CAPÍTULO I

INTRODUCCIÓN

Contenidos del capítulo:

1.1 Contexto **3**

1.1.1 Diseño Basado en Componentes 5

1.1.2 Desarrollo de Software Orientado a Aspectos 6

1.2 Organización **8**

INTRODUCCIÓN

PRISMA es un modelo para la definición de arquitecturas software que unifica dos nuevas tendencias en el desarrollo de sistemas software: el diseño basado en componentes y el desarrollo orientado a aspectos. PRISMA permite la especificación de sistemas complejos, distribuidos y altamente dinámicos, y para ello proporciona un lenguaje de descripción de arquitecturas.

En este proyecto se estudia cómo implementar un modelo orientado a aspectos y basado en componentes como PRISMA preservando la reutilización de aspectos y componentes y utilizando para ello la plataforma .NET. Debido a que las arquitecturas PRISMA son dinámicas en tiempo de ejecución, pueden añadirse o eliminarse aspectos y alterar la configuración de la arquitectura añadiendo o eliminando componentes y/o conexiones entre ellos en tiempo de ejecución. Por ello, su implementación requiere mecanismos de reflexión y emisión de código que permitan modificar el funcionamiento del código en ejecución. Aunque la mayoría de tecnologías orientadas a aspectos han sido desarrolladas en la plataforma Java y es donde presentan un mayor grado de madurez, se ha descartado esta plataforma porque, aunque permite la reflexión de código, no proporciona mecanismos para la emisión dinámica de código. En cambio, la plataforma .NET, además de la reflexión, proporciona mecanismos para la emisión de código en tiempo de ejecución e introduce el concepto de metadatos, mediante los cuales es posible extender el comportamiento del lenguaje. Además, se ha podido comprobar que con la plataforma .NET es posible construir sistemas orientados a aspectos flexibles y dinámicos, sin necesidad de realizar cambios a la infraestructura interna de .NET *Framework*.

Como trabajo previo al desarrollo del modelo se ha realizado un análisis de las distintas propuestas de implementación orientadas a aspectos existentes en .NET, estudiando los diferentes mecanismos empleados para el entretejido de aspectos y código base y detectando las ventajas e inconvenientes de cada implementación. Teniendo en cuenta las conclusiones de dicho análisis, se ha construido desde cero una implementación del modelo PRISMA persiguiendo una equivalencia lo más cercana posible al lenguaje de descripción de arquitecturas, para así favorecer el proceso de compilación de las especificaciones PRISMA a .NET y su posterior evolución.

Como resultado de este proyecto, por una parte, se ha construido una primera versión de un *middleware* PRISMA que permite la implementación, carga, ejecución y evolución de modelos arquitectónicos PRISMA. Por otra parte, se han establecido las correspondencias entre el modelo PRISMA y la implementación en .NET, lo que permitirá en un futuro especificar los patrones de generación de código para la creación del compilador de modelos PRISMA.

1.1 Contexto

Los sistemas de información cada vez son más complejos debido al incremento de la competitividad en el mercado, al rápido desarrollo de la tecnología provocando un aumento de los requisitos no-funcionales y a la tendencia creciente hacia la movilidad, distribución e interconexión de recursos. Además, debido a que las empresas están sometidas a constantes cambios, los sistemas de información a su vez están sometidos a la evolución constante de sus requisitos. Éstas, entre otras razones, hacen que los entornos de desarrollo faciliten la construcción de aplicaciones con arquitecturas complejas, distribuidas, evolutivas y reutilizables.

Debido a la amplia oferta de servicios de desarrollo de software, y consecuentemente una mayor competitividad en el sector, las diferentes compañías de software se ven obligadas constantemente a minimizar el tiempo de desarrollo de software y los costes asociados a sus procesos de análisis, desarrollo y mantenimiento. Es por ello que cada vez cobra más importancia el desarrollar o importar elementos reutilizables y fáciles de mantener, con la finalidad de disminuir costes y tiempos de desarrollo.

En la actualidad existen numerosas herramientas CASE capaces de generar aplicaciones siguiendo el paradigma de la prototipación automática de Balzer [Bal85]. Se conocen como compiladores de modelos y son capaces de generar el código fuente y el esquema de la base de datos del sistema de información a partir de su esquema conceptual. La generación de código puede ser completa como lo hacen Oblog Case [Ser94] y OlivaNova Model Execution System (OO-Method/CASE [Pas97]), o parcial, como Rational Rose [RRose], Enterprise Architect [Sparx] y otros.

Sin embargo, el hecho de que la mayoría de estos compiladores de modelos sigan un enfoque orientado a objetos es un inconveniente, ya que por sí solo no permite abordar adecuadamente los requisitos de los sistemas software complejos actuales. Esto ha hecho surgir dos tendencias en el desarrollo de sistemas software, el Desarrollo de Software Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA) [Aosd]. No obstante, hasta ahora no existe ningún compilador de modelos que soporte la especificación de sistemas de información utilizando estas dos nuevas aproximaciones.

En cambio, el modelo PRISMA permite especificar sistemas de información complejos, distribuidos, dinámicos y reutilizables, utilizando para ello las dos tendencias de desarrollo DSBC y DSOA, obteniendo las ventajas de ambas aproximaciones. En un futuro, el modelo será parte de un marco de trabajo que permitirá la generación de código a partir de los modelos arquitectónicos especificados en dicho entorno (compilador de modelos PRISMA).

1.1.1 Diseño Basado en Componentes

A medida que ha ido aumentando la demanda de sistemas software más flexibles, adaptables, extensibles y robustos, también ha aumentado la demanda de nuevas metodologías de desarrollo de software. Las nuevas metodologías deben permitir la construcción de sistemas formados por la unión de componentes software altamente flexibles, codificados por diferentes desarrolladores en distintos momentos y favorecer la reutilización de dichos componentes.

Frente a las metodologías que implicaban un desarrollo del sistema prácticamente desde cero, surgen las metodologías basadas en componentes (*Component-Based Software Development – CSBD)*, provocando un cambio en los hábitos y expectativas del desarrollo de software. Las distintas herramientas de desarrollo de software y las tecnologías actuales han conseguido que en la actualidad los componentes sean la clave para la reutilización de bloques de código complejos. De este modo, el desarrollo de software basado en componentes permite reducir el tiempo de desarrollo y mantenimiento de las aplicaciones.

El proceso tradicional de desarrollo software no hace hincapié en la reutilización, no especifica cómo y cuándo se debe hacer uso de la reutilización para el diseño de sistemas complejos. Por ello, el DSBC propone un cambio en el proceso de desarrollo, en el que se incluye una etapa de identificación de elementos potencialmente reutilizables (y por tanto, posibles componentes), y de puntos del sistema donde se puedan incorporar componentes ya desarrollados. De este modo, se consigue un ahorro en costes y tiempo de desarrollo y un incremento en la calidad del código, pues dichos componentes han sido ya testeados, optimizados y se suponen libres de errores.

Aunque en la actualidad no existe un consenso sobre la definición de componente, en este proyecto se ha adoptado la definición que ha adoptado el modelo PRISMA, pues es la más genérica y que proporciona un mayor nivel de abstracción. Se define un componente como un artefacto desarrollado específicamente para ser reutilizado. De esta forma, un componente puede ser tanto un caso de uso como una clase o cualquier

elemento que surja durante el proceso de desarrollo susceptible de ser reutilizable, permitiendo que la identificación de componentes se pueda realizar en cualquier etapa del ciclo de desarrollo de software.

Por otra parte, a la hora de incorporar componentes a un sistema software, han de considerarse dos alternativas: o bien desarrollarlos facilitando su futura reutilización o adquirir componentes comerciales externos (COTS - *commercial-off-the-shelf*). Cada alternativa tiene sus pros y sus contras. Desarrollar componentes por la propia compañía de software tiene como inconveniente la necesidad de invertir el tiempo necesario para su desarrollo y pruebas. Sin embargo, no se crea una dependencia hacia otra compañía externa, como ocurre al adquirir y utilizar componentes COTS. Adquirir componentes externas tiene la ventaja de, a un precio público y razonable, obtener la funcionalidad necesaria con la certeza de estar libre de errores. Además, es una práctica común que el proveedor de los componentes COTS cuando detecta errores envíe los componentes actualizados, con lo que la probabilidad de fallos en dichos componentes se ve reducida.

Los beneficios que aporta el Diseño de Software Basado en Componentes se pueden resumir en los siguientes puntos:

- Reducción en el coste de desarrollo y tiempo de salida al mercado de la aplicación final, ya que los desarrolladores pueden construir los sistemas a partir de elementos reutilizables en lugar de construirlos desde cero.
- Aumenta la fiabilidad de los sistemas software desarrollados, pues los componentes incorporados suelen pasar por una serie de etapas de revisión e inspección en su proceso de desarrollo.
- Mayor mantenibilidad de los sistemas permitiendo que los nuevos componentes, de mejor calidad, reemplacen a los antiguos sin más que cambiar uno por otro.
- Aumenta la reutilización, pues los nuevos componentes desarrollados para un sistema concreto podrán ser reutilizados en cualquier otra aplicación.
- Mayor calidad final del sistema, pues los componentes son desarrollados por expertos en un determinado dominio de aplicación y los ingenieros del software se dedican a ensamblarlos adecuadamente, centrándose en la arquitectura de la aplicación final.

1.1.2 Desarrollo de Software Orientado a Aspectos

La Programación Orientada a Aspectos (*Aspect Oriented Programming, AOP)* surge como solución a ciertos problemas detectados en el modelo de programación orientado a objetos (POO). En el paradigma POO, toda tarea específica debe ser responsabilidad de una clase o de un pequeño número de clases agrupadas de alguna forma lógica. Sin embargo, existen ocasiones en

las que determinados servicios se utilizan en el seno de diversas clases y no tienen suficiente entidad para incluirlos en una clase específica, lo que acaba provocando repetición de código a lo largo de toda la aplicación. Ejemplos de estos servicios o bloques de código son los dedicados a la sincronización o a la optimización de los accesos a los recursos, a la persistencia de los datos, al tratamiento de excepciones, al registro de auditorías (logs), etc. Todos estos servicios o bloques de código son características o temas de interés dentro del sistema software (*concern*). La diseminación de estos *concerns* a través de varias clases son los denominados *crosscutting concerns*.

El objetivo del Desarrollo de Software Orientado a Aspectos (DSOA) es encapsular cada *crosscutting concern* en una entidad separada, el *aspecto*, de forma que se localicen los cambios relacionados con un *concern*. El DSOA está teniendo un gran auge en la comunidad informática debido a que su código modular consigue que los costes de desarrollo, mantenimiento y evolución del software se reduzcan. Por este motivo, cada vez es mayor la tendencia hacia la incorporación del concepto de aspecto en las primeras fases del ciclo de vida, como la etapa de requisitos y la de diseño.

En la Figura 1 puede observarse cómo el agrupar los diferentes *crosscutting concerns* en entidades separadas favorece el mantenimiento y la evolución de dicha funcionalidad a consecuencia de una mayor modularización del código.

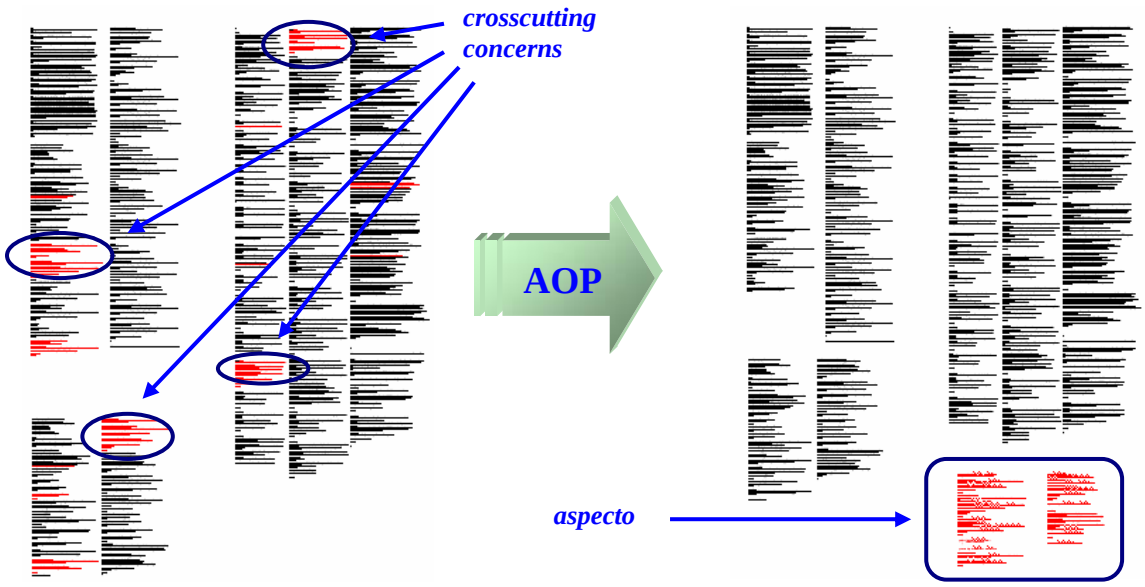

Figura 1 - Mejora de la modularidad con AOP

Un aspecto [Kic97] puede definirse como la encapsulación de un *crosscutting concern* en una unidad modular que, a través de la especificación de unos puntos de unión (*joinpoints*), será emparejado con el código de la aplicación (o código base, pues no contiene *crosscutting concerns*). Este enlace entre aspectos y código base, o *weaving*, producirá una aplicación final en la que el comportamiento definido por los aspectos se aplicará sobre el código base. Es

importante resaltar que el código base desconoce por completo la existencia de los aspectos, con la finalidad de preservar la independencia. Toda la información necesaria debe residir en el aspecto o en entidades adicionales que definan el proceso de *weaving*.

Como se ha señalado anteriormente, los aspectos pueden existir a nivel de diseño, si se entremezcla en la estructura de otras partes del diseño, o puede existir a nivel de programación, si aparece en otras unidades funcionales del programa. Además, dentro de los distintos modelos de aspectos existentes, se pueden observar dos tendencias, diferenciadas entre sí por el proceso de generación de código: los modelos estáticos y los modelos dinámicos. Los modelos estáticos (como AspectJ) generan un sólo componente en el que se encuentra mezclado el código funcional y el código de los aspectos, mientras que en los modelos dinámicos se generan distintas entidades para los distintos aspectos y el componente. La ventaja de los modelos dinámicos es la facilidad que proporcionan para incluir o eliminar aspectos durante el proceso de ejecución. En el siguiente capítulo se analizarán los distintos modelos de aspectos que existen en la actualidad y se describirá el proceso de generación de código que lo hace posible.

1.2 Organización

El proyecto se ha organizado en cinco capítulos. En primer lugar, tras presentar los objetivos y contexto del proyecto, se presentan las tendencias en el diseño de modelos arquitectónicos de software.

En el capítulo 2 se describe el estado del arte en cuanto a tecnologías orientadas a aspectos, centrándose principalmente en aquellas propuestas basadas en .NET. Se han analizado las principales aproximaciones que se pueden encontrar en la actualidad, comparando entre sí los distintos mecanismos utilizados para realizar la unión de aspectos y los nuevos conceptos que cada tecnología aporta a la AOP.

En el capítulo 3 se describe el modelo PRISMA, detallando los conceptos del modelo y el lenguaje de definición de arquitecturas.

El capítulo 4 explica de forma detallada la implementación que ha permitido transformar especificaciones PRISMA a la plataforma de desarrollo elegida, describiendo en primer lugar el diseño de la arquitectura, y en segundo lugar analizando cómo se han ido transformando los diferentes conceptos del modelo PRISMA a elementos del lenguaje destino.

Finalmente, en el capítulo 5 se resumen las correspondencias entre el modelo PRISMA y el código desarrollado, y se presentan las conclusiones y los trabajos futuros.