Departamento de Informática

UNIVERSIDAD DE OVIEDO

PLATAFORMA DE DESARROLLO SOFTWARE CON TEJIDO ESTÁTICO Y DINÁMICO DE ASPECTOS

TESIS DOCTORAL

José Manuel Félix Rodríguez

Dirigida por
Francisco Ortín Soler

Programa de Doctorado Ingeniería Informática

Oviedo, Enero 2015

RESUMEN

En el desarrollo de software se busca la división de las aplicaciones en módulos independientes entre sí, mejorando su reutilización, comprensibilidad, extensibilidad y mantenibilidad. Aunque los paradigmas de programación existentes utilizan distintas abstracciones para realizar esta división en módulos, en ocasiones estos mecanismos no son suficientes para modularizar todas las funcionalidades. El resultado es que parte del código de una misma funcionalidad acaba diseminado por toda la aplicación y entremezclado con el código de otras funcionalidades.

El principio de *Separation of Concerns* se basa en separar una aplicación en módulos distintos, de forma que cada uno de ellos abstraiga un único asunto, concepto o interés (*concern*), evitando que dos o más *concerns* se ubiquen en un único módulo, o alguno de ellos esté diseminado a lo largo de varios. Puesto que los paradigmas existentes tienen limitaciones a la hora de modularizar algunos *concerns*, surge el paradigma orientado a aspectos. Aquel código que no pueda modularizarse con los paradigmas clásicos será implementado mediante un aspecto, y un tejedor se encargará de combinar (tejer) su código con el de los componentes de la aplicación. De esta forma, tanto los componentes como los aspectos pueden abstraer los *concerns* de una aplicación sin sufrir los problemas del entremezclado y dispersión de código mencionados.

En función de los requisitos de un sistema, el tejido de componentes y aspectos se suele realizar o bien antes de ejecutar una aplicación (estáticamente), o bien dinámicamente en cualquier punto de ejecución del programa. Las técnicas utilizadas para realizar ambos tipos de tejido son significativamente distintas, causando diferencias importantes en el modo de codificar una aplicación en función del tejido (estático o dinámico) utilizado. En consecuencia, las plataformas orientadas a aspectos no separan el *concern* del dinamismo del tejedor, haciendo dependiente a la aplicación de este asunto. Adicionalmente, los pocos sistemas que permiten tanto el tejido estático como el dinámico poseen limitaciones importantes en el número de puntos de ejecución que pueden ser interceptados (tejidos), carencias en el tejido y destejido dinámico de aspectos y penalizaciones significativas en el rendimiento en tiempo de ejecución de las aplicaciones.

La principal contribución de la tesis presentada es el diseño, formalización e implementación de una plataforma de aspectos que separe el *concern* del dinamismo de tejido. Los aspectos y componentes podrán ser tejidos tanto estática como dinámicamente sin necesidad de ser modificados, permitiendo cambiar su dinamismo con la simple modificación de un parámetro pasado al tejedor. La plataforma presentada está formalizada mediante una semántica operacional independiente del lenguaje de programación base (el lenguaje que es tejido). El sistema permite el tejido y destejido en cualquier punto de ejecución del lenguaje base, ampliando así la funcionalidad del tejido dinámico existente en los sistemas actuales.

Posteriormente, se diseña el sistema conforme a esta semántica, y se implementa sobre una plataforma virtual estándar independiente del lenguaje y del sistema operativo, permitiendo además la adaptación de aplicaciones distribuidas. En la plataforma implementada se incluyen diversas optimizaciones novedosas para incrementar su rendimiento sin penalizar el consumo de memoria. Finalmente, se presenta una evaluación exhaustiva de la plataforma implementada, mostrando cómo el rendimiento de sus dos tejedores es superior a los tejedores más rápidos existentes, requiriendo menos recursos de memoria. La evaluación presentada también muestra cómo el tejedor de nuestra plataforma es el que posee la menor penalización en ambos tipos de tejido.

Palabras Reservadas

Separation of concerns, desarrollo de software orientado a aspectos, tejido dinámico, tejido estático, independencia del lenguaje, dinamismo, *meta-object protocol*, optimizaciones de rendimiento de ejecución, instrumentación de código binario, medición del rendimiento.

ABSTRACT

Designing modular systems is essential for managing software complexity and improving its reusability, understandability, extensibility and maintainability. Although each programming paradigm provides different modularization mechanisms, it is not always possible to separate different application concerns into separate modules. The result is that the source code of crosscutting concerns is tangled and scattered across the application.

The Separation of Concerns principle is aimed at dividing applications into modules, so that each single module represents a single concern. Two different concerns should not be placed in the same module, and one concern should not be located in different modules. Since the existing paradigms have limitations in supporting the Separation of Concerns principle, the main objective of the Aspect-Oriented Paradigm is to overcome this limitation. The code that could not be modularized with classic paradigms is implemented in an aspect. Then, an aspect weaver combines the code of the aspect with the code of the application components. Therefore, aspects and components modularize each application concern, avoiding the code tangling and scattering problems mentioned.

Depending on the requirements of each application, components and aspects may be woven before running the application (statically) or dynamically, at any execution point. The techniques used for both kind of weavers are significantly different. This dissimilarity commonly causes variations in the way applications are coded, depending on the type of weaver (static or dynamic) that is going to be used. Consequently, most aspect-oriented platforms do not separate the "weaving-dynamism" concern in software development, making the source code dependent on the type of weaver used. Additionally, the few aspect platforms that support both kinds of weavers show other limitations, such as a reduce number of join points, not supporting the dynamic weaving and unweaving at any execution point, and significant penalties on runtime performance.

The main contribution of this PhD thesis is the design, formalization and implementation of an aspect platform that separates the weaving-dynamism concern in the software development process. Application com-

ponents and aspects can be statically and dynamically woven, without changing their source code depending on the weaving strategy required. Weaving time is another parameter passed to the aspect weaver. The aspect platform presented is formalized with a small-step operational semantics, independent of the base language. Aspects can be woven and unwoven at any execution point of the base language, enhancing the functionality provided by current dynamic weavers.

Our aspect-oriented platform is designed following the proposed formalization. An implementation is provided in a language-agnostic widespread platform, providing the aspectization of distributed applications. The implemented system incorporates different innovative optimizations to improve the runtime performance of woven applications. Finally, a comprehensive evaluation of runtime performance and memory consumption is presented. The measurements show how our platform outperforms the existing systems for both static and dynamic weaving, consuming less memory resources. Moreover, our platform has incurred in the lowest performance penalties.

Keywords

Separation of concerns, aspect-oriented software development, static weaving, dynamic weaving, language neutrality, dynamism, meta-object protocol, runtime performance optimizations, binary code instrumentation, performance measurement.

TABLA DE CONTENIDOS

TABLA DE FIGURAS

1 INTRODUCCIÓN

1.1 Motivación

El Desarrollo de Software Orientado a Aspectos (DSOA) [Kiczales97] – *Aspect Oriented Software Development* (AOSD) – es una aproximación específica del principio general de Separación de Incumbencias o Asuntos [Hürsch95] –*Separation of Concern*s (SoC). DSOA ofrece soporte directo para modularizar diferentes *concerns*[1] que se entremezclan (*crosscutting*) en un sistema software, y se encuentran diseminadas (*scattered*) y entremezcladas (*tangled*) a lo largo de toda la aplicación junto con otras *concerns*. La modularización de los *concerns* transversales previene el entremezclado y diseminado del código fuente de la aplicación, haciéndola más fácil de depurar, mantener y modificar [Parnas72]. Algunos ejemplos típicos de incumbencias transversales son la persistencia, la autentificación, el sistema de *logging*, o el sistema de traza [Ortin04].

El proceso de integrar los aspectos dentro del código de la aplicación principal es llamado tejido (*weaving*), y un tejedor de aspectos (*aspect weaver*), o simplemente tejedor, es la herramienta encargada de llevar a cabo este proceso [Kiczales97]. El tejido puede ser realizado estáticamente (en tiempo de compilación o de carga) o dinámicamente (en tiempo de ejecución de la aplicación).

[1] En esta tesis usaremos el término *concern* para referirnos a asuntos, intereses, propiedades, conceptos o incumbencias con el objetivo de definir los requisitos o características demandados por una aplicación.

Existen numerosos entornos de programación que soportan DSOA proveyendo únicamente tejedores estáticos. En estos entornos, una vez la aplicación final ha sido compilada, tejida y se está ejecutando, no es posible añadirle nuevos aspectos o eliminar algunos de los existentes durante su ejecución. Sin embargo, existen escenarios donde es necesario adaptar las aplicaciones en ejecución en respuesta a requisitos emergentes que aparecen en ese momento. Ejemplos de uso de tejido dinámico son la gestión de la calidad del servicio (*Quality of Service*, QoS) de los requisitos en sistemas distribuidos CORBA [Zinky97], el manejo del *prefetching* de una caché web [Ségura03], el balanceo de la carga de aplicaciones que usen RMI [Stevenson08], y para cambiar la política de control de sistemas distribuidos [Popovici01]. En estos casos, los tejedores dinámicos son poderosas herramientas que facilitan la construcción de software adaptable en tiempo de ejecución.

Ambos tipos de tejedores, estáticos y dinámicos, tienen elementos a favor y en contra. Uno de los principales beneficios de los tejedores estáticos es el rendimiento que ofrecen en tiempo de ejecución. Dado que la combinación de componentes y aspectos es realizada antes de la ejecución de la aplicación, el código de ambos es entremezclado en la aplicación final, produciéndose tan solo una pequeña penalización del rendimiento en comparación con el desarrollo orientado a objetos tradicional [Garcia13] [Böllert99] [Haupt04]. En contraste, el tejido (y destejido) en tiempo de ejecución realizado por las herramientas dinámicas DSOA normalmente implica una penalización adicional en tiempo de ejecución, aunque proporciona una mayor adaptabilidad dinámica en el desarrollo de software [Garcia12]. Las aplicaciones pueden ser adaptadas en tiempo de ejecución dinámicamente, añadiendo o eliminando aspectos que modifiquen el comportamiento de la aplicación. Además, la adaptación dinámica es preferible en la fase de desarrollo de aplicaciones orientadas a aspectos, porque facilita la depuración interactiva siguiendo un desarrollo *edit-and-continue* [Dmitriev02]. El esquema de depuración *edit-and-continue* (también conocido como *fix-and-continue*) se basa en la capacidad para detectar un error en tiempo de ejecución, modificar el código original de los aspectos, recompilarlos, tejerlos en la aplicación en ejecución, y continuar con la ejecución del sistema adaptado [Eaddy05].

Varios trabajos previos han identificado los beneficios de integrar ambos tipos de tejedores en el mismo entorno de desarrollo [Blackstock04] [Böllert99] [Gilani07], obteniendo los beneficios de ambas aproximaciones en el proceso de desarrollo software. Además, si la plataforma DSOA está correctamente diseñada, los aspectos deberían poder ser cambiados de estático a dinámico, y viceversa, sin realizar cambios sobre su código fuente. El tejido

dinámico se podría usar cuando el sistema está siendo probado, haciendo el desarrollo del software más fácil [Eaddy05]. En el despliegue, los aspectos utilizados a lo largo de toda la ejecución de la aplicación serían tejidos estáticamente para mejorar el rendimiento en tiempo de ejecución de todo el sistema. Durante la ejecución de la aplicación, si surge algún requisito nuevo se podría tejer un aspecto dinámicamente que implementase dicho requisito. También se podrían (des)tejer temporalmente aspectos en base a las necesidades de adaptabilidad dinámica de la aplicación. Este tipo de aplicaciones podrían ser adaptadas a la vez por aspectos tejidos estática y dinámicamente.

1.2 Principal Contribución

La principal contribución de esta tesis es DSAW (*Dynamic and Static Aspect Weaver*), una plataforma orientada a aspectos que soporta tanto el tejido estático como el dinámico. DSAW ofrece el rendimiento en tiempo de ejecución del tejido estático, y el dinamismo y el desarrollo interactivo del tejido dinámico completo. La principal contribución de nuestro trabajo es la creación de una plataforma DSOA que ofrece los beneficios de ambos tejedores, estático y dinámico, de una forma transparente. Los aspectos y los componentes pueden ser desarrollados en DSAW sin tener en cuenta el dinamismo que requieren, haciendo posible la transacción de tejido dinámico a estático (y en el sentido contrario), sin realizar ningún cambio en el código fuente. En los primeros pasos del proceso de desarrollo de software, el tejido dinámico puede ser usado para facilitar la depuración interactiva de las aplicaciones. En el despliegue, si la adaptación en tiempo de ejecución no es requerida, los aspectos podrían ser tejidos estáticamente, obteniendo un elevado rendimiento. Los aspectos dinámicos también pueden ser utilizados si la aplicación necesita adaptación dinámica. Por lo tanto, DSAW aplica el principio de *Separation of Concerns* en su propio diseño, separando el *concern* del dinamismo del proceso de desarrollo software orientado a aspectos.

DSAW es un sistema independiente del lenguaje, que permite al desarrollador usar cualquier lenguaje de alto nivel. También es independiente de la plataforma, ya que su tejedor no utiliza ninguna característica particular de un sistema operativo. Además, el conjunto de puntos de enlace ofrecido por DSAW es más amplio que el ofrecido por los tejedores dinámicos existentes. Por último, nuestra plataforma está basada en la especificación estándar de una máquina virtual comercial y ampliamente utilizada. Esto hace a DSAW portable, siendo posible ejecutarla sobre cualquier implementación estándar.

1.3 Aportaciones

Las principales aportaciones de esta tesis doctoral son:

1. Una plataforma orientada a aspectos con tejido estático y dinámico. Existen trabajos previos que soportan ambos tejidos en el mismo sistema (§ 3.3) con la intención de proporcionar los beneficios de ambos enfoques al desarrollador, pero el dinamismo del tejido influye en el desarrollo de los aspectos y componentes. Nosotros proponemos e implementamos una nueva plataforma (Capítulo 5) que contempla ambos tipos de tejidos de forma homogénea, proporciona un amplio conjunto de puntos de enlace y es independiente del lenguaje y plataforma.

2. Rendimiento optimizado. El tejido dinámico provoca una penalización en tiempo de ejecución de la aplicación aspectizada. Aunque el tejido estático también provoca una penalización, ésta es mucho menor [Garcia13]. La plataforma DSOA desarrollada introduce nuevas optimizaciones (Capítulo 7), reduciendo las penalizaciones producidas por ambos tipos de tejido. Estas optimizaciones han conseguido que nuestra plataforma sea competitiva con otras de amplio uso en el desarrollo software hoy en día como AspectJ o JBoss AOP (Capítulo 8).

3. Separación del *concern* dinamismo. Los aspectos usados por la plataforma DSAW son independientes del tipo de tejido. Cualquier aspecto puede ser tejido estática o dinámicamente. Un aspecto tejido estáticamente puede pasar a ser dinámico sin realizar ninguna modificación sobre él, y viceversa. Con este planteamiento, se puede aplicar el tejido dinámico para el desarrollado rápido de aplicaciones y, cuando la aplicación se va a desplegar en producción, utilizar el tejido estático para mejorar el rendimiento de la aplicación. Del mismo modo, nuestra plataforma facilita la adaptabilidad de la aplicación, ya que en tiempo de ejecución ante la aparición de un requisito emergente, es posible utilizar el tejido dinámico para modificar el comportamiento de la misma.

4. Especificación formal de la semántica de la plataforma con tejido estático y dinámico. Basándonos en trabajos previos, hemos desarrollado una especificación de la semántica de DSAW (Capítulo 4) formalizando cómo se puede hacer tejido estático y dinámico en

una misma aplicación, así como la posibilidad de que sobre un punto de enlace se tejan varios aspectos.

5. Realización de un conjunto de aplicaciones para comparar plataformas orientadas a aspectos. Estas aplicaciones abarcan la realización de varios *benchmarks* sintéticos, la utilización de otros ya existentes, y la implementación de varias aplicaciones reales (§ 8.4) utilizando DSOA estático y dinámico para su desarrollo.

6. Evaluación del rendimiento y consumo de memoria de las plataformas orientadas a aspectos existentes. Hemos realizado una evaluación y comparación de un conjunto de plataformas que soportan tejido estático, dinámico y de ambos tipos (§ 8.1.2). Esta comparación muestra empíricamente los beneficios y carencias de la plataforma que hemos desarrollado.

1.4 Estructura del Documento

Esta tesis está estructurada de la siguiente forma. En el siguiente capítulo se presenta DSOA, la programación orientada a objetos y el tejido estático y dinámico de aplicaciones. En el Capítulo 3 se discute el trabajo relacionado, analizándolo en función a los objetivos buscados, descritos en § 1.3. El Capítulo 4 especifica la semántica formal de DSAW. En el Capítulo 5 se da una visión general de la arquitectura de DSAW, para mostrar el diseño completo del sistema en el Capítulo 6. El Capítulo 7 describe las optimizaciones realizadas para mejorar el rendimiento de la plataforma. La evaluación realizada de los tiempos de ejecución y consumos de memoria es detallada en el Capítulo 8. Finalmente, el Capítulo 9 presenta las conclusiones y el trabajo futuro.

Como capítulos auxiliares, el Apéndice A muestra la extensión del entorno integrado de desarrollo (IDE) Visual Studio para facilitar el desarrollo de aplicaciones con DSAW. El Apéndice B contiene las tablas completas de los tiempos de ejecución medidas, y el Apéndice C los resultados detallados de consumos de memoria. Por último, el Apéndice D presenta la lista de publicaciones derivadas del trabajo de investigación realizado para presentar esta tesis doctoral.

2 DESARROLLO DE SOFTWARE ORIENTADO A ASPECTOS

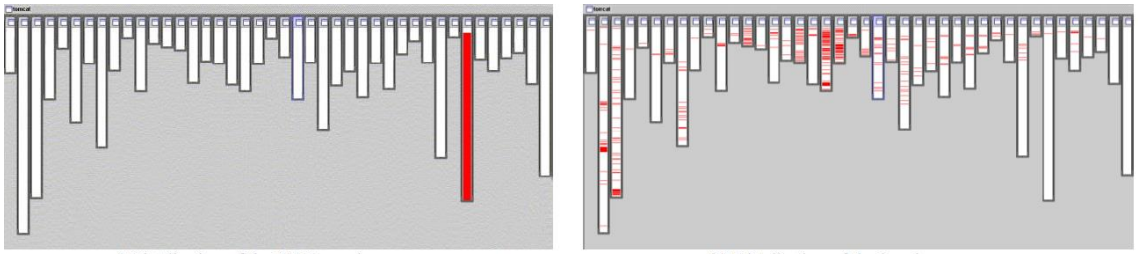
El diseño de sistemas modulares es fundamental para la gestión de la complejidad del software y mejorar su reusabilidad, comprensibilidad, extensibilidad y mantenibilidad [Meyer97]. Al nivel de implementación, los lenguajes de programación proporcionan mecanismos para realizar esta modularización. Algunas características comunes de los lenguajes orientados a objetos utilizados para facilitar la modularización de las abstracciones de las aplicaciones son métodos, clases, paquetes, espacios de nombres y anotaciones. Existen otros mecanismos de modularización, no soportados directamente por los lenguajes de programación, los cuáles proporcionan un nivel más alto de abstracción. Ejemplos de estos mecanismos son componentes, patrones de diseño, *frameworks* y patrones arquitectónicos.

En el proceso de desarrollo software, hay casos donde algunas de las abstracciones del sistema no pueden ser directamente modularizadas con los mecanismos proporcionados por el lenguaje de programación [Parnas72]. Por ejemplo, un algoritmo para ordenar un vector puede ser implementado en un método único de una clase. Sin embargo, en muchos casos, los *concerns* significativos en las aplicaciones software no pueden ser directamente modularizados con los mecanismos proporcionados por los lenguajes orientados a objetos usados normalmente [Hürsh95]. Ejemplos de tales *concerns* son las transacciones, la seguridad, el sistema de *logging* o la persistencia. Con el paradigma clásico de la orientación a objetos, el código que implementa estas

concerns se encuentra generalmente diseminado en muchos módulos de la aplicación, y entremezclado con otros *concerns*.

Los diferentes requerimientos (funcionales y no funcionales) demandados a una aplicación son llamados software *concerns* [Hürsh95]. El principio de diseño *Separation of Concerns* (SoC) [Parnas72] [Hürsch95] está dirigido a separar una aplicación informática en distintos módulos, de tal forma que cada uno gestione un *concern* aislado [Hürsh95]. La SoC gestiona la complejidad del desarrollo software, separando los *concerns* cuya implementación de otro modo estarían diseminadas en varios módulos y entremezcladas con el código de otros *concerns*. Los principales beneficios del principio de *Separation of Concerns* son: un nivel más alto de abstracción, la reutilización de *concerns*, una mayor legibilidad de cada *concern* aislada, y un software más fácil de mantener [Hürsch95].

Algunos *concerns* no pueden ser directamente modularizados en lenguajes orientados a objetos clásicos debido a que estos lenguajes no tienen la suficiente expresividad para implementarlos en módulos independientes [Felix2014]. En ese caso, las implementaciones de estos *concerns* están entremezcladas con múltiples abstracciones de un programa. Por esta razón, se dice que estos *concerns* están entremezclados a lo largo de (*cross across*) la aplicación –denominadas *crosscutting concerns* o transversales [Kizcales97]. La Figura 1 presenta un ejemplo real. Esta figura muestra la modularización de la implementación del servidor de aplicaciones Apache Tomcat [Kizcales03]. Cada barra vertical muestra la implementación de un módulo, y su tamaño es proporcional al número de líneas de código. La parte de la izquierda de la Figura 1 muestra en rojo las líneas de código cuyo *concern* es el parseado de documentos XML. Podemos ver cómo esa funcionalidad está situada en un único módulo. La parte de la derecha de la Figura 1 muestra la distribución del código fuente del *concern* de *logging*. Esto es un ejemplo de un *crosscutting concern*: su código fuente no está situado en un único módulo (*code scattering*), y cada módulo, incluyendo el de parseado de XML, contiene código de este *concern* (*code tangling*). Estos dos problemas de esparcimiento y entremezclado de código indican que la modularización a nivel de implementación no es apropiada, trayendo consigo limitaciones en la reusabilidad, comprensibilidad y mantenibilidad [Hürsch95].



a) Distribution of the XML parsing concern

b) Distribution of the logging concern

Figura 1: Distribución de *concerns* (código fuente por módulo) en la implementación de Apache Tomcat [Kizcales03].

La Figura 2 muestra un ejemplo de una aplicación de procesamiento de tarjetas de crédito, que usaremos como un ejemplo motivador a lo largo de este documento. La figura ilustra parte del código fuente inicial, donde varios *concerns* están entremezclados con el núcleo de la aplicación. Podemos identificar tres *concerns* diferentes: la funcionalidad central (la gestión del pago, que primero valida la tarjeta y después realiza la transferencia), un *concern* de *logging* (usando Apache log4net [Apache12]), y un *concern* de *profiling* (que mide el tiempo necesario para ejecutar cada método). Cada *concern* es mostrado en un color diferente en la Figura 2. En este programa orientado a objetos, el código fuente de los *concerns* de *logging* y *profiling* está enmarañado con la lógica de negocio, y diseminado sobre muchas partes de la aplicación –por ejemplo el código del *profiler* que calcula cuánto tiempo lleva ejecutar un método es repetido en varios métodos.

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

Figura 2: Código fuente con tres *concerns* entremezclados.

2.1 Programación Orientada a Aspectos

La separación de los diferentes *concerns* de una aplicación es un objetivo en todos los pasos del procesos de desarrollo software. La orientación a aspectos es un mecanismo particular para alcanzar esta meta. DSOA se enfoca en la identificación, especificación y representación de *concerns* entremezclados y su modularización a lo largo de todo el proceso de desarrollo software. Por lo tanto, la orientación a aspectos puede ser aplicada a la ingeniería de requerimientos, a la gestión de los procesos de negocio, a la arquitectura del sistema, al modelado y diseño [AOSD13]. La Programación Orientada a Aspectos (POA) (*Aspect-Oriented Programming*, AOP) está centrada en las técnicas de programación y las herramientas existentes que soportan la modularización de *concerns* al nivel de código fuente de la aplicación [Kizcales97].

La programación orientada a aspectos [Kiczales97] proporciona soporte explícito en un lenguaje para modularizar los *concerns* transversales al código fuente de la aplicación. La orientación a aspectos busca una mejor modularización de los *concerns* que la orientación a objetos o cualquier otro paradigma tradicional. Los aspectos expresan la funcionalidad transversal al sistema de una forma modular, permitiendo así al desarrollador diseñar un sistema libre de *concerns* transversales, proporcionando un único punto central para las modificaciones. La modularización de los *concerns* entremezclados evita el entremezclado del código fuente de la aplicación, siendo éste más fácil de depurar, mantener y modificar [Parnas72].

En la programación orientada a aspectos, las aplicaciones finales son construidas por medio de sus componentes más sus aspectos (*concerns* específicas). La tarea de componer ambas es realizada por el tejedor de aspectos. La Figura 3 muestra el esquema de modularización con el paradigma orientado a aspectos. Cada módulo puede representar un *concern* separado, evitando las desventajas mencionadas antes de código entremezclado y diseminado. Los tejedores de aspectos realizan la transición de los aspectos modularizados a la tradicional representación modular, generando la aplicación final.

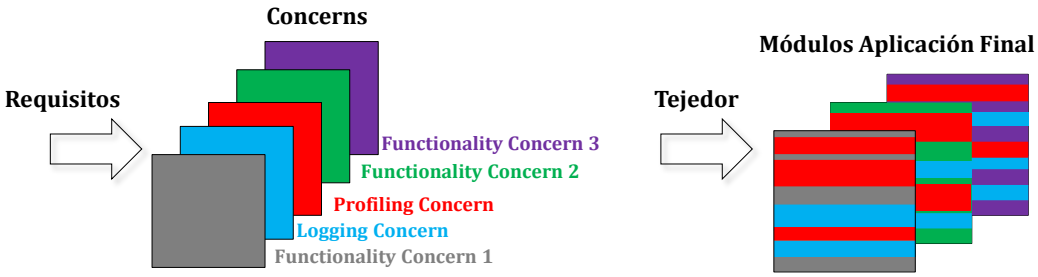


Figura 3: Modularización de incumbencias basadas en el desarrollo orientado a aspectos.

Como mostramos en la Figura 3, los diferentes *concerns* de una aplicación son identificados desde la fase de definición de requerimientos. Antes de su implementación, los *concerns* de una aplicación son separados conceptualmente. Estos *concerns* comprenden tanto requerimientos funcionales (es decir, del dominio del problema) como no funcionales (propios de distintas aplicaciones de distintos dominios, como persistencia o *logging*). El objetivo de POA es ubicar cada *concern* en un módulo separado. El tejedor de aspectos es la herramienta que toma los diferentes *concerns* de una aplicación y genera el programa final. Si la aplicación generada está codificada en un lenguaje orientado a objetos, como Java o C#, el código final de los *concerns* estará probablemente entremezclado y diseminado. La Figura 3 muestra como la implementación final de los *concerns* funcionales 1, 2 y 3 están situados en un único modulo (no están diseminados), al contrario del resto de *concerns*. Todos los concerns están enmarañados con el código de otros *concerns*. Por lo tanto, POA proporciona un nivel de abstracción más alto, ofreciendo al programador un mecanismo de modularización superior al proporcionado por la orientación a objetos. El tejedor de aspectos es la herramienta que transforma las abstracciones orientadas a aspectos en código orientado a objetos.

2.2 Clasificación de Herramientas POA

Considerando cuándo los aspectos son tejidos, las plataformas y tejedores orientados a aspectos pueden clasificarse en estáticos y dinámicos. Los tejedores estáticos realizan el tejido de los aspectos antes de la ejecución de la aplicación (en tiempo de compilación o de carga), mientras que las herramientas dinámicas proporcionan la aspectización de la aplicación en tiempo de ejecución.

En POA existen dos aproximaciones para representar los aspectos. La asimétrica diferencia entre el código base (clases tradicionales en la orientación a objetos clásica) del de los aspectos. Los aspectos representan *concerns* entremezclados que, gracias al tejedor de aspectos, pueden ser modulariza-

dos. Por lo tanto, en POA asimétrica, un aspecto debe ser usado para ser tejido con una clase (u otro aspecto). AspectJ es un ejemplo de una herramienta POA asimétrica [Eclipse13b]. En contraste, POA simétrica está basada en un único concepto clase/aspecto. Cualquier clase puede actuar como un aspecto y ser tejida con otra clase (o aspecto). Hyper/J es un ejemplo de POA simétrica [Ossher01].

En general, el tejido de los aspectos es realizado estáticamente, antes de la ejecución de la aplicación. AspectJ también proporciona tejido en tiempo de carga (*load time*), cuando las clases están a punto de cargarse en la memoria de la máquina virtual. También existen herramientas POA que permiten realizar el tejido y el destejido en tiempo de ejecución, cuando la aplicación está siendo ejecutada [Ortin11]. Estos tejedores dinámicos adaptan las aplicaciones en ejecución en puntos estables de la ejecución. Algunos ejemplos de herramientas dinámicas POA son las plataformas JAsCo, PROSE y DSAW [Vinuesa08].

La programación orientada a aspectos ha sido incluida en *frameworks* ampliamente usados en la actualidad como JBoss o Spring. Hay otras implementaciones, como JAC [Pawlak01], dirigidas a desarrollar una capa de middleware orientada a aspectos. Otro criterio para clasificar las herramientas POA es su dominio. Por un lado, están las herramientas de propósito general, tales como AspectJ, Compose* [Roo08] o CaesarJ [Aracic06]; y por otro lado estarían las específicas para un dominio: DiSL [Marek12], para el análisis de programas dinámicos; AGOL [Amor07], para sistemas multi-agentes; y LLDSAL [Payer12] para la generación dinámica de código y modificación de programas.

2.3 Léxico Común del Desarrollo en POA

Vamos a definir a continuación los conceptos de punto de enlace (*join point*), punto de corte (*pointcut*) y *advice* usados en general en POA, y de un modo más particular en AspectJ [Kizcales97]. Un punto de enlace es un punto de ejecución en el flujo de control de un programa. Un punto de enlace especifica cuándo, en la ejecución de un programa, el código del aspecto podría ser ejecutado. Algunos ejemplos de puntos de enlaces ofrecidos por AspectJ a los programadores son invocación de un método (*call*), ejecución de un método (*execution*), creación de un objeto (*new*), acceso a un campo (*get* y *set*) y gestión de excepciones (*handler*). Un punto de corte es un conjunto de puntos de enlace especificados con algún tipo de sintaxis, normalmente haciendo uso

de expresiones regulares. En cualquier momento que la ejecución del programa llega a uno de los puntos de enlace descrito por medio de un punto de corte, una pieza de código asociada a ese punto de corte es ejecutada. Estas piezas de código Java son llamadas *advice*. Un *advice* también indica si su código debería ser ejecutado *before*, *after* o en lugar de (*around*) el punto de enlace interceptado.

2.4 DSOA Dinámico

2.4.1 Limitaciones de los Sistemas Dinámicos DSOA

Las herramientas dinámicas DSOA realizan el tejido en tiempo de ejecución, consiguiendo la adaptación dinámica de aplicaciones mediante la adaptación de componentes por medio de aspectos. Muchas herramientas DSOA no soportan la adaptación en tiempo de ejecución, proporcionando tejedores estáticos (o en tiempo de carga). Una vez que la aplicación final ha sido generada (tejida), este tipo de sistema no será capaz de adaptarla en tiempo de ejecución (des)tejiendo nuevos aspectos. Hay diferentes casos en los cuales la adaptación de los *concerns* de una aplicación debería ser hecha dinámicamente, en respuesta a cambios en el entorno en tiempo de ejecución –algunos ejemplos fueron comentados en la Sección 1.1.

A diferencia de las plataformas de tejido estático, los tejedores dinámicos ofrecen algún tipo de adaptación en tiempo de ejecución para que los aspectos puedan ser (des)tejidos mientras las aplicaciones están ejecutándose. Algunos ejemplos bien conocidos de este tipo de tejedores son PROSE [Popovici01] [Nicoara05], JBoss AOP [RedHat12] y JAsCo [Suvee03]. Sin embargo, los sistemas dinámicos DSOA comúnmente adolecen alguna de las siguientes limitaciones:

1. **Adaptabilidad en tiempo de ejecución limitada.** Muchos sistemas dinámicos no ofrecen un nivel alto de adaptabilidad en tiempo de ejecución, requiriendo la especificación estática de los puntos de enlace que los aspectos van a usar en tiempo de ejecución (por ejemplo, Rapier-LOOM.Net [Frei04]), aunque la adaptación se realice, posteriormente, de forma dinámica. Si estos puntos de enlace no han sido identificados antes de la ejecución, su futura adaptación no puede llevarse a cabo.
   Otros ofrecen la adaptación dinámica de nuevos aspectos pero no

soportan el borrado dinámico de ellos (por ejemplo Wicca [Eaddy07b]).

2. **Limitado conjunto de puntos de enlace.** El conjunto de puntos de enlace que ofrecen muchos tejedores dinámicos es significativamente más pequeño que el ofrecido por los estáticos [Blackstock04] [Vinuesa03]. Un ejemplo significativo es JAsCo, una plataforma de tejido dinámico que proporciona un rendimiento excelente en tiempo de ejecución, pero sólo permite la intercepción en el punto de enlace `methodcall`.

3. **Portabilidad e interoperabilidad limitada.** Algunas implementaciones de los tejedores dinámicos obtienen un rendimiento extraordinario en tiempo de ejecución (bastante cercana a la de los estáticos). Este buen rendimiento lo obtienen modificando la implementación de una máquina virtual. Un ejemplo de este caso es la aproximación de la plataforma Steamloom que modifica la máquina virtual de investigación Jikes para detectar las sombras de los puntos de enlace y realizar el tejido de los *advices* en tiempo de ejecución [Bockisch06]. Sin embargo, la modificación de una máquina virtual específica implica limitaciones de portabilidad e interoperabilidad [Ogel05]. Esto significa que no es posible usar cualquier implementación estándar de la máquina virtual para la que fueron creados.

4. **Dependencia de la plataforma.** Hay tejedores dinámicos que sólo pueden ser usados en un sistema operativo o hardware específico. El tejedor dinámico Arachne para aplicaciones C es un ejemplo de este tipo de sistemas. Arachne rescribe el código binario de los ficheros ejecutables en tiempo de ejecución siempre que estos ficheros se ajusten al mapeado definido por el estándar Unix entre C y el lenguaje ensamblador x86 [Dounce04]. Aunque el rendimiento obtenido en tiempo de ejecución es extraordinariamente alto, esta técnica de tejido dinámico hace que Arachne sea dependiente de una plataforma específica [Gilani07].

Nuestra plataforma supera estas limitaciones, tal y como se describe en el Capítulo 9. DSAW implementa un tejedor dinámico que permite en tiempo de ejecución añadir y borrar aspectos, incluso en puntos de enlace que no fueron tejidos antes de la ejecución de la aplicación. Tanto el tejedor estático como el dinámico ofrecen el mismo número de puntos de enlace,

acercándose, aunque no siendo el mismo, a los ofrecidos por AspectJ. Además, DSAW es independiente del lenguaje y de plataforma.

2.4.2 Tejido Estático Donde sea Posible, Tejido Dinámico Cuando sea Necesario

Aunque las implementaciones existentes de los tejedores estáticos normalmente ofrecen beneficios en el rendimiento en tiempo de ejecución, también tienen algunas limitaciones. Un ejemplo de estas limitaciones se describe en el siguiente escenario. Si un programador está desarrollando una aplicación tejida estáticamente que requiera aspectos de *logging* o *testing*, la aplicación debe ser compilada, tejida, ejecutada y depurada. El contexto en tiempo de ejecución para depurar (aquél donde se detectó el error) debe ser reproducido y, después de esto, toda la información generada por el aspecto debe ser analizada. Si algún error ocurre durante este proceso, la aplicación debe ser modificada recompilada, retejida y ejecutada de nuevo [Böllert99] [Eaddy07c]. Además, la depuración de una aplicación orientada a aspectos es más difícil que la de una aplicación que no siga este paradigma: la semántica de la orientación a aspectos realiza invocaciones implícitas no descritas en el código fuente de los componentes, haciendo que el depurado sea menos intuitivo [Pothier08].

En contraste, el tejido dinámico permite al programador añadir y eliminar aspectos en los puntos exactos de ejecución deseados, produciendo menos información para ser analizada. Además, la ejecución de la aplicación no debe ser parada en caso de que queramos modificar un aspecto –puede ser modificado, recompilado y nuevamente tejido en tiempo de ejecución. Esta forma de trabajo ha sido referida como el desarrollo *edit-and-continue* [Eaddy05]. En caso de necesitar depurar la aplicación, se pueden eliminar los aspectos inyectados dinámicamente para una mejor comprensión de la ejecución.

Las ventajas e inconvenientes de ambas aproximaciones motivan el uso de ambas alternativas: el tejido estático dónde sea posible y el tejido dinámico cuándo sea necesario [Gilani07] [Böllert99] [Schröder-Preikschat06]. Una herramienta que soporte ambas técnicas debería definir los aspectos independientemente de su dinamismo (momento de tejido). Este planteamiento se beneficiará de la existencia de ambos tipos de tejido, estático y dinámico, en el mismo sistema, utilizando el más conveniente cuando sea adecuado.

DSAW es una plataforma orientada a aspectos que proporciona tejido estático y dinámico de forma homogénea. Su principal objetivo es alcanzar la neutralidad en el momento de tejido, y ser independiente de la plataforma y del lenguaje. La plataforma permite que para ambos tipos de tejedores, ni el código fuente de los aspectos ni de los componentes dependan del momento de tejido. Ni los componentes, ni los aspectos necesitan ser modificados si el programador necesita convertir un aspecto de estático a dinámico, y viceversa. De este modo, por ejemplo, es posible usar la orientación a aspectos para el prototipado rápido (tejido dinámico) y más tarde, antes del despliegue, optimizar la aplicación (tejido estático) sin realizar ningún cambio sobre su código fuente. El programador debería finalmente buscar el adecuado equilibrio entre rendimiento y flexibilidad en los requerimientos del sistema [Gilani07].

3 TRABAJO RELACIONADO

Actualmente existen sistemas de programación orientada a aspectos que ofrecen tejido estático, dinámico o ambos. En este capítulo analizaremos principalmente los sistemas que ofrecen ambas alternativas, no sin haber presentado anteriormente un breve análisis sobre las características dinámicas de los sistemas con tejido estático y una pequeña descripción de los sistemas con tejido dinámico.

3.1 Sistemas Estáticos

Existen multitud de sistemas estáticos. Entre ellos destaca el conocido AspectJ [Kiczales01] que es una extensión orientada a aspectos para Java. AspectJ es comúnmente usado como la referencia con la que comparar otros lenguajes orientados a aspectos, y su terminología ha influenciado ampliamente POA [Haupt06]. Muchos sistemas de tejido estático son versiones de AspectJ realizadas para otros lenguajes, como por ejemplo AspectC++ [Aspectc12] para C++ o AspectR para Ruby [Aspectr13].

A algunos tejedores estáticos se les han atribuidos capacidades dinámicas. Sin embargo, éstas se refieren únicamente a las propiedades del código que es tejido estáticamente para tomar decisiones en tiempo de ejecución, más que al tejido en sí. AspectJ ofrece algunas instrucciones de este estilo como, por ejemplo, `whithin`, `cflow` o `pertarget`. Si bien es cierto que estas instrucciones ayudan a desarrollar un software más dinámicamente adaptable, AspectJ no es un sistema que pueda ser clasificado como de tejido dinámico [Haupt06]. AspectJ inyecta código de los aspectos estáticamente a

los componentes, dotando de alta adaptabilidad al código inyectado, pero no permitiendo destejer éste, ni tejer nuevos aspectos.

3.2 Sistemas Dinámicos

En la actualidad, existe un menor número de sistemas orientados a aspectos que soportan tejido dinámico en comparación con aquellos que sólo soportan tejido estático. Muchos de estos sistemas dinámicos son proyectos de investigación como JAsCo [Suveé03], PROSE [Popovici01] [Nicoara05] o Steamloom [Bockisch06] más que sistemas comerciales como AspectJ. Aunque mejoran la adaptabilidad de una aplicación permitiendo tejer y destejer aspectos en tiempo de ejecución, casi todos ellos penalizan severamente el rendimiento del sistema global, además de proporcionar un conjunto significativamente más reducido de puntos de corte que los tejedores estáticos. Un análisis exhaustivo de los sistemas POA que ofrecen separación dinámica de aspectos puede consultarse en [Vinuesa07].

A modo de ejemplo vamos a analizar el sistema de tejido dinámico CAM/DAOP (*Component–Aspect Model/Dynamic Aspect–Oriented Platform*) [Pinto01] [Pinto02] para conocer las características de estos sistemas. Este sistema, desarrollado en la Universidad de Málaga, ofrece un modelo basado en componentes y aspectos, y una plataforma dinámica que soporta dicho modelo. El sistema consta de tres componentes principales:

- CAM. El modelo, especificado mediante UML, que sirve para diseñar aplicaciones basadas en componentes y aspectos.

- DAOP–ADL. Un lenguaje basado en XML que define la arquitectura del sistema, especificando los componentes, los aspectos y las reglas de composición entre ellos.

- DAOP. La plataforma dinámica que implementa el modelo y realiza la composición de forma completamente dinámica. Esta plataforma es la que ofrece servicios como la creación de una instancia de un componente, el envío de mensajes, etc.

El conjunto de puntos de enlace que soporta el sistema está basado en el Desarrollo de Software Basado en Componentes (*Component Based Software Development*) [Brown98] [Szyperski02], entendiendo los componentes como “cajas negras” que se intercambian mensajes y lanzan eventos. No se permite el acceso al comportamiento interno de los componentes, pues esto

se consideraría una violación del encapsulamiento. Por lo tanto, los puntos de enlace que soporta la plataforma son la creación y destrucción de instancias de componentes, el envío y la recepción de mensajes y el lanzamiento de excepciones y eventos. El sistema permite utilizar los tiempos *before y after* para la creación y destrucción de instancias y para el envío y recepción de mensajes, el tiempo *after* para el lanzamiento de excepciones, y el tiempo *around* para el lanzamiento de eventos.

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

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

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

El funcionamiento del sistema se basa en el uso que hacen de la plataforma los componentes. Cuando un componente necesita crear o eliminar una instancia de otro componente o enviar un mensaje o un evento se lo solicita a la plataforma, que es la que ofrece los servicios adecuados para ello. Estas acciones representan los puntos de enlace de la aplicación. En el momento que la plataforma recibe la solicitud, comprueba si se ha definido algún aspecto para el punto de enlace en cuestión (mediante los puntos de corte procesados) y, en el caso de ser así, ejecuta el *advice* correspondiente del aspecto.

Si durante la ejecución de la aplicación es necesario cambiar puntos de corte de algún aspecto que ya existe, basta con modificar el fichero de definición y añadir las modificaciones correspondientes al mismo, que serán procesadas de forma automática por la plataforma DAOP. Si es necesario añadir algún nuevo aspecto no contemplado previamente, se añadirá su especificación a la descripción de la arquitectura de la aplicación, indicando las interfa-

ces que implementa, sus clases, propiedades, etc. Esta información será procesada por la plataforma DAOP que la tendrá en cuenta de forma inmediata.

Aunque la implementación actual del sistema sólo trabaja con aplicaciones escritas en Java, el modelo CAM puede implementarse sobre cualquier plataforma [Fuentes03]. La definición de CAM/DAOP es independiente del lenguaje.

Este sistema de tejido dinámico es muy apropiado para aplicaciones distribuidas basadas en componentes, ya que el sistema permite realizar el diseño de una aplicación mediante CAM, traducirlo a DAOP–ADL y ejecutarlo sobre DAOP. El hecho de tener totalmente separadas la implementación de los aspectos de los puntos de corte es una ventaja a la hora de la posible reutilización del código ya que los aspectos no tienen que tener información del contexto. El proceso de adaptación se realiza de forma realmente dinámica (no en tiempo de carga), y no necesita ningún tipo de instrumentación de los componentes antes de realizar el tejido. CAM/DAOP es un sistema dinámico, permite activar y desactivar aspectos en tiempo de ejecución sin que sea necesario tener un conocimiento previo de ellos. Como contrapartida, el sistema está muy acoplado con la plataforma DAOP, no permitiendo su utilización sobre aplicaciones Java ya existentes.

El conjunto de puntos de enlace que soporta el sistema es no invasivo –por decisión de los autores–, impidiendo por tanto el acceso a la parte privada de la implementación. Dicho conjunto de puntos de enlace viene limitado por el hecho de soportar el desarrollo basado en componentes. Un componente debe comportarse como una caja negra que ofrece una serie de interfaces y requiere otras, pero no debe acceder a su parte privada. CAM/DAOP expresa los puntos de corte mediante un fichero XML externo al código de los aspectos, por lo que no se crean dependencias en el código y se facilita la posible reutilización de los aspectos dentro del sistema.

Existen otros sistemas de tejido dinámico avanzados como PROSE y JAsCo que son evaluados en § 8.1.2, así como otros más experimentales como CLAW, AORTA, Reflex o μDyner discutidos en [Vinuesa07].

3.3 Sistemas con Tejido Estático y Dinámico

3.3.1 JBoss AOP

JBoss AOP [Redhat12] forma parte del proyecto JBoss [RedHat13c]. Inicialmente, JBoss AOP era un marco de trabajo que permitía POA sobre Java en el entorno del servidor de aplicaciones JBoss. Hoy en día es además un sistema mixto que permite tanto tejido estático como tejido dinámico para el desarrollo de aplicaciones.

JBoss AOP no ha implementado ninguna extensión al lenguaje Java. En su lugar ofrece, mediante librerías, la definición de los puntos de corte, usando ficheros XML externos o anotaciones Java. El conjunto de puntos de enlace que soporta es muy rico, similar al de AspectJ, lo que dota al sistema de una gran versatilidad. El modo de expresar los puntos de corte es muy parecido al de AspectJ, haciendo uso de expresiones regulares.

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

El funcionamiento de JBoss AOP se basa en instrumentar la clase que se quiere ejecutar. Esta instrumentación se basa en realizar modificaciones al *byte-code* con el objetivo de añadir información extra a las clases para conectarlas con las librerías POA. JBoss AOP soporta dos tipos de instrumentación: *precompiled*, las clases son instrumentadas en un paso separado de compilación POA antes de ser ejecutadas; o *loadtime*, las clases son instrumentadas en el momento de ser cargadas.

Adicionalmente a los dos tipos de instrumentación, JBoss AOP ofrece tres modos diferentes de ejecución de las aplicaciones aspectizadas: *compiletime*, *loadtime* o *hotswap*. En JBoss AOP, los aspectos tienen que ser tejidos con las clases que se quieren aspectizar. Este tejido se puede realizar en tiempo de compilación (*compiletime*) usando el precompilador AOP, en el momento en el que la clase es cargada por la máquina virtual (*loadtime*) o en el momento en el que el código aspectizado va a ser ejecutado por primera vez (*hotswap*).

La instrumentación *loadtime* implica el procesamiento de los *byte-codes* de las clases en tiempo de carga. Esto significa que la inicialización de la aplicación es sensiblemente más lenta, ya que JBoss AOP tiene que hacer un trabajo extra en el momento en que las clases son cargadas. Además de esta penalización en el rendimiento, es necesario crear múltiples estructuras de datos Javassist que representan el *byte-code* de una clase particular, consumiendo una cantidad significativa de memoria adicional [Chiba00]. Aunque JBoss AOP intenta que parte de esa memoria sea liberada por el recolector de basura, parte de ella debe de ser mantenida en memoria.

Cuando se usa *hotswap*, las clases son instrumentadas lo mínimo necesario antes de ser cargadas, sin afectar el flujo de control. Si algún punto de enlace es activado debido a una operación de tejido, la clase afectada es tejida para que los interceptores y los aspectos añadidos puedan ser invocados. Esta forma de ejecución tiene los mismos inconvenientes de memoria que en el modo *loadtim*e, y además sufre de una penalización en el rendimiento [Hannad10].

Usando el modo *hotswap* de JBoss AOP es posible cambiar los enlaces a interceptores o aspectos en tiempo de ejecución. Se pueden desregistrar enlaces existentes, y activar nuevos enlaces en caliente, siempre que los puntos de enlace hayan sido instrumentados previamente.

En la Figura 4, primero se crea un `AdviceBinding` pasándole una expresión de un punto de corte. Después, se añade un interceptor usando su clase al enlace creado, y éste a su vez es añadido al `AspectManager`. El `AspectManager` iterará a través de todas las clases cargadas para ver los puntos de enlace que se emparejan con la expresión.

```
01: public org.jboss.aop.advice.AdviceBinding
02:        binding = new  AdviceBinding(
03:              "execution(POJO->new(..))", null);
04:      binding.addInterceptor(SimpleInterceptor.class);
05:      AspectManager.instance().addBinding(binding);
```

Figura 4: Código para añadir enlaces en JBoss AOP.

Aunque JBoss AOP es una plataforma madura que puede competir con AspectJ, si consideramos la plataforma como un sistema mixto, éste presenta las siguientes limitaciones con respecto al sistema buscado (§ 1.3):

- **Preselección de puntos de enlace (dinamismo limitado)**. En el código fuente de los componentes se debe indicar estáticamente dónde se enlazarán los aspectos y dónde se desenlazarán. El programador no puede tejer aspectos en tiempo de ejecución sobre otros puntos de enlace diferentes a los indicados. Además, una vez que se activa el tejido con el aspecto, éste no se puede eliminar hasta llegar al punto de desactivación del tejido.

- **Dependencia del dinamismo**. El funcionamiento de los dos tipos de tejido (estático y dinámico) no es homogéneo. En el tejido estático no es necesario modificar el código fuente de la aplicación, mientras que en el tejido dinámico se requiere añadir código para indicar cuándo se deben activar o desactivar los enlaces [RedHat13b].

- **Menor número de puntos de enlace**. El tejido dinámico ofrece un menor número de puntos de enlace que el estático [RedHat13b].

- **Necesidad del código fuente**. Es necesario cambiar el código fuente de los componentes para poder realizar un tejido dinámico de los mismos. Los aspectos e interceptores deben ajustarse a unas signaturas específicas y deben usar las librerías de JBoss AOP, tanto para el tejido estático como para el dinámico. Es decir, no se puede utilizar código compilado existente como un aspecto.

- **Penalización en el rendimiento y en el consumo de memoria**. El tejido dinámico sufre una penalización del rendimiento en tiempo de ejecución y un consumo de memoria mayor, debido a la utilización de Javassist [Hannad10].

- **Dependiente del lenguaje**. JBoss AOP sólo puede ser utilizado con el lenguaje Java.

3.3.2 Wicca

Wicca es otro ejemplo de sistema orientado a aspectos estático y dinámico [Eaddy07]. Ha sido desarrollado sobre la plataforma .NET haciendo uso del *framework* Phoenix –una infraestructura *back-end* de compilación [Microsoft13b]. Wicca adapta aplicaciones .NET permitiendo el tejido de aspectos con el objetivo de mejorar la modularidad y evolución, reduciendo las tareas tediosas y proclives a errores de la programación [Eaddy07]. Wicca puede transformar código fuente y *byte-code*, tanto en tiempo de compilación

como en tiempo de ejecución. También puede adaptar aplicaciones sin modificarlas, realizando un tejido de los puntos de enlace en memoria.

Wicca realiza el tejido estático por medio de la instrumentación de código. Para ello proporciona dos herramientas: Phx.Morph y wcs. Phx.Morph es un tejedor estático de *byte-codes* (instrumenta código binario) para aplicaciones .NET construido sobre Microsoft Phoenix. Los aspectos son especificados como ensamblados normales con atributos personalizados (*custom attributes*) `[Advice]`. Wcs es un tejedor de código fuente y un compilador. Es similar a AspectJ excepto que soporta C# en lugar de Java. Los aspectos son especificados con ficheros C# con atributos personalizados `[Advice]`. Puede ser usado en línea de comandos o programáticamente mediante una API.

Para conseguir el tejido dinámico se hace uso del API de depuración del CLR (*Common Language Runtime*), MDBG (*Microsoft Manager Debugger*), y de la herramienta AssemblyDiff (*Wicca's diff-and-update* API). Esta herramienta permite actualizar dinámicamente aplicaciones. Para ello utiliza la API *Microsoft Debugger Edit-and-Continue*. El tejido dinámico está liberado como una versión alfa, y actualmente no soporta el destejido de aspectos dinámicos.

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

Wicca también soporta tejido dinámico usando puntos de ruptura. Para ello, Phx.Morph teje 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 la API *Microsoft Debugger Breakpoint*. Cuando un punto de ruptura es alcanzado, Wicca obtiene los parámetros de contexto usando la *Microsoft Debugger* API y entonces ejecuta el código del *advice* en el espacio de proceso del programa en ejecución (mediante *Microsoft Debugger FuncEva*l API).

Algunas de las limitaciones que presenta el sistema son:

- **Implementación no homogénea**. Wicca utiliza dos herramientas con diferentes capacidades para realizar el tejido, Phx.Morph y wcs. Para el tejido en tiempo de compilación se puede usar cual-

quiera de las dos, pero en tiempo de carga o de ejecución sólo se puede utilizar Phx.Morph. Además, el tejido estático es más expresivo que el dinámico [Eaddy07b].

- **Necesidad del código fuente**. Para adaptar las aplicaciones, es necesario añadir anotaciones en el código fuente. Wicca impide tejer aspectos dinámicamente que no se hubiesen sido considerados estáticamente, y hace las aplicaciones dependientes del sistema de tejido [Eaddy07].

- **Dependencia de la plataforma**. Wicca hace uso de la API de depuración específica de una implementación de la plataforma .NET (el CLR), perdiendo así su potencial independencia de la plataforma.

- **Penalización de rendimiento**. El tejido dinámico tiene limitaciones significativas del rendimiento en tiempo de ejecución porque deshabilita la optimización JIT y habilita el soporte *Edit-and-Continue* del CLR [Eaddy07b]. Por ello, las aplicaciones se ejecutan en modo depuración con la consiguiente penalización de rendimiento.

- **Destejido de aspectos dinámicos**. Wicca no permite deshabilitar o destejer aspectos tejidos dinámicamente, en tiempo de ejecución de la aplicación [Eaddy07b].

3.3.3 AOP.NET

AOP.NET, anteriormente llamado NAop, es otra propuesta de tejedor estático y dinámico. Hasta el momento sólo se ha liberado un prototipo inicial de test muy limitado [Blackstock04]. Su diseño está basado en la utilización de un proxy para la decoración de componentes. Este proxy es usado tanto en el escenario de tejido estático, como en el del dinámico. El tejedor crea una clase proxy para cada clase de los componentes. El proxy generado tiene por objetivo adaptar el comportamiento de la clase decorada. Dependiendo de los puntos de corte, el proxy delega su funcionalidad en 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 antes de la ejecución de la aplicación, mientras que el tejedor dinámico lo hace dinámicamente.

No se han definido extensiones del lenguaje, ni se necesitan ficheros para especificar los puntos de corte. Toda la información relativa a POA está incluida en la declaración del aspecto. Los puntos de corte son expresados usando atributos personalizados empleados para anotar métodos, campos y clases con metadatos de cualquier lenguaje .NET.

Este sistema se encuentra en un estado inicial de desarrollo. Por ello posee importantes carencias [Blackstock04]. El prototipo actual es muy limitado, permitiendo sólo tejer un componente y un aspecto. El conjunto de puntos de enlace es escaso. La utilización de decoradores puede producir errores cuando un componente se referencia a sí mismo, saltándose el proxy que lo decora. Finalmente, la utilización de anotaciones implica la necesidad de tener disponible el código fuente.

3.3.4 LOOM.NET

El proyecto LOOM.NET [LOOM12], cuya versión actual es la 4.2 RC1, proporciona tejido estático y dinámico sobre el mismo núcleo implementado para la plataforma .NET [Schult03]. Inicialmente se creó el sistema de tejido dinámico Rapier LOOM.Net [Köhne05]. Ante los problemas de rendimiento y las restricciones impuestas para la adaptación dinámica, se desarrolló el sistema estático Gripper LOOM.Net [LOOM12].

Los puntos de corte en los aspectos son expresados por medio de atributos personalizados (*custom attributes*) de .NET, indicando las acciones de tejido al tejedor. Estos puntos de corte son evaluados en tiempo de ejecución o previamente al arranque de la aplicación realizando el tejido necesario. Esta evaluación es llevada a cabo mediante introspección, accediendo a los metadatos de la aplicación.

LOOM.Net soporta un conjunto de puntos de enlace más reducido que AspectJ. En tiempo de carga, la aplicación es tejida con los aspectos. El proceso de tejido se realiza sobre código intermedio, consiguiendo así independencia del lenguaje.

Las clases de los componentes a tejer deben cumplir una serie de requisitos para poder ser utilizadas en el sistema. Uno de esos requisitos es que los métodos de los componentes deben ser virtuales, o bien definirse mediante un interface. Otra restricción del tejedor dinámico es la imposibilidad de usar el operador `new` (o el operador o instrucción equivalente en el lenguaje que se haya implementado la aplicación), obligando a usar en su lugar la el

método de clase `Loom.Weaver.Create` para crear instancias de las clases tejidas. En la Figura 5 se muestran las dos formas de crear un objeto dependiendo del tipo de tejedor. Además, la clase `Base` tejida debe añadir atributos personalizados (no mostrado en la figura).

```
// Tejido dinámico
01: Base b = Loom.Weaver.Create<Base>(ta);
02: b.Hello(name);
03: Console.ReadLine();
```

```
// Tejido estático
01: Base b = new Base();
02: b.Hello(name);
03: Console.ReadLine();
```

Figura 5: Creación de objetos en LOOM.Net.

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. El tejido dinámico se realiza en el momento de la instanciación de la clase, quedando la clase tejida hasta la finalización de la ejecución del programa. En LOOM.Net no es posible destejer un aspecto tejido dinámicamente.

Los aspectos deben derivar de la clase `Loom.Aspect`. Un aspecto debe indicar, mediante atributos personalizados, el punto de enlace de la aplicación base donde debe ser añadido. Es decir, debe especificar el punto de corte. Aunque no proporciona el mismo conjunto de puntos enlace que AspectJ, permite adaptar muchas partes de una aplicación, como por ejemplo la creación de objetos, las llamadas a métodos o los accesos a propiedades. Además de los puntos de corte típicos, se han añadido otros nuevos como `Include` o `IncludeAll` [LOOM12].

El sistema desarrollado presenta algunas carencias en relación a los objetivos buscados:

- **Tejido no homogéneo**. El tejido estático y el dinámico es realizado de distinta manera. A modo de ejemplo, el tejido dinámico no permite utilizar el operador `new` (se debe invocar a `Loom.WeaverCreate`), mientras que en el estático su uso sí está permitido.

- **Adaptación de código existente**. La plataforma impone un conjunto de requisitos como que los métodos deben definirse virtuales (o mediante un interface). Estos requisitos tienen implicaciones negativas en la reutilización de código, ya que no es posible adaptar aplicaciones ya existentes sin modificarlas [Schult03] [Köhne05]. Además, obliga a utilizar *custom attributes* en las clases a te-

jer para ser instrumentadas, requiriendo en muchas ocasiones tener el código fuente de la aplicación.

- **Conjunto de puntos de enlace reducido**. 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**. Las reglas de tejido son especificadas mediante atributos personalizados en el código de los aspectos, resultando en un elevado acoplamiento que dificulta la reutilización de los aspectos [LOOM12].

- **Adaptación a nuevos aspectos surgidos dinámicamente**. Aunque el tejido se realice dinámicamente, no se permite tejer aspectos no contemplados en tiempo de diseño (creados posteriormente a la ejecución de la aplicación). Tampoco se permite el destejido de un aspecto tejido dinámicamente [Frei04]. No existe un verdadero dinamismo en el proceso de tejido, entendido como la adaptación a nuevos requisitos no previstos en el diseño del sistema.

3.3.5 Familia de Tejedores Dinámicos

En esta subsección analizamos la interesante iniciativa de "una familia de tejedores dinámicos" [Gilani04], realizada en Universidad de Erlangen-Nuremberg usando AspectC++ [Aspectc12]. Este trabajo está inspirado en los trabajos realizados sobre el ORB ZEN [Klefstad02].

La investigación se dirige principalmente a sistemas embebidos. Tradicionalmente, la investigación en el desarrollo de herramientas de tejido de aspectos se centra en ofrecer un mayor número de características DSOA. No obstante, las aplicaciones que se ejecutan de un modo embebido poseen restricciones significativas de recursos. En estos casos, la adaptabilidad dinámica de aplicaciones puede significar un consumo elevado de recursos y por ello no son comúnmente soportadas.

La propuesta de la Universidad de Nuremberg se basa en la creación de tejedores dinámicos, pero con la particularidad de que se realizarán *ad hoc* para los requisitos particulares de un tipo de aplicación específico. 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 satisfaga los requisitos de tejido particulares. Para la realización de este tejedor se tienen en cuenta características como: si los aspectos son conocidos, la interacción de los aspectos, si los puntos de enlace son conocidos, si los puntos de enlace son filtrados, o las características DSOA soportadas. De esta forma, se pueden reducir los recursos necesarios dependiendo de los requisitos de adaptabilidad, acomodándose al entorno donde se vaya a ejecutar la aplicación a adaptar.

El diseño de los tejedores se basa en que se debe ofrecer tanto tejido estático como dinámico, eligiendo uno u otro en función de los requisitos específicos de la aplicación y considerando su coste. Este diseño se basa, pues, en una familia de sistemas tejedores. Esta familia ha sido definida mediante una clasificación de los tejedores en función de las características DSOA que proveen.

AspectC++ [Aspectc12] es una extensión AOP del lenguaje C++, influenciada por AspectJ. Permite escribir aspectos utilizando el lenguaje AspectC++. Se ha implementado como un pre-procesador de C++, que permite transformar de AspectC++ a este lenguaje. Después del preprocesado, un compilador convencional de C++ es usado para generar las librerías compartidas.

En la Figura 6 se muestra el funcionamiento de esta plataforma. El tejedor estático de AspectC++ genera un documento XML con los puntos de enlace utilizados por los aspectos inyectados [Lohmann04]. Esta información es usada por el Monitor en Tiempo de Ejecución (*Run-Time Monitor*) para crear las estructuras de datos de cada punto de enlace, donde potencialmente se puede 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 aquéllos que estén 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.

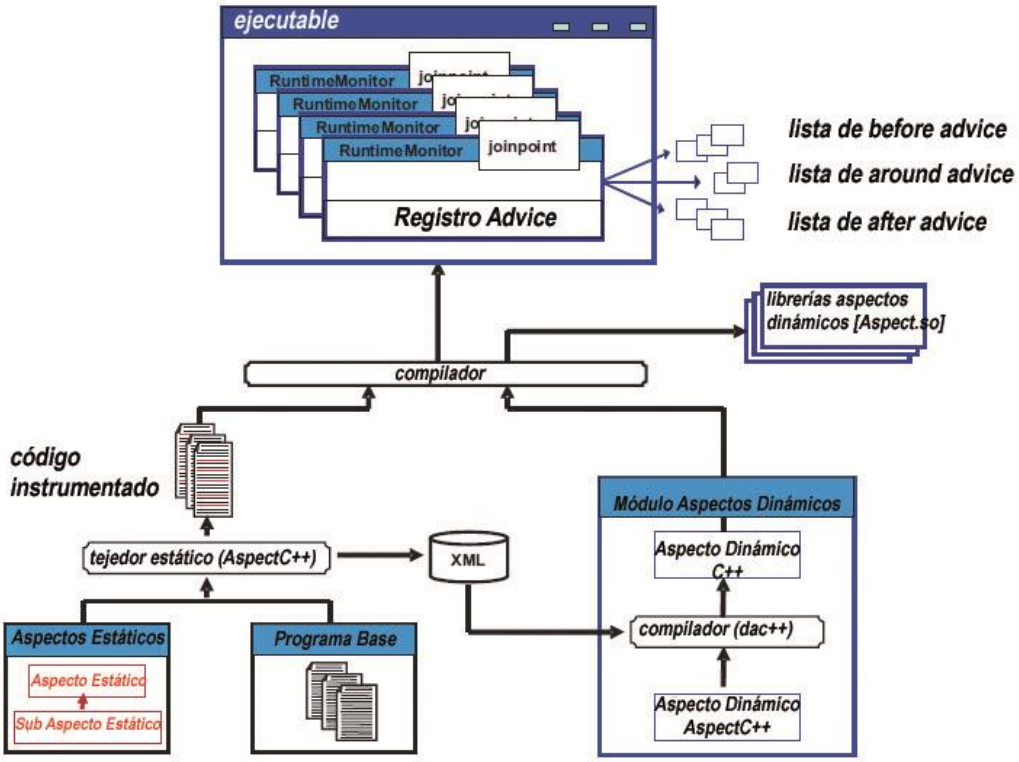


Figura 6: Arquitectura de la familia de tejedores dinámicos de aspectos [Gilani07].

En la implementación actual, los aspectos, tanto los estáticos como los dinámicos, son escritos usando AspectC++. Los aspectos son librerías compartidas. Éstas 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 utilizan cadenas de caracteres incluyendo comodines. El fichero XML contiene las signaturas de los puntos de corte y sus identificadores únicos.

La temática específica de esta iniciativa hace que existan las siguientes carencias en relación a los objetivos buscados en esta tesis:

- **Dependencia del lenguaje**. Sólo se puede usar el lenguaje de programación C++ para desarrollar las aplicaciones.

- **Dominio específico**. La solución propuesta están enfocada a los sistemas embebidos, dificultando su extensión a otro tipo de aplicaciones.

- **Reutilización de código existente**. Los aspectos se debe definir de forma obligatoria usando la extensión AspectC++, no permitiendo la utilización directa de código utilizado en otros escenarios.

3.3.6 RTZen

Se han realizado extensiones y modificaciones de *middlewares* para soportar comportamiento dinámicamente adaptativo. Dado que el rol tradicional de un *middleware* es ocultar la distribución de recursos y la heterogeneidad de las plataformas de la capa de negocio de las aplicaciones, es lógico tratar de adaptar el comportamiento relacionado con *concerns* entremezclados a lo largo del *middleware*, tales como la calidad del servicio, gestión de la energía, tolerancia a fallos, o seguridad [Sadjadi04]. Algunos ejemplos son ACT [Sadjadi04b], IRL [Baldoni02] o Eternal [Moser99].

Algunas investigaciones [Klefstad02] han sugerido la necesidad de usar sistemas de tejido de aspectos mixtos para adaptar las aplicaciones. Estas investigaciones han sido realizadas principalmente sobre la tecnología CORBA [CORBA12], concretamente sobre implementaciones de ORB (*Object Request Broker*) donde el número de parámetros a configurar es muy alto. En estos sistemas existen parámetros que no se pueden configurar estáticamente, ya que la información necesaria sólo es conocida en tiempo de ejecución, como por ejemplo la carga de trabajo [Klefstad02]. También existen entornos embebidos donde los recursos están muy limitados, la sobrecarga que implica el dinamismo no puede ser absorbida por el entorno o, peor aún, no soportan la utilización de configuraciones dinámicas.

El proyecto ZEN [Klefstad02], realizado en la Universidad de California en Irvine, es un ORB basado en Java en tiempo real, que se aplica principalmente a sistemas distribuidos en tiempo real y embebidos (*distributed real-time and embedded* DRE). Este proyecto remarca la importancia de poder utilizar configuraciones estáticas y dinámicas para poder configurar los múltiples parámetros usados por los ORBs, indicando los problemas de usar configuraciones de un solo tipo.

En investigaciones posteriores de este mismo grupo, se han utilizado componentes y aspectos tejidos estáticamente para la configuración de una versión posterior de ZEN denominada RTZen [Gorappa05]. Los componentes son usados como módulos software que encapsulan características del núcleo de CORBA en tiempo de ejecución. Estos componentes son clases o paquetes que pueden integrarse en cualquier momento en el ORB. Este diseño permite tener varias implementaciones de un mismo componente con diferentes versiones de la misma característica, y extender fácilmente el *framework* RTZen. Los aspectos son usados para algunas características que se encuentran diseminadas por el núcleo del ORB, como puede ser la gestión de la memoria en

tiempo real. Usando la separación de *concerns* se consigue maximizar la modularidad del ORB.

Aunque esta plataforma no es realmente un sistema POA, y sus trabajos por el momento se han centrado en la parte de adaptación estática, sus ideas han influido en otros trabajos posteriores [Gilani04b] que sí han utilizado y motivado la necesidad de sistemas POA mixtos.

4 SEMÁNTICA DEL SISTEMA

Existen distintos trabajos enfocados a describir la semántica de los lenguajes y sistemas orientados a aspectos [Walker03] [Jagadeesan06] [Lämmel02]. Sin embargo, la mayor parte de ellos crean la semántica como un todo, no permitiendo aislar las características especiales de la orientación a aspectos independientemente del lenguaje de programación utilizado. En este capítulo presentamos la formalización de la semántica de DSAW, independientemente del lenguaje de programación. La formalización presentada está basada en la semántica *Common Aspect Semantics Base* (CASB) propuesta por Djoko *et al.* [Djoko06]. CASB se basa en describir la semántica de las primitivas de la separación de aspectos independientemente del lenguaje base, introduciendo las mínimas construcciones necesarias para inyectar aspectos al lenguaje base utilizado. Nuestra semántica extiende y modifica distintos elementos de CASB para conseguir formalizar un sistema dotado de tejido tanto estático como dinámico.

4.1 Semántica del Lenguaje Base

La semántica del lenguaje base se describe en términos de una semántica operacional *small-step*, formalizada a través de la reducción $\rightarrow_b$ sobre configuraciones compuestas por un programa *C* y un estado Σ. Un programa *C* es una secuencia de instrucciones básicas (*i*) terminadas en la instrucción vacía ε, donde : denota la concatenación de 2 instrucciones:

$$C ::= i : C \mid \varepsilon$$

Una configuración es una tupla (C, Σ) donde Σ representa el estado del intérprete y C el programa por ejecutar. Σ se mantiene tan abstracto como sea posible, pudiendo contener las variables del entorno de ejecución, la pila y *heap* de ejecución, o cualquier otro elemento necesario para representar la semántica del lenguaje considerado y los detalles de su implementación.

Describimos cada paso de reducción (*small-step*) de la semántica del lenguaje de la siguiente forma:

$$(i : C, \Sigma) \rightarrow_b (C', \Sigma')$$

Donde i representa la instrucción actual a ejecutar y C las siguientes. Tras la ejecución, la configuración obtenida es (C', Σ'), donde i ya no tendrá que ser ejecutada en el siguiente paso. La configuración final, tras concluir la ejecución de todo el programa, tendrá la forma (ε, Σ).

4.2 Tejido Estático de Aspectos

La semántica de los aspectos es representada mediante una función ψ que, aplicada a la instrucción i actual, devuelve una lista de tripletas con todos los elementos de los aspectos tejidos en la instrucción i. Cada tripleta contiene un *advice* ϕ; un tipo t (*before*, *after* o *around*) indicando el tipo de aspecto; y ω, que indica cuándo el aspecto ha sido tejido (*static* o *dynamic*). Un *advice* es una estructura similar a un método, usada para definir el comportamiento adicional a ejecutar en un punto de enlace [Kiczales01]. Desde el punto de vista de la semántica del lenguaje, los puntos de enlace son aquellos elementos de la semántica que permiten a los aspectos ejecutarse [Kiczales97].

En CASB, ϕ es una función que toma Σ y devuelve un *advice*. Con el objetivo de mantener la definición lo más simple posible, asumiremos que los aspectos son directamente devueltos como instrucciones ejecutables (es decir, un *advice* es una secuencia de instrucciones). Así, ψ se define como:

$$\psi(i) = ((\phi_1, t_1, w_1) \cdots (\phi_n, t_n, w_n))$$
$$\text{where } t \in \{before, after, around\} \quad \text{and}$$
$$w \in \{static, dynamic\}$$

La función ψ puede ser vista como aquélla que decide qué puntos de enlace van a ser tejidos. Los aspectos estáticos son tejidos antes de que la aplicación se ejecute, y no van a cambiar mientras el programa esté corriendo; los aspectos dinámicos pueden ser añadidos y eliminados en tiempo de ejecución (§ 4.3). La función ψ devuelve la instrucción vacía ε cuando ningún aspecto ha sido tejido sobre la instrucción pasada como argumento. Nótese que este tipo de formalización permite tejer cualquier instrucción del lenguaje base, no sólo llamadas a funciones (o métodos).

La semántica del tejido es descrita en términos de reducciones → sobre configuraciones, incluyendo por tanto la semántica del lenguaje base ($\rightarrow_b$). La regla NoAdvice ejecuta la instrucción actual cuando ésta no tiene ningún aspecto tejido.

$$\frac{\psi(i) = \varepsilon \qquad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma) \rightarrow (C', \Sigma')} \text{ (NoAdvice)}$$

4.2.1 Aspectos *Before*, *After* y *Around*

Para describir la semántica de los aspectos de tipo *before*, *after* y *around*, primero vamos a mostrar el comportamiento dinámico en el escenario en el que sólo un aspecto es tejido sobre una instrucción. Después, generalizaremos el escenario a aquél en el que varios aspectos interceptan el mismo punto de enlace.

Con el objetivo de formalizar los aspectos de tipo *around*, añadimos al lenguaje base la instrucción *proceed*. Esta instrucción podrá ser utilizada únicamente en el código de los *advice* de tipo *around*. Para un aspecto *around,* el sistema ejecuta su *advice* en lugar de ejecutar la instrucción actual. Sin embargo, el código del *advice* puede ejecutar la instrucción original usando la instrucción *proceed*. Si *proceed* no es invocado, el *advice* remplazará la ejecución de la instrucción actual (ésta no será computada). En general, un *advice* de tipo *around* puede contener varios *proceeds* dando lugar a múltiples ejecuciones de la instrucción que el aspecto *around* ha interceptado.

Para representar el comportamiento de los aspectos de tipo *around* usamos una pila especial *P*, llamada la pila de *proceeds*. Esta pila es usada para describir la semántica de la instrucción *proceed*. La regla Around inserta el código del *advice* (ϕ) seguido por una instrucción pop_p, y pone la instrucción *i* actual en la pila de *proceeds* para que pueda ser posteriormente ejecutada por un *proceed*. La instrucción pop_p descrita con la regla Pop simplemente

elimina el tope de la pila de *proceeds* para restaurar la pila a su estado original.

$$\text{(AROUND)}\quad \frac{\psi(i) = (\phi, around, w)}{(i : C, \Sigma, P) \rightarrow (\phi : pop_p : C, \Sigma, \bar{i} : P)} \qquad \text{(POP)}\quad \frac{}{(pop_p : C, \Sigma, i : P) \rightarrow (C, \Sigma, P)}$$

Con el objetivo de prevenir que una instrucción *i* interceptada por un advice *around* pueda volver a ser interceptada, hemos introducido el concepto de instrucción etiquetada (denotada como $\bar{i}$). Una instrucción etiquetada $\bar{i}$ tiene exactamente la misma semántica que *i* excepto que no puede ser objeto de una operación de tejido. Formalmente:

$$\forall\,(i, C, \Sigma), (i : C, \Sigma) \rightarrow_b (C', \Sigma') \Rightarrow (\bar{i} : C, \Sigma) \rightarrow (C', \Sigma')$$
$$\forall\, i, \psi(\bar{i}) = \varepsilon$$

La regla PROCEED ejecuta la instrucción en la cima de la pila *proceed*. Esta instrucción es eliminada de la pila porque *i* puede ser el código dentro de un aspecto *around* cuyo *proceed* se referiría al tope de la pila *P*. Después de hacer *proceed*, *i* es puesta de nuevo (PUSH) en la pila, dado que el *advice* puede contener otros *proceeds*.

$$\text{(PROCEED)}\quad \frac{}{(\texttt{proceed} : C, \Sigma, i : P) \rightarrow (i : push_p i : C, \Sigma, P)} \qquad \text{(PUSH)}\quad \frac{}{(push_p i : C, \Sigma, P) \rightarrow (C, \Sigma, i : P)}$$

Es importante resaltar que los aspectos pueden ser tejidos en cualquier instrucción, no sólo en llamadas a funciones o métodos. También hemos asumido que el *advice* de un aspecto de tipo *around* puede ser interceptado por otro aspecto de tipo *around*, y que pueden existir aspectos anidados –AspectJ evita este caso utilizando restricciones sintácticas.

Después de definir la semántica de *around*, podemos describir la semántica de *before* y *after* por medio de una función *γ*, traduciendo cualquier tupla de un aspecto en otra tupla equivalente de un aspecto *around* de la siguiente forma:

$$\gamma(\phi, before, w) = (\phi : \texttt{proceed}, around, w)$$
$$\gamma(\phi, after, w) = (\texttt{proceed} : \phi, around, w)$$
$$\gamma(\phi, around, w) = (\phi, around, w)$$

Un aspecto de tipo *before* se traduce en uno de tipo *around* que inyecta el *advice* antes del proceder con el siguiente aspecto. Simétricamente, un aspecto de tipo *after* es traducido en otro de tipo *around* que sitúa el *advice* después de que el siguiente aspecto sea ejecutado.

4.2.2 Múltiples Aspectos en el mismo Punto de Enlace

Tras formalizar la ejecución de aspectos cuando sólo uno puede ser tejido en un punto de enlace, vamos a considerar el caso del tejido de varios aspectos en el mismo punto de enlace. Aunque podemos tejer varios aspectos de diferentes tipos (*after*, *before* y *around*), vimos cómo gracias a la función γ podemos traducir los tres a casos particulares de aspectos a *around*. Por lo tanto, sólo es necesario enfrentarnos al caso de que múltiples aspectos de tipo *around* sean tejidos en el mismo punto de enlace.

El modo en que se deben ordenar los aspectos tejidos en el mismo punto de enlace no es una tarea trivial. Por ejemplo, AspectJ por defecto ordena sus tripletas como *before*, *after* y *around*, aunque el programador puede modificarlo haciendo uso de la sentencia `declare precedence`. Para generalizar esto, una nueva función α ha sido previamente considerada en otros trabajos [Assaf08].

$$\alpha(\Sigma, (\phi_1, t_1, w_1), \ldots, (\phi_n, t_n, w_n)) = ((\phi'_1, t'_1, w'_1) \cdots (\phi'_n, t'_n, w'_n))$$

Esta función ordena el conjunto de tripletas tejidas en un punto de enlace, permitiendo la posibilidad de realizar una planificación dinámica basada en el contexto de ejecución (Σ es pasado como un parámetro), código del *advice* (ϕ), tipo de los aspectos (t), y su momento de tejido (ω). En la Sección 6.8 describimos cómo hemos definido la función α en nuestra implementación actual.

Utilizando esta función α, podemos especificar la regla AROUND* que realiza el tejido de cualquier número de aspectos en un mismo punto de enlace.

$$(\textsc{Around}^*)$$
$$\frac{\begin{array}{c}\psi(i) = ((\phi_1, t_1, w_1) \cdots (\phi_n, t_n, w_n)) \\ \alpha(\Sigma, (\phi_1, t_1, w_1), \ldots, (\phi_n, t_n, w_n)) = ((\phi'_1, t'_1, w'_1) \cdots (\phi'_n, t'_n, w'_n)) \\ \gamma(\phi'_1, t'_1, w'_1) = (\phi''_1, around, w'_1) \cdots \gamma(\phi'_n, t'_n, w'_n) = (\phi''_n, around, w'_n)\end{array}}{(i : C, \Sigma, P) \rightarrow (\phi''_1 : \ldots : \phi''_n : pop_p : C, \Sigma, \bar{i} : P)}$$

La semántica del tejido múltiple en un único punto de enlace difiere de AspectJ en el modo en el que la instrucción *proceed* funciona [Djoko06]. En AspectJ, el primer *advice* es ejecutado y el resto de los *advices* son colocados en la pila *proceed*. Esto significa que, si el código del primer *advice* ejecuta un *proceed*, el segundo *advice* será ejecutado, y así sucesivamente. En el caso de DSAW, cada *advice* es ejecutado en lugar de la instrucción *i*. Cuando uno de ellos ejecuta un *proceed*, entonces se ejecuta la instrucción *i*. Es importante resaltar que la semántica de AspectJ puede ser obtenida en nuestro sistema tejiendo aspectos de tipo *around* a otros aspectos (*before*, *after* o *around*), estableciendo una cadena de los aspectos tejidos.

4.3 Tejido Dinámico de Aspectos

Dado que el tejido dinámico puede modificar el conjunto de aspectos tejidos en un punto de enlace mientras que un programa está siendo ejecutado, vamos a ampliar la tupla de configuración con la función ψ. La reducción $\rightarrow$ ahora describirá cómo añadir y eliminar aspectos de un punto de enlace específico en tiempo de ejecución.

El tejido dinámico es ofrecido con dos nuevas instrucciones: `weave` y `unweave`. La primera instrucción añade un aspecto nuevo dinámicamente a la instrucción *i*, especificando el *advice* (ϕ) y el tipo del aspecto (t). $\psi[i \mapsto (\phi, t, dynamic)]$ denota la función ψ actualizada de forma que la instrucción *i* devuelva el aspecto (ϕ, *t*, *dynamic*), indicando que el nuevo aspecto ahora va a ser tejido en la instrucción *i*. Por lo tanto, la siguiente vez que la regla semántica AROUND* sea ejecutada, los nuevos aspectos dinámicamente tejidos serán tomados en cuenta.

$$\frac{\psi(i) = ((\phi_1, t_1, w_1) \cdots (\phi_n, t_n, w_n)) \qquad \psi' = \psi[i \mapsto ((\phi_1, t_1, w_1) \cdots (\phi_n, t_n, w_n)(\phi, t, dynamic))]}{(\texttt{weave}\ i, \phi, t : C, \Sigma, P, \psi) \rightarrow (C, \Sigma, P, \psi')} \quad (\text{WEAVE})$$

La instrucción *unweave* hace lo contrario: elimina (desteje) un aspecto previamente tejido en una determinada instrucción[2].Cabe destacar, que los aspectos tejidos y destejidos en tiempo de ejecución son siempre dinámicos.

[2] Consideramos la colección de aspectos tejidos en el mismo punto de enlace como un conjunto. Esta simplificación limita la posibilidad de tejer el mismo *advice* con el mismo tipo de aspecto, al mismo punto de enlace más de una vez. Para evitar esta limitación, se tendría que añadir un identificador único a cada aspecto.

(UNWEAVE)

$$\frac{\psi(i) = ((\phi_1, t_1, w_1) \cdots (\phi_j, t_j, dynamic)^{j \in [1,n]} \cdots (\phi_n, t_n, w_n)) \quad \psi' = \psi[i \mapsto ((\phi_1, t_1, w_1) \cdots (\phi_{j-1}, t_{j-1}, w_{j-1})(\phi_{j+1}, t_{j+1}, w_{j+1})^{j \in [1,n]} \cdots (\phi_n, t_n, w_n))]}{(\texttt{unweave}\ i, \phi_j, t_j^{j \in [1,n]} : C, \Sigma, P, \psi) \rightarrow (C, \Sigma, P, \psi')}$$

5 ARQUITECTURA

En este capítulo vamos a mostrar la arquitectura de DSAW, el sistema propuesto que satisface los requisitos descritos en § 1.3. Comenzaremos con una descripción somera de las responsabilidades de cada subsistema, detallando cada módulo en el Capítulo 6.

5.1 Join-Point Injector y Tejido Estático

Cualquier aplicación .NET existente, independientemente del lenguaje de programación usado para desarrollarla, puede ser adaptada por DSAW. La Figura 7 muestra cómo el inyector de puntos de enlace, *Join-Point Injector* (JPI), toma el código binario (*assemblies*) de la aplicación y, previamente a su ejecución, realiza el proceso de instrumentación del código en memoria.

Si el tejido de la aplicación es estático, un fichero con una especificación de puntos de corte debe ser pasado como parámetro al JPI (un punto de corte es un conjunto de puntos de enlace más, opcionalmente, algunos de los valores en el contexto de ejecución de estos puntos de enlace [Kizcales01]). En este caso, la aplicación es modificada con llamadas a funciones específicas (*advices*) de los aspectos definidos en el fichero de especificación de puntos de corte. Esta funcionalidad es la función ψ descrita en la Sección 4.2 cuando las reglas WEAVE o UNWEAVE no han sido ejecutadas, indicando qué puntos de enlace han sido estáticamente tejidos.

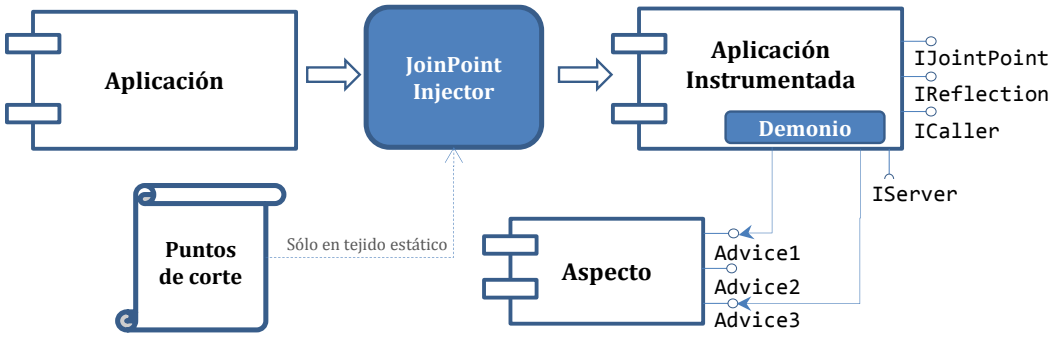


Figura 7: Instrumentación y tejido estático.

En caso de realizar tejido dinámico (o ambos), el JPI también instrumenta la aplicación con código para permitir en tiempo de ejecución la inyección de aspectos. Dado que el JPI desconoce en qué puntos de enlace el desarrollador de un aspecto dinámico puede estar interesado, se puede instrumentar toda la aplicación para que cualquier punto de enlace pueda ser interceptado en tiempo de ejecución –en nuestra formalización cualquier punto de enlace puede ser interceptado–, o sólo una parte. El JPI también añade a la aplicación la interface `IJoinPoint` que se usará posteriormente para saber los aspectos que se han tejido en cada punto de enlace. Además, se añade una implementación de la interface `IReflection` que permite a los aspectos tener acceso reflectivo a las aplicaciones en tiempo de ejecución. Esto es particularmente poderoso en el caso de los aspectos de tipo *around*.

5.2 Tejido Dinámico Local

El tejedor dinámico puede inyectar los aspectos en DSAW de dos formas: inyección local o remota. El tejido local permite la adición y supresión de aspectos a una aplicación en tiempo de ejecución, teniendo que estar ambos en la misma máquina. Por el contrario, el remoto también permite la adaptación de una aplicación, pudiendo estar los aspectos ejecutándose en otros equipos informáticos.

En el caso de inyección local, el JPI añade, además de los componentes descritos en el punto anterior, un hilo demonio (*daemon*) que va a analizar periódicamente el directorio de ejecución comprobando los cambios en los ficheros de especificación de los puntos de corte. Como vemos en la Figura 8, al ejecutar la aplicación, si el demonio detecta cambios en el fichero de los puntos de corte, lo leerá para saber dónde y qué aspectos deben ser inyectados. Dinámicamente, se generará el código de un *stub* que se encargará de realizar las llamadas a los *advices* desde la aplicación. Cuando la ejecución de

la aplicación alcance un punto de enlace que ha sido instrumentado, se comprobará si sobre ese punto de enlace se ha tejido un aspecto y, en ese caso, se ejecutará el método `Execute` del *stub* que, a su vez, ejecutará el *advice* adecuado.

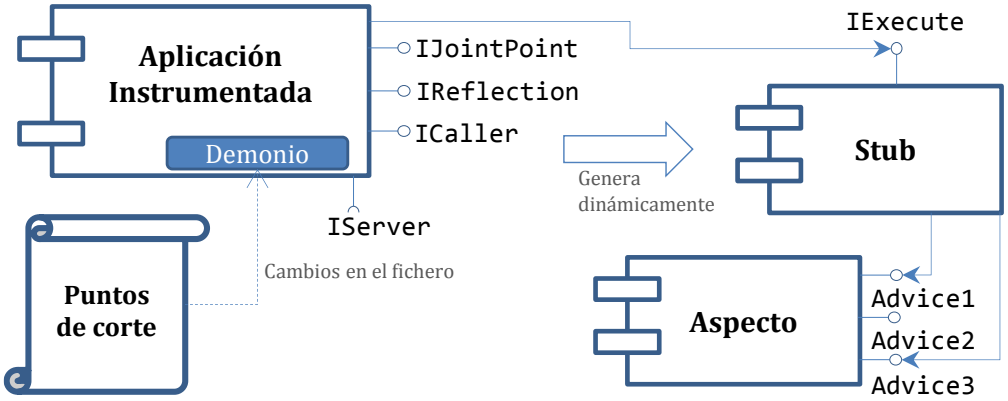


Figura 8: Tejido y destejido local y dinámico

La creación dinámica de este *stub* tiene dos beneficios. El primero, permite desacoplar totalmente el componente (aplicación) de los aspectos, puesto que el código que los enlaza (el *stub*) es generado dinámicamente personalizado para la estructura de ambos. El segundo beneficio es el rendimiento obtenido al no tener que utilizar reflexión ni comprobación dinámica de tipos (Capítulo 8).

5.3 Tejido Dinámico Remoto

El tejido dinámico remoto se basa en adaptar dinámicamente una aplicación a través de otra que se encuentre en otra máquina (sistema distribuido). Para el tejido remoto utilizamos un servidor de aplicaciones (*Application Server* AS). El AS coordina las aplicaciones y los aspectos dinámicos, que pueden estar ubicados en máquinas distintas. En la Figura 9 vemos que el funcionamiento es el mismo que para el tejedor local pero, en este caso, el JPI instrumenta la aplicación con código adicional para que la aplicación se registre en el AS al comienzo de la ejecución. Cuando se quiere tejer un aspecto dinámicamente, una aplicación lanzadora (*launcher*), que puede ser el mismo aspecto, registrará el aspecto en el AS utilizando `IServer`. El *launcher* leerá la especificación de los puntos de enlace y *advices* a tejer, enviando esta información a la aplicación (`IJoinPoint`) a través del AS. En tiempo de ejecución se generará el *stub* que interconecte la aplicación y los aspectos remotos. La ejecución de la aplicación será la misma que en el tejido dinámico local: al

alcanzarse un punto de enlace en el que algún aspecto haya sido tejido, se ejecutarán los *advice* asociados.

Para destejer los aspectos tejidos dinámicamente, en el caso de tejido local bastaba con eliminar el correspondiente fichero con la especificación de los puntos de corte. El demonio detectaba el cambio en el directorio de trabajo, y automáticamente elimina la información de esos aspectos utilizando `IJoinPoint`. En el caso remoto, se debe finalizar la ejecución de la aplicación lanzadora que dio lugar al tejido. Al finalizar este proceso, se desregistrará el aspecto del AS. El AS notificará el destejido a la aplicación a través de `IJoinPoint`, eliminando todas las intercepciones que la aplicación tuviese con el aspecto desregistrado.

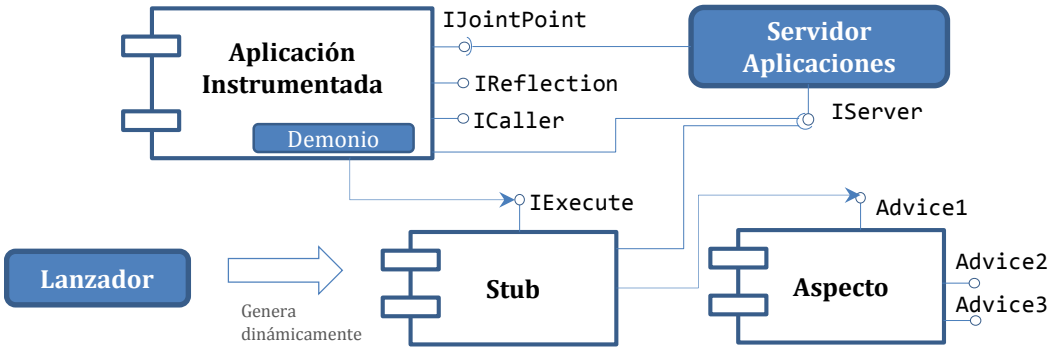


Figura 9: Tejido y destejido remoto y dinámico.

La arquitectura presentada no impone restricción alguna respecto al código y lenguaje utilizado para implementar las aplicaciones y los aspectos, pudiendo adaptar programas ya existentes sin necesidad de modificarlos. En el caso de los aspectos, existe alguna restricción que han de cumplir las signaturas de los métodos que implementen los *advices*. Dependiendo de la información que se va a enviar al aspecto, el código debe tener un número y tipo de parámetros determinados (Capítulo 6).

El código del *advice*, representado con la función ϕ en la Sección 4.2, es el código del aspecto que implementa el nuevo comportamiento a añadir al componente. El tipo de *advice* (*before*, *after* o *around*) es especificado en el fichero de descripción de los puntos de enlace (§ 6.4).

6 DISEÑO DEL SISTEMA

Una vez descrita la semántica del sistema (Capítulo 4) y la arquitectura compuesta de los módulos principales que asumen las responsabilidades (Capítulo 5) identificadas en la formalización, pasaremos a detallar el diseño de cada uno de las entidades identificadas en la arquitectura.

6.1 Aplicaciones

DSAW ha sido desarrollado sobre la plataforma .NET, siguiendo su guía de referencia [Ecma06] para no modificar ni extender su semántica. Al utilizar una plataforma virtual extendida, se garantiza la independencia del lenguaje de programación y sistema operativo utilizados. Adicionalmente, al utilizar la guía de referencia estándar, garantizamos el despliegue de nuestro sistema sobre cualquier implementación de .NET (como por ejemplo Mono, SSCLI, DotGNU o el propio CLR) [Lafferty03]. Por ello, no sólo es posible utilizar cualquier lenguaje de la plataforma .NET para desarrollar aplicaciones y aspectos, sino que también es viable crear aspectos y componentes en lenguajes de programación diferentes (Figura 10).

En la Figura 10 se aprecia un ejemplo práctico de utilización de DSAW. La funcionalidad central se puede implementar mediante componentes, utilizando el lenguaje de programación C#. Esta aplicación es compilada a binario de la plataforma .Net y, posteriormente, ejecutada en DSAW, tal y como se describió en el Capítulo 5. Adicionalmente, se pueden desarrollar dos *concerns*, uno de *profiling* y otro de *logging*, en los lenguajes Visual Basic e IronPython, respectivamente. Tras la compilación de ambos, éstos podrán adaptar la ejecución de la aplicación central utilizando una de las dos apro-

ximaciones de tejido dinámico. En función de los requisitos del sistema, el tejido de estos aspectos también podrá realizarse estáticamente.

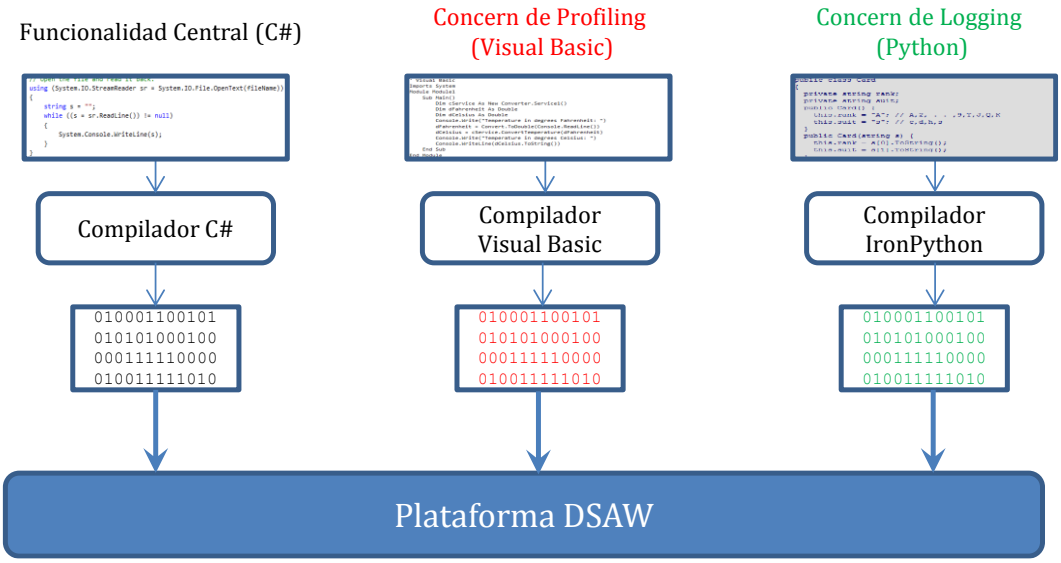


Figura 10: Independencia del lenguaje en DSAW.

DSAW realiza la adaptación del software a nivel del *byte-code* de la máquina virtual (conocidos como *assemblies*, ficheros ejecutables y bibliotecas dinámicas). El tejedor no requiere el código fuente de los componentes ni aspectos, siendo independiente del lenguaje. Al mismo tiempo, no es necesario implementar interfaces específicos o heredar de ninguna clase dada, tal y como sucede en varias de las plataformas analizadas en § 3.3. Cualquier aplicación (.exe) o librería (.dll) .NET puede ser ejecutada en DSAW sin cambiar su implementación –seguimos la idea de POJO de la *Java Persistence API* [Sun06].

El código C# en la Figura 11 muestra el componente central de un ejemplo real de una funcionalidad de pago de una tarjeta de crédito. Este componente será tejido posteriormente con varios aspectos estáticos y dinámicos utilizando DSAW. Se puede apreciar en el código cómo no existe referencia alguna a los aspectos, ni tampoco a entidades de la plataforma DSAW. El código no tiene ningún acoplamiento con estos dos elementos de la arquitectura y, por tanto, podría haberse obtenido de cualquier aplicación previamente existente.

```
01: namespace Payment {
02:     class PaymentService {
03:         public bool payment(CreditCard card, double amount) {
04:             if (!validateCard(card.Number, card.ExpDate,
                                  card.CardType))
05:                 return false;
06:             CardCompany company = CardCompany.getCardCompany(
                                                  card.CardType);
07:             return company.transfer(card, this.myAccount, amount);
08:         }
09:     }
10: }
```

Figura 11: Implementación en C# de un componente de ejemplo en la plataforma DSAW.

6.2 Join-Point Injector

El Inyector de Puntos de Enlace (*Join-Point Injector,* JPI) es el componente de la plataforma DSAW que realiza la instrumentación a nivel de *byte-code* de las aplicaciones (componentes). Tal y como se ha mencionado, el JPI añade a los componentes las referencias oportunas para poder ser utilizada en DSAW, permitiendo así adaptar cualquier aplicación binaria .NET existente. El *byte-code* .NET es la representación binaria del lenguaje CIL (*Common Intermediate Language*), lenguaje ensamblador de la máquina abstracta de .NET, independiente del lenguaje y plataforma [Ecma06].

Antes de ser ejecutada, la aplicación a adaptar es procesada en memoria por el JPI, añadiendo el código que permita la adaptación dinámica de los aspectos en tiempo de ejecución (Figura 12). Adicionalmente, el JPI también realiza el tejido estático (esta característica es descrita en la Sección 6.6). El JPI añade a la funcionalidad central: el registro y desregistro de la aplicación en el Servidor de Aplicaciones utilizado en el tejido dinámico remoto (§ 5.3); los puntos de enlace para que puedan ser interceptados dinámicamente (§ 5.1); una interfaz para poder tejer y destejer *advices* dinámicamente (§ 5.2); y el demonio para generar dinámicamente los *stubs* que interconectan la aplicación con los aspectos.

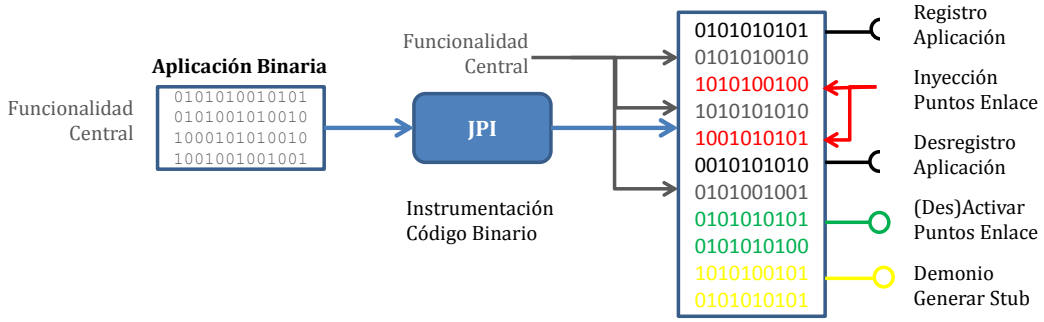


Figura 12: Instrumentación de *byte-code* realizada por el JPI.

Una limitación de los tejedores dinámicos existentes, comparados con los estáticos, es el conjunto de puntos de enlace ofrecidos, normalmente más reducido que el de los estáticos (§ 3.3). Esto es debido principalmente a la complejidad de implementar las adaptaciones en tiempo de ejecución frente al procesamiento estático de programas [Blackstock04]. Para evitar esta limitación, éste es un punto que hemos considerado en la creación de DSAW, hemos diseñado la plataforma para proporcionar un elevado conjunto de puntos de enlace. La colección de puntos de enlace ofrecidos es muy similar a la proporcionada por AspectJ [Kizcales01] (ver Sección 8.2). En estos momentos soportamos los siguientes puntos de enlace, tanto en tejido estático como en dinámico: ejecución de método (*Method Execution*) y constructor (*Constructor Execution*), llamada de método (*Method Call*) y constructor (*Constructor Call*), y lectura (*Field Get*) y escritura (*Field Set*) de campo y propiedad. Para todos ellos, además, proporcionamos los momentos de ejecución *before*, *after* y *around*.

El JPI realiza una instrumentación exhaustiva de los componentes con el objeto de añadir todos los puntos de enlace mencionados. Para ello, añade a los programas binarios un protocolo de meta-objeto (*Meta-Object Protocol*, MOP) [Kiczales91]. Un MOP es un sistema reflectivo que ofrece adaptación dinámica de la estructura y comportamiento de aplicaciones en ejecución [Ortin03]. El MOP inyectado proporciona la modificación dinámica de la semántica del programa procesado, permitiendo modificar, por ejemplo, el paso de mensajes o el acceso a los campos. De este modo, el MOP inyectado facilita la adaptación de aplicaciones en ejecución siguiendo la aproximación orientada a aspectos [Ortin04b].

El JPI analiza el *byte-code* para detectar sombras de puntos de enlace. Una sombra de un punto de enlace es un patrón de código binario en el que se detecta un punto de enlace y se podría inyectar una llamada a un aspecto. Estos puntos son donde realmente el JPI instrumenta el código [Masuhara03]. Cuando la sombra de un punto de enlace es detectada, un nuevo *byte-code* es añadido, inyectando un MOP a la aplicación. Este *byte-code* añadido comprueba en tiempo de ejecución si algún aspecto ha sido suscrito a ese punto de enlace; en ese caso, los aspectos suscritos serán llamados cuando el punto de enlace es alcanzado.

Adicionalmente, el JPI añade una implementación del interface `IJoinPoint` (Figura 12). Este interface provee métodos para tejer y destejer aspectos en un determinado punto de enlace de la aplicación instrumentada.

El JPI también inyecta el código para el hilo demonio (*daemon*) utilizado en el tejido dinámico local (§ 5.2). Este hilo demonio se encarga de detectar los cambios en los ficheros de especificación de los puntos de corte en tiempo de ejecución. El demonio utiliza nuestra librería `Caller`, encargada de generar dinámicamente el *stub* que interconecta la aplicación con el aspecto. El *stub* es generado utilizando CodeDOM [Microsoft13], y evita llamar al aspecto utilizando reflexión. El *stub* ofrece su funcionalidad mediante la implementación de un interface `IExecute`, que provee distintos métodos en función de los puntos de enlace interceptados.

En el caso de tejido dinámico remoto, el JPI modifica la aplicación para registrarla en el arranque, accediendo al interface `IServer` del Servidor de Aplicaciones (AS). También incluye una subrutina de desregistro que se ejecutará al finalizar la aplicación. En ambos casos, también se añade el interface `IReflection`. `IReflection` publica la información reflectiva (introspección) proporcionada por la plataforma .NET, permitiendo a los aspectos (externos a los componentes), inspeccionar la estructura de la aplicación en tiempo de ejecución.

Una vez realizada la instrumentación de la aplicación, ésta ya está lista para ser ejecutada.

6.3 Aspectos

Los aspectos también pueden ser desarrollados en cualquier lenguaje de programación de la plataforma .NET. Además, no existe ninguna diferencia entre los aspectos que van a ser tejidos estática o dinámicamente. DSAW no necesita el código fuente de los aspectos y, por tanto, permite tomar código binario de terceras partes e incluirlo en la plataforma como si fuesen aspectos.

Un aspecto puede ser desarrollado siguiendo dos aproximaciones. La primera requiere que el programador decida la información del componente que queremos que se pase al aspecto. Dependiendo de la funcionalidad del aspecto, éste necesitará saber más o menos información del componente interceptado. Esta aproximación permite afinar la información pasada y, por tanto, obtiene un mejor rendimiento.

La posible información a pasar desde el componente al aspecto ha sido clasificada en los siguientes grupos:

- *WithoutReflection*. El aspecto no necesita ninguna información del componente al que está tejido. En este caso el aspecto no recibe ningún parámetro.

- *StaticPart*. El aspecto va a utilizar información estática, como el nombre de la clase o el nombre del método interceptado en la ejecución del aspecto. La información que se le pasa al aspecto es el momento de ejecución, el tipo de punto de enlace, el nombre de la clase, el nombre del método y los tipos de los parámetros del método.

- *DynamicPart*. Cuando el aspecto necesita acceder a información dinámica, como por ejemplo el valor de los parámetros del método que dio lugar a la ejecución del aspecto. La información pasada como parámetros es el nombre del método original, una referencia al objeto original, los valores en tiempo de ejecución de los parámetros del método, y su valor de retorno.

- *Full*. Este es el caso, en el que el aspecto necesita acceder tanto a la información estática como a la información dinámica. Se le pasa la unión de los valores de *StaticPart* y *DynamicPart*.

En los puntos de enlace de lectura y escritura de campos, la agrupación anterior cambia. En este caso no es necesario pasar ni el valor de los parámetros ni el nombre del método. De forma similar, para los constructores, la referencia al objeto original no va a existir puesto que éste todavía no se ha creado.

En el caso de tejido estático, el nombre de los *advices* y su localización se conoce durante el proceso de instrumentación. No obstante, en el tejido dinámico estos datos no se van a conocer hasta la inyección de los aspectos en tiempo de ejecución. Para solucionar este problema, en la versión anterior de DSAW, llamada ReadyAOP [Vinuesa07], los aspectos tenían que implementar al menos uno de los siguientes interfaces: `IMethodCall`, `IMethodExecution` o `IPropertyFieldAccess`. Todos ellos tenían un solo método `exec` donde debía implementarse el *advice*. En esta versión de DSAW ya no utilizamos estas interfaces. Ahora, en el código instrumentado añadimos llamadas a un método de un interface genérico (`IExecute.Execute`) en los puntos de enlace instrumentados. En tiempo de ejecución, el tejido dinámico genera un código auxiliar o *stub* que invoca, implementando `IExecute`, conectando el punto de enlace de la aplicación a los aspectos (esta funcionali-

dad se ofrece mediante `IJoinPoint`). Así, cuando la aplicación alcanza un punto de enlace donde se ha tejido un aspecto dinámicamente, la aplicación ejecuta la llamada al método genérico `Execute` del *stub* con la información acerca del *advice* que se necesita ejecutar. Finalmente, el *stub* llama al aspecto tejido. El *stub* generado conoce dinámicamente el tipo de los parámetros a utilizar y el método elegido, por lo que es capaz de generar código que no haga uso de reflexión, obteniendo así un mejor rendimiento (Capítulo 8).

Siguiendo con nuestro ejemplo, vamos a suponer que el programador está interesado en adaptar la aplicación en tiempo de ejecución. Supongamos que, en un cierto punto de ejecución, el rendimiento de la aplicación de pago con tarjeta de crédito puede ser pobre. Bajo estas circunstancias, un aspecto de *profiling* podría ser añadido en tiempo de ejecución para analizar la causa del bajo rendimiento, y destejido más tarde cuando el rendimiento de la aplicación se recupere. La información de *profiling* generada podrá ser utilizada por el programador para mejorar el rendimiento de la aplicación.

El código fuente C# de este aspecto dinámico es mostrado en la Figura 13. La información pasada en este caso es la identificada como *StaticPart*. Nótese cómo el *concern* de *profiling* ha sido claramente modularizada aparte de la funcionalidad central mostrada en la Figura 11. Además, la única referencia externa no propia del aspecto es la clase `ReflectiveStaticInformation`, utilizada en este modo de creación de aspectos para obtener un mayor rendimiento.

```
01: namespace Payment {
02:    public class ProfilerAspect {
03:        private double startTime = 0;
04:        public void profiling(ReflectiveStaticInformation aux) {
05:            if (aux.time.CompareTo(Interfaces.Time.Before)==0)
06:                startTime = DateTime.Now.Ticks;
07:            if (aux.time.CompareTo(Interfaces.Time.After)==0)
08:                measure(aux.fieldName,
                              (DateTime.Now.Ticks-startTime)
09:                                 / TimeSpan.TicksPerMillisecond;
10:        }
11:    }
12: }
```

Figura 13: Implementación en C# del aspecto de *profiling* en la plataforma DSAW.

El segundo modo de crear un aspecto es usando una aplicación o librería existente sobre .NET (*assembly*). Este aspecto podría ser un componente de un tercero o un aspecto estático que el programador está interesado en tejerlo dinámicamente. Con este modo de creación de aspectos, nuestra plataforma hace posible convertir un aspecto de estático a dinámico y viceversa,

sin cambiar su implementación. De esta forma es cómo nuestra plataforma ofrece una separación transparente de la incumbencia de dinamismo.

Con esta aproximación DSAW toma los binarios del aspecto y un documento XML describiendo el *advice*, y crea dinámicamente la implementación apropiada de un *stub* que permita la comunicación entre la aplicación y el aspecto. En este caso, debería crearse un documento XML describiendo los *advices* de los aspectos [Kizcales01] (la estructura de este documento XML es descrita en las Secciones 6.4 y 6.7). Estos dos elementos (código del aspecto más tipo de *advice*) comprenden los dos primeros elementos de la formalización presentada en el Capítulo 4: ϕ y t. En este caso, ω (tiempo de tejido) es dinámico.

6.4 Especificación de Puntos de Corte

Hemos visto cómo cualquier aplicación o librería existente puede ser usada como componente o aspecto en DSAW, aunque es necesario describir el *mapping* entre los puntos de enlace y los aspectos por medio de los puntos de corte. En DSAW, los puntos de corte son especificados por medio de documentos XML que describen cómo se deben asociar los puntos de enlace y los aspectos. El esquema de estos documentos XML es una evolución del usado por la plataforma Weave.NET [Lafferty03]. Hemos ampliado esta especificación para proporcionar un conjunto más amplio de puntos de corte, acercándonos al suministrado por AspectJ. Como describimos en el Apéndice A, hemos desarrollado un *plug-in* para Visual Studio que automáticamente genera esos documentos XML, haciendo más fácil la programación orientada a aspectos con DSAW.

La descripción del *mapping* entre componentes y aspectos en ficheros XML separados proporciona una separación completa (sin acoplamiento) entre ambos, mejorando la reutilización tanto de los aspectos como de los componentes. De hecho, los aspectos pueden ser tratados como componentes. Los aspectos también pueden ser adaptados por otros aspectos, estática o dinámicamente, sin tener en cuenta el lenguaje de programación utilizado para su implementación.

La Figura 14 muestra el fichero del punto de corte de nuestro ejemplo de *profiler* dinámico. Para añadir la funcionalidad de *profiling*, hemos usado los momentos de ejecución *before* y *after* del punto de enlace *MethodCall* (`methodcall`). Estamos interesados en cualquier tipo de valor de retorno (`*`)

–se pueden usar expresiones regulares– y sólo aquellos métodos que sean públicos (`public`) –todos los `flags` de los miembros usados en CLI [Ecma06] son soportados. Nuestro ejemplo adapta todos los métodos (`qualified_method_name`) en cada clase (`class` e `identifier_name`) del espacio de nombres `Payment` (`namespace` y `type_name`), excepto el método `Main` (`name`, `not`, `identifier_pattern` e `identifier_name`).

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <aspect_definitions xmlns="urn:gramaticapointcuts-schema" ...>
03:    <pointcut_definition id="dynamicMethodCall">
04:       <aspectized_type>StaticPart</aspectized_type>
05:       <time>before</time><time>after</time>
06:       <joinpoint_type><methodcall>
07:             <method_signature>
08:                <return_type><type_name>*</type_name></return_type>
09:                <method_flags><public/></method_flags>
10:                <qualified_method_name>
11:                   <qualified_class>
12:                      <namespace><type_name>Payment
                         </type_name></namespace>
13:                      <class><identifier_name>*
                         </identifier_name></class>
14:                   </qualified_class>
15:                   <name><not><identifier_pattern>
                            <identifier_name>Main</identifier_name>
16:                   </identifier_pattern></not></name>
17:                </qualified_method_name>
18:             </method_signature>
19:       </methodcall></joinpoint_type>
20:    </pointcut_definition>
21:    ...
22: </aspect_definitions>
```

Figura 14: Descripción XML de un punto de corte para la inyección dinámica de un aspecto.

La especificación XML de la Figura 14 se le pasará al JPI para realizar la instrumentación de los puntos de enlace, y añadir el MOP necesario para permitir la adaptación en tiempo de ejecución.

Cuando se ejecute la aplicación será necesario indicar los *advices* que se quieren inyectar dinámicamente en los puntos de enlace. Para ello, en la línea 21 de la Figura 14 se añadirá la localización del aspecto, mostrado en la Figura 15.

En la línea 1, mediante el atributo `idTypeOfInjection`, indicamos que este *advice* se va a usar para tejido dinámico. Los siguientes atributos especifican la librería donde se encuentra el *advice* (`AspectPayment.dll`) y su nombre (`ProfilerAspect`). Por último, el atributo `idRef` enlaza el *advice* con la especificación del punto de corte que se usará para activar los puntos de enlace.

```
01: <advice_definition idAdvice="AspectProfiling"
                       idTypeOfInjection="DynamicInjection">
02:    <assembly>AspectPayment.dll</assembly>
03:    <type>Aspects.ProfilerAspect</type>
04:    <behaviour>profiling</behaviour>
05:    <priority>1</priority>
06:    <pointcut_definitionRef idRef="dynamicMethodCall"/>
07: </advice_definition>
```

Figura 15: Localización del *advice* para inyección dinámica.

6.5 Ejecución del Sistema

6.5.1 Local

Para el tejido dinámico local, el hilo demonio (*daemon*) previamente inyectado por el JPI es el encargado de detectar la necesidad de tejer dinámicamente los componentes y aspectos. El demonio detecta las variaciones de los ficheros XML con las especificaciones de puntos de corte y *advices*. Si un nuevo fichero aparece, se inyectan estos nuevos aspectos a la aplicación; si un fichero se elimina, se destejen los aspectos referenciados por ese fichero de la aplicación.

Siguiendo con el ejemplo del pago de una tarjeta de crédito, así es cómo la aplicación, el aspecto de *profiling* con tejido dinámico local y el hilo demonio trabajan juntos para adaptar en tiempo de ejecución el sistema de pago con tarjeta de crédito (ilustrado en Figura 16):

1. La aplicación, una vez procesada por el JPI, es ejecutada. En el arranque el hilo demonio, añadido durante la fase de instrumentación, se activa. Este hilo demonio comprobará periódicamente los cambios en los ficheros XML de especificaciones de puntos de corte.

2. Cuando el desarrollador quiere realizar *profiling* de la aplicación en ejecución (por ejemplo, porque detecta bajo rendimiento en tiempo de ejecución) preparará un fichero XML especificando los puntos de corte y la localización del *advice*. También implementará el aspecto de *profiling*. Este fichero XML se copia al directorio donde el hilo demonio está comprobando los cambios en los ficheros de puntos de corte (por defecto es el directorio donde se inició la aplicación).

3. El demonio detecta el nuevo fichero XML y lee la especificación de los puntos de corte, cuyo formato es similar al mostrado en la Figura 14. Utilizando CodeDOM [Microsoft13], genera en tiempo de ejecución un *stub* encargado de hacer las llamadas a las funciones del aspecto (*advices*). También almacena la correlación entre el nombre del *advice* y su llamada a través del *stub*.

4. El demonio procesa el documento XML buscando los puntos de corte pasados. Los puntos de enlace descritos por estos puntos de corte son activados en la aplicación por medio del MOP inyectado por el JPI (`IJoinPoint`). En esos puntos de enlace se almacenan las llamadas a los *advices* oportunos a través del *stub* generado. Esta activación implicará la futura invocación de los aspectos tejidos dinámicamente. Ésta es la instrucción `weave` *i*, ϕ, *t* formalizada en Sección 4.3, donde *i* y *t* son, respectivamente, el punto de enlace y el tipo de *advice* descritos en el fichero XML; y ϕ es el código del aspecto.

5. Cuando la ejecución de la aplicación de la tarjeta de crédito alcanza un punto de enlace tejido (por ejemplo, la llamada al método `payment`), llama al aspecto de *profiling* invocando al método `Execute` del interface `ICaller.IExecute`. Entonces, la aplicación pasa el control al aspecto enviando información relativa al punto de enlace (en nuestro ejemplo se pasa información *StaticPart*, siendo posible pasar otra tal y como se describió en § 6.4). El aspecto de *profiling* en ese momento recibirá el flujo de la ejecución para almacenar la hora y medir el rendimiento dinámico.

6. Aunque en este ejemplo no se utiliza, el aspecto puede usar la referencia a la aplicación para acceder a ella por medio de reflexión. El JPI añade el interface `IReflection` para este propósito. De esta forma, el aspecto puede inspeccionar o modificar los valores de cualquier campo o propiedad de la aplicación, así como invocar a cualquiera de sus métodos (no sólo el método interceptado).

7. Cuando ya no sea necesario que el aspecto se ejecute, el desarrollador puede eliminar el fichero XML que originó el tejido. Para ello, debe suprimirlo del directorio donde el demonio está controlando los cambios en los ficheros de corte XML. Esta eliminación implicará el destejido del aspecto.

8. El hilo demonio, al detectar una variación en los ficheros de especificaciones de puntos de corte, desactiva en la aplicación los puntos de enlace previamente activados con el aspecto. Esos puntos de enlace podrían todavía seguir activados por otros aspectos diferentes que los hayan activado. Esta operación es la formalizada con la semántica de la instrucción `unweave` i, ϕ, t en la Sección 4.3.

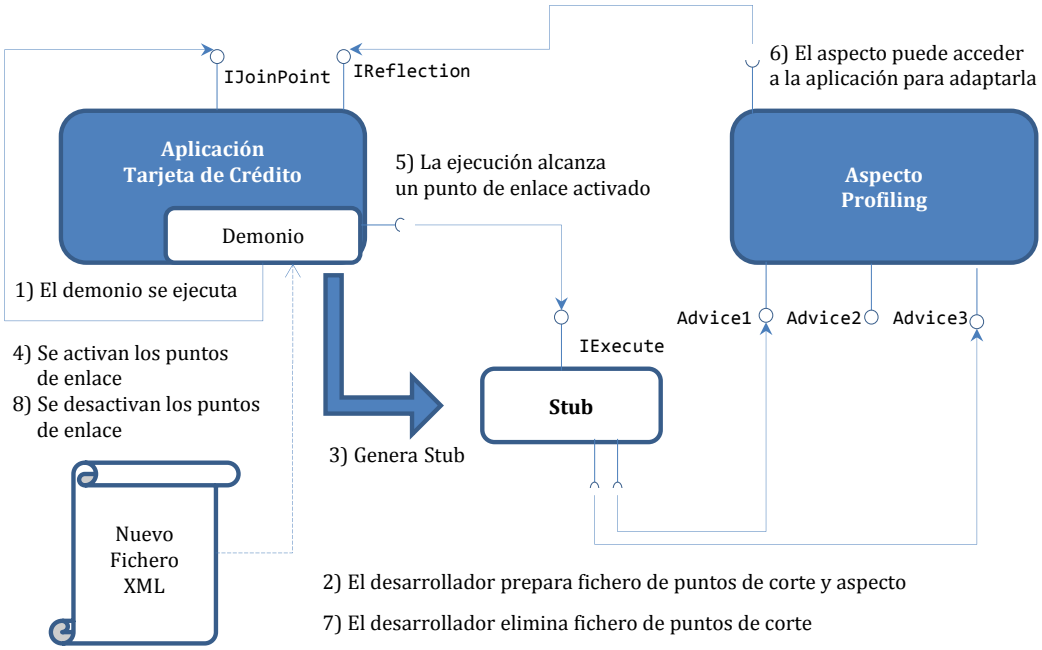


Figura 16: Adaptación dinámica local de una aplicación en tiempo de ejecución.

También podría darse la necesidad de modificar los puntos de corte usados por un aspecto en tiempo de ejecución. Esta operación también es ofrecida por DSAW, tanto para el caso local como para el remoto. En este caso, el desarrollador debería preparar un nuevo fichero XML especificando los nuevos puntos de corte. El hilo demonio detectará el nuevo fichero y lo analizará, activando los nuevos puntos de enlace y desactivando los existentes en tiempo de ejecución.

El hilo demonio en el caso local (y el AS en el caso remoto) actúa como un *mediador* entre los aspectos y las aplicaciones [Gamma94]. Esta mediación es sólo realizada cuando los puntos de enlace son tejidos o destejidos. Una vez estas operaciones han sido realizadas, la aplicación y el aspecto interaccionan directamente a través del *stub* generado. La aplicación llama al aspecto invocando al método `Execute` del interface `IExecute` utilizando el *stub* cuando un punto de enlace activado es alcanzado. Entonces el aspecto puede inspeccionar la aplicación mediante la información recibida, o utilizando `IReflection` si necesita una funcionalidad más versátil.

Con este diseño, las aplicaciones no necesitan conocer los aspectos que van a ser tejidos en tiempo de ejecución. Al mismo tiempo, los aspectos pueden ser aplicados a cualquier aplicación, o incluso cualquier aspecto, sin ninguna dependencia estática. Este comportamiento reduce el acoplamiento y promueve la reutilización tanto de componentes como de aspectos.

6.5.2 Remoto

DSAW también permite el tejido y destejido de aspectos de forma distribuida mediante el tejido remoto. En ese caso, la ejecución del sistema es controlada por el Servidor de Aplicaciones (*Application Server*, AS). El AS coordina componentes y aspectos, proporcionando la adaptación dinámica de los programas en tiempo de ejecución. El AS actúa como un sistema de registro de las aplicaciones que se encuentran en ejecución, ofreciendo a los aspectos la lista de las aplicaciones activas para facilitar su adaptación dinámica.

Siguiendo con el ejemplo del pago de una tarjeta de crédito, la siguiente secuencia describe los pasos de cómo la aplicación, el aspecto de profiling y el AS trabajan juntos en tiempo de ejecución para llevar a cabo un tejido y destejido dinámico y remoto (ilustrado en Figura 17):

1. La aplicación es ejecutada tras ser procesada por el JPI. En el arranque ésta se registra en el AS con un identificador único global (GUID), siguiendo la especificación OSF/DCE [Miller92]. Tanto el GUID como el código de registro fue previamente añadido a la aplicación por el JPI. Este GUID es utilizado a lo largo de la ejecución de la aplicación para que los aspectos puedan identificarla en el sistema.

2. Cuando se desee adaptar la aplicación, una aplicación lanzadora (*launcher*), que puede ser el propio aspecto, es ejecutada. El objetivo del *launcher* es adaptar dinámicamente la aplicación (es una adaptación programática realizada por un programa, en lugar de por un humano).

3. DSAW, a petición del *launcher*, lee el fichero XML con la especificación de los puntos de corte (Figura 14). Utilizando CodeDOM [Microsoft13] se genera en tiempo de ejecución un programa *stub* que hace llamadas a las funciones del aspecto (*advices*), y almacena la relación entre el nombre del *advice* y su llamada a través del *stub*.

4. DSAW llama al AS (mediante `IServer`) pasándole el fichero XML, la estructura de datos generada en el punto 3, y el GUID de la aplicación.

5. El AS procesa el documento XML buscando los puntos de corte pasados recibidos. Los puntos de enlace descritos por estos puntos de corte son activados en la aplicación por medio del MOP inyectado por el JPI (`IJoinPoint`). En el punto de enlace se almacenan las llamadas a los advices a través del *stub* generado. Esta activación implicará la eventual invocación del aspecto en tiempo de ejecución. Esta acción es la operación `weave` i, ϕ, t formalizada en la Sección 4.3.

6. Cuando la ejecución de la aplicación alcanza un punto de enlace tejido (por ejemplo, la llamada al método `payment`), se invocará al aspecto de *profiling* a través de la llamada al método `Execute` del interface `ICaller`. Entonces, la aplicación pasa el control al aspecto enviando información del punto de enlace (*StaticPart* en nuestro ejemplo).

7. Cuando ya no es necesario que el aspecto se ejecute, el *launcher* envía al AS una orden solicitando la desactivación de los puntos de enlace de la aplicación interceptados por el aspecto.

8. El AS desactiva los puntos de enlace de la aplicación utilizando `IJoinPoint` (instrucción `unweave` i, ϕ, t de la Sección 4.3).

9. Finalmente, cuando la aplicación finaliza su ejecución, el código añadido por el JPI notificará al AS que la aplicación ha terminado y debe ser desregistrada.

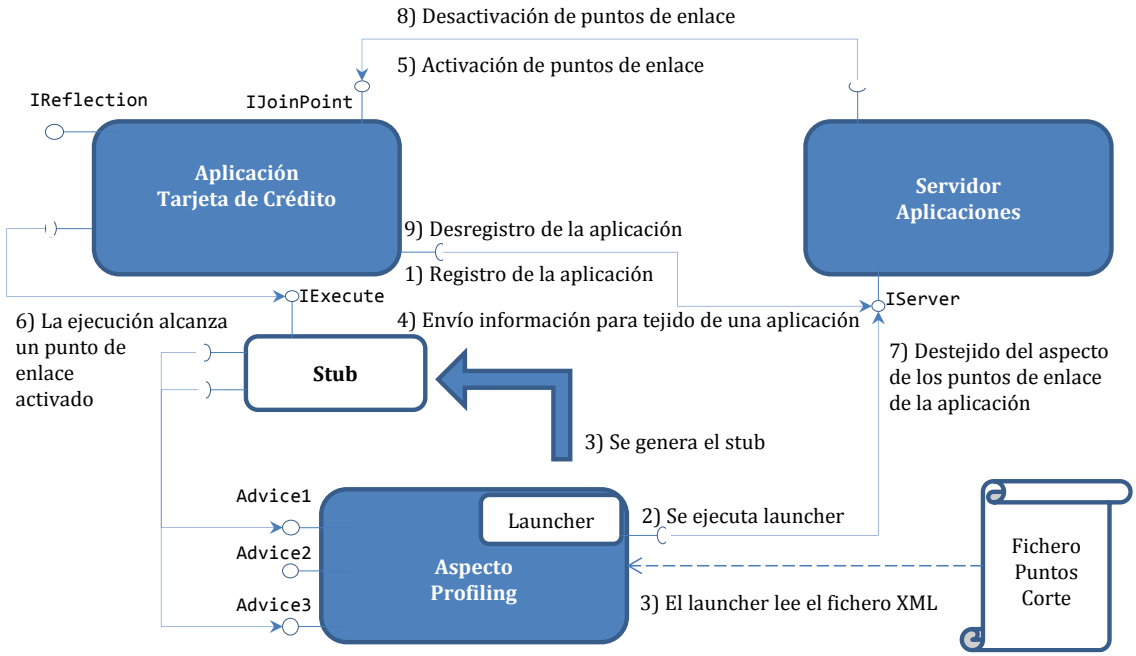


Figura 17: Adaptación dinámica remota de una aplicación en tiempo de ejecución.

Para intercomunicar el AS, las aplicaciones y los aspectos hemos usado *.Net Remoting* [Microsoft13d] (ahora parte del *framework Windows Communication Foundation*). *.Net Remoting* es un servicio estándar sobre la plataforma .NET proporcionando canales (protocolos) independientes del lenguaje y plataforma, permitiendo la adaptación de aplicaciones sobre entornos distribuidos. Puesto que toda la información es pasada por valor, su rendimiento es significativamente menor que la opción de tejido dinámico local.

6.6 JPI como Tejedor Estático

Aunque el tejido dinámico facilita el desarrollo de software interactivo siguiendo la aproximación DSOA, también implica comúnmente un mayor coste de rendimiento en tiempo de ejecución [Garcia12]. Al mismo tiempo, aunque hay escenarios donde la adaptación dinámica usando aspectos es apropiada, muchos otros escenarios no requieren ese nivel de dinamismo. Esta es la razón por la que hemos diseñado DSAW para soportar los dos tipos de tejido simultáneamente, proporcionando de esta forma los beneficios de ambas aproximaciones.

Hemos aplicado el principio de *Separation of Concerns* (SoC) a DSAW. Particularmente, hemos separado el *concern* dinamismo (tiempo de tejido) para facilitar el uso de DSAW en múltiples escenarios. Este proceso ha sido realizado de forma transparente, reduciendo el impacto de cambiar el *con-*

cern dinamismo en el código fuente de la aplicación. El programador puede usar el tejido dinámico para desarrollo interactivo y, una vez la aplicación ha sido probada, usar el tejido estático para obtener un mejor rendimiento. Los aspectos pueden ser tejidos estática o dinámicamente indistintamente, sin ningún proceso de conversión. Además, una misma aplicación puede tener aspectos tejidos estática y dinámicamente de forma simultánea.

Hemos visto cómo el JPI instrumenta el *byte-code* para conseguir la adaptación dinámica de una aplicación, usando un documento XML que describe los puntos de corte a activar. Tal y como se muestra en la Figura 18, a estos documentos XML se les pueden añadir *advices* para así realizar el tejido estático. Para ello, se etiquetaran los *advices* de tipo `StaticInjection`.

```
01: <advice_definition idAdvice="AspectProfiling"
         idTypeOfInjection="StaticInjection">
02:    <assembly>AspectPayment.dll</assembly>
03:    <type>Aspects.ProfilerAspect</type>
04:    <behaviour>loggingStaticPart</behaviour>
05:    <priority>1</priority>
06:    <pointcut_definitionRef idRef="dynamicMethodCall"/>
07: </advice_definition>
```

Figura 18: Localización del advice para inyección estática.

Estos ficheros XML se pasarán como argumentos al JPI. El JPI, además de realizar la instrumentación del tejido dinámico, realizará el tejido estático entre componentes y aspectos. De esta forma, tal y como se muestra en la Figura 19, el JPI no sólo instrumenta aplicaciones para ser adaptadas en tiempo de ejecución, sino que también las puede tejer estáticamente con aspectos.

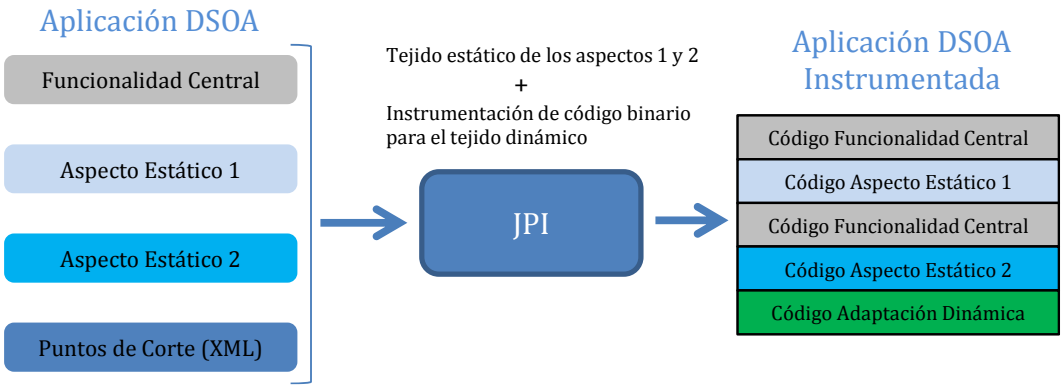


Figura 19: Tejido estático realizado por el JPI.

Siguiendo con nuestro ejemplo, podemos reutilizar cualquier aspecto existente de *logging* tomado de otro programa orientado a aspectos. Durante la instrumentación podemos tejer este aspecto estáticamente y, en tiempo de

ejecución, adaptar la aplicación dinámicamente mediante el aspecto de *profiling*. El código fuente de la Figura 20 muestra el *concern* de *logging* de nuestro ejemplo. Es importante resaltar que la signatura del *advice* es la del caso *Full*, ya que se accede tanto a información estática (el momento de tejido), como dinámica (los valores de los argumentos).

```
01: namespace Payment {
02:    public class LoggerAspect {
03:        static public object logPaymet(string ns, string cl,
04:                  string member, TypeOfMembers type, JPoint jp,
05:                  Time time, object ResultVal, Object[] Params,
06:                  object OBJECT_THIS) {
07:          ILog logger = IogManager.GetLogger(
08:                   MethodBase.GetCurrentMethod().DeclaringType);
09:          if (time = Time.Before) {
10:            logger.Info("Entering the " + member " method");
11:            StringBuilder sb = new StringBuilder("Arguments: {");
12:            for (int i = 0; i < Params.Lenght; i++) {
13:              sb.Append("Value = " + Params[i]);
14:              sb.Append(i < Params.Length - 1 ? ", " , ": ");
15:            }
16:          logger.Debug(sb.ToString());
17:          }
18:          if (time = Time.After) {
19:            logger.Debug("The payment has been " +
20:                     ((bool)ResultVal? "succesful":"erroneous"));
21:            logger.Info("Exiting the " + member + " method");
22:          }
23:         return ResultVal;
24:    }
25:  }
26: }
```

Figura 20: Implementación en C# de un aspecto de *logging* en la plataforma DSAW.

6.7 Puntos de Corte y Descripción de Advices para el Tejido Estático

Como hemos visto, el JPI puede realizar tejido estático recibiendo un documento XML como argumento desde la línea de comandos. Este documento debe describir los puntos de corte, pero también es necesario que describa los *advices*. En DSAW, un *advice* indica qué métodos en un aspecto deben ser llamados cuando los puntos de enlace (descritos por los puntos de corte del fichero XML) son alcanzados. Esto es lo que hemos formalizado con la función ψ en la Sección 4.2.

La Figura 21 muestra el documento XML usado para tejer estáticamente el aspecto de *logging* de nuestro ejemplo con el resto de la aplicación. Los puntos de corte son definidos primero, del mismo modo que en la Sección 6.4, y a continuación viene la especificación del *advice* (`advi-`

`ce_definition`). Después del nombre del aspecto (`LoggerAspect`), se indica que el *advice* se va a ejecutar estáticamente (mediante el atributo `idTypeOfInjection`). Con el *tag* `assembly` se identifica el nombre del fichero binario (`Aspect.logger.dll`) y su localización. Por último, indicamos el nombre de la clase incluyendo su espacio de nombres (`Payment.LoggerAspect`) y el método que se debe invocar en la intercepción del punto de enlace (`logPayment`). Este método será el que finalmente se ejecute desde el punto de enlace tras llamar al interface `IExecute.Execute`. La prioridad (explicada en la siguiente sección) es opcional y, por último, es posible usar una referencia (`idRef`) a la descripción de su correspondiente punto de corte.

En el ejemplo de la Figura 21, en los puntos de enlace `methodcall` y momentos de ejecución `before` y `after` del método `payment`, se invocará al método `logPayment` de la clase `Payment.LoggerAspect`.

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <aspect_definitions xmlns="urn:gramaticapointcuts-schema" … >
03:   <pointcut_definition id="StaticMethodCall">
04:     <aspectized_type>StaticPart</aspectized_type>
05:     <time>before</time><time>after</time>
06:     <joinpoint_type>
07:        <methodcall>
08:            <method_signature>
09:                <return_type> <type_name>*
                                 </type_name> </return_type>
10:                <qualified_method_name>
11:                  <qualified_class>
12:                    <namespace> <type_name>*
                                   </type_name> </namespace>
13:                    <class>
14:                      <identifier_name>*</identifier_name>
15:                    </class>
16:                  </qualified_class>
17:                  <name>
18:                    <identifier_name>payment</identifier_name>
19:                  </name>
20:                </qualified_method_name>
21:            </method_signature>
22:         </methodcall>
23:       </joinpoint_type>
24:   </pointcut_definition>
25:   <advice_definition idAdvice="LoggerAspect"
                         idTypeOfInjection="StaticInjection">
26:     <assembly>Aspect.logger.dll</assembly>
27:     <type>Payment.LoggerAspect</type>
28:     <behaviour>logPayment</behaviour>
29:     <priority>1</priority>
30:     <pointcut_definitionRef idRef="StaticMethodCall"/>
31:  </advice_definition>
32: </aspect_definitions>
```

Figura 21: Descripción XML de un punto de corte y un *advice* para tejido estático.

Como hemos mencionado en la Sección 6.4, el tejido dinámico de aspectos usa también documentos XML para especificar los *advices*. Para los dos tipos de tejidos el aspecto es el mismo, por lo que no se debe hacer ningún cambio para modificar la forma de tejido de un aspecto. Esto hace posible la transición de dinámico a estático, y viceversa, sin modificar la implementación de los aspectos.

El JPI usando el atributo `idTypeOfInjection` distingue si un *advice* se va a usar para tejido estático, para tejido dinámico o para ambos. Para el caso de tejido dinámico los *advices* también pueden, opcionalmente, tener el atributo `idTypeOfDynamicInjection`. Este atributo permite indicar si el aspecto se va a tejer de forma local o remota.

6.8 Resolución de Conflictos

En DSAW, es posible crear aplicaciones con aspectos que hayan sido tejidos estáticamente junto con aspectos que hayan sido añadidos más tarde en tiempo de ejecución. Por lo tanto, es necesario definir un mecanismo de resolución de conflictos [Nagy05]. Un conflicto entre aspectos se produce cuando dos aspectos diferentes son tejidos en el mismo punto de enlace. Un algoritmo de resolución de conflictos debería hacer posible especificar una estrategia flexible para determinar el orden de invocación de aquellos aspectos que hayan sido tejidos en el mismo punto de enlace.

En la Sección 4.2.2 formalizamos la estrategia de resolución de conflictos como una función α que podía ser implementada de distintas formas, teniendo en cuenta la información del entorno dinámico. Nuestra implementación actual de la función α para la resolución de conflictos, aunque sencilla, tiene en cuenta cuatro variables de cada aspecto: su dinamismo, su tipo de *advice*, su prioridad y cuándo fue tejido.

1. Teniendo en cuenta su dinamismo, DSAW da prioridad a los aspectos dinámicos. Puesto que la necesidad de tejer un aspecto dinámicamente puede ser causada por un problema o un nuevo requisito surgido durante la ejecución de la aplicación, requiriendo que se resuelva inmediatamente, DSAW da prioridad a los aspectos tejidos de esta forma.

2. Considerando el tipo de *advice*, el código de un aspecto es ejecutado siguiendo el orden *before*, *around* y *after*. Como mencionamos en la Sección 4.2.2, si el programador quiere que los aspectos de tipo *around* adapten el resto de los aspectos tejidos en el mismo punto de enlace (siguiendo la semántica de AspectJ), el aspecto *around* debería tejer no sólo a la aplicación, sino también los aspectos existentes.

3. Para cada aspecto, el programador puede especificar su prioridad con un número entre 1 y 100. Cuanto mayor es este número, más pronto será ejecutado el aspecto. Este mecanismo es bastante similar a la construcción `declare precedence` de AspectJ. Sin embargo, DSAW establece la precedencia entre puntos de corte, mientras que AspectJ usa aspectos. Esto permite que DSAW pueda resolver más de un conflicto con el mismo aspecto.

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

Aunque nuestra implementación actual de la resolución de conflictos es simple, hemos separado el mecanismo de resolución de conflictos de la implementación del tejedor de aspectos para facilitar la incorporación de nuevas estrategias de resolución de conflictos en el futuro (nuevas funciones α). La primera aproximación podría ser extendiendo la especificación XML de descripción de un *advice*, permitiendo al programador especificar un método resolutor de conflictos que sería invocado dinámicamente por DSAW si se diese un conflicto. Esta aproximación es similar a los *connectors* de JAsCo [Suvee03], pero en nuestro caso sería independiente del lenguaje. Los siguientes pasos podrían incluir soluciones más avanzadas como *stateful aspects* [Dounce04] para mejorar el dinamismo de la resolución de conflictos (teniendo en cuenta la historia de la computación), o incluso la detección y corrección de conflictos semánticos [Durr05].

7 OPTIMIZACIONES

Para la realización de la plataforma DSAW se ha partido de Ready AOP [Vinuesa07], un tejedor totalmente dinámico independiente del lenguaje y de la plataforma, basado en un servidor de aplicaciones. Ready AOP inyecta código para interceptar los puntos de enlace en una etapa previa a la ejecución de la aplicación. En tiempo de ejecución, un servidor de aplicaciones intercambia mensajes entre la aplicación tejida y los aspectos inyectados. La gran desventaja de Ready AOP es su lentitud, ya que este intercambio de mensajes se hace utilizando la librería *.Net Remoting*, provocando una gran penalización en el rendimiento de la aplicación.

Uno de los objetivos que se ha buscado en la realización de esta tesis ha sido reducir las penalizaciones que sufrían las aplicaciones aspectizadas. Por ello, hemos investigado en el diseño de optimizaciones de tejido estático y dinámico. En DSAW hemos diseñado e incorporado varias optimizaciones, consiguiendo reducir las penalizaciones en el rendimiento en ambos tipos de tejido [Garcia13]. Iremos describiendo dichas optimizaciones, incluso si ellas fueron posteriormente modificadas parcialmente con otras optimizaciones.

7.1 Tejedor Estático

La primera optimización que se ha realizado ha sido la inclusión de un tejedor estático homogéneo junto con el tejedor dinámico de partida, Ready AOP. De esta forma surge DSAW [Vinuesa08], una plataforma orientada a aspectos que permite tanto tejido estático como dinámico. En la fase de instrumentación, la plataforma, además de instrumentar la aplicación para per-

mitir tejido dinámico, ahora también es capaz de inyectar las llamadas a los aspectos que se necesitan tejer estáticamente.

Como los puntos de corte y los aspectos son conocidos estáticamente en tiempo de instrumentación, en el *byte-code* de la aplicación se incluyen las librerías requeridas por DSAW. También se añaden a los puntos de enlace identificados por los puntos de corte las llamadas a los *advices* de los aspectos, todo ello sin hacer uso de reflexión. La invocación a los métodos de los aspectos (*advice*) es directa, sin utilizar un objeto intermedio que haga de delegado (*proxy*) o decorador (*decorator*). De esta forma, la penalización en el rendimiento en tiempo de ejecución para el tejido estático es mucho menor que la del tejido dinámico (§ 8.5) [Garcia12].

7.2 Tejedor Dinámico Local

La mayor optimización encontrada en materia de tejido dinámico es la realizada por JAsCo [Vanderperren04]. Esta plataforma remplaza dinámicamente las clases aspectizadas por aquéllas resultantes del proceso de tejido. Esto es posible porque hace uso de la librería Javassist para Java [Chiba00]. No obstante, .NET no ofrece la posibilidad de cambiar el *byte-code* de las clases dinámicamente. Los primeros pasos para mejorar el rendimiento fueron:

- Eliminación del Servidor de Aplicaciones. El AS activaba y desactivaba los puntos de enlace, enlazándolos con los *advices*. Este proceso ha pasado a realizarse mediante un hilo demonio (*daemon*), dentro del propio proceso de la aplicación. De esta forma, es posible eliminar el proceso del AS.

  El hilo demonio revisa de forma periódica un directorio comprobando los cambios realizados (creación, modificación o borrado de ficheros) sobre los ficheros XML de puntos de corte. Cuando se detecta algún cambio, se procesa el fichero obteniendo los puntos de corte y los *advices*. Se recorre la estructura de los puntos de enlace (una tabla hash de tablas hash de 5 niveles), y si algún punto de enlace se empareja con el punto de corte, se crea un objeto de tipo `ExecuteLibrary`, activando así el punto de enlace.

- Utilización del interface `IExecute`. Este tipo define una interfaz con las posibles llamadas que se pueden hacer desde un punto de enlace al aspecto. El tipo de llamada depende del tipo de punto en-

lace. En la Figura 22 vemos que pueden ser llamadas a métodos o constructores (`Execute`), accesos en modo escritura a un campo o propiedad (`AccessFieldSet`) y acceso en modo lectura a un campo o propiedad (`AccessFieldReference`). Cada uno de estos métodos recibe un número elevado de parámetros, incluyendo el nombre del espacio de nombres, el nombre de la clase interceptada y el nombre del método a ejecutar (*advice*).

```
01: Object Execute(string typeAdvice, string behaviour,
02:                string ns, string cl, string member,
03:                TypeOfMembers type, JPoint jp, Time time,
04:                object ResultVal, Object[] Params,
05:                object OBJECT_THIS , IReflection ir);
06:
07: Object AccessFieldSet(string typeAdvice, string behaviour,
08:                        string ns, string cl, string member,
09:                        TypeOfMembers type, JPoint jp, Time time,
10:                        object ResultVal, object OBJECT_THIS);
11:
12: void  AccessFieldReference(string typeAdvice, string behaviour,
13:                        string ns, string cl, string member,
14:                        TypeOfMembers type, JPoint jp, Time time,
15:                        object ResultVal, object OBJECT_THIS);
```

Figura 22: Interface `IExecute`.

- Utilización de la clase `ExecuteLibrary`. Esta clase implementa el interface `IExecute`. Es la encargada de recibir las peticiones desde la aplicación y hacer las correspondientes llamadas a los aspectos. Para ello, cuando el hilo demonio teje un aspecto en tiempo de ejecución, se carga el fichero con el *byte-code* de los *advices* en memoria. Esa operación se realiza en la inicialización de la clase. Cuando la aplicación llega a un punto de enlace activo donde el aspecto esté tejido, se llama al aspecto a través del interface `IExecute`. Esta invocación conlleva una llamada a la clase `ExecuteLibrary` que implementa el interface, llamando reflectivamente al método tal y como se muestra en la Figura 23. Inicialmente se cargan los parámetros de la llamada y posteriormente, en la línea 17, se hace la llamada reflectiva.

  Esta clase es muy general y permite enlazar cualquier punto de enlace con cualquier aspecto. No obstante, el uso de reflexión y la generalización de cualquier valor con el tipo `object` provocan un coste de rendimiento. Este coste fue subsanado con subsiguientes optimizaciones presentadas en este capítulo.

```
01: public Object Execute(string typeAdvice, string behaviour, ... ) {
02:     Object[] parameters = new object[10];
03:     parameters[0] = ns;
04:     parameters[1] = cl;
05:     parameters[2] = member;
        ...
16:     typeObject = assembly.GetType(typeAdvice);
17:     Object obj = typeObject.InvokeMember(behaviour,
18:                  BindingFlags.Default | BindingFlags.InvokeMethod,
19:                  null, null, parameters);
20:     return obj;
21: }
```

Figura 23: Llamada reflectiva al *advice*.

- Cambios en la instrumentación de la aplicación. En la inicialización de la clase se inyecta la llamada para crear y arrancar el hilo demonio. En los puntos de enlace identificados por los puntos de corte, además de inyectar un MOP para comprobar si hay algún aspecto inyectado, se hace una llamada a un método de la interface `IExecute` apropiado para cada punto de enlace.

7.3 Tejedor Dinámico Remoto

Las optimizaciones de `IExecute` y `ExecuteLibrary` han sido también incorporadas al tejido dinámico remoto para que el Servidor de Aplicaciones funcione de forma similar al *daemon*. De esta forma, no existe diferencia entre los aspectos tejidos estática o dinámicamente, y de forma local o remota. No es necesario ya implementar los *advices* adaptándolos a las interfaces `IMethodCall`, `IPropertyFieldAccess` o `IStaticInitializer`.

7.4 Especialización de la Información Transferida

Otra optimización incorporada es la de especializar la información enviada a un aspecto, tal y como lo realiza AspectJ [Kiczales01]. En AspectJ cuando se llama a un aspecto no siempre se le pasa toda la información disponible en tiempo de ejecución. Este enfoque es altamente práctico, ya que en muchos casos la información que necesitan los aspectos para realizar sus tareas es bastante menor que toda la información disponible en tiempo de ejecución. Por este motivo, afinando la información realmente necesaria se consigue una mejora notable de rendimiento [Kiczales01].

Se crearon cuatro clasificaciones en la especialización de información a transferir a los aspectos: *WithoutReflection* no pasa ningún parámetro al aspecto; *StaticPart* pasa información estática al aspecto (nombre del método interceptado, nombre de la clase, nombre del espacio de nombres y signatura del método); *DynamicPart* pasa información dinámica (referencia al objeto que ejecuto el método interceptado, parámetros del método y valor de retorno); y *Full*, con la que se pasa tanto la información estática como la dinámica.

Debido a esta especialización, tuvimos que modificar la interface `IExecute`. Tal y como se aprecia en la Figura 24, el método `Execute` encargado de interceptar el punto de enlace *MethodCall* fue sobrecargado con cuatro métodos, dependiendo la cantidad de información que se quiere pasar al *advice*.

```
01: void Execute(string typeAdvice, string behaviour); // No Reflection
02: void Execute(string typeAdvice, string behaviour,  //Static Part
03:           ReflectiveStaticInformation reflectiveStaticInformation);
04: Object Execute(string typeAdvice,  string behaviour, //Dynamic Part
05:                string member, object ResultVal, Object[] Params,
06:                object OBJECT_THIS);
07: Object Execute(string typeAdvice, string behaviour,  //Full
08:              string ns, string cl, string member,
09:              TypeOfMembers type, JPoint jp, Time time,
10:              object ResultVal, Object[] Params, object OBJECT_THIS,
11:              IReflection ir);
```

Figura 24: Método `Execute` especializado por tipo de información recibida.

Para pasar la información a los aspectos en el caso *StaticPart*, en lugar de pasar varios parámetros, creamos una clase `ReflectiveStaticInformation` donde se almacena el momento de tejido, el tipo del miembro, el tipo del punto de enlace, el nombre de la clase y la signatura del método (línea 3 de la Figura 24). Otra optimización realizada consiste en preinstanciar los objetos de esta clase, puesto que almacenan información estática. Mientras el JPI va instrumentando la aplicación, cada vez que un punto de enlace se empareja con un punto de corte, la información de este punto de enlace se almacena en un vector de objetos instancia de `ReflectiveStaticInformation`. El JPI también instrumenta la aplicación añadiendo este vector a la aplicación, cargándolo junto con los datos de los puntos de enlace instrumentados en el arranque de aplicación. De esta forma, si un punto de enlace instrumentado sólo necesita la información estática se le pasa el objeto apropiado de ese tipo obtenido del vector cargado en memoria en el inicio de la ejecución de la aplicación.

Los cambios descritos en este subapartado supusieron una mejora importante en el rendimiento de la plataforma, especialmente para los casos *WithoutReflection* y *StaticPart*.

7.5 Generación Dinámica de Stubs

A pesar de la mejora de rendimiento descritas, el rendimiento del tejido dinámico en DSAW no era competitivo en comparación con otras herramientas como JAsCo (recordemos que en Java sí se puede cambiar el *bytecode* de una clase gracias a los Java *agents*). Para mejorar aún más los tiempos de ejecución, nos basamos en técnicas de optimización basadas en la generación dinámica de código [Ortin14].

Utilizamos la librería CodeDOM [Microsoft13] para generar dinámicamente los *stubs* personalizados para cada uno de los *advices* de un aspecto. Creamos un *assembly* temporal *on-the-fly* que se ocupase de hacer las llamadas a las los métodos de los aspectos, evitando las llamadas reflectivas realizadas en el tejido dinámico (Figura 23). El resultado es como si hiciésemos un *dispatcher* que conectase cada tipo de punto de enlace con cada tipo de *advice*, salvo que, en lugar de implementarlo nosotros, es DSAW el que lo genera dinámicamente.

De esta forma, remplazamos la clase `ExecuteLibrary` que anteriormente se comportaba como un *dispatcher* genérico utilizando reflexión para todos los puntos de enlace y todos los *advices*. Para proporcionar los servicios de *dispatching*, `ExecuteLibrary` consultaba la información del punto de enlace y, en tiempo de ejecución, utilizaba reflexión para invocar a los aspectos (Figura 23). En su lugar, ahora pasamos a generar dinámicamente el código C# que enlaza un punto de enlace y un *advice*. Usando CodeDOM compilamos el código C# generado y lo cargamos en memoria.

El gran beneficio de la generación dinámica de código es que, como en el momento en el que el hilo demonio detecta el fichero XML se conoce toda la información de los aspectos, no es necesaria la utilización de reflexión para hacer las llamadas a los *advices*. Este cambio supuso una mejora importante en el rendimiento de las aplicaciones que eran tejidas dinámicamente.

7.6 Intercepción de los Puntos de Enlace

Para la intercepción de los puntos de enlace activos, el JPI inyecta código que detecta si un punto de enlace está activo o no. En el caso de estarlo, invoca a la lista de *advices* interesados en interceptar ese punto de ejecución. Para identificar un punto de enlace, utilizábamos una tabla hash de 5 niveles, que, a partir del espacio de nombres, clase, método, momento de ejecución y tipo de información transferida, obtenía un punto de enlace.

El calcular los códigos *hash* de toda la información utilizada para identificar un punto de enlace tiene un coste computacional. Si bien no es muy alto, éste se ejecuta muchas veces en la ejecución de una aplicación. Por tanto, eliminamos la estructura por un índice de punto de enlace. Al inyectar el código, el JPI crea una nueva estructura de datos que, a partir de la información de un punto de enlace, le asigna un índice consecutivo. El código inyectado sólo tendrá que utilizar este índice (un número entero) en un vector, para conocer los *advices* registrados en ese punto de enlace.

Para implementar este sistema de indexación, reaprovechamos lo realizado al especializar la información. Al hacer los cambios para *StaticPart* se implementó una tabla con información de todos los puntos de enlace instrumentados (vector de elementos de tipo `ReflectiveStaticInformation`). Esta tabla contiene toda la información existente en la estructura de tablas hash. Esta tabla va a guardar una correspondencia entre el punto de enlace almacenado en el índice *i* y los aspectos inyectados dinámicamente en ese punto de enlace. Éstos se obtendrán en un `ArrayList`.

7.7 Indexación de los Advices

Otra optimización introducida fue la forma de ejecutar los métodos (*advices*) en el *dispatcher* generado dinámicamente. En lugar de pasar como parámetros al *dispatcher* el nombre del método, la clase y el espacio de nombres del *advice*, ahora pasamos únicamente un entero que actúa como índice de un vector de funciones. Esta técnica es similar a la anterior, pero optimiza el *stub* creado dinámicamente en lugar del código inyectado por el JPI.

Por un lado, esta optimización reduce la información a enviar entre la aplicación y los aspectos El número de parámetros que recibe el *dispatcher* influye en el rendimiento global de la aplicación. A mayor número de parámetros, más operaciones se deben realizar para almacenar los datos en la pila, requiriendo más tiempo de ejecución. Cuando en un punto de enlace se

inyecta una llamada al método `Execute` para ejecutar el aspecto dinámicamente, estamos además inyectando código para almacenar en la pila el espacio de nombres, la clase y el nombre del *advice*. Estos tres parámetros de tipo `string` los podemos codificar con un índice, pasando sólo un número entero.

La segunda mejora de esta optimización se refleja en la parte del *dispatcher*. Para seleccionar el método que se debe ejecutar, antes se tenían que realizar comprobaciones sobre cadenas de caracteres para seleccionar el método adecuado. Ahora sólo se debe acceder a la posición del vector indicado por el código.

Para realizar esta optimización hemos modificado otra vez la clase `ExecuteLibrary`, utilizando delegados en C# (métodos como entidades de primer orden). Para cada tipo posible de llamada a un *advice* en un punto de enlace (`methodWithReflection`, `methodStaticPart`, `methodDynamicPart`, `methodFull`, `fieldAccessSetDynamicPart`, etc.), la clase `IExecuteLibrary` crea un tipo de vector en el *dispatcher*. En la Figura 25 vemos como es la declaración de estos vectores para el caso de métodos.

```
01: class Class1:  Caller.IExecute {
02:     static Action[] methodWithoutReflection = { … };
03:     static Action<Interfaces.ReflectiveStaticInformation>[]
                        methodStaticPart = { };
04:     static Func<string, object, object[], object, object>[]
                        methodDynamicPart = { };
05:     static Func<string, string, string, TypeOfMembers, JPoint, Time,
06:               object, object[], object, object>[] methodFull = { };
07:     …
08: }
```

Figura 25: Vectores de delegados para el caso de un método.

La clase `ExecuteLibrary` es la encargada de, en tiempo de ejecución, inicializar estos vectores con los valores resultantes de tejer los puntos de enlace con los *advices* (la información incluida en los ficheros XML). De esta forma conseguimos codificar la identificación de un *advice* mediante un código entero, la posición que ocupa en el vector de delegados.

En la Figura 26 vemos en la línea 1 cómo es la declaración del vector para un punto de enlace *Method Call,* pasándole la información del caso *DynamicPart*. Ese vector se rellena con un delegado (línea 2) que encapsula la llamada al *advice*. En este caso, el aspecto es el método a ejecutar en la operación de *logging* del ejemplo del cajero.

```
01: static Func<string, object, object[], object, object>[]
       methodDynamicPart = {
02:            new Func<string, object, object[], object, object>(
03:                         Payment.LoggerAspect.logPayment)
04             ....
05:};
```

Figura 26: Ejemplo de vector de delegados del `methodDynamicPart`.

En el proceso de generación dinámica del *stub*, el índice de cada delegado es utilizado para ser almacenado con la información de los *advices* tejidos en los puntos de enlace. Así, al llegar a un punto de enlace con un aspecto tejido dinámicamente, se recupera la información de esta posición y se le pasa al método `Execute` para que obtenga el delegado apropiado.

Una vez creadas las estructuras de datos anteriores, el código generado con CodeDOM que realizar la llamada a los *advices* se reduce significativamente (Figura 27). Los métodos `Execute` reciben como parámetro el índice entero para identificar el *advice* a invocar (línea 1). Este código es usado para acceder el delegado apropiado, del vector de delegados de ese tipo, y ejecutarlo (línea 6).

```
01: public object Execute(Int32 index,
02:                       string member,
03:                       object ResultVal,
04:                       object[] Params,
05:                       object OBJECT_THIS){
06:    return methodDynamicPart[index](member, ResultVal, Params,OBJECT_THIS);
07: };
```

Figura 27: Llamada al *advice* usando un *delegate*.

7.8 Eliminación de *Dispatchers*

La anterior optimización incluía una indexación de los *advices* a ejecutar, haciendo que la implementación del *dispatcher* generado ofreciese una mejora importante de rendimiento. No obstante, la implementación de *dispatchers* no es estrictamente necesaria, puesto que en tiempo de ejecución conocemos exactamente el componente y aspecto a tejer.

Tal y como se muestra en la Figura 27, los *dispatchers* generados realizan una única implementación de cada tipo de punto de enlace. El método `Execute` mostrado en la figura es llamado en la intercepción de cualquier invocación a método. El índice pasado indica el *advice* que se debe ejecutar (su índice).

Puesto que en el momento del tejido se conoce tanto el componente como el aspecto a tejer, no es necesario implementar un *dispatcher*. Por cada

punto de enlace interceptado, se puede crear una implementación de `Execute` distinta. Se creará una nueva clase que implemente `IExecute` para cada punto de enlace. El código mostrado en la Figura 28 muestra la generación de código para el ejemplo de la sección anterior. El código generado se limita a llamar al método apropiado del aspecto tejido, para ese punto de enlace. Se crearán tantas clases derivadas de `IExecute` como métodos de enlace interceptados.

```
01: public class Class1: Caller.IExecute {
02:     public object Execute(string member,
03:                           object ResultVal,
04:                           object[] Params,
05:                           object OBJECT_THIS){
06:       return Payment.LoggerAspect.logPayment(member, ResultVal,
07:                                              Params,OBJECT_THIS);
08:     }
09: };
```

Figura 28: Generación específica para cada punto de enlace interceptado.

En el momento de realizar el tejido dinámicamente, se creará un objeto de una clase distinta (derivada de `IExecute`), almacenando este objeto como interceptor del punto de enlace activado. Esta optimización remplaza la necesidad de tener un vector de *advices* en el código generado y el paso del parámetro entero para acceder al *advice* adecuado. En contraposición, se basa en el mecanismo de enlace dinámico proporcionado por la máquina abstracta: el componente llama a `Execute` de `IExecute` y la máquina virtual encuentra la implementación adecuada para ese punto de enlace. La consecuente eliminación de los *dispatchers* generados ha supuesto una mejora de rendimiento sustancial.

8 EVALUACIÓN

En este capítulo realizaremos una evaluación empírica del rendimiento y consumo de memoria en tiempo de ejecución, comparando varias plataformas de desarrollo software orientado a aspectos con DSAW. Primero describiremos la metodología utilizada, las plataformas comparadas, los *benchmarks* utilizados y el entorno de evaluación. Seguidamente, para cada tipo de test, presentaremos y discutiremos los datos de rendimiento en tiempo de ejecución y el consumo de memoria. El conjunto de datos completo obtenido en todas las evaluaciones puede ser consultado en los Apéndices (Apéndice B y Apéndice C).

8.1 Metodología

Para evaluar la eficiencia de todas las plataformas hemos seleccionado un conjunto de *benchmarks*. Hemos clasificado estos *benchmarks* en tres grupos, dependiendo de las características de los programas medidos (§ 8.1.1). Para realizar la evaluación, hemos seleccionado un amplio grupo de plataformas orientadas a aspectos (§ 8.1.2).

En lo relacionado con el análisis de los datos, hemos seguido la metodología de evaluación *estadísticamente rigurosa* propuesta por Georges *et al.* [Georges07] (§ 8.1.3). Esta metodología fue creada para comparar el rendimiento de aplicaciones en tiempo de ejecución, en especial aquellas que se ejecutan sobre máquinas virtuales que proporcionan compilación JIT. Finalmente describimos el entorno donde hemos realizado la evaluación (§ 8.1.4).

8.1.1 Descripción de los Experimentos

Para evaluar la eficiencia de las plataformas seleccionadas hemos realizado varios tests de rendimiento. Éstos están divididos en cuatro grupos diferentes, dependiendo de las características evaluadas:

- **Micro-benchmark**. Éste es un *benchmark* sintético que hemos creado para medir el coste de cada una de las primitivas de las plataformas orientadas a aspectos. Para ello, las operaciones que realizan los aspectos tienen un coste computacional muy bajo para evitar "oscurecer" la medición de las primitivas, pero no trivial, para evitar que el optimizador de código las elimine. Estos aspectos se inyectan tanto estática como dinámicamente sobre una aplicación sintética, y se ejecutan intensivamente para medir su eficiencia en todas plataformas. Los resultados nos dan una primera idea del coste de cada primitiva, para después poder establecer discusiones en la evaluación de aplicaciones reales. Las mediciones han sido realizadas sobre todos los puntos de corte y momentos de ejecución (*after*, *before* o *around*) proporcionados por DSAW.

- **Benchmarks existentes**. Tras un periodo de búsqueda de *benchmarks* existentes para aplicaciones orientadas a objetos, sólo encontramos el *benchmark* AWBench codificado en Java [Vasseur04]. Este test introduce una mayor carga computacional en el código de los aspectos, en comparación con el micro-benchmark. Aunque AWBench se diseñó originalmente para tejedores estáticos, en este trabajo lo hemos adaptado para poder ser utilizado por los tejedores dinámicos existentes.

- **Aplicaciones reales**. Es importante la realización de mediciones sobre aplicaciones reales desarrolladas usando aspectos. De esta forma, se pueden comparar cargas reales de componentes y aspectos, en las que el desarrollador, tras diseñar el software, ha decidido el sitio más apropiado donde ubicarlos. Hemos tomado varias aplicaciones reales utilizadas en trabajos de investigación anteriores [Garcia12] [Garcia13] [Lago10]. Éstas han sido traducidas a distintas plataformas orientadas a aspectos para poder realizar una comparación entre ellas.

- **Coste del tejido**. Por último, hemos comparado las aplicaciones orientadas a aspectos con las mismas aplicaciones implementadas

sin la utilización de aspectos. De esta forma podemos hacer una estimación del coste de la utilización del paradigma orientado aspectos. Para esta cuarta evaluación no hemos creado un nuevo tipo de programas, utilizando las aplicaciones reales del punto anterior.

8.1.2 Plataformas Evaluadas

Las plataformas que deberían ser comparadas con DSAW son aquéllas que proporcionasen tanto tejido estático como dinámico. Lamentablemente, las únicas plataformas que han alcanzado un grado de madurez aceptable son LOOM.NET y JBoss AOP. No obstante, LOOM.NET emplea diferentes tipos de estrategias para los tejedores de los escenarios estáticos y dinámicos (Gripper y Rapier respectivamente) cambiando los tejedores en función del dinamismo y, por tanto, teniendo más oportunidades para realizar optimizaciones.

Puesto que DSAW ofrece un elevado nivel de dinamismo, también hemos incluido en la evaluación los dos sistemas POA que proporcionan un mayor nivel de dinamismo: PROSE y JAsCo. Aquellos sistemas que sean dependientes de la plataforma (sistemas compilados a código nativo) no los hemos considerado, puesto que la independencia de la plataforma es una característica que tiene una elevada influencia en el rendimiento dinámico [Garcia14].

Finalmente, para enriquecer la evaluación, hemos decidido seleccionar también aquellos sistemas POA más usados en la actualidad para el desarrollo de aplicaciones reales (AspectJ, Spring tanto para Java como .NET y la ya citada JBoss AOP). Estos sistemas, además de ser muy extendidos, son los que ofrecen el mayor nivel de rendimiento.

Éstos han sido los sistemas evaluados:

- **AspectJ** 1.7.1 [Kiczales01]. Ésta es muy probablemente la herramienta más utilizada actualmente en el desarrollo de software orientado a aspectos. Se integra con el lenguaje Java mediante una extensión que proporciona capacidades de orientación a aspectos de una forma fluida. Aunque ofrece algo de dinamismo mediante puntos de corte como `cflow` o `within`, el tejido de los aspectos no es realizado en tiempo de ejecución (posteriormente a cargarse la aplicación). Para la evaluación hemos usado el compilador `ajc`.

- **JAsCo** 0.8.7 [Suvee03]. Es una plataforma orientada a aspectos con tejido dinámico. Originalmente fue diseñada para el campo de los

componentes y los servicios Web. JAsCo se comporta excelentemente en la integración dinámica y eliminación de aspectos con una penalización mínima en el rendimiento dinámico. El lenguaje de JAsCo es una extensión orientada a aspectos basada en Java. Necesita la máquina virtual de Java 1.5.

- **JBoss AOP** 2.1.8.GA [Redhat12]. JBoss AOP es un *framework* 100% Java orientado a aspectos, usable en cualquier entorno de programación que disponga de esa máquina virtual. Esta plataforma está integrada en el servidor de aplicaciones JBoss. Ofrece un paquete inicial con un conjunto de aspectos que se inyectan estática o dinámicamente mediante anotaciones de puntos de corte. Se necesita una versión de Java igual o superior a la 1.5.

- **LOOM.NET** (Rapier y Gripper) 4.2 (RC1) [Schult03]. Los objetivos del proyecto LOOM.NET son investigar y promover el uso de la orientación a aspectos en el contexto del Microsoft .NET Framework. Para lograrlo, han desarrollado dos herramientas POA implementadas con dos tejedores diferentes. Por un lado, Gripper es un tejedor estático que hace permanentes los resultados del proceso de tejido. Por otro lado, Rapier permite el tejido de aplicaciones en tiempo de ejecución, aunque no es posible añadir un nuevo aspecto codificado posteriormente a la ejecución de la aplicación, o remplazar un aspecto inyectado previamente. Ambos tejedores son independientes del lenguaje, ya que operan sobre ensamblados binarios .NET.

- **PROSE** 1.4.0 [Nicoara05]. PROSE, PROgrammable extenSions of sErvices, permite la modificación de aplicaciones Java en tiempo de ejecución mediante el tejido dinámico de aspectos. PROSE es un tejedor totalmente dinámico, soportando el tejido y destejido de aspectos en tiempo de ejecución, incluso siendo éstos desconocidos en tiempo de compilación. Hemos evaluado el tejedor basado en la notificación de eventos JVMDI/JVMTI para JDK 1.5 Windows XP Release.

- **Spring Java** 3.1.2 [Spring12a]. Spring es un *framework* de desarrollo de aplicaciones JEE que soporta la programación orientada a aspectos. Spring Java funciona con versiones de Java superiores a la 1.4. Proporciona un API para añadir o eliminar aspectos en tiempo de ejecución, soportando tanto el tejido en tiempo de carga

como en tiempo de ejecución. Spring Java utiliza AspectJ para soportar tejido estático.

- **Spring .NET** 1.3.2 [Spring12b]. Spring .NET es un *framework* de desarrollo de aplicaciones .NET orientado a facilitar el desarrollo de soluciones empresariales. Su diseño está basado en la versión Java de Spring Framework. Proporciona varias librerías de aspectos con funcionalidades predefinidas para gestionar operaciones como transacciones, *logging*, monitorización del rendimiento, cacheo, reintento de lanzamiento de métodos y manejo de excepciones.

- **DSAW estático** [Vinuesa08]. Es nuestro tejedor estático desarrollado dentro de la plataforma DSAW. Esta plataforma ha sido desarrollada sobre .NET buscando la independencia del lenguaje y plataforma. Contempla un amplio conjunto de puntos de corte. En DSAW los aspectos pueden ser tejidos estática o dinámicamente sin necesidad de realizar ningún cambio.

- **DSAW dinámico** [Ortin11]. El tejedor dinámico de la plataforma DSAW. Ofrece las mismas funcionalidades que en el caso estático, pero realizando el tejido y destejido de aspectos en tiempo de ejecución. Este tejido dinámico supone una penalización de rendimiento cuando se compara con el tejido estático. Al estar integrado con el tejedor estático, facilita la conversión de aspectos estáticos en dinámicos y viceversa.

Inicialmente evaluamos todas las plataformas propuestas con la ejecución del micro-benchmark. Para los siguientes experimentos hemos realizado un filtrado de las plataformas orientadas a aspectos a evaluar, eliminando aquéllas que hayan mostrado un rendimiento significativamente peor que el resto de las plataformas, sin proporcionar una ventaja importante en el consumo de memoria.

8.1.3 Análisis de los Datos

Para el análisis de los datos hemos seguido la metodología propuesta por Georges *et al.*, *Statically Rigorous Java Performance Evaluation* [Georges07]. Esta metodología fue diseñada para medir de un modo estadísticamente riguroso el rendimiento de aplicaciones ejecutadas sobre máquinas virtuales que proporcionan compilación JIT.

La metodología de evaluación considera dos aproximaciones: *start-up* y *steady-state*. El rendimiento *start-up* mide cómo de rápido un sistema puede procesar una ejecución relativamente corta de una aplicación, considerando el tiempo de ejecución de todo el proceso incluyendo la compilación JIT. El rendimiento *steady-state* considera las ejecuciones de larga duración, típicamente de aplicaciones servidoras, en las que la compilación JIT ya no implica una variabilidad significativa en el tiempo total de ejecución. Este mecanismo de evaluación está orientado a analizar los beneficios de la optimización dinámica de código, mediante la detección de *hot-spots*.

Para medir el rendimiento *start-up*, la metodología identifica dos pasos:

1. Medir el tiempo de ejecución de correr múltiples veces el mismo programa. Esto generará n (se ha tomado $n = 30$) mediciones x_i de tiempo de ejecución del programa, con $1 \leq i \leq n$.

2. Calcular el intervalo de confianza para un nivel de confianza dado (tomamos un nivel de confianza del 95%). El intervalo de confianza es calculado para eliminar posibles errores en las mediciones que puedan introducir sesgos en la evaluación. El intervalo de confianza se calcula usando la distribución t de *Student* ya que hemos tomado $n = 30$ [Lilja00]. Por lo tanto, el intervalo de confianza $[c_1, c_2]$ se define como:

$$c_1 = \bar{x} - t_{1-\propto/2;n-1}\frac{s}{\sqrt{n}} \qquad\qquad c_2 = \bar{x} + t_{1-\propto/2;n-1}\frac{s}{\sqrt{n}}$$

Donde $\bar{x}$ es la media aritmética de las mediciones x_i, $\alpha = 0.05$ (95%), s es la desviación estándar de las x_i y $t_{1-\propto/2;n-1}$ está definida como una variable aleatoria T, que sigue la distribución t de *Student* con $n-1$ grados de libertad y, por tanto, cumple que

$$\Pr\left[T \leq t_{1-\propto/2;n-1}\right] = 1 - \propto/2$$

Los datos que proporcionamos (Apéndice B y Apéndice C) para cada medición son la media del intervalo de confianza más un porcentaje indicando la anchura dicho intervalo de confianza con respecto a la media.

En esta evaluación no hemos utilizado la metodología *steady-state* puesto que .NET no dispone de optimizaciones dinámicas de código (*hot-spot*) y se introduciría un sesgo entre las distintas implementaciones de máquinas virtuales.

El consumo dinámico de memoria también sigue la metodología descrita. Para este propósito, se ha usado el tamaño máximo del conjunto de memoria de trabajo (la propiedad *PeakWorkingSet*) empleado por el proceso desde que fue arrancado. El conjunto de memoria de trabajo de un proceso es el grupo de páginas de memoria visibles por el proceso en la memoria física RAM en un momento dado. Estas páginas están residentes y disponibles para la aplicación para ser usadas sin lanzar un fallo de página. Este conjunto de memoria incluye tanto los datos compartidos como los privados. Los datos compartidos comprenden las páginas que contienen todas las instrucciones que el proceso ejecuta, incluyendo las de los módulos de proceso y las de las librerías del sistema. Hemos medido la propiedad *PeakWorkingSet* con llamadas explícitas al *Windows Management Instrumentation*, WMI [Microsoft12].

En cuanto al modo de presentar los resultados, si las plataformas *P1* y *P2* ejecutan el mismo *benchmark* en T y $2.5 \times T$ milisegundos respectivamente, diremos que *P1* es un 150% (o 1,5 veces) más rápido que *P2*, el rendimiento en tiempo de ejecución de *P1* es un 150% (o 1,5 veces) más alto que el de *P2*, *P2* requiere 150% (o 1,5 veces) más tiempo de ejecución que *P1* o el beneficio en el rendimiento de *P1* comparado con *P2* es de 150% –lo mismo se aplica para el consumo de memoria. Para calcular la media de porcentajes, factores y órdenes de magnitud, usamos la media geométrica.

8.1.4 Entorno de Medición

Todos las mediciones han sido llevadas a cabo sobre un sistema Intel Core I7 3610QM a 2.30 GHz con 4GB de memoria RAM y una versión de 64 bits actualizada de Microsoft Windows 7 Professional SP1. La versión usada de la plataforma .NET es la 4.0.30319 de 64 bits en modo *release* y la de la plataforma Java es la 1.7.0_05 de 64 bits.

Los *benchmarks* fueron ejecutados después de reiniciar el sistema, eliminando las cargas ajenas y esperando a que el sistema operativo estuviese completamente cargado (el momento en que el uso de la CPU cae por debajo de un 2% y permanece a ese nivel por 30 segundos).

8.2 Micro-benchmark

El objetivo del micro-benchmark es medir cada una de las primitivas de las plataformas orientadas a aspectos. Para evaluar su rendimiento y con-

sumo de memoria, creamos un primer componente sintético que ejecuta un método `addInt` que será tejido estática y dinámicamente con varios micro-aspectos (Figura 29).

El componente de la Figura 29 ejecuta inicialmente el método `addInt` (línea 15). En función de la operación a medir, esta primera invocación puede ser sustituida por la operación de lectura o escritura de un campo, o construcción de un objeto. Esta primera operación se tejerá con el objetivo de realizar la carga de todas las librerías necesarias para ejecutar el aspecto, pero su tiempo de ejecución no es evaluado.

Posteriormente, la aplicación invoca la misma operación tejida 10.000.000 veces (líneas 22 y 23), repitiendo este proceso 10 iteraciones (líneas 20 a 26). Finalmente ejecuta un método de test (línea 29), también tejido con un aspecto de test, para verificar que la computación se ha realizado correctamente.

```
01: public class Program {
02:
03:    public int addInt(int b, int c, int d) {
04:           return b + c + d;
05:    }
06:
07:    public int testAddInt(int b, int c, int d) {
08:           return b + c + d;
09:    }
10:
11:    public Program() {}
12:
13:    public static void main(String[] args)  {
14:           Program a = new Program();
15:           int auxiliar = a.addInt(5, 6, 27);
16:
17:           long beginLong = 0, endLong = 0;
18:           int iterations = 10, totalTime = 0;
19:           int j = 0;
20:           for ( ; j < iterations; j++) {
21:                  beginLong = System.currentTimeMillis();
22:                  for (long i = 0; i <= 10000000; i++)
23:                         auxiliar = a.addInt(5 + j, 6 + j, 27 + j);
24:                  endLong = System.currentTimeMillis();
25:                  totalTime = totalTime + (endLong - beginLong);
26:           }
27:           System.out.println("The media duration is " +
28:                                totalTime / iterations);
29:           auxiliar = a.testAddInt(5 + j, 7 + j, 21 + j);
30:    }
31: }
```

Figura 29: Código aplicación sintética en Java.

El método tejido `addInt` recibe tres parámetros de tipo entero y devuelve otro entero (líneas 3 a 5) –excepto en el caso del momento de ejecución *around* sin *proceed*, que no devuelve nada.

Se miden las operaciones de cuatro tipos de componentes: invocación a métodos, constructores, lectura de campos/propiedades y escritura de campos/propiedades. En la Figura 30 mostramos las signaturas de cada tipo de componente a tejer. En el caso de la llamada a constructores, éstos también recibirán tres parámetros enteros. Las operaciones se miden para todos los puntos de enlace y tiempos (*before*, *after* y *around*). En el caso de *around* medimos esta operación con y sin invocación a *proceed* por parte del aspecto.

```
01:    public int addInt(int b, int c, int d)
02:    public int testAddInt(int b, int c, int d)
03:    public ProgramConstructor(int value1, int value2, int value3)
04:    int getValue()
05:    void setValue(int value)
```

Figura 30: Signaturas de los componentes a tejer en AspectJ.

Para evaluar la eficiencia de las primitivas ofrecidas por las plataformas orientadas a aspectos elegidas, hemos creado un conjunto de micro-

aspectos que inyectaremos estática y dinámicamente sobre los componentes sintéticos. Estos aspectos tienen un coste computacional muy bajo ya que nuestro objetivo es medir el coste de intercepción de cada punto de enlace. La operación básica que realizan es acceder a la información estática y/o dinámica que se pasa al aspecto.

En la Figura 31 se muestra el código de un micro-aspecto codificado en AspectJ. Este código dependerá de la información que recibe el aspecto (§ 7.4):

- *Without Reflection* (líneas 1 a 4). El aspecto no recibe ninguna información como parámetro. La única operación que realiza es actualizar un contador que actuará como testigo posteriormente en el aspecto de test.

- *Static Part* (líneas 5 a 9). El aspecto recibe información estática (`thisJoinPointStaticPart`) como el nombre y la clase del método interceptado. Esta información es convertida en un número calculando el número de caracteres y actualizará el contador que actuará como testigo.

- *Dynamic Part* (líneas 10 a 21). El aspecto recibe la información dinámica a través de `thisJoinPoint`, una referencia al objeto donde está el método que generó su ejecución. Esta referencia también ofrece los parámetros en tiempo ejecución del método interceptado. La información también es convertida a un número, actualizando el contador.

- *Full* (líneas 22 a 35). El aspecto recibe información tanto estática (líneas 24 a 27) como dinámica (líneas 28 a 34), convirtiéndola a un número y actualizando el contador.

```
01: // WithoutReflection
02: before() : beforeAddInt() {
03:   ValorSuma.sumTotal++;
04: }
05: // Static Part
06: before() : beforeAddInt() {
07:   ValorSuma.sumTotal += thisJoinPointStaticPart.getKind().length()+
08:     thisJoinPointStaticPart.getSignature().toString().length();
09: }
10: // Dynamic Part
11: before() : beforeAddInt() {
12:   ValorSuma.sumTotal += thisJoinPoint.getArgs().length +
13:             thisJoinPoint.getTarget().toString().length();
14:   Class[] types= ((CodeSignature)thisJoinPoint.getSignature())
                        .getParameterTypes();
15:   Object[] args = thisJoinPoint.getArgs();
16:
17:   for (int i = 0; i < args.length; i++) {
18:     ValorSuma.sumTotal +=  types[i].getName().length() +
19:       args[i].toString().length();
20:   }
21: }
22: // Full
23: before() : beforeAddInt() {
24:   ValorSuma.sumTotal += thisJoinPointStaticPart.getKind().length()+
25:     thisJoinPointStaticPart.getSignature().toString().length() +
26:     thisJoinPoint.getArgs().length +
27:     thisJoinPoint.getTarget().toString().length();
28:   Class[] types= ((CodeSignature)thisJoinPoint.getSignature())
                    .getParameterTypes();
29:   Object[] args = thisJoinPoint.getArgs();
30:
31:   for (int i = 0; i < args.length; i++) {
32:     ValorSuma.sumTotal +=  types[i].getName().length() +
33:       args[i].toString().length();
34:   }
35: }
```

Figura 31: Código de los aspectos por información recibida de AspectJ.

8.2.1 Puntos de Corte

Para la elección de los puntos de corte y los momentos de ejecución, hemos seleccionado los proporcionados por DSAW. Los puntos de corte son *Method Execution*, *Method Call*, *Constructor Execution*, *Constructor Call*, *Field Set* y *Field Get*. Los momentos de ejecución son *before*, *after* o *around*.

En la Tabla 1 mostramos los puntos de corte y tiempos de ejecución soportados por cada una de las plataformas. "*Todos*" significa que la plataforma contempla los tres momentos de ejecución (*before*, *after* y *around*). Aunque PROSE ofrece más puntos de corte, no hemos sido capaces de realizar esas mediciones debido a la ausencia de documentación.

	DSAW	DSAW	AspectJ	JBoss	JAsCo	PROSE	Gripper-	Rapier-	Spring	Spring

	Estático	Dinámico	AOP				Loom	Loom	Java	.Net
Method Call	Todos	Todos	Todos	Todos	Todos	Before, After	Todos	Todos	Todos	Todos
Method Execution	Todos	Todos	Todos	Todos						
Constructor Call	Todos	Todos	Todos	Todos			Todos	Todos		
Constructor Execution	Todos	Todos	Todos	Todos						
Field Get	Todos	Todos	Todos	Todos		Around	Before, After	Before, After		
Filed Set	Todos	Todos	Todos	Todos		Around	Todos	Todos		
Total	**18**	**18**	**18**	**18**	**3**	**4**	**11**	**11**	**3**	**3**

Tabla 1: Conjunto de los puntos de corte ofrecidos por las plataformas.

8.2.2 Tiempo de Ejecución

A continuación presentamos los tiempos de ejecución de los distintos puntos de enlace de las plataformas identificadas en § 8.1.2, agrupados por los momentos de ejecución y por la cantidad de información que se le pasa al aspecto como parámetro. En el Apéndice B, están las tablas de datos con los tiempos de ejecución en milisegundos y sus porcentajes de error. Para la mayor parte de las mediciones hemos obtenido un porcentaje de error inferior al 2% (con un intervalo de confianza del 95%).

En los gráficos mostrados, para cada punto de corte se presentan las medias geométricas de los momentos de ejecución *before*, *after*, *around* con *proceed* y *around* sin *proceed,* agrupadas por el tipo de información enviada al aspecto, *Without Reflection* (WR), *Static Part* (SP), *Dynamic Part* (DP) y *Full*. Los datos mostrados son relativos a AspectJ en el caso de tejedores estáticos, y a JAsCo o DSAW Dinámico en el caso de tejedores dinámicos.

8.2.2.1 Method Call

El punto de corte *Method Call* es implementado por todas las plataformas analizadas. Además, todas ellas, a excepción de PROSE, contemplan los tres momentos de ejecución. PROSE aunque soporta el momento *around* en la última versión, no permite la ejecución del código original mediante una llamada de tipo de *proceed* (utilizando la terminología de AspectJ).

En la Figura 32 mostramos un gráfico comparando los tejedores seleccionados para el punto de corte *Method Call*. Este gráfico se divide en 2 partes con 2 subpartes cada una. En la parte superior se muestran los tejedores estáticos y en la inferior el tejido dinámico. Los tiempos de ejecución están agrupados por los casos WR (*Without Reflection*), SP (*Static Part*), DP (*Dynamic Part*) y *Full*. Los datos de los tejedores estáticos están calculados con respecto AspectJ y los de los dinámicos con respecto JAsCo.

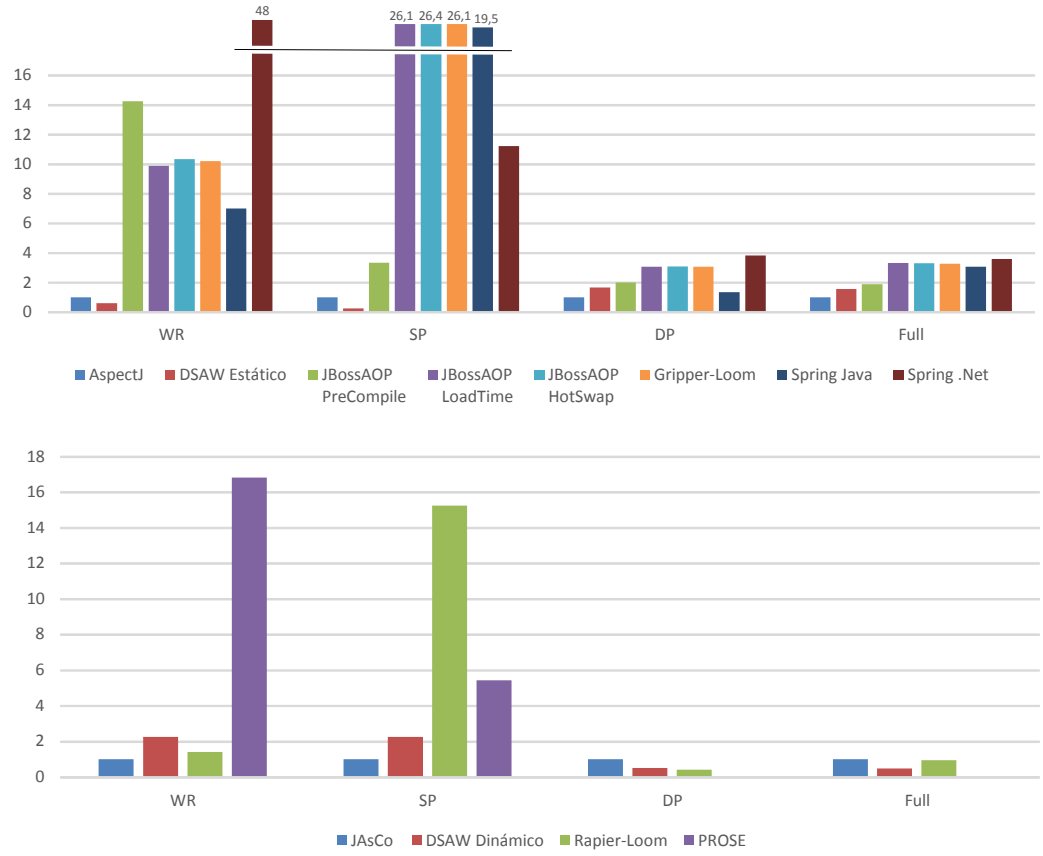


Figura 32: Tiempo de ejecución del punto de corte *Method Call*.

En el caso WR y SP, DSAW estático ofrece el mejor rendimiento de todas las plataformas, siendo un 62,6% y 346% superior a AspectJ, respectivamente. En estos dos escenarios, las instrucciones que inyectan AspectJ [Hilsdale2004] y DSAW para manipular la pila de la máquina virtual antes y después de llamar al aspecto son similares en cantidad y complejidad. En WR no se pasa ningún parámetro y en SP se pasa un único parámetro en ambas plataformas. Por este motivo, parece que el resultado obtenido podría ser debido a un mejor comportamiento de la plataforma .NET con respecto a JVM. El resto de los tejedores estáticos tienen un tiempo de ejecución entre 11 (en el caso Gripper-Loom) y 80 veces mayor (Spring Java) que DSAW estático.

En la parte superior derecha de la Figura 32 se muestran los escenarios DP y *Full,* donde la información enviada al aspecto es mayor. En este caso, DSAW estático tiene un tiempo de ejecución un 35,1% y 31,7% superior a AspectJ, la plataforma más rápida en estos dos escenarios. Esta penalización está provocada por el número de parámetros pasados a los aspectos. AspectJ sólo utiliza dos parámetros, `thisJoinPoint` para el caso DP, y éste y `thisJoinPointStaticPart` para *Full*. Sin embargo, DSAW estático utiliza 5 parámetros para el caso DP (líneas 1 y 2 de la Figura 33) y 9 parámetros para el caso *Full* (líneas 4 a 6 de la Figura 33). Esto provoca más operaciones para manipular la pila de la máquina virtual y un mayor tiempo de ejecución, aunque la legibilidad del código es mayor.

```
01: static public object traceTimeDynamicPart(string member,
02:    object ResultVal, object[] Params, object OBJECT_THIS)
03:
04: static public object traceTimeFull(string ns, string cl,
05:    string member, TypeOfMembers type, JPoint jp, Time time,
06:    object ResultVal, object[] Params, object OBJECT_THIS)
```

Figura 33: Cabecera aspectos para DP y Full de DSAW.

Para el caso DP, el resto de tejedores estáticos son más lentos que AspectJ: las variantes de JBoss AOP (167,8%), Gripper-Loom (207,7%), Spring Java (35,5%) y Spring .Net (282,5%). DSAW estático se comporta mejor que todos tejedores a excepción de Spring Java. Para el caso *Full*, DSAW estático es el segundo sistema más rápido, después de AspectJ. Las variantes de JBoss AOP son las mejores después de DSAW, con una penalización 2 veces superior a nuestra plataforma.

La parte inferior de la Figura 32 muestra los tiempo de ejecución de los tejedores dinámicos (DSAW, JAsCo, PROSE y Rapier-Loom) respecto a JAsCo. El tejedor más rápido para los casos WR y SP es JAsCo, siendo su rendimiento en ambos casos 125% mejor que DSAW dinámico. No obstante, estos tiempos cambian para DP y *Full*, siendo DSAW 99,6% y 107,3% más rápido que JAsCo. Esta diferencia se debe al pequeño coste del enlace dinámico en la generación de código dinámica realizada (§ 7.8) que JAsCo no realiza. En cuanto se realiza alguna computación no trivial en el aspecto o se pasan varios parámetros (DP y Full) esta diferencia no es significativa y DSAW pasa a ser más rápido que JAsCo. Debido a esto, el rendimiento de las pruebas reales presentadas en la Sección 8.4 DSAW es más rápido que JAsCo.

En *Full*, DSAW es el mejor tejedor dinámico siendo un 107,3% y un 95,2% más rápido que JAsCo y Rapier-Loom respectivamente. Para el caso DP, Rapier-Loom obtiene un rendimiento 23% superior a DSAW.

Hemos analizado el coste del código instrumentado por el JPI de DSAW. Si instrumentamos el componente para tejido dinámico sin tejer ningún aspecto, se produce una penalización del rendimiento de un 73,2% en el caso WR. Esta penalización se produce por la comprobación que se realiza en cada punto de enlace instrumentado, chequeando para ver si hay algún aspecto tejido de forma dinámica. Supone la octava parte si se inyectan aspectos dinámicamente, y se va haciendo menor según se pasa más información al aspecto. De hecho, esta penalización es sólo un 0,02% en el caso *Full*. Este mismo comportamiento se repite en el resto de los puntos de corte.

8.2.2.2 Method Execution

El punto de corte *Method Execution* sólo es implementado por AspectJ, DSAW y JBoss AOP. Todos ellos soportan los momentos de ejecución *before*, *after* y *around* (con y sin *proceed*).

En la Figura 34 mostramos un gráfico comparando el coste de este punto de enlace para cada tejedor con respecto a AspectJ (la Tabla B.3 muestra la lista completa de los tiempos de ejecución). Además, se incluye DSAW dinámico junto con los tejedores estáticos, al ser la única plataforma dinámica que incluye este punto de enlace. Los datos están agrupados por la información enviada al aspecto (WR, SP, DP y *Full*).

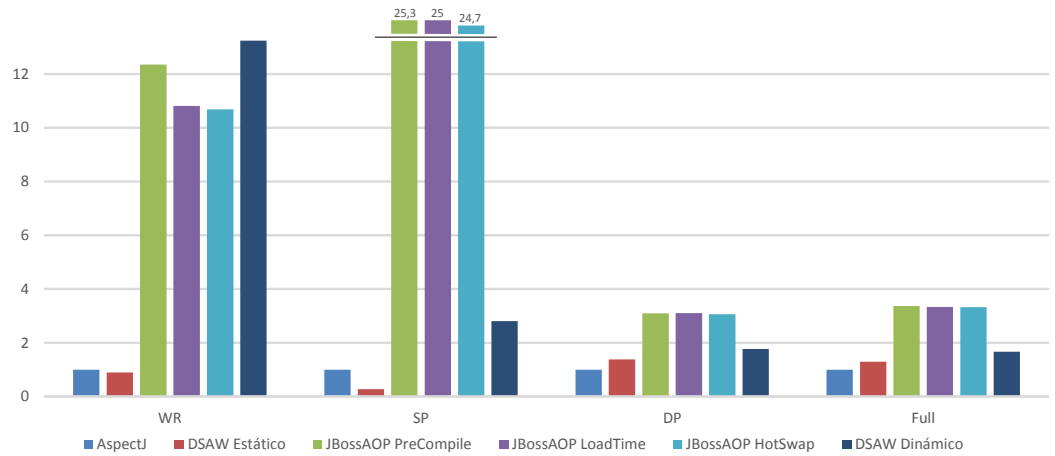


Figura 34: Tiempo de ejecución del punto de corte *Method Execution*.

Al analizar los datos de la Figura 34, vemos que las plataformas que soportan *Method Execution* se comportan de forma similar a lo visto en el punto de corte *Method Call*. DSAW estático tiene un rendimiento mejor con respecto a AspectJ para el caso WR (11% mejor) y SP (72,5%), siendo la mejor plataforma en estos dos casos. En los casos DP o *Full*, DSAW estático obtiene el segundo mejor rendimiento, siendo un 38,6% y 29,4% más lento que

AspectJ. En este caso DSAW es la segunda plataforma más rápida, ofreciendo un mejor rendimiento (129% y 258%) que JBoss AOP.

DSAW es la única plataforma que proporciona tejido dinámico para este punto de corte, por lo que no podemos comparar su rendimiento con tejedores de su mismo tipo. Aun así, podemos apreciar en la Figura 34 que su rendimiento es incluso mejor que un tejedor estático para todos los casos excepto WR: JBoss AOP.

8.2.2.3 Constructor Call

El punto de corte *Constructor Call* es implementado por AspectJ, DSAW, JBoss AOP y LOOM.Net. Puesto que la llamada al constructor debe devolver un nuevo objeto, en el *around* se han realizado mediciones con invocación a *proceed*.

En la Figura 35, vemos los tiempos de ejecución para este punto de corte (detallados en la Tabla B.4). Todos los tiempos de ejecución, tanto dinámicos como estáticos, han sido divididos por los obtenidos al ejecutar AspectJ. Para WR, AspectJ es 2 veces más rápido que DSAW, siendo nuestra plataforma la segunda más rápida en tejido estático. La más próxima es JBoss AOP en su variante *preCompile*, que tarda un 81,6% más que DSAW en ejecutar el mismo programa. En el caso SP, DSAW estático tarda 25% más que AspectJ, y los otros tejedores estáticos empeoran su rendimiento considerablemente. En este caso, JBoss AOP *preCompile* es 21 veces más lento que DSAW.

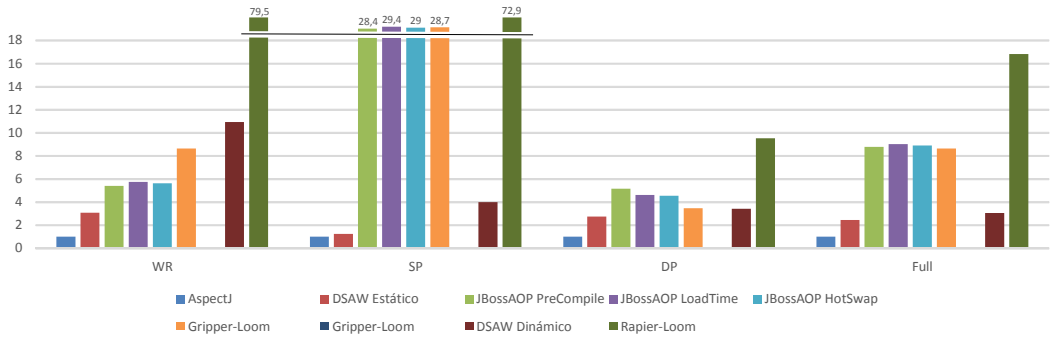


Figura 35: Tiempo de ejecución del punto de corte *Constructor Call*.

Para DP y *Full*, DSAW estático es 1,7 y 1,4 veces más lento que AspectJ respectivamente. También vemos cómo en estos dos escenarios DSAW estático sigue teniendo un mejor rendimiento que el resto de los tejedores estáticos.

Respecto al tejido dinámico, sólo DSAW y Rapier-Loom permiten interceptar este punto de corte. En la Figura 35 se puede apreciar como DSAW tiene un mejor comportamiento que Rapier-Loom, siendo éste entre 1,7 (DP) y 17 (SP) veces más rápido que Rapier-Loom.

8.2.2.4 Constructor Execution

Este punto de corte solo es implementado por AspectJ, DSAW y JBoss AOP. Igual que la llamada al constructor, la ejecución debe devolver un nuevo objeto, por lo que sólo hemos medido el momento *around* con *proceed*.

Vemos en la Figura 36 (Tabla B.5) que DSAW estático tiene un peor rendimiento que AspectJ. En el caso SP es donde la diferencia es menor, siendo DSAW estático un 25,5% más lento que AspectJ. El otro tejedor estático, JBoss AOP, se comporta peor que DSAW en todos los escenarios siendo de media un 240% más lento que nuestra plataforma.

DSAW es la única plataforma que contempla el tejido dinámico para este punto de enlace, siendo en los escenarios excepto WR más rápido que JBoss.

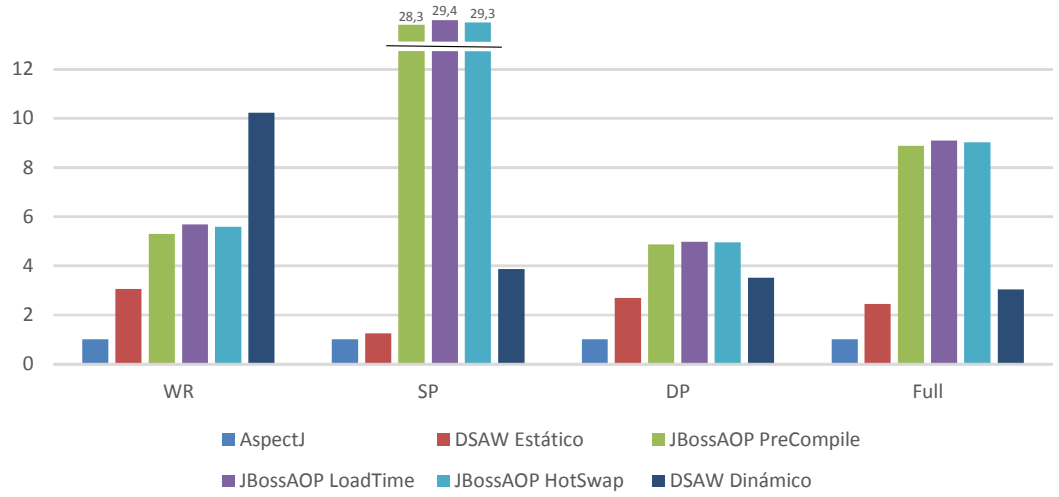


Figura 36: Tiempo de ejecución del punto de corte *Constructor Execution*.

8.2.2.5 Field Get

El punto de corte *Field Get* es implementado por AspectJ, DSAW, JBoss AOP, Loom.NET y PROSE. Todos los tejedores contemplan los tres momentos de ejecución excepto PROSE, quién sólo considera *around*.

En la parte superior de la Figura 37 se muestra el tiempo de ejecución relativo de los tejedores estáticos relativo a AspectJ, mientras que en la parte

inferior se muestran los de las plataformas dinámicas respecto a DSAW (en la Tabla B.6). Para el tejedor dinámico PROSE no hay datos para los escenarios WR y SP (tanto en este punto de corte como en el siguiente, *Field Set*), puesto que sólo implementa el *around*. Además, para los casos DP y *Full* se ha cortado la gráfica ya que, su valor distorsiona los datos de los tejedores restantes.

Vemos como en todos los casos, excepto WR, DSAW estático es más rápido que AspectJ: 62,1% (SP), 38,5% (DP) y 40,3% (*Full*) de beneficio. Para el caso WR, DSAW es 25,2% más lento que AspectJ. Comparando DSAW con el resto de tejedores estáticos, sus tiempos son siempre mejores a excepción Gripper-Loom en el escenario DP, dónde éste es un 51,3% más rápido que DSAW.

En la Figura 37 se aprecia cómo DSAW dinámico es aproximadamente 67 veces más rápido que PROSE. DSAW también obtiene un mejor rendimiento que Rapier-Loom en los casos SP y *Full*, siendo 4,9 y 1,5 veces más rápido respectivamente. En los escenarios WR y DP, el tejido dinámico de Rapier-Loom es el más rápido.

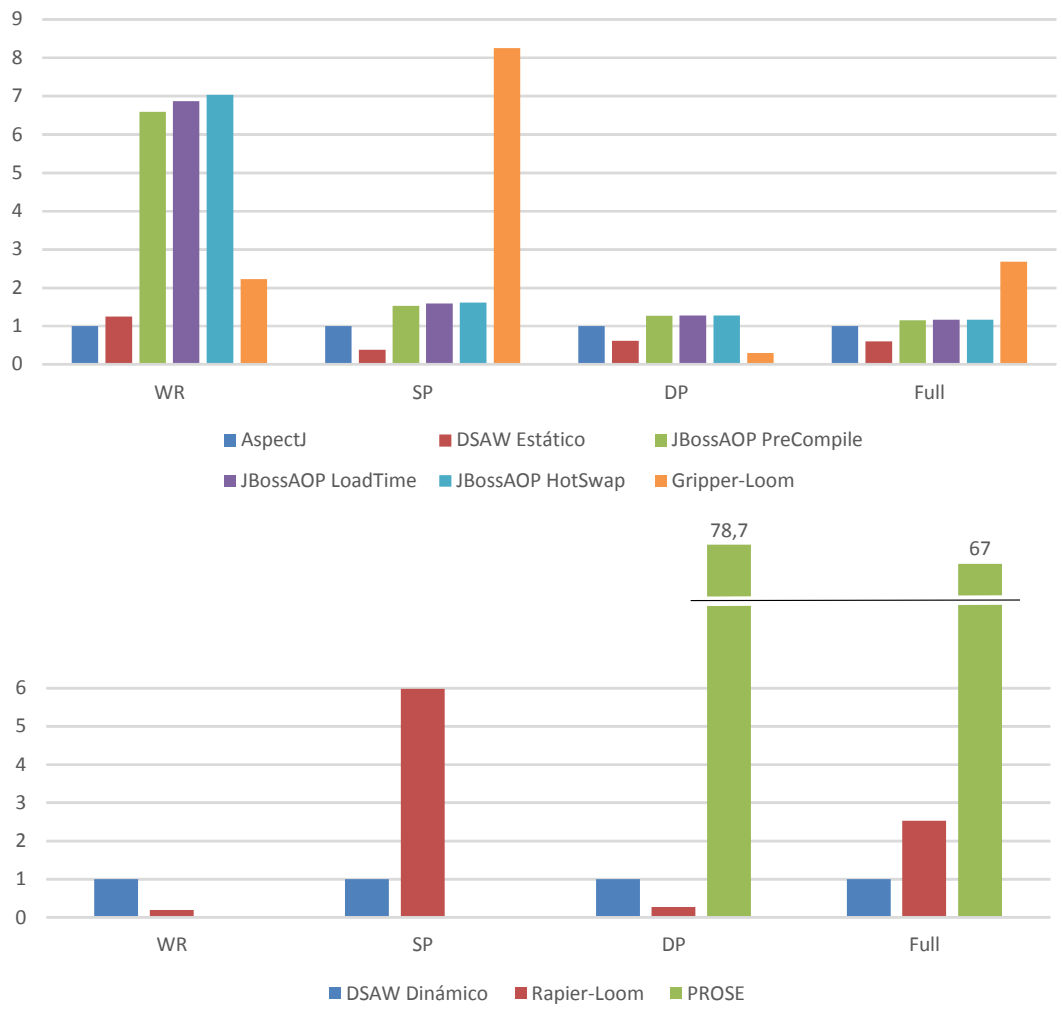


Figura 37: Tiempo de ejecución del punto de corte *Field Get*.

8.2.2.6 Field Set

Las plataformas orientadas a aspectos que contemplan el punto de corte *Field Set* son las mismas que para el punto de corte *Field Get*: AspectJ, DSAW, JBoss AOP, Loom.NET y PROSE (sólo el momento *around*). En la Figura 38 (Tabla B.7) vemos un gráfico representando los datos de rendimiento siguiendo la misma distribución que el gráfico anterior. Los datos de los tejedores estáticos son respecto AspectJ, y los de los tejedores dinámicos respecto DSAW dinámico.

DSAW estático obtiene un rendimiento mejor que AspectJ en todos los casos, siendo un 62,1% (WR), 77,8% (SP), 52,5%(DP) y 55,8% (*Full*) más rápido. Respecto a los otros tejedores estáticos, su rendimiento es mejor en todos los casos excepto para Gripper-Loom en DP, que es un 83,4% más rápido que DSAW.

Respecto al tejido dinámico, la comparación entre DSAW y Rapier-Loom es igual que en el punto de enlace *Field Get*. DSAW es 3,8 y 2,1 veces más lento en WR y DP. Sin embargo, en los casos SP y *Full* DSAW es 5,3 y 1,9 veces más rápido que Rapier-Loom. También podemos apreciar cómo PROSE es la plataforma más lenta, llegando a tener tiempos de ejecución se hasta 129 veces superiores a los de DSAW (dos órdenes de magnitud).

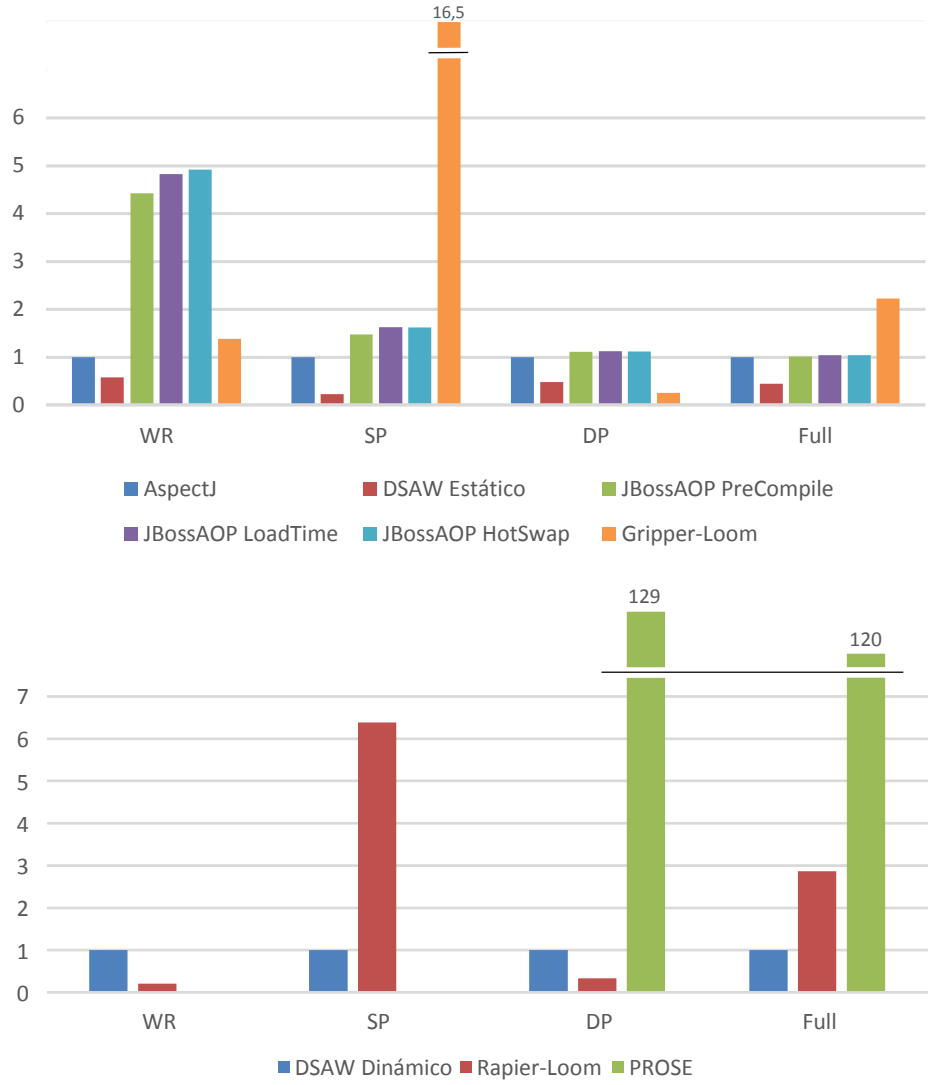


Figura 38: Tiempo de ejecución del punto de corte *Field Set*.

8.2.2.7 Valores Promedio

En esta subsección vamos a resumir los valores promedio de los puntos de enlace de cada plataforma, computando la media geométrica de todos los momentos de ejecución (*before, after* y *around* con y sin *proceed*) y de la cantidad de información que se le pasa al aspecto como parámetro (WR, SP,

DP y *Full*). Todos los tiempos de ejecución serán mostrados relativos a aquellos de AspectJ. Aunque agrupamos los tejedores en estáticos y dinámicos, los valores obtenidos son comparables al estar expresados en la misma escala (tiempo de ejecución relativo a AspectJ). Los datos utilizados para realizar esta evaluación son las medias geométricas de los datos usados en la subsecciones previas de la Sección 8.2.2.

En la Figura 39 se muestran los valores promedio para los puntos de corte *Method Call* y *Method Execution*. En ambos casos DSAW es el tejedor estático más rápido. DSAW es un 27,4% más rápido que AspectJ en *Method Call* y un 14,4% en *Method Execution*. El resto de los tejedores estáticos son más lentos que DSAW y AspectJ, siendo JBoss *precompile* es el más rápido de los restantes (3,3 veces más lento que DSAW).

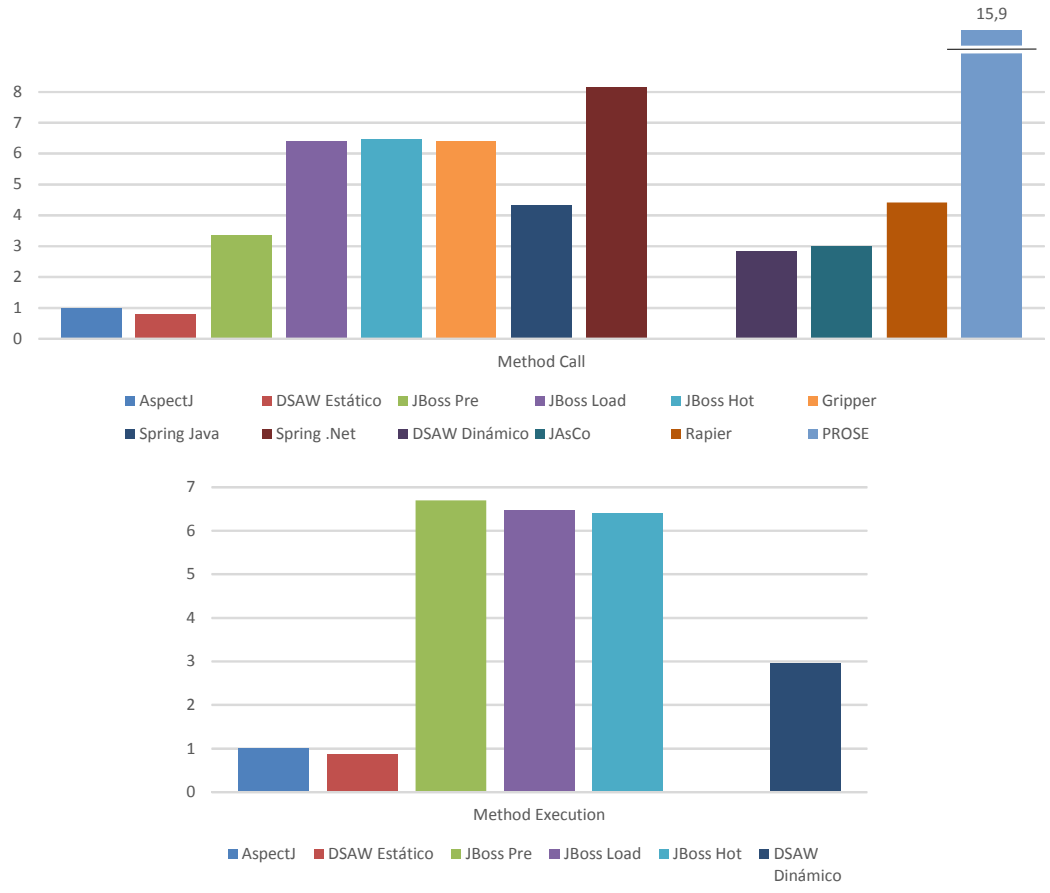


Figura 39: Tiempo de ejecución en *Method Call* y *Execution*.

Para el caso de los tejedores dinámicos, se puede apreciar cómo el tejido dinámico supone un coste de rendimiento adicional, comparando, por ejemplo, los dos tejedores de DSAW y AspectJ con JAsCo. DSAW es el tejedor dinámico más rápido para *Method Call*, y el único que ofrece el punto de enla-

ce *Method Execution*. DSAW dinámico es 5,8% más rápido que JAsCo, y 55% y 591% más que Rapier-Loom y PROSE, respectivamente.

La Figura 40 muestra los tiempos de ejecución de *Constructor Call* y *Constructor Execution* en relación con AspectJ. En ambos puntos de enlace el tejedor estático de DSAW obtiene un rendimiento similar, siendo un 129% y un 131% más lento que AspectJ. Las variantes de JBoss son 2,8 veces más lentas que DSAW. Para el caso de los tejedores dinámicos, DSAW es un 628% más rápido que Rapier-Loom.

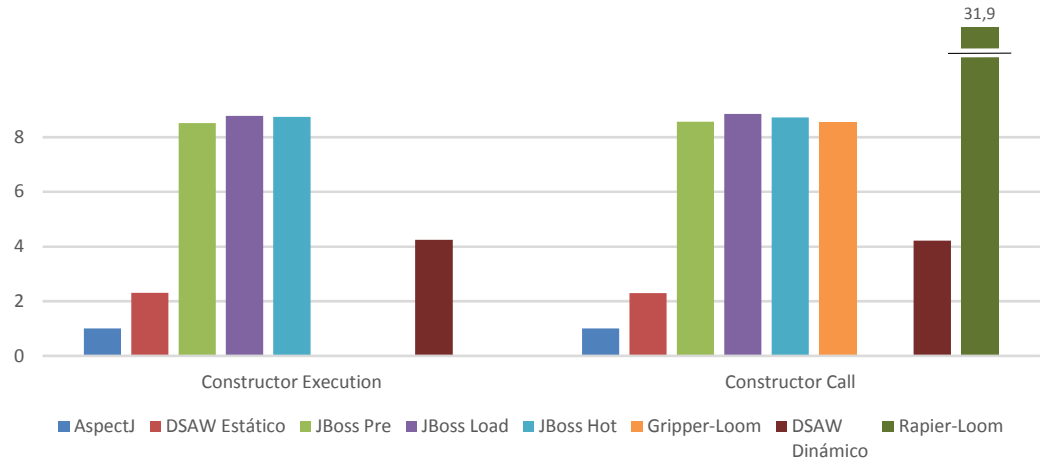


Figura 40: Tiempo de ejecución en *Constructor Call* y *Execution*.

En la Figura 41 se muestran los tiempos de ejecución de los dos puntos de enlace para accesos a campos, relativos a AspectJ. En este caso DSAW estático también proporciona el mejor rendimiento, siendo 56,7% (*Field Get*) y 141% (*Field Set*) más rápido que AspectJ.

En el caso del tejido dinámico, DSAW también obtiene el mejor rendimiento en los dos puntos de enlace. Para *Field Get*, nuestra plataforma es 1,7 y 82 veces más rápida que Rapier-Loom y PROSE respectivamente. En el punto de enlace *Field Set*, DSAW ofrece un rendimiento 66,6% superior a Rapier-Loom y casi dos órdenes de magnitud (99,8 veces) mayor que el de PROSE.

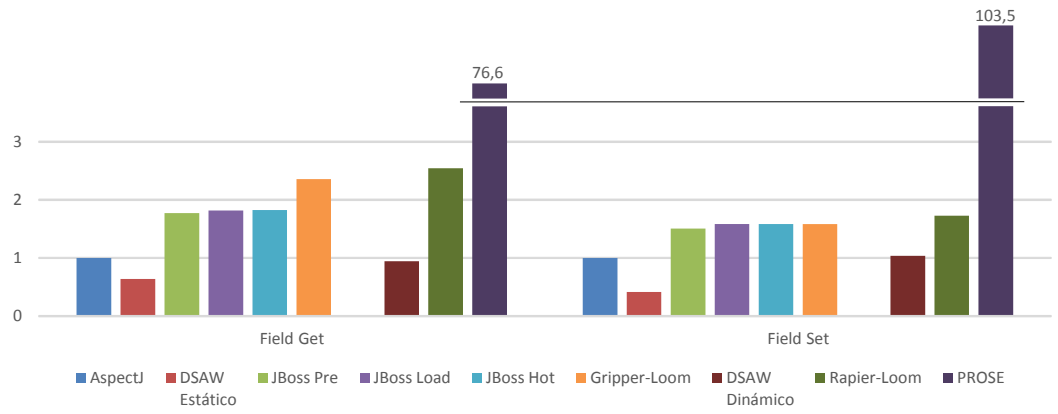


Figura 41: Tiempo de ejecución en *Field Get* y *Set*.

A modo de resumen de los tiempos de ejecución del micro-benchmark, DSAW estático ofrece el mejor rendimiento en todos los puntos de enlace, a excepción de la intercepción de constructores donde ocupa la segunda posición tras AspectJ. En el caso del tejido dinámico, siempre obtiene el mejor rendimiento.

8.2.3 Consumo de Memoria

En el Apéndice C detallamos los consumos de memoria para todos los puntos de corte y momentos de ejecución medidos. En esta sección nos limitaremos a comparar los datos del punto de corte *Method Call*, ya que es el único implementado por todos los tejedores en todos los momentos de ejecución. Para el resto de puntos de corte no existen variaciones significativas (véase el Apéndice C).

La Figura 42 muestra la memoria consumida en el punto de corte *Method Call* en relación con AspectJ (los tiempos absolutos se detallan en la Tabla C.1). En la parte superior están los tejedores estáticos y en la inferior los dinámicos. Los datos están agrupados por la información pasada al aspecto: WR, SP, DP y *Full.*

En todos los casos DSAW estático es la plataforma que consume menos memoria. Si la comparamos con AspectJ, según aumentamos la información enviada al aspecto, se aumenta aún más la diferencia entre el consumo de memoria de estas dos plataformas. En WR, DSAW consume un 60% de la consumida por AspectJ; en *Full* este valor se reduce al 5%. Se aprecia cómo las plataformas desarrolladas sobre .NET son las que menos memoria consumen, siendo este dato muy significativo en los escenarios en los que más información se pasa al aspecto (DP y *Full*).

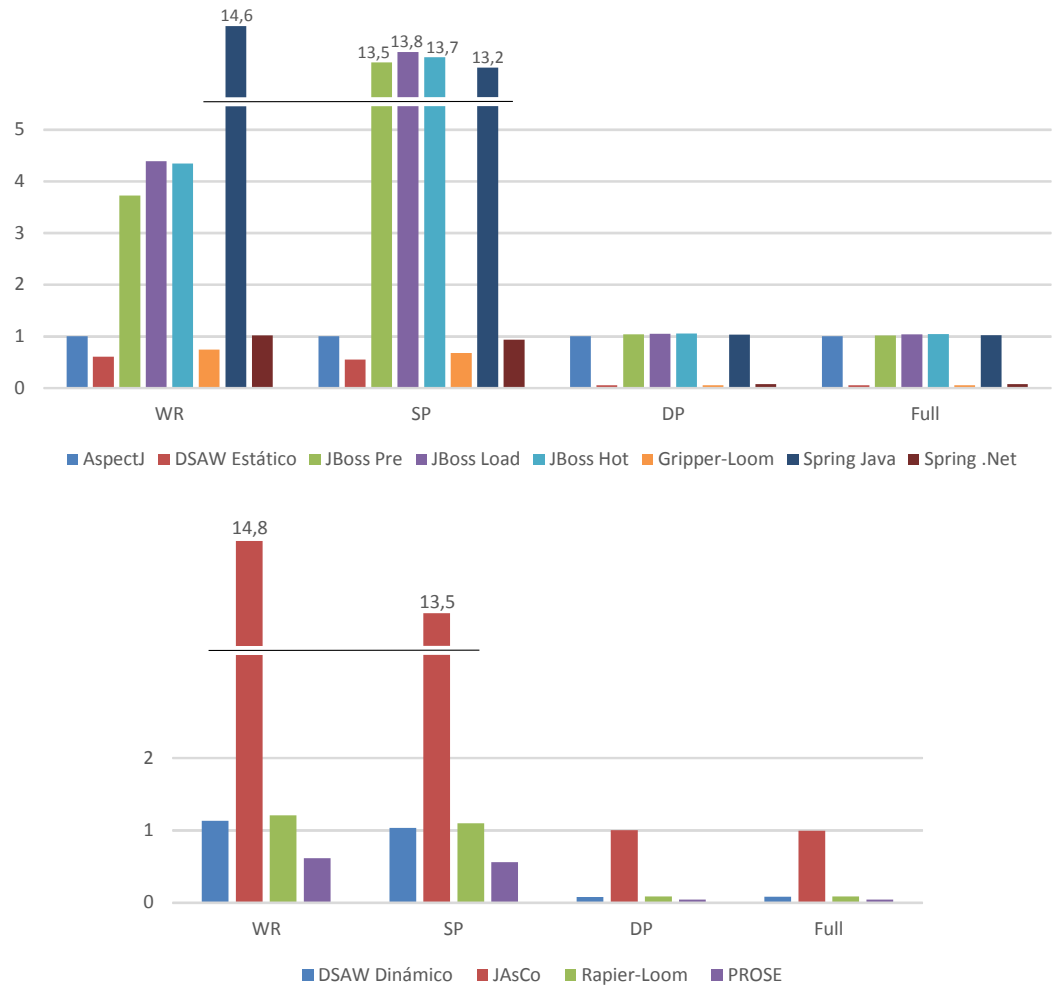


Figura 42: Consumo de memoria en el punto de enlace *Method Call*.

Para el caso de los tejedores dinámicos, PROSE es la plataforma que menos memoria consume, siendo DSAW la segunda con un consumo superior medio del 81,6%. Rapier-Loom requiere un 6,7% más recursos que DSAW y 628% en el caso de JAsCo. Vemos, pues, cómo JAsCo que era ligeramente más rápida que DSAW requiere muchos más recursos de memoria; de forma similar, PROSE, la plataforma dinámica menos eficiente, es la que consume menos memoria. En el caso de PROSE, la diferencia de rendimiento con DSAW (2.082%) es mucho mayor que el beneficio de memoria (81,6%).

8.3 AWBench

El *benchmark* AWBench [Vasseur04] es ampliamente conocido en la evaluación del rendimiento de plataformas orientadas a aspectos. Su objetivo es medir el tiempo de ejecución de las primitivas de la orientación a aspectos, intentando responder a la pregunta de cuánto rendimiento dinámico se pier-

de al ejecutar un aspecto desde una aplicación. Originalmente fue diseñado e implementado en Java. Su código fuente lo hemos traducido a C#, y hemos realizado las modificaciones oportunas para poder usarlo con DSAW, JAsCo y JBoss (con tejido tanto estático como dinámico).

La estructura de la aplicación se basa en ejecutar varias operaciones 20 millones de veces para calcular el tiempo total y por ciclo. Esos datos se recogen y se muestran al final del test. Como en el caso anterior, seguimos aplicando la metodología propuesta por Georges *et al.* [Georges07] para medir el tiempo total de toda la ejecución.

Las operaciones que se ejecutan son llamadas a métodos con y sin parámetros. Estos métodos sólo ejecutan una operación: incrementar un testigo para que el optimizador no elimine el código. La Figura 43 muestra un ejemplo de alguno de los métodos de la aplicación: sin parámetros, con un parámetro de tipo primitivo y con un parámetro de tipo objeto (`Integer`). Todos los métodos devuelven `void`, a excepción de uno que devuelve un `string` (no mostrado en la Figura 43).

```
01:     public void beforeJP() { m_count++; }
02:
03:     public void beforeWithPrimitiveArgs(int i) { m_count++; }
04:
05:     public void beforeWithWrappedArgs(Integer i) { m_count++; }
```

Figura 43: Ejemplo de métodos a tejer.

Sobre esos tres métodos se tejen un grupo de aspectos simples (Figura 44). Las operaciones que hacen estos aspectos son incrementar un campo, evitando que el optimizador elimine la invocación al tratarse de código no operacional. Adicionalmente, en algunos de ellos se accede a la información estática o dinámica del aspecto: en la línea 6, el aspecto accede a la signatura del método usando `thisJoinPointStaticPart`; y en la línea 11 se accede a información dinámica usando `thisJoinPoint`.

```
01: before() : execution(* awbench.method.Execution.before_()) {
02:         Run.ADVICE_HIT++;
03:         }
04:
05: before() : execution(* awbench.method.Execution.beforeSJP()) {
06:         Object sig = thisJoinPointStaticPart.getSignature();
07:         Run.ADVICE_HIT++;
08:         }
09:
10: before() : execution(* awbench.method.Execution.beforeJP()) {
11:         Object target = thisJoinPoint.getTarget();
12:         Run.ADVICE_HIT++;
13:         }
```

Figura 44: Ejemplos aspectos para *before*.

Algunos de los métodos son tejidos con un aspecto en el momento de ejecución *before* (Figura 44) y con otro aspecto en el momento *after* (Figura 45). Otros métodos son tejidos con aspectos en el momento de ejecución *around* (Figura 46).

```
01: after() : execution(* awbench.method.Execution.beforeAfter()) {
02:         Run.ADVICE_HIT++;
03:        }
04:
05: after() returning(String s) :
06:             execution(* awbench.method.Execution
                                  .afterReturningString()) {
07:         String returnValue = s;
08:         Run.ADVICE_HIT++;
09:         }
```

Figura 45: Ejemplos aspectos para *after*.

El aspecto realiza una llamada al componente mediante *proceed* en las líneas 5 y 12 de la Figura 46. En las plataformas en las que no se ofrece esta palabra reservada, se fuerza el tipo del componente y se invoca al método interceptado (evitando hacer uso de reflexión).

```
01: Object around() :
02:          execution(* awbench.method.Execution.aroundSJP()) {
03:         Run.ADVICE_HIT++;
04:         Object o = thisJoinPointStaticPart.getSignature();
05:         return proceed();
06:        }
07:
08: Object around() :
09:          execution(* awbench.method.Execution.aroundJP()) {
10:         Run.ADVICE_HIT++;
11:         Object o = thisJoinPoint.getTarget();
12:         return proceed();
13:         }
```

Figura 46: Ejemplos aspectos para *around*.

8.3.1 Tiempo de Ejecución

Mediremos primero las plataformas con tejido estático, para después evaluar aquéllas que soportan tejido dinámico. En ambos casos hemos seleccionado los tejedores que han obtenido un mejor rendimiento en el micro-benchmark: AspectJ, DSAW y las variantes de JBoss AOP en tejido estático, y JAsCo, DSAW y JBoss AOP para tejido dinámico [Rainone2008]. PROSE no ha sido incluido en la evaluación debido a los pobres resultados obtenidos en el micro-benchmark.

8.3.1.1 Tejido Estático

La Figura 47 muestra los tiempos de ejecución relativos a AspectJ (los tiempos absolutos se detallan en la Tabla B.8). En el tejido estático, DSAW es la plataforma con un mayor rendimiento, siendo un 16,27% más rápida que AspectJ. Aunque las técnicas de tejido utilizadas por DSAW y AspectJ son similares, la principal diferencia entre ambas es la cantidad de información transferida al método. Estas pequeñas diferencias en la información enviada causa esta mejora de rendimiento.

Comparando el rendimiento de DSAW y JBoss AOP, nuestra plataforma es un 347%, 351% y 350% más rápida que las variantes *preCompile*, *loadTime* y *hotSwap* respectivamente.

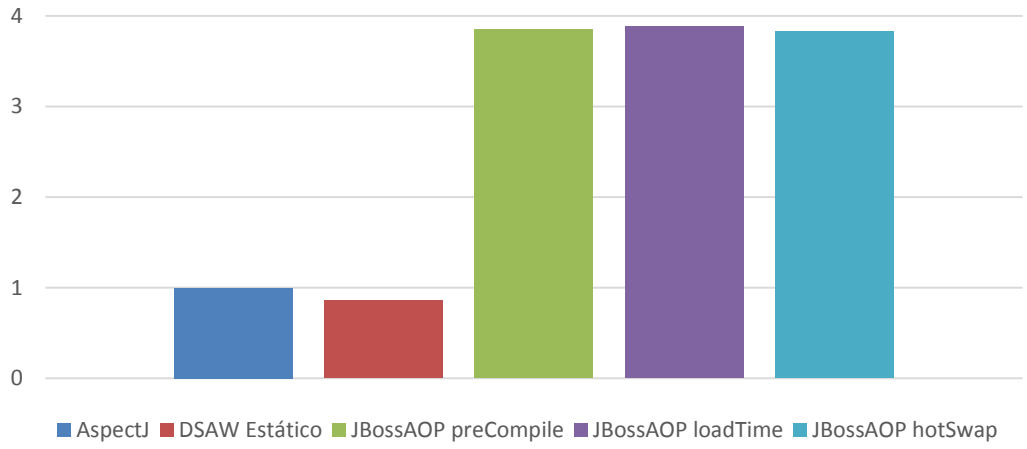


Figura 47: Tiempo de ejecución AWBench de los tejedores estáticos respecto a AspectJ.

8.3.1.2 Tejido Dinámico

En la Figura 48 mostramos un gráfico con los tiempos de ejecución de los tejedores dinámicos relativos a JAsCo (Tabla B.8). En este caso, JAsCo obtiene un rendimiento un 58% mejor que DASW. Esta diferencia es causada

por la forma de hacer el tejido dinámico en ambas plataformas. JAsCo [Fraine05] ha desarrollado Jutta [Vanderperren04] (*Just-in-time combined aspect compilation*). Jutta se basa en cachear y ordenar los aspectos que se tejen en un punto de enlace, para lo cual la librería de manipulación de código Javassist [Chiba00] genera el código en tiempo de ejecución. Este código es reemplazado por el *framework* JAsCo HotSwap mediante Java *agents*, que permiten cambiar el código de la aplicación con el nuevo código generado. La plataforma .NET, en la que se encuentra desarrollado DSAW dinámico, no permite alterar el ensamblado de esta forma, por lo que, cada vez que durante la ejecución de la aplicación se llega a un punto de enlace instrumentado, se deben hacer las validaciones necesarias para ver si algún aspecto ha sido tejido allí.

Puesto que JAsCo evita realizar las comprobaciones dinámicas en cada punto de enlace instrumentado, puede obtener un beneficio de rendimiento elevado en función de la estructura de la aplicación. En este *benchmark*, inicialmente diseñado para tejedores estáticos, se ejecutan operaciones de forma repetitiva sin tejer o destejer aspectos. Por ello, AWBench no considera el coste de tejer o destejer un aspecto en el tiempo de ejecución global de la aplicación (veremos cómo este coste es elevado en JAsCo). En aquellos tipos de aplicaciones donde los aspectos sean tejidos y destejidos dinámicamente con cierta frecuencia, esta alternativa puede provocar un empeoramiento en el rendimiento mayor que otras (véanse las subsecciones siguientes).

Tal y como sucedía en el caso de tejido estático, DSAW es la segunda plataforma dinámica que ha obtenido un mejor rendimiento ejecutando AWBench. DSAW ha sido un 79,7% y un 82,2% más rápida que las dos variantes de JBoss AOP, en tejido dinámico.

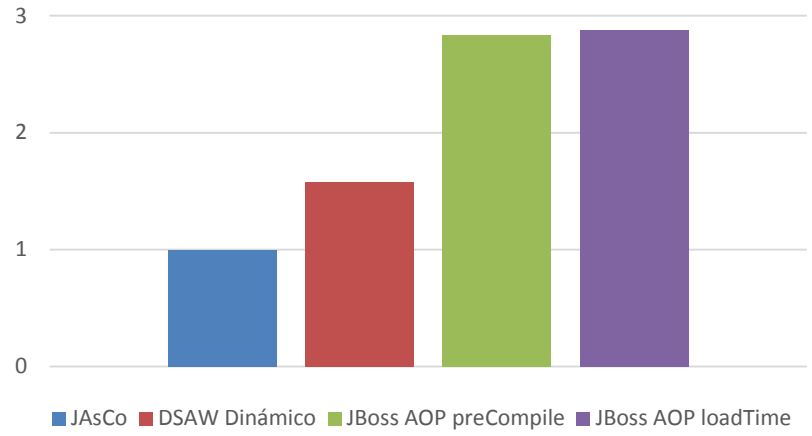


Figura 48: Tiempo de ejecución AWBench de los tejedores dinámicos respecto a JAsCo.

8.3.2 Consumo de Memoria

La Figura 49 muestra un gráfico con los consumos de memoria obtenidos en la ejecución de AWBench. La parte izquierda del gráfico es para los tejedores estáticos y la derecha para los dinámicos. Todos los consumos se muestran relativos a AspectJ. Se aprecia que DSAW, tanto estática como dinámicamente, es la plataforma que menos consume con diferencias muy notables. DSAW estático consume un 5% de la memoria consumida por AspectJ y el tejedor dinámico requiere el 7% de la utilizada por JAsCo.

Debemos resaltar en este punto que DSAW es la única plataforma para .NET, mientras que el resto de las otras plataformas están implementadas sobre la JVM. Tal y como indicamos en el micro-benchmark, los tejedores sobre .NET parecen ofrecer un consumo de memoria mejor que aquellos implementados sobre la plataforma Java. No obstante, medimos la misma aplicación sin la utilización de aspectos obteniendo que Java consumía un 60% más memoria que .NET. Por ello, parece razonable pensar que el mecanismo de tejido implementado por DSAW es más eficiente en consumo de memoria que el resto, en la ejecución de AWBench (esta afirmación será ratificada con la evaluación presentada en la Sección 8.5.3).

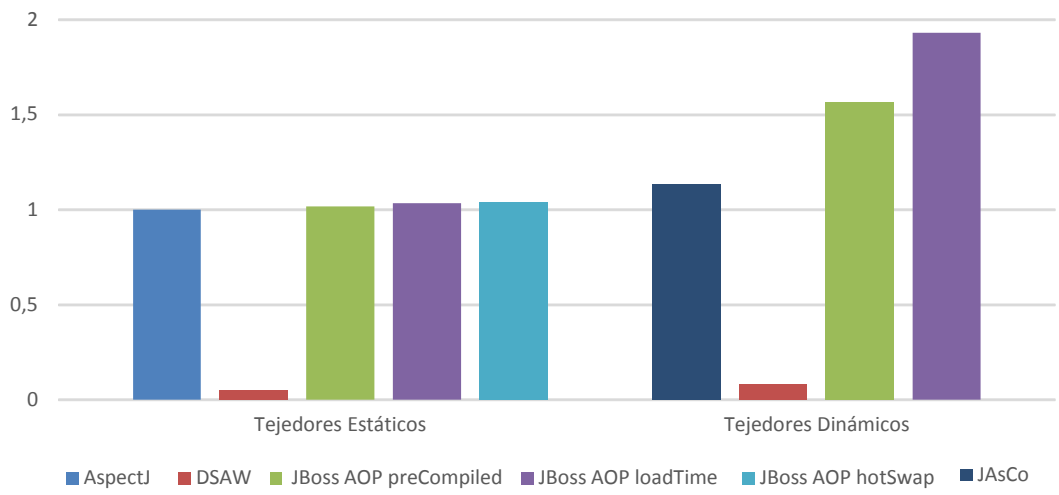


Figura 49: Consumo de memoria AWBench.

8.4 Aplicaciones Reales

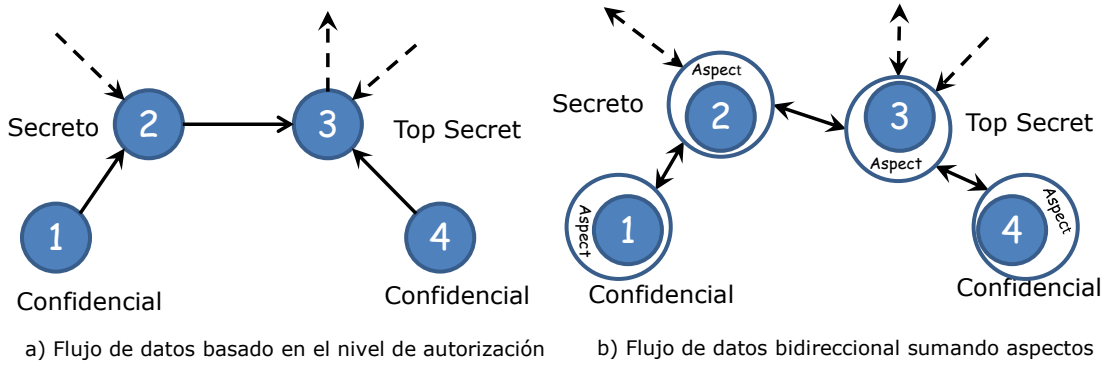
Los dos *benchmarks* evaluados hasta ahora están orientados a medir el coste de las primitivas de orientación a aspectos de un modo sintético. En esos programas, el código de los componentes y de los aspectos apenas consume tiempo de ejecución, siendo casi toda la medición correspondiente al coste de la intercepción. En este punto, sin embargo, evaluamos varias aplicaciones reales que utilizan este paradigma y poseen cargas de trabajo más típicas. Es en estos escenarios en los que podemos apreciar las diferencias más cercanas a las obtenidas en aplicaciones reales.

Para medir las siguientes aplicaciones, partimos de programas orientados a objetos implementadas en Java y C#, y las hemos rediseñado utilizando orientación a aspectos. De este modo, hemos podido aplicar todo el potencial de los aspectos para la construcción de software, ya que hemos analizado estas aplicaciones y decidido qué partes de ellas podrían estar mejor implementadas con aspectos, y si éstos deberían ser tejidos estática o dinámicamente. Las aplicaciones que hemos usado son: un sistema distribuido de control de acceso [Garcia12], un sistema de comunicación para móviles [Garcia12], un servicio de FTP encriptado [Garcia13] y una aplicación de dibujo para pintar figuras geométricas aplicando el patrón arquitectónico MVC [Lago10].

8.4.1 Sistema de Control de Acceso

Los sistemas distribuidos no siempre tienen una topología de red bien definida y es una práctica común que la información deba pasar a través de nodos intermedios para viajar a través de la red (una aproximación que algunas veces se denomina *multi-hop routing* [Broch98]). Esta aproximación no es un problema cuando todos los nodos del sistema tienen los derechos necesarios para acceder a la información que viaja a través del mismo. Sin embargo, pueden existir nodos que no tengan privilegios para ver cierto tipo de información, causando un problema de seguridad. En ese caso, es necesario restringir el acceso a la información, con el objetivo de prevenir que nodos sin permisos adecuados accedan a datos a los que no están autorizados. Esta aplicación implementa un sistema de control de acceso (*Access Control*), estableciendo quién puede acceder a la información, determinando mediante el flujo de datos cómo la información fluye a través de la red.

En la Figura 50.a mostramos un ejemplo. Cada nodo tiene etiquetado un nivel de acceso (confidencial, secreto o *top secret*) y las flechas indican el sentido en el que puede viajar la información. Los nodos 1 y 4 pueden enviar información a cualquier otro nodo de la red porque su nivel es confidencial, el más bajo de la red (cualquier nodo tiene permiso para leer esa información). Sin embargo, el nodo 2 sólo puede enviar información al nodo 3, ya que el nivel de autorización secreto es más bajo que *Top Secret*. No puede enviar información al nodo 1, ya que tiene un nivel confidencial, que no le permite leer información secreta. Finalmente, el nodo 3 no puede enviar información debido a que tiene un nivel de autorización más alto que el de sus nodos contiguos.



a) Flujo de datos basado en el nivel de autorización b) Flujo de datos bidireccional sumando aspectos

Figura 50: Uso de aspectos para modificar el flujo de datos.

Mecanismos tradicionales como CORBA [Lang02] resuelven estos problemas considerando sólo relaciones uno-a-uno, sin tomar en cuenta todos los dispositivos que son parte del sistema distribuido. Aunque esta solución es efectiva en redes cliente-servidor o con topologías prefijadas, en redes punto-a-punto o aquellas con topologías *fuzzy*, el uso de estos mecanismos puede implicar restricciones sobre el flujo de datos. Si un nodo no está autorizado a enviar datos a otro, pero este último es el único modo de conseguir llegar a un tercero, la transmisión de datos al tercer nodo no es posible, incluso sin una restricción específica en el sistema para hacerlo. Un ejemplo de este caso aparece en la Figura 50.a: los nodos 1 y 4 tienen el mismo nivel de acceso, pero no pueden intercambiar mensajes entre ellos porque el nodo 3 no puede enviar mensajes ni al nodo 2 ni al 4.

Otras soluciones analizan la topología de la red con el objetivo de identificar posibles problemas potenciales en la distribución de la información y establecer restricciones sobre el acceso y el flujo de datos [NCSC90]. Para hacer esto, se usan políticas de seguridad con diferentes niveles de autorización para cada nodo. Estos tipos de políticas de seguridad son ampliamente usadas en sistemas militares o para gestión de catástrofes, donde el control de acceso a la información es esencial [NCSC90]. Los nodos sólo pueden enviar la información a aquellos nodos que tienen un nivel de autorización más alto o igual al suyo, y además necesitan conocer los niveles de autorización de aquellos nodos a los que están conectados. El problema es que esta aproximación carece de flexibilidad; cada vez que cambia el número de nodos o la topología de la red, es necesario reanalizar todo el sistema y actualizar las políticas con las nuevas restricciones. Como en el caso previo, en topologías difusas esta solución puede resultar muy costosa, pudiendo apare-

cer restricciones en el flujo de datos que impidan que dos nodos intercambien información.

En los escenarios de sistemas distribuidos, estos tipos de soluciones son difíciles de implementar, principalmente en aquellas redes que sean muy flexibles, donde el número de nodos y la topología cambien muy frecuentemente. En estos casos, los mecanismos de seguridad deben ofrecer una flexibilidad mayor. Debería estar permitido controlar el acceso a la información en cualquier momento sin interferir con el flujo de datos, sin tener en cuenta el número de nodos y la topología del sistema.

Partiendo de una aplicación de intercambio de mensajes como la de la Figura 50.a, hemos rediseñado la aplicación con aspectos para implementar un sistema distribuido con una política de seguridad que garantice la transmisión segura sobre topologías cambiantes. Se permite el etiquetado de datos con los niveles de autorización de los nodos. También se permite añadir de forma sencilla la encriptación de la información. La aplicación fue construida basándose en las operaciones clásicas *send* y *receive*. Los aspectos interceptan estos mensajes para incluir las siguientes funcionalidades:

1. Autentificación para otorgar al usuario el nivel apropiado de autorización.

2. Etiquetado de los datos para determinar cómo la información fluye a través de la red y controlar el acceso a ella.

3. Posibilidad de añadir la encriptación de la información para evitar el acceso no autorizado a ella.

Cuando un nodo envía la información, un aspecto cifra la información y añade a ésta una etiqueta con el nivel de autorización del nodo fuente. En el destino, otro aspecto recibe y verifica si el nivel de autorización del nodo es lo suficientemente alto como para acceder a los datos y, si así es, descifra la información. Si el nivel es suficiente, el aspecto pasa los datos al nodo. Con este mecanismo, la información viaja cifrada y etiquetada con el nivel de autorización, de un modo totalmente transparente a través del sistema.

Aunque el acceso es controlado, el problema de restringir el flujo de datos todavía no ha sido evitado. Para resolverlo, los aspectos pueden tener en cuenta información acerca del nodo fuente. Cuando un nodo intermedio recibe los datos a ser reenviados, éstos se reenvían con el nivel de autorización del nodo origen y no el del propio nodo. Como podemos ver en la Figura 50.b, con esta aproximación todos los nodos pueden enviar datos a cualquier

nodo, independientemente del nivel de autorización que ellos tengan porque los aspectos restringen el flujo de información y controlan el flujo de datos. El nodo 1 puede ahora enviar datos al nodo 4 a través de los nodos 2 y 3. Si un nodo intermedio altera la información de algún modo, la información automáticamente pasará a tener el mismo nivel de autorización y los aspectos restringirán su diseminación.

Para implementar esta solución utilizando orientación a aspectos, es necesario determinar los puntos de corte del sistema distribuido donde se van a tejer los aspectos y la información que se va enviar al aspecto. Podemos usar un tejedor estático para modificar los valores de los parámetros de un método antes de su ejecución y el resultado justamente después de ello. De este modo, los aspectos introducirán el etiquetado de datos y su cifrado cuando enviamos la información, y el descifrado y verificación de su nivel de autorización cuando la recibimos. El punto de enlace que hemos usado es *Method Call*. Como debemos acceder a los parámetros para cambiarlos antes y después de ejecutar el método, hemos utilizado el momento de ejecución *around*. Además, como necesitamos acceder a los parámetros del método, hemos usado el caso *DynamicPart*, para implementar el aspecto.

En la Figura 51 mostramos la sección del código C# que serializa y deserializa los datos para transmitirlos por la red. El código que debemos adaptar son los métodos que se encargan de enviar y recibir la información sobre la red. Los lugares ideales para hacerlo son los métodos `SerializeData` y `DeserializeData` de `Stream`. Estos métodos son responsables de que los datos sean serializados sobre el canal de salida y deserializados sobre el canal de entrada.

```
01:      public Data DeserializeData(Stream streamData) {
02:          return (Data)((new BinaryFormatter())
                    .Deserialize(streamData));
03:      }
04:
05:      public void SerializeData(Stream streamData, Data data) {
06:          BinaryFormatter binaryFormatter = new BinaryFormatter();
07:          binaryFormatter.Serialize(streamData, data);
08:      }
```

Figura 51: Ejemplo simple de código C# para serializar/deserializar los datos.

Uno de los mayores beneficios de usar el paradigma orientado a aspectos es que todos los puntos en el programa responsables de enviar y recibir datos pueden ser tejidos de forma simultánea, usando el tejedor. Se tejerá el aspecto en todas las llamadas a estos métodos a lo largo de programa.

Aunque, desde la perspectiva de la seguridad, debemos tener cuidado para asegurar que todas las llamadas a la red utilizadas sean cubiertas por la especificación del punto de corte.

En la Figura 52 mostramos el *advice* `SensitiyLevelSerialize` implementado para añadir el nivel de autorización a un mensaje. Este *advice* obtiene mediante reflexión la definición del método que provocó la ejecución del aspecto (línea 3) y, posteriormente (línea 5), obtiene el nivel de autorización del nodo fuente. Entonces, si el nodo fuente previamente recibió algún dato y es igual a los que debe enviar ahora (por ejemplo, el nodo está transmitiendo un mensaje), el nivel de autorización que debe usar es el de los datos recibidos. Para etiquetar los datos con el nivel de autorización, el aspecto recupera los datos a enviar de los parámetros del *advice* (línea 7), crea un nuevo objeto de tipo `DataWrapper` con los datos a enviar y el nivel de autorización obtenido (líneas 8 a 11), y llama reflectivamente al método original mediante *proceed* (línea 12). La clase `DataWrapper` hereda de la clase `Data`, haciendo que los métodos `SerializeData` y `DeserializeData` sigan funcionando correctamente.

```
01:  static public object SensitivityLevelSerialize(string member,
02:            object ResultVal, object[] Params, object OBJECT_THIS) {
03:        MethodInfo mi =OBJECT_THIS.GetType().GetMethod(member,
04:                       BindingFlags.Public | BindingFlags.Instance);
05:        int sensitivityLevel = ((Program)OBJECT_THIS).SecurityLevel;
06:
07:        Data dataToSend = (Data)Params[1];
08:        if (receivedData != null &&
09:            receivedData.Information.Equals(dataToSend.Information))
10:          sensitivityLevel = receivedData.SensitivityLevel;
11:        Params[1]= new DataWrapper(dataToSend,
                        sensitivityLevel); // possible encriptado
12:        return mi.Invoke(OBJECT_THIS, Params);
13:  }
14:
15:  static public object SensitivityLevelDeserialize(string member,
16:             object ResultVal, object[] Params, object OBJECT_THIS){
17:        MethodInfo mi =OBJECT_THIS.GetType().GetMethod(member,
18:                       BindingFlags.Public | BindingFlags.Instance);
19:        int sensitivityLevel = ((Program)OBJECT_THIS).SecurityLevel;
20:        DataWrapper dataWrapper = (DataWrapper)mi.Invoke(
21:                                                 OBJECT_THIS, Params);
22:        int resultSensitivityLevel = dataWrapper.SensitivityLevel;
23:        if (resultSensitivityLevel >= sensitivityLevel){
24:            receivedData = dataWrapper;
25:            return new Data(
                     dataWrapper.Information); // posible descifrado
26:        }
27:        return new Data("");
28:  }
```

Figura 52: Código C# de los aspectos para serializar y deserializar.

Al final de la Figura 52 se encuentra el código del *advice* `SensitivityLevelDeserialize`, a tejer en el método de deserialización. Este código descifra los datos (en caso de que los hayamos cifrado) y verifica que el nivel de autorización del nodo sea suficiente para acceder a los datos recibidos; de otro modo, la información se descarta. Como en el caso previo, es necesario obtener el nivel de autorización del nodo (línea 19). Se llama reflectivamente a `DeserializeData` (*proceed*) para deserializar los datos (línea 20). Los datos que devuelve el método son de tipo `DataWrapper`. Se comprueba si el nivel de acceso es apropiado (línea 22) y en ese caso se devuelven los datos (línea 25), en caso contrario se devuelve la información vacía (línea 27).

De este modo, el comportamiento de los nodos es adaptado cuando los mensajes son enviados y recibidos. Los datos son etiquetados en el proceso de envío; antes de ser accedidos, se chequean los permisos de lectura. Usando esta aproximación, el sistema distribuido puede variar en el número de nodos y la topología sin comprometer la seguridad, ya que el control de acceso y el flujo de datos son controlados dinámicamente, y no hay necesidad de analizar la red o actualizar las políticas. El proceso entero trabaja de un modo transparente con respecto tanto a los nodos como al sistema. La funcionali-

dad de la aplicación es modularizada aparte de la comunicación, estando los *concerns* de seguridad implementados como aspectos separados.

8.4.1.1 Tiempo de Ejecución

Hemos elegido los tejedores estáticos que mostraron los mejores rendimientos en ejecución en las mediciones anteriores: AspectJ, DSAW y JBoss AOP. Del tejedor JBoss AOP hemos medido las variantes *preCompile*, *loadTime* y *hotSwap*. Dado que se necesita tejer el aspecto en el momento de ejecución *around* y hacer una llamada reflectiva al método llamador, los *advices* se han implementado usando la función `proceed` proporcionadas por las plataformas orientadas a aspecto. También implementamos la misma aplicación, haciendo uso de reflexión.

El sistema distribuido orientado a objetos se divide en 3 componentes: un nodo servidor, un nodo cliente y un nodo intermedio. Se envían datos desde el cliente al servidor pasando por el nodo intermedio que se encarga de retransmitirlos. El tiempo de ejecución de las aplicaciones en C# y Java no es estadísticamente significativo (su diferencia es menor al 1%, siendo el 2% el margen de error), por lo que no se pueden considerar como estadísticamente distintos.

En el Apéndice B se muestra la Tabla B.9 con los datos de las mediciones realizadas en milisegundos, aplicando la metodología *start-up* propuesta por Georges *et al.* [Georges07]. Todas las mediciones tienen un error inferior al 2% con un intervalo de confianza del 95%.

En la Figura 53 mostramos una gráfica con los datos de las mediciones sobre las implementaciones realizadas con las plataformas orientadas a aspectos. En el gráfico, todos los tiempos de ejecución mostrados son relativos a AspectJ. En la parte izquierda se usa la sentencia `proceed` y en la derecha se utiliza reflexión.

En ambos casos DSAW ofrece el mejor rendimiento, siendo un 3,3% y un 6,3% más rápida que AspectJ en el uso de `proceed` y reflexión respectivamente. Con respecto a las variantes de JBoss AOP, DSAW es un 36,2% (*preCompile*), 50,0% (*loadTime*) y 47,6% (*hotSwap*) más rápida utilizando `proceed`, siendo estos valores 36,5%, 48,7% y 48,5% cuando se usa reflexión.

Aunque los tiempos de ejecución de DSAW son ligeramente superiores a los de AspectJ, en general estarán en valores similares puesto que usamos la misma técnica de tejido. No obstante, en el modo *Full*, DSAW pasa más infor-

mación facilitando el desarrollo de los aspectos pero con una penalización de rendimiento (§ 8.3.1.1) que tiene que ser considerada por el programador.

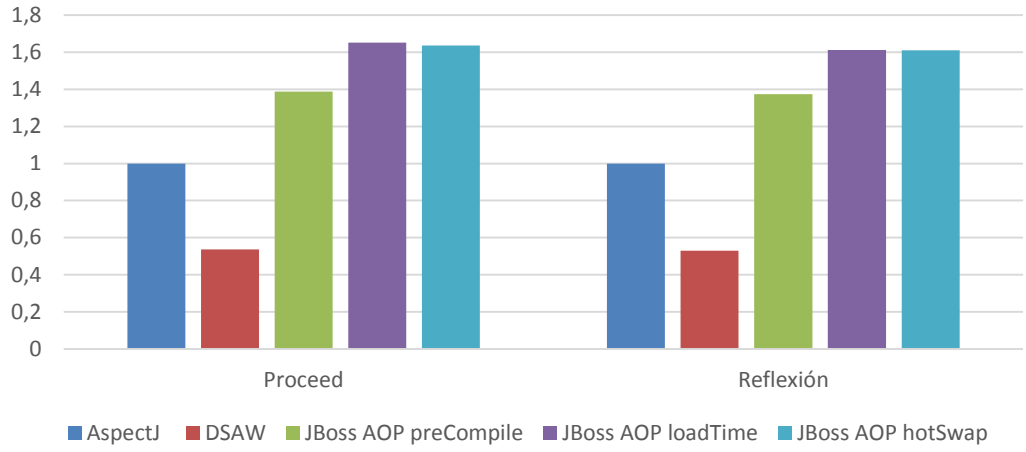


Figura 53: Tiempo de ejecución del Control de Acceso respecto a AspectJ.

8.4.1.2 Consumo de memoria

En la Figura 54 mostramos una gráfica con los datos de consumo de memoria, relativos a AspectJ, para la aplicación de control de acceso. Tanto en el uso de `proceed` como de reflexión, DSAW es la plataforma que menos memoria consume: 46,3% y 47,1% menos que AspectJ. Las variantes de JBoss AOP consumen un 158,56% (*preCompile*), 200,75% (*loadTime*) y un 200,48% (*hotSwap*) más que DSAW de media.

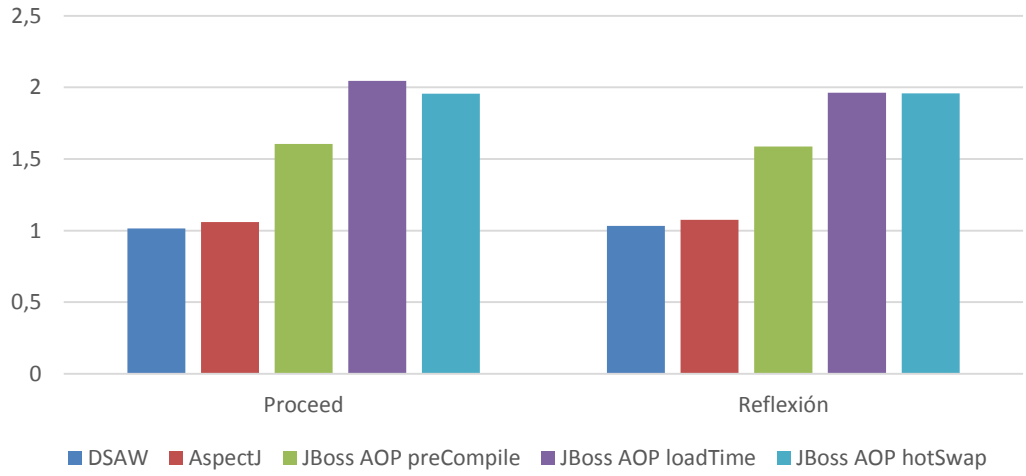


Figura 54: Consumo de memoria del Control de Acceso respecto a AspectJ.

Para ver la influencia de la máquina virtual, hemos medido el consumo de memoria de las implementaciones realizadas en C# y Java sin usar aspec-

tos (las mediciones del consumo de memoria, sin aspectos y con aspectos, se encuentran en la Tabla C.8 del Apéndice C). Hemos calculado la media geométrica del consumo de memoria de los tres componentes del sistema. Con los datos obtenidos vemos que la versión realizada en Java consume un 54% más que la realizada en C#. Vemos pues cómo la máquina virtual tiene una influencia total en el consumo de memoria, sin haber grandes diferencias entre DSAW y AspectJ debido a su similitud en el proceso de tejido.

8.4.2 Comunicaciones Móviles Cifradas

El encriptado de las comunicaciones es una medida de seguridad cuyo objetivo principal es evitar que usuarios no autorizados accedan a la información intercambiada entre los nodos de un sistema. Típicamente se hace usando algoritmos que transforman datos en mensajes ilegibles. Un problema que tiene el cifrado de datos es que puede consumir muchos recursos. Por ejemplo, en sistemas distribuidos compuestos de dispositivos móviles, este proceso es bastante costoso y tiene un gran impacto en el consumo de batería [Alonso04]. Para prevenir la sobrecarga de los dispositivos, es normal usar nodos específicos del sistema con mayor poder de computación para cifrar y descifrar la información, siempre y cuando la conexión a éstos sea segura.

En un escenario concreto podría haber varios dispositivos móviles intercambiando datos a través de una red pública insegura Wi-Fi. Si dos nodos necesitan intercambiar información confidencial, pueden usar una conexión UMTS [Holma00] más segura. Este canal, aunque es más lento, podría tener un nivel más alto de seguridad que el canal público Wi-Fi no cifrado. Una vez que la transmisión de los datos ha finalizado, los dispositivos móviles pueden volver a usar la conexión Wi-Fi original.

En las dos primeras partes de la Figura 55 mostramos los dos posibles escenarios. En el primer escenario, hay una comunicación directa entre los nodos 3 y 4. Entonces, se detecta que hay una zona insegura entre estos dos nodos (segundo escenario), y la comunicación pasa a realizarse a través de dos nodos E/D (*encrypt* / *decrypt*). La ventaja del segundo escenario es que los dispositivos quedan liberados del trabajo de cifrado/descifrado, aunque se introduce un retraso al enviar y recibir los mensajes. Es necesario enviar la información a estos nodos especiales E/D, cifrar la información, intercambiarla entre ellos, descifrar los datos y entregarla al nodo destinatario. Para minimizar este retraso, es mejor usar el cifrado (segundo escenario) sólo cuando sea necesario (por ejemplo, cuando la información enviada sea confidencial, o cuando el canal de comunicación entre 3 y 4 sea altamente insegu-

ro). En otro caso, el encriptado no sería necesario, evitando la sobrecarga introducida y el consiguiente retraso innecesario en el envío de los mensajes.

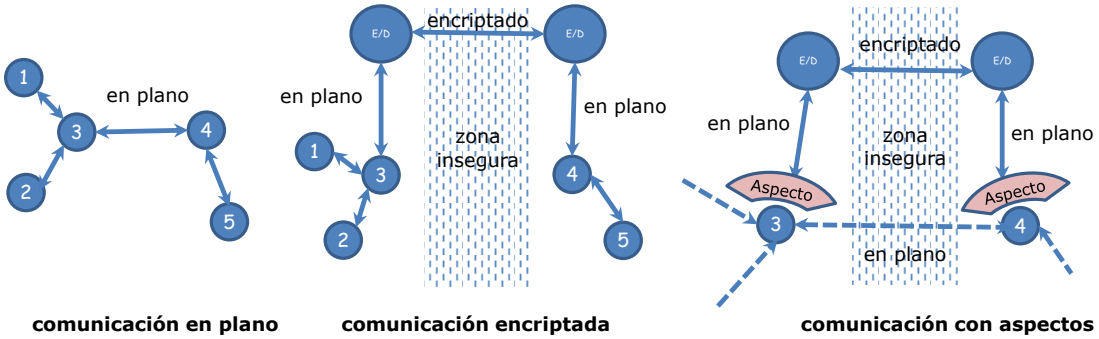


Figura 55: Comunicación con y sin encriptación.

Hemos desarrollado este escenario donde, bajo estas circunstancias detectadas dinámicamente, el sistema es capaz de cambiar las comunicaciones entre dos nodos de un sistema distribuido. Hemos adaptado una aplicación de mensajería instantánea para que envíe o reciba mensajes cifrados dependiendo del nivel de seguridad requerido. Por ejemplo, en una red Wi-Fi pública se activaría el cifrado de los mensajes y en una red privada no. De este modo, cuando la información debe ser cifrada, las transmisiones son enviadas a los nodos que cifran y descifran la información (comunicación encriptada en la Figura 55) y, cuando no es necesario, se utilizará la ruta original (comunicación en plano en la Figura 55). Usando el tejido dinámico de aspectos es posible cambiar dinámicamente el comportamiento de los nodos 3 y 4. Para cambiar los canales de comunicación, el aspecto tejido dinámicamente establecerá una nueva conexión entre el nodo fuente y el nodo E/D, dirigiendo el tráfico a través de esta nueva conexión. Finalmente, cuando el aspecto es destejido, la comunicación original es restablecida.

La conexión con los nodos E/D para cifrar y descifrar toma lugar cuando los aspectos son tejidos. En tiempo de ejecución, los nodos 3 y 4 son conectados cada uno a un nodo E/D que están interconectados entre sí. De este modo, tal y como mostramos en la parte derecha de la Figura 55, se establecerá una nueva conexión segura entre los nodos 3 y 4.

Una vez conectados, los aspectos alteran la funcionalidad de los nodos 3 y 4 con el objetivo de cambiar el flujo de la información. Todo el tráfico es redireccionado a través de la conexión segura, manteniendo a los nodos inconscientes de este hecho. De este modo, cuando el nodo 3 y el 4 intercambian información, ésta irá a través de los correspondientes nodos E/D que cifran y retransmiten la información al otro nodo E/D. Una vez recibida por el

segundo nodo E/D, los datos son decodificados y transmitidos al nodo destino usando un canal seguro donde los datos viajan en plano.

La transmisión cifrada finaliza cuando los aspectos son destejidos. Entonces, la comunicación original es restablecida transparentemente. Cuando los aspectos no interfieren en el acceso de los nodos a su conexión, la información es enviada en plano, como originalmente.

Como discutimos antes, siempre que la conexión es usada para enviar o recibir datos en un entorno inseguro, queremos que un aspecto intervenga y reemplace la conexión por una más segura. Para hacer esto, los mensajes son intercambiados a través de canales de comunicación (*streams*), enviando línea a línea la información de cada mensaje. Las funciones empleadas para recibir la información entre los nodos a través de los *streams*, son los métodos `get` de las propiedades `SwSender` y `SrReceiver`. Estos métodos van a ser tejidos por dos puntos de corte de tipo *Method Call* en el momento de ejecución *around*. El método `ChangeCommunication` de la clase `ChangeCommunicationAspect` es tejido en ambos puntos de enlace. De este modo, en lugar de ejecutar estos métodos, el código de los *advices* es ejecutado cuando el aspecto es tejido.

La Figura 56 muestra el código del aspecto que modifica la conexión. Cuando el aspecto es tejido, una nueva conexión con el nodo E/D es creada (usando una IP y un puerto predefinido) (líneas 8 a 15). Los nodos E/D deben estar fuera de la zona insegura, como mostramos en la Figura 55 (comunicación con aspectos). Estos nodos cifran, descifran e intercambian datos entre ellos a través de la zona insegura. Cuando el método `ChangeCommunication` es invocado, el aspecto verifica cuál de las propiedades está siendo llamada (está información se obtiene del nombre del método tejido que está almacenado en el parámetro `member`) y devuelve el correspondiente canal seguro en cada caso (líneas 33 a 40).

Por lo tanto, si el aspecto es tejido en tiempo de ejecución, el comportamiento de los nodos cambiará dinámicamente y, cuando los métodos `ReceiveMessage` o `SendMessage` sean invocados (y las propiedades son usadas para acceder a los *streams*), el código del aspecto proporcionará una conexión segura de forma transparente. Con esta aproximación, cuando se detecte dinámicamente la necesidad de cifrar las comunicaciones, será suficiente con tejer los aspectos. De forma similar, cuando ya no sea necesaria la comunicación cifrada, se destejerán los aspectos tejidos.

```
01: public class ChangeClientCommunicationAspect {
02:
03:    private static TcpClient tcpServer;
04:    private static StreamWriter swSender;
05:    private static StreamReader srReceiver;
06:    private static bool ACKSended;
07:
08:    private static void Connect(){
09:        String ip = "127.0.0.1";
10:        String port = "5000";
11:        tcpServer = new TcpClient();
12:        tcpServer.Connect(IPAddress.Parse(ip), int.Parse(port));
13:        swSender = new StreamWriter(tcpServer.GetStream());
14:        srReceiver = new StreamReader(tcpServer.GetStream());
15:    }
16:
17:    static public object ChangeCommunication(string member,
18:            object ResultVal, object[] Params, object OBJECT_THIS){
19:        if (!ACKSended) {
20:          Type myType = OBJECT_THIS.GetType();
21:          FieldInfo[] myFields = myType.GetFields(
                        BindingFlags.NonPublic | BindingFlags.Public |
22:                      BindingFlags.Instance);
23:          for (int i = 0; i < myFields.Length; i++){
24:            if (myFields[i].Name.Equals("swSender")) {
25:              StreamWriter originalSwSender =
26:                    (StreamWriter)myFields[i].GetValue(OBJECT_THIS);
27:              originalSwSender.WriteLine("\n");
28:              originalSwSender.Flush();
29:              ACKSended = true;
30:            }
31:          }
32:        }
33:        if (member.Equals("GetSwSender")) {
34:          if (swSender == null) Connect();
35:            return swSender;
36:        }
37:        if (member.Equals("GetSrReceiver")) {
38:          if (swSender == null) Connect();
39:            return srReceiver;
40:          }
41:        return 1;
42:    }
```

Figura 56: Código `ChangeClientCommunicationAspect`.

8.4.2.1 Tiempo de Ejecución

Inicialmente medimos el tiempo de ejecución de las aplicaciones sin usar aspectos (en C# y Java). Posteriormente, implementamos la misma aplicación introduciendo la comunicación segura mediante el tejido dinámico de aspectos. Los tejedores dinámicos evaluados han sido DSAW, JBoss AOP, JAsCo y PROSE. Para JBoss AOP usamos las alternativas de tejido dinámico con las variantes *preCompile* y *loadTime*.

En la Tabla B.10 del Apéndice B se muestran los resultados de las mediciones en milisegundos utilizando la misma metodología [Georges07] que en las secciones previas. Todas las mediciones tienen un error inferior al 2,5% con un intervalo de confianza del 95%, excepto en el caso de PROSE cuyo error es de un 8,18%.

El sistema distribuido medido se divide en 4 componentes: un nodo cliente A, un nodo cliente B y dos nodos intermedios E/D encargados de cifrar y descifrar la información. Se han simulado dos escenarios de comunicaciones entre los dos nodos clientes, A y B:

1. Escenario 1. El nodo cliente B envía 30.000 mensajes dos veces hacia el nodo cliente A. Después se envían 30.000 mensajes del nodo cliente A al nodo cliente B.

2. Escenario 2. Los nodos A y B intercambian 4.000 mensajes en cada dirección, repitiendo esta transmisión 15 veces.

Se ha implementado una aplicación para cada escenario. Cada aplicación ejecuta inicialmente el escenario sin aspectos (la información viaja en plano), después con aspectos tejidos dinámicamente (la información viaja cifrada), y finalmente de nuevo sin aspectos (la información vuelve a viajar en plano). Puesto que medimos la ejecución total de la aplicación, los tiempos incluyen el coste del tejido y destejido dinámico.

La medición inicial de las aplicaciones sin utilizar aspectos (en Java y C#) tienen un rendimiento muy similar. Para el primer escenario, la implementación en C# es un 3,5% más rápida que la realizada en Java; para el segundo escenario, las diferencias son inferiores al 1%, y por cierto no tienen significancia estadística (los intervalos de confianza del 95% se solapan).

La Figura 57 muestra los tiempos de ejecución de los dos escenarios, considerando el tejido dinámico en todas las plataformas orientadas a aspectos. En el gráfico el tiempo de ejecución de las plataformas está dividido por los valores obtenidos por DSAW.

DSAW tiene el mejor rendimiento en ambos escenarios. JAsCo requieren un 20,15% (escenario 1) y un 24,15% (escenario 2) más tiempo de ejecución que DSAW, y el rendimiento de JBoss AOP es de media un 24,43% (escenario 1) y un 34,05% (escenario 2) menor que el DSAW. PROSE es la plataforma más lenta, requiriendo de media un 143% más tiempo de ejecución que el utilizado por DSAW.

Las posibles diferencias entre esta aplicación y los tiempos obtenidos en los dos *benchmarks* sintéticos utilizados son debidas a dos factores. El primero es que en este caso se tiene en cuenta el tiempo de tejido. La técnica de *hotswapping* proporcionada por los Java *agents* proporciona muy buen rendimiento al poder remplazar el código binario de una clase en tiempo de ejecución. No obstante, el procesamiento del binario dinámicamente en el momento de tejido, tiene un coste muy superior al tejido realizado por DSAW (Capítulo 6). El segundo factor es la incorporación de código real a la aplicación. En los *benchmarks* sintéticos es posible que una máquina virtual específica haga optimizaciones al tratarse de un código muy simple, cuando otra máquina virtual puede no implementar las mismas optimizaciones. A la hora de añadir código real, las optimizaciones aplicadas suelen variar y con ellas los tiempos de ejecución obtenidos.

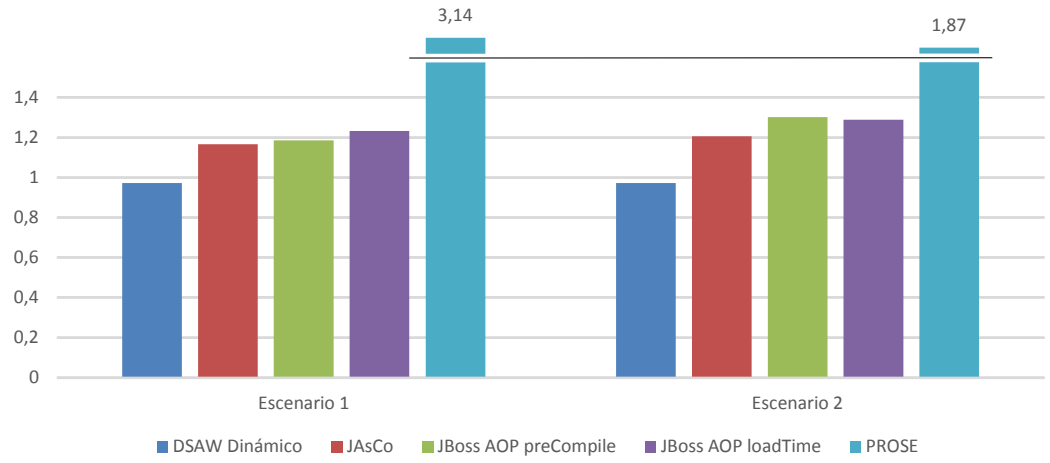


Figura 57: Tiempo de ejecución de Comunicaciones Móviles respecto a DSAW dinámico.

8.4.2.2 Consumo de Memoria

En la Figura 58 mostramos los consumos de memoria de las aplicaciones sobre las distintas plataformas orientadas a aspectos con tejido dinámico (valores relativos a DSAW; valores en la Tabla C.9). En ambos escenarios la plataforma que menos memoria consume es PROSE, algo menos del 50% de la memoria utilizada por DSAW. Vemos, pues, como PROSE tiene un tiempo de ejecución más de 2 veces superior a DSAW, frente al consumo de memoria 1 factor superior en el caso de DSAW. Esto implica que el rendimiento obtenido por DSAW en base a los recursos de memoria utilizados (eficiencia) es el mejor.

JAsCo consume un 38% más memoria que DSAW en ambos escenarios. Comparado con JBoss AOP *preCompile*, DSAW consume al menos 37%

menos memoria. El consumo de la variante *loadTime* de JBoss AOP es casi dos veces superior al de DSAW en ambos escenarios.

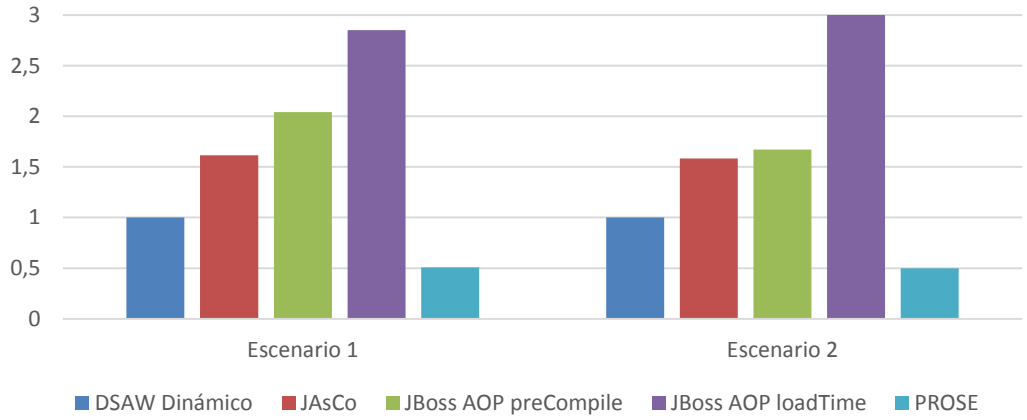


Figura 58: Consumo de memoria de Comunicaciones Móviles respecto a DSAW dinámico.

8.4.3 FTP Cifrado

El cifrado/descifrado de protocolos de comunicación es necesario en diferentes escenarios. En el apartado 8.4.2 hemos visto su aplicación para un caso particular de comunicaciones entre dispositivos móviles. Otro caso donde puede ser necesaria esta funcionalidad es en los servidores de FTP. En estos servidores es común que existan ficheros con información sensible, cuyo contenido hace necesario que se trasmitan de forma cifrada (por ejemplo, datos personales sobre enfermedades infecciosas o afiliación política y sindical). Estos servidores también pueden almacenar otros ficheros que no contengan información sensible. En este caso, los ficheros podrían ser transmitidos en plano, con la correspondiente mejora en el tiempo de transmisión.

Una posible solución sería la utilización de dos servidores FTP distintos, uno que cifrase toda la información que recibiese y enviase, y otro que transmitiese y recibiese toda la información en plano. Esta solución supondría la duplicación de la infraestructura con el consiguiente aumento de los gastos y de la complejidad. Otra alternativa para solucionar este problema podría ser enviar toda la información cifrada, pero esto implicaría malgastar recursos de computación para cifrar/descifrar información que no es confidencial.

La alternativa propuesta en este trabajo es una solución basada en aspectos, similar a la descrita en el punto 8.4.2. Esta solución usa el tejido de aspectos dinámicos orientados a las tareas de cifrado y descifrado de la información. Los ficheros intercambiados entre el cliente y el servidor FTP van

en plano por defecto. En caso de que se necesite intercambiar ficheros con información sensible, se activa el cifrado de la información mediante un comando. Una vez los ficheros con información sensible hayan sido trasmitidos, se desactivaría la comunicación cifrada con la ejecución de otro comando, provocando el destejido de los aspectos.

Para implementar este escenario de tejido dinámico se ha adaptado una aplicación FTP existente [Garcia12], añadiéndole la funcionalidad de cifrar y descifrar ficheros dinámicamente mediante el tejido de aspectos.

8.4.3.1 Tiempo de Ejecución

Hemos empleado el mismo enfoque que anteriormente. Primero hemos realizado un comparación entre dos implementaciones, en C# y en Java, sin utilizar aspectos. Después, hemos usado los mismos tejedores dinámicos que en el punto anterior para añadir la comunicación segura a la aplicación. La Tabla B.11 del Apéndice B detalla los resultados del tiempo de ejecución de las mediciones realizadas, con un error inferior al 3,5%, excepto en el caso de PROSE cuyo error es inferior a 6,9%, y un intervalo de confianza del 95.

El escenario de cifrado dinámico de FTP se compone de un servidor, un cliente y dos componentes encargados de cifrar y descifrar la información. La parte gráfica de la aplicación ha sido eliminada para hacer las mediciones. Se han simulado dos escenarios de comunicaciones entre el nodo cliente y el servidor:

1. Escenario 1. El cliente sube un fichero de 11 megabytes al servidor, y éste responde con una confirmación de fichero recibido. Después el cliente descarga un fichero de 11 megabytes del servidor con la correspondiente confirmación.

2. Escenario 2. El cliente sube 10 ficheros de 11 megabytes al servidor, descargando posteriormente otros 10 ficheros del mismo tamaño. Al igual que en el caso anterior, en ambos casos el servidor y el cliente responden con una confirmación cada vez que se completa la transmisión de un fichero.

La aplicación medida implementa cada escenario enviando inicialmente los ficheros en plano, posteriormente los transmite cifrados y por último los vuelve a enviar en plano de nuevo.

Los tiempos de ejecución de las implementaciones usando Java y C# sin hacer uso de aspectos obtuvieron un rendimiento muy similar. En el caso del primer escenario, la implementación en C# es un 2% más rápida que la realizada en Java, mientras que en el segundo escenario es la implementación Java la que es un 2% más rápida.

La Figura 59 muestra los tiempos de ejecución relativos a DSAW de las plataformas de tejido dinámico para los dos escenarios. En ambos casos el tejedor dinámico de DSAW ofrece el mejor rendimiento. La plataforma con un rendimiento más próximo a DSAW es JAsCo, siendo DSAW un 6,55% y 6,23% más rápido. PROSE muestra el peor rendimiento, siendo DSAW la menos de 3 veces más rápido.

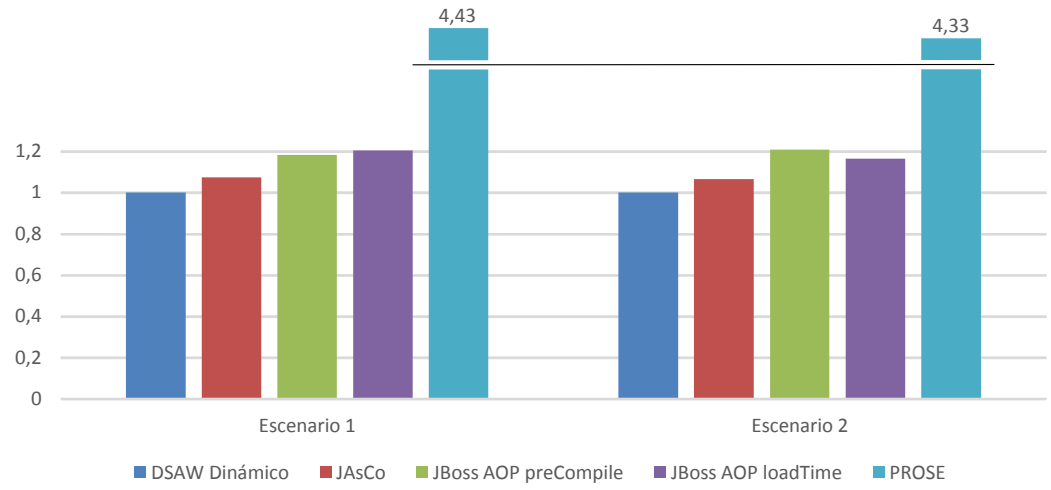


Figura 59: Tiempo de ejecución del FTP cifrado respecto a DSAW Dinámico.

8.4.3.2 Consumo de Memoria

La Figura 60 (Tabla C.10) presenta los consumos de memoria de las distintas plataformas orientadas a aspectos respecto a DSAW. Una vez más, PROSE es el tejedor dinámico que menos memoria consume: el 46% de la consumida por DSAW. DSAW es la segunda plataforma que menos memoria requiere para ejecutar los dos escenarios, seguida de JAsCo que consume un 22,5% más. JBoss AOP consumió un 62,6% más que DSAW.

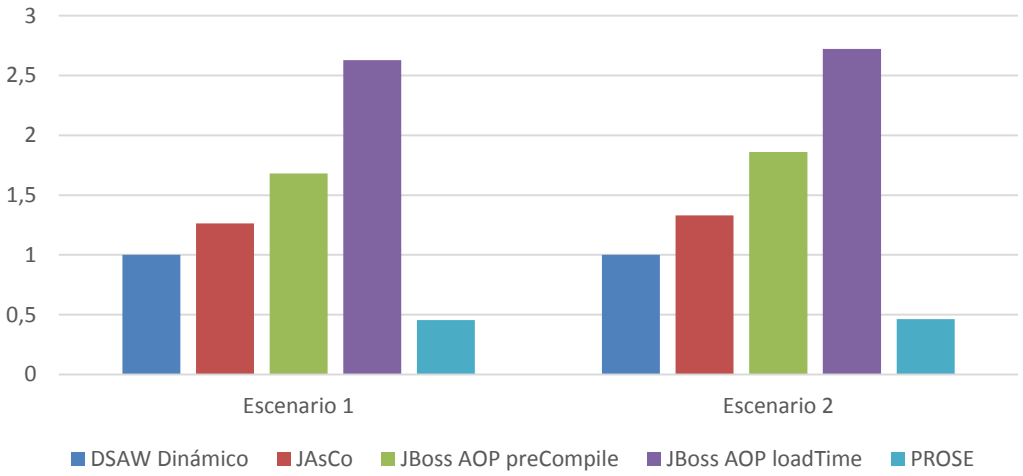


Figura 60: Consumo de memoria del FTP cifrado respecto a DSAW dinámico.

8.4.4 HotDraw

Dentro del campo de la orientación a aspectos, algunos de los primeros artículos realizados acerca de AspectJ muestran ejemplos del uso de este paradigma para mejorar el diseño de editores gráficos [Kizcales01]. Este es un ejemplo de un caso muy ilustrativo de cómo usar la orientación a aspectos, ya que separa el modelo de distintas funcionalidades de la aplicación, incluyendo las posibles vistas, que pueden ser modularizadas mediante aspectos. El ejemplo se basa en el conocido *framework* de dibujo bidimensional para Smalltalk denominado HotDraw [Johnson92].

Esta aplicación ha partido de una implementación realizada para el dibujo de figuras gráficas bidimensionales en Java basada en HotDraw, aplicando patrones de diseño [Lago10]. Esta aplicación permite dibujar figuras (rectángulos, triángulos, círculos…) de diferentes tamaños y colores. Además permite realizar operaciones con las figuras tales como modificar su tamaño, moverlas o borrarlas. En la Figura 61 vemos una captura de la aplicación donde se muestran las operaciones que pueden realizarse con las figuras.

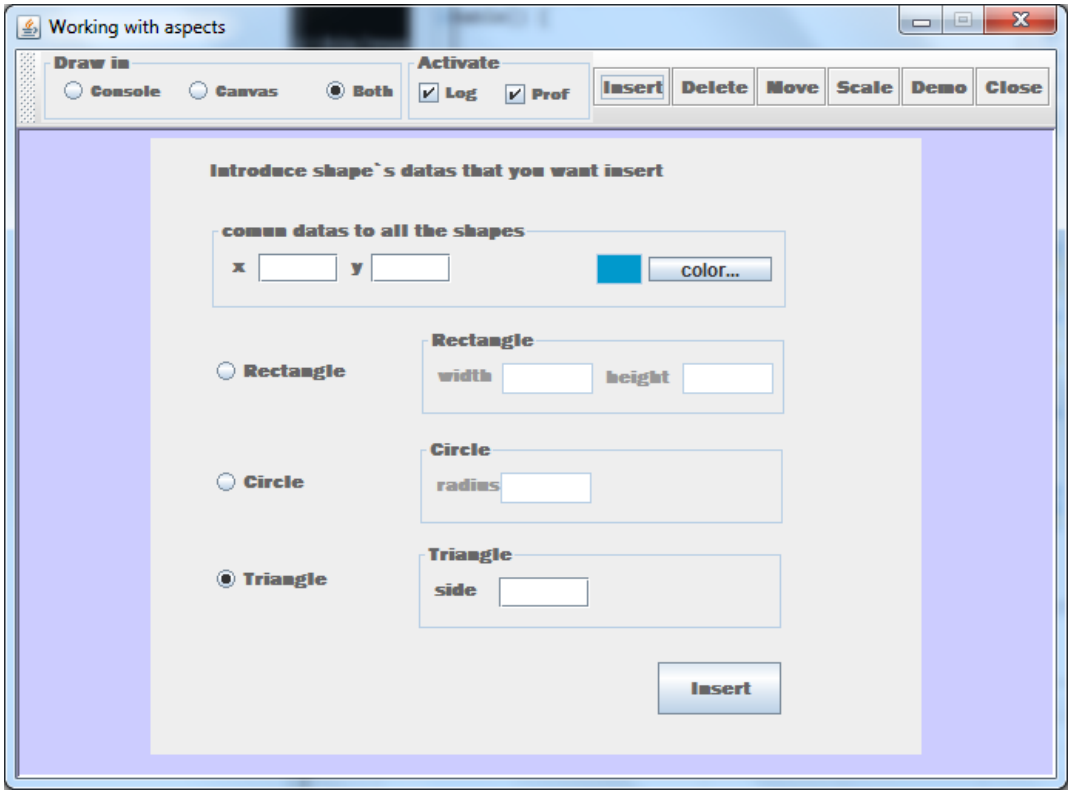


Figura 61: Paleta de controles de la aplicación HotDraw.

Las operaciones se aplican a un lienzo (Figura 62), donde se muestra el resultado gráfico; sobre la consola, donde la información se escribe en pantalla; o en ambos sitios a la vez. Las operaciones son aplicadas a los objetos que representan las figuras, pero su representación está modelada mediante aspectos (lienzo y consola). Los cambios en el modelo implican cambios en las vistas de un modo transparente, gracias a la orientación a aspectos.

Existe la posibilidad de generar un *log* a fichero de las operaciones que se realizan, así como de activar un *profiler* para medir el tiempo de ejecución de cada operación. Estas funcionalidades también han sido implementadas como aspectos.

Se han implementado varios escenarios con distintas secuencias de operaciones que la aplicación carga de forma reflectiva y ejecuta como si un usuario estuviese utilizando la aplicación. Esto permite la automatización de operaciones sin necesidad de que un usuario interactúe con la interfaz de la aplicación, ayudando a realizar las mediciones de rendimiento con un menor sesgo.

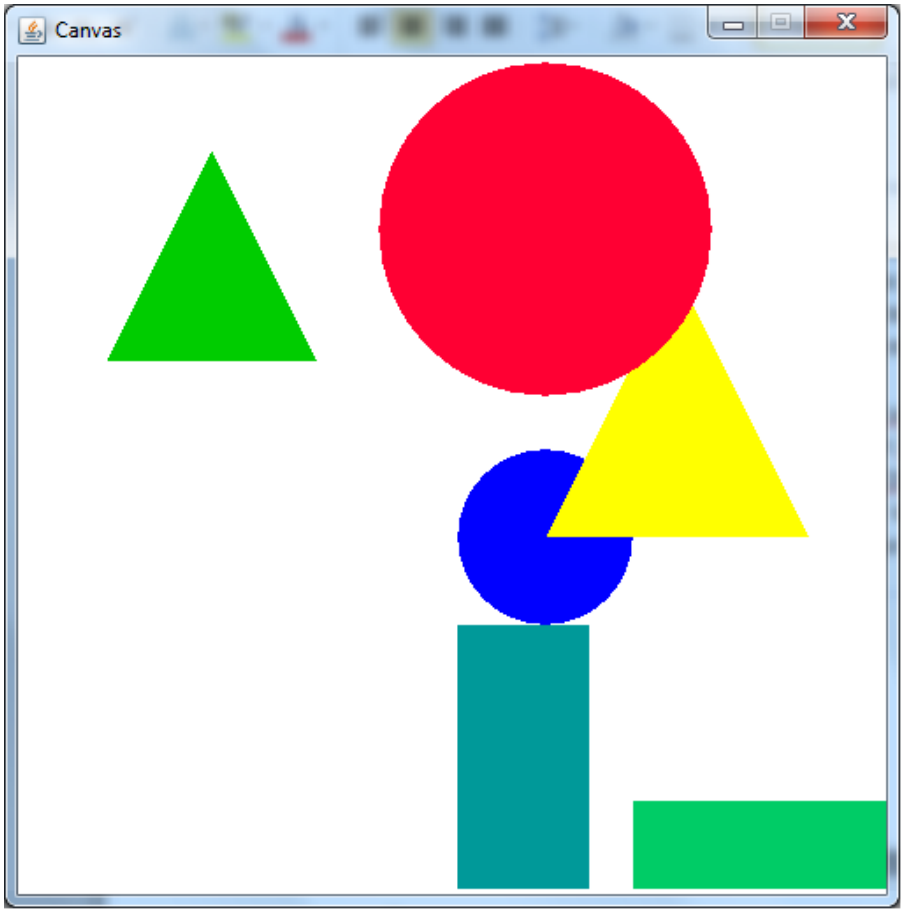


Figura 62: Ventana gráfica de Aplicación HotDraw.

Los métodos del modelo `insert`, `delete`, `move`, `scale`, `moveShape` y `scaleShape` son instrumentados con un punto de corte de tipo *Method Execution* en los momentos de ejecución *before* y *after*. El método `paintIDMS`, que se encarga de dibujar las figuras en el lienzo, es instrumentado con un punto de corte de tipo *Method Execution* en el momento de ejecución *after*. Como los aspectos que se tejen en estos puntos de enlace necesitan acceder a los parámetros del método (para imprimir o pintar el valor de la figura) para DSAW se debe usar el caso *DynamicPart*.

Para la ejecución de los distintos escenarios, también hemos instrumentado los métodos `run` y `execute` con un punto de corte de tipo *Method Execution* en los momentos de ejecución *before* y *after*. Los aspectos que se van a tejer en estos puntos de enlace van a ocuparse de medir el tiempo de ejecución de cada escenario; como no es necesario acceder a la información estática o dinámica del método, hemos usado el caso *WithoutReflection.*

En la Figura 63 se muestra parte del código del aspecto `DrawAspect`. Este aspecto se encarga de insertar, borrar, mover y escalar las figuras en el

lienzo. Todos estos métodos funcionan de la misma forma: primero se obtiene una referencia a la ventana y al lienzo (líneas 4 y 5, y 16 y 17); después se crea la operación a realizar –0 para borrar, 1 para insertar…– (líneas 6 a 8 y 18 a 20); y se toma la figura gráfica (líneas 9 y 21); y finalmente se dibuja en el lienzo (líneas 10 y 22). Las otras dos operaciones para escalar y mover son exactamente iguales cambiando el tipo de operación.

```
01: public class DrawAspect {
02:    static public object drawInsertCanvasDynamicPart(
                    string member,object ResultVal,
03:                 object[] Params, object OBJECT_THIS) {
04:          PanelPrincipal canvas = PanelPrincipal.getInstance();
05:          PaintCanvas.PaintCanvas lienzo = canvas.getPaintCanvas();
06:          Operation o = new Operation();
07:          o.addOperation(1);
08:          lienzo.SetOperation(o);
09:          lienzo.tomaShape((Shape)Params[0], o, null);
10:          lienzo.paint(canvas.GetGraphics());
11:          return ResultVal;
12:    }
13:
14:    static public object drawDeleteCanvasDynamicPart(
                   string member,object ResultVal,
15:                object[] Params, object OBJECT_THIS) {
16:          PanelPrincipal canvas = PanelPrincipal.getInstance();
17:          PaintCanvas.PaintCanvas lienzo = canvas.getPaintCanvas();
18:          Operation o = new Operation();
19:          o.addOperation(2);
20:          lienzo.SetOperation(o);
21:          lienzo.tomaShape((Shape)Params[0], o, null);
22:          lienzo.paint(canvas.GetGraphics());
23:          return ResultVal;
24:    }
25:
26:    …
27: }
```

Figura 63: Código del aspecto `DrawAspect`.

En el caso que la información de la figura sea mostrada por consola, el aspecto es el mostrado en la Figura 64. Se comprueba de qué tipo es la figura (líneas 3, 6 y 9) y se escriben los datos en la consola (líneas 4, 7 y 10).

```
01: static public object drawInsertConsoleDynamicPart(string member,
02:            object ResultVal, object[] Params, object OBJECT_THIS){
03:   if( Params[0].GetType().FullName
                              .Equals("ShapesPackage.RectangleA")){
04:     System.Console.WriteLine("INSERT RECTANGLE with datas {0} ",
05:             utils.datasShow((Shape)Params[0]));
06:   } else if(Params[0].GetType().FullName
                              .Equals("ShapesPackage.CircleA")){
07:     System.Console.WriteLine("INSERT CIRCLE with datas  {0} ",
08:             utils.datasShow((Shape)Params[0]));
09:   } else if (Params[0].GetType().FullName
                              .Equals("ShapesPackage.TriangleA")){
10:     System.Console.WriteLine("INSERT TRIANGLE with datas {0} ",
11:             utils.datasShow((Shape)Params[0]));
12:   }
13: return ResultVal;
14: }
```

Figura 64: Método para insertar figuras por consola.

La utilización de aspectos, como podemos apreciar en la Figura 65, hace que los métodos `paint` y `paintIDMS` sean más simples. Además, permite separar las operaciones de manipulación de las figuras de su representación, tal y como promulga el principio de *Separation of Concerns*.

```
01: public void paintIDMS(Graphics g, int n, Shape shape, Operation o){
02:    if (n!=1){ // delete, move, scale
03:          Int32[] a = logical.calculateRect(shapeDelete);
04:          if ((n == 3) || (n == 4)) // move, scale
05:                logical.changePositionShape(shape);
06:    }
07:    return;
08: }
10:
11: public void paint(Graphics g) {
12:    Object[] ope;
13:    ope = this.operation.getOperation();
14:    n = ((Int32)ope[0]);
15:    if (n!=-1)
16:          paintIDMS(g,n,this.shape,this.operation);
17: }
```

Figura 65: Métodos `paintIDMS` y `paint` de la aplicación.

8.4.4.1 Tiempo de Ejecución

Como en el resto de las aplicaciones reales evaluadas, medimos tanto las implementaciones orientadas a objetos como su homóloga orientada a aspectos. Las plataformas utilizadas que soportan tejido dinámico son DSAW, JAsCo y PROSE. JBoss AOP no ha sido incluido al tener unos tiempos similares en todas las pruebas realizadas en las subsecciones de § 8.4.

La Tabla B.12 del Apéndice B muestra los datos de las mediciones realizadas. El porcentaje de error es inferior al 6% con un intervalo de confianza

del 95%. Para realizar distintas mediciones, hemos implementado varios escenarios que insertan, borran, mueven y escalan objetos. Estas son las operaciones realizas en cada escenario:

- Escenario 1. Creamos 2 rectángulos, 2 triángulos y un círculo. Iteramos 200 veces realizando las siguientes operaciones: mover 3 veces el rectángulo, mover el círculo, mover un triángulo y escalar una figura de cada tipo.

- Escenario 2. Insertamos un rectángulo y un triángulo. Iteramos 500 veces moviendo 2 veces el rectángulo, moviendo 2 veces el triángulo y escalando el triángulo.

- Escenario 3. Iteramos 7 veces insertando un triángulo y un círculo. Dentro de cada iteración, iteramos 100 veces, escalando 2 veces el triángulo y moviendo una vez el círculo. Al finalizar cada iteración borramos las figuras insertadas.

- Escenario 4. Iteramos 60 veces realizando las siguientes operaciones: borramos primero todas las figuras gráficas existentes en el lienzo; insertamos 3 rectángulos, 3 círculos y 4 rectángulos; y por último, escalamos cada uno de los triángulos.

- Escenario 5. Iteramos 90 veces borrando primero todas las figuras gráficas existentes en el lienzo. Después, insertamos un rectángulo, un círculo y 2 triángulos y escalamos todas las figuras insertadas. Posteriormente movemos todas las figuras insertadas dos veces y, por último, borramos un rectángulo.

Hemos medido el tiempo de ejecución de todos los escenarios realizados usando las implementaciones en Java y C# sin la utilización de aspectos. En todos los casos, el rendimiento es similar. La media geométrica de los valores obtenidos muestra que la implementación realizada en C# un 2,3% más rápida que la realizada en Java.

En la Figura 66 mostramos una gráfica con los datos de las mediciones de los cinco escenarios sobre las plataformas orientadas a aspectos (los tiempos de ejecución son relativos a DSAW dinámico). En todos los escenarios DSAW obtiene el mejor rendimiento. De media DSAW es un 18% y un 92% más rápido que JAsCo y PROSE.

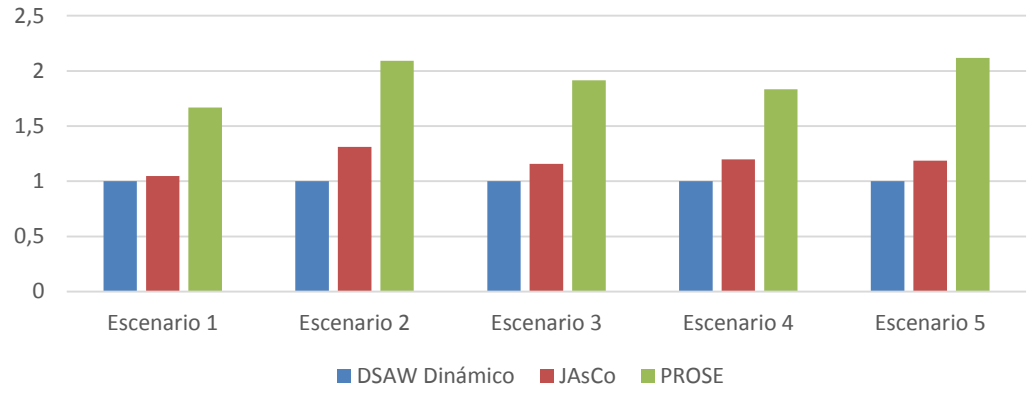


Figura 66: Tiempo de ejecución de HotDraw respecto a DSAW dinámico.

8.4.4.2 Consumo de memoria

Hemos medido los consumos de memoria de los cinco escenarios, detallados en la Tabla C.11 y mostrados en la Figura 67. De media, DSAW consume un 5% más memoria que PROSE y JAsCo 2,7 veces más que nuestra plataforma.

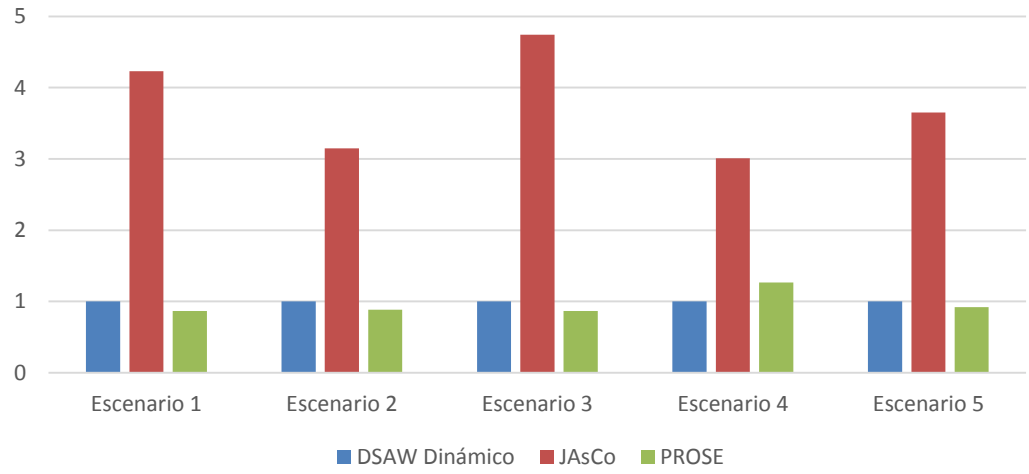


Figura 67: Consumo de memoria de HotDraw respecto a DSAW.

8.5 Coste del Tejido

El paradigma orientado a aspectos ofrece beneficios de reutilización, comprensibilidad, extensibilidad y mantenibilidad. No obstante la intercepción de operaciones por parte del código tejido también puede implicar ciertas penalizaciones en los tiempos de ejecución de las aplicaciones. Algunos estudios muestran que AspectJ añade un sobrecoste máximo en el rendimien-

to de un 20% respecto a implementaciones realizadas sin aspectos [Hilsdade04].

Aprovechando los datos obtenidos en la Sección 8.4, hemos realizado una comparación entre las implementaciones realizadas en Java y C# frente a las correspondientes implementaciones orientadas a aspectos, utilizando la misma metodología de medición [Georges07]. Los valores obtenidos proporcionan una medición del coste del tejido en cada plataforma, para distintos escenarios.

8.5.1 Penalización en el Rendimiento

8.5.1.1 Sistema de Control de Acceso

La aplicación de control de acceso envía mensajes entre nodos tejidos estáticamente con aspectos que etiquetan y cifran la información transmitida. Como el aspecto necesita realizar una operación *proceed*, hemos realizado dos casos: una utilizando reflexión para invocar al método asociado al *proceed*, y otra invocando directamente al método mediante un forzado del tipo. Se miden los tiempos de ejecución de las aplicaciones usando orientación a objetos y orientación a aspectos.

En la Figura 68 mostramos los tiempos de ejecución de cada una de las plataformas orientadas a aspectos divididos por los tiempos de sus versiones orientadas a objetos en C# y Java. El coste de usar aspectos para esta aplicación supone una penalización en el caso de DSAW de un 1,58% para la invocación directa y un 3,33% usando reflexión. En el caso de utilizar AspectJ estas penalizaciones son mayores, pasando a ser de un 5,89% para *proceed* y un 7,53% para la versión que usa reflexión. El coste de las variantes de JBoss AOP varía entre un 58,8% y un 104,6%.

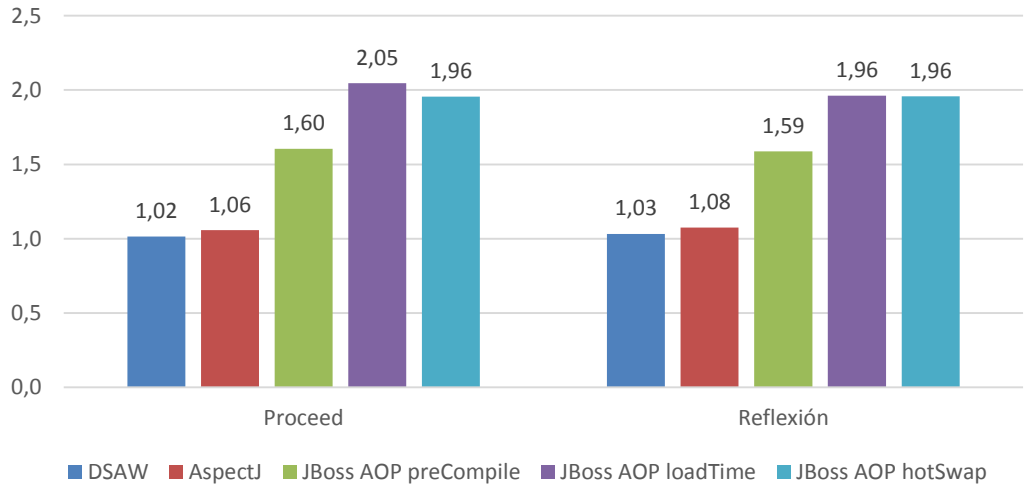


Figura 68: Tiempo de ejecución respecto a la aplicación Control de Acceso sin aspectos.

8.5.1.2 Comunicaciones Móviles Cifradas

La segunda aplicación real medida intercambia mensajes de información entre dos nodos, tejiendo dinámicamente aspectos para realizar una comunicación cifrada. Habíamos identificado dos escenarios distintos de ejemplo. La Figura 69 muestra la relación entre el tiempo de ejecución de las aplicaciones orientadas a aspectos y sus correspondientes implementaciones orientadas a objetos, para los dos escenarios.

La Figura 69 muestra cómo las penalizaciones del tejido dinámico son mayores pero, en este caso, no distan mucho de las del tejido estático. El tejedor dinámico de DSAW es el que muestra la menor penalización de tejido: 6,7% y 5,8%. JAsCo tiene penalizaciones del 23,7% y 14,7%, JBoss AOP del 25,6% hasta el 30,7% y PROSE llega a mostrar una penalización del 224%.

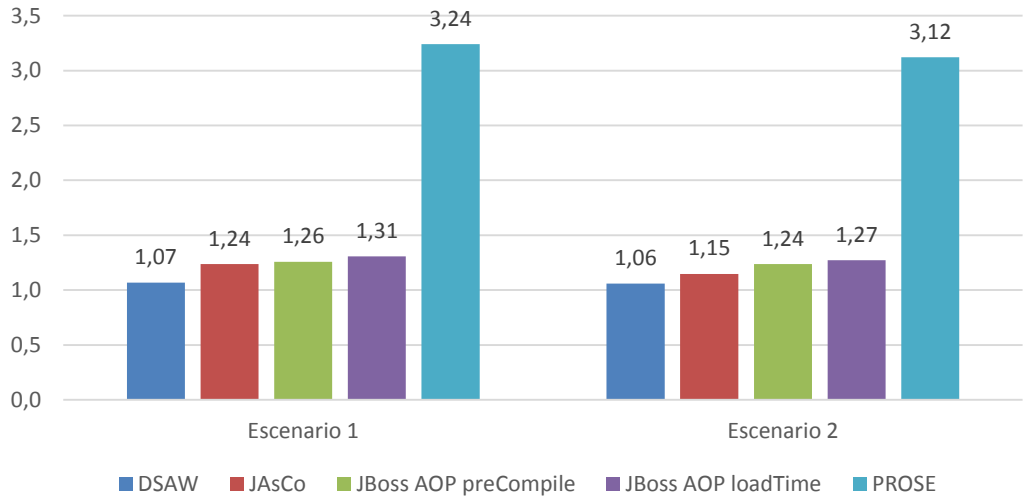


Figura 69: Tiempo de ejecución respecto a la aplicación Comunicaciones Móviles sin aspectos.

8.5.1.3 FTP Cifrado

Esta aplicación toma una aplicación de FTP real y le añade un sistema de cifrado / descifrado dinámico [Garcia12]. Para medir su rendimiento, identificamos dos escenarios distintos en § 8.4.3. La Figura 70 muestra los tiempos de ejecución en relación con la implementación Java o C# correspondiente.

DSAW vuelve a ser la plataforma orientada a aspectos que genera la menor penalización sobre la aplicación original: un 10% para el escenario 1 y un 7,1% para el escenario 2. JAsCo es la segunda plataforma con menor penalización (19,4% y 11,8%), después JBoss AOP *preCompile* (31,7% y 26,7%), JBOSS AOP *loadTime* (34,08% y 22,2%) y, finalmente, PROSE (393,2% y 354%).

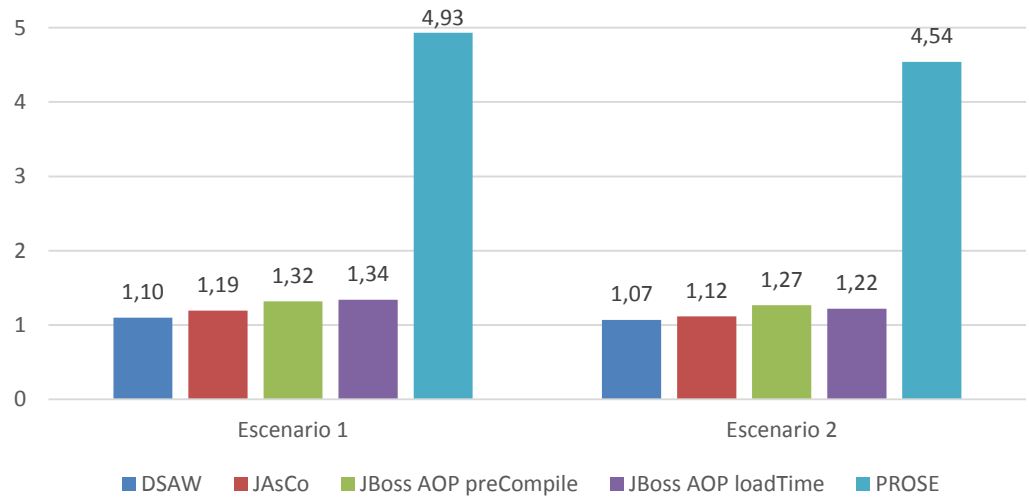


Figura 70: Tiempo de ejecución respecto a la aplicación FTP Cifrado sin aspectos.

8.5.1.4 HotDraw

HotDraw es la última aplicación real evaluada, tratándose de un editor de figuras geométricas bidimensionales. Contempla una serie de operaciones sobre las figuras así como operaciones de *logging* o *profiling*, todas ellas desarrolladas mediante aspectos tejidos dinámicamente. Para realizar las mediciones se han ejecutado 5 escenarios distintos (Sección 8.4.4).

La Figura 71 muestra la relación entre los tiempos de ejecución de las aplicaciones con y sin utilizar aspectos. En este caso, el coste del tejido dinámico es significativamente mayor para todas las plataformas. Esto es debido a que se realizan muchas más operaciones de tejido y destejido en tiempo de ejecución, haciendo que la ejecución de los tejedores dinámicos represente un porcentaje mayor del tiempo de ejecución de toda la aplicación.

DSAW también es la plataforma con menor coste en este caso. De media, posee una penalización del 43,4%, frente a un 72,5% de JAsCo y un 182% de PROSE.

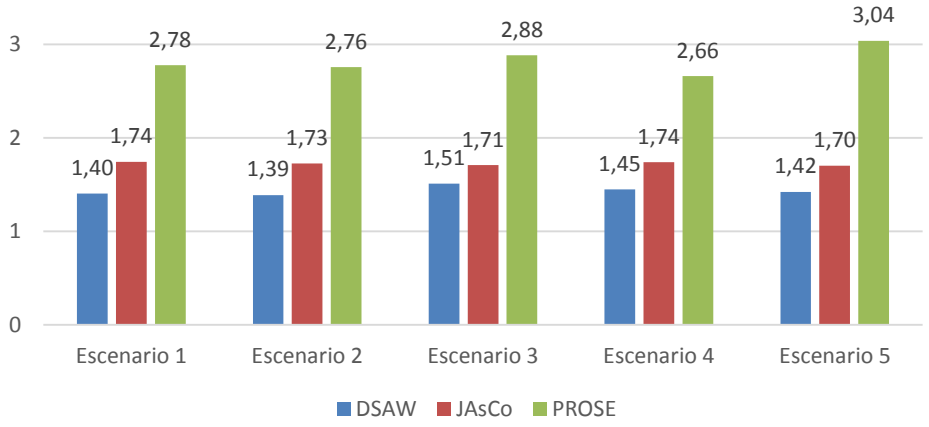


Figura 71: Tiempo de ejecución respecto a la aplicación HotDraw sin aspectos.

8.5.2 Distribución de las Penalizaciones

Tanto el tejido estático como el dinámico suponen un coste adicional en el rendimiento de las aplicaciones. Además si el tejido es dinámico, el coste de éste es mayor, aunque hemos visto cómo depende del número de tejidos y destejidos realizados dinámicamente. En cualquier caso, las penalizaciones de DSAW son siempre las menores, para ambos tipos de tejido. Esto implica que las optimizaciones descritas en el Capítulo 7 han hecho ser a DSAW la implementación más eficiente, relativa a la plataforma de ejecución utilizada (es decir, sin considerar las diferencias entre Java y .NET).

Para el caso específico de DSAW, hemos medido el coste adicional del código inyectado por el JPI. Cuando no hay ningún aspecto tejido dinámicamente, el coste del código de comprobación de los puntos de enlace se ha medido en un 5,94%. El resto de la penalización en el rendimiento es derivada del envío dinámico de mensajes entre los componentes y los aspectos. Este envío dinámico selecciona el *advice* a ser llamado cuando muchos aspectos interceptan el mismo punto de enlace (la función α descrita en § 4.2.2). Recordemos que para realizar la invocación dinámica de un *advice*, DSAW crea un nuevo *stub* dinámicamente usando CodeDOM [Microsoft13]. La invocación de este método fuertemente tipado evita la importante penalización en el rendimiento causada por el uso de reflexión en la plataforma .NET [Ortin09].

8.5.3 Penalización en el Consumo de Memoria

También hemos evaluado las penalizaciones introducidas por cada uno de los tejedores, en relación con la plataforma de programación utilizada

(Java y .NET). La Figura 72 muestra dichas penalizaciones, computando la media geométrica de las cuatro aplicaciones reales descritas en la Sección 8.4.

Analizando los valores de DSAW, se aprecia cómo el tejido dinámico (los tres gráficos más a la derecha en la Figura 72) requiere más memoria que el tejido estático (Control de Acceso). Esto también sucede con JBoss AOP, la otra plataforma que ofrece tejido estático y dinámico.

PROSE obtiene unos resultados sorprendentes, ya que la aplicación tejida consume menos memoria (26% menos) que la aplicación original codificada en Java. Esto es debido a que PROSE se ejecuta haciendo uso del JVMDI/JVMTI que desactiva muchas optimizaciones del compilador JIT de la máquina virtual [OHair04], consumiendo menos memoria.

En el tejido estático, DSAW es la plataforma que menos penalización de memoria introduce (3%), seguida de AspectJ (8%) y JBoss AOP (47%). En el caso del tejido dinámico, el tejedor de DSAW requiere un incremento de memoria del 35%, frente a las penalizaciones del 156% de JAsCo y 302% de JBoss AOP.

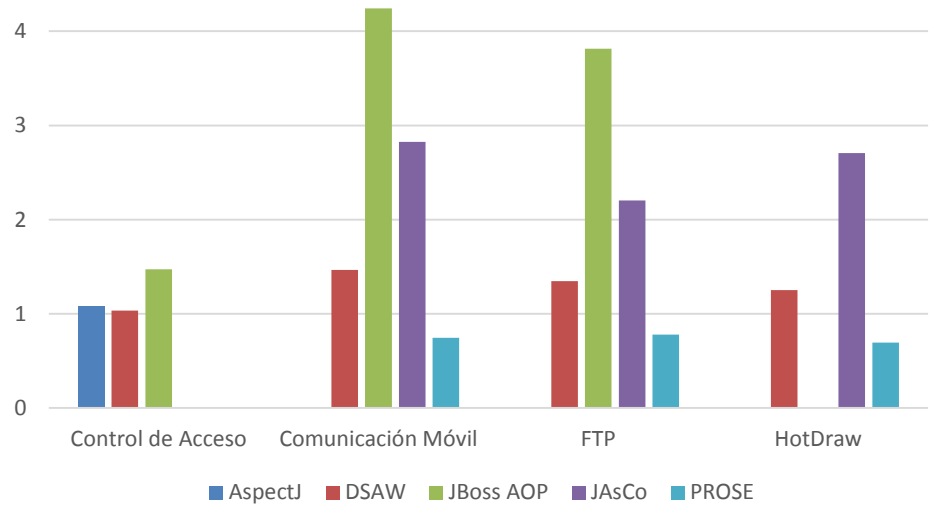


Figura 72: Consumos de memoria de las aplicaciones reales, relativos a la implementación orientada a objetos.

9 CONCLUSIONES

Esta tesis muestra cómo es posible diseñar e implementar una plataforma orientada a aspectos que proporciona tanto tejido estático como dinámico, ofreciendo un rendimiento muy elevado y mostrando la menor penalización de todas las herramientas existentes en la actualidad. Con esta aproximación, se separa el *concern* de dinamismo en el desarrollo orientado a aspectos, de forma que no hay que realizar cambio alguno a una aplicación cuando cambia el tipo de tejido a realizar.

El diseño de DSAW sigue el principio de la *Separation of Concerns*, para que el *concern* del momento de tejido no interfiera en el proceso de desarrollo orientado a aspectos. Los aspectos pueden ser cambiados de estáticos a dinámicos y viceversa dependiendo de los requisitos de una aplicación e incluso de su ciclo de vida. Ambos tipos de tejedores pueden ser usados sobre la misma aplicación simultáneamente. Esto facilita tanto el desarrollo *edit-and-continue* (en los primeros pasos del desarrollo software) como el tejido eficiente estático (cuando la aplicación está a punto de ser desplegada). Por lo tanto, DSAW separa completamente el *concern* del dinamismo del tejido. El programador puede equilibrar las características de flexibilidad y rendimiento de la aplicación durante el ciclo de vida de desarrollo dependiendo de sus necesidades.

La semántica de la plataforma propuesta ha sido formalizada para describir inequívocamente su funcionamiento. Ésta no depende del lenguaje base utilizado (el lenguaje empleado para codificar la aplicación) y permite tejer *advices* en cualquier operación básica de dicho lenguaje base. Esta for-

malización es totalmente novedosa, no existiendo especificación formal de un sistema de tejido mixto en la fecha de escritura de este documento.

Tras la formalización, DSAW fue desarrollada sobre la plataforma .NET, aprovechando las características proporcionadas por esta plataforma. Tanto el tejedor estático como el dinámico fueron implementados como instrumentadores de código a nivel de *byte-code*. De esta forma, DSAW es independiente del lenguaje fuente y del sistema operativo utilizado. Esto permite la adaptación de aplicaciones legadas, incluso si no se dispone de su código fuente, facilitando la reutilización de aspectos y componentes.

El sistema proporciona un amplio conjunto de puntos de enlace, igual para ambos tipos de tejido. En el caso del tejido dinámico, este conjunto supera significativamente al proporcionado por las herramientas existentes. La instrumentación de código también mejora la adaptabilidad de las aplicaciones, permitiendo tejer y destejer aspectos en cualquier punto de ejecución. No es necesario (como sí sucede en otras aproximaciones) prever qué componentes van a ser adaptados. Además, se permite crear un aspecto posteriormente a la ejecución de la aplicación (los componentes están totalmente desacoplados de los aspectos). Los aspectos pueden ser tejidos y destejidos cualquier número de veces en tiempo de ejecución.

Además del tejido estático y dinámico, se proporciona la funcionalidad de tejido dinámico remoto. Éste tiene por escenario de uso la adaptación de aplicaciones distribuidas. Un aspecto, ejecutándose en una máquina, podrá adaptar un componente que esté corriendo en un ordenador diferente. El caso de que esta adaptación remota no sea requerida, DSAW realiza múltiples optimizaciones para mejorar su rendimiento.

Para conseguir un rendimiento elevado con un consumo de memoria bajo, hemos realizado una serie de optimizaciones en nuestra plataforma. La incorporación de un tejedor estático que emplea las mismas técnicas de tejido que AspectJ fue una primera optimización. Posteriormente definimos un mecanismo de especialización de la información a enviar a los *advices*, utilizando cuatro signaturas distintas en función de la información requerida por el aspecto; a menor información, mayor rendimiento. Otra optimización significativa fue la eliminación del uso de reflexión a la hora de conectar los componentes y con los aspectos. Utilizando generación dinámica de código, el tejedor dinámico es capaz de generar *stubs* en tiempo de ejecución que hagan de *dispatchers* sin el uso de reflexión. La siguiente optimización consistió en mejorar el rendimiento del código inyectado mediante la indexación de los

puntos de enlace. Se desarrolló un sistema de indexación de modo que, a partir de un índice entero, se podría acceder a un punto de enlace de un programa, conociendo la lista de aspectos tejidos. Finalmente, se realizó una optimización del código *stub* generado, especializando el código para cada punto de enlace interceptado, eliminando la necesidad de implementar un *dispatcher*.

Todas estas optimizaciones han conseguido hacer que DSAW sea una plataforma muy eficiente. Para poder contrastar esta hipótesis, hemos hecho una evaluación exhaustiva de las plataformas orientadas a aspectos que proporcionan un mejor rendimiento. DSAW ha sido el sistema con menor penalización de rendimiento, tanto en tejido estático como en dinámico. Esto indica que los tejedores desarrollados ofrecen el mayor nivel de optimización. Además, ha sido la plataforma que menos coste de memoria ha mostrado, tras la plataforma con peor rendimiento.

La comparación de rendimiento absoluto con el resto de plataformas también ha mostrado el beneficio de las optimizaciones implementadas. En las mediciones de aplicaciones reales, nuestro sistema ha obtenido el mejor rendimiento para tejido estático y dinámico. DSAW estático ha sido un 3,3% más rápido que AspectJ. DSAW dinámico ha mostrado un beneficio medio de un 12,5% frente a JAsCo. AspectJ y JAsCo son los tejedores (estático y dinámico, respectivamente) más rápidos evaluados hasta la aparición de DSAW. Este rendimiento es ofrecido con un menor consumo de memoria que cualquier plataforma, a excepción de PROSE que obtiene un rendimiento muy pobre a costa de un consumo de memoria muy bajo.

9.1 Trabajo Futuro

El trabajo futuro estará enfocado en dos direcciones. Por un lado, a investigar en mejorar las capacidades de la plataforma. Por el otro, a aplicar la plataforma desarrollada en otras áreas de investigación de la Ingeniería Software.

Los objetivos principales que nos hemos marcado para mejorar las capacidades de la plataforma son tres: mejorar el rendimiento de la plataforma, diseño de un nuevo mecanismo de resolución de conflictos y facilitar las labores de los programadores de la plataforma. Para mejorar el rendimiento de la plataforma, aunque la arquitectura .NET no permite la modificación de los *byte-codes* en tiempo de ejecución, consideramos que pueden existir algunas líneas aún por investigar. La primera es la generación de interfaces `IExecu-`

`te` específicos a cada punto de enlace. Así, el número y tipo de parámetros de su método `Execute` variará en función del punto de enlace.

Otra posibilidad de optimización a investigar es la utilización de un mecanismo de caché para los *call-site* de los puntos de enlace, similar al proporcionado por el DLR [Chiles13]. Se trata de compensar la carencia de la existencia de una herramienta como Javassist en C# con la implementación de una caché en las comprobaciones de los puntos de enlace. Esto permitiría eliminar la necesidad de repetir continuamente las operaciones para comprobar si sobre ese punto de enlace se han tejido aspectos dinámicamente.

El siguiente punto será el diseño de un mecanismo de resolución de conflictos. Actualmente nuestro mecanismo, la implementación de la función formalizada como α, se basa en cuatro variables, pero es simplista. Nuestra idea es aplicar soluciones más elaboradas como introducir métodos para decidir la prioridad [Suvée18], usar una aproximación de *stateful aspects* para tener en cuenta la historia de computación [Douence04], o incluir técnicas semánticas de resolución de conflictos [Durr05].

Por último, han surgido discusiones acerca del uso de la POA en relación con la complejidad derivada de la modularización de *concerns* entremezclados. Los nuevos elementos de programación, la nueva sintaxis del lenguaje, la complejidad inherente de estos nuevos elementos de programación y la dificultad para depurar las aplicaciones orientadas a aspectos [Eaddy07] han levantado dudas acerca de si esta complejidad adicional merecen la pena para los beneficios obtenidos [Steimann06]. Esto ha provocado que el impacto del paradigma orientado a aspectos no haya sido tan significativo como inicialmente se esperaba [Popovici01a]. Para facilitar el trabajo de los desarrolladores, hemos desarrollado un *plug-in* para trabajar con DSAW en Visual Studio (Apéndice A) [Perez10]. Estamos convencidos que es necesario ampliar este componente para permitir operaciones más complejas como proporcionar soporte para la interacción del código de los componentes y aspectos de la aplicación, abstraer por completo al usuario del empleo de archivos de definición de puntos de corte y facilitar la depuración de código.

Actualmente, el paradigma de la orientación de aspectos se está aplicando en varias áreas de investigación en el desarrollo de software. Los Lenguajes de Dominio Específico (*Domain-Specific Languages, DSL*) son lenguajes particularmente expresivos en ciertos dominios de problemas. Algunas veces la definición de un DSL puede poseer una naturaleza orientada a aspectos [Strembeck06]. Consideramos que las características de DSAW pueden hacer-

lo útil en este ámbito de investigación, para poder desarrollar DSL sobre esta plataforma.

Otra área de investigación donde planteamos aplicar DSAW es en programación reactiva (*reactive programming*), basada en la ejecución de reacciones ante cambios en entidades de un modelo [Reactivity13]. La reactividad se debe expresar propiamente mediante abstracciones adecuadas del lenguaje de programación. El código reactivo debe ser modular y extensible y fácil de extender y de analizar. El tejido y destejido dinámico puede representar fácilmente estas reacciones mediante aspectos que, tejidos dinámicamente, representarán la reacción a los cambios en las entidades del problema, modeladas mediante componentes. De hecho, ya hay investigadores que han propuesto varias soluciones para llevar a cabo estas cuestiones, incluyendo entre ellas POA [Salvaneschi13].

La versión actual de DSAW, su código fuente, y todos los *benchmarks* y ejemplos presentados en esta Tesis Doctoral están disponibles para su descarga en http://www.reflection.uniovi.es/dsaw.

Apéndice A. DSAW Development Tools

En general, la separación de incumbencias usando aspectos mejora la legibilidad del código, facilita la depuración y hace que la aplicación sea más flexible, reutilizable y mantenible. Sin embargo, la necesidad de especificar en las aplicaciones, que hacen uso de POA, los puntos de enlace en los que se debe producir la adaptación mediante un aspecto puede llegar a complicar en gran medida la tarea de desarrollo, siendo difícil para el programador conocer las relaciones existentes entre los diferentes elementos de la aplicación. Por ello, el empleo de herramientas de ayuda al desarrollo cobra especial importancia en el DSOA.

Al amparo de esta tesis doctoral, se ha desarrollado un proyecto [Perez10] que resuelve la integración entre un Sistema para el Desarrollo Orientado a Aspectos y un Entorno de Desarrollo Integrado. En el proyecto *DSAW Development Tools* se implementó un *plug-in* para la integración de DSAW con el entorno de desarrollo Microsoft Visual Studio 2008.

El componente desarrollado permite: la administración de proyectos; la visualización de los elementos de una aplicación, así como las relaciones existentes entre ellos; el soporte para la edición textual y gráfica de los archivos que forman parte de una aplicación desarrollada sobre DSAW; la integración del proceso de generación y ejecución de aplicaciones y la creación de un componente de instalación para facilitar el despliegue del programa en la máquina del usuario.

A.1. Antecedentes

Desde el punto de vista de las herramientas, en la actualidad, el uso de Entornos de Desarrollo Integrado *(Integrated Development Environment* IDE) se ha extendido como una práctica habitual en cualquier proceso de desarrollo y, como consecuencia, el mercado de este tipo de productos se ha ampliado, ofreciendo soluciones cada vez más completas. Lo que anteriormente se limitaba a la edición de código, ahora pretende abarcar con un simple programa todas las actividades habituales de un proyecto de software, una realidad más cercana a la de una Herramienta CASE. En ese sentido, incluso resulta habitual la disponibilidad de soporte para que cada usuario o desarrollador pueda extender el entorno, creando herramientas personalizadas que le ayuden en el desempeño de su trabajo.

Algunos sistemas orientados a aspectos de amplio uso hoy en día como AspectJ y JBoss AOP ya cuentan con *plug-ins* para entornos de desarrollo integrados como Eclipse [Eclipse13][RedHat13]. También algunas plataformas desarrolladas en el mundo académico han desarrollado este tipo de integraciones como JAsCo [JAsCo13] y PROSE [Nicoara13].

A.1.1. AspectJ Development Tools

AspectJ es probablemente la plataforma DSOA más utilizada en la actualidad. Fue desarrollado por la compañía Xerox PAC. Actualmente forma parte de los proyectos de código libre de Eclipse Foundation.

Al contrario que DSAW, AspectJ utiliza recursos lingüísticos explícitos, ampliando la gramática del lenguaje de programación Java, para la definición de aspectos. Sin embargo, se ha convertido prácticamente en un estándar POA, destacando especialmente por su sencillez y usabilidad de cara al usuario final. Gran parte de su éxito se debe a la existencia de AspectJ Development Tools (AJDT), un proyecto de integración que, desde sus comienzos, ha dado soporte para el empleo de la plataforma en diversos IDE (Eclipse, JBuilder, JDeveloper y NetBeans).

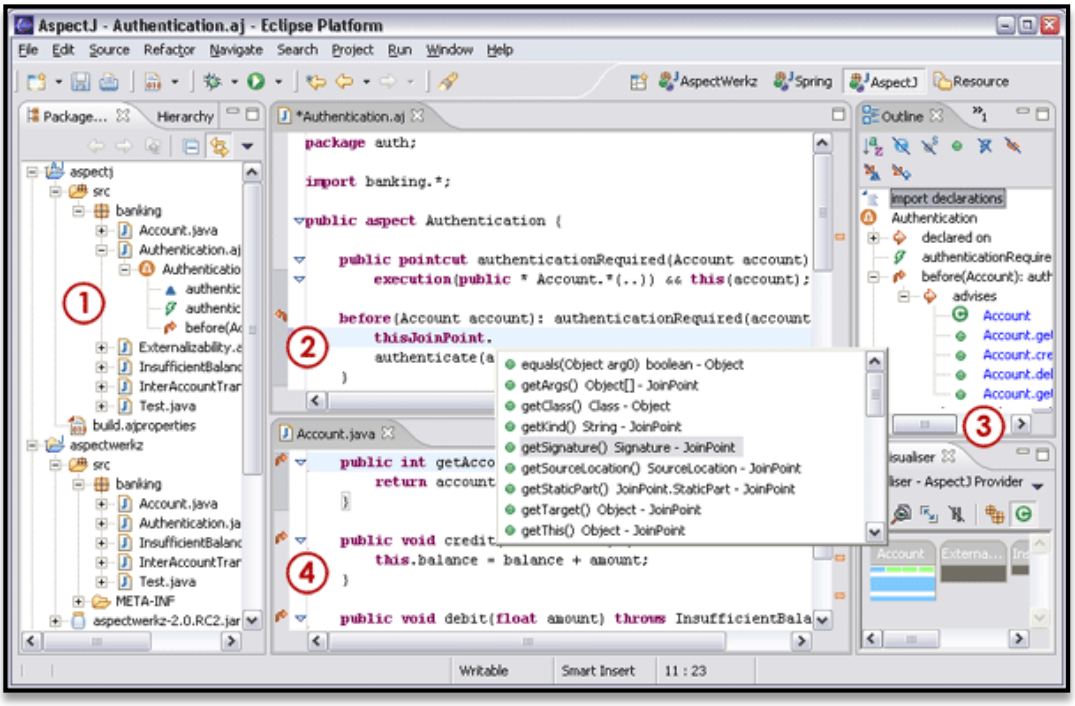


Figura 73: AspectJ Development Tools *plug-in* para Eclipse.

En la Figura 73 vemos el aspecto gráfico de las funcionalidades de este proyecto. Estas funcionalidades se dividen en:

1. Explorador de Paquetes. Muestra los aspectos definidos en la aplicación como una entidad individual. De igual forma, proporciona información acerca de los puntos en los que se produce la inserción de cada aspecto y especifica su implementación.

2. Editor de Aspectos. Proporciona anotaciones que permiten la navegación entre los diferentes elementos de la aplicación. Además, la ayuda contextual permite mostrar las partes del código a las que un aspecto puede adaptar, dependiendo de sus características y de la situación puntual.

3. Esquema del Documento. Muestra la estructura de relaciones cruzadas del editor activo, indicando de forma concreta los puntos de la aplicación que se ven afectados por el aspecto que está siendo editado. Del mismo modo, se muestran las relaciones en el sentido inverso cuando se lleva a cabo la edición de un miembro adaptado.

4. Editor de Código Ampliado. El editor de código habitual también proporciona anotaciones, que permiten la navegación ha-

cia la declaración de los aspectos por los que un miembro está siendo adaptado.

A.1.2. JBoss AOP Tools

JBoss AOP fue lanzado por primera vez en el año 2004 como un añadido para el servidor de aplicaciones JBoss. Al igual que DSAW, la definición de aspectos se lleva a cabo mediante el uso de documentos XML, empleando técnicas reflectivas para añadir la implementación de las incumbencias transversales a la aplicación final.

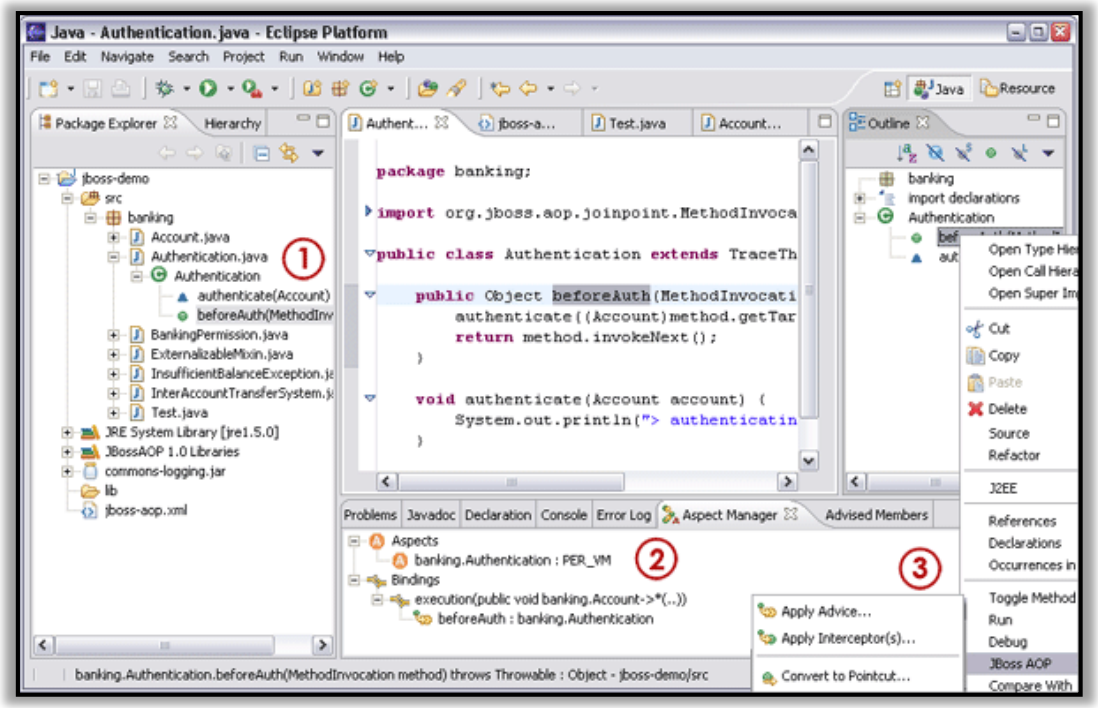


Figura 74: .JBoss AOP Tools *plug-in* para Eclipse.

En la Figura 74 mostramos las funcionalidades de este proyecto. Estas funcionalidades se dividen en:

1. Explorador de Paquetes. No muestra los aspectos definidos en la aplicación como una entidad individual si no que toman la forma de miembros normales del lenguaje.

2. Gestor de Aspectos. Es una vista gráfica de los ficheros XML que se emplean para definir los diferentes elementos de la aplicación. Esto permite abstraer al usuario de la engorrosa edición

manual de los documentos. Además muestra de manera precisa las relaciones cruzadas de cada uno de los aspectos.

3. Nuevos Comandos. Se añaden una serie de comandos al menú contextual del editor de código que permiten insertar un aspecto en un punto de la aplicación o designar su implementación real, sin que sea necesario la edición manual de los ficheros de definición.

A.2. Microsoft Visual Studio

Visual Studio es el IDE de Microsoft. Como tal, es uno de entornos de desarrollo más utilizados a nivel mundial, especialmente en el desarrollo de aplicaciones para el sistema operativo Windows y/o para el Framework .NET.

Ofrece todas las características habituales de esta clase de entornos: editores de código, depurador integrado y herramientas para la construcción de Interfaces Gráficas de Usuario (GUI). Soporta gran cantidad de lenguajes, tales como Visual C++, Visual C#, Visual J#, ASP.NET y Visual Basic .NET. Adicionalmente, se han desarrollado gran cantidad de paquetes de integración para dar soporte para muchos otros lenguajes de programación como Python, o StaDyn[3] [Ortin2014b] [Garcia13].

Visual Studio ofrece tres formas de extender el entorno: a través de macros, utilizando *add-ins* o mediante el empleo de un SDK. Cada uno de estos métodos está orientado hacia un propósito concreto. Las macros sirven para automatizar tareas simples, los *add-ins* permiten la inclusión de nuevos comandos o elementos de interfaz, etc. Pero las tareas más complejas de integración requieren la creación de componentes denominados paquetes, por medio de una serie de servicios que proporciona la plataforma Visual Studio SDK [Microsoft13c].

[3] Lenguaje Estático y Dinámico desarrollado en la Universidad de Oviedo por *Reflection Research Group*.

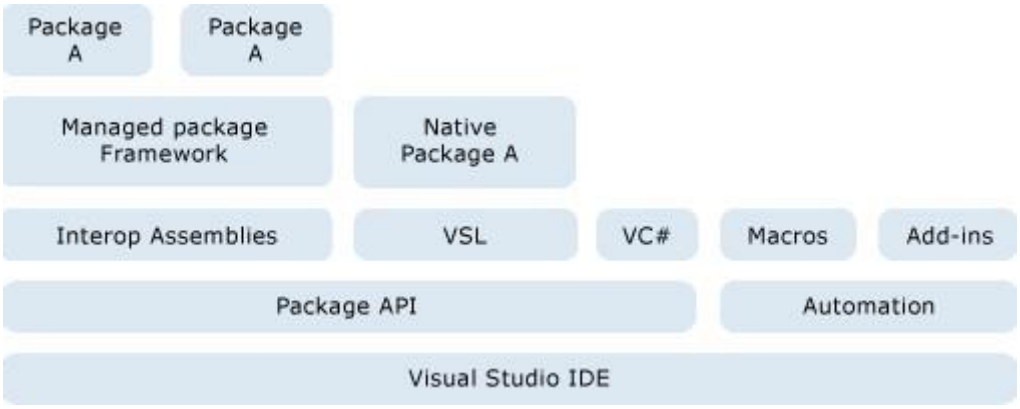


Figura 75: Mapa de la Arquitectura del Visual Studio SDK.

En la Figura 75 se muestra una visión esquematizada de la arquitectura del SDK de Visual Studio. El objetivo es comprender que elementos son necesarios para la integración en este IDE y como han de interaccionar las aplicaciones que la facilitan [Microsoft13d].

Visual Studio está formado por componentes software llamados *VSPackages*. Estos paquetes integran herramientas, editores, servicios, tipos de proyecto, depuradores, etc. con un propósito común. Utilizando el SDK para crear un nuevo *VSPackage*, es posible incluir nuevas funcionalidades para su uso propio o distribución.

El *Managed Package Framework* es una librería que proporciona un conjunto de clases para que las extensiones implementadas en código administrado .NET sean interoperables con el núcleo (*Shell*) de Visual Studio, que está desarrollado en lenguaje nativo (C y C++) [Microsoft13d]. En general, la mayor parte de las tareas soportadas por Visual Studio pueden ser implementadas mediante la redefinición y extensión de la jerarquía de clases del MPF, facilitando de esta forma el proceso de integración.

El desarrollo de paquetes de integración en Visual Studio es una tarea que puede complicarse debido a su interacción con el Registro de Windows. Por ello, la plataforma SDK hace uso en depuración de una clave independiente denominada *Experimental Instance* [Microsoft13d].

La forma de extender el conjunto de comandos en el SDK es a través de la modificación de la tabla VSCT (*Visual Studio Command Table*). Se trata de un archivo XML, único para cada paquete, en el que se definen sus comandos, así como los grupos de menú y menús en los que estos se sitúan. De esta forma, es posible separar la apariencia y situación de los elementos de la in-

terfaz, permitiendo la ubicación de la misma función abstracta en diversos contextos dentro del entorno de desarrollo.

A.3. Desarrollo del Plug-in

El *plug-i*n ha sido implementado usando el lenguaje de programación Visual C# 3.0. A nivel de análisis y diseño, el proyecto ha sido dividido en dos componentes que agrupan funcionalidades comunes. Por una parte, el Modelo, que proporciona las entidades representativas y emula el funcionamiento del sistema real (tejido de aspectos), para que sea posible conocer las relaciones existentes en la aplicación en desarrollo. Por otra parte, la Integración, que proporciona la Interfaz Gráfica de Usuario y funciona como intermediaria con la plataforma de extensibilidad de Visual Studio, para que sea posible extender el entorno de trabajo con nuevas herramientas.

Un requisito básico en el análisis y el diseño, ha sido aislar por completo el Modelo de la aplicación, de manera que es posible su reutilización en la integración del sistema en cualquier otro entorno de desarrollo, o incluso facilitar los mismos servicios para la integración de otras plataformas de similares características, como vemos en la Figura 76.

Figura 76: Esquema del Diseño.

Considerando la complejidad de la plataforma DSAW, se ha obviado el funcionamiento interno de la plataforma (modelo de Caja Negra), únicamente interactuando con ésta en el momento de la generación y ejecución, cuando es necesario llevar a cabo la inyección del código de los aspectos de la aplicación.

A.3.1. Características del Modelo

El Modelo proporciona todas las funcionalidades necesarias para simular el funcionamiento de la plataforma DSAW. Esto requiere, fundamentalmente, mantener una representación de los elementos que forman parte de la aplicación (clases, métodos, campos, etc.). Dado que DSAW funciona a nivel de ensamblado .NET para proporcionar independencia con respecto al lenguaje y la plataforma, es necesario utilizar técnicas de reflexión para obtener esta información.

Adicionalmente, se encarga de analizar los archivos que determinan los aspectos existentes y los puntos de la aplicación en los que se inserta cada uno de ellos, creando una representación básica del conocimiento en forma de restricciones anidadas, que permita posteriormente determinar cuáles de los elementos del programa se ven afectados por cada aspecto. Para conocer las relaciones existentes entre estos elementos, es necesario simular el tejido que se lleva a cabo en la plataforma real, utilizando las condiciones definidas anteriormente.

A.3.2. Características de la Integración

La Integración proporciona todas las funcionalidades necesarias para interactuar con la plataforma Visual Studio SDK, para que sea posible extender el entorno. Además de la implementación de los módulos que requiere el *Framework* para ser extendido, es necesaria la creación de las interfaces gráficas de usuario que representan las diferentes herramientas que se pretenden proporcionar.

A.4. Entorno Desarrollado

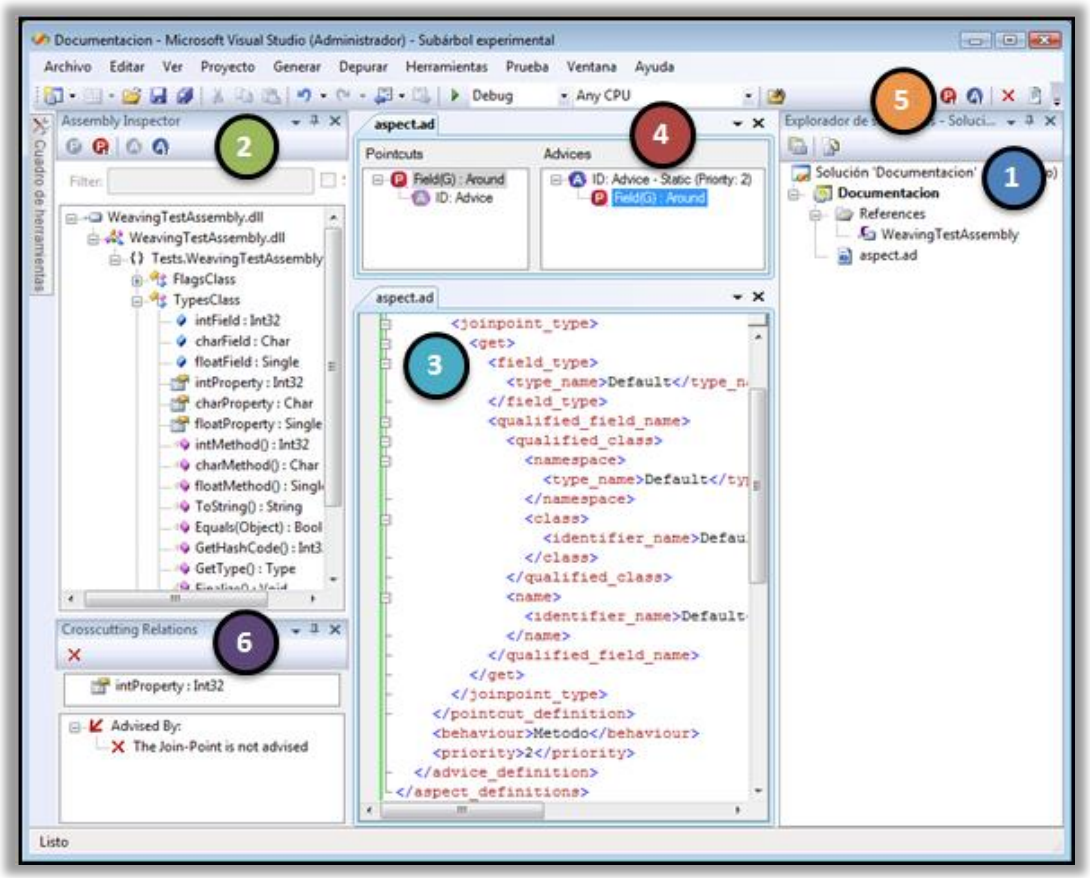


Figura 77: Entorno de trabajo DSAW Development Tools.

El entorno de trabajo resultante de la integración de todas las herramientas es el mostrado en la Figura 77. A continuación se describen de forma superficial los principales componentes que lo forman y su propósito dentro de la aplicación:

1. Explorador de Soluciones. Se ha extendido la herramienta que proporciona Visual Studio para la administración de los ítems de un proyecto (en este caso particular, archivos de definición de aspectos y referencias a ensamblados), proporcionando las funcionalidades adicionales propias de este escenario.

2. Inspector de Ensamblados. Se ha creado una herramienta específica cuyo propósito es visualizar la representación de los elementos de la aplicación que almacena el modelo, es decir, muestra los módulos, clases, enumeraciones, métodos, campos, propiedades, etc. que forman parte de un determinado ensamblado, según el formato habitual del Explorador de Objetos.

3. Editor Textual de Aspectos. Se ha extendido el Editor XML que proporciona Visual Studio para facilitar funciones como el auto-completado, la sintaxis coloreada, y la ayuda contextual, en función del formato específico de los archivos empleado en DSAW. Su contenido se encuentra sincronizado con la vista gráfica del documento.

4. Editor Gráfico de Aspectos. Se ha desarrollado una herramienta específica que muestra una representación gráfica de los elementos definidos en un archivo de definición de aspectos, abstrayendo al usuario de su representación textual y permitiendo su modificación mediante diferentes tipos de interacciones: botones, *Drag and Drop*, etc. Su contenido se encuentra sincronizado con la vista textual del documento.

5. Barra de Herramientas. Se ha creado una barra de herramientas que agrupe los comandos correspondientes a las funciones específicas de este paquete de integración.

6. Vista de Relaciones Cruzadas. Se ha desarrollado una herramienta específica que visualiza las relaciones cruzadas de cualquier elemento seleccionado, tanto en el Inspector de Ensamblados como en el Editor Gráfico, esto es, los aspectos insertados en cada punto, los elementos a los que afecta una determinada restricción, etc.

Adicionalmente, también se han desarrollado ventanas de diálogo que se muestran convenientemente cuando es necesario editar características concretas de algunos elementos, como la posibilidad de crear estructuras de condiciones complejas (Figura 78).

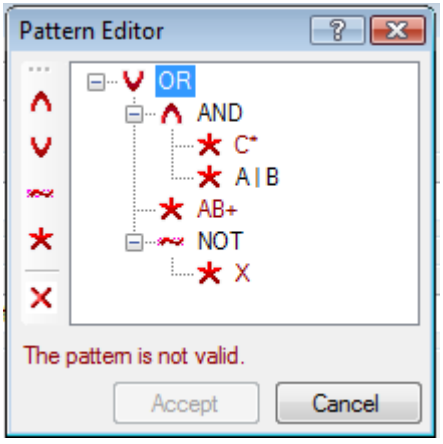


Figura 78: Editor de condiciones.

Apéndice B. TIEMPOS DE EJECUCIÓN

B.1. Micro-Benchmark

		AspectJ		DSAW Estático		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap		Gripper-Loom 4.2 (RC1)		Spring Java 3.1.3		Spring .Net 1.3.2	
Before	WR	437,00	±0,00	464,55	±1,96	3.872,75	±1,71	4.187,46	±1,85	4.060,59	±1,96	4.116,24	±1,70	34.552,33	±1,70	32.882,11	±1,95
	SP	2.366,00	±1,97	786,90	±1,98	66.705,66	±1,00	69.168,50	±0,87	66.669,33	±1,18	52.592,99	±0,47	34.638,28	±1,99	35.651,33	±0,70
	DP	26.934,99	±1,65	37.358,50	±1,13	79.691,19	±1,54	80.102,35	±1,92	79.131,99	±1,82	37.191,50	±0,02	111.097,74	±1,73	67.370,00	±1,20
	Full	28.649,24	±1,97	37.697,49	±0,50	93.916,33	±1,90	92.820,33	±1,57	91.852,66	±1,17	90.833,33	±1,16	112.878,43	±2,02	70.479,87	±1,90
After	WR	798,80	±1,90	478,11	±1,97	3.889,10	±1,93	4.022,74	±1,95	4.028,49	±1,33	4.002,33	±1,55	34.539,00	±0,63	33.174,89	±2,07

		AspectJ		DSAW Estático		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap		Gripper-Loom 4.2 (RC1)		Spring Java 3.1.3		Spring .Net 1.3.2	
	SP	2.642,60	±1,97	764,00	±0,00	66.852,30	±1,85	67.113,57	±1,91	66.687,27	±1,80	50.980,99	±1,99	34.749,00	±1,34	36.037,33	±0,78
	DP	26.925,99	±1,06	36.301,49	±0,02	79.302,74	±1,34	78.712,33	±1,98	81.118,22	±1,95	35.965,49	±1,97	100.648,66	±1,35	67.466,50	±1,45
	Full	29.023,50	±0,46	36.941,71	±1,92	93.253,80	±1,78	93.405,49	±0,75	92.352,22	±1,86	87.974,89	±1,81	101.223,53	±1,92	69.779,00	±0,41
	WR	652,79	±1,95	238,33	±0,98	14.677,08	±1,91	14.970,66	±1,06	14.839,78	±1,89	4.769,33	±0,83	21.117,66	±1,72	16.552,00	±0,00
Around (no proceed)	SP	2.929,33	±1,54	441,99	±1,68	73.597,00	±1,50	73.283,66	±1,92	73.374,49	±1,34	51.107,99	±0,45	21.519,00	±1,41	19.991,50	±0,76
	DP	27.416,24	±1,79	36.351,99	±1,77	91.922,33	±1,93	91.310,75	±1,98	90.355,00	±0,61	35.895,33	±1,56	98.199,33	±1,14	49.748,49	±0,55
	Full	29.637,04	±1,95	36.620,33	±0,66	101.605,66	±1,83	100.261,50	±0,54	100.698,20	±1,86	87.674,33	±1,88	99.193,00	±1,99	55.072,50	±0,37
Around	DP	28.724,66	±1,63	38.592,66	±1,50	88.218,33	±1,40	90.714,23	±1,97	88.311,66	±0,68	39.567,49	±1,11	111.388,54	±1,83	64.778,99	±1,08
	Full	30.143,00	±1,48	43.754,00	±1,84	102.315,33	±1,35	102.134,39	±1,90	101.753,66	±1,69	93.880,49	±1,10	111.236,74	±1,50	70.359,00	±1,83

Tabla B.1: Tiempo de ejecución (milisegundos) *Method Call* para Tejedores Estáticos.

		JAsCo		DSAW Dinámico		PROSE 1.4.0	Rapier-Loom 4.2 (RC1)	
Before	WR	3.584,49	±0,25	7.613,49	±1,55	2.544.000,00	4.305,33	±1,46
	SP	3.647,80	±1,85	7.675,99	±0,87	2.474.000,00	51.880,66	±0,69
	DP	91.907,99	±1,17	44.326,60	±1,72	2.574.000,00	36.691,00	±0,00
	Full	96.197,50	±0,75	43.988,00	±0,89	2.584.000,00	87.086,33	±0,99
After	WR	3.568,68	±1,92	8.147,56	±1,41	2.474.000,00	4.782,66	±1,34
	SP	3.694,00	±0,76	8.274,33	±0,77	2.644.000,00	51.565,99	±0,27
	DP	91.328,99	±1,69	44.337,49	±1,27	2.503.000,00	36.529,99	±1,40
	Full	93.504,89	±1,99	44.911,00	±0,95	2.554.000,00	84.149,19	±1,95
Around (no proceed)	WR	3.110,23	±3,95	7.324,50	±1,83		5.342,49	±0,84
	SP	3.029,49	±1,48	7.386,74	±1,39		54.036,33	±0,96
	DP	91.674,50	±1,91	47.181,66	±1,18		38.146,00	±1,71
	Full	91.887,57	±1,97	47.478,00	±0,94		91.202,00	±0,80
Around	DP	92.468,00	±1,88	48.306,49	±1,46		38.167,99	±1,35
	Full	92.703,65	±1,97	44.614,00	±1,22		90.800,46	±1,97

Tabla B.2: Tiempo de ejecución (milisegundos) *Method Call* paraTejedores Dinámicos.

Momento		AspectJ		DSAW Estático		DSAW Dinámico		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap	
Before	WR	481,00	±0,84	489,49	±1,83	7.668,24	±1,44	4.040,49	±0,66	4.257,40	±1,73	4.248,66	±1,73
	SP	2.565,37	±1,87	831,69	±1,91	7.769,66	±1,67	67.690,14	±1,85	69.263,00	±1,27	69.015,99	±1,97
	DP	27.842,33	±1,96	37.480,12	±1,73	47.977,66	±1,54	83.127,00	±0,77	82.080,33	±1,94	81.944,66	±1,59
	Full	29.587,49	±0,39	37.443,25	±1,30	48.892,50	±1,08	97.105,66	±0,93	96.882,50	±0,52	97.620,53	±1,88
After	WR	859,79	±1,52	490,49	±1,83	8.347,00	±1,22	6.406,83	±1,95	4.191,79	±1,82	4.173,50	±1,42
	SP	3.086,46	±1,93	789,66	±1,18	8.370,49	±1,39	71.558,05	±1,96	69.112,49	±0,09	68.955,49	±0,66
	DP	28.206,25	±1,75	37.681,33	±0,94	49.139,99	±0,03	80.949,82	±1,75	81.699,50	±1,66	82.066,99	±0,94
	Full	30.037,59	±1,71	37.745,50	±1,24	49.126,66	±1,35	97.703,10	±1,99	96.847,00	±1,33	96.436,49	±0,86
Around (no proceed)	WR	519,06	±1,97	630,33	±1,85	7.776,49	±1,78	15.613,38	±1,90	15.190,49	±1,33	14.721,33	±0,98
	SP	2.911,78	±1,92	727,33	±1,79	7.847,00	±0,00	77.077,99	±1,42	75.516,80	±1,85	73.350,99	±1,91
	DP	27.781,75	±1,44	37.045,50	±0,26	47.878,49	±1,74	94.181,80	±1,91	94.095,00	±1,38	90.253,49	±0,46
	Full	29.968,00	±1,91	37.161,33	±0,84	48.401,50	±0,24	105.582,49	±1,20	103.414,66	±1,54	101.174,00	±1,39
Around	DP	27.794,33	±0,56	42.768,43	±1,93	52.820,00	±0,71	88.120,75	±1,32	89.337,50	±1,81	88.004,99	±0,79
	Full	30.009,25	±1,97	42.731,29	±1,93	52.817,00	±1,29	102.784,50	±1,49	101.610,50	±1,22	102.185,33	±1,87

Tabla B.3: Tiempo de ejecución (milisegundos) *Method Execution*.

		AspectJ		DSAW Estático		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap		Gripper-Loom 4.2 (RC1)		DSAW Dinámico		Rapier-Loom 4.2 (RC1)	
Before	WR	1.029,50	±0,87	3.151,43	±1,94	5.387,33	±0,69	5.758,66	±1,05	5.470,33	±1,78	8.758,49	±1,74	10.764,66	±1,77	80.973,29	±2,24
	SP	2.735,22	±1,89	3.535,25	±1,87	78.202,79	±1,43	81.262,24	±1,95	78.598,74	±1,90	80.257,63	±2,57	10.927,99	±1,31	202.286,06	±1,99
	DP	13.756,99	±1,90	37.799,11	±1,94	65.875,88	±1,97	67.785,00	±1,82	65.176,99	±0,44	37.164,66	±1,13	47.149,33	±1,98	120.890,56	±2,13
	Full	15.417,99	±0,93	37.603,99	±1,09	139.604,66	±1,92	144.725,99	±1,75	141.418,54	±1,83	113.388,50	±1,86	46.989,00	±1,56	264.013,33	±0,94
After	WR	980,16	±1,98	3.036,07	±1,94	5.459,99	±1,98	5.788,44	±1,80	5.849,74	±1,99	8.618,37	±1,74	11.154,40	±1,89	78.780,00	±0,70
	SP	2.778,75	±1,85	3.410,49	±1,98	78.195,00	±0,89	81.076,99	±0,88	81.192,00	±1,99	78.146,03	±2,27	11.052,50	±1,21	199.836,50	±0,00
	DP	13.794,49	±1,73	37.588,00	±1,86	64.973,99	±1,29	67.287,00	±1,06	66.879,00	±0,13	37.268,49	±1,47	47.203,60	±1,55	133.812,73	±4,32
	Full	15.619,50	±1,92	37.970,49	±0,73	139.893,00	±1,69	145.768,99	±0,73	144.816,28	±1,80	121.331,66	±1,79	47.796,00	±1,39	257.168,36	±2,97
Around	DP	17.687,50	±1,50	43.530,99	±0,75	107.535,78	±1,94	72.972,33	±1,41	72.229,50	±1,80	101.717,14	±0,80	51.481,99	±1,57	179.116,82	±1,90
	Full	17.687,50	±1,32	43.625,33	±0,51	147.684,66	±1,88	148.237,66	±1,57	147.523,37	±1,57	199.930,70	±1,99	51.467,00	±1,64	298.708,63	±1,94

Tabla B.4: Tiempo de ejecución (milisegundos) *Constructor Call*.

		AspectJ		DSAW Estático		DSAW Dinámico		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap	
Before	WR	1.040,76	±1,98	3.044,28	±1,86	10.350,75	±1,95	5.260,20	±1,58	5.690,53	±1,95	5.611,50	±1,76
	SP	2.742,49	±1,94	3.540,42	±1,97	10.686,24	±1,42	77.984,33	±1,27	81.403,25	±1,53	81.239,00	±1,88
	DP	14.111,42	±1,64	37.327,79	±1,72	51.161,49	±1,35	65.188,49	±1,59	67.767,99	±1,84	67.467,00	±0,15
	Full	15.403,39	±1,87	37.518,00	±1,82	47.184,99	±0,22	139.126,64	±1,90	145.665,00	±1,81	144.604,66	±1,31
After	WR	973,40	±1,99	3.106,76	±1,96	10.246,60	±1,60	5.413,75	±1,95	5.749,05	±1,96	5.655,50	±1,74
	SP	2.779,00	±1,86	3.391,40	±1,84	10.670,66	±0,28	78.764,66	±1,74	81.168,79	±1,92	81.004,99	±0,44
	DP	13.860,49	±0,97	37.213,75	±1,97	50.962,00	±1,09	65.785,50	±1,70	67.463,33	±1,54	67.515,99	±1,11
	Full	15.490,49	±0,05	37.419,33	±1,24	47.408,40	±1,74	144.394,20	±1,90	144.932,50	±1,89	144.265,33	±0,57
Around	DP	18.361,99	±1,71	50.692,42	±0,26	60.279,00	±0,95	96.337,27	±1,97	97.110,55	±1,93	95.995,24	±1,59
	Full	20.265,39	±1,85	50.575,00	±0,63	60.779,99	±1,86	168.955,50	±0,82	172.193,23	±1,85	169.949,41	±1,86

Tabla B.5: Tiempo de ejecución (milisegundos) *Constructor Execution*.

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap	Gripper-Loom 4.2 (RC1)	DSAW Dinámico	PRoSe 1.4.8	Rapier-Loom 4.2 (RC1)
Before	WR	410,07 ±1,99	1.127,00 ±1,95	3.791,00 ±1,71	3.955,99 ±1,80	3.977,16 ±1,94	1.264,39 ±1,46	8.400,49 ±1,81		1.511,46 ±1,92
	SP	2.532,33 ±1,47	1.411,75 ±1,74	4.139,33 ±0,90	4.336,83 ±1,94	4.323,74 ±1,94	47.573,28 ±1,79	8.406,99 ±1,40		47.653,00 ±1,55
	DP	16.726,99 ±1,65	10.785,00 ±0,68	19.671,99 ±0,00	20.244,74 ±1,27	19.780,80 ±1,53	5.101,33 ±1,23	18.471,33 ±1,01		5.267,66 ±0,71
	Full	18.928,00 ±1,36	11.067,99 ±1,29	20.189,40 ±1,90	20.498,33 ±1,60	20.439,99 ±1,66	51.074,49 ±0,54	18.807,00 ±0,66		51.787,42 ±1,70
After	WR	811,21 ±1,96	462,49 ±1,94	3.801,33 ±1,04	3.971,22 ±1,84	4.128,74 ±1,98	1.298,50 ±0,69	7.687,25 ±1,88		1.615,66 ±0,80
	SP	2.927,00 ±1,93	755,33 ±1,72	4.149,99 ±0,00	4.317,37 ±1,84	4.467,49 ±1,83	47.309,66 ±1,41	7.777,49 ±1,96		49.071,50 ±0,97
	DP	17.112,99 ±1,92	10.395,99 ±0,51	20.146,00 ±1,90	20.104,49 ±1,37	20.253,99 ±1,94	5.007,49 ±0,17	21.645,99 ±0,58		5.445,49 ±0,49
	Full	19.385,66 ±1,24	10.586,66 ±0,54	20.357,99 ±1,97	20.706,66 ±1,06	20.699,00 ±1,47	51.242,00 ±1,32	22.877,99 ±0,54		53.130,66 ±1,77
Around	DP	17.978,99 ±0,69	10.680,49 ±0,92	26.360,86 ±1,93	26.275,66 ±1,20	26.530,66 ±1,20		18.713,50 ±0,62	1.472.000,00	
	Full	20.081,25 ±1,50	13.404,74 ±1,83	26.816,50 ±1,69	27.076,66 ±1,08	27.286,99 ±0,72		21.529,00 ±1,25	1.442.000,00	

Tabla B.6: Tiempo de ejecución (milisegundos) *Field Get*.

		AspectJ		DSAW Estático		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap		Gripper-Loom 4.2 (RC1)		DSAW Dinámico		PROSE 1.4.8	Rapier-Loom 4.2 (RC1)	
Before	WR	827,00	±0,00	452,50	±1,98	3.963,75	±1,87	4.271,14	±1,96	4.315,24	±1,96	1.201,00	±0,00	7.612,85	±1,82		1.585,39	±1,91
	SP	2.954,12	±1,97	666,87	±1,72	4.285,27	±1,91	4.649,00	±0,00	4.765,74	±1,68	47.205,50	±1,77	7.636,00	±1,88		49.129,57	±1,96
	DP	20.397,00	±1,82	9.828,00	±0,00	21.481,88	±1,95	21.481,40	±1,50	21.481,24	±1,76	4.913,99	±1,23	17.574,00	±0,92		5.288,24	±1,34
	Full	22.552,66	±1,26	10.046,00	±0,00	21.633,33	±1,88	22.069,99	±1,93	22.186,99	±1,84	51.043,33	±0,81	18.509,50	±0,72		52.611,50	±0,25
After	WR	935,99	±0,00	573,61	±1,88	3.808,07	±1,96	4.220,46	±1,98	4.322,18	±1,93	1.228,50	±1,72	7.573,50	±1,77		1.564,00	±1,40
	SP	2.793,81	±1,98	607,86	±3,35	4.180,50	±0,21	4.719,88	±1,97	4.539,66	±1,38	47.440,99	±0,63	7.722,49	±0,11		48.797,66	±0,85
	DP	20.966,25	±1,44	9.994,66	±1,94	21.291,25	±1,75	21.247,33	±1,09	20.878,00	±1,72	4.988,24	±1,31	17.425,50	±1,35		5.335,50	±0,16
	Full	23.202,20	±1,94	10.054,49	±1,34	21.672,00	±1,73	22.032,66	±1,58	21.988,24	±1,46	51.209,66	±1,03	17.763,33	±1,59		52.868,49	±0,52
Around	DP	21.973,49	±1,86	10.269,99	±0,71	28.185,50	±1,75	29.304,99	±0,49	29.277,24	±1,53	6.671,00	±1,00	18.584,85	±1,73	2.404.000,00	7.035,66	±1,53
	Full	24.247,17	±1,92	10.868,00	±0,89	27.874,20	±1,75	29.481,18	±1,93	29.258,25	±1,69	53.310,66	±1,08	19.703,00	±1,43	2.374.000,00	54.647,00	±0,52

Tabla B.7: Tiempo de ejecución (milisegundos) *Field Set*.

B.2. AWBench

Tejedores Estáticos	AspectJ		DSAW Estático		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap	
	4.801,00	±1,90	4.128,99	1,86	18.469,99	1,79	18645,33	1,82	18.373,75	1,31

Tejedores Dinámicos	DSAW Dinámico		JaSCo 0.8.7		JBossAOP PreCompile		JBossAOP LoadTime	
	24.938,55	±1,95	15.789,49	±1,75	44.815,49	±1,18	45.437,33	±1,85

Tabla B.8: Tiempo de ejecución (milisegundos) AWBench.

B.3. Aplicaciones Reales

B.3.1. Access Control

	Java		C#	
Aplicación sin aspectos	1.856,99	±0,00	1.840,85	±1,88

	AspectJ		DSAW Estático		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap	
Aplicación con aspectos (proceed)	1.966,49	±1,37	1.902,99	±0,94	2.979,49	±0,90	3.799,99	±0,00	3.631,00	±1,52

	AspectJ		DSAW Estático		JBossAOP PreCompile		JBossAOP LoadTime		JBossAOP HotSwap	
Aplicación con aspectos (reflectiva)	1.996,99	±0,00	1.870,00	±1,98	2.948,99	±0,00	3.646,28	±1,84	3.635,49	±0,24

Tabla B.9: Tiempo de ejecución (milisegundos) Access Control.

B.3.2. Comunicaciones Encrypt

		Java		C#	
Aplicación sin aspectos	Escenario 1	15.398,99	±1,74	19.348,56	±2,43
	Escenario 2	34.125,13	±5,23	28.146,59	±1,86

		DSAW Dinámico		JAsCo 0.8.7		JBossAOP PreCompile		JBossAOP LoadTime		PROSE	
Aplicación con aspectos	Escenario 1	15.862,43	±2,42	19.060,00	±1,50	19.348,56	±2,43	20.128,66	±2,01	49.921,80	±8,18
	Escenario 2	37.286,38	±1,98	46.293,13	±2,16	49.984,26	±2.36	49.959,53	±2,97	69.900,31	±2,25

Tabla B.10: Tiempo de ejecución (milisegundos) Comunicaciones Encrypt.

B.3.3. FTP Encrypt.

		Java		C#	
Aplicación sin aspectos	Escenario 1	80.648,50	±3,43	81.561,99	±1,84
	Escenario 2	107.775,71	±1,97	105.543,99	±1,83

		DSAW Dinámico		JAsCo 0.8.7		JBossAOP PreCompile		JBossAOP LoadTime		PROSE	
Aplicación con aspectos	Escenario 1	89.742,00	±1,90	96.322,55	±1,99	106.286,66	±1,77	108.135,85	±1,98	397.767,80	±2,19
	Escenario 2	113.053,49	±1,87	120.553,53	±2,55	136.614,50	±1,55	131.725,26	±2,62	489.303,04	±6,90

Tabla B.11: Tiempo de ejecución (milisegundos) FTP Encrypt.

B.3.4. HotDraw

		Java		C#	
Aplicación sin aspectos	Escenario 1	4.192,36	±1,98	4.971,73	±4,92
	Escenario 2	4.332,85	±1,97	4.109,86	±5,57
	Escenario 3	4.325,37	±1,95	4.227,53	±3,33
	Escenario 4	4.124,14	±1,91	4.129,86	±3,59
	Escenario 5	4.249,44	±1,97	4.283,10	±5,97

		DSAW Dinámico		JAsCo 0.8.7		PROSE	
Aplicación con aspectos	Escenario 1	6.984,86	±5,37	7.312,33	±1,89	11.648,15	±6,00
	Escenario 2	5.712,06	±4,16	7.485,90	±1,90	11.949,58	±4,50
	Escenario 3	6.383,99	±5,51	7.393,42	±1,93	12.219,37	±5,25
	Escenario 4	5.985,69	±4,85	7.175,99	±1,75	10.974,78	±5,54
	Escenario 5	6.097,03	±3,82	7.242,25	±1,59	12.907,84	±2,23

Tabla B.12: Tiempo de ejecución (milisegundos) HotDraw.

Apéndice C. CONSUMO DE MEMORIA

C.1. Micro-benchmark

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap	Gripper-Loom 4.2 (RC1)	Spring Java 3.1.3	Spring .Net 1.3.2	JAsCo	DSAW Dinámico	PROSE 1.4.0	Rapier-Loom 4.2 (RC1)
Before	WR	24.780,00	15.112,00	47.176,00	60.292,00	59.520,00	18.532,00	362.364,00	24.760,00	374.084,00	29.124,00	15.343,00	30.284,00
	SP	26.644,00	15.072,00	372.644,00	377.032,00	377.204,00	18.592,00	362.724,00	25.696,00	371.088,00	29.168,00	15.589,00	30.316,00
	DP	361.268,00	18.524,00	375.488,00	380.964,00	375.928,00	18.428,00	370.520,00	25.796,00	358.312,00	29.028,00	15.429,00	30.040,00
	Full	365.908,00	18.484,00	365.124,00	377.968,00	374.332,00	18.668,00	373.860,00	25.620,00	359.736,00	31.052,00	15.581,00	30.068,00
After	WR	25.632,00	15.192,00	47.284,00	59.960,00	59.280,00	18.476,00	377.080,00	25.784,00	372.884,00	28.864,00	15.511,00	30.268,00

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap	Gripper-Loom 4.2 (RC1)	Spring Java 3.1.3	Spring .Net 1.3.2	JAsCo	DSAW Dinámico	PROSE 1.4.0	Rapier-Loom 4.2 (RC1)
	SP	29.320,00	15.252,00	375.708,00	380.836,00	377.052,00	18.716,00	362.692,00	25.728,00	373.444,00	31.088,00	15.749,00	30.292,00
	DP	360.256,00	18.716,00	375.916,00	383.944,00	382.692,00	18.440,00	374.544,00	25.720,00	358.360,00	29.028,00	15.687,00	30.080,00
	Full	364.396,00	18.728,00	368.436,00	374.128,00	382.300,00	18.708,00	368.180,00	25.808,00	360.276,00	29.008,00	15.671,00	30.128,00
	WR	24.920,00	15.096,00	364.972,00	368.756,00	367.244,00	18.568,00	363.760,00	25.732,00	372.260,00	29.064,00		30.364,00
Around (no proceed)	SP	26.764,00	15.108,00	370.172,00	379.588,00	380.080,00	18.636,00	364.184,00	25.580,00	372.936,00	29.008,00		30.284,00
	DP	360.244,00	18.468,00	375.980,00	366.760,00	374.636,00	18.560,00	370.224,00	25.520,00	366.468,00	28.972,00		30.112,00
	Full	366.380,00	18.624,00	371.156,00	377.444,00	384.260,00	18.640,00	370.220,00	25.524,00	367.988,00	28.888,00		30.176,00
Around	DP	361.556,00	18.560,00	369.360,00	377.836,00	383.968,00	18.572,00	373.696,00	26.684,00	359.600,00	31.172,00		30.220,00
	Full	361.532,00	18.532,00	372.588,00	376.396,00	374.488,00	18.664,00	375.444,00	25.608,00	358.844,00	29.140,00		30.148,00

Tabla C.1: Memoria consumida (Kbytes) *Method Call*.

Momento		AspectJ	DSAW Estático	DSAW Dinámico	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap
Before	WR	24.828,00	15.524,00	29.008,00	47.936,00	58.920,00	58.792,00
	SP	26.612,00	15.652,00	28.916,00	377.688,00	381.136,00	379.820,00
	DP	363.316,00	19.404,00	28.872,00	368.920,00	381.204,00	372.008,00
	Full	366.260,00	19.496,00	29.048,00	365.240,00	381.140,00	372.740,00
After	WR	25.256,00	14.696,00	30.136,00	47.964,00	56.976,00	56.028,00
	SP	26.768,00	14.968,00	29.660,00	373.456,00	377.400,00	376.344,00
	DP	360.288,00	13.972,00	28.884,00	369.308,00	383.484,00	373.844,00
	Full	360.276,00	14.024,00	29.116,00	372.028,00	373.828,00	376.016,00
Around (no proceed)	WR	25.052,00	15.140,00	29.000,00	367.072,00	366.816,00	368.412,00
	SP	26.812,00	15.188,00	29.080,00	374.968,00	388.264,00	387.588,00
	DP	360.188,00	18.472,00	28.888,00	369.100,00	378.268,00	379.460,00
	Full	360.544,00	18.492,00	29.116,00	374.616,00	376.492,00	374.728,00
Around	DP	366.272,00	18.676,00	29.160,00	368.880,00	374.968,00	383.140,00
	Full	366.272,00	18.640,00	29.040,00	374.728,00	395.148,00	395.084,00

Tabla C.2: Memoria consumida (Kbytes) *Method Execution*.

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap	Gripper-Loom 4.2 (RC1)	DSAW Dinámico	Rapier-Loom 4.2 (RC1)
Before	WR	26.276,00	18.384,00	358.232,00	248.196,00	366.188,00	18.456,00	29.308,00	30.128,00
	SP	32.300,00	18.340,00	375.068,00	374.260,00	382.288,00	18.780,00	29.300,00	30.272,00
	DP	365.824,00	18.604,00	366.000,00	383.412,00	378.204,00	18.556,00	29.212,00	30.184,00
	Full	359.580,00	18.676,00	370.604,00	377.300,00	375.308,00	18.608,00	29.348,00	30.224,00
After	WR	26.164,00	18.460,00	358.356,00	364.928,00	365.280,00	18.540,00	29.152,00	30.300,00
	SP	32.508,00	18.320,00	371.540,00	373.612,00	382.944,00	18.640,00	29.372,00	30.196,00
	DP	364.920,00	18.580,00	369.348,00	375.340,00	376.880,00	18.492,00	29.304,00	30.208,00
	Full	366.156,00	18.532,00	374.336,00	384.000,00	374.652,00	18.808,00	29.324,00	30.216,00
Around	DP	359.736,00	19.228,00	377.396,00	370.764,00	385.252,00	18.796,00	29.276,00	30.100,00
	Full	364.104,00	19.348,00	385.854,00	390.836,00	387.804,00	18.832,00	29.208,00	30.304,00

Tabla C.3: Memoria consumida (Kbytes) *Constructor Call*.

		AspectJ	DSAW Estático	DSAW Dinámico	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap
Before	WR	27.112,00	17.836,00	29.640,00	366.240,00	364.804,00	366.332,00
	SP	354.104,00	18.112,00	29.840,00	375.168,00	383.580,00	383.520,00
	DP	359.420,00	18.544,00	27.960,00	364.600,00	368.392,00	368.512,00
	Full	364.048,00	18.568,00	28.072,00	368.696,00	374.592,00	381.656,00
After	WR	27.000,00	17.880,00	27.932,00	369.024,00	365.148,00	368.164,00
	SP	37.916,00	18.100,00	27.944,00	372.736,00	374.132,00	377.168,00
	DP	362.044,00	18.596,00	29.740,00	364.556,00	368.876,00	368.376,00
	Full	359.732,00	18.540,00	28.060,00	374.168,00	383.108,00	374.612,00
Around	DP	360.116,00	19.280,00	29.268,00	369.656,00	383.236,00	385.600,00
	Full	359.576,00	19.352,00	29.296,00	383.100,00	380.520,00	385.288,00

Tabla C.4: Memoria consumida (Kbytes) *Constructor Execution*.

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap	Gripper-Loom 4.2 (RC1)	DSAW Dinámico	PRoSe 1.4.8	Rapier-Loom 4.2 (RC1)
Before	WR	24.856,00	15.352,00	47.036,00	58.556,00	57.212,00	18.296,00	29.748,00		30.916,00
	SP	26.800,00	15.212,00	47.720,00	59.044,00	58.972,00	18.716,00	29.772,00		30.088,00
	DP	365.504,00	18.456,00	373.812,00	380.736,00	368.176,00	18.348,00	29.176,00		29.812,00
	Full	361.960,00	18.592,00	373.784,00	367.956,00	380.436,00	18.704,00	29.172,00		29.900,00
After	WR	26.780,00	15.236,00	46.820,00	58.972,00	58.036,00	18.288,00	29.832,00		30.996,00
	SP	26.988,00	15.220,00	47.624,00	58.652,00	58.068,00	18.820,00	29.760,00		30.148,00
	DP	364.416,00	18.560,00	372.308,00	380.516,00	382.100,00	18.408,00	31.320,00		30.068,00
	Full	366.628,00	18.452,00	366.664,00	369.092,00	382.892,00	18.664,00	31.224,00		30.004,00
Around	DP	366.476,00	18.392,00	375.928,00	375.988,00	368.412,00		31.204,00	19.046,40	
	Full	361.972,00	18.440,00	372.392,00	382.672,00	367.528,00		28.988,00	19.025,92	

Tabla C.5: Memoria consumida (Kbytes) *Field Get*.

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap	Gripper-Loom 4.2 (RC1)	DSAW Dinámico	PROSE 1.4.8	Rapier-Loom 4.2 (RC1)
Before	WR	25.260,00	15.164,00	46.976,00	59.548,00	58.144,00	18.376,00	29.680,00		30.796,00
	SP	27.696,00	15.220,00	47.936,00	58.424,00	60.084,00	18.672,00	29.732,00		29.988,00
	DP	365.832,00	18.436,00	367.032,00	368.136,00	379.100,00	18.448,00	29.012,00		29.996,00
	Full	366.648,00	18.364,00	365.512,00	383.308,00	378.312,00	18.680,00	29.012,00		29.940,00
After	WR	25.208,00	15.200,00	46.888,00	59.696,00	59.580,00	18.328,00	29.728,00		30.812,00
	SP	27.488,00	15.316,00	47.900,00	58.456,00	60.000,00	18.812,00	29.632,00		30.132,00
	DP	362.724,00	18.436,00	366.916,00	380.480,00	378.444,00	18.348,00	29.176,00		29.848,00
	Full	362.160,00	18.444,00	375.124,00	380.332,00	377.464,00	18.676,00	28.976,00		30.316,00
Around	DP	366.144,00	18.480,00	375.136,00	382.400,00	381.788,00	18.620,00	29.220,00	19.038,20	30.012,00
	Full	362.436,00	18.452,00	370.352,00	377.964,00	376.500,00	18.752,00	28.972,00	19.070,97	30.072,00

Tabla C.6: Memoria consumida (Kbytes) *Field Set*.

C.2. AWbench

Tejedores Estáticos	AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap
	361.856,00	18.684,00	368.112,00	374.288,00	377.044,00

Tejedores Dinámicos	DSAW Dinámico	JaSCo 0.8.7	JBossAOP PreCompile	JBossAOP LoadTime
	29.868,00	411.156,00	235.296,00	249.052,00

Tabla C.7: Memoria consumida (Kbytes) AWBench.

C.3. Aplicaciones Reales

C.3.1. Access Control

		Java	C#
Aplicación sin aspectos	Servidor	36.780,00	25.024,00
	Servidor/Cliente	41.372,00	25.024,00
	Cliente	32.880,00	22.120,00

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap
Aplicación con aspectos (proceed)	Servidor	40.032,00	24.300,00	52.872,00	63.712,00	64.868,00
	Servidor/Cliente	45.072,00	24.176,00	55.652,00	65.448,00	65.828,00
	Cliente	35.236,00	21.120,00	55.884,00	65.120,00	66.288,00

		AspectJ	DSAW Estático	JBossAOP PreCompile	JBossAOP LoadTime	JBossAOP HotSwap
Aplicación con aspectos (reflectiva)	Servidor	41.320,00	24.388,00	53.436,00	62.740,00	64.304,00
	Servidor/Cliente	44.480,00	24.360,00	57.592,00	65.836,00	65.604,00
	Cliente	35.372,00	21.176,00	56.168,00	65.664,00	65.880,00

Tabla C.8: Memoria consumida (Kbytes) Access Control.

C.3.2.Comunicaciones Encrypt.

			Java	C#
Aplicación sin aspectos	Escenario 1	Nodo A	157.552,00	22.320,00
		Nodo B	94.786,00	22.320,00
	Escenario 2	Nodo A	151.940,00	23.166,00
		Nodo B	152.992,00	23.796,00

			DSAW Dinámico	JAsCo 0.8.7	JBossAOP PreCompile	JBossAOP LoadTime	PROSE
Aplicación con aspectos	Escenario 1	Nodo A	32.732,00	51.176,00	66.536,00	88.656,00	16.695,29
		Nodo B	32.700,00	54.496,00	66.984,00	97.848,00	16.637,95
	Escenario 2	Nodo A	33.224,00	50.032,00	102.908,00	178.744,00	16.044,03
		Nodo B	33.212,00	49.188,00	62.968,00	105.012,00	16.486,40

Tabla C.9: Memoria consumida (Kbytes) Comunicaciones Encrypt.

C.3.3.FTP Encrypt.

			Java	C#
Aplicación sin aspectos	Escenario 1	Servidor	105.444,00	26.212,00
		Cliente	107.904,00	26.168,00
	Escenario 2	Servidor	150.808,00	26.100,00
		Cliente	92.828,00	26.208,00

			DSAW Dinámico	JAsCo 0.8.7	JBossAOP PreCompile	JBossAOP LoadTime	PROSE
Aplicación con aspectos	Escenario 1	Servidor	35.124,00	45.424,00	49.716,00	172.452,00	16.527,36
		Cliente	35.236,00	43.496,00	68.528,00	82.756,00	15.380,48
	Escenario 2	Servidor	35.184,00	51.076,00	64.356,00	126.196,00	16.297,98
		Cliente	35.232,00	42.532,00	66.656,00	135.884,00	16.322,56

Tabla C.10: Memoria consumida (Kbytes) FTP Encrypt.

C.3.4.HotDraw

		Java	C#
Aplicación sin aspectos	Escenario 1	65.640,00	38.584,00
	Escenario 2	67.440,00	39.168,00
	Escenario 3	67.260,00	38.508,00
	Escenario 4	64.764,00	38.972,00
	Escenario 5	65.452,00	37.732,00

		DSAW Dinámico	JAsCo 0.8.7	PROSE
Aplicación con aspectos	Escenario 1	48.588,00	205.636,00	42.192,90
	Escenario 2	47.808,00	150.488,00	42.336,26
	Escenario 3	48.552,00	230.372,00	41.996,29
	Escenario 4	48.616,00	146.240,00	61.566,21
	Escenario 5	47.976,00	175.212,00	44.097,54

Tabla C.11: Memoria consumida (Kbytes) HotDraw.

Apéndice D.
PUBLICACIONES DERIVADAS

Fruto de la investigación realizada en la tesis doctoral aquí presentada son los siguientes artículos indexados en revistas JCR (artículos 1 y 2) y SCImago (artículo 3), así como un congreso internacional core B con un factor de aceptación del 16,5% (artículo 4):

1. The DSAW Aspect-Oriented Software Development Platform. Francisco Ortin, Luis Vinuesa, Jose M. Félix. International Journal of Software Engineering and Knowledge Engineering, Volume 21, Issue 7, pp. 891-929. November 2011. Revista indexada en el *Journal Citation Reports*.
2. Efficient Aspect Weaver for the .Net Platform. Jose M. Félix, Francisco Ortín. IEEE Latin America Transactions, 2014. Artículo aceptado, pendiente de publicación. Revista indexada en el *Journal Citation Reports*.
3. Aspect-Oriented Programming to Improve Modularity of Object-Oriented Applications. Jose M. Félix, Francisco Ortin. Journal of Software, Volume 9, Issue 9, pp. 2454-2460, September 2014. Revista indexada en el *SCImago journal ranking*.
4. DSAW: A Dynamic and Static Aspect Weaving Platform. Luis Vinuesa, Francisco Ortín, José M. Félix, Fernando Álvarez. Proceedings of the International Conference on Software and Data Technologies (ICSOFT), pp. 55-62, Porto (Portugal). July 2008. Congreso Core B. Factor de aceptación del 16,5%.

REFERENCIAS

[Alonso04] Gustavo Alonso. PROSE: Automated Software Adaptation for Pervasive Computing. *ERCIM News Nº 58*, 2004.

[Amor07] Mercedes Amor, Alessandro Garcia, Lidia Fuentes. AGOL: An Aspect-Oriented Domain-Specific Language for MAS. *Proceed. of the Early Aspects at ICSE: Workshops in Aspect-Oriented Requirements Engineering and Architecture Design (EARLYASPECTS '07). IEEE Computer Society*, 2007.

[AOSD13] Aspect-Oriented Software Association Aspect-Oriented Software Development. *http://www.aosd.net/*, 2013.

[Apache12] Apache Software Foundation. Apache log4net: Home. *http://logging.apache.org/log4net/*, 2012.

[Aracic06] I. Aracic, V. Gasiunas, M. Mezini, K.Ostermann. Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I. LNCS, Vol. 3880,* páginas 135-173, 2006,

[Aspectc12] AspectC++. The home of AspectC++. *http://www.aspectc.org/*, 2012.

[Aspectr13] AspectR. Module: AspectR. *http://www.ruby-doc.org/gems/docs/a/aspectr-0.3.7/AspectR.html*, 2013.

[Assaf08] A. Assaf, J. Noyé. Dynamic AspectJ. *Proceed. of the 2008 Symposium on Dynamic languages (DLS '08*), ACM, 2008.

[Baldoni02] R. Baldoni, C. Marchetti, A. Termini. Active software replication through a three-tier approach. *Proceed. of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, paginas 109-118, 2002,

[Blackstock04] M. Blackstock. Aspect weaving with C# and .Net. *http://www.cs.ubc.ca/michael/publications/AOPNET5.pdf*, 2004.

[Bockisch06] C.M. Bockisch, M. Arnold, T. Dinkelaker, M. Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. *Proceed. of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA, páginas 109-124, 2006.

[Böllert99] K. Böllert. On weaving aspects. *Proceed. of the Workshop on Object-Oriented Technology*, páginas 301-302, Springer-Verlag, 1999.

[Broch98] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. *Proceed. of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, páginas 85-97, 1998.

[Brown98] Alan W. Brown, Kurt C. Wallnau. The Current State of CBSE. *IEEE Softw. 15*, páginas 37-46, 1998.

[Chiba00] S. Chiba. Load-time structural reflection in Java. *Proc. of ECOOP, LNCS,* páginas 313-336, 2000.

[Chiles13] Bill Chiles, Alex Turner. Dynamic Language Runtime. *http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=127512*, 2013.

[CORBA12] CORBA. OMG's CORBA homepage. *http://www.corba.org/*., 2012

[Djoko06] S. D. Djoko, P. Fradet, D. L. Botlan, CASB: Common Aspect Semantics Base, Deliverable 5. *AOSD-Europe, EU Network of Excellence in AOSD*, 2006.

[Dmitriev02] M. Dmitriev. Applications of the hotswap technology to advanced profiling. *USE2002: First International Workshop on Unanticipated Software Evolution,* Springer, 2002.

[Dounce04] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. *Proceed. of the 3rd international conference on Aspect-oriented software development (AOSD '04)*, ACM, páginas 141-150, 2004.

[Durr05] P. Durr, T. Staijen, L. Bergmans, M. Aksit. Reasoning about semantic conflicts between aspects. *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, páginas 10-18, 2005.

[Eaddy05] M. Eaddy, S. Feiner. Multi-Language Edit-and-Continue for the Masses. *Technical Report CUCS-015-05, Department of Computer Science, Columbia University*, 2005.

[Eaddy07] M. Eaddy. Wicca 2.0: Dynamic Weaving using the .Net 2.0 Debugging API. *Demonstration at Aspect-Oriented Software Development (AOSD'07)*, 2007.

[Eaddy07b] M. Eaddy. Wicca v2 home. *http://www1.cs.columbia.edu/~eaddy/wicca/*, 2007

[Eaddy07c] M. Eaddy, A. Aho, W. Hu, P. McDonald, J. Burger. Debugging aspect-enabled programs. *SC2007: Proceedings of Software Composition 2007*, páginas 200-215, 2007.

[Eclipse13] Eclipse Foundation. AspectJ Development Tools (AJDT). *http://www.eclipse.org/ajdt/*, 2013.

[Eclipse13b] Eclipse Foundation. AspectJ, crosscutting objects for better modularity. *http://www.eclipse.org/aspectj*, 2013.

[Ecma06] European Computers Manufacturing Association. Standard ECMA-335, Common Language Infrastructure (CLI), 4th edition. *ECMA - European Computers Manufacturing Association*, 2006.

[Félix14] Jose M. Felix, Francisco Ortin. Aspect-Oriented Programming to Improve Modularity of Object-Oriented Applications. *Journal of Software*, volume 9, issue 9, pp. 2454-2460, septiembre 2014.

[Fraine05] Bruno De Fraine, Wim Vanderperren, Davy Suvée, Johan Brichau. Jumping aspects revisited. *Dynamic Aspects Workshop (DAW'05)*, 2005.

[Frei04] A. Frei, P. Grawehr, G. Alonso. A Dynamic AOP-Engine for .Net. *Technical Report 445. Zurich: Department of Computer Science, ETH*. 2004.

[Fuentes03] L. Fuentes, M. Pinto, A. Vallecillo. How MDA can help designing Component and Aspect Based applications. *Proc. of 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03)*, páginas 124-135, 2003

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. *Addison Wesley*, 1994.

[Garcia12] Miguel Garcia, David Llewellyn-Jones, F. Ortin, Madjid Merabti. Applying dynamic separation of aspects to distributed systems security: A case study. *IET Software 6(3)*, páginas 231-248, 2012.

[Garcia13] Miguel García, Francisco Ortin, David Llewellyn-Jones, Madjid Merabti. A Performance Cost Evaluation of Aspect Weaving. *ACM Proceedings of the 36 Australian Computer Science Conference (ACSC)*, páginas 79-86, 2013.

[Garcia13a] Miguel García. Improving the Runtime Performance and Robustness of Hybrid Static and Dynamic Typing Languages. *Tesis doctoral. Universidad de Oviedo*, 2013.

[Garcia14] Miguel Garcia, Francisco Ortin, Jose Quiroga. Design and implementation of an efficient hybrid dynamic and static typing language. Software: Practice and Experience. Aceptado, pendiente de publicación. doi: 10.1002/spe.2291.

[Georges07] A. Georges, D. Buytaert, L. Eeckhout. Statistically rigorous Java performance evaluation. *Proc. of the ACM Conference on Object-Oriented Programming Systems and Applications (OOSPLA '07)*, páginas 57-56, 2007.

[Gilani04] W. Gilani, O. Spinczyk. A Family of Aspect Dynamic Weavers. *AOSD Dynamic Aspects Workshop (DAW'04)*, 2004

[Gilani04b] W. Gilani, N. Hasan Naqvi, O. Spinczyk. On Adaptable Middleware Product Lines. *3rd Workshop on Reflective and Adaptive Middleware, ACM/IFIP/USENIX 5th International Middleware Conference*, páginas 207-213, 2004.

[Gilani07] W. Gilani, F. Scheler, D. Lohmann, O. Spinczyk, W. Schröder-Preikschat. Unification of static and dynamic aop for evolution in em-

bedded software systems. *Software Composition, LNCS*, páginas. 216-234, 2007.

[Gorappa05] S. Gorappa, J.A. Colmenares, H. Jafarpour, R. Klefstad. 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'05)*, páginas 342-349, 2005.

[Haupt04] M. Haupt, M. Mezini. Micro-measurements for dynamic aspect-oriented systems. *Net.ObjectDays*, páginas 81-96, 2004.

[Haupt06] Michael Haupt. Virtual machine support for aspect-oriented programming languages. *Tesis Doctoral. Darmstadt University of Technology, 2006.*

[Hannad10] Youssef Hannad, Fabrice Mourlin. Mobile agent driven by aspect. *International Journal of Computer Science Issues (IJCSI)*, 2010.

[Hilsdale04] Erik Hilsdale, Jim Hugunin. Advice weaving in AspectJ. *Proc. of the 3rd International Conference on Aspect-oriented Software Development (AOSD '04)*, páginas 26-35, 2004.

[Holma00] H. Holma, A. Toskala, et al. WCDMA for UMTS: Radio access for third generation mobile communications*. Wiley, 1th edition*, 2000.

[Hürsch95] W. Hürsch, C. Lopes. Separation of Concerns. *Technical Report NU-CCS-95-03. Boston: Northeastern University*, 1995.

[Jagadeesan06] R. Jagadeesan, A. Jeffrey, J. Riely. Typed parametric polymorphism for aspects. *Sci. Comput. Program*, *vol. 63, no. 3*, páginas. 267-296, 2006.

[JAsCo13] JAsCo. eclipse:updatemanager - JAsCo - System and Software Engineering Lab. *http://ssel.vub.ac.be/jasco/eclipse_updatemanager.html*, 2013

[Johnson92] Ralph E. Johnson. Documenting frameworks using patterns. *OOPSLA: International conference on Object-oriented programming systems, languages, and applications,* páginas 63-76, 1992.

[Kiczales91] G. Kiczale, J. D. Rivieres. The Art of the Metaobject Protocol*. Cambridge, MA, USA: MIT Press*, 1991.

[Kiczales96] G. Kiczale. Aspect-Oriented Programming. *ACM Comput. Survey 28,* 1996.

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier y J. Irwin. Aspect-Oriented Programming. *ECOOP: European Conference on Object-Oriented Programming*, páginas 220-242, 1997.

[Kiczales01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. An Overview of AspectJ. *European Conference on Object-Oriented Programming (ECOOP'01),* páginas 327-353, 2001.

[Kizcales03] Gregor Kizcales. The fun has begun. *Aspect-Oriented Software Development (AOSD)*, 2003.

[Klefstad02] Klefstad, R., Schmidt, D. C., O'Ryan, C. Towards Highly Configurable Real-time Object Request Brokers. *Symposium on Object-Oriented Real-time Distributed Computing (ISORC'02)*, páginas 437-447,2002.

[Köhne05] K. Köhne, W. Schult, A. Polze. Design by contract in .Net Using Aspect Ori-ented Programming. *Technical Report. Instituto Hasso Plattner, University of Potsdam*. 2005.

[Lafferty03] D. Lafferty y V. Cahill. Language-independent aspect-oriented programming. *Proceed. of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA'03)*, páginas 1-12, 2003.

[Lago10] Noelia Lago Ouviaño. Modelo-Vista-Controlador con separación dinámica de aspectos. *Proyecto Fin de Carrera. Escuela de Ingeniería de Gijón. Universidad de Oviedo*, 2010.

[Lämmel02] R. Lämmel. A Semantical Approach to Method-Call Interception. *Proc. Of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, páginas 41-55, 2002.

[Lang02] U. Lang, R. Schreiner. Developing secure distributed systems with CORBA. *Artech House Publishers*, 2002.

[Lilja00] D. J. Lilja. Measuring computer performance: a practitioner's guide. *Cambridge University Pres*, 2000.

[Lohmann04] D. Lohmann, W. Gilani, O. Spinczyk. On Adapable Aspect-Oriented Operating Systems. *ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS'04)*, 2004.

[LOOM12] LOOM.NET. *http://loom.codeplex.com/*, 2012.

[Marek12] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, Zhengwei Qi. DiSL: a domain-specific language for bytecode instrumentation. *Proceed. of the 11th annual international conference on Aspect-oriented Software Development (AOSD '12), ACM*, páginas 239-250, 2012.

[Masuhara03] H. Masuhara, G. Kiczales, C. Dutchyn. A compilation and optimization model for aspect-oriented programs. *Compiler Construction, volume 2622 of Springer Lecture Notes in Computer Science*, páginas 46-60, 2003.

[Meyer97] Bertand Meyer. Object-Oriented Software Construction, 2nd edition. *Pretince-Hall*, 1997.

[Microsoft12] Microsoft. Windows Management Instrumentation. *http://msdn.microsoft.com/en-us/library*, 2012.

[Microsoft13] Microsoft. Generación de código usando CodeDOM. *http://msdn.microsoft.com/es-es/library/bb972255.aspx*, 2013.

[Microsoft13b] Phoenix Compiler and Shared Source Common Language Infrastructure. *http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx*, 2013.

[Microsoft13c] Microsoft. Visual Studio Software Development Kit (SDK). *http://msdn.microsoft.com/en-us/library/vstudio/bb166217%28v=vs.90%29.aspx*, 2013.

[Microsoft13d] Microsoft. Visual Studio Integration SDK Roadmap. *http://msdn.microsoft.com/en-us/library/cc138569%28v=vs.90%29.aspx*, 2013.

[Microsoft13d] Microsoft. .Net Remoting. *http://msdn.microsoft.com/en-us/library/bb166560%28v=vs.100%29.aspx*, 2013.

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

[Moser99] L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, V. Kalogeraki. The Eternal System: An architecture for enterprise applications. *Proceed. of the Third International Enterprise Distributed Object Computing Conference (EDOC'99)*, páginas 214-222, 1999.

[Nagy05] I. Nagy, L. Bergmans, M. Aksit. Composing aspects at shared join points. *Conference proceedings : Erfurt, Germany, Lecture Notes in Informatics* 69, páginas 19-38, 2005.

[NCSC90] National Computer Security Center NCSC. Trusted network interpretation environments guideline. 1990.

[Nicoara05] Angela Nicoara, Gustavo Alonso. Dynamic AOP with PROSE. *Proc. of the International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05) in conjunction with the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05*), 2005.

[Nicoara13] Angela Nicoara, A. Popovici. PROSE - Browse /prose/1.4.0 at SourceForge.net. *http://sourceforge.net/projects/jprose/files/prose/1.4.0/*, 2013.

[Ogel05] F. Ogel, G. Thomas, B. Folliot. Supporting efficient dynamic aspects through reflection and dynamic compilation. *Proc. of the 2005 ACM symposium on Applied computing*, ACM, páginas 1351-1356, 2005.

[OHair04] Kelly O'Hair. The JVMPI Transition to JVMTI. *Oracle Technology Network*, 2004.

[Ossher01] H. Ossher, P. Tarra, Hyper/J: Multi-dimensional separation of concerns for Java. *International Conference in Software Engineering (ICSE)*, páginas 821-822, 2001.

[Ortin03] Francisco Ortin, Juan Manuel Cueva, Ana Belén Martínez, The reflective nitro abstract machine. *SIGPLAN Not., Vol. 38, Nº. 6*, páginas 40-49, 2003.

[Ortin04] Francisco Ortin, Benjamín Lopez, J. Baltasar G. Perez-Schofield. Separating adaptable persistence attributes through computational reflection. *IEEE Software, Vol. 21, Nº.. 6,* páginas 41-49, 2004.

[Ortin04b] Francisco Ortin, Juan Manuel Cueva. Dynamic Adaptation of Application Aspects. *Journal of Systems and Software, Vol. 71, Nº. 3,* páginas 229-243, 2004.

[Ortin09] Francisco Ortin, José Manuel Redondo, J. Baltasar García Perez-Schofield. Efficient virtual machine support of runtime structural reflection. *Science Computing Programming, Vol. 74, Nº. 10*, páginas 836-860, 2009.

[Ortin11] Francisco Ortin, Luis Vinuesa, José M. Félix. The DSAW Aspect-Oriented Software Development Platform. *International Journal of Software Engineering and Knowledge Engineering,* volume 21, número 7, páginas 891-929, 2011.

[Ortin14] Francisco Ortin, Patricia Conde, Daniel F. Lanvin, Raúl Izquierdo. The Runtime Performance of invokedynamic: an Evaluation with a Java Library. *IEEE Software,* volumen 31, número 4, páginas 82-90, 2014.

[Ortin14b] Francisco Ortin, Francisco Moreno, Anton Morant. Static Type Information to Improve the IDE Features of Hybrid Dynamically and Statically Typed Languages*. Journal of Visual Languages & Computing, Volume 25, Issue 4*, páginas 346-362, 2014.

[Parnas72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM, Vol. 15*, páginas 1053-1058, 1972.

[Pawlak01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. *Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION*, páginas 1-24, 2001.

[Payer12] Mathias Payer, Boris Bluntschli, Thomas R. Gross. LLDSAL: a low-level domain-specific aspect language for dynamic code-generation and program modification. *Proceed. of the seventh workshop on Domain-Specific Aspect Languages (DSAL '12). ACM*, 2012.

[Perez10] Jesús Pérez Rodríguez. DSAW Development Tools: integración de una plataforma para el desarrollo de software orientado a aspectos. *Proyecto Fin de Carrera. Escuela de Ingeniería Técnica Informática de Oviedo. Universidad de Oviedo*, 2010.

[Pinto01] M. Pinto, M. Amor, L. Fuentes, J.M. Troya. Run-time Coordination of Components: Design Patterns vs. Component-Aspect based Platforms. *Workshop Advance Separation of Concerns collocated with ECOOP'01*, 2001.

[Pinto02] M. Pinto, L. Fuentes, M. E. Fayad, J.M. Troya. Separation of Coordination in a Dynamic Aspect Oriented Framework. *Proc. of AOSD'02*, *ACM*, páginas 134-140, 2002.

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

[Popovici01a] A. Popovici, The impact of aspect-oriented programming on future application design. *Information and Communication Research Group Seminar, ETH Zurich*, 2001.

[Pothier08] Guillaume Pothier, Éric Tanter. Extending omniscient debugging to support aspect-oriented programming. *Proceed. of the 2008 ACM symposium on Applied computing (SAC,08)*, páginas 266-270,2008.

[Rainone08] Flavia Rainone. Dynamic AOP Tutorial Part I y Part 2. *http://jbossaop.blogspot.com.es/2008/06/dynamic-aop-tutorial-part-1.html http://jbossaop.blogspot.com.es/2008/07/dynamic-aop-tutorial-part-2.html*, 2008.

[Reactivity13] Reactivity. REM'13. *http://soft.vub.ac.be/REM13/*, 2013.

[Redhat12] RedHat. JBoss AOP Homepage. *http://www.jboss.org/jbossaop*, 2012.

[RedHat13] RedHat. JBoss AOP IDE. *http://docs.jboss.org/aop/1.1/aspect-framework/reference/en/html/aopide.html*, 2013.

[RedHat13b] RedHat. Dynamic AOP: Problem Method Call. *https://community.jboss.org/message/445460?_sscc=t#445460*, 2013.

[RedHat13c] RedHat. JBoss Community driven open source middleware. *http://www.jboss.org/overview/*, 2013.

[Roo08] A. de Roo, M.F.H. Hendriks, W. Havinga, P. Durr, L. Bergmans. *Compose*: a Language- and Platform-Independent Aspect Compiler for*

Composition Filters. First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008, 2008.

[Sadjadi04] Seyed Masoud Sadjadi, Philip K. McKinley, Betty H. C. Cheng, R. E. Kurt Stirewalt. TRAP/J: Transparent Generation of Adaptable Java Programs. *CoopIS/DOA/ODBASE*, páginas 1243-1261, 2004.

[Sadladi04b] Sadjadi, S.M., McKinley, P.K. ACT: An adaptive CORBA template to support unanticipated adaptation. *Proceed. of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, páginas 74-83, 2004.

[Salvaneschi13] Guido Salvaneschi, Joscha Drechsler and Mira Mezini. Towards Distributed Reactive Programming. *COORDINATION: 15th International Conference on Coordination Models and Languages*, 2013.

[Schult03] Wolfgang Schult, Peter Tröger. Loom.NET - an Aspect Weaving Tool. *Workshop on Aspect-Oriented Programming (ECOOP'03)*, 2003.

[Schröder-Preikschat06] Wolfgang Schröder-Preikschat, Daniel Lohmann, Fabian Scheler, Wasif Gilani, Olaf Spinczyk. Static and Dynamic Weaving in System Software with AspectC++. *Proceed. Of the 39 th Annual Hawaii International Conference on System Sciences (HICSS'06)*, 2006

[Ségura03] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, J. L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. *Proceed. of the 2nd international conference on Aspect-oriented software development, (AOSD '03)*, páginas 110-119, ACM, 2003.

[Spring12a] SpringSource. Reference Manual, Spring Framework, version 3.1. http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/, 2012.

[Spring12b] SpringSource. Reference Manual, Spring Framework version 1.3.2. http://www.springframework.net/doc-latest/reference/html/, 2012.

[Steimann06] F. Steimann. The paradoxical success of aspect-oriented programming. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, páginas 481-497, 2006.

[Stevenson08] A. Stevenson, S. MacDonald. Dynamic aspect-oriented load balancing in java RMI. *PDPTA: Parallel and Distributed Processing Techniques and Applications*, páginas 485-491, 2008.

[Strembeck06] M. Strembeck, Uwe Zdun. Definition of an Aspect-Oriented DSL using a Dynamic Programming Language. *Proc.of the Workshop on Open and Dynamic Aspect Languages (ODAL)*, 2006.

[Sun06] Sun Microsystems. Java Specification Request (JSR) 220. Enterprise Java Beans, version 3.0. *Java Persitence API. Java Community Process*, 2006.

[Suvee03] Davy Suvée, Wim Vanderperren, Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. *International Conference on Aspect Oriented Software Development (AOSD'03)*, páginas 21-29, 2003.

[Szyperski02] Clemens Szyperski, Dominik Gruntz, Stephan Murer. Component Software - Beyond Object-Oriented Programming – Second Edition. *Addison-Wesley / ACM Press,* 2002.

[Vasseur04] Alexander Vasseur. AOP Benchmark: AWbench. http://docs.codehaus.org/display/AW/AOP+Benchmark, 2004.

[Vanderperren04] W. Vanderperren, D. Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. *Proc. of Dynamic Aspects Workshop*, 2004.

[Vinuesa03] Luis Vinuesa, Francisco Ortín Soler. A Dynamic Aspect Weaver over the .NET Platform. *Metainformatics*, páginas 197-212, 2003.

[Vinuesa07] Luis Vinuesa. Dynamic Separation of Aspects by means of Language and Platform Neutral Computational Refection*. Tesis Doctoral. Universidad de Oviedo*, 2007.

[Vinuesa08] Luis Vinuesa, Francisco Ortin, José M. Félix, Fernando Álvarez. DSAW - A Dynamic and Static Aspect Weaving Platform. *International Conference on Software and Data Tecnologies (ICSOFT'08)*, 2008.

[Walker03] D. Walker, S. Zdancewic, J. Ligatti. A theory of aspects. *ACM International Conference on Functional Programming*, 2003.

[Zinky97] J. A. Zinky, D. E. Bakken, R. E. Schantz. Architectural support for quality of service for CORBA objects. *TAPOS, Vol. 3, Nº. 1*, páginas 55-73, 1997.