0	1	0	1	1	1
D.5	1	0	0	0	1	0	1
D.6	1	0	0	0	0	0	0
Sistemas	6	2	3	3	4	2,3	4

Tabla 10: Datos de los criterios de los aspectos.

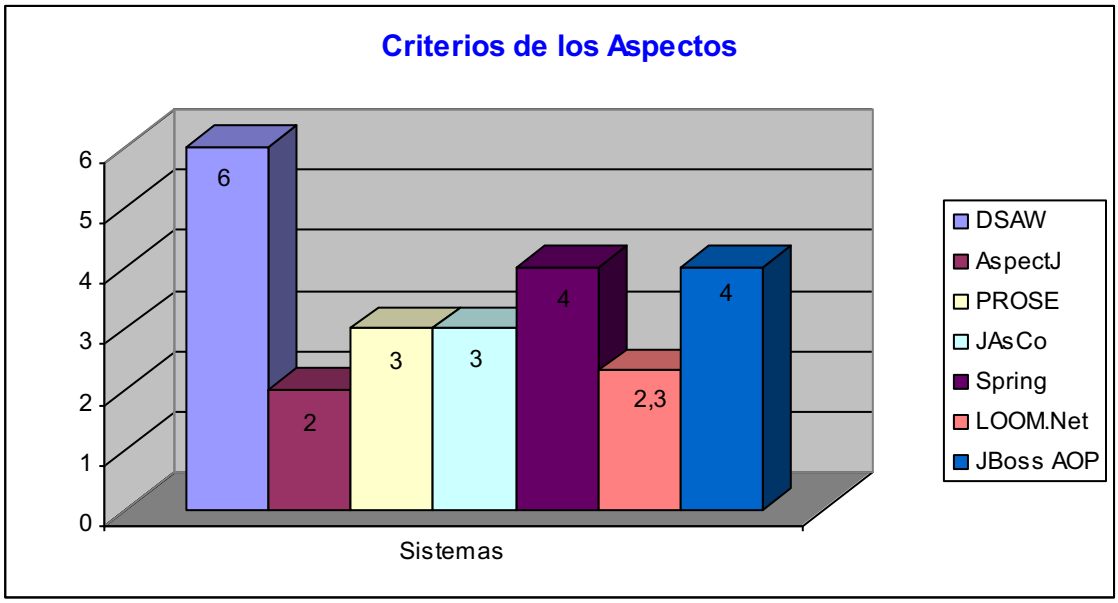


Figura 40: Criterios de los aspectos.

Evaluación Global							
Criterio	DSAW	AspectJ	PROSE	JAsCo	Spring	LOOM.Net	JBoss AOP
A	10	5	4	4	6,6	7,1	5,6
B	4	4	3	4	4	4	4
C	2	2	1,8	2	1	0	2
D	6	2	3	3	4	2,3	4
Sistemas	22	13	11,8	13	15,6	13,4	15,6

Tabla 11: Datos de la evaluación global.

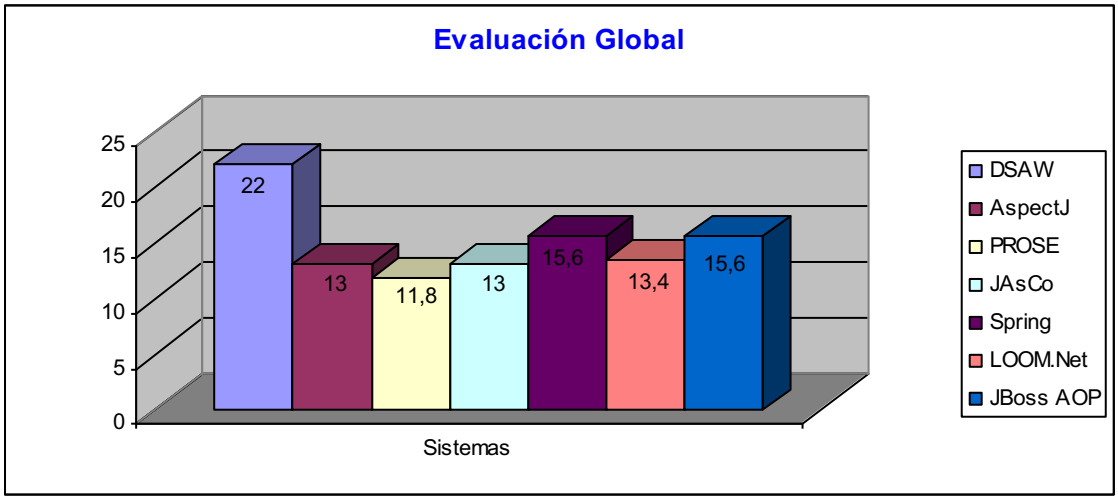


Figura 41: Evaluación global.

6.2.2.2 Justificación de la Evaluación Cualitativa

Se describe a continuación, de forma somera, la evaluación de los distintos criterios seleccionados previamente para cada uno de los sistemas estudiados.

A. Criterios Generales.

1. Sistema estático. PROSE y JAsCo son los únicos sistemas que no permiten tejer aspectos estáticamente.
2. Sistema completamente dinámico. DSAW, PROSE y JAsCo son los únicos sistemas que permiten adaptar una aplicación de forma totalmente dinámica. Rapier-LOOM.Net necesitan que los aspectos estén definidos previamente a la ejecución de la aplicación. Spring (Java y .Net) retrasa la carga de los aspectos hasta la carga de su fichero de configuración, una vez realizada está carga sólo se puede tejer un nuevo aspecto, destejer un aspecto o retejer mediante el código programado en la aplicación previamente (*advisors*). JBoss Server proporciona facilidades para el despliegue y el repliegue en caliente de aspectos o aplicaciones, pero su sistema de desarrollo JBoss AOP sólo posterga la carga de las clases de los aspectos y la inclusión de *hooks* mediante *hotSwap*, siendo necesario modificar el código de la aplicación para tejer y destejer aspectos en tiempo de ejecución sino se contempló inicialmente.
3. Tejido estático y dinámico homogéneo. Los únicos sistemas que proporcionan los dos tipos de tejido transparentemente son DSAW y JBoss. Ambos lo proporcionan de forma homogénea. LOOM.Net tiene distintos tipos de puntos de corte para su versión estática y dinámica, además sus tejedores imponen requisitos a las aplicaciones diferentes. Spring especifica las clases y los puntos de enlace para tejido estático basándose en AspectJ, mientras que para su sistema dinámico utiliza otra especificación.
4. Tejido simultaneo de aspectos estáticos y dinámicos. Sólo DSAW proporciona esta capacidad si se habla de sistemas **realmente dinámicos**. Spring, JBoss AOP y LOOM.Net lo proporcionan de forma parcial.
5. Independencia del lenguaje de programación. Esto sólo es proporcionado por DSAW, LOOM.Net y el framework de Spring desarrollado para .Net

por estar construidos sobre una plataforma que lo es. El resto de sistemas sólo pueden trabajar con el lenguaje Java.

6. Interacción entre distintos lenguajes de programación. La interacción de sistemas sólo es posible en los sistemas que están basados en la plataforma .Net.
7. Sistema de propósito general. El único sistema que no es totalmente de propósito general es JAsCo. Este sistema está basado en componentes y orientado a los servicios Web.
8. Amplio conjunto de puntos de enlace. LOOM.Net soporta únicamente las invocaciones a métodos como puntos de enlace. PROSE no soporta invocaciones a métodos, pero si soporta el resto de puntos de enlace estándar mientras que JAsCo carece de tratamiento de excepciones. Spring sólo soporta puntos de enlace para métodos, no permite puntos de enlace sobre campos o propiedades.
9. Formas de adaptar un punto de enlace. JBoss AOP sólo permite el *around* como forma de adaptar una aplicación, mientras que PROSE, aunque permite los tres tipos normales, presenta limitaciones en el momento *around*.
10. Expresividad del lenguaje de puntos de corte. Todos los sistemas presentan mecanismos de definición de puntos de corte con gran potencia y flexibilidad.

B. Criterios correspondientes a la plataforma.

1. Independencia de la plataforma. Todos los sistemas han sido desarrollados sobre implementaciones de las máquinas abstractas de Java o .Net que son independientes de la plataforma.
2. Plataforma comercial ampliamente difundida. Las plataformas JVM y .Net gozan de amplia aceptación.
3. Plataforma con buen rendimiento. Existen implementaciones de las plataformas comentadas que proporcionan un rendimiento eficiente.
4. Sistema estándar. PROSE ha modificado la implementación de la máquina JVM para realizar su sistema, por lo que no es un sistema estándar.

C. Criterios correspondientes a las aplicaciones.

1. No necesidad del código fuente de las aplicaciones. LOOM.Net impone una serie de condiciones que hacen necesario disponer del código fuente de la aplicación, pese a que el tejido se hace a nivel de código intermedio. PROSE debe de añadir el manejador de aspectos al código fuente de la aplicación.
2. No imposición de condiciones a las aplicaciones. LOOM.Net impone condiciones a la aplicación que se va a adaptar. Las clases deben ser definidas como interfaces, y los métodos deben de ser virtuales. Además se restringe el uso del operador *new*.

D. Criterios correspondientes a los aspectos.

1. Adaptación simultánea de las aplicaciones. Todas las aplicaciones permiten la adaptación simultánea de sí mismas por más de un aspecto.
2. Mecanismo de coordinación de múltiples aspectos. Todos los sistemas ofrecen mecanismos para definir la precedencia en la ejecución de los

aspectos. Los sistemas de LOOM.Net establecen el orden de ejecución dependiendo del orden de creación de los aspectos, lo que no deja al desarrollador la posibilidad de seleccionar el orden de ejecución, y en el caso de que los aspectos surjan en distintos momentos puede que no sea posible conseguir el orden de ejecución deseado.

3. Aspectos como aplicaciones. Sólo JAsCo y DSAW soportan los aspectos como aplicaciones, posibilitando así que puedan beneficiarse de la AOP.
4. Uso de un lenguaje estándar. JAsCo y AspectJ han realizado extensiones del lenguaje Java, por lo que no hacen uso de un lenguaje estándar.
5. Separación del código de los aspectos del de los puntos de corte. AspectJ y JAsCo definen los puntos de corte en el código de los aspectos mediante extensiones realizadas al lenguaje. PROSE especifica los puntos de corte mediante invocaciones a ciertos métodos definidos por el sistema 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.
6. No imposición de condiciones a los aspectos, y no necesidad del código fuente de los aspectos. Sólo DSAW permite la utilización de cualquier librería, desarrollada con algún lenguaje soportado por la plataforma, ser usada como un aspecto. Para ello se ha implementado una utilidad que crea a partir de esa librearía original un *stub* que será el encargado de comunicarse con esa librería. Esta utilidad accede a la librería de los aspectos leyendo su código en lenguaje intermedio, por lo que no es necesario el código fuente del aspecto.

6.2.2.3 Discusión de los Resultados de la Evaluación Cualitativa

6.2.2.3.1 Discusión de los Criterios Generales

Los sistemas implementados sobre la máquina JVM se ven más desfavorecidos que los sistemas basados en la plataforma .Net. Esto es debido a las características de independencia del lenguaje propias de la plataforma .Net que, en el caso de la máquina de JVM, aunque de forma teórica existe cierta independencia del lenguaje, en la realidad no es tal. Estos sistemas se ven restringidos por el uso exclusivo de lenguaje Java tanto en las aplicaciones a adaptar como en los aspectos que adaptan.

6.2.2.3.2 Discusión de los Criterios de la Plataforma

Para la realización del sistema PROSE se ha modificado la máquina virtual JVM, con lo que ha dejado de ser estándar. El resto de los sistemas no han modificado las plataformas donde se implementan.

6.2.2.3.3 Discusión de los Criterios de las Aplicaciones

Todos los sistemas obtienen la máxima puntuación excepto los sistemas de la iniciativa LOOM.Net, Spring y PROSE. En el caso de los sistemas de LOOM.Net se imponen condiciones sobre las clases que se pretenden adaptar, sus métodos deben ser virtuales o haberse definido mediante una *interface*, el sistema dinámico no permite usar el operador *new* (o su equivalente en el lenguaje en que está programado la aplicación) debiendo realizar una llamada a una librería suministrada por el sistema. Esto implica la necesidad acceder al código fuente para proceder a realizar modificaciones en la aplicación que le permita ser adaptada por el sistema.

PROSE necesita el código fuente de la aplicación para añadir su manejador de aspectos. Spring puede añadir código para crear los aspectos al código fuente de la aplicación, o puede usar el fichero de configuración de *beans* para adaptar la aplicación.

6.2.2.3.4 Discusión de los Criterios de los Aspectos

Todos los sistemas permiten la adaptación simultánea de una aplicación por parte de varios aspectos en el mismo punto de enlace. Todos los sistemas ofrecen un mecanismo para coordinar la ejecución de múltiples aspectos cuando éstos afectan al mismo punto de enlace, pero el mecanismo de los sistemas de LOOM.Net no es suficiente.

Respecto a la posible reutilización de los aspectos, los sistemas que han realizado extensiones del lenguaje (AspectJ y JAsCo) se ven penalizados, pues su código no puede usarse fuera del sistema por no ser un lenguaje estándar.

JBoss AOP y DSAW separan la definición de puntos de corte del código de los aspectos, facilitando así la reutilización tanto de los propios aspectos, como de los puntos de corte.

Entre todos los sistemas evaluados, sólo JAsCo y DSAW ofrecen la posibilidad de que los aspectos sean aplicaciones dentro del sistema, con lo que también podrían beneficiarse de la AOP en su implementación, es decir, pueden ser adaptados por otros aspectos.

Por último, la transparencia que se ha buscado sobre el momento de tejido de un aspecto en DSAW, ha tenido como resultado la no imposición de condiciones sobre los aspectos que pueden ser usados por el sistema, y la no necesidad de su código fuente.

6.2.2.3.5 Discusión Absoluta

En la figura 41 se ve como los sistemas evaluados a excepción de DSAW tienen unas puntuaciones muy similares, manteniéndose todos los sistemas en una horquilla pequeña. DSAW obtiene la máxima puntuación en base a los criterios con los que se ha realizado esta comparación.

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 los sistemas basados en Java fueron penalizados por su dependencia del lenguaje, pero al realizar un cómputo general de todos los criterios éstos no son los peor calificados. Esto se debe a su mayor grado de desarrollo y madurez, que les hace obtener una mejor puntuación en otros criterios, compensando así la puntuación absoluta.

AspectJ, que es uno de los sistemas más ampliamente difundidos junto con JBoss AOP y Spring Java, ha sido diseñado como un sistema basado en Java y con extensiones del propio lenguaje. Estas características penalizan su puntuación por la dependencia del lenguaje y por la nula reutilización que se puede hacer de los aspectos. Además el bajo dinamismo ofrecido [Haupt06] (pese a la versión 5 realizada después de unirse al proyecto AspectWerkz) limita su puntuación.

PROSE es un sistema basado en Java que ofrece un alto grado de dinamismo. Aunque su eficiencia es buena como se puede comprobar en la comparación cuantitativa, las modificaciones realizadas en la máquina virtual, la inexistencia de algunas funcionalidades típicas de la AOP y su dependencia del lenguaje Java han lastrado su puntuación.

JAsCo es un sistema basado en el lenguaje Java, pero al igual que AspectJ ha sido realizado con extensiones del lenguaje, viéndose igualmente penalizado por ello. Junto con DSAW, JAsCo es el único sistema evaluado que permite adaptar aspectos de forma totalmente dinámica, sin necesidad de definirlos previamente al arranque de la aplicación. Un punto negativo de este sistema es su orientación hacia el desarrollo de servicios Web.

El framework de desarrollo Spring obtiene una buena puntuación. La existencia de una versión para la plataforma JVM y otra para la plataforma .Net es un punto muy favorable. Además, muchas de sus funcionalidades se han basado en las especificaciones del sistema AspectJ. Su gran lastre es no ofrecer ninguna posibilidad para la adaptación **realmente** dinámica de aplicaciones, y un limitado conjunto de puntos de enlace.

Los sistemas de LOOM.Net, pese a ser independientes del lenguaje y ofrecer tejedor estático y dinámico, no alcanzan una buena nota. Las condiciones que obligan a cumplir a las aplicaciones que se deben adaptar, su bajo grado de dinamismo, y la falta de homogeneidad entre los dos tipos de tejedores penalizan considerablemente su puntuación.

JBoss AOP ofrece un cierto grado de dinamismo y unas buenas características generales, además los esfuerzos que se están dedicando desde la compañía y su entorno están mejorando considerablemente el sistema a pasos agigantados. El hecho de ser un sistema dependiente del lenguaje Java, y sus limitaciones respecto a la adaptación dinámica es lo que más ha penalizado esta evaluación.

7 Conclusiones y Trabajo Futuro

El AOSD es una aproximación eficaz para alcanzar los beneficios del principio de la Separación de Incumbencias (SoC). En este paradigma, la aplicación final es construida a partir de las funcionalidades principales añadiendo las funcionalidades secundarias que se encuentran diseminadas y entremezcladas a lo largo de toda la aplicación. Estas funcionalidades secundarias se implementan como aspectos, y son tejidas sobre la aplicación final.

Hay muchas herramientas que soportan el AOSD. Algunas de ellas permiten tejer la aplicación previamente a su ejecución (tejido estático), mientras otras proporcionan esta adaptación en tiempo de ejecución (tejido dinámico). Aunque la aproximación estática es idónea en muchos casos, la adaptación dinámica puede también ser requerida cuando la aplicación debe responder a contextos emergentes en tiempo de ejecución, o requerimientos nuevos. El tejido estático soporta de forma eficiente el AOSD, mientras que el tejido dinámico permite la adaptación en tiempo de ejecución de las aplicaciones penalizando severamente su rendimiento.

En este trabajo de investigación se ha desarrollado DSAW, una plataforma de tejido estático y dinámico que proporciona independencia del lenguaje y de la plataforma, y la separación de la incumbencia momento de tejido (dinamismo). De esta forma, se proporcionan los beneficios de los dos tipos de tejidos en el mismo sistema.

La plataforma DSAW ha sido diseñada sobre la plataforma .Net, beneficiándose de sus características. Tanto el tejedor estático como el dinámico trabajan a nivel de código intermedio (IL), instrumentado las aplicaciones a ser adaptadas. Esto hace a DSAW independiente del lenguaje, permite la adaptación de aplicaciones legadas, y promueve la reutilización de componentes y aspectos. Además, el sistema ofrece adaptación realmente dinámica de aplicaciones, independencia de lenguaje y plataforma, y el mismo rico conjunto de puntos de enlace para ambos tejedores.

Un aspecto puede ser tejido estática o dinámicamente sin cambiar su código fuente, ni cambiar tampoco él del componente. Esto facilita el desarrollo *fix-and-continue*, mediante tejido dinámico en las primeras fases del desarrollo software, y posteriormente en las fases finales pasar a usar tejido estático más eficiente cuando la aplicación está a punto de ser puesta en producción. Esto es, DSAW separa completamente la incumbencia momento de tejido. El desarrollador puede modificar la flexibilidad/rendimiento durante el ciclo de vida de desarrollo, equilibrando la aplicación como mejor se ajuste a sus intereses.

Finalmente, es posible crear aplicaciones con aspectos que han sido tejidos estáticamente, junto con aspectos que son añadidos más tarde en tiempo de ejecución. El mecanismo para la resolución de conflictos se basa en hacer que los aspectos dinámicos tengan más precedencia que los aspectos estáticos. La resolución de conflictos entre aspectos del mismo tipo sigue una estrategia basada en la asignación de prioridades numéricas.

7.1 Objetivos

Objetivo 1. Realización de un sistema de tejido no invásivo que permita realizar tejido estático y dinámico de aspectos. Cumplido. Se ha implementado un prototipo (DSAW) con todas las funcionalidades necesarias de una herramienta AOSD. Éste se puede descargar en *http://www.reflection.uniovi.es*. El sistema implementado permite equilibrar los parámetros de rendimiento y flexibilidad. En las mediciones realizadas, se ha comprobado como se puede mejorar el rendimiento de una aplicación tejiendo los aspectos estáticamente en lugar de dinámicamente, si los requisitos lo permiten.

Objetivo 2. Combinación de adaptación dinámica y estática. Cumplido. No sólo se pueden inyectar aspectos estática o dinámicamente, es también viable sobre la misma aplicación inyectar a la vez aspectos con los dos tipos de tejido.

Objetivo 3. Ambos tipos de tejido, estático y dinámico, deben de ser soportados por el sistema de un modo homogéneo. Cumplido. No se deben realizar cambios para cambiar el momento de inyección de un aspecto. Además, se proporciona una utilidad que permite generar un *dispatcher* o *stub* para facilitar el lanzamiento de aspectos inyectados dinámicamente; de esta forma el desarrollador puede encapsular los aspectos desarrollados por él en una librería, y usarlos posteriormente estática o dinámicamente sin cambiar nada de su código de forma transparente. Esto tiene como consecuencias la no imposición de condiciones al código de los aspectos, y la no necesidad tampoco del código fuente de los aspectos.

Objetivo 4. Separación de la incumbencia momento de tejido. Cumplido. El desarrollador puede cambiar el momento de tejido sin cambiar la aplicación, o los aspectos.

Objetivo 5. Sistema de tejido dinámico totalmente adaptable. Cumplido. No existe ninguna restricción sobre las aplicaciones que utilicen tejido dinámico, y los aspectos tejidos dinámicamente pueden añadirse o eliminarse en tiempo de ejecución en cualquier momento.

Objetivo 6. Proporcionar un conjunto rico de puntos de enlace. Cumplido. Se proporciona el conjunto de puntos de enlace muy similar al proporcionado por AspectJ.

Objetivo 7. Independencia del lenguaje. Cumplido. Se trabaja a nivel de lenguaje IL, con lo que cualquier aplicación desarrollada sobre la plataforma .Net puede usar el sistema independientemente del lenguaje usado para su desarrollo.

Objetivo 8. Independencia de la plataforma. Cumplido. Para el diseño e implementación del sistema no se han usado características fuera del estándar .Net.

Objetivo 9. Adaptación de código binario. Cumplido. El uso del código intermedio permite inyectar aspectos sobre aplicaciones desarrolladas por terceros, de las que no se dispone el código fuente.

Objetivo 10. No debe existir ningún acoplamiento entre las aplicaciones y los aspectos que las adaptan. En las pruebas desarrolladas, no se necesitó añadir ningún código o realizar ningún cambio para que las aplicaciones pudiesen ser adaptadas por los aspectos tanto estática, como dinámicamente.

Objetivo 11. Reutilización de los componentes y los aspectos. Cumplido. En las pruebas realizadas los aspectos y los componentes se han reutilizado sin ninguna dependencia.

Objetivo 12. Adaptación un aspecto con otro aspecto. Cumplido. Los aspectos inyectados dinámicamente son componentes también en este sistema.

7.2 Trabajo Futuro

El trabajo futuro sobre esta investigación se encuadra dentro de la mejora del sistema desarrollado, DSAW. Se debe de optimizar el prototipo implementado para mejorar su rendimiento al usar tejido dinámico con respecto a otros sistemas AOSD. También se considera necesario la inclusión de un mecanismo más avanzado para la resolución de conflictos.

Las tareas que se pretenden realizar son:

- Optimizar el prototipo implementado. Aunque el prototipo desarrollado se acomoda bien para el desarrollo ágil orientado a aspectos debido a su robustez, nuestro futuro trabajo estará enfocado sobre la investigación de técnicas *hotswap in proccess* sobre la plataforma .Net. Así, si ambos tipos de tejido dinámico estuviesen disponibles sobre la plataforma, el tipo de tejido dinámico actual podría ser usado para desarrollo *edit-and-continue*, y el futuro para la añadir dinamismo a las aplicaciones finales desplegadas en caso de que no se pudiera eliminar esa característica de sus aspectos.
- Mejorar el mecanismo de resolución de conflictos. Ahora este mecanismo está basado en prioridades, dando más prioridad al tejido dinámico que al estático. Se considera necesario implementar una técnica para detectar conflictos semánticos sobre puntos de enlace compartidos [Durr05].
- Adaptar un grupo de *benchmarks* genéricos. Para la realización de las comparaciones de DSAW con otros sistemas se necesita disponer de un conjunto de pruebas que sean usadas por la comunidad científica AOSD. Estos *benchmarks* podrían ser de tipo *production aspects* [Diaz01] [Xalan08] [Zhang03] donde las funcionalidades proporcionadas por DSAW mejorasen la solución al problema planteado, y aportasen una mayor calidad a la aplicación resultante.
- Implementar características avanzadas AOSD. El sistema diseñado y muchos de los sistemas actuales AOSD no implementan características complejas como por ejemplo `pertarget`, `perthis` o `within`. La incorporación de algunas de estas características en este sistema incrementará de forma notable su calidad.
- Ampliar la expresividad del lenguaje de definición de puntos de corte y *advices*. La implementación de las características avanzadas AOSD comentadas en el punto anterior obligará a ampliar la gramática XSD para permitir su uso por los desarrolladores que usen el sistema.

7.3 Líneas de Investigación

Una línea de investigación, que se continúa en este proyecto, es la utilización del sistema DSAW para facilitar el desarrollo de proyectos siguiendo una metodología ágil

[Ortin02a]. Con este sistema se puede aplicar tejido dinámico cuando se desarrollan las aplicaciones para facilitar la labor de los programadores, y tejido estático cuando se finaliza el desarrollo para conseguir un rendimiento mejor.

Se considera que en esta línea se debe de continuar, y con el incremento de los lenguajes dinámicos y su incorporación a la plataforma .Net y JVM, se debería de trabajar para conseguir que los lenguajes más robustos -Java y C#- puedan tener capacidades muy similares a las de los lenguajes dinámicos cuando se realizan depuraciones, modificaciones, o se añaden nuevos requisitos durante el desarrollo.

De esta forma, se conseguiría reducir el tiempo para visualizar un cambio en tiempo de ejecución *–zero turn-around time-* (ZTAT), reduciendo así los tiempos de re-despliegue y re-inicio de una aplicación ante las modificaciones, preservando el estado de una ejecución de la aplicación, y eliminando también el tiempo necesario para cachear la información utilizada por una aplicación. Al finalizar el proceso de desarrollo, este código que se ha añadido mediante aspectos dinámicos se podría inyectar estáticamente, en caso de que fuese posible y necesario, para mejorar el rendimiento.

7.4 Difusión de los Resultados

La realización de este proyecto ha proporcionado ya algunos frutos. Se han obtenido datos de mediciones, ejemplos y pruebas que han sido usados para la realización de varios artículos de investigación explicando las características de esta plataforma.

Se realizó un artículo inicial, "DSAW: A Static and Dynamic Weaving Platform" (ISBN: 978-989-8111-51-7), explicando el diseño del sistema, y sus ventajas. Este artículo fue leído en el *"3rd International Conference on Software and Data Technologies"* realizado en Oporto (Portugal) del 5 al 8 de Julio de 2008. Este artículo fue realizado por: Dr. Luis Vinuesa, Martínez, Dr. Francisco Ortín Soler, Dr. Fernando Álvarez García y José M. Félix Rdgz.

En la etapa actual del proyecto se acaba de finalizar la redacción de un artículo más amplio, "Separating the Dynamism Concern of an Aspect-Oriented Platform", explicando de forma más detallada el sistema diseñado y su implementación, también se muestran ejemplos de su uso. Además, se hicieron unas mediciones para comparar de forma cualitativa y cuantitativamente DSAW con otros sistemas AOSD. Este artículo ha sido enviado a la revista *"Information and Software Tecnologhy"* de la editorial Elsevier (JCR 0.581 en el 2007), y fue realizado por los mismos autores.

Se está trabajando también en otro artículo explicando de forma concienzuda la arquitectura del sistema previo (Ready AOP), y los beneficios que aportan su tejido realmente dinámico. Además, se explican los cambios y mejoras que proporciona el nuevo sistema (DSAW) en relación con Ready AOP.

8 Presupuesto

La duración de este proyecto ha sido de 8 meses. Durante este tiempo ha trabajado en el proyecto una persona a tiempo completo, y su trabajo ha sido supervisado por dos investigadores con porcentajes de dedicación de 25% y de un 10% respectivamente.

El coste de recursos humanos ha sido de:

Tipo	Porcentaje	Coste/Mes(€)	Meses	Coste Total(€)
Investigador	100	1.200	8	9.600
Investigador Senior 1	25	1.800	8	2.400
Investigador Senior 2	10	1.800	8	960
Total				**12.960 €**

El coste del software utilizado ha sido de:

Software	Coste(€)
Visual Studio Pro 2005	1.029,90
XMLSpy	360
Visual SVN	49
Total	**1.438,90 €**

Los costes de documentación, fungibles, y un puesto de trabajo con ordenador y conexión a Internet han sido de:

Otros	Coste (€)
Fungibles	60
Documentación	60
Puesto Trabajo (8 meses/1 persona)	1.600
Total	**1.720 €**

El coste total del proyecto ha sido de:

Partida	Coste (€)
Recursos Humanos	12.960
Software	1.438,90
Otros	1.720
Total	**16.118,9 €**
16% IVA	2.570,02
Total (IVA incluido)	**18.697,92**

No se ha añadido a este documento la planificación de las tareas realizadas porque se considera una información no relevante.

A. Ejemplo de Utilización

Aplicación Inicial

A continuación se muestra un método (*payment*) donde se encuentran entremezcladas tres incumbencias. Por un lado la funcionalidad principal de la aplicación relacionada con la validación de una tarjeta de crédito, y la realización posterior de una operación de transferencia. Y por otro lado existe una incumbencia de logging (usando Apache Log4Net [Log4Net08]) del proceso, que se ejecuta al inicio (líneas 3 a 7) y al final del proceso(líneas 14 y 15) imprimiendo los parámetros y el valor de retorno del proceso. Además, también existe otra incumbencia de profiling de la duración del proceso (líneas 2 y 16) –las incumbencias de logging y profiling se encontrarian repetidas en cada método de la aplicación.

```
01 public bool payment(CreditCard card, double ammount) {
02     double startTime = DateTime.Now.Ticks;
03     log4net.Config.XmlConfigurator.Configure();
04     ILog logger =
05     LogManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);
06     logger.Info("Entering the payment method");
07     logger.Debug("Arguments: card=" + card + ", ammount=" + ammount);
08     bool correct = validateCard(card.Number, card.ExpDate, card.CardType);
09     if (correct)
10     {
11          CardCompany company = CardCompany.getCardCompany(card.CardType);
12          correct = company.transfer(card, this.myAccount, ammount);
13     }
14     logger.Debug("The payment has been " + (correct ? "successful" :
       "erroneous"));
15     logger.Info("Exiting the payment method");
16     profiler.measure("Payment::payment", (DateTime.Now.Ticks - startTime) /
TimeSpan.TicksPerMillisecond);
17     return correct;
18 }
```

Tabla 12: Función inicial con incumbencias secundarias.

Aspecto de Logging

Se puede inyectar un aspecto con la incumbencia de logging. De esta forma el código del procedimiento *payment* queda de la siguiente forma:

```
01 public bool payment(CreditCard card, double ammount) {
02   double startTime = DateTime.Now.Ticks;
03   bool correct = validateCard(card.Number, card.ExpDate, card.CardType);
04   if (correct) {
05     CardCompany company = CardCompany.getCardCompany(card.CardType);
06     correct = company.transfer(card, this.myAccount, ammount);
07   }
08   profiler.measure("Payment::payment", (DateTime.Now.Ticks - startTime) /
TimeSpan.TicksPerMillisecond);
09   return correct;
10 }
```

Tabla 13: Función inicial sin la incumbencia secundaria de traza.

```
static public object exec(   //loggingConcern
 string ns, string cl, string member,TypeOfMembers type, JPoint jp,Time time,
 Type ResultType, object ResultVal, Param[] Params, object OBJECT_THIS,
IReflection ir)
 {
  Type[] types; //Store types of parameters of the method or constrcutor
  Object[] par; //Store values of parameters of the method or constrcutor
  if (Params == null) {
    types = new Type[0];
    par = null;
  } else {
    types = new Type[Params.Length];
    par = new object[Params.Length];
    for (int i = 0; i < Params.Length; i++) {
      types[i] = Params[i].type;
      par[i] = Params[i].val;
   }
  }
  log4net.Config.XmlConfigurator.Configure();
  ILog logger =
    logManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);
  if (time == Time.Before) {
    logger.Info("Entering the " + member + " method");
    logger.Debug("Arguments:");
    for (int i = 0; i < Params.Length; i++) {
      logger.Debug("Name= " + Params[i].name + " | Value= " + Params[i].val);
    }
  }
  if (time == Time.After) {
    logger.Debug("The return value is " + ResultVal);
    logger.Info("Exiting the " + member + " method");
  }
  return ResultVal;
}
```

Tabla 14: Incumbencia de traza encapsulada en un aspecto.

Este aspecto se puede inyectar estáticamente en un punto de enlace de tipo *Method Call* en los momentos *Before* y *After* utilizando las siguientes definiciones XML:

```
<advice_definition>
       <name>StaticAspectLogger</name>
       <assembly>Test6StaticAspect.dll</assembly>
       <type>Test6StaticAspect.Test6StaticAspect</type>
       <behaviour>exec</behaviour>
       <pointcut_definition>
              <time>before</time><time>after</time>
              <joinpoint_type>
              <methodcall>
              <method_signature>
              <return_type><type_name>*</type_name></return_type>
              <qualified_method_name>
                     <qualified_class>
                     <namespace><type_name>*</type_name></namespace>
                     <class><identifier_name>*</identifier_name></class>
                     </qualified_class>
                     <name><identifier_name>payment</identifier_name></name>
              </qualified_method_name>
              </method_signature>
              </methodcall>
              </joinpoint_type>
       </pointcut_definition>
</advice_definition>
```

Tabla 15: Fichero XML para inyección estática de un aspecto de traza.

Aspecto de Profiling

Para eliminar el profiling de la aplicación se puede inyectar un aspecto que lo haga. Así el procedimiento *payment* quedaría:

```
01 public bool payment(CreditCard card, double ammount) {
02  bool correct = validateCard(card.Number, card.ExpDate, card.CardType);
03  if (correct) {
04    CardCompany company = CardCompany.getCardCompany(card.CardType);
05    correct = company.transfer(card, this.myAccount, ammount);
06  }
07  return correct;
08 }
```

Tabla 16: Función inicial sin incumbencias secundarias.

```
public Profiler() {
  this.startTime = 0;
}

private static void measure(string methodName, double millis) {
  Console.WriteLine("> {0}:({1}).", methodName, millis);
}
public object exec(string ns, string cl, string member, TypeOfMembers type,
  JPoint jp, Time time, Type ResultType, object ResultVal, Param[] Params,
  object OBJECT_THIS, IReflection ir)
 {
   object returnValue = (System.Int32)1;
   if ( time == Time.Before)
      startTime = DateTime.Now.Ticks;
   if (time == Time.After)
      measure(member, (DateTime.Now.Ticks - startTime) /
TimeSpan.TicksPerMillisecond);
   return returnValue;
}
```

Tabla 17: Incumbencia de profiling encapsulada en un aspecto.

Se debe remarcar que la signatura del método es exactamente igual que el aspecto estático de logging creado anteriormente.

Este aspecto se puede inyectar dinámicamente en un punto de enlace de tipo *Method Call* en un momento *Before* y *After* utilizando las siguientes definiciones XML:

```
<pointcut_definition id="pcPayment">
       <time>before</time><time>after</time>
       <joinpoint_type>
       <methodcall>
       <method_signature>
       <return_type><type_name>*</type_name></return_type>
       <qualified_method_name>
       <qualified_class><namespace><type_name>*</type_name></namespace>
              <class><identifier_name>*</identifier_name></class>
       </qualified_class>
              <name><identifier_name>payment</identifier_name></name>
       </qualified_method_name>
       </method_signature>
       </methodcall>
       </joinpoint_type>
</pointcut_definition>
```

Tabla 18: Fichero XML para inyección dinámica de un aspecto de profiling.

Se puede reaprovechar está definición de punto de corte para el tejido estático, ya que la instrumentación de código para el tejido dinámico se va realizar en los mismos puntos de enlace y para el mismo momento.

```
<advice_definition>
       <name>StaticAspectLogger</name>
       <assembly>Test6StaticAspect.dll</assembly>
       <type>Test6StaticAspect.Test6StaticAspect</type>
       <behaviour>exec</behaviour>
       <pointcut_definitionRef idRef="pcPayment"/>
</advice_definition>
```

Tabla 19: Reutilización de una definición de punto de corte.

B. Datos de las Mediciones

Datos de la Comparación del Tipo deTejido

	Method Call Around (Escenario 1)			
	Aplicación	Aplicación	Aspecto	Total
Tipo Aplicación	Tiempo (ms.)	Peak WorkingSet (Kb.)	Peak WorkingSet (Kb.)	Peak WorkingSet (Kb.)
Sin incumbencia	1161	4840		4840
Con incumbencia	1161	4804		4804
Con aspecto estático	991	9332		9332
Con aspecto dinámico	33748	20524	11636	32160

Tabla 20: Datos del Escenario 1.

	Method Call Before (Escenario 2)			
	Aplicación	Aplicación	Aspecto	Total
Tipo Aplicación	Tiempo (ms.)	Peak WorkingSet (Kb.)	Peak WorkingSet (Kb.)	Peak WorkingSet (Kb.)
Sin incumbencia	1161	4840		4840
Con incumbencia	1161	5132		5132
Con aspecto estático	2894	9412		9412
Con aspecto dinámico	16163	20104	11600	31704

Tabla 21: Datos del Escenario 2.

Datos de la Comparación Cuantitativa con Otros Sistemas

	Aplicación	
Plataforma	Time (Ms)	Peak Working Set (K)
C# Sin Aspectos	871	4764
Java Sin Aspectos	1042	7016

Tabla 22: Datos de las plataformas sin aspectos.

	Method Call Before			
	Aplicación		Aspecto Dinámico	Total
Plataforma	Tiempo (Ms)	Peak Working Set (K)	Peak Working Set (K)	Peak Working Set (K)
Aspecto	981	7060		7060
DSAW Static Aspect	831	9284		9284

DSAW Dynamic Aspect	14811	10636	11352	21988
Prose 1.3.0	7080	9924		9924
Jasco0.8.7	961	12160		12160
Spring Java 2.5	982	17756		17756
Spring Net 1.1.2	911	12536		12536
Rapier-Loom.NET 2.21	881	11096		11096
Gripper-Loom.Net 0.92	911	7900		7900

Tabla 23: Datos de la inyección de un aspecto en *Method Call Before*.

Method Call After				
		Aplicación	Aspecto Dinámico	Total
Plataforma	Tiempo (Ms)	Peak Working Set (K)	Peak Working Set (K)	Peak Working Set (K)
Aspecto	942	7456		7456
DSAW Static Aspect	801	9276		9276
DSAW Dynamic Aspect	14901	10636	11344	21980
Prose 1.3.0	6950	9924		9924
Jasco0.8.7	981	12136		12136
Spring Java 2.5	992	17780		17780
Spring Net 1.1.2	841	12500		12500
Rapier-Loom.NET 2.21	901	11136		11136
Gripper-Loom.NET 0.92	911	7900		7900

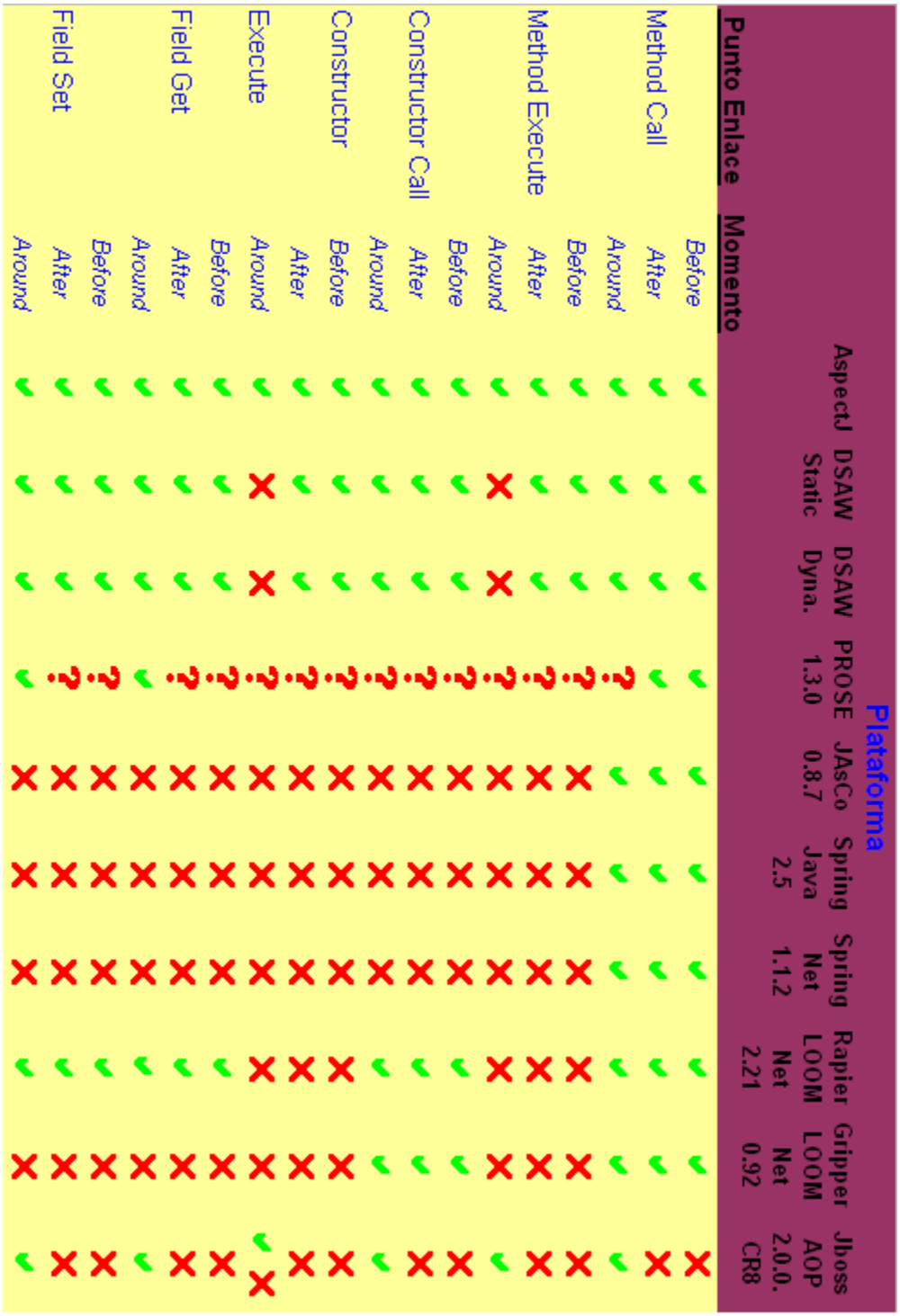
Tabla 24: Datos de la inyección de un aspecto en *Method Call After*.

Method Call Around				
		Aplicación	Aspecto Dinámico	Total
Plataforma	Tiempo (Ms)	Peak Working Set (K)	Peak Working Set (K)	Peak Working Set (K)
Aspecto	951	7364		7364
DSAW Static Aspect	851	9380		9380
DSAW Dynamic Aspect	31325	11032	11540	22572
Jasco0.8.7	961	12304		12304
Spring Java 2.5	992	17760		17760
Spring Net 1.1.2	861	12448		12448
Rapier-Loom.NET 2.21	881	11476		11476
Gripper-Loom.NET 0.92	901	7900		7900
JBoss AOP 2.0.0.CR8 (Compile Time)	1042	11252		11252
JBoss AOP 2.0.0.CR8 (Load Time)	1011	14840		14840

JBoss AOP 2.0.0.CR8 (HotSwap)	1102	15204	15204

Tabla 25: Datos de la inyección de un aspecto en *Method Call Around*.

C. Tipos de Puntos de Enlace Soportados



Punto Enlace	Momento	Plataforma										
		AspectJ	DSAW Static	DSAW Dyna.	PROSE 1.3.0	JAsCo 0.8.7	Spring Java 2.5	Spring Net 1.1.2	Rapier LOOM Net 2.21	Gripper LOOM Net 0.92	Jboss AOP 2.0.0. CR8
Method Call	Before	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	After	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	Around	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
Method Execute	Before	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	After	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	Around	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Constructor Call	Before	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	After	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	Around	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
Constructor	Before	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
	After	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
	Around	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
Execute	Before	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
	After	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
	Around	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Field Get	Before	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	After	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	Around	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
Field Set	Before	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	After	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓
	Around	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗

Tabla 26: Tipos de enlace soportados por las plataformas.

En esta tabla se muestran los tipos de enlace y los momentos de inyección soportados por cada plataforma probada para la realización de las mediciones. Las

casillas con interrogación representan que la documentación aportada es insuficiente para saber si ese tipo de enlace y/o momento es contemplado en esa plataforma. Las casillas con el signo afirmativo y el signo negativo representan que el tipo de enlace en ese momento está soportado por la plataforma, pero se han encontrado algunos problemas al realizar las pruebas.

D. Referencias

[Absil08] Absil, 2008. *Abstract IL.* [En línea] http://research.microsoft.com/projects/ilx/absil.aspx [accedido 15 Abril 2008].

[Aksit92] Aksit M., Bergmans L. & Vural S., 1992. An object-oriented language-database integrationmodel: The composition filters approach. In *Proceedings of ECOOP '92*.

[Aspectc08] AspectC++, 2008. *The home of AspectC++.* [En línea] http://www.aspectc.org/ [accedido 15 Febrero 2008].

[AspectJ08] AspectJ, 2008. *The AspectJ Project.* [En línea] http://www.eclipse.org/aspectj/ [accedido 23 Febrero 2008].

[AspectJ08b] Aspecto, 2008b. *The AspectJTM 5 Development Kit Developer's Notebook*. [En línea] http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html [accedido 27 Febrero 2008].

[AspectWerkz08] AspectWerkz, 2008. *AspectWerkz – Plain Java AOP- Overview.* [En línea] http://aspectwerkz.codehaus.org/ [accedido 28 Febrero 2008].

[Autonomic08] Autonomic, 2008. *IBM Research | Autonomic computing.* [En línea] http://www.research.ibm.com/autonomic/ [Accedido 17 Febrero 2008].

[Baker02] Baker, J. & Hsieh, W., 2002. Runtime aspect weaving through metaprogramming. *AOSD 2002 conference Proceedings*, Pages: 86 – 95

[Bergmans94] Bergmans, L., 1994. *Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*. Ph. D. Dissertation. University of Twente.

[Bergmans00] Bergmans, L. & Aksit, M., 2000. Composing Multiple Concerns Using Composition Filters. *ICSE 2000*.

[Blackstock04] Blackstock, M., 2004. *AOPNET5* [En línea] http://www.cs.ubc.ca/~michael/publications/AOPNET5.pdf [accedido 12 Abril 2008].

[Böllert99] Böllert, K., 1999. On Weaving Aspects. In: *European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect Oriented Programming* p.301-302.

[Cech06] Cech Previtali, S. & Gross, T.R., 2006. Dynamic Updating of Software Systems Based on Aspects. *22nd IEEE International Conference on Software Maintenance (ICSM'06)*. 83-99.

[Cecil08] Cecil, 2008. *Cecil – Mono*. [En línea] http://www.mono-project.com/Cecil [accedido 15 Abril 2008].

[Chitchyan04] Chitchyan, R. & Sommerville, I., 2004. Comparing Dynamic AOP Systems. *Proceedings of the 2004 Dynamic Aspects Workshop.*Technical Report RIACS Technical Report No.04.01, RIACS

[Clarke99] Clarke, S., Harrison, W., Ossher, H. & Tarr, P., 1999. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)* Denver, Colorado U.S., Noviembre.

[Cohen04] Cohen, T. & Gil, J., 2004. AspectJ2EE = AOP + J2EE: towards an Aspect Based, Programmable and Extensible Middleware Framework. *ECOOP 2004 – Object-Oriented Programming*, 3086 pp: 219-243, Spronger-Verlag.

[CodeDOM08] CodeDom, 2008. *Using the CodeDOM*. [En línea] http://msdn.microsoft.com/en-us/library/y2k85ax6.aspx [accedido 15 Junio 2008].

[DCS408] Dept.of CS 4, 2008. *Informatik 4 - Distributed Systems and Operating Systems.* [En línea] http://www4.informatik.uni-erlangen.de/ [accedido 15 Marzo 2008]. Universidad de Erlangen-Nuremberg. Alemania.

[Demeter08] Demeter, 2008. *Demeter in Action*. [En línea] http://www.ccs.neu.edu/research/demeter/center/DemInAction.html [accedido 23 Febrero 2008].

[Diaz01] Día Pace, J.A. & Campo, M.R., 2001. Analyzing the Role of Aspects in software. *Design Communications of the ACM*, 44(10):66-73.

[Dijkstra76] Dijkstra, E.W., 1976. *A Discipline of Programming*. Prentice Hall, Inc.

[DOC08] Group DOC, 2008. *Center for Distributed Object Computing.* [En línea] http://doc.ece.uci.edu/index.php [accedido 15 Febrero 2008]. Universidad de California, Irvine.

[Dufour04] Dufour, B., Goard, C., Hendren, L., Verbrugge, C.,de Moor, O. & Sittampalam, G., 2004. Measuring the Dynamic Behaviour of AspectJ Programs. In *Porceedings of OOSPLA*.

[Durr05] Durr P., Staijen T., Bergmans L. & Aksit M., 2007. Reasoning about semantic conflicts between aspects. In K. Gybels, M. D'Hondt, I. Nagy, and R. Douence, editors, *2nd European Interactive Workshop on Aspects in So*ftware, 2005.

[Eclipse08] Eclipse, 2008. *Eclipse.org home.* [En línea] http://www.eclipse.org/ [accedido 25 Febrero 2008].

[Eaddy05] Eaddy, M. & Freiner, S., 2007. Multi-Language Edit-and-Continue for the Masses. Technical Report CUCS-015-05, Departament of Computer Science, Columbia University. Abril, 2005.

[Eaddy07a] Eaddy, M., Aho, A., Weiping, H., McDonald, P. & Burger, J., 2007. Debugging Aspect-Enabled Programs. *International Symposium on Software Composition (SC 2007)*, Braga, Portugal, March 25-26.

[Eaddy07b] Eaddy, M., 2007b. Wicca 2.0: Dynamic Weaving using the .Net 2.0 Debugging API. Demonstration at *Aspect-Oriented Software Development (AOSD 2007)*, Vancouver, British Columbia, March 12-16.

[Eaddy07c] Eaddy, M., 2007c. *Wicca.* [En línea] http://www1.cs.columbia.edu/~eaddy/wicca/ [accedido 15 Enero de 2008].

[Ecma2008] Ecma, 2008. *Standard ECMA-335.* [En línea] http://www.ecma-international.org/publications/standards/Ecma-335.htm [accedido 15 Enero 2008].

[Elrad01] Elrad, T., Aksits, M., Kiczales, G., Lieberherr, K. & Ossher, Ha., 2001. Discussing aspects of AOP. *Communications of the ACM* October Volume 44 Issue 10

[Frei04] Frei, A., Grawehr, P. & Alonso, G., 2004. A Dynamic AOP-Engine for .NET. Technical Report 445, Department of Computer Science, ETH Zürich.

[Gilani04a] Gilani, W. & Spinczyk, O., 2004a. A Family of Aspect Dynamic Weavers. *AOSD 2004 - Dynamic Aspects Workshop (DAW04)*, Lancaster, UK.

[Gilani04b] Gilani, W., Hasan Naqvi, N. & Spinczyk, O.., 2004b. On Adaptable Middleware Product Lines. *3rd Workshop on Reflective and Adaptive Middleware, ACM/IFIP/USENIX 5th International Middleware Conference*, Toronto, Ontario, Canada October 18th - 22nd.

[Gilani07a] Gilani, W., Sincero, J. & Spinczyk, O., 2007a. Aspectizing a Web Server for Adaptation. *Proceedings of the 12th IEEE*

Symposium on Computers and Communications, July, Aveiro, Portugal.

[Gilani07b] Gilani, W., Scheler, F., Lohman, D., Spinczyk, O. & Schröder-Preikschat, W., 2007b. Unification of Static and dynamic AOP for Evolution in Embedded Software Systems. *6th International Symposium on Software Composition*, March, Braga, Portugal.

[Gorappa05] Gorappa, S., Colmenares, J.A., Jafarpour, H. & Klefstad, R., 2005. Tool-based Configuration of Real-time CORBA Middleware for Embedded Systems. *Proc. of the 8th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC 2005)*. Seattle, Washington, USA.

[Griffith05] Griffith, R. & Kaiser, G., 2005. Manipulating Managed Execution runtimes to Support Self-Healing Systems. *ICSE 2005 Workshop - Design and Evolution of Autonomic Application Software*.

[Harrison93] Harrison, W. & Ossher, H., 1993. Subject-Oriented Programming – A Critique of Pure Objects. *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September.

[Haupt04] Haupt, M. & Mezini, M., 2004. Micro-Measurements for Dynamic Aspect-Oriented Systems *Proc. of Net.ObjectDays 2004 (NODe)*, LNCS 3263.

[Haupt06] Haupt, M., 2006. *Virtual Machine Support for Aspect Oriented Languages.* Tesis Doctoral. Universidad Darmstadt.

[Hürsch95] Hürsch, W.L. & Lopes, C.V., 1995. Separation of Concerns, Technical Report UN-CCS-95-03, Northeastern University, Boston, USA.

[Hyperspace08] Hyperspace, 2008. *Multi-Dimensional Separation of Concerns : Software Engineering Using Hyperspaces*. [En línea] http://www.research.ibm.com/hyperspace/ [accedido 1 Febrero 2008].

[HyperJ08] HyperJ, 2008, *HyperJ: Multi-Dimensional Separation of Concerns for Java*. [En línea] http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm [accedido 2 Febrero 2008].

[IBM2000] IBM, 2000. "Multi-dimensional Separations of Concerns. International Business Machines corporation, IBM Research. 2000.

[ICSE2000] ICSE, 2000. Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering. ICSE'2000. Limerick (Irlanda). Junio de 2000.

[JasCo08] JasCo, 2008. *start - JAsCo - System and Software Engineering Lab*. [En línea] http://ssel.vub.ac.be/jasco/ [accedido 2 Junio 2008].

[JavaGrande08] Java Grande, 2008. Java Grande Benchmark. [En línea] http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/ [accedido 18 Junio 2008].

[JavaRevel08] ZeroTurnaround, 2008. *Zeroturnaround >> JavaRebel.* [En línea] http://www.zeroturnaround.com/javarebel/ [accedido 2 Julio 2008].

[JBossAOP08] JBoss AOP, 2008. *jboss.org: community driven.* [En línea] http://www.jboss.org/jbossaop/ [accedido 10 Febrero 2008].

[JSR-220.08] Java Community Process, 2008. *JSR-000220 Enterprise Java Beans 3.0 – Final Release.* [En línea] http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html [accedido 15 Febrero 2008].

[Kiczales92] Kiczales, G., Rivieres, J. & Bobrow, D.G., 1992 *The Art of Meta-object Protocol*. MIT Press.

[Kiczales97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M.& Irwin, J, 1997. Aspect Oriented Programming. In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, vol. 1241 of Lecture Notes in Computer Science, Springer Verlag. 1997.

[Kiczales01] Kizcales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W., 2001. An Overview of AspectJ. *Proceedings of the European Conference on Object-Oriented Programming*, Budapest, Hungary, 18--22 June.

[Klefstad02] Klefstad, R., Schmidt, D. C. & O'Ryan, C., 2002. Towards Highly Configurable Real-time Object Request Brokers. *Proc. of the IEEE Int'l Symposium on Object-Oriented Real-time Distributed Computing (ISORC 2002)*. Washington DC, USA. 2002.

[Köhne05] Köhne, K., Schult, W. & Polze, A., 2005. Design by contract in .NET Using Aspect Oriented Programming.

[Krinstensen96] Kristensen, B.B. & Østerby. K., 1996. Roles: Conceptual Abstraction Theory and Practical Languages Issues. *Theory and Practice of Object Systems*, Volume 2(3), 143-160.

[Kristensen96b] Kristensen, B. B., 1996. Object-oriented ModeHing with Roles. *OOIS'95, Proceedings of the 2nd International Conferenceon Object-oriented Information Systems*, Dublin, Ireland.

[Laddad03] Laddad, R., 2003. *AspectJ in Action.* Grenwich: Manning.

[Lafferty03] Lafferty, D & Cahill, V., 2003. Language-Independent Aspect-Oriented Programming. *OOPSLA'03.* October 26-30, 2003. Anaheim, California, USA.

[Lieberherr96] Lieberherr, K. J., 1996. *Adaptative Object-Oriented Software: The Demeter Method with Propagation Patterns.* Boston: PWS Publishing Company.

[Log4Net08] Log4Net, 2008. *Apache log4net - Apache log4net: Home.* [En línea] http://logging.apache.org/log4net/index.html [accedido 15 Junio 2008].

[Lohmann04] Lohmann, D., Gilani, W. & and Spinczyk, O., 2004. On Adapable Aspect-Oriented Operating Systems. *In Proceedings of the 2004 ECOOP Workshop on Programming Languages and Operating Systems (* ECOOP-PLOS 2004*)*. Oslo, Noruega. Junio.

[LOOM.Net08] LOOM.Net, 2008. *Operating Systems and Middleware Group at HPI - LOOM.NET.* [En Línea] http://www.dcl.hpi.uni-potsdam.de/research/loom/ [accedido 25 Marzo 2008]. Instituto Hasso Plattner. Universidad de Potsdam.

[Loughran05] Loughran, N., Parlavantzas, N., Pinto, M., Sánchez, P., Webster, M. & Colyer, A., 2005. Survey of Aspect-Oriented Middleware. *AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-10.* Editor(s): N. Loughran, M.Pinto.

[Luo08] Luo, Haibo, 2008. *Haibo Luo's WebLog: System.Reflection – based ILReader.* [En línea] http://blogs.msdn.com/haibo_luo/archive/2006/11/06/system-reflection-based-ilreader.aspx [accedido 15 Abril 2008]

[Maes87] Maes, P., 1987. *Computational Reflection*. PhD. Thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Belgium.

[Masuhara03a] Masuhara, H. & Kizcales, G., 2003. Modelling Crosscutting Aspect-Oriented Mechanisms. *In Proc. ECOOP 2003*.

[Masuhara03b] Masuhara, H. & Kizcales, G., 2003. A Compilation and Optimization Model for Aspect-Oriented Programs. *Compiler Construction* (CC2003), LNCS 2622.

[Matthijs97] Matthijs, F., Joosen, W., Vanhaute, B., Robben, B. & Verbaten, P., 1997. Aspects should not die. In: *European Conference on*

Object-Oriented Programming (ECOOP) Workshop on Aspect-Oriented Programming.

[Miller92] Miller, S., 1992. *DEC/HP Network Computing Architecture Remote Procedure Call Run Time Extensions version OSF TX1.0.11*. Cambridge, MA: Open Software Foundation.

[Nagy05] Nagy, I, Bergmans, L.M.J. & Aksit, M. Composing Aspects at Shared Join Points. In: *Proceedings of International Conference NetObjectDays*, NODe, 2005.

[Nicoara05] Nicoara, A. & Alonso, G., 2005. Dynamic AOP with PROSE. *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.

[Nicoara08] Nicoara, A., Alonso, G., Gross, T. & Roscoe, T., 2008. Controlled, systematic and efficient code replacement for running Java programs. In: *Proceedings of the ACM EuroSys 2008 Conference (EuroSys 2008)*, Glasgow, Scotland, UK.

[Ortin02a] Ortin, F., Cueva, J., 2002. Implementing a real computational-environment jump in order to develop a runtime-adaptable reflective platform. *ACM SIGPLAN Notices, Volume 37, Issue 8.*

[Ortin02b] Ortin, F., 2002. Sistema computacional de programación flexible diseñado sobre una máquina abstracta reflectiva no restrictiva. ISBN: 84-8317-304-2. Tesis Doctoral. Febrero de 2002. Universidad de Oviedo.

[Ortin03a] Ortin, F., Cueva, JM. Non-Restrictive Computational Reflection. Computer Standards and Interfaces, volume 25, issue 3, pp. 241-251. Junio 2003.

[Ortin03b] Ortin, F., Martínez, AB., Cueva, JM. The Reflective nitrO Abstract Machine. ACM SIGPLAN Notices, volume 38, issue 6, pp. 40-49. Junio 2003.

[Ortin04a] Ortin, F., López, B. & Pérez-Schofield, J.B.G., 2004. Separating Adaptable Persistence Attributes through Computational Reflection. *IEEE Software, Volume 21, Issue 6.*

[Ortin04b] Ortin, F., Cueva, JM. Dynamic Adaptation of Application Aspects. Journal of Systems and Software, volume 71, issue 3, pp. 229-243. Mayo 2004.

[Ortin05] Ortin, F., Díez, D. Designing an Adaptable Heterogeneous Abstract Machine by means of Reflection. Information and Software Technology, volume 47, issue 2, pp. 81-94. Febrero 2005.

[Ortin07] Ortin, F., Zapico, D., Cueva, JM. Design Patterns for Teaching Type Checking in a Compiler Construction Course. IEEE Transactions on Education, volume 50, issue 3, pp. 273-283. Agosto 2007.

[OOPSLA99] OOSPLA, 1999. First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems. OOPSLA'99. Denver (EE.UU.). Noviembre 1999.

[Ossher99] Ossher, H. & Tarr, P., 1999. Multi-Dimensional Separation of Concerns using Hyperspaces. IBM Research Report 21452. Abril.

[O'Brien01] O'Brien, L., 2001. The First Aspect-Oriented Compiler. *Software Development Magazine*, Septiembre.

[Parnas72] Parnas, D., 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, Vol. 15, No. 12.

[Phoenix08] Phoenix, 2008. *Phoenix Academic Program.* [En línea] http://research.microsoft.com/Phoenix/ [accedido 15 Abril 2008].

[Pinto02] Pinto, M., Fuentes, L., Fayad, M.E. & Troya, J.M., 2002. Separation of Coordination in a Dynamic Aspect Oriented Framework. *AOSD 2002 Proceedings*, Pages 134-140

[Popovici01] Popovici, A., Gross, T. & Alonso, G., 2001. Dynamic Homogenous AOP with PROSE. Technical Report, Dept. of Computer Science, ETH Zürich.

[Popovici03] Popovici, A., Alonso, G. & Gross, T., 2003. Just in Time Aspects: Efficient Dynamic Weaving for Java. En: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA.

[Pryor02] Pryor, J., Diaz Pace, A. & Campo, M., 2002. Reflecting on Separation of Concerns. *Revista de Informática Teórica y Aplicada (RITA)* Volume X, No. 2.

[Rail08] Rail, 2008. *RAIL – Runtime Assembly Instrumentation Library Poject.* [En línea] http://rail.dei.uc.pt/ [accedido 15 Abril 2008].

[Redondo08] Redondo, JM., Ortin, F., Cueva, JM. Optimizing Reflective Primitives of Dynamic Languages. International Journal of Software Engineering and Knowledge Engineering, volume 18, issue 6, pp. 759-783. Septiembre 2008.

[Roeder08] Roeder, Lutz, 2008. .Net Reflector. [En línea] http://www.aisto.com/roeder/dotnet/Download.aspx?File=Resourcer [accedido 15 Abril 2008].

[Sato03] Sato, Y., Chiba, S. & Tatsubori, M., 2003. A Selective, Just-in-Time Aspect Weaver. *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03),* LNCS 2830, pp.189-208, Springer-Verlag.

[Schröder06] Schröder-Preikschat, W., Lohmann, D., Gilani, W., Schele & F., Spinczyk, O., 2006. Static and dynamic weaving in System Software with AspectC++. In Yvonne Coady, Jeff Gray, and Raymond Klefstad, editors, *HICSS '06 Mini-Track on Adaptive and Evolvable Software Systems, IEEE*, January.

[Schult03] Schult, W., Trögger, P., 2003. LOOM.Net – an Aspect Weaving Tool. *Workshop on Aspect-Oriented Programming, ECOOP'03,* Darmstadt.

[Schult08] Schult, W., 2008. Documentación Gripper-LOOM.Net. [En línea] http://www.dcl.hpi.uni-potsdam.de/research/loom /Download/GripperLoomInstallerV0.91.1066.msi [accedido 15 Enero 2008].

[Segura-Deville03] Segura-Devillechaise, M., Menaud, J., Muller, G.& Lawall, J., 2003. Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution. *AOSD 2003 Proceedings*, pp: 110-119.

[Specjvm98] Spec Jvm98, 2008. SPEC JVM 98. [En línea] http://www.spec.org/osg/jvm98/ [accedido 19 Junio 2008].

[SpringAOP08] SpringAOP, 2008. Aspect Oriented Programming with Spring. [En línea] http://static.springframework.org/spring/docs/2.5.x/ reference/aop.html [accedido 23 Febrero 2008].

[Spring08] Spring, 2008. Springframework.org. [En línea] http://www.springframework.og [accedido 10 Junio 2008].

[SpringNET08] Spring .Net, 2008. Spring.NET. [En línea] http://www.springframework.net [accedido 9 Junio 2008].

[Sullivan01] Sullivan, G.T., 2001. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*. October 2001/ Vol 44 No. 10 Pages: 95 – 97.

[Suvee03] Suvee, D., Vanderperren, W. & Jonckers, V., 2003. 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, ACM Press. Marzo, 2003.

[Tarr99] Tarr, P., Ossher, H., Harrison, W. & Sutton, S., 1999. N Degrees of separation: Multi-Dimensional Separation of Concerns. In: *Proceedings of the 1999 International Conference on Software Engineering*.

[Vinuesa04] Vinuesa, L. & Ortin, F., 2004. A Dynamic Aspect Weaver over the .NET Platform. *Metainformatics International Symposium, MIS 2003.* Graz, Austria. Septiembre 2003 Springer-Verlag Lecture Notes in Computer Science 3002.

[Vinuesa07] Vinuesa, L., 2007. *Separación Dinámica de Aspectos independiente del Lenguaje y Plataforma mediante el uso de Reflexión Computacional.* Tesis Doctoral. Universidad de Oviedo.

[Vinuesa08] Vinuesa, L., Ortin, F., Álvarez, F. & Félix, J., 2008.DSAW: A Static and Dynamic Weaving Platform. *3rd International Conference on Software and Data Technologies.* Oporto Portugal. Julio, 2008.

[Walls07] Walls, C., 2007. *Spring in Action, 2th Edition.* Grenwich: Manning.

[Wanderperren05] Wanderperren, W., Suv'ee, D., De Fraine, B & Jonckers Viviane, 2005. Aspect Oriented Programming using JasCo. *In Proceedings of AOSD 2005*, ACM Press, Chicago, USA..

[Weave08] Weave, 2008. *Weave.NET – Intro.* [En línea] http://www.dsg.cs.tcd.ie/dynamic/?category_id=-26 [accedido 12 Abril 2008].

[Wicca08] Wicca, 2008. *Wicca.* [En línea] http://www1.cs.columbia.edu/~eaddy/wicca/ [accedido 15 Abril 2008].

[Xalan08] Xalan 2008. Xalan-Java Version 2.7.1. [En línea] http://xml.apache.org/xalan-j/ [accedido 15 Junio 2008].

[XSLTMark08] XSLTMark, 08. IBM – WebSphere DataPower SOAAppliances – Family Overview. [En línea] http://www-306.ibm.com/software/integration/datapower/ [accedido 15 Junio 2008].

[Zhang03] Zhang, C. & Jacobsen, 2003. Qunatifying Aspects in Middleware Platforms. *In Proc. AOSD 2003*; páginas 10-139. ACM Press.

[Zinky97] Zinky, J., Bakken D. & Schantz, R., 1997. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems (TAPOS)*, Abril.