

UNIVERSIDAD POLITÉCNICA DE VALENCIA

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN



fiv

DSIC

ESTUDIO E IMPLEMENTACIÓN DE UN MODELO DE ARQUITECTURAS ORIENTADO A ASPECTOS Y BASADO EN COMPONENTES SOBRE TECNOLOGÍA .NET

Proyecto Final de Carrera

Febrero de 2005, Valencia

Realizado por:
Cristóbal Costa Soria

Dirigido por:
Jennifer Pérez Benedí, José A. Carsí Cubel

Agradecimientos

*A mis padres, por su apoyo incondicional, paciencia y cariño.
Sin ellos, yo no estaría hoy aquí.*

*A Sagrario, por su amor, alegría y abnegación.
Sin ella, nada para mí tiene sentido.*

*A Raúl, hermano inquieto e incorregible,
por su alegría y su forma de ser.*

*A mi tío Pepe, por haber despertado en mí
el deseo de aprender y la pasión por la ciencia.*

*A Jennifer y a Nour por sus ánimos, simpatía y amistad.
Gracias por su colaboración y todo su apoyo en este proyecto.*

*A Rafa, Jose Manuel y Jose Antonio,
mis compañeros pero también mis amigos.*

*A Pepe Carsí e Isidro Ramos por haber confiado en mí,
por compartir sus conocimientos y sus consejos.*

*A Artur, Manolo, Carlos, Elena, Nelly, Isabel y Javi,
por hacerme pasar tan buenos ratos.*

*A todos los miembros del grupo de investigación ISSI,
por su atención y colaboración.*

ÍNDICE GENERAL

ÍNDICE GENERAL.....	v
ÍNDICE DE FIGURAS.....	ix
ÍNDICE DE TABLAS.....	xi
1 INTRODUCCIÓN	1
1.1 CONTEXTO.....	4
1.1.1 Diseño Basado en Componentes.....	5
1.1.2 Desarrollo de Software Orientado a Aspectos	6
1.2 ORGANIZACIÓN.....	8
2 TECNOLOGÍAS ORIENTADAS A ASPECTOS	9
2.1 PRECEDENTES EN JAVA	11
2.1.1 AspectJ.....	11
2.1.1.1 Aspects	12
2.1.1.2 Join Points	13
2.1.1.3 Pointcuts	14
2.1.1.4 Advices.....	14
2.1.2 Hyper/J.....	15
2.2 PRIMERAS APROXIMACIONES EN .NET	18
2.2.1 CLAW (Cross Language Aspect Weaving).....	18
2.2.2 AOP#	20
2.3 ASPECTC#.....	20
2.4 JAsCo.NET.....	23
2.4.1 Funcionalidad.....	24
2.4.1.1 Aspectos y Hooks	24
2.4.1.2 Conectores	25
2.4.2 Modelo de Ejecución de JAsCo	26
2.4.3 Ventajas e Inconvenientes.....	27
2.5 LOOM.NET	28
2.6 RAPIER-LOOM.NET	31
2.7 SETPOINT!	34
2.8 OTRAS APROXIMACIONES	36
2.8.1 AOP.NET.....	36
2.8.2 Weave.NET y SourceWeave.NET	36
2.8.3 EOS.....	37
2.8.4 AspectDNG.....	37
2.9 CONCLUSIONES.....	37

3 PRISMA	39
3.1 EL MODELO PRISMA.....	41
3.2 VISIÓN ORIENTADA A ASPECTOS.....	42
3.3 VISIÓN ORIENTADA A COMPONENTES	44
3.3.1 Componentes	44
3.3.2 Conectores	45
3.3.3 Sistemas	46
3.4 LENGUAJE DE DEFINICIÓN DE ARQUITECTURAS	47
3.4.1 Definición de Tipos.....	48
3.4.1.1 Interfaces	48
3.4.1.2 Aspectos	49
3.4.1.3 Componentes y Conectores	55
3.4.1.4 Sistemas, Attachments y Bindings	58
3.4.2 Configuración Arquitectónica.....	61
3.5 OTROS ASPECTOS AVANZADOS DE PRISMA.....	63
4 IMPLEMENTACIÓN EN .NET	65
4.1 DISEÑO DE LA ARQUITECTURA.....	67
4.1.1 Introducción.....	67
4.1.2 El Middleware PRISMA.....	69
4.2 IMPLEMENTACIÓN DEL MIDDLEWARE.....	71
4.2.1 La clase <i>AsyncResult</i>	74
4.2.2 Interfaces.....	78
4.2.3 Aspectos.....	79
4.2.3.1 Correspondencia entre PRISMA y la implementación.....	79
4.2.3.2 Tipos de Aspectos.....	81
4.2.3.3 Modelo de ejecución.....	83
4.2.4 Componentes y Conectores.....	89
4.2.4.1 Modelo de ejecución.....	89
4.2.4.2 Gestión de Aspectos	94
4.2.4.3 Weavings	95
4.2.4.4 Puertos.....	109
4.2.4.5 Definición de Componentes y Conectores en .NET	116
4.2.5 Sistemas	118
4.2.5.1 Modelo de ejecución.....	118
4.2.5.2 Attachments.....	122
4.2.5.3 Bindings.....	130
4.2.5.4 Definición de Sistemas en .NET.....	138
4.2.6 <i>MiddlewareSystem</i>	140
4.3 RESUMEN DEL CAPÍTULO	148
5 CONCLUSIONES	151
5.1 CONCLUSIONES.....	153
5.2 TRABAJOS FUTUROS.....	154

BIBLIOGRAFÍA	157
APÉNDICE - LA PLATAFORMA DE DESARROLLO .NET	161
A.1 INTEROPERABILIDAD ENTRE LENGUAJES: EL CTS Y LOS METADATOS	162
A.2 EL CLR: LA MÁQUINA VIRTUAL DE .NET	163
A.3 LA LIBRERÍA DE CLASES	165
A.4 .NET REMOTING	166
A.5 ASPECTOS AVANZADOS DE .NET	167
A.5.1 Atributos	167
A.5.2 Delegados	167
A.5.3 Reflexión	168
A.5.4 Emisión de código	168
A.5.5 CodeDom	168

ÍNDICE DE FIGURAS

Figura 1 - Mejora de la modularidad con AOP.....	7
Figura 2 - Hyper/J.....	16
Figura 3 - Arquitectura CLAW y AOP#.....	19
Figura 4 - Compilador de AspectC#.....	21
Figura 5 - Ejemplo de descriptor de aspectos en AspectC#.....	22
Figura 6 - Ejemplo de weaving en AspectC#.....	22
Figura 7 - Ejemplo de Aspecto JAsCo.NET.....	25
Figura 8 - Ejemplo de conector JAsCo.NET.....	26
Figura 9 - Modelo de componentes de JAsCo.....	26
Figura 10 - Modelo de ejecución de JAsCo.....	27
Figura 11 - Interfaz de Loom.NET.....	30
Figura 12 - Especificación XML de un aspecto en Loom.NET.....	30
Figura 13 - Ejemplo de aspecto en Rapier-Loom.Net.....	32
Figura 14 - Ejemplo de Weaving en Rapier-Loom.Net.....	33
Figura 15 - Vista interna de un elemento arquitectónico PRISMA.....	42
Figura 16 - Vista externa de un elemento arquitectónico PRISMA.....	42
Figura 17 - Weavings entre aspectos.....	43
Figura 18 - Vistas de un Sistema PRISMA.....	46
Figura 19 - Esquema de funcionamiento de un protocolo.....	55
Figura 20 - Ejemplo BankSystem.....	60
Figura 21 - Aplicación de PRISMA sobre .NET.....	68
Figura 22 - Arquitectura del Middleware PRISMA.....	70
Figura 23 - Espacio de nombres de la implementación de PRISMA.....	73
Figura 24 - Esquema de funcionamiento de la clase AsyncResult.....	77
Figura 25 - La clase AsyncResult.....	78
Figura 26 - Clases SubprocessCollection y SubProcessClass.....	80
Figura 27 - Interfaz IAspectType y definiciones de tipos de aspectos.....	82
Figura 28 - Interfaz IAspect y clase AspectBase.....	84
Figura 29 - Esquema de ejecución de un Aspecto.....	86
Figura 30 - Interfaces IComponent, IConector y clase ComponentBase.....	90
Figura 31 - Estructura interna de un componente en .Net.....	92
Figura 32 - Clases asociadas al Weaving.....	99
Figura 33 - Clases InPort, OutPort y ListenerPort.....	110
Figura 34 - Interfaz ISystem y clase SystemBase.....	119
Figura 35 - Esquema de funcionamiento de los Attachments.....	124
Figura 36 - Clases asociadas al Attachment.....	126
Figura 37 - Esquema de funcionamiento de un Binding.....	131
Figura 38 - Clases SystemBinding y SystemBindingServer.....	134
Figura 39 - Clases asociadas al ComponentBinding.....	136
Figura 40 - Estructura LOC.....	140
Figura 41 - Diagrama de despliegue del middleware.....	142
Figura 42 - Clase MiddlewareSystem.....	143
Figura 43 - Proceso de compilación en .NET.....	164
Figura 44 - .NET Framework.....	165

ÍNDICE DE TABLAS

<i>Tabla 1 - Comparación entre las diferentes tecnologías AOP en .NET</i>	38
<i>Tabla 2 - Plantilla de un aspecto PRISMA</i>	49
<i>Tabla 3 - Sintaxis de Pi-Cálculo Poliádico</i>	52
<i>Tabla 4 - Plantilla de un componente/conector PRISMA</i>	56
<i>Tabla 5 - Plantilla de un sistema PRISMA</i>	59
<i>Tabla 6 - Plantilla de un modelo arquitectónico PRISMA</i>	61
<i>Tabla 7 - Resumen correspondencias PRISMA y la implementación C#</i>	148
<i>Tabla 8 - Correspondencias Aspecto PRISMA y la implementación C#</i>	148
<i>Tabla 9 - Correspondencias Componente PRISMA y la implementación C#</i>	149
<i>Tabla 10 - Correspondencias Conector PRISMA y la implementación C#</i>	149
<i>Tabla 11 - Correspondencias Sistema PRISMA y la implementación C#</i>	150
<i>Tabla 12 - Correspondencias Attachment PRISMA y la implementación C#</i>	150
<i>Tabla 13 - Correspondencias Binding PRISMA y la implementación C#</i>	150

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.

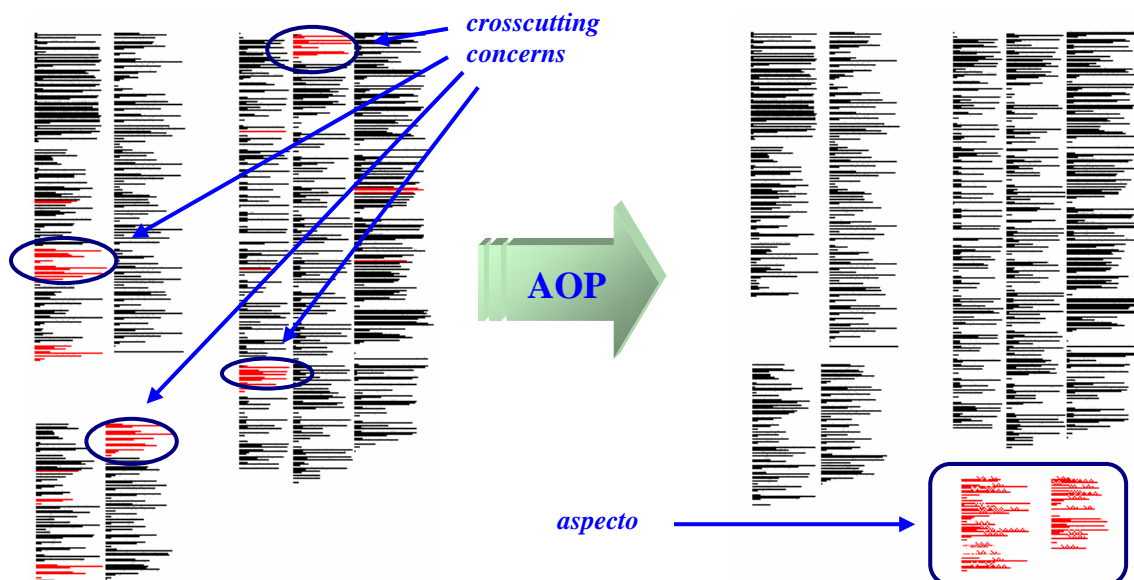


Figura 1 - Mejora de la modularidad con AOP

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.

TECNOLOGÍAS ORIENTADAS A ASPECTOS

Contenidos del capítulo:

2.1 Precedentes en JAVA	11
2.1.1 AspectJ	11
2.1.2 Hyper/J	15
2.2 Primeras aproximaciones en .NET	18
2.2.1 CLAW (Cross Language Aspect Weaving)	18
2.2.2 AOP#	20
2.3 AspectC#	20
2.4 JAsCo.NET	23
2.5 Loom.NET	28
2.6 Rapier-Loom.NET	31
2.7 SetPoint!	34
2.8 Otras aproximaciones	36
2.8.1 AOP.NET	36
2.8.2 Weave.NET y SourceWeave.NET	36
2.8.3 EOS	37
2.8.4 AspectDNG	37
2.9 Conclusiones	37

TECNOLOGÍAS ORIENTADAS A ASPECTOS

En este capítulo se describirá el estado del arte de las tecnologías orientadas a aspectos, en particular las diseñadas para la plataforma .NET. La Programación Orientada a Aspectos, conocida por el término inglés *Aspect Oriented Programming* (AOP) fue originariamente empleada en Java y es por ello que en este lenguaje es donde dicha tecnología ha ido evolucionando principalmente. Java es el lenguaje de programación que más herramientas de desarrollo dispone para dar soporte a la orientación a aspectos. El lenguaje por excelencia es *AspectJ*, y además de ser el que más avanzado se encuentra, es en el que se han basado la mayoría de aproximaciones existentes para otros lenguajes.

Actualmente, la AOP está teniendo un gran empuje. Por esta razón, a pesar de que .NET es una plataforma relativamente nueva, existen bastantes proyectos basados en ella que ofrecen soporte a AOP. Algunos de los más importantes son *AspectC#*, *JAsCo.NET*, *Rapier-Loom.NET* y *SetPoint!*, aunque aún se encuentran en fase de desarrollo.

En las siguientes secciones se definen las principales características de estas tecnologías, así como sus ventajas e inconvenientes.

2.1 Precedentes en JAVA

2.1.1 AspectJ

Desarrollado originariamente por el PARC¹ a finales de 1997, y financiado conjuntamente por el NIST² y el DARPA³, a finales de 2002 fue liberado como proyecto de desarrollo abierto en *eclipse.org*. En sus 6 años de vida, AspectJ [AsJ04] ha pasado de ser un prototipo de investigación a ser el lenguaje más utilizado por la comunidad de desarrolladores en AOP, y utilizado en sistemas en producción.

¹ Palo Alto Research Center, USA

² National Institute of Standards and Technology, USA

³ Defense Advanced Research Projects Agency, USA

AspectJ utiliza un compilador propio mediante el cual se realizan los enlaces entre aspectos y aplicaciones Java. AspectJ es una tecnología que extiende el lenguaje origen, es decir, el programador que necesite utilizar AOP necesariamente deberá aprender una sintaxis adicional para utilizar aspectos en su aplicación. Además, AspectJ es un enlazador de aspectos *estático*, porque el proceso de enlazado (*weaving*) se realiza en tiempo de compilación, y por tanto, no podrá modificarse en tiempo de ejecución.

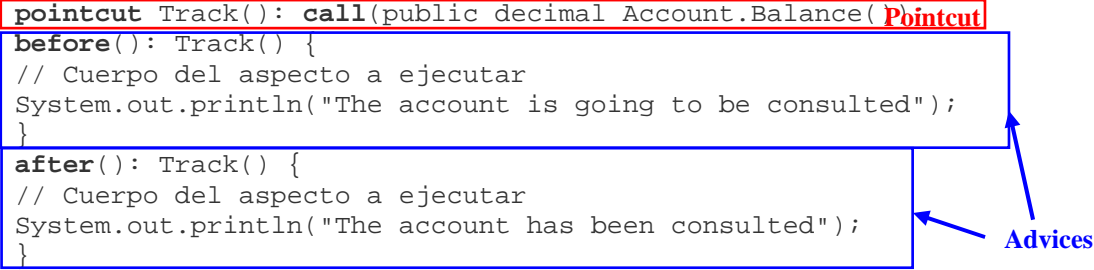
Los lenguajes basados en AspectJ incorporan las siguientes construcciones: *Join Points*, *Pointcuts*, *Advices* y *Aspects*. Los *joinpoints* están ligados al código base de la aplicación, mientras que los *aspects*, *pointcuts* y *advices* se definen en el código aspecto. Cada una de estas construcciones se explica en detalle a continuación:

2.1.1.1 Aspects

Un aspecto es una construcción del lenguaje que encapsula un *crosscutting concern*. Su definición es muy similar a una clase de Java, con la salvedad que además añade *advices* y *pointcuts*, estructuras sintácticas que definen cómo y a qué código se enlazarán los aspectos, respectivamente.

Un aspecto se enlaza con uno o varios métodos del código base, a través de los *pointcuts* (puntos de corte), y mediante los *advices* se especifica si la ejecución del código del aspecto será antes, después o en lugar de la ejecución del método indicado por el *pointcut*. En el siguiente ejemplo, cuya finalidad es auditar el acceso a una cuenta, se muestra cómo es un aspecto sintácticamente en Java, y se señalan las construcciones sintácticas que lo forman (*pointcuts* y *advices*):

```
Aspect AccountTracking {  
    pointcut Track(): call(public decimal Account.Balance(Pointcut  
    before() : Track() {  
        // Cuerpo del aspecto a ejecutar  
        System.out.println("The account is going to be consulted");  
    }  
    after() : Track() {  
        // Cuerpo del aspecto a ejecutar  
        System.out.println("The account has been consulted");  
    }  
}
```



La forma en que AspectJ combina (*aspect weaving*) los *concerns* con el código origen, es mediante el enlazado en tiempo de compilación. El código original es transformado por el compilador de AspectJ de forma que los *advices* son transformados en métodos estándar, cuya llamada es insertada en aquellos puntos del código origen indicada por los *pointcuts*. Si existen zonas del código que son resueltas dinámicamente, el código es dirigido a las zonas controladas por los *advices*.

2.1.1.2 Join Points

Un *Join Point* (punto de unión) es un concepto semántico introducido en *AspectJ* que se define como un punto bien definido en el flujo de ejecución de un programa. Una llamada a un método o un acceso a un atributo serían ejemplos de *joinpoints*. Existen diferentes tipos de *joinpoints* definidos en AspectJ. Los más utilizados son los que se identifican con:

- la llamada o invocación a un método o constructor,
- la ejecución del cuerpo de un método o constructor,
- los métodos *get* y *set* de propiedades,
- el tratamiento de una determinada excepción.

Un *joinpoint* debe considerarse como un punto en la ejecución de la aplicación susceptible de ser interceptado por los aspectos, es decir, como un punto de entrada donde podrá introducirse el código de los aspectos para alterar el flujo de ejecución normal del código base. Es importante tener en cuenta que cada *joinpoint* es distinto en tiempo de ejecución, tiene un contexto de ejecución diferente. Por ejemplo, un *joinpoint* correspondiente a la invocación de un método, será diferente según sea el objeto que lo llame.

Los *joinpoints* forman parte del código base y no necesitan de ninguna construcción explícita del lenguaje para especificarlos. A continuación se muestra la definición de una clase *Account* en Java con la definición de sus correspondientes atributos y métodos públicos, en la que se han señalado los diferentes *joinpoints* que podrían ser interceptados.

```
class Account {
    // Definición de variables
    private string ID;           // ID de la cuenta
    private float money;        // Dinero depositado en la cuenta

    // Definición de métodos
    // Constructor de la cuenta
    public Account(string ID, float money) { ← Posibles joinpoints
        this.ID = ID;
        this.money = money;
    }
    // Devuelve el saldo de la cuenta
    public float Balance() { ← Posibles joinpoints
        return money;           // Devuelve el dinero que hay en cuenta
    }
    // ... otros métodos
}
```

Como se puede observar, en la definición de una clase sencilla existen numerosos puntos de su ejecución que pueden ser interceptados (*joinpoints*): por ejemplo, la ejecución del cuerpo del constructor o del método *Balance* correspondería a dos *joinpoints* diferentes.

2.1.1.3 Pointcuts

Por tanto, una aplicación tendrá un conjunto de *joinpoints*, de los cuales tan sólo un pequeño subconjunto interesará interceptar. Para ello, se definen los *pointcuts* (puntos de corte) mediante los cuales se podrá seleccionar, del conjunto de todos los *Join Points* posibles, aquellos que se desean utilizar para asociarlos con un aspecto concreto. Es por esto que los *pointcuts* son los elementos de AspectJ que permiten definir el *weaving* (enlazado) de un aspecto con el código base.

Existe una gran variedad de *Pointcuts*, algunos de ellos son:

- **call**(<patron_método>): captura todos los *joinpoints* de llamadas a métodos o constructores cuya signatura coincida con el patrón especificado.

Por ejemplo, el *pointcut* definido como **call**(public decimal Account.Balance(*)), capturará todas las llamadas al método *Balance* de los objetos de *Account*, con cualquier número de argumentos y que retornen un valor *decimal*.

- **args**(<lista_tipos>): captura todos aquellos *joinpoints* cuyos argumentos concuerden, en orden y tipo, con la lista de tipos especificada.

Por ejemplo, el siguiente *pointcut* **args**(object, int, string) capturará todos aquellos métodos cuyos parámetros sean un objeto, un entero y una cadena, en ese orden.

- **target**(<tipo_clase>): selecciona aquellos *joinpoints* que pertenezcan a una instancia cuyo tipo es la clase especificada.

Por ejemplo, **target**(Account) capturará el conjunto de *joinpoints* que hagan referencia a instancias de la clase *Account*.

Además, también pueden utilizarse los operadores **&&** (and), **||** (or) y **!** (not) para permitir la combinación de varios *Pointcuts*.

Por tanto, mediante la siguiente declaración:

```
pointcut Track(): call(public decimal Account.Balance());
```

se define un *pointcut* cuyo nombre identificativo es *Track()*, que no toma ningún argumento, y selecciona los *joinpoints* correspondientes a la invocación del método *Balance* de la clase *Account* (habrá tantos como instancias diferentes de la clase *Account*).

2.1.1.4 Advices

Un advice define el código que deberá ser ejecutado en los *joinpoints* seleccionados por los *pointcuts*. La ejecución de dicho código estará

condicionada por el tipo de *advice* que lo contenga. AspectJ dispone de diferentes tipos:

- *before*: el código enlazado se ejecuta antes que cada *joinpoint*
- *after*: el código enlazado se ejecuta después de la ejecución de cada *joinpoint*. Existen dos subtipos: *after throwing* y *after returning*. El primero es utilizado cuando se lanza una excepción, y el segundo cuando se finaliza con normalidad la ejecución del *joinpoint*
- *around*: el código del aspecto reemplaza el código que se iba a ejecutar en el *joinpoint*.

Volviendo al ejemplo mostrado en el apartado 2.1.1.1, se puede observar cómo se han definido dos *advices*, uno de tipo *before* y otro de tipo *after*:

```
before(): Track() {
    // Cuerpo del aspecto a ejecutar
    System.out.println("The account is going to be consulted");
}
after(): Track() {
    // Cuerpo del aspecto a ejecutar
    System.out.println("The account has been consulted");
}
```

Cada uno de los *advices* define un código que deberá ejecutarse antes y después, respectivamente, de los *joinpoints* capturados por el *pointcut* *Track()*, definido en el apartado anterior.

Por tanto, y viendo el proceso de forma global, cuando en el flujo de ejecución del programa base se active uno o más de los *joinpoints* capturados por el *pointcut* *Track()* (siguiendo el ejemplo, la invocación de los servicios cuyo nombre sea *Balance*), se ejecutará en primer lugar el código del *advice* *before*. Después se ejecutará el código del método interceptado en la aplicación base, y cuando finalice su ejecución se ejecutará el código asociado al *advice* *after*. Una vez finalice, el flujo del programa volverá al punto en el que se había quedado la aplicación.

Las principales ventajas de esta aproximación son las sencillas extensiones del lenguaje Java realizadas con la potencia del modelo de *joinpoints*. Además, también ha asentado los diferentes tipos de enlace de aspectos, que como se verá a lo largo de este capítulo, ha influido en las demás aproximaciones. Para más información sobre AspectJ, puede consultarse la documentación completa y la descripción completa de la sintaxis en [AsJ03].

2.1.2 Hyper/J

Una alternativa a AspectJ es Hyper/J [Ossh00], ésta propone una forma diferente de agrupar los *crosscutting concerns*. La mayoría de las tecnologías orientadas a aspectos son paradigmas asimétricos: por una parte se consideran los objetos, y por otra parte los aspectos, como unidades a las que

se redirige el flujo de información desde los objetos. Por contra, Hyper/J es un paradigma simétrico, ya que considera a los aspectos como entidades de primer orden, al igual que los objetos. Propone un modelo de composición mediante el cual los *concerns* pueden ser integrados en un programa o en un componente, además de poder ser utilizados en cualquier etapa del ciclo de desarrollo de software.

Hyper/J define el término *hiperespacio* (*Hyperspaces*) como el conjunto de aquellos elementos del lenguaje que van a ser utilizados para la composición de *concerns*: pueden ser paquetes Java, interfaces, clases, operaciones o atributos. La composición de estos elementos en estructuras declarativamente completas⁴ forman los denominados *Hyperslices* (o *hipersecciones*), que permiten la encapsulación de código con una funcionalidad determinada en una estructura independiente, favoreciendo la reutilización y la independencia y limitando el impacto de los cambios. Cada *hyperslice* puede ser un aspecto o un objeto perteneciente a la lógica de negocio. En la Figura 2 puede observarse cómo los diferentes elementos del hiperespacio se agrupan en *hyperslices*, algunos de los cuales, como *Logging* son aspectos, mientras que otros son de la lógica de la aplicación (como *Employee*).

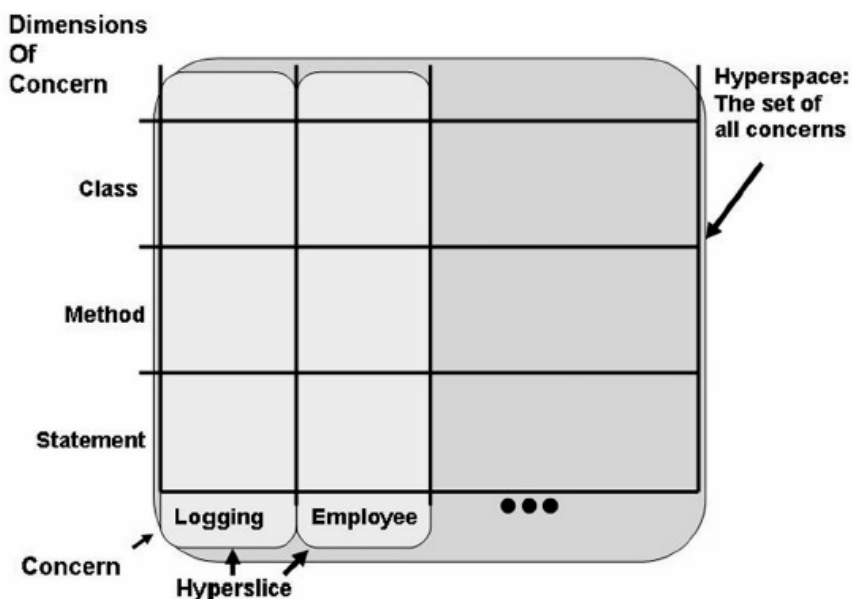


Figura 2 - Hyper/J

Finalmente, los *hypermodules* o *hipermódulos* incluyen un conjunto de *hyperslices* que se pueden componer entre sí, junto con un conjunto de reglas de composición que especifican cómo han de hacerlo. Pueden anidarse entre sí permitiendo la composición de estructuras más complejas.

⁴ Una estructura declarativamente completa es un elemento funcional que declara todo aquello que referencia. Por ejemplo, en Java podría corresponderse con un paquete de código que declara como abstracto aquellos métodos externos que referencia.

A continuación se muestra un ejemplo muy sencillo, cuyo objetivo es mostrar cómo se pueden combinar dos clases independientes entre sí mediante Hyper/J:

```
public class A {
    public void diHola() {
        System.out.println("Hola");
    }
}
public class B {
    public void diAdios() {
        System.out.println("Adios");
    }
}
```

La composición de ambas en Hyper/J:

```
hyperspace
  hyperspace demo
  composable class A;
  composable class B;

-concerns
  class A : Feature.hola
  class B : Feature.adios

-hypermodules
  hypermodule DemoHM
    hyperslices: Feature.hola, Feature.adios;
    relationships:
      mergeByName;
      equate operation Feature.hola.diHola, Feature.adios.diAdios
        into saluda;
      merge class Feature.hola.A, Feature.adios.B;
  end hypermodule;
```

Para realizar la composición entre ambas clases, en primer lugar se definen los paquetes donde se encuentran los elementos del hiperespacio. En este caso son las clases A y B del código original que desea combinarse. En la sección *concerns*, se define la asociación entre los elementos del hiperespacio (que pueden ser paquetes, clases o métodos) y los *concerns* que definen. En este caso, ambas clases pertenecen a la misma dimensión: *Feature*, pero definen distintos *concerns*. Todos los concerns de la misma dimensión y mismo nombre pertenecen a un mismo *hyperslice*. En este ejemplo, se tiene un *hyperslice* distinto para cada clase. Por último, la sección *hypermodules* define el módulo que se va a construir, que está formado por las *hyperslices* definidas anteriormente (*Feature.hola* y *Feature.adios*). El apartado de *relationships* define las reglas para combinar dichos *hyperslices*: *mergeByName* indica que los elementos con el mismo nombre en distintas *hyperslices* se fusionarán en una nueva unidad, mientras que *equate operation ... into* indica que las operaciones, aunque tienen distinto nombre, se fusionarán en una nueva unidad compuesta de nombre *saluda*, en este caso. El resultado de esta composición sería un paquete con un método *Saluda* que provocaría la ejecución del código asociado a *A.diHola()* y *B.diAdios()*.

Mediante este sencillo ejemplo, se pretende mostrar una pequeña parte de la potencia expresiva que Hyper/J posee. Existe también una versión prototipo que, de forma gráfica y a través de sencillos modelos, permite componer o modificar los diferentes *hiperslices*, que se traducen por las reglas de composición de Hyper/J. Por otra parte, a diferencia de AspectJ, Hyper/J trabaja con *bytecodes*, es decir, con módulos ya compilados. Por lo tanto, es posible trabajar con COTS (*commercial off-the-self*) de las cuales en la mayoría de los casos no se dispone del código fuente.

La principal aportación de esta tecnología es la consideración de los aspectos como entidades de primer orden y la facilidad para la integración de módulos que encapsulan diferentes *concerns* independientes entre sí. Todo ello sin necesitar extender el lenguaje. El lenguaje específico de Hyper/J sólo se utiliza para la composición de módulos. Dicha composición puede hacerse mediante el prototipo gráfico que proporciona Hyper/J, tal y como se ha mencionado anteriormente.

2.2 Primeras aproximaciones en .NET

Una vez vistas algunas de las más importantes aproximaciones AOP de Java, a continuación se presentan las diferentes aproximaciones existentes en .NET. Mientras que en Java las herramientas existentes son de gran calidad y su estado es bastante avanzado, en .NET aún se encuentran en fases tempranas de desarrollo. En esta sección se esbozan las características de las primeras aproximaciones, algunas ya abandonadas, que surgieron para .NET.

2.2.1 CLAW (Cross Language Aspect Weaving)

CLAW, desarrollado por John Lam⁵ en 2002 y cuya arquitectura se define en [Lam02], comparte muchas similitudes con AspectJ, como los *advices* y algunos *Join points*. CLAW se diferencia de AspectJ por su arquitectura, concretamente por la forma de enlazar los aspectos con código base. Los objetivos perseguidos por el proyecto eran:

- Evitar extender el lenguaje utilizado por el programador para definir los aspectos.
- Separar el código base (o código de aplicación) del código que define los aspectos.
- Utilizar un mecanismo sencillo para unir código base y aspectos.
- Permitir que cualquier lenguaje pueda formar parte del código base o de los aspectos.

⁵ Naleco Research Inc., Toronto, Canada – <http://www.iunknown.com>

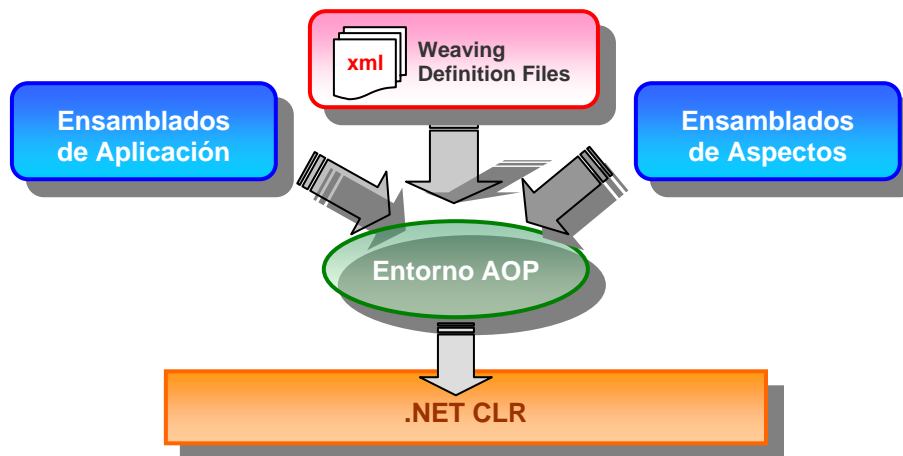


Figura 3 - Arquitectura CLAW y AOP#

La arquitectura se puede observar en la Figura 3. Los ensamblados de aplicación contienen todo el código de la aplicación y la lógica de negocio, mientras que los ensamblados de Aspectos contienen todo el código asociado a los aspectos a utilizar. El weaving se realiza en tiempo de ejecución de forma dinámica, uniendo dichos ensamblados. Esto posibilita que el código origen sea totalmente independiente de los aspectos, hasta el punto que el código fuente no necesita estar escrito en el mismo lenguaje que el código de los aspectos, o proceder del mismo proveedor de software. El weaving se define mediante unos ficheros XML (*weaving definition files*) que definen el mapeado entre los ensamblados base y los aspectos. En tiempo de ejecución, previamente a la ejecución del código por el compilador JIT (*Just In Time*) de .NET, CLAW modifica el código IL (*Intermediate Language*) del ensamblado base para que el código de los aspectos sea ejecutado de acuerdo con el weaving especificado. No obstante, dicha modificación crea una fuerte dependencia con el núcleo de .NET, lo que conlleva problemas de compatibilidad importantes con futuras versiones de la plataforma. Esto es debido a que la generación de código IL por el compilador de .NET, o la ejecución de dicho código en el CLR (*Common Language Runtime*, la capa más interna de la plataforma .NET) puede ser susceptible de cambios en futuras versiones. Como consecuencia, la herramienta debería actualizarse cada vez que saliese una nueva versión del *framework* .NET, lo cual puede ser una desventaja importante frente a otras herramientas.

Por otra parte, el código IL es generado por .NET exclusivamente para código manejado, es decir, para ensamblados que son interpretados por el CLR. Por ello, otra limitación es que CLAW no podrá unir entre sí ensamblados de código no manejado (o código nativo). Además, este modelo de ejecución también acarrea problemas de seguridad importantes, pues supone que cualquier desarrollador podría modificar el comportamiento de un ensamblado del cual el usuario confía, con tan sólo añadirle aspectos con código malicioso.

Es por estas razones que el proyecto fue abandonado en 2002, sin llegar a disponer de un prototipo estable. No obstante, al ser uno de los primeros

prototipos para .NET, ha influido en otras aproximaciones posteriores, como AspectC#, que han utilizado algunos conceptos del modelo de ejecución introducidos en CLAW. Por ejemplo, la interoperabilidad de los ensamblados y la no extensión del lenguaje utilizado por el desarrollador mediante el uso de la definición de los *weavings* en ficheros separados del código.

2.2.2 AOP#

El proyecto AOP#[Schü02], diseñado para Siemens Corporation, es casi idéntico al anteriormente expuesto en cuanto a sus objetivos y arquitectura. La diferencia entre ambos es que AOP# puede enlazar/desenlazar dinámicamente los aspectos al código base (“*Aspectual Polymorphism*”). El entorno AOP intercepta el código en tiempo de ejecución e inserta el código de los aspectos (a nivel de código IL de .NET) mediante la *Profiling API*⁶ de la plataforma .NET. En cuanto a la definición de los aspectos, y a diferencia del caso anterior, los aspectos se definen como clases que heredan de una clase base llamada *Aspect*. No obstante, al funcionar casi de la misma manera que el anterior, presenta los mismos inconvenientes: sólo permite tratar con código manejado, depende del núcleo de .NET y puede originar problemas de seguridad.

2.3 AspectC#

AspectC# [Kim02] es una aproximación orientada a aspectos que tiene puntos en común con CLAW y AOP# como: separar el código base y los aspectos, no extender el lenguaje y definir los *weavings* mediante un fichero XML. A diferencia de AOP#, no soporta *Polimorfismo Aspectual*, los aspectos son enlazados en tiempo de compilación y por tanto el código generado es estático. Actúa sobre el código fuente, en concreto en C#: no actúa sobre el CLR ni modifica código intermedio de .NET, por lo que no adolece de los problemas comentados anteriormente.

Está formado por un compilador cuyas entradas son el código base, el código de los aspectos, y una especificación XML (*Aspect Deployment Descriptor*) del *weaving* a realizar entre ambos. La estructura interna del compilador se puede observar en la figura siguiente:

⁶ La *Profiling API* de .NET permite, aparte de analizar el funcionamiento de aplicaciones sobre el CLR de .NET (código manejado), interceptar entradas/salidas a métodos y llamar a funciones personalizadas antes o después de llamar a dichos métodos. Puede encontrarse más información en: <http://msdn.microsoft.com/msdnmag/issues/01/12/hood/>

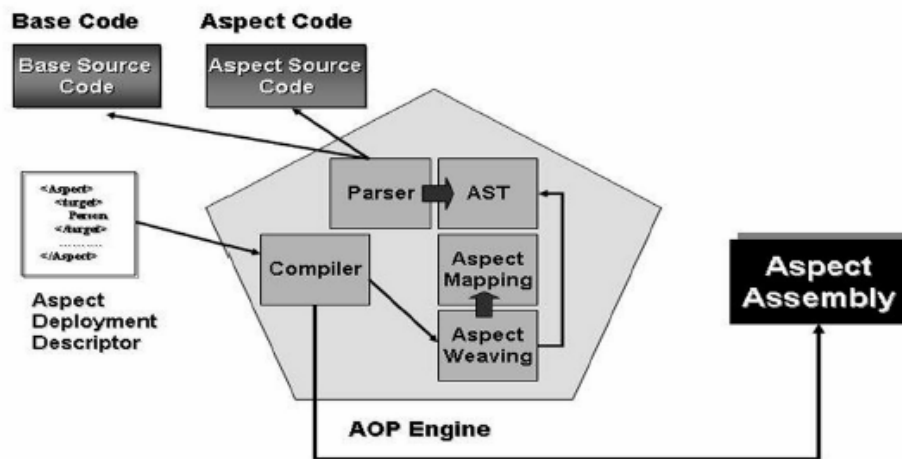


Figura 4 - Compilador de AspectC#

El código base es analizado por el *Parser*, que tras comprobar que la sintaxis es válida, genera un *Abstract Syntax Tree* (AST) en la que se plasma la estructura del código fuente. Con el código correspondiente a los aspectos se genera otro árbol AST de la misma manera. Estos dos árboles son fusionados teniendo en cuenta los *weavings* definidos en el *Aspect Deployment Descriptor*, generando un único árbol AST. Finalmente, mediante la tecnología *CodeDOM*⁷ de .NET, éste árbol es transformado en código ejecutable a través de cualquier compilador estándar de .NET, generando el ensamblado con la aplicación final.

A continuación se muestra un ejemplo de definición de *weavings* en AspectC# mediante XML (Figura 5), y el resultado tras generar el ensamblado y enlazar los aspectos (Figura 6):

⁷ CodeDOM (Code Document Object Model) es una tecnología ofrecida por .NET que permite el desarrollo automático de generadores de código fuente, mediante la representación de código fuente en forma de árboles abstractos de datos: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconCodeDOMQuickReference.asp>

```

<?xml version="1.0" encoding="utf-8" ?>
<Aspect>
  <TargetBase>C:\AspectCSDemo\CodigoBase</TargetBase>
  <AspectBase>C:\AspectCSDemo\CodigoAspectos</AspectBase>
  <Aspect-Method>
    <Name>AspectoFuncional1</Name>
    <Namespace>OtroNameSpace</Namespace>
    <Class>AspectoFuncional</Class>
    <Method>EmitirMensaje()</Method>
  </Aspect-Method>
  <Aspect-Method>
    <Name>AspectoFuncional2</Name>
    <Namespace>OtroNameSpace</Namespace>
    <Class>AspectoFuncional</Class>
    <Method>Suma(int a, int b)</Method>
  </Aspect-Method>
  <Target>
    <Namespace>Test</Namespace>
    <Class>Base</Class>
    <Method>
      <Name>Saluda()</Name>
      <Type>after</Type>
      <Aspect-Name>AspectoFuncional1</Aspect-Name>
    </Method>
    <Method>
      <Name>Saluda()</Name>
      <Type>before</Type>
      <Aspect-Name>AspectoFuncional1</Aspect-Name>
    </Method>
  </Target>
</Aspect>

```

Figura 5 - Ejemplo de descriptor de aspectos en AspectC#

```

namespace Test {
  using Ie.Tcd.AspectCSharp;
  using System;

  public class Base {
    private string test;

    public Base() {
      JoinPoint ThisJoinPoint = new JoinPoint("Base ", "Base",
        @"C:\AspectCSDemo\CodigoBase\Base.cs");
      // TODO: Add constructor logic here
    }

    public virtual void Saluda() {
      JoinPoint ThisJoinPoint = new JoinPoint("Base ", "Saluda",
        @"C:\AspectCSDemo\CodigoBase\Base.cs");
      Console.WriteLine("Soy el metodo EmitirMensaje de la clase AspectoFuncional");
      Console.WriteLine("Estoy introducido en el metodo: " + ThisJoinPoint.GS_MethodName);
      Console.WriteLine("");
      Console.WriteLine("Hola Mundo");
      Console.WriteLine("Soy el metodo EmitirMensaje de la clase AspectoFuncional");
      Console.WriteLine("Estoy introducido en el metodo: " + ThisJoinPoint.GS_MethodName);
      Console.WriteLine("");
    }

    // Method renamed to: funcion001a
    public virtual int funcion(int a, int b) {
      if(a+b > 10) {
        return -1;
      }
    }
  }
}

```

Definición de JoinPoint

Aspecto Funcional: Weaving BEFORE

Aspecto Funcional: Weaving AFTER

Figura 6 - Ejemplo de weaving en AspectC#

En la Figura 5 se puede observar la estructura del descriptor de aspectos, en el que se define en primer lugar la ubicación del ensamblado base y el de los aspectos (*TargetBase* y *AspectBase*). Cada método de los aspectos que se

quiera enlazar (el equivalente en *AspectJ* a los *advices*) se define mediante un nodo *AspectMethod*, y la definición de dónde se enlaza el aspecto con el método base (los *pointcuts*) se especifica mediante un nodo *<Target><Method>*. En este nodo se define para cada método del código base el aspecto que se va a aplicar y el tipo (*after*, *before* o *around*).

En la Figura 6 se detalla el código generado por el compilador de AspectC# antes de generarse el ensamblado correspondiente. Únicamente resaltar cómo se inserta el código correspondiente a los aspectos antes/después del código base, y cómo se introducen los *joinPoints* mediante los cuales el aspecto puede obtener información de dónde ha sido ubicado su código.

No obstante, el proyecto, al igual que los anteriores, ha sido abandonado en 2002, y también presenta algunos inconvenientes importantes. La principal desventaja es que utilizar un *parser* propio de código fuente, a no ser que incorpore toda la semántica de .NET existente para C#, limita la expresividad del programador, pues si utiliza estructuras no definidas en el *parser* el ensamblado no podrá generarse produciendo diversos errores de compilación. Además, si dicho *parser* incorporase toda la funcionalidad ofrecida por .NET, tarea inabordable pues equivaldría a generar un compilador casi idéntico al de la plataforma y estaría condenado a quedarse desfasado tan pronto apareciesen nuevas versiones de la plataforma con nuevas estructuras de código. Aparte de este inconveniente, AspectC# no es una plataforma apta para desarrollar aplicaciones AOP, debido a que se abandonó en un estado temprano de desarrollo. La documentación es escasa y el compilador no dispone de la funcionalidad mínima que cabría esperar, como permitir añadir librerías externas, tampoco soporta estructuras básicas como: anidaciones de clases, estructuras, delegados, eventos y atributos.

2.4 JAsCo.NET

JAsCo⁸ [Suv03] es una tecnología que, siguiendo la tendencia actual, trata de unir el Diseño Basado en Componentes (DSBC) y el Desarrollo de Software Orientado a Aspectos (DSOA). Diseñado originariamente para Java, cuya versión se encuentra en mayor estado de desarrollo, se está portando a dispositivos Java embebidos, se han creado una serie de plugins para *eclipse*, y dispone de un prototipo para .NET: JAsCo.NET [Vers03]. Este prototipo es el analizado en esta sección, y es importante resaltar que únicamente tiene implementadas las funcionalidades básicas.

⁸ En <http://ssel.vub.ac.be/jasco/files/JAsCo.ppt> se puede encontrar una presentación en diapositivas bastante detallada de gran parte de la funcionalidad ofrecida por JAsCo, aunque en su versión para Java.

2.4.1 Funcionalidad

A diferencia de las tecnologías AOP anteriores, introduce tres nuevos conceptos semánticos: los aspectos, los *hooks* y los conectores, que serán descritos en detalle a continuación.

2.4.1.1 Aspectos y Hooks

Un aspecto en JAsCo es un concepto que, como define la filosofía AOP, está concebido para ser reutilizable en diferentes contextos de ejecución. Además, JAsCo permite el *polimorfismo aspectual*: el *weaving* de los aspectos con el código base es completamente dinámico, es decir, pueden agregarse/eliminarse aspectos en tiempo de ejecución sin que para ello deba detenerse la aplicación. Esto es un requerimiento fundamental para una tecnología que ha sido diseñada para el campo de los Servicios Web, en el que la disponibilidad del servicio es prioritaria. Gracias a esto, si un aspecto que implementa la conexión con la Base de Datos no funciona adecuadamente, puede compilarse por separado y sustituir dinámicamente al que estaba en ejecución sin llegar a detener la ejecución de la aplicación servidora.

Un aspecto está formado por uno o varios *hooks*, o garfios, que definen la forma de enlazarse al contexto (los *pointcuts* de AspectJ), y el comportamiento frente a los distintos tipos de *advices*. Cada *hook* consta de dos partes:

- cuándo se activa (*pointcut*): Puede indicarse mediante el constructor del *hook* o mediante un método condicional *IsApplicable* que permite especificar condiciones más específicas y detalladas.

El constructor especifica la signatura del método o métodos que van a provocar su activación. Los nombres de dichos métodos son abstractos, es decir, no se corresponden con los nombres de los métodos con los cuales se enlazarán en el código base, aunque sí que tienen la misma signatura. Dichos métodos abstractos están definidos en los conectores, que establecen el mapeado entre el código base y los aspectos, con el objetivo de aumentar la reutilización de los aspectos y su independencia del código base al que se van a enlazar.

- que hará (*advice*): Permite indicar el comportamiento del aspecto para los distintos casos *after*, *before* o *replace*, cuyo significado es el mismo que en las otras aproximaciones AOP. Será en los conectores donde se especifique si el enlace con el método del código base es de un tipo o del otro.

JAsCo presenta la ventaja de tener un lenguaje de definición de aspectos muy rico, a diferencia de las otras aproximaciones. Los aspectos pueden combinarse entre sí mediante relaciones de agregación o herencia, pues son ciudadanos de primer orden, y por ello pueden formar estructuras más complejas. También puedan acceder al contexto en el que se están

ejecutando, definir métodos abstractos que sean posteriormente definidos en los conectores, utilizar la reflexión o incluso definir transiciones de estados.

A modo de ejemplo, en la siguiente figura se muestra la definición de un aspecto sencillo en JAsCo.Net:

```
namespace Aspectos {
    class AspectoTest {
        hook ReemplazarHook {
            Definición de Hook
            ReemplazarHook(string metodo()) {
                execute(metodo);
            }
        }
        Replace() {
            Comportamiento de un Aspecto
            MessageBox.Show("Metodo reemplazado!", "AspectoTest");
            Console.WriteLine("Metodo reemplazado por AspectoTest.ReemplazarHook");
            return null;
        }
    }
    hook InformacionHook {
        InformacionHook (string metodo()) {
            execute(metodo);
        }
        Replace() {
            string original = (string)calledmethod.Invoke(calledobject,null);
            original = original + " - Interceptado por un aspecto";
            return original;
        }
    }
}
}
```

Figura 7 - Ejemplo de Aspecto JAsCo.NET

Como se puede observar, un aspecto está formado por uno o varios *hooks*, que definen en su constructor qué método se va a interceptar, así como qué hacer frente a cada posible *advice*, donde se define el comportamiento del aspecto.

2.4.1.2 Conectores

Una vez vista la parte reutilizable del modelo, a continuación queda por describir la parte que se encarga de unir dichos elementos con el código base: los conectores. Éstos definen la unión entre un hook y uno o varios elementos del código base, que pueden ser: métodos, eventos, objetos o clases. La unión se realiza mediante el uso de estructuras sintácticas como los símbolos comodín “*” o “?”.

Asimismo, pueden definirse relaciones de precedencia entre *hooks* asociados a un mismo conector, encadenamientos o combinaciones más complejas (mediante el uso de clases específicas que definen cómo los diferentes *hooks* pueden combinarse entre sí). También se permite el uso de la reflexión para conocer dinámicamente el contexto en el cual se ubica el conector: obtener la lista de *hooks* del conector, añadir dinámicamente un objeto para realizar

cualquier acción, comprobar si el conector está habilitado o deshabilitado, decidir qué *hooks* aplicar o no, etc.

A modo de ejemplo, en la siguiente figura se muestra un conector, y cómo define un *hook* asociándolo con un método específico del código base. La línea `hook.Replace()` especifica qué *advice* debe aplicarse.

```

using System.IO;
using System.Collections;
using Aspectos;

namespace Conectores {

    static connector Reemplazar {

        AspectoTest.ReemplazarHook hook = AspectoTest.ReemplazarHook(
            sring ProyectoJASCO.frmBase.EmitirMensaje() );

        hook.Replace();

    }

}

```

Figura 8 - Ejemplo de conector JAsCo.NET

2.4.2 Modelo de Ejecución de JAsCo

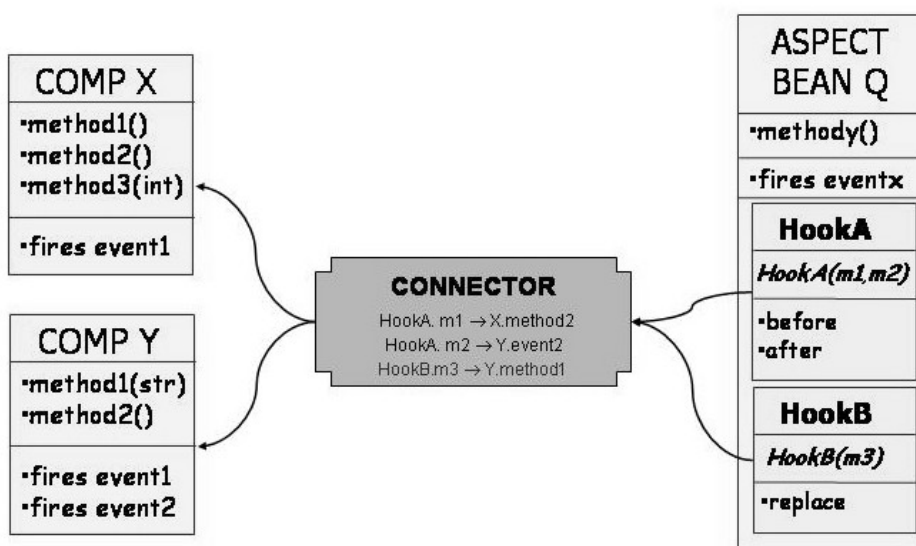


Figura 9 - Modelo de componentes de JAsCo

Una vez vistos los diferentes elementos que forman parte de JAsCo, en la Figura 9 se resume el modelo de componentes de JAsCo y la conexión entre ellos, desde los aspectos hasta el código base. Puede observarse cómo un aspecto Q, que representa una determinada funcionalidad, está formado por dos *hooks* que contienen los diferentes tipos de *advices* (*before*, *after* y *replace*). Cada uno de los *hooks* intercepta unos métodos determinados: *m1*, *m2* para el *hookA* y *m3* para el *hookB*. No obstante, el conector es el que

realmente establece el mapeado entre los métodos del código base con dichos *hooks* y permite separar la implementación de los aspectos del código base.

Para enlazar los conectores de forma dinámica con los componentes del código base en ejecución se utiliza el Registro de Conectores de JAsCo. Todos los componentes de la aplicación base son adaptados (se insertan llamadas a funciones externas (*traps*) a cada método público) para que las ejecuciones de sus métodos sean reenviadas a este Registro, el cual contiene una base de datos con todos los conectores cargados en el sistema y detecta qué *hooks* se pueden aplicar o no, en función del contexto de ejecución.

Por tanto, la plataforma de ejecución de JAsCo queda formada por los siguientes componentes:

- El compilador de aspectos (ficheros *.asp*) y el compilador de conectores (ficheros *.con*): se encargan de compilar el código con las definiciones de aspectos y conectores, respectivamente.
- *DotNetTransformer*: es la aplicación que se encarga de adaptar los ensamblados en .NET para que sean soportados por JAsCo, que se reduce a insertar *traps* en cada una de las ejecuciones de métodos publicos.
- *Introspector*: es la plataforma de ejecución de JAsCo, que permite insertar, eliminar, habilitar, o deshabilitar conectores en tiempo de ejecución a una aplicación determinada.

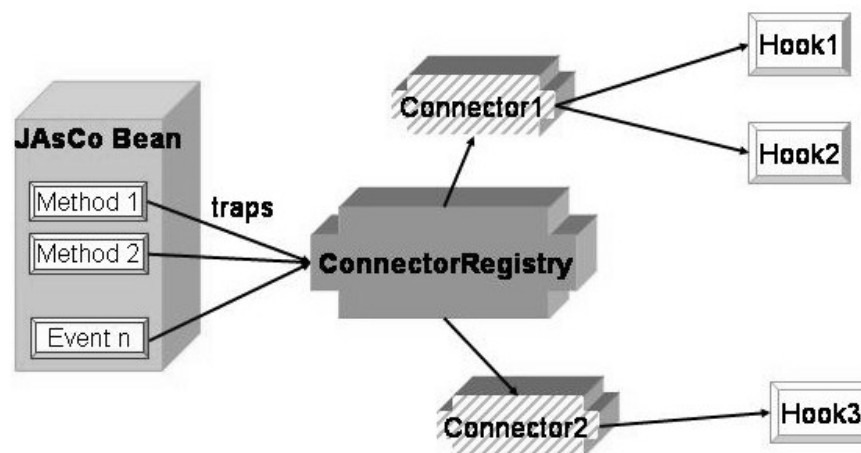


Figura 10 - Modelo de ejecución de JAsCo

2.4.3 Ventajas e Inconvenientes

JAsCo es una aproximación que intenta unificar las corrientes orientadas a Aspectos y basadas en Componentes, añadiendo el concepto de Conector. Además, dispone de un lenguaje muy expresivo: permite la combinación de aspectos, la utilización de mecanismos de herencia y una alta reutilización. Permite la incorporación de aspectos en tiempo de ejecución sin necesidad de

detener la aplicación objetivo y no necesita el código fuente del código base (por lo que no afecta a los COTS).

No obstante, también tiene inconvenientes a tener en cuenta:

- Extiende el lenguaje utilizado con nuevas estructuras sintácticas
- El registro de conectores se puede realizar por código mediante llamadas a la API del *Introspector*, pero esta funcionalidad no está aún implementada en .NET.
- La versión disponible para .NET es aún muy prematura y sólo incluye la funcionalidad básica. Además, los compiladores de componentes y conectores son incompletos en cuanto a flexibilidad sintáctica (por ejemplo, para definir métodos no se permite situar las llaves en la línea siguiente a la que define el método, y esto no está documentado). Tampoco se especifica qué funcionalidad del modelo completo (la versión de Java) se encuentra ya implementada, lo que dificulta su aprendizaje y uso.

2.5 Loom.NET

Loom.NET [Sch00], desarrollado por Wolfgang Schult y Andreas Polze en la Universidad de Potsdam, Alemania, se diferencia de las demás tecnologías en la forma de definir los aspectos. Mientras que las analizadas hasta ahora utilizaban un lenguaje extendido o el mismo lenguaje en el que se programaba la aplicación base, Loom.Net utiliza plantillas de código reutilizables.

Estas plantillas o patrones en realidad son código C# con etiquetas que indican dónde se situarán las definiciones de clases, métodos y propiedades. Estas etiquetas serán sustituidas por los elementos correspondientes del código base cuando el usuario enlace las plantillas con el código fuente. Como ejemplo, a continuación se muestra una plantilla para definir el constructor de una clase, que se limita a mostrar un mensaje por pantalla:

```
/*[PROTECTION]*/ /*[CLASSNAME]*/( /*[PARAMDECLARATION]*/ ) :
    base (/*[PARAMLIST]*/) {
    System.Windows.Forms.MessageBox.Show("Se ha llamado al constructor
    de ClaseAux");
}
```

El significado de las etiquetas es bastante intuitivo:

- [Protection] indica el nivel de protección de la clase
- [Classname] indica el nombre de la clase
- [ParamDeclaration] indica la definición de los parámetros del constructor
- [ParamList] es una lista de tipos de parámetros

Otro posible ejemplo sería la definición de un método, cuya apariencia es la siguiente:

```
public new virtual /*[RESULTTYPE]*/ /*[METHODNAME]*/ (
/*[PARAMDECLARATION]*/ ) {
    /*[RETVALINIT]*/
    System.Windows.Forms.MessageBox.Show("Ejecucion del aspecto en
el contexto de /*[METHODNAME]*/" );
    /*[RETVALASSIGN]*/base./*[METHODNAME]*/(/*[PARAMLIST]*/) + " "
+ /*[CADENA]*/;
    System.Windows.Forms.MessageBox.Show("El mensaje ya se ha
enviado hacia el cliente" );
    /*[RETVALRETURN]*/
}
```

Este método se limita a mostrar un mensaje que indica qué método se está ejecutando y a componer una cadena que será devuelta por el método. Las etiquetas utilizadas tienen el siguiente significado:

- [ResultType] indica el tipo de retorno del método
- [MethodName] indica el nombre del método definido
- [RetValInit] se sustituye por la inicialización del valor de retorno
- [RetValAssign] se reemplaza por una asignación a la variable de retorno
- [RetValReturn] equivale a devolver el contenido de la variable de retorno.

La sustitución de las etiquetas por el código correspondiente se realiza mediante las funcionalidades ofrecidas por .NET para expresiones regulares, ubicadas en el espacio de nombres *System.Text.RegularExpression*. El compilador sustituye todos los patrones encapsulados por los comentarios de C# `/*` y `*/` por el código necesario del código base. De esta manera, los aspectos se definen mediante patrones de código reutilizables.

Loom.NET no trabaja sobre código fuente base, sino que trabaja a nivel de ensamblados. Mediante una aplicación gráfica (ver Figura 11 - Interfaz de Loom.NET) el usuario elige el ensamblado base, y por reflexión se obtienen todos los métodos y propiedades públicas de dicho ensamblado. Los aspectos son cargados por la aplicación desde una librería de plantillas. Para añadir un nuevo aspecto a dicha librería, el desarrollador ha de añadir manualmente la definición XML del aspecto (ver Figura 12 - Especificación XML de un aspecto en Loom.NET) en la que se especifican parámetros adicionales que la plantilla pueda requerir. Después el usuario enlaza los aspectos con los métodos o clases del ensamblado, por reflexión se rellenan las plantillas de los aspectos con el código necesario y finalmente, se genera el ensamblado.

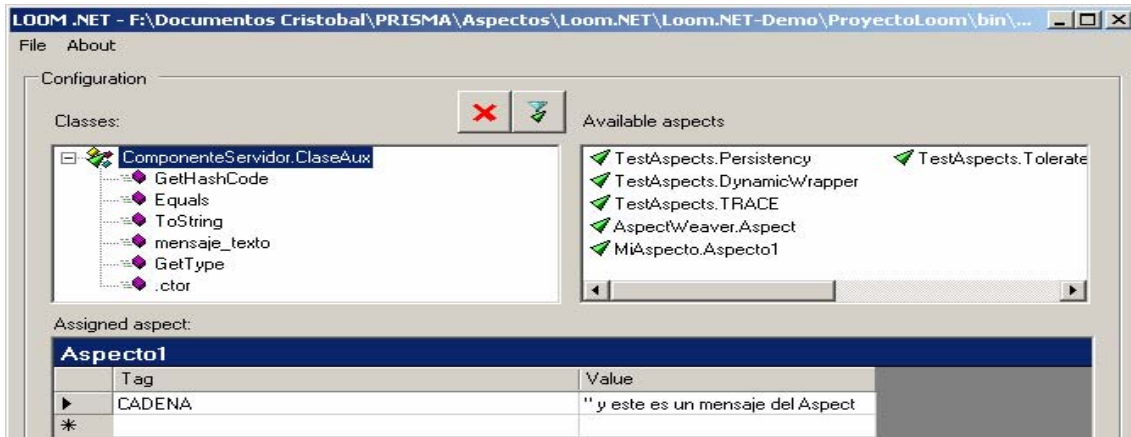


Figura 11 - Interfaz de Loom.NET

```
<Aspect Name="MiAspecto.Aspecto1" Description="Simplemente engancha una cadena a otro metodo">
  <Tag Type="file" Name="CTOR">ctor_MiAspecto-Cadena.tpl</Tag>
  <Tag Type="file" Name="METHOD">method_MiAspecto-Cadena.tpl</Tag>
  <Tag Type="text" Name="BASECLASS"> /*[BASENAMESPACE] */./ /*[CLASSNAME] */</Tag>
  <Tag Type="parameter" Name="CADENA">" y este es un mensaje del Aspecto ;-)"</Tag>
</Aspect>
</Templates>
```

Figura 12 - Especificación XML de un aspecto en Loom.NET

Finalmente, el código generado tras enlazar los aspectos con el ensamblado base(etapa previa a la generación del ensamblado), es el siguiente:

```
/// _____
/// <autogenerated>
///   This class is generated by LOOM .NET
///   (C) 2001-2003 by Wolfgang Schult
///   http://www.dcl.hpi.uni-potsdam.de
///
///   Changes of this file may cause incorrect
///   behavior and will be lost on next generation.
/// </autogenerated>
/// _____

namespace Aspecto1Proxy {
  public class ClaseAux:ComponenteServidor.ClaseAux {
    public ClaseAux(): base() {
      System.Windows.Forms.MessageBox.Show("Se ha llamado al
        constructor de ClaseAux");
    }

    public new virtual System.String mensaje_texto() {
      System.String _RetVal=null;
      System.Windows.Forms.MessageBox.Show("Ejecucion del aspecto en
        el contexto de mensaje_texto" );
      _RetVal=base.mensaje_texto() + " " + " y este es un mensaje del
        Aspecto =====;-)";
      System.Windows.Forms.MessageBox.Show("El mensaje ya se ha
        enviado hacia el cliente" );
      return _RetVal;
    }
  }
}
```

Como se puede observar, mediante mecanismos de herencia se genera una clase derivada (*Aspecto1Proxy.ClaseAux*) de la clase base (*Componente Servidor.ClaseAux*) que sobrecarga los métodos públicos de la clase base.

No obstante, esta aproximación está más orientada a la definición de *proxies* que a los aspectos concebidos como *concerns*. Como se puede observar en el código anterior, el compilador no enlaza los aspectos con el código base, sino que genera un nuevo componente que encapsula a la clase correspondiente del código base e implementa el código del aspecto. El cliente de la aplicación llamará a esta clase *proxy* generada, y en función del comportamiento definido para el aspecto, decidirá si llama o no al código base. Además, presenta limitaciones importantes como que sólo se puede enlazar un aspecto por clase (o método), y además no permite la conexión / desconexión dinámica de aspectos (*polimorfismo aspectual*).

Por ello, se puede afirmar que esta aproximación no es una AOP estricta como se ha visto hasta ahora, sino que únicamente se limita a encapsular código para integrar los aspectos.

2.6 Rapier-Loom.NET

Por otra parte, Rapier-Loom.NET [Sch02] sí que está diseñado para soportar el polimorfismo aspectual, y además no extiende el lenguaje.

En primer lugar, los aspectos se definen en cualquier lenguaje de .NET, mientras que el código base puede ser código fuente o un ensamblado. Cada aspecto se corresponde con una clase que hereda de *Loom.Aspect*, y mediante el uso de atributos se declaran las siguientes propiedades:

- Atributos de Enlace (*Interweaving*):

Permiten señalar la inclusión/exclusión de métodos de la clase objetivo con la que se va enlazar el aspecto (corresponderían a los *pointcuts* de AspectJ). El filtrado de métodos puede realizarse por el nombre, por el tipo de la clase a la que pertenece, por la interfaz que implementa o por atributos que tenga definidos.

- Connection Points:

Indican la forma de enlazarse al método (*advices* de AspectJ). Permite los típicos operadores *before*, *after*, *instead*, además de *after returning* y *after throwing*. Con *After returning* sólo es ejecutado el aspecto si el método devuelve un valor y en el caso de *After throwing* el aspecto es ejecutado si es lanzada una excepción.

- Acceso al contexto:

En el caso de *weavings instead*, se permite al aspecto acceder al contexto en el que se está ejecutando, con lo que es posible acceder a la

instancia del objeto en el que se iba a ejecutar el método, cambiar los parámetros que se pasarán al método e invocarlo mediante la sentencia *invoke*, etc.

A modo de ejemplo, se muestra a continuación un fragmento de código con la definición de un aspecto. En él se puede comprobar cómo se define el *ConnectionPoint* para indicar que el método que lo sigue se va a enlazar con cualquier método que se llame “IniciarFuncionalidad”, así como el tipo de *weaving* que se va a realizar, mediante el atributo `[Call(Invoke.Instead)]`. También puede observarse cómo se accede al contexto del objeto en el que se ejecute el aspecto para cambiar o consultar propiedades, `Context.Instance` o `Context.Invoke` para ejecutar el método que se iba a ejecutar y modificar el valor que va a retornar.

```
// Hereda de Loom.Aspect
public class AspectoFuncional : Loom.Aspect {
    const string version = "AspectoFuncional v.01";

    // Estos atributos indican el nombre del método con el que debe enlazarse
    // (la clase se especificará en el Weaving)
    [Loom.ConnectionPoint.Include("IniciarFuncionalidad")]
    [Loom.Call(Invoke.Instead)]
    // El código del aspecto propiamente dicho
    public string Identificarme() {
        string cadena;

        MessageBox.Show("Estoy en el metodo AspectoFuncional.Identificarme" +
            " y voy a llamar al CodigoBase", "AspectoFuncional");

        // Podemos acceder al contexto del objeto al que estamos enlazados
        // Accedemos a la clase base para modificar un atributo
        ((ClaseBase) Context.Instance).nombre = ";;Me han cambiado el nombre!!!";
        // Llamamos otra vez al código base para observar el resultado
        cadena=(Context.Invoke()).ToString() ;

        MessageBox.Show("Tras llamar a ClaseBase me ha devuelto: " + cadena,
            "AspectoFuncional");

        return "Y ahora me presento. Soy: " + version ;
    }

    // Ejemplo de función accesible desde fuera (sin weaving)
    // De esta forma, el aspecto puede proporcionar funcionalidad adicional que pueda
    // ser requerida por otras clases.
    public string Informacion() {
```

Figura 13 - Ejemplo de aspecto en Rapier-Loom.Net

En segundo lugar, permite instanciar dinámicamente los aspectos (y eliminarlos) en tiempo de ejecución gracias a que el *weaving* con el código base se realiza por código, a diferencia de las aproximaciones anteriores, que en su mayoría requerían de un compilador adicional para enlazar los aspectos con el código base. Para ello, define un nuevo operador *Weaver.CreateInstance* que sustituye al operador *new* en la creación de objetos, y que permite asociar una clase, método o propiedad a un aspecto. La sintaxis de este método es la siguiente:

```
public static object CreateInstance(
    Type classtype,
```

```

    object[] args,
    Aspect[] aspectarr
);

```

El primer parámetro es el tipo del objeto del código base que se quiere crear, el segundo parámetro contiene la lista de argumentos que requiere el constructor de dicho objeto y el tercer parámetro define la lista de aspectos que se quiere enlazar a dicho objeto. Como resultado, devuelve una instancia del objeto interceptado por los aspectos.

Para ilustrar esto, en la siguiente figura se muestra el código correspondiente en el que se realizaría el *Weaving* entre los aspectos y el código base.

```

// Aquí realizamos el Weaving con los Aspectos
public string LlamarComponenteConAspectos() {
    Aspect[] listaAspectos;
    string cadena;

    // Primero creamos los dos aspectos
    Aspectos.AspectoDistribucion dis = new AspectoDistribucion();
    Aspectos.AspectoFuncional asp = new AspectoFuncional();
    // Los metemos en un vector
    listaAspectos = new Aspect[2]; listaAspectos[0]=dis; listaAspectos[1]=asp;

    // Ahora enlazamos los aspectos con la clase Base
    // El orden de llamadas sigue el orden del vector, de forma que sólo se
    // inicia el segundo si el primero le pasa el control al método que sustituye.
    ComponenteEstructural.ClaseBase obj =
        (ComponenteEstructural.ClaseBase) Loom.Weaver.CreateInstance(
            typeof(ComponenteEstructural.ClaseBase), null, listaAspectos);

    // Ejecutamos el método de la clase Base que tiene código enlazado
    return obj.IniciarFuncionalidad();
}

```

Figura 14 - Ejemplo de Weaving en Rapier-Loom.Net

Además, también permite enlazar varios aspectos entre sí, a diferencia de Loom.Net, de forma que los aspectos puedan comunicarse entre sí, saber si existen determinados aspectos enlazados al método actual, ejecutarse unos antes de otro, etc. Por ejemplo, en el siguiente código se puede observar cómo un aspecto obtiene por reflexión los aspectos que están enlazados al mismo método al que él está enlazado y acceder a propiedades o métodos:

```

public class AspectoDistribucion : Loom.Aspect {
    // ...
    Aspect[] listaAspectos = Weaver.GetAspects(Context.Instance,
        typeof(Aspectos.AspectoFuncional));
    ((Aspectos.AspectoFuncional) listaAspectos[0]).Informacion2("hola");
    // ...
}

```

No obstante, todo método del código base susceptible de tener aspectos asociados debe definirse como *virtual* o ser llamado a través de una interfaz. Esto es debido a la forma en que Rapier-Loom.Net realiza el entretejido entre código base y aspectos. En el CLR de .NET, por cada clase existe una tabla que almacena los punteros a los métodos que define. Con el objetivo de

permitir la sobrecarga de métodos en la herencia, los métodos definidos como virtuales se almacenan en dicha tabla con un indicador que permite su modificación, para que las clases derivadas puedan sustituir la referencia de dicho método con el puntero al método sobrecargado. Rapier-Loom.Net aprovecha este mecanismo para que dichas entradas apunten a los métodos de los aspectos. Por ello, otra limitación derivada de esto, es que tampoco pueden enlazarse aspectos con métodos estáticos. Los autores recomiendan que, para superar estas limitaciones, utilizar la idea de *proxy* implementada en Loom.Net.

Además de este pequeño inconveniente, los aspectos no pueden desenlazarse dinámicamente sin detener la ejecución del objeto que los tiene asociados. Para ello, el objeto ha de ser vuelto a crear. Por otra parte, no hay una separación clara de los *weavings* y los aspectos, sino que los *pointcuts* (o connection points) son definidos en el mismo código del aspecto, degradando la reutilización de los aspectos.

2.7 SetPoint!

SetPoint! [SetP04] nace como Proyecto de Tesis de Licenciatura en la Universidad de Buenos Aires, y pese a encontrarse en fase muy prematura, ofrece un enfoque totalmente distinto a los vistos anteriormente. Basándose en la idea de la Web Semántica, introduce el concepto de *Semantic Pointcuts*, en un intento de unir la programación orientada a aspectos y la semántica para realizar el enlazado entre aspectos y código base.

El objetivo perseguido es etiquetar de alguna forma el código base para que los aspectos se enlacen (estática o dinámicamente) mediante predicados lógicos. Para ello, se define el término *Descriptor Semántico* (*setPoint*) como aquella marca en el código que permite asignarle una característica particular, de forma equivalente a los *Join Points* de AspectJ. Este Descriptor Semántico equivaldría a un atributo de .NET.

De esta forma, los *pointcuts* pasan a ser predicados lógicos sobre descriptores, en lugar de utilizar mecanismos de filtrado basados en la sintaxis del código como los vistos hasta ahora, que creaban una dependencia sintáctica del weaving con el código base. Por ejemplo, en las otras aproximaciones el *weaving* se crea en la mayoría de los casos en base al nombre del método a enlazar, mediante el uso o no de comodines. Si quisiéramos enlazar todos los métodos “Get” del código base (aquellos que devuelven los valores de las variables privadas de una clase) para auditar su acceso mediante un aspecto, en Rapier-Loom.Net se haría de esta forma:

```
// CODIGO DEL ASPECTO
[Loom.ConnectionPoint.Include("Get*")]
public void AuditGetterMethods() { /* ... */ }
```

Mientras que en *SetPoint* se traduciría en el predicado lógico: “Asociar el aspecto *AuditGetterMethods* a todos los métodos etiquetados como *[GetterMethod]*”, cuya traducción a .NET⁹ podría ser la siguiente: en primer lugar se insertarían los *SetPoints* en los métodos “Get” del código base,

```
// CODIGO BASE
// ...
[GetterMethod]
public int GetStatus() { /* ... */ }
// ...
```

y después se definirían los *pointcuts* en el aspecto:

```
// CODIGO DEL ASPECTO
[SetPoint.SelectAll("GetterMethod")]
public void AuditGetterMethods() { /* ... */ }
```

Con esto, se consigue una mayor independencia del código, pues si se cambia el código base sin eliminar los *SetPoints*, el enlace de aspectos no se vería afectado.

Un inconveniente que se puede observar es que los *SetPoints* se agregan al código fuente del código base. No obstante esto se puede solucionar si los descriptores semánticos se implementan como atributos .Net y, gracias a la reflexión y a la emisión de código que brinda la plataforma, no habría problemas en volver a generar el ensamblado con los *SetPoints* ya aplicados.

Por otra parte, este proyecto se encuentra en una fase muy temprana y hay escasa documentación al respecto. La sintaxis del lenguaje de definición de *SetPoints* no está definido, como tampoco la forma de realizar los *pointcuts*, por lo que no se han podido efectuar pruebas sobre el prototipo para evaluar características de reutilización, usabilidad y posibles carencias.

⁹ Los siguientes ejemplos únicamente son aproximaciones sobre la sintaxis que podría tener en .NET, puesto que *SetPoint!* aún no tiene publicada ninguna sintaxis específica, únicamente los conceptos.

2.8 Otras aproximaciones

Las tecnologías mostradas en las secciones anteriores no son las únicas existentes, pues la programación orientada a aspectos aún está dando sus primeros pasos y constantemente aparecen nuevos proyectos de investigación. A continuación se citan brevemente otras aproximaciones existentes pero que no han sido analizadas en profundidad por distintas cuestiones: no disponer de prototipos o documentación suficiente, por no aportar nuevos conceptos, etc.

2.8.1 AOP.NET

AOP.NET [Schm02] fue un proyecto de Siemens del año 2002 en el que se implementaron conceptos de AOP usando .Net.

Realiza el *weaving* de clases en ejecución a nivel del CLR, trabajando con la interfaz *.NET Profiling API* no manejada y modificando en tiempo de ejecución las tablas donde se ubican las definiciones de los métodos (de forma similar a *Rapier-Loom.Net*) para interceptar el código y redirigirlo a los aspectos.

Esta aproximación trabaja a muy bajo nivel con el CLR de .Net, no dispone de un prototipo accesible ni documentación al respecto y además, el proyecto no parece haber seguido adelante.

2.8.2 Weave.NET y SourceWeave.NET

Estas dos aproximaciones son proyectos del mismo grupo de investigación que desarrolló *AspectC#*, del cual *SourceWeave* es una evolución. *SourceWeave.NET* [Jack04] es una aproximación que siguiendo con *AspectC#* intenta realizar la unión de aspectos con código base a nivel del código fuente utilizando para ello la plataforma .NET. La idea es definir el código base, al igual que los aspectos, en cualquier lenguaje soportado por .NET y a través de un descriptor XML que define el *weaving*, generar el código fuente resultante de unir los aspectos al código base. Esto plantea los mismos problemas que aparecían en *AspectC#*, entre ellos el no soportar el polimorfismo aspectual.

Por otra parte, de forma muy similar a CLAW y AOP#, *Weave.Net* [Laff03] genera el ensamblado destino a partir de un ensamblado con el código base, otro con el código de los aspectos y una especificación XML con los *weavings* a realizar. Por ello, también presenta los mismos problemas que dichas aproximaciones.

2.8.3 EOS

EOS [Raj03] es otra aproximación cuya aportación reside en el concepto de *Instance-Level Aspects*. Con este término definen a los aspectos como entidades que se enlazan a instancias de clases, diferenciando entre ellas si aplicar el *weaving* o no mediante diferentes construcciones sintácticas. Esto permite decidir en tiempo de ejecución si un aspecto se va a activar para un determinado objeto o no. Sin embargo, la asignación de aspectos a clases se realiza en tiempo de compilación. Para llevar a cabo estos conceptos, proporciona una extensión al lenguaje C# para .NET y por ello requiere de un compilador que está aún en fase beta.

2.8.4 AspectDNG

AspectDNG [Dng04] es una aproximación que ha sido desarrollada recientemente por la comunidad *OpenSource* y presenta otra alternativa basada en la unión de ensamblados de código base y aspectos, mediante la definición de los *weavings* en XML. La diferencia radica en que se desensamblan los diferentes *assemblies* a un lenguaje intermedio llamado *ILML*, al que posteriormente se le aplica el *weaving*. Después se genera el ensamblado final con todo el código enlazado.

2.9 Conclusiones

Una vez mostradas las principales tecnologías orientadas a Aspectos disponibles en .NET, a continuación se muestra una tabla resumen en la que se pueden comparar cómo las diferentes tecnologías aplican el *weaving*, cómo definen los aspectos, o qué ideas nuevas aportan al AOP.

AspectC# y SourceWeave.NET	<i>Nivel de Unión</i>	Código Fuente (Parser propio + CodeDom)
	<i>Weaving</i>	Parsea el código fuente uniendo código Base con Aspectos, basándose en un fichero XML para los <i>pointcuts</i> , generando código fuente ya enlazado.
	<i>Definición Aspectos</i>	No extiende el lenguaje, lo utiliza para definir los aspectos.
	<i>Ideas aportadas</i>	Definición de <i>pointcuts</i> de forma externa al código. Separación clara de Código Base y Aspectos.
	<i>Dinámico</i>	<input checked="" type="checkbox"/> La compilación de aspectos es estática.

Loom.Net	<i>Nivel de Unión</i>	A través del CLR (usando Introspección y Reflexión)
	<i>Weaving</i>	Los Aspectos encapsulan al código Base, y el cliente llama al nuevo componente, a modo de Proxy.
	<i>Definición Aspectos</i>	Se definen mediante plantillas reutilizables y XML para realizar el <i>weaving</i> . Extiende el lenguaje.
	<i>Ideas aportadas</i>	Utilización de expresiones regulares en el código para las sustituciones de aspectos y uso de plantillas.
	<i>Dinámico</i>	<input checked="" type="checkbox"/> La generación de aspectos se realiza en compilación
Rapier-Loom.Net	<i>Nivel de Unión</i>	La interceptación de código base se realiza modificando la tabla de métodos virtuales.
	<i>Weaving</i>	Mediante atributos se configuran los <i>pointcuts</i> . El <i>weaving</i> se realiza en tiempo de ejecución al crear una nueva instancia de una clase del código base.
	<i>Definición Aspectos</i>	Añade estructuras sintácticas adicionales para realizar los <i>weavings</i> en tiempo de ejecución, como los atributos.
	<i>Ideas aportadas</i>	Definición de un constructor para unir instancias de Aspectos y clase Base. Utilización de atributos para indicar los <i>pointcuts</i> .
	<i>Dinámico</i>	<input checked="" type="checkbox"/>
JAsCo.Net	<i>Nivel de Unión</i>	A través del CLR (Parser propio + modificación posterior del ensamblado generado)
	<i>Weaving</i>	Los Aspectos son reutilizables, el <i>weaving</i> se realiza a través de los Conectores. Al componente base se le insertan <i>traps</i> capturadas por una aplicación intermedia.
	<i>Definición Aspectos</i>	Extiende el lenguaje, pero añade mayor expresividad para el uso de los aspectos.
	<i>Ideas aportadas</i>	Definición de <i>pointcuts</i> mediante otra entidad: los conectores. Los Aspectos son abstractos, heredables y separables en estados.
	<i>Dinámico</i>	<input checked="" type="checkbox"/>
SetPoint!	<i>Nivel de Unión</i>	A través del CLR para etiquetar ensamblados ya compilados. Si se dispone del código fuente, a través de atributos.
	<i>Weaving</i>	Se establecen Meta-Etiquetas en el código Base, que permitirán definir los <i>pointcuts</i> por su contenido semántico
	<i>Definición Aspectos</i>	No extiende el lenguaje. Se definen en el mismo lenguaje objetivo.
	<i>Ideas aportadas</i>	El <i>weaving</i> está basado en funciones que enlazan el código base a los aspectos según sea su contenido semántico.
	<i>Dinámico</i>	<input checked="" type="checkbox"/>

Tabla 1 - Comparación entre las diferentes tecnologías AOP en .NET

PRISMA

Contenidos del capítulo:

3.1 El Modelo PRISMA	41
3.2 Visión orientada a Aspectos	42
3.3 Visión orientada a Componentes	44
3.3.1 Componentes	44
3.3.2 Conectores	45
3.3.3 Sistemas	46
3.4 Lenguaje de definición de Arquitecturas	47
3.4.1 Definición de Tipos	48
3.4.2 Configuración Arquitectónica	61
3.5 Otros aspectos avanzados de PRISMA	63

PRISMA

En un futuro, PRISMA será un marco de trabajo o *framework* que incluirá un modelo, unos lenguajes, una herramienta y una metodología. En este capítulo se presentará tanto el modelo como los lenguajes de este *framework*.

PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables. Para ello, se basa en componentes y aspectos para la construcción de modelos arquitectónicos y se caracteriza por la integración que realiza del Desarrollo de Software Basado en Componentes (DSBC), del Desarrollo de Software Orientado a Aspectos (DSOA) y por sus propiedades reflexivas.

PRISMA define una metodología guiada y (semi-)automatizada para obtener el producto software final basándose en el paradigma de la prototipación automática de Balzer [Bal85]. En esta metodología, el usuario utilizará una herramienta de modelado (compilador de modelos) que le permitirá validar el modelo mediante la animación de los modelos arquitectónicos definidos en la especificación. PRISMA está basado en un lenguaje de especificación formal Orientado a Objetos llamado OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos, que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA preserva las ventajas de OASIS, garantizando la compilación de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos arquitectónicos, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

3.1 El Modelo PRISMA

El modelo arquitectónico PRISMA [Per03] integra dos aproximaciones de desarrollo de software, el DSBC y el DSOA. Esta integración se consigue definiendo los tipos mediante la composición de aspectos. La mayoría de modelos arquitectónicos analizan cuáles son las primitivas básicas para la especificación de sus arquitecturas y exponen su sintaxis y semántica. El modelo PRISMA, además de definir los elementos arquitectónicos básicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las necesidades de cada uno de ellos.

Un elemento arquitectónico de PRISMA puede ser analizado desde dos vistas diferentes: la interna y la externa. La vista interna (ver Figura 15) define un elemento arquitectónico como un prisma con tantas caras como aspectos considere. Dichos aspectos están definidos desde la perspectiva del problema y no de su solución, aumentando el nivel de abstracción y evitando el solapamiento de código que puede sufrir la programación orientada a aspectos.

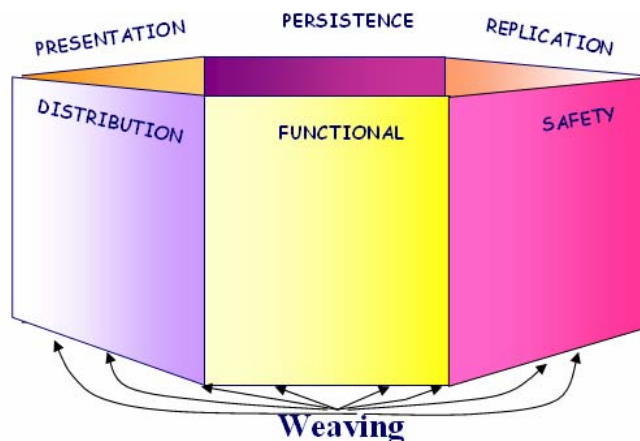


Figura 15 - Vista interna de un elemento arquitectónico PRISMA

Por otro lado, la vista externa de un elemento arquitectónico encapsula la funcionalidad como una caja negra y publica el conjunto de servicios que ofrece al resto de elementos arquitectónicos (ver Figura 16)



Figura 16 - Vista externa de un elemento arquitectónico PRISMA

Por ello, el modelo PRISMA puede analizarse desde dos perspectivas diferentes, la orientada a aspectos (vista interna) y la orientada a componentes (vista externa). A continuación se presenta el modelo desde ambas perspectivas.

3.2 Visión orientada a Aspectos

Las técnicas AOP, como han sido mostradas en los capítulos anteriores, permiten encapsular en un solo elemento (los aspectos) aquella funcionalidad que se repite a lo largo de todo el sistema. Un aspecto puede verse como la unión de un conjunto de interfaces que incluyen servicios y atributos del tipo del aspecto, más la especificación semántica de su estructura y comportamiento. No obstante, el modelo PRISMA va más allá y amplía el concepto de aspecto en el sentido de que cada aspecto representa una vista específica de la arquitectura del sistema (vista funcional,

distribución, calidad, etc.). Por ello, cada primitiva de la arquitectura está formada a su vez por varios aspectos que la describen desde diferentes puntos de vista. Algunos tipos de aspectos pueden ser:

- **Aspecto Funcional:** Captura la semántica del sistema de información mediante la definición de sus atributos, sus servicios y su comportamiento
- **Aspecto de Coordinación:** Define la sincronización entre elementos arquitectónicos durante su comunicación.
- **Aspecto de Distribución:** Especifica las características que definen la localización dinámica del elemento arquitectónico en el cual se integra. También define los servicios necesarios para implementar estrategias de distribución de los elementos arquitectónicos (como movilidad, equilibrio de carga, etc.) con el objetivo de optimizar la distribución de la topología del sistema resultante [Ali03].

Existen diferentes tipos de aspecto, definiéndose cada uno de ellos de forma independiente. El número de tipos no está limitado, ya que gracias al metanivel pueden definirse nuevos tipos. Los mostrados son sólo algunos tipos de aspectos, en otros sistemas de información puede haber otros tipos diferentes, que emergerán de los requisitos de los sistemas a los que se aplique el modelo PRISMA.

No obstante, no es suficiente con definir únicamente los tipos de aspectos, sino que también hay que enlazarlos entre sí: esto se realiza mediante los *weavings*. Los *weavings* indican que la ejecución de un servicio de un aspecto puede provocar la invocación de servicios en otros aspectos.

A diferencia de las tecnologías orientadas a aspectos vistas hasta ahora, en la que los aspectos se enlazaban al código base, en PRISMA no existe el código base como tal, sino que la funcionalidad global de una primitiva del modelo arquitectónico se define mediante la unión de los distintos aspectos que la forman. Otra diferencia importante a resaltar, es que en las otras tecnologías (como *AspectJ*) los *weavings* se definen en el mismo aspecto, resultando en una pérdida de reutilización por parte del aspecto. Es por esto que en PRISMA los *weavings* entre los aspectos se definen en el elemento arquitectónico que los integra.

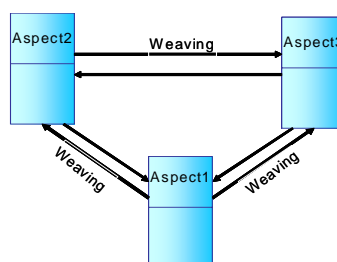


Figura 17 - Weavings entre aspectos

Como se puede observar en la figura, los aspectos se definen independientemente de la arquitectura donde se vayan a integrar, y son los *weavings* los que unen el conjunto de aspectos de un mismo elemento arquitectónico y forman la vista interna mostrada anteriormente. De la misma forma que en las tecnologías vistas anteriormente, PRISMA dispone de varios tipos de *weavings*, que son los siguientes:

- *After*: aspect1.service es ejecutado después de aspect2.service.
- *AfterIf (condition)*: igual que *After*, sólo si se cumple la condición.
- *Before*: aspect1.service es ejecutado antes de aspect2.service.
- *BeforeIf (condition)*: igual que *Before*, sólo si se cumple la condición.
- *Instead*: aspect1.service es ejecutado en lugar de aspect2.service.

Como se puede observar, los aspectos PRISMA son altamente reutilizables y favorecen la mantenibilidad, pues un cambio en un *concern* específico (i.e. el cambio en la estrategia de distribución de elementos arquitectónicos) sólo ha de realizarse en un aspecto y no en todo el conjunto del sistema. Además, la importación de COTS se expresa gracias al carácter opcional de los aspectos, lo que permite ver a las primitivas arquitectónicas como cajas negras en cuyo interior pueden incorporarse componentes COTS.

3.3 Visión orientada a Componentes

La visión externa de los elementos arquitectónicos está orientada hacia el concepto de componentes, a diferencia de la orientación a aspectos de la visión interna. El modelo arquitectónico consta de tres tipos de primitivas: *componentes*, *conectores* y *sistemas*. Y como se ha visto anteriormente, cada uno de estos tipos a su vez está compuesto por tantos **aspectos** como se consideren relevantes para definir el sistema de información con precisión.

3.3.1 Componentes

Un componente PRISMA se define como un elemento arquitectónico que captura la funcionalidad del sistema de información y no actúa como coordinador entre otros sistemas arquitectónicos. Puede verse como una parte del sistema que no se puede disgregar en partes más simples. Está formado por:

- un identificador,
- un conjunto de aspectos, que le proporcionan su funcionalidad,
- las relaciones de *weaving*, que proporcionan la cohesión entre los aspectos que lo forman,

- y los puertos de entrada y salida, cuyo tipo es una interfaz específica (un conjunto de servicios).

Los puertos son aquellos elementos que permiten la interacción del componente con los demás elementos arquitectónicos. Un puerto puede ofrecer un comportamiento servidor, cliente o cliente/servidor, en función de los servicios que lo definan. El comportamiento servidor especifica los servicios que el componente ofrece al resto de los demás elementos mientras que el comportamiento cliente especifica los servicios que puede requerir de los demás elementos arquitectónicos.

Los componentes PRISMA han de cumplir ciertas restricciones:

- todo componente debe especificar su aspecto funcional, excepto en el caso de componentes externos (COTS)
- un componente nunca puede contener un aspecto de coordinación
- un componente nunca puede contener dos aspectos del mismo tipo
- los tipos de los puertos de un componente sólo podrán ser aquellas interfaces que usen algunos de los aspectos que formen a dicho componente.

3.3.2 Conectores

Un conector PRISMA es un elemento arquitectónico que actúa como coordinador entre otros elementos arquitectónicos. El conector permite describir interacciones complejas entre componentes mediante su aspecto de coordinación. Está formado por:

- un identificador,
- un conjunto de aspectos, que le proporcionan su funcionalidad,
- las relaciones de *weaving*, que proporcionan la cohesión entre los aspectos que lo forman,
- y los roles de entrada y salida, cuyo tipo es una interfaz específica (un conjunto de servicios).

Los conectores sincronizan y conectan componentes, otros conectores o sistemas a través de sus roles, que de la misma forma que los puertos de los componentes, definen un conjunto de servicios que el conector ofrece (comportamiento servidor), que el conector necesita (comportamiento cliente) o un comportamiento mixto (cliente/servidor).

Existen una serie de restricciones que se han de tener en cuenta:

- un conector siempre ha de contener un aspecto de coordinación,
- un conector nunca puede contener un aspecto funcional,
- un conector nunca puede tener dos aspectos del mismo tipo,

- los tipos de los roles de un conector sólo podrán ser aquellas interfaces que usen alguno de los aspectos que formen a dicho conector,
- una instancia de conector al menos ha de conectar dos elementos arquitectónicos PRISMA

3.3.3 Sistemas

En la mayoría de modelos arquitectónicos, surge la necesidad de tener mecanismos de abstracción que permitan tener elementos de mayor granularidad, aumentando la modularidad, composición y reutilización de elementos arquitectónicos. En PRISMA esto se consigue mediante los sistemas, los cuales permiten encapsular un conjunto de conectores, componentes y otros sistemas correctamente conectados entre sí. En dicha encapsulación pueden surgir propiedades emergentes.

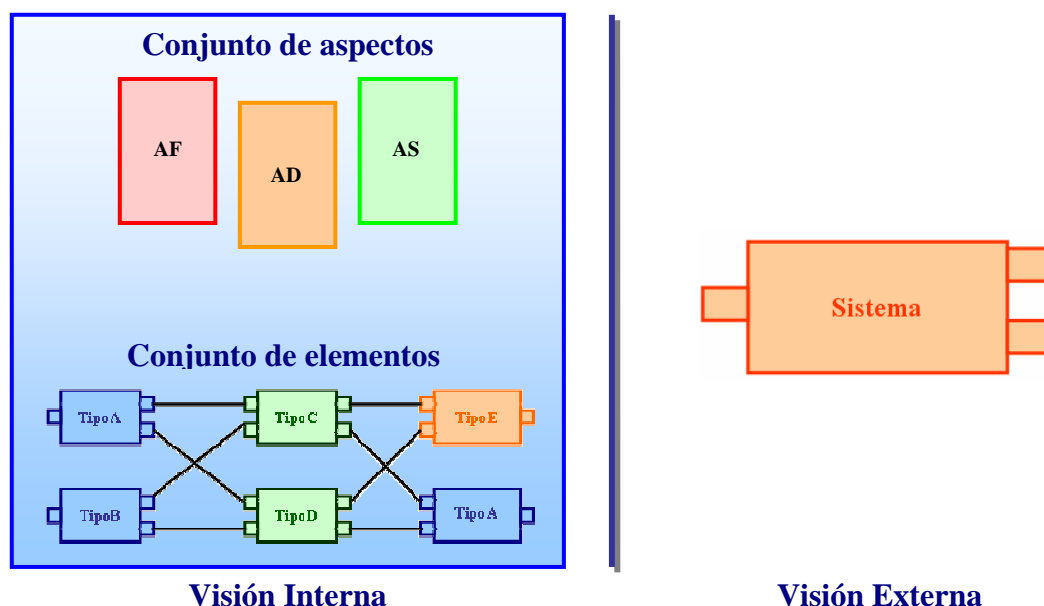


Figura 18 - Vistas de un Sistema PRISMA

Un sistema está formado por los mismos elementos y restricciones que un componente: una función de identificación, un conjunto de aspectos, una serie de *weavings* que los interconecta, y los puertos. No obstante, al ser un tipo compuesto, dispone también de un conjunto de elementos arquitectónicos que pueden o no estar conectados entre sí y realizan una determinada función para el sistema.

La especificación de la conexión entre estos elementos y el propio sistema se realiza mediante los **bindings**, que son los enlaces entre los puertos del sistema y los puertos (o roles en caso de ser conectores) de los elementos que encapsula. Permiten mantener un enlace entre los distintos niveles de granularidad de los elementos arquitectónicos que forman un sistema.

Por otra parte, las conexiones entre los distintos elementos que contiene (componentes, conectores y attachments) se realizan mediante los **attachments**, que establecen la conexión entre puertos de componentes y roles de conectores.

3.4 Lenguaje de definición de Arquitecturas

El Lenguaje de Descripción de Arquitecturas de PRISMA (ADL) está basado en OASIS [Let98], como ya se comentó anteriormente. A diferencia de otros modelos, el ADL (Architecture Definition Language) de PRISMA está dividido en dos niveles de especificación: el *nivel de definición de tipos* y el *nivel de configuración*.

El nivel de definición de tipos de PRISMA potencia la reutilización y combina el DSBC y DSOA. Este nivel permite definir los tipos necesarios para especificar un sistema arquitectónico. Dichos tipos se guardan en una librería para posteriormente reutilizarlos en la definición de distintos modelos arquitectónicos. Sus ciudadanos de primer orden son: interfaces, aspectos, componentes y conectores.

El nivel de configuración permite definir las instancias y especificar la topología del modelo arquitectónico. Para ello, en primer lugar se han de importar todos aquellos tipos (conectores, componentes y sistemas) definidos mediante el lenguaje de definición de componentes, que se necesiten para un determinado modelo arquitectónico. Después, se ha de definir el conjunto de instancias necesarias de cada uno de los tipos importados. Finalmente, se debe especificar la topología, interconectando adecuadamente las instancias del modelo.

Esta diferenciación de nivel de granularidad proporciona importantes ventajas frente a la mezcla de distintos niveles de granularidad dentro de la especificación. Una de ellas es que permite gestionar de forma independiente los tipos y las topologías específicas de cada sistema, incrementando la reutilización y obteniendo un mejor mantenimiento de las librerías de tipos. Además, permite discernir claramente entre la evolución de tipos (nivel de definición de tipos) y la evolución arquitectónica (nivel de configuración), teniendo meta-eventos diferentes en cada lenguaje para hacer evolucionar a los tipos o a las conexiones existentes entre sus instancias.

3.4.1 Definición de Tipos

A continuación se describirá la sintaxis de los ciudadanos de primer orden de PRISMA: interfaces, aspectos, componentes y conectores. Los sistemas también se definen en este nivel con el objetivo de que se puedan almacenar conjuntamente con los demás elementos y favorecer la reutilización.

3.4.1.1 Interfaces

Las interfaces son el mecanismo usado en PRISMA para establecer canales de comunicación entre los componentes, conectores y sistemas. Una interfaz describe la visibilidad parcial que tienen el resto de elementos, del estado y comportamiento del tipo que define la semántica de dicha interfaz. Por lo tanto, una instancia sólo puede solicitar y consultar aquellas propiedades y servicios publicados en sus interfaces. De este modo, una interfaz actúa como mecanismo de seguridad y permite que los componentes, conectores y sistemas sean vistos como cajas negras, preservando un alto nivel de abstracción.

Las interfaces definen los servicios sin tener en cuenta los puertos, roles o aspectos que las van a utilizar, con el objetivo de favorecer la reutilización. No obstante, se ha de tener en cuenta que todos los servicios que define una misma interfaz han de pertenecer al mismo tipo: funcional, distribución, etc. pues serán implementadas por completo por un aspecto de un tipo concreto, y no puede haber solapamiento entre aspectos de tipos diferentes.

A continuación, se presenta la interfaz *ICreditCardTransactions*, que define las operaciones típicas para el manejo de cuentas bancarias, y *IMobility*, que define los servicios de movilidad que puede contener un componente.

```
Interface ICreditCardTransactions
    Withdrawal(input quantity: decimal, output newMoney: decimal);
    Balance(output newMoney: decimal);
    ChangeAddress(input newAdd: string);
End_Interface ICreditCardTransactions;
```

```
Interface IMobility
    Move(input newLoc: LOC);
End_Interface IMobility;
```

Como se puede observar, los servicios definen los parámetros que son de entrada mediante la palabra reservada *input*, mientras que los de salida se definen mediante la palabra reservada *output*.

3.4.1.2 Aspectos

Para definir los aspectos se ha definido una sintaxis específica basada en OASIS [Let98]. La plantilla genérica se presenta a continuación:

```

1  tipo_aspecto Aspect nombre using interface1, ... interfacen;

2  Attributes
   [Constant | Variable]
   <nombre_atributo1> : <tipo_atributo>;
   ...
   <nombre_atributon> : <tipo_atributo>;
   [Derived]
   <fórmula_derivación>

3  Constraints
   static <restricciones_estáticas>

Services
4  Begin [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio>]];
   Valuations
   <fórmulas_observabilidad_atributo>

   [in | out] <nombre_servicio> (<arg_servicio>)
   [as <nombre_servicio> (<arg_servicio>)];
5  Valuations
   <fórmulas_observabilidad_atributo>

   End [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio>]];
   Valuations
   <fórmulas_observabilidad_atributo>

6  Preconditions
   <fórmula_precondición_evento>

7  Triggers
   <fórmula_disparo_evento>

8  Operations [transaction]
   <fórmula_transacción>

9  Played Roles
   <fórmula_played_role>

10 Protocols
   <fórmula_protocolo>

End_ tipo_aspecto Aspect name;
```

Tabla 2 - Plantilla de un aspecto PRISMA

La cabecera de un aspecto (*sección 1*) define el tipo del aspecto (funcional, distribución, etc.), el nombre que lo identifica y las interfaces a las que se da semántica. La *sección 2* define la lista de atributos que almacenan el estado de un aspecto. Para cada atributo se indica el nombre y el tipo de valores que va a contener. Puede ser de tres tipos:

- **Constante:** su valor se establece en la inicialización y no es modificado
- **Variable:** su valor puede cambiar cuando ocurra una acción relevante (como la invocación de un servicio o la activación de un disparador)
- **Derivado:** su valor se especifica en términos de los valores de otros atributos, mediante una fórmula de derivación.

La *sección 3* define las restricciones de integridad que deben cumplirse a lo largo de toda la vida de las instancias del tipo del aspecto. Son fórmulas basadas en el estado del aspecto y se pueden clasificar en estáticas o dinámicas, dependiendo de si se refieren sólo a un estado o relacionan diferentes estados, respectivamente. Las restricciones dinámicas se caracterizan por el uso de operadores temporales como *sometimes* o *always*.

La *sección 4* define los servicios que implementa el aspecto. Todos los servicios de la interfaz o interfaces a las que da semántica el aspecto deben definirse, además de aquellos servicios propios y no públicos del aspecto. Existen diferentes tipos de servicio: el servicio *begin* permite inicializar atributos, el servicio *end* permite liberar la memoria asociada al aspecto y finalmente, los servicios de modificación y consulta. Es importante resaltar que los servicios *begin* y *end* no indican que un aspecto sea instanciable, sino que hacen referencia al servicio de creación y destrucción del elemento arquitectónico al cual el aspecto pertenecerá. Además, cada servicio de modificación o consulta puede tener un comportamiento servidor (*in*), comportamiento cliente (*out*) o un comportamiento servidor y cliente (*in/out*). El comportamiento servidor se caracteriza porque el aspecto proporciona un servicio que puede ser requerido por otros aspectos, mientras que el comportamiento cliente indica que el aspecto va a requerir dicho servicio a otro aspecto/componente.

De la misma forma que se ha visto en la definición de los servicios en la interfaz, los argumentos de los servicios que incluye un aspecto pueden ser de entrada (*input*) o de salida (*output*), pero no devuelven ningún valor.

Por otro lado, el lenguaje ADL también proporciona un mecanismo de renombrado de servicios, mediante el operador *as*, con lo que se consigue que el nombre del servicio definido en una interfaz no tenga que coincidir con el nombre del servicio definido en el aspecto.

La semántica que define el comportamiento de los servicios se especifica mediante las *evaluaciones* (*sección 5*), que permiten definir el cambio de estado de los atributos del aspecto ante la ejecución de un servicio determinado. Se definen mediante fórmulas en lógica dinámica del tipo “ $\varphi \rightarrow [a]\psi$ ” y se interpretan como: “si en un determinado estado del aspecto se satisface φ y ocurre la acción ‘a’, en el estado inmediatamente posterior se satisface ψ ”. Se puede no declarar la condición de evaluación φ asumiéndose *true*.

Las *secciones 6 y 7* definen respectivamente las precondiciones y los disparadores (*triggers*). Las precondiciones definen las condiciones que se deben cumplir para que un servicio se pueda ejecutar, mientras que los disparos permiten especificar la ejecución de un servicio cuando se cumple una determinada condición.

La *sección 8* define las operaciones y transacciones. Una operación es un servicio no elemental y no atómico, en el que se define una serie de servicios que se van a ejecutar secuencialmente. Para que estos servicios se ejecuten de forma atómica se emplea la palabra reservada *transaction*. Por ejemplo, la definición de una operación transaccional para el envío de noticias a una lista de correo en PRISMA sería:

Operations

```
transaction EnviarNoticias(input listaCorreo: String, input
    listaNoticias: StringList):
    ENVIARNOTICIAS = SeleccionarListaCorreo(listaCorreo).NOTICIAS;
    NOTICIAS = SeleccionarNoticias(listaNoticias, coleccion).ENVIO;
    ENVIO = envioSimultaneo(coleccion).CONFIRMACIÓN;
    CONFIRMACION = confirmacionLlegada.ENVIARNOTICIAS;
```

En el ejemplo se puede observar cómo se realiza el mapeo de parámetros de la transacción a cada uno de los servicios y cómo se especifica el siguiente estado alcanzado *nombre_servicio.nuevo_estado*.

Por último, en la *sección 9 y 10* se especifican los *Played_Roles* y los *Protocolos*. El *Protocolo* define un proceso describiendo un conjunto de acciones cuya ocurrencia es posible. Está compuesto por un conjunto de procesos que establecen los servicios y transiciones a otros procesos posibles. Además, también permite especificar las prioridades de ejecución de cada uno de los servicios implicados.

Por otra parte, un *Played_Role* es una proyección del protocolo que define el comportamiento parcial de un rol o papel determinado. Por lo tanto, todo *Played_Role* debe ser compatible en signatura y proceso con el protocolo de forma que, dicho papel o rol se desempeña dentro del comportamiento global del protocolo y sus cálculos son un subconjunto de los cálculos del proceso.

Un *Played_Role* es una vista parcial del protocolo que tiene sentido por sí solo, un comportamiento específico y restrictivo que es posible asociarlo posteriormente a un puerto o rol de un componente o conector. Dicha asociación permite definir de forma exacta el comportamiento que debe ejercer un puerto o rol además de los servicios que publica, establecidos por la interfaz que lo tipa.

Ambos se basan en el *Cálculo Poliádico* [Mil91], el cual permite describir de forma sencilla la ejecución de procesos y servicios susceptibles de ejecutarse concurrentemente. Una parte de su sintaxis es la siguiente:

x,y,z	Nombre (interfaz, servicio, played_role, parámetro, identif.)
x .y	Jerarquía de nombres, tal que x = interfaz e y = servicio

All *	Todos los played_roles
x(m, n)	Vector de nombres tal que m, n son parámetros
P,Q,R	Proceso
P ::= x?(m,n).P	Prefijo de entrada o acción de recepción
x!(m,n).P	Prefijo de salida o acción de envío
P + Q	Selección no determinista
P ∧ Q	Conjunción
P Q	Composición paralela (conurrencia de procesos)
P → Q	Composición secuencial
x(m,n):k.P	k indica la prioridad del servicio x, tal que k = 0, 1, ...
if b then P else Q	Alternativa (derivado)
case(b ▷ P → Q)	Alternativa múltiple
x=y P	Comparación o <i>matching</i>

Tabla 3 - Sintaxis de Pi-Cálculo Poliádico

Para ilustrar estos conceptos, se muestran el conjunto de aspectos (funcional, distribución y coordinación) de un sistema bancario sencillo. El primero de ellos es el aspecto funcional *BankInteraction*, que implementa la funcionalidad básica de una cuenta bancaria.

```

Functional Aspect BankInteraction using ICreditCardTransactions
  Attributes
    Constant
      numberId: decimal;
    Variable
      address: String;
      money: decimal;
  Constraints
    static { money >= 0}
  Services
    Begin(input accountId: decimal);
      Valuations
        [in begin(accountId)] numberId = accountId;
      in/out Withdrawal(input quantity: decimal,
        output newMoney: decimal);
        Valuations
          [in Withdrawal(quantity, newMoney)] money = money - quantity;
      in/out Balance(output newMoney: decimal);
        Valuations
          [in Balance(newMoney)] newMoney = money;
      in/out ChangeAddress(input newAdd: String);
        Valuations
          [in ChangeAddress(newAdd)] address = newAdd;
    end;
  Preconditions
    in withdrawal(quantity)
      if quantity <= money;
  Protocols
    BANKINTERACTION = begin.TRANSACTION;
    TRANSACTION = ICreditCardTransactions.Withdrawal?(quantity,
newMoney).TRANSACTION +
ICreditCardTransactions.Balance?(newMoney).TRANSACTION +
ICreditCardTransactions.ChangeAddress?(newAdd).TRANSACTION +

```



```

    end;
End_Functional Aspect BankInteraction;

```

Como puede observarse, se han definido una serie de atributos, uno de los cuales es constante (el número de cuenta, que se define en el momento de creación). Se han definido los servicios correspondientes para sacar dinero de la cuenta (*Withdrawal*), para la consulta de saldo (*Balance*), y para cambiar la dirección (*ChangeAddress*). Obsérvese cómo se ha definido la semántica de dichos servicios mediante la lógica dinámica.

Se ha definido una restricción (sección *constraints*) mediante la cual no se permite extraer dinero si la cuenta no dispone de saldo, y una precondition (sección *preconditions*) para el servicio *Withdrawal* por la cual se establece que sólo se podrá sacar dinero si se dispone de dicha cantidad en la cuenta.

En la sección *Protocols* se han definido dos procesos: BANKINTERACTION, que es el proceso inicial, y TRANSACTION. Obsérvese la sintaxis en Pi-Cálculo Poliádico para indicar que la petición (comportamiento servidor, mediante el símbolo ‘?’) de cualquier servicio (*Interfaz.Nombre_Servicio*) volverá al proceso TRANSACTION.

En segundo lugar se muestra el aspecto de distribución y el aspecto de coordinación. Ambos son importantes, pues formarán parte de los componentes que formarán el sistema que se construirá como ejemplo. La finalidad del aspecto de distribución *ExtMbile* no es más que indicar que el componente que lo utilice puede distribuirse a una nueva máquina.

```

Distribution Aspect ExtMbile using IMobility;
  Attributes
    Location: LOC NOT NULL;

  Services
    begin(input initialLOC: LOC);
      Valuations
        [in begin(initialLOC)] Location = initialLOC;
    in Move(input newLoc:LOC);
      Valuations
        [move(newLoc)] Location:= newLoc;
    end;

  Protocols
    CREATION = begin.EXTMBILE;
    EXTMBILE = IMobility.Move?(newLoc).EXTMBILE + end;
End Distribution Aspect ExtMbile;

```

La finalidad del aspecto de coordinación es coordinar las peticiones entre componentes que demanden el servicio *Withdrawal* o *Balance* al componente que actúe de servidor:

```

Coordination Aspect BankCoordination using ICreditCardTransactions
  Services
    in/out Withdrawal(input quantity: decimal, output money: decimal);
    in/out Balance(output money: decimal);

```

```
begin;
end;
```

Played_Roles

```
CLIENT =
  (ICreditCardTransactions.Withdrawal?(quantity, money)
  →
  ICreditCardTransactions.Withdrawal!(quantity, money) )
+
  (ICreditCardTransactions.Balance?(money)
  →
  ICreditCardTransactions.Balance!(money) )

SERVER =
  (ICreditCardTransactions.Withdrawal!(quantity, money)
  →
  ICreditCardTransactions.Withdrawal?(quantity, money) )
+
  (ICreditCardTransactions.Balance!(money)
  →
  ICreditCardTransactions.Balance?(money) )
```

Protocol

```
BANKCOORDINATION = begin.COORD;
COORD =
  (CLIENT.Withdrawal?(quantity, money) →
  SERVER.Withdrawal!(quantity, money) →
  SERVER.Withdrawal?(quantity, money) →
  CLIENT.Withdrawal!(quantity, money)).COORD
+
  (CLIENT.Balance?(money) →
  SERVER.Balance!(money) →
  SERVER.Balance?(money) →
  CLIENT.Balance!(money)).COORD + end;
```

End_Coordination Aspect BankCoordination;

Con este ejemplo, más complejo que los anteriores, puede observarse la potencia expresiva del *Pi-Cálculo Poliádico*. Este aspecto se utilizará en un conector para coordinar dos componentes, uno que se comportará como cliente y el otro como servidor. Para ello, se han definido dos *played_roles*: uno para el comportamiento CLIENT y otro para el comportamiento SERVER. El *played_role* CLIENT describe el flujo de entradas y salidas para el comportamiento cliente. Por ello, cuando se reciba una petición del servicio *Withdrawal* (flujo de entrada, indicado mediante el símbolo "?") la siguiente acción válida sólo podrá ser devolver el resultado de dicha petición (flujo de salida, indicado mediante el símbolo '!'). En cambio, el *played_role* SERVER modela el otro lado de la comunicación: cuando se reciba un flujo de salida (invocar un servicio), la única acción válida sólo podrá ser un flujo de entrada (obtener los resultados de dicho servicio). Los símbolos '!' deben verse como flujos de *salida (output)* y los "?" como flujos de *entrada (input)*.

Por otra parte, el protocolo modela el proceso global: coordina y sincroniza los servicios de los distintos *played_roles*. En la Figura 19 - Esquema de funcionamiento de un protocolo se puede observar el esquema de funcionamiento de un protocolo de forma gráfica. Cuando llegue una

petición de *Withdrawal* (?) al aspecto de coordinación, la siguiente acción que se realizará es transmitir dicha petición (!) al componente que actúa como servidor (el componente *Banco*). Entonces, el aspecto esperará hasta que se reciba la respuesta por parte del servidor (SERVER.*Withdrawal*?), tras lo cual se retransmitirá al componente cliente (CLIENT.*Withdrawal*!). Una vez se enviase al cliente, el siguiente estado válido sería COORD, y podría procesarse otro servicio diferente.

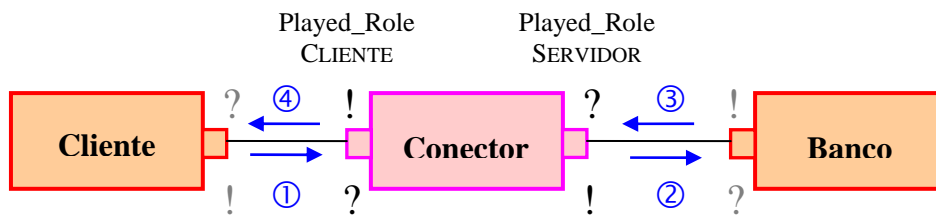


Figura 19 - Esquema de funcionamiento de un protocolo

Si se quisiese permitir la ejecución de otro servicio (como *Balance*) mientras el aspecto de coordinación espera la respuesta del servidor, se indicaría mediante el uso del operador '|' (composición paralela), en lugar del operador '+' (selección no determinista), con lo que se especificaría la ejecución concurrente de servicios.

3.4.1.3 Componentes y Conectores

La plantilla de un componente y de un conector, muy similares entre sí, están formadas por cuatro partes básicas: la cabecera, la definición de puertos o roles de comunicación, los aspectos que lo forman y los *weavings*, que los entretujan entre sí. A continuación se muestra la plantilla genérica, y se describirá cada sección en detalle:

```

1  [Component | Connector]_type <nombre_Componente|Conector>
2  [Port | Role]
   <nombre_i> : <interfaz_i>
3  [Played_Role <nombre_aspecto>.<nom_playedRole_del_aspecto>];
   End_[Port | Role];
4  [Functional | Coordination] Aspect Import <nombre_aspecto_j>;
   <tipo_aspecto_k> Aspect Import <nombre_aspecto_k>;
5  [Weaving
   <nombre_aspecto_k>.<nombre_servicio_k>
   <operador_weaving>
   <nombre_aspecto_n>.<nombre_servicio_n>;
   End_Weaving; ]
6  Initialize
   New(<argumentos>) {
   <nombre_aspecto_i>.<begin(<args_constructor_i>);
   }
   End_Initialize;

```

```
7 Destruction
  Destroy() {
    <nombre_aspectoi>.end(<args_destructori>);
    <nombre_aspectoi+1>.end(<args_destructori+1>);
  }
End_Destruction;

End_[Component | Connector]_type <nombre_Componente|Conector>;
```

Tabla 4 - Plantilla de un componente/conector PRISMA

La cabecera (sección 1) se compone de la palabra reservada *component_type* (para el caso de los componentes) o de *connector_type* (para el caso de los conectores), seguida del nombre que reciba el componente o conector, y que lo identificará unívocamente en la librería PRISMA.

Los puertos de comunicación (o roles en el caso de conectores) se definen en la sección 2, indicando qué interfaz implementan. Además, se puede asociar un *played_role* al puerto (sección 3) para especificar el comportamiento del conjunto de servicios que forman parte de la interfaz. El *played_role* pertenece al aspecto indicado mediante la notación punto *<nombre_aspecto>.<nom_playedRole_del_aspecto>*. De esta forma, pueden crearse varios puertos con la misma interfaz pero con *played_roles* diferentes, con lo que cada puerto con idéntica interfaz tendrá un comportamiento diferente: el especificado por el *played_role*.

La importación de los aspectos (sección 4) se realiza indicando el tipo y el nombre del aspecto. Al menos debe importarse un aspecto funcional, en el caso de los componentes, o un aspecto de coordinación, en el caso de los conectores. También cabe recordar que no pueden importarse dos aspectos del mismo tipo dentro del mismo componente o conector.

El *weaving* entre los aspectos (sección 5) se define al importar cada aspecto, indicando con qué aspecto van a entretrearse sus servicios. Éstos se declaran especificando el nombre del aspecto en el cual están definidos mediante la notación: *<nombre_aspecto>.<nombre_servicio>*. El *weaving* entre los servicios se realiza mediante *<operador_weaving>*, que indica si la sincronización se realiza *after* (después), *before* (antes) o *instead* (en lugar de), como se mostró en el apartado 3.2.

Por último, las secciones 6 y 7 describen el proceso de creación y destrucción de los componentes/conectores. En la sección *Initialize* se especifica cómo el servicio *New* desencadena la ejecución del servicio *Begin* de los distintos aspectos que forman el componente, mientras que la sección *Destruction* desencadena la destrucción de los aspectos. Además, en el servicio *New* pueden indicarse los parámetros que necesitan los aspectos para su inicialización, que a su vez pueden ser proporcionados por el elemento arquitectónico que cree el componente/conector.

A continuación, se muestran dos ejemplos de componentes: el componente *ATM*, y el componente *Account*, que importa el aspecto funcional definido anteriormente y realiza el *weaving* del servicio *ChangeAddress* con el servicio *Move* del aspecto de distribución:

```

Component_type ATM
  Port
    VISACreditCard_port: ICreditCardTransactions;
    AccountTrans_port: ICreditCardTransactions;
  End_port;

  Functional Aspect import BankInteraction;
  Distribution Aspect import ExtMbile;
  Initialize
    New(accountId: decimal, initialLocation: LOC) {
      BankInteraction.begin(accountId);
      ExtMbile.begin(initialLocation);
    }
  End_Initialize;

  Destruction
    Destroy() {
      BankInteraction.end();
      ExtMbile.end();
    }
  End_Destruction;
End_Component_Type ATM;

```

```

Component_type Account
  Port
    Account_port: ICreditCardTransactions;
  End_port;

  Functional Aspect import BankInteraction;
  Distribution Aspect import ExtMbile;

  Weaving
    BankInteraction.ChangeAddress(newAdd: string)
      before ExtMbile.Move(newAdd: string);
  End_Weaving;

  Initialize
    New(accountId: decimal, initialLocation: LOC) {
      BankInteraction.begin(accountId);
      ExtMbile.begin(initialLocation);
    }
  End_Initialize;

  Destruction
    Destroy() {
      BankInteraction.end();
      ExtMbile.end();
    }
  End_Destruction;
End_Component_Type Account;

```

También se muestra un conector cuya función sería la de coordinar entre sí los dos componentes definidos anteriormente:

```

Connector_type ATMAccount
  Role
    ATM_role: ICreditCardTransactions,

```

```
    Played_Role BankCoordination.CLIENT;
    Account_role: ICreditCardTransactions,
    Played_Role BankCoordination.SERVER;
End_role;

Coordination Aspect import BankCoordination;
Distribution Aspect import ExtMbile;

Initialize
    New(initialLocation: LOC) {
        BankCoordination.begin();
        ExtMbile.begin(initialLocation);
    }
End_Initialize;

Destruction
    Destroy() {
        BankInteraction.end();
        ExtMbile.end();
    }
End_Destruction;
End_Connector_Type ATMAccount;
```

Como puede observarse, el conector define dos roles con la misma interfaz, pero mediante los played_roles puede distinguir las peticiones de servicio que recibe (comportamiento cliente de los componentes a él conectados) y los servicios ofrecidos (comportamiento servidor). La semántica de coordinación de dichos played_roles vendrá definida por el aspecto de coordinación que lo define.

3.4.1.4 Sistemas, Attachments y Bindings

La plantilla genérica para definir un sistema es la mostrada en la

Tabla 5:

```
1  System_type <nombre_sistema>
2  Ports
    <nom_puertoi> : <interfazi>;
    <nom_puertoi+1> : <interfazi+1>;
End_Ports
3  Variables
    <nom_Vari> : <tipo_componente | tipo_conector>;
    ...
End_Variables
4  Attachments
    <nom_Vari>.<nom_puerto> ↔ <nom_Vark>.<nom_rol>;
    ...
End_Attachments;
5  Bindings
    <nom_Vari>.<nom_puerto> ↔ <nombre_sistema>.<nom_puerto>;
    ...
End_Bindings;
```

```

6   Initialize
    New() {
        <nom_Vari>= new <(nom_componente | nom_conector)>(<args>);
        ...
    }
    End_Initialize;

7   Destruction
    Destroy() {
        <nom_Vari>.destroy();
        ...
    }
    End_Destruction;
End_System_type <nombre_sistema>;

```

Tabla 5 - Plantilla de un sistema PRISMA

En la cabecera de un sistema (sección 1) se indica el nombre identificativo de dicho tipo, que lo identificará dentro de la librería de tipos PRISMA. La definición de puertos (sección 2) permite indicar los puertos de comunicación que tendrá dicho sistema con el resto de elementos arquitectónicos. Mediante los *bindings* se establecerá la relación entre los puertos del sistema y los componentes que les darán semántica a los servicios de dichos puertos.

La definición de variables (sección 3) permite especificar los componentes y/o conectores que formarán parte del sistema. Cada componente importado se asociará con una variable, cuyo tipo será el del componente a importar. La construcción/destrucción de dichos componentes se desencadenará en el momento de creación/destrucción del sistema. Estas operaciones se especifican en la sección *Initialize* y *Destruction* (secciones 6 y 7), respectivamente, y permiten indicar los parámetros que necesitará cada componente para su creación.

La definición de un sistema especifica las relaciones de conexión (*attachments*) y composición (*bindings*) entre los elementos arquitectónicos que contiene. Como se definió anteriormente, los *attachments* establecen la conexión entre los puertos de los componentes y los roles de los conectores, mientras que los *bindings* definen la composición entre el sistema y los componentes o subsistemas que contiene. Esto se define en las secciones 4 y 5. Los *attachments* se especifican mediante una relación entre dos variables, cuyos tipos son un componente y un conector, y a través de la notación punto se indica el puerto y el rol que se va a conectar:

```
<variable: tipo_componente>.<puerto> ↔ <variable: tipo_conector>.<rol>
```

Los *bindings* se especifican de la misma forma, sólo que en lugar de definir la relación entre componentes y conectores, se define entre un componente o conector y un sistema.

Como ejemplo, en la siguiente figura se muestra el sistema *BankSystem* que encapsula los componentes y conectores definidos anteriormente, junto con la especificación PRISMA:

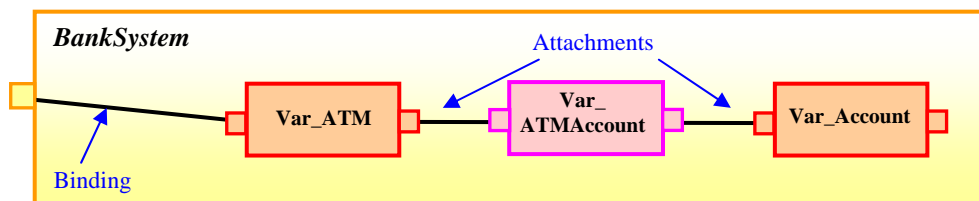


Figura 20 - Ejemplo BankSystem

```

System_type BankSystem
  Ports
    CreditCard_port : ICreditCardTransactions;
  End_Ports

  Variables
    Var_ATM : ATM;
    Var_Account : Account;
    Var_ATMAccount : ATMAccount;
  End_Variables

  Attachments
    Var_ATM.AccountTrans_port ↔ Var_ATMAccount.ATM_role;
    Var_Account.Account_port ↔ Var_ATMAccount.Account_role;
  End_Attachments;

  Bindings
    Var_ATM.VISACreditCard_port ↔ BankSystem.CreditCard_port;
  End_Bindings;

  Initialize
    New() {
      Var_ATM = new ATM(accountId: decimal, initialLocation: LOC);
      Var_Account=new Account(accountId:decimal,initialLocation:LOC);
      Var_ATMAccount = new ATMAccount(initialLocation: LOC);
    }
  End_Initialize;

  Destruction
    Destroy() {
      Var_ATM.destroy();
      Var_Account.destroy();
      Var_ATMAccount.destroy();
    }
  End_Destruction;
End_System_type BankSystem;

```

Como puede observarse, se ha definido un sistema cuyo único puerto, *CreditCard_port*, está relacionado con el puerto *VisaCreditCard_port* del componente *ATM*, mediante una relación de *binding*. Además, los componentes que encapsula el sistema están unidos entre sí mediante relaciones de *attachment*, uniendo el puerto *AccountTrans_port* del componente *ATM* con el rol *ATM_role* del conector *ATMAccount*, y el puerto *Account_port* del componente *Account* con el rol *Account_role* del conector. Con esto se está especificando que todas las peticiones de servicio que lleguen al sistema se redirijan al componente *ATM*, que decidirá cómo procesar dichas peticiones. A su vez, este componente puede requerir

comunicarse con el componente *Account*, y lo hará a través del conector *ATMAccount*, que coordinará las comunicaciones entre ambos.

3.4.2 Configuración Arquitectónica

El nivel de configuración del lenguaje permite definir las instancias y la topología del modelo arquitectónico final. En la tabla siguiente se puede observar la plantilla genérica, muy semejante a la del sistema visto en el punto anterior:

```

1  Architectural Model <nombre_modelo>
2    Variables
      <nom_Vari>: <tipo_componente | tipo_conector | tipo_sistema>;
    End_Variables

3    Attachments
      <nom_Vari>.<nom_puerto> ↔ <nom_Vark>.<nom_rol>;
    End_Attachments;

4    Initialize
      New() {
        <nom_Vari>= new <(nom_componente | nom_conector)>(<args>);
      }
    End_Initialize;

5    Destruction
      Destroy() {
        <nom_Vari>.destroy();
      }
    End_Destruction;
End Architectural Model <nombre_modelo>;

```

Tabla 6 - Plantilla de un modelo arquitectónico PRISMA

Como se puede observar, las secciones que forman parte de un modelo arquitectónico son muy similares a las de los sistemas. La cabecera (sección 1) define el nombre del modelo arquitectónico. La cláusula *Variables* (sección 2) define los componentes, conectores o sistemas que forman el modelo arquitectónico, que a la vez que importa los tipos de la librería de PRISMA, las asocia con un nombre de variable para posteriormente poderles hacer referencia en las siguientes secciones. La especificación de *Attachments* (sección 3) especifica, al igual que en los sistemas, la conexión entre los diferentes elementos arquitectónicos que conforman el modelo.

El bloque *Initialize* (sección 4) especifica cómo instanciar los elementos arquitectónicos que conforman el modelo, proporcionando los valores de inicialización que requieran sus constructores. Pueden seguirse dos aproximaciones. La primera de ellas consiste en especificar en el modelo arquitectónico las instancias que lo forman, proporcionando los correspondientes valores de inicialización. El problema que plantea esta aproximación es que cuando se quieran cambiar los valores de inicialización

deberá modificarse la especificación del modelo arquitectónico. La otra aproximación consiste en definir el modelo arquitectónico como un tipo, es decir, no especificar los valores de inicialización que tomarán las instancias, tan sólo la secuencia de llamadas a los constructores de los elementos arquitectónicos que forman el modelo arquitectónico. Más tarde, cuando se decidiese instanciar el modelo arquitectónico definido, se proporcionarían los valores de inicialización requeridos (ver ejemplo). Con esto, se favorece la reutilización del modelo arquitectónico definido. Además, la inicialización puede ser anidada, es decir, pueden especificarse los valores de inicialización de componentes encapsulados dentro de sistemas, con el objetivo de evitar que los constructores de dichos sistemas acumulen una larga lista de inicialización.

Por último, el bloque de destrucción (sección 5) define qué secuencia de destrucción de los componentes debe seguirse, y si es necesario, indicar servicios adicionales que deban especificarse, como por ejemplo, la persistencia de los datos.

Como ejemplo de modelo arquitectónico, va a tomarse únicamente el sistema definido en el apartado anterior:

```
Architectural Model SimpleBankSystem
  Variables
    Var_BankSystem : BankSystem;
  End_Variables

  Initialize
    New() {
      Var_BankSystem = new BankSystem();
    }
  End_Initialize;

  Destruction
    Destroy() {
      Var_BankSystem.destroy();
    }
  End_Destruction;
End Architectural Model SimpleBankSystem;
```

Como se puede observar en el ejemplo, la definición del modelo arquitectónico está formada únicamente por un único sistema, el definido anteriormente. El bloque de inicialización únicamente define cómo construirlo, pero no lo instancia, con el objetivo de favorecer la reutilización del modelo arquitectónico. La instanciación del modelo arquitectónico se haría de la siguiente forma:

```
BANKSYSTEM = new BankSystem {
  ATM1 = new ATM(123456789, new LOC("tcp://garbi.dsic.upv.es"));
  ACCOUNT1 = new Account(987654321, new LOC("tcp://sigil.dsic.upv.es"));
  ATMAccount1 = new ATMAccount(new LOC("tcp://garbi.dsic.upv.es"));
}
```

Obsérvese cómo se han ido anidando los constructores para inicializar los componentes que forman el sistema. El *Framework* de PRISMA sería el encargado, mediante reflexión de código, de obtener los componentes que

forman el sistema y de pedir al usuario los parámetros de inicialización necesarios, bien de forma gráfica mediante interfaces de usuario, o bien por consola. La otra alternativa consistiría en definir el constructor del sistema final con tantos argumentos como argumentos tuvieran cada uno de los constructores de los componentes/conectores que forman el sistema.

3.5 Otros aspectos avanzados de PRISMA

La especificación del modelo PRISMA abarca otras áreas que no se han cubierto en este capítulo por no incluirse entre los objetivos del presente proyecto. No obstante, la especificación PRISMA también incluye un lenguaje gráfico, basado en UML 1.5, para diseñar la arquitectura del sistema de forma amigable para el usuario. Pueden encontrarse más detalles en [Per03]. En la actualidad, otras líneas de trabajo están definiendo plugins gráficos para diseñar modelos PRISMA en la herramienta Microsoft® Visio 2003, aunque en un futuro también se diseñarán para Rational Rose® y ArgoUML.

Por otra parte, una característica importante que aporta PRISMA es que está concebido para permitir la evolución del modelo en tiempo de ejecución: el modelo arquitectónico podrá reconfigurarse dinámicamente sin necesidad de detener la ejecución de la aplicación. Podrá evolucionar tanto la estructura como el comportamiento de los distintos elementos arquitectónicos que forman el modelo PRISMA, mediante la definición de un metanivel para convertir los componentes en datos y poderlos modificar.

Finalmente, también existe otra línea de investigación que se encarga del estudio de las propiedades de distribución del modelo y de extenderlo para permitir la movilidad, replicación y comunicación distribuida de los elementos arquitectónicos. Para ello, se han definido patrones de distribución que describen situaciones en las cuales la arquitectura software necesita reconfigurar su topología de localización dinámicamente bien moviendo o reemplazando sus elementos arquitectónicos para hacer frente a problemas de distribución, nuevos requerimientos del sistema o otros cambios en el rendimiento de la arquitectura software. Entre los múltiples patrones que se recogen en [Ali03], a continuación se citan algunos: Invocación de servicio, exceso de invocación de un servicio, exceso del volumen de datos intercambiado con un origen, exceso del tiempo de latencia de un mensaje, etc.

IMPLEMENTACIÓN EN .NET

Contenidos del capítulo:

4.1 Diseño de la arquitectura	67
4.1.1 Introducción	67
4.1.2 El Middleware PRISMA	69
4.2 Implementación del Middleware	71
4.2.1 La clase <i>AsyncResult</i>	74
4.2.2 Interfaces	78
4.2.3 Aspectos	79
4.2.4 Componentes y Conectores	89
4.2.5 Sistemas	118
4.2.6 <i>MiddlewareSystem</i>	140
4.3 Resumen del Capítulo	148

IMPLEMENTACIÓN EN .NET

Tras haber presentado las diferentes tecnologías existentes orientadas a aspectos en .NET y el modelo PRISMA, a continuación se describe cómo se ha implementado el modelo sobre la plataforma .NET.

El diseño realizado da soporte a la concurrencia, la comunicación distribuida, la movilidad, la reutilización y la evolución dinámica de código. Éstos son aspectos característicos del modelo arquitectónico PRISMA y proporcionan a sus elementos una gran potencia expresiva. Sirva como ejemplo de la capacidad que brinda el prototipo desarrollado, la propiedad de los componentes de poder agregar o eliminar aspectos, *weavings* o puertos en tiempo de ejecución sin necesidad de recompilar y, en algunos casos, sin detener la ejecución del componente. Esta propiedad es posible gracias a que los componentes se han implementado de forma totalmente independiente de las partes que lo componen, con lo que además también se favorece la reutilización.

Para la ejecución de las aplicaciones PRISMA se ha desarrollado un *middleware* que da soporte a las características requeridas por el modelo, como la ejecución concurrente de los elementos arquitectónicos, la comunicación distribuida, la movilidad o la evolución dinámica.

A continuación se describirá el diseño de la arquitectura, pasando después a detallar cómo se ha implementado el *middleware*.

4.1 Diseño de la arquitectura

4.1.1 Introducción

En la Figura 21 puede observarse cómo las diferentes tecnologías orientadas a aspectos analizadas en el capítulo anterior actúan sobre los distintos niveles que constituyen el .NET *Framework*. Éstas se pueden dividir en tres grupos [Jack04], separados por la forma de realizar el *weaving* entre código base y aspectos:

- **Modificación del motor de ejecución (el CLR):**

Son aquellas aproximaciones que introducen mecanismos para extender la funcionalidad del CLR. Por ejemplo, en JAsCo.Net se insertan *traps*¹⁰ a los ensamblados generados, y en Rapier-Loom.Net se modifican las estructuras internas del CLR para que los aspectos puedan ser llamados desde el código base. Los principales inconvenientes de estas aproximaciones son, por una parte, que el modelo de puntos de unión está limitado por los mecanismos utilizados para hacer el *weaving* en el CLR (por ejemplo, en Rapier-Loom.Net, todos los métodos debían ser virtuales, lo que repercute en el rendimiento. Por otra parte, las herramientas de depuración existentes no pueden ser utilizadas ya que el CLR ha visto modificado su comportamiento.

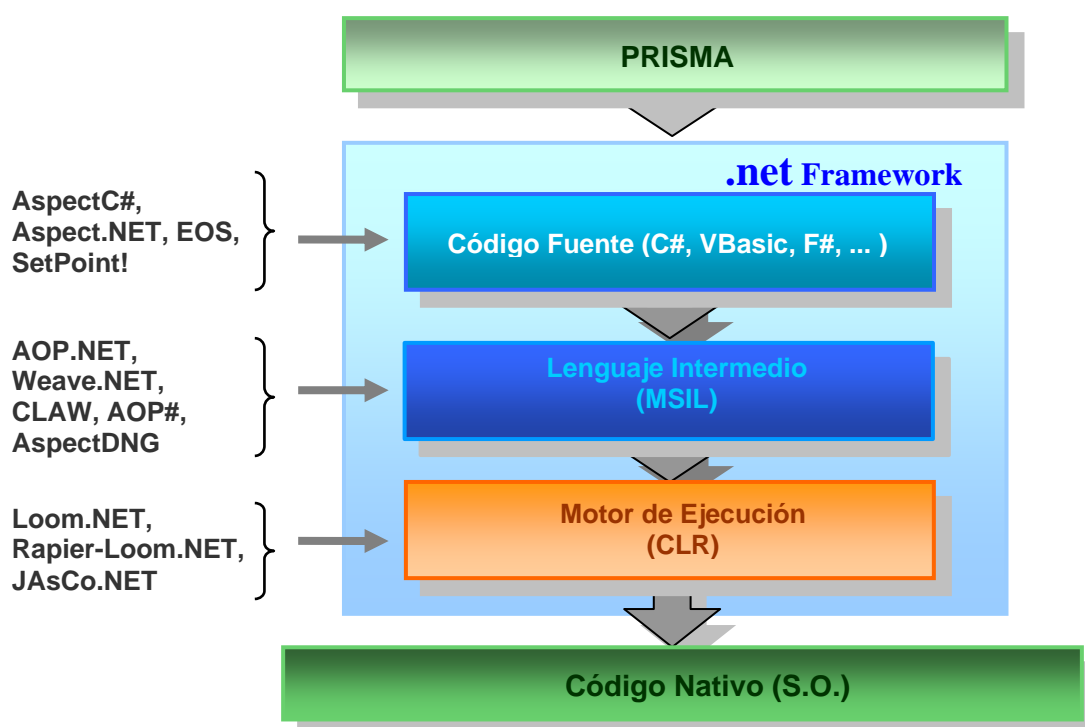


Figura 21 - Aplicación de PRISMA sobre .NET

- **Alteración del código intermedio generado (MSIL):**

Son las aproximaciones que manipulan de alguna forma el lenguaje intermedio generado para enlazar el código base con los aspectos. Para ello, en la mayoría de los casos se modifica el código intermedio de los ensamblados para que sus métodos sean interceptados en tiempo de ejecución y se procesen los aspectos. Aunque se mantiene la independencia del lenguaje utilizado, el principal inconveniente es que al igual que en el caso anterior, tampoco es posible depurar el código fuente.

¹⁰ En este contexto, una *trap* (trampa) es una invocación hacia un método externo.

- **Extensión del lenguaje CLS/CTS:**

Se incluyen en este grupo aquellas aproximaciones que extienden el CLS/CTS para permitir la unión de aspectos en tiempo de compilación. El problema que presentan es que los compiladores desarrollados deben actualizarse al mismo ritmo que la evolución de los lenguajes .NET que extienden, y además, se pierde la independencia del lenguaje.

Los tres grupos descritos tienen el inconveniente adicional de crear una dependencia con la plataforma .NET, con lo que futuras versiones provocarán una actualización de dichas aproximaciones para mantener la compatibilidad.

En cambio, PRISMA actúa a un nivel superior, ya que utiliza un lenguaje de descripción de arquitecturas independiente de cualquier plataforma de desarrollo, lo que le permite incorporar funcionalidad adicional y un mayor nivel de abstracción. Por otro lado, la implementación de PRISMA se ha realizado basándose en los lenguajes estándar definidos, sin añadir ninguna extensión a la plataforma de desarrollo, con el objetivo de no tener que adaptar la implementación a nuevas versiones del CLR, MSIL, etc.

El proceso de crear un modelo PRISMA y ejecutarlo pasa por una serie de etapas (ver Figura 22). En primer lugar, a través de una herramienta visual, el analista diseña y especifica el modelo arquitectónico. En la segunda etapa, a partir del modelo arquitectónico se genera su código intermedio, dependiente de la plataforma de desarrollo, en nuestro caso C#. Por último, el código generado es puesto en ejecución por el *middleware PRISMA*, el cual implementa el modelo PRISMA y proporciona los servicios de distribución y evolución requeridos por el modelo.

En este proyecto se ha construido en C# un prototipo de dicho *middleware* y se han detectado las correspondencias entre el modelo PRISMA y la implementación, lo que permitirá en un futuro construir el compilador.

4.1.2 El Middleware PRISMA

Por una parte, el *Middleware* ofrece una serie de clases que implementan la funcionalidad básica de los tipos PRISMA. Mediante la extensión de estas clases es posible la implementación de las aplicaciones PRISMA.

Por otra parte, el *Middleware* ofrece servicios tanto a componentes, conectores y sistemas del modelo PRISMA que están ejecutándose localmente, como a otros *Middlewares* PRISMA en ejecución en otras máquinas. Algunos de los servicios ofrecidos son la carga de componentes, creación de los hilos de ejecución, gestionar las listas de componentes existentes localmente, etc.

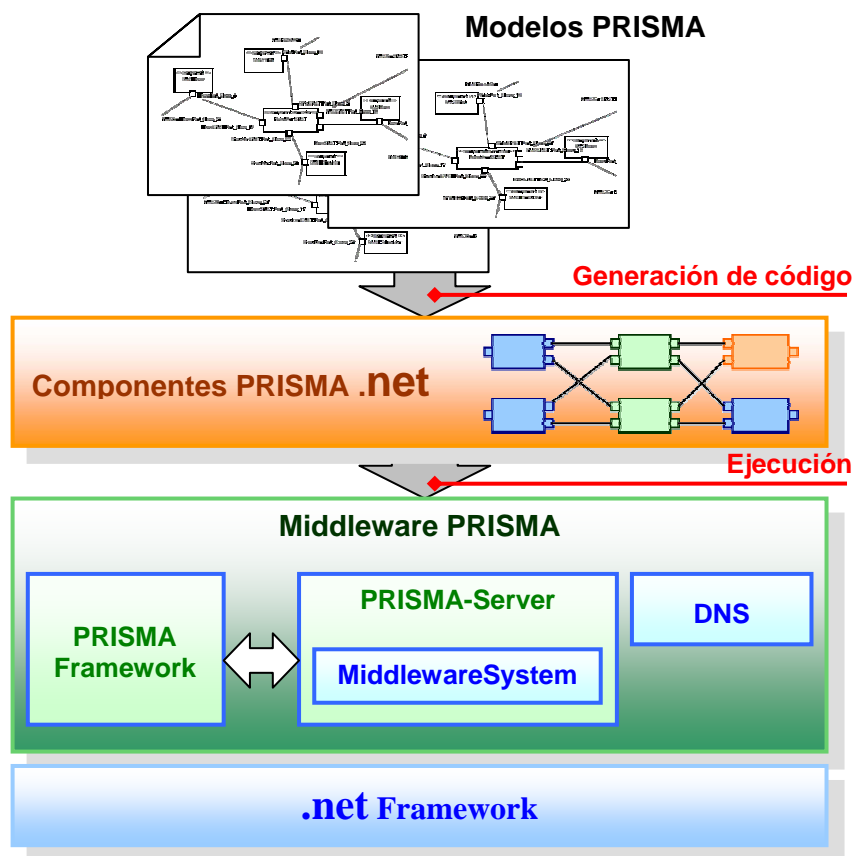


Figura 22 - Arquitectura del Middleware PRISMA

El *middleware* (ver Figura 22) es una capa adicional que se ejecuta sobre la plataforma .NET, formada a su vez por tres módulos:

- **PRISMA Server:** Módulo que contiene el núcleo del *Middleware*, implementado en la clase *MiddlewareSystem*, y proporciona los servicios de gestión, distribución y evolución de los componentes PRISMA.
- **PRISMA Framework:** Consola de administración del *Middleware* que ofrece al usuario los servicios proporcionados por el núcleo, y muestra los mensajes de estado generados por el *middleware*.
- **Servidor DNS:** Servicio de nombres para localizar la ubicación de los distintos componentes del sistema.

Los tres módulos son independientes entre sí, para que en caso de fallar uno de ellos, el sistema pueda recuperarse fácilmente sin más que volver a cargar el módulo. Sin embargo, la tolerancia a fallos no ha sido incorporada, y queda como una tarea futura para próximos prototipos.

Un modelo PRISMA en ejecución puede ser distribuido, tal como se contempla en el lenguaje de descripción de arquitecturas. Esto implica que en cada nodo donde se quiera ejecutar PRISMA debe estar instalado y cargado el *middleware*, ya que además de gestionar los componentes PRISMA también proporciona los servicios de movilidad, replicación, acceso remoto, etc. La arquitectura de dichos nodos es totalmente descentralizada.

Los distintos *middleware* se encargan de gestionar los elementos que están ejecutándose en su nodo y de establecer las comunicaciones con sus *middlewares* vecinos. Los elementos compilados del modelo PRISMA se encuentran distribuidos por los distintos nodos, de forma que si uno de ellos falla, el sistema formado por el conjunto de *middlewares* puede redistribuirse las tareas para mantener estable al sistema.

El prototipo que se describe en este proyecto, aunque diseñado para soportar una arquitectura descentralizada, utiliza un servicio de nombres centralizado para albergar la ubicación de todos los elementos arquitectónicos: es el módulo DNS. No obstante, al estar diseñado como un elemento independiente, la sustitución futura de dicho módulo por una estrategia descentralizada no supondrá un gran impacto sobre el prototipo implementado. Según el ADL de PRISMA mostrado en el capítulo anterior, el servicio de nombres del modelo se encontrará distribuido en los *attachments* y *bindings*, por lo que no será necesario el módulo DNS.

Dada una aplicación PRISMA, se pueden distinguir tres clases de comunicaciones:

- las realizadas por parte de los elementos arquitectónicos de la aplicación al *Middleware*, en las que se solicitan servicios de movilidad o de replicación.
- las comunicaciones entre los distintos elementos arquitectónicos, según lo requieran las especificaciones del modelo (solicitud de servicios unos a otros).
- las realizadas entre los distintos *Middlewares* para averiguar las localizaciones de los distintos elementos, mover elementos, transferir tipos, evolucionar tipos y arquitecturas, actualizar el estado, etc.

Debido a la naturaleza distribuida del modelo, la implementación de las comunicaciones se ha realizado con la tecnología .Net Remoting.

Por otra parte, el PRISMA *Framework* es un módulo opcional, únicamente necesitará cargarse en un nodo desde el que se desee administrar la configuración del sistema. Entre sus funciones están las de cargar modelos PRISMA, permitir la evolución del modelo, comprobar el estado de los distintos elementos, etc.

4.2 Implementación del Middleware

Una vez vista la arquitectura del *middleware PRISMA*, a continuación se describe cómo se han implementado los distintos conceptos PRISMA en .NET. En primer lugar se presentan de forma general los mecanismos que permiten ejecutar los servicios de forma asíncrona, gracias a la clase *AsyncResult*. En segundo lugar se presentan las interfaces y los aspectos,

describiendo las correspondencias entre PRISMA y .NET, cómo se definen los distintos tipos de aspectos y su modelo de ejecución. En tercer lugar se presenta la implementación de los componentes, su modelo de ejecución y cómo gestionan los aspectos, weavings y puertos. En cuarto lugar, se presenta la implementación de los sistemas, que permiten la creación de componentes complejos, y de los attachments y bindings, los cuales establecen el modelo de comunicación entre componentes, conectores y sistemas. Por último, se presenta la implementación de la clase *MiddlewareSystem* y se describen los distintos tipos de servicios ofrecidos.

Los diferentes tipos PRISMA se han implementado en una serie de clases, agrupadas según su funcionalidad en diferentes espacios de nombres, mostrados en la Figura 23. Estas clases proporcionan el comportamiento genérico de los distintos tipos definidos en PRISMA. La implementación de cada tipo específico definido mediante el ADL de PRISMA, es generada por el compilador de acuerdo a las plantillas de código que en este capítulo se presentan. Dichas plantillas únicamente definen la estructura básica de cada tipo, el comportamiento es heredado de las clases genéricas agrupadas en el espacio de nombres de PRISMA.

En los apartados siguientes se describen estas clases y las plantillas de código asociadas. Para cada tipo PRISMA se muestran los diagramas de clases correspondientes a la etapa de diseño. También es preciso indicar que, cuando se mencione al *middleware*, se hace referencia a la clase *MiddlewareSystem*, que proporciona toda la funcionalidad de bajo nivel.

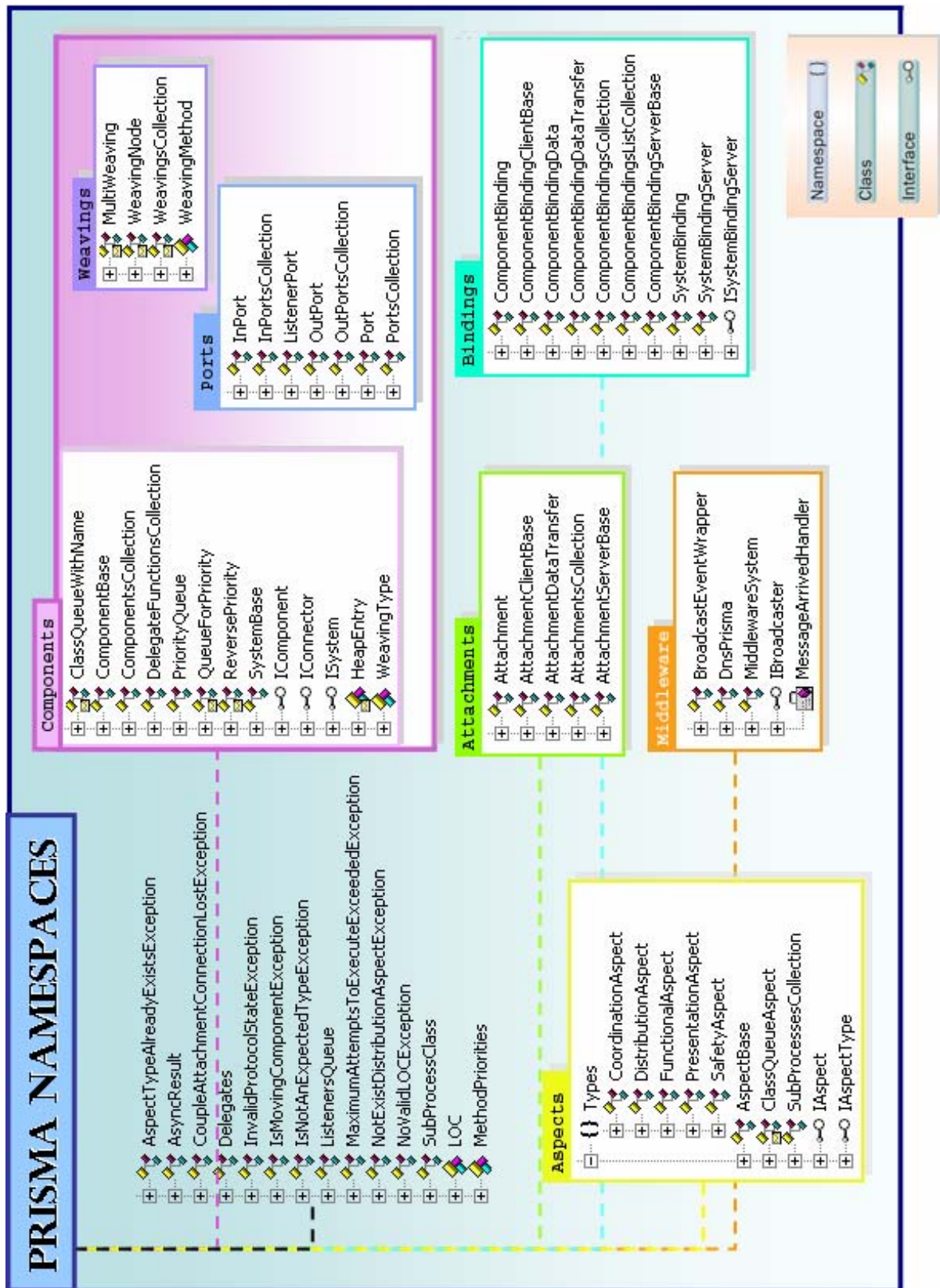


Figura 23 - Espacio de nombres de la implementación de PRISMA

4.2.1 La clase *AsyncResult*

Antes de pasar a los tipos PRISMA, es necesario introducir la clase *AsyncResult*, pues es un tipo fundamental que ha permitido implementar la concurrencia y las llamadas asíncronas entre los diferentes procesos.

En .NET existen mecanismos para realizar llamadas asíncronas, a través de las clases *Delegate* y *AsyncResult*. El cliente de un servicio asíncrono debe crear un delegado para el servicio requerido y ejecutarlo mediante el servicio *BeginInvoke*, obteniendo una estructura *AsyncResult* que le permitirá conocer la finalización del servicio y obtener los resultados. Si el cliente no desea procesar los resultados y desea continuar su proceso interno, necesariamente siempre deberá ejecutar el servicio de terminación *AsyncResult.EndInvoke*. Por otra parte, si dicha estructura se genera en una máquina remota, no puede ser accedida remotamente, pues tiene fuertes dependencias de implementación con la máquina sobre la que se ha generado. Además, la asincronía proporcionada es gestionada por la plataforma de desarrollo, lo cual es un inconveniente para la implementación del modelo PRISMA, ya que éste requiere una serie de mecanismos adicionales para gestionar las peticiones recibidas, establecer un orden de ejecución, redireccionar servicios a otros elementos, etc.

Por ello, se ha creado la clase *AsyncResult* (que no debe confundirse con la proporcionada por la plataforma .NET) cuya finalidad es la de servir como nexo de unión entre el cliente y el proveedor de un determinado servicio. Esta clase permite implementar el comportamiento asíncrono de forma transparente para el cliente, ocultando los detalles específicos de la plataforma de desarrollo (como el uso de delegados), y puede ser extendida para añadir funcionalidad adicional. Por ejemplo, en una primera versión, el tratamiento de excepciones no estaba contemplado. En la versión actual se ha implementado un mecanismo sencillo que permite propagar al cliente de un servicio las excepciones generadas remotamente.

La asincronía en la ejecución de un servicio es proporcionada a través de varios mecanismos, siendo esta clase la estructura que permite posteriormente comprobar si la petición ha sido procesada. El modelo de ejecución mediante el cual se implementa la asincronía, cuyos detalles pueden variar en cada caso concreto, se presenta a continuación. Para ello, se ilustra utilizando un ejemplo concreto.

De forma general, un cliente solicita un servicio a un proveedor de servicios (servidor). Todos los servicios susceptibles de ser invocados asíncronamente devolverán siempre una estructura *AsyncResult*, mediante la cual podrá comprobarse la finalización de dichos servicios o acceder a los resultados devueltos. Por ejemplo, la definición del servicio *Balance* publicada por el servidor sería la siguiente:

```
AsyncResult Balance(ref decimal money);
```

Nótese que este servicio tiene un parámetro de salida en el cual se depositará el saldo de una cuenta, y que devuelve un objeto *AsyncResult*. Debido a que el servidor puede encontrarse ejecutando un servicio solicitado previamente, debe incorporar un mecanismo que le permita almacenar las nuevas peticiones de servicio recibidas para su posterior tratamiento. Por esta razón, por cada petición de servicio recibida, el servidor crea un delegado para ejecutar posteriormente el servicio solicitado y una instancia de la clase *AsyncResult* asociada a dicho servicio. Esta instancia, junto con el delegado, son almacenados en la cola del servidor para ser procesados posteriormente. Por ejemplo, el código correspondiente al cuerpo del método *Balance* implementado en el servidor sería el siguiente:

```
public AsyncResult Balance(ref decimal money) {
    // Creación del delegado hacia el método privado con la
    // implementación de Balance.
    balanceDelegate = new _BalanceDelegate(this._Balance);
    // Creación del vector de argumentos
    object[] args = new object {money};

    // Creación del objeto AsyncResult
    AsyncResult result = new PRISMA.AsyncResult(args.Length);
    // Creación del nodo de la cola
    queueNode newNode = new queueNode(balanceDelegate, args, result);
    // Se encola la petición en la cola de servicios
    queue.Enqueue(newNode);
    // Se devuelve el objeto AsyncResult
    return result;
}
```

La creación de un objeto *AsyncResult* se realiza proporcionando la longitud del vector de parámetros del método al cual va asociado, ya que internamente se crea un vector de dicho tamaño en el que se almacenarán los parámetros devueltos por dicho método. Al cliente le retorna la referencia al objeto *AsyncResult* creado, mediante el cual éste podrá comprobar si su petición de servicio ha sido procesada. A partir de este momento, el cliente podrá continuar realizando sus tareas, hasta que el servidor retorne los resultados del servicio. Esto puede comprobarse mediante la invocación del método *HasResult()*, que indica si el objeto ya tiene los valores devueltos por el servicio asociado. Continuando con el ejemplo anterior, el código correspondiente para que el cliente pueda comprobar si el método *Balance* ya se ha procesado, sería el siguiente:

```
result = Balance(ref money);
// Realización de otras tareas ...
// ...
// Ahora espera hasta obtener el resultado del método Balance.
while (!result.HasResult()) System.Threading.Thread.Sleep(100);
```

Cuando el servidor procese la petición almacenada en la cola, lo que implica ejecutar el delegado y con ello invocar el servicio asociado, depositará los resultados obtenidos en el objeto *AsyncResult* creado anteriormente, y al cual el cliente tiene acceso. Para ello, se invoca el método *SetParameters*

proporcionando como argumento un vector que corresponde a los parámetros devueltos por la ejecución del servicio requerido. El código correspondiente al procesamiento del servicio por parte del servidor es el siguiente:

```
// Se desencola el siguiente servicio de la cola
queueNode newNode = (queueNode) queue.Dequeue();
// Se ejecuta el delegado almacenado en dicho nodo, proporcionándole
// los parámetros proporcionados cuando se creó
newNode.delegate.DynamicInvoke(newNode.Parameters);
// Se actualiza el objeto AsyncResult
newNode.Result.SetParameters(newNode.Parameters);
```

Finalmente, en la parte del cliente, cuando el método *HasResult* devuelva verdadero, éste podrá obtener los resultados del servicio solicitado a través de la invocación del método *GetParameters*, que devolverá el vector de parámetros devuelto por el servicio.

```
// Ya ha finalizado el servicio, se obtienen los resultados
object[] args = result.GetParameters();
```

Por otro lado, las peticiones de servicio recibidas pueden ser reenviadas a otros elementos, que pueden ser locales o remotos, de forma totalmente transparente para el cliente. Esto es posible encadenando entre sí una serie de objetos *AsyncResult* de tal forma que la actualización de sus resultados se propague desde un extremo de la cadena, que puede residir en otra máquina distinta, hasta el otro extremo, el cliente del servicio. Para ello, se ha definido un atributo *derivedResult*, que puede contener una referencia a otro objeto *AsyncResult*. Cuando es invocado el método *HasResult* o *GetParameters*, se comprueba previamente si dicho atributo contiene una referencia válida, en cuyo caso se invoca recursivamente hasta obtener el resultado. Como la clase *AsyncResult* hereda de *MarshalByRefObject*, puede ser accedida remotamente, lo que permite que los resultados de un servicio ejecutado en otra máquina sean accesibles remotamente a través del objeto *AsyncResult*.

En la Figura 24 se muestra el diagrama de secuencia correspondiente para el ejemplo mostrado anteriormente, en el que un objeto *Client* solicita un servicio *Balance* a otro objeto *Server*. Éste, cuando procesa la petición recibida, la reenvía a otro servidor remoto (*RemoteServer*) que le devuelve una referencia al objeto *AsyncResult* creado en aquella máquina. Para que el cliente pueda obtener posteriormente los resultados calculados en la otra máquina, el objeto *Server* invoca el método *SetDerivedParameters*, pasando como parámetro la referencia del objeto *AsyncResult* remoto. De esta forma, cuando el cliente solicita los resultados, éstos son devueltos desde la ubicación remota.

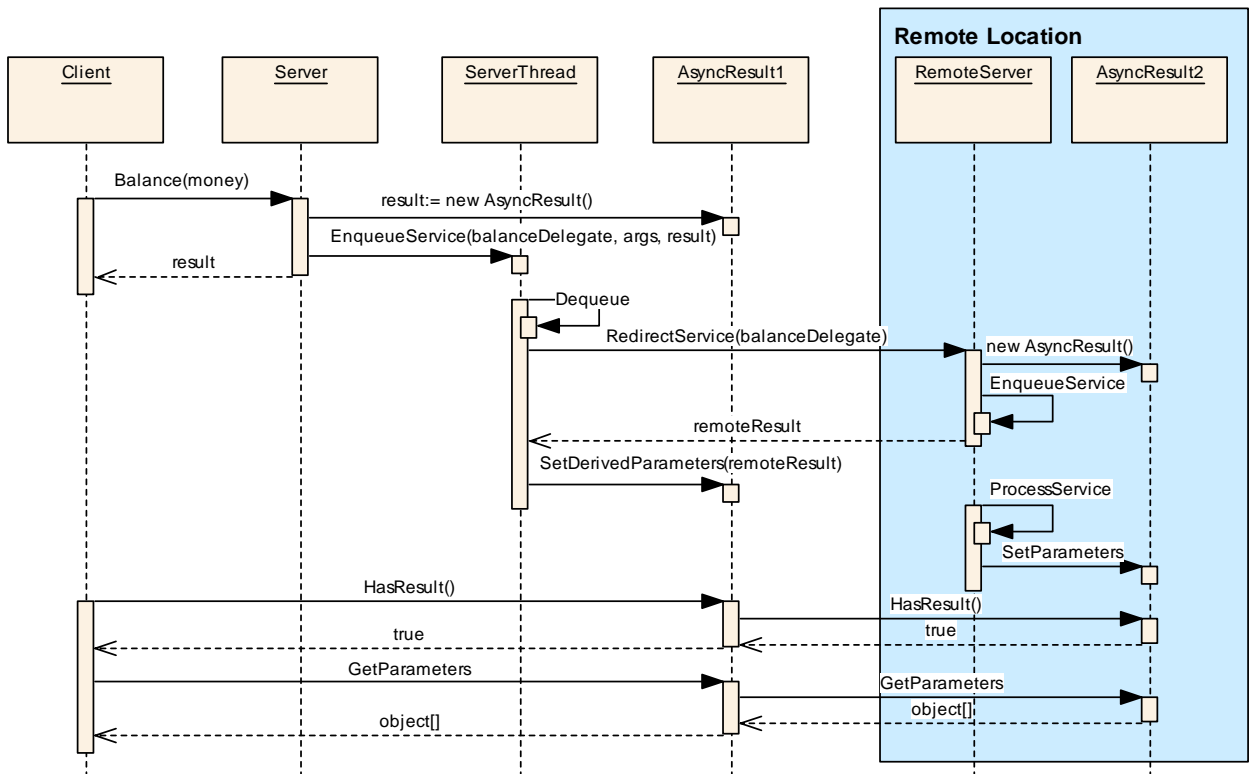


Figura 24 - Esquema de funcionamiento de la clase `AsyncResult`

En la Figura 25 se muestran los atributos y servicios que la clase `AsyncResult` define:

- **parameters**: Vector de objetos en el que se almacenan los parámetros devueltos por la ejecución del servicio asociado.
- **exception, ThrowExceptionToClient()**: el atributo privado *exception* permite almacenar una excepción para ser retransmitida al cliente, cuyo valor es establecido a través de la invocación del método *ThrowExceptionToClient*. Cuando el servidor detecta una excepción provocada por el servicio que estaba en ejecución, almacena dicha excepción en el objeto *asyncResult* mediante la invocación de este método, que establece el atributo *hasResults=true*, para que el cliente sepa que la ejecución del servicio ha finalizado. Cuando el cliente invoque el método *GetParameters*, la excepción será relanzada, pero esta vez en el ámbito de ejecución del cliente.
- **derivedResult, setDerivedParameters()**: contiene una referencia a otro objeto *asyncResult*, permitiendo que un resultado obtenido remotamente se transmita de forma recursiva hasta la instancia a la cual el cliente tiene acceso. El método permite inicializar este atributo con otro objeto *asyncResult* proporcionado como parámetro.
- **hasResults y HasResult()**: el atributo *hasResults* es inicializado a *true* cuando los parámetros son almacenados en el atributo *parameters*. El método *HasResult* devuelve el valor de este atributo,

pero comprueba previamente si existe otro objeto *AsyncResult* asociado al atributo *derivedResult*.

- **SetParameters(), GetParameters():** el método *SetParameters* permite establecer el valor del atributo *parameters*, mientras que el método *GetParameters* permite obtener el vector almacenado en dicho atributo. La invocación del primer método conlleva también que el atributo *hasResults* sea puesto a verdadero, para que el cliente sepa que la ejecución del servicio ha finalizado. La invocación del método *GetParameters* comprueba previamente si hay otro objeto *AsyncResult* asociado a esta instancia. En caso afirmativo, devolverá el resultado de la invocación *derivedResult.GetParameters()*.

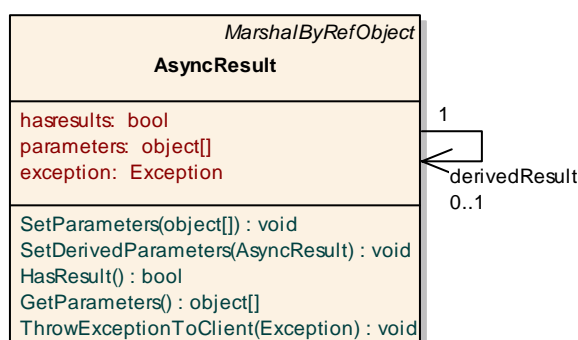


Figura 25 - La clase *AsyncResult*

De este modo, para incorporar la ejecución asíncrona de los servicios ofrecidos por los tipos PRISMA, y permitir el procesamiento concurrente de múltiples servicios, se establece que todo servicio ofrecido por los tipos PRISMA devuelva como resultado un objeto de tipo *AsyncResult*. Esto es posible gracias a que en la especificación PRISMA todos los servicios se comportan como procedimientos, es decir, devuelven sus resultados a través de parámetros de salida, por lo que puede utilizarse el valor de retorno para devolver el objeto correspondiente *AsyncResult*.

4.2.2 Interfaces

Las interfaces PRISMA se corresponden con las interfaces de .NET. Continuando con el ejemplo de la cuenta bancaria presentado en el capítulo anterior (ver sección 3.4.1.1), seguidamente se muestra cómo se han implementado en C# las interfaces de dicho sistema de información:

```

// Definición de la interfaz ICreditCardTransactions
public interface ICreditCardTransactions {
    AsyncResult Withdrawal(decimal quantity, ref decimal money);
    AsyncResult Balance(ref decimal money);
    AsyncResult ChangeAddress(string newAdd);
}

// Definición de la interfaz IMobility
public interface IMobility {

```

```

    AsyncResult Move(LOC newLoc);
}

```

Los parámetros de salida en .NET se identifican por aquellos que llevan la palabra reservada **ref**. Nótese que todos los servicios devuelven un objeto del tipo *AsyncResult*, para permitir el procesamiento asíncrono de los servicios y la ejecución concurrente, como se ha descrito en el apartado anterior.

4.2.3 Aspectos

La implementación de los aspectos PRISMA se compone, por una parte, de la definición del comportamiento especificado para un aspecto en el ADL PRISMA, y por otra parte, de la infraestructura de ejecución que permite la concurrencia de aspectos y la integración con los demás tipos PRISMA.

4.2.3.1 Correspondencia entre PRISMA y la implementación

Para describir la correspondencia entre un aspecto PRISMA y la implementación en .NET, se utilizará como ejemplo el aspecto funcional *BankInteraction*, cuya especificación se mostró en la sección 3.4.1.2, y el código C# asociado.

Un aspecto PRISMA se corresponde con una clase C# cuyo nombre se corresponde con el definido en la especificación (*BankInteraction*), que hereda de otra clase que representa al tipo del aspecto (*FunctionalAspect*) y que implementa la interfaz definida en la especificación (*ICreditCardTransactions*).

```

// Definición del Aspecto Funcional "BankInteraction"
public class BankInteraction : FunctionalAspect, ICreditCardTransactions { }

```

Los atributos de la especificación se corresponden con variables privadas C#, y mediante una enumeración *ProtocolState* se definen los estados posibles de la ejecución de un aspecto. El estado actual en el que se encuentra un aspecto se almacena también en una variable privada.

```

// Definición de atributos PRISMA
int numberId;
string address;
decimal money;

// Definición de los estados posibles del aspecto
enum protocolStates {
    BANKINTERACTION,
    TRANSACTION
}

// Almacena el estado actual del aspecto
protocolStates estado;

```

Un *played_role* se corresponde con la clase *SubprocessClass* (ver Figura 26), que almacena la prioridad y el nombre de los servicios asociados a cada

played_role. Los distintos objetos *SubprocessClass* (los *played_roles* definidos) se almacenan en la clase *SubprocessCollection*, que a su vez está agregada al aspecto a través de la propiedad *subProcessesList*. Con el siguiente fragmento de código se entenderá mejor el proceso de definir y agregar un nuevo *played_role*:

```
// Creación de un nuevo played_role "SERVIDOR"
SubProcessClass subProcessICredit = new SubProcessClass("SERVIDOR");
// Se añaden los métodos que lo forman, junto a su prioridad relativa
subProcessICredit.AddMethod("Withdrawal", 10);
subProcessICredit.AddMethod("Balance", 10);
subProcessICredit.AddMethod("ChangeAddress", 10);
subProcessICredit.AddMethod("Transfer", 10);

// Se añade el nuevo played_role al aspecto
this.subProcessesList.Add(subProcessICredit);
```

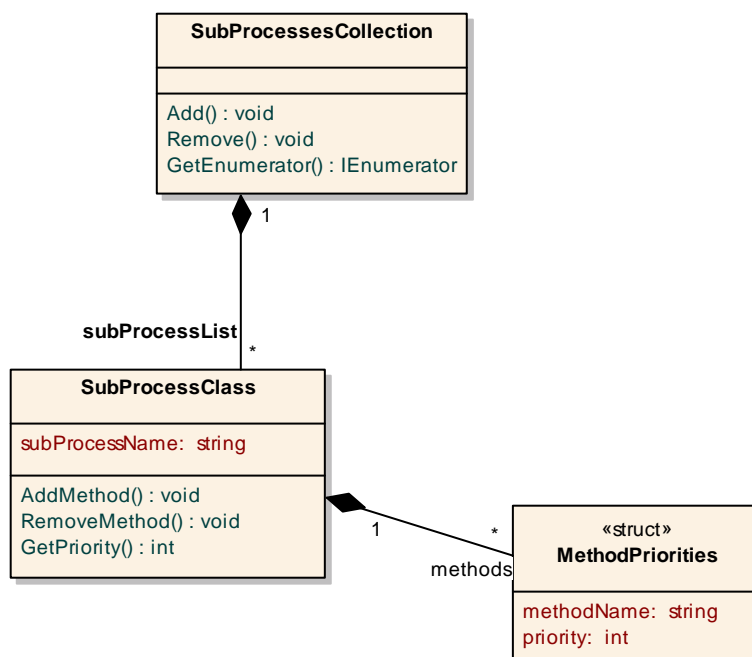


Figura 26 - Clases SubprocessCollection y SubProcessClass

Los servicios PRISMA se corresponden con métodos privados de la clase que implementa al aspecto, en este caso *BankInteraction*. La estructura de un servicio sigue el siguiente patrón:

1. Se comprueba si el estado actual es válido para ejecutar dicho servicio, sino se lanza una excepción.
2. Se comprueba que se cumplan las precondiciones asociadas a dicho servicio.
3. Se comprueba si el estado previo de la valuación se cumple.
4. Se lleva a cabo la funcionalidad asociada.
5. Se comprueba si se debe ejecutar algún disparador.
6. Se asigna el nuevo estado alcanzado a la variable local del estado.

El código correspondiente para definir el servicio *Withdrawal* sería el siguiente:

```
private AsyncResult _Withdrawal(decimal quantity, ref decimal money) {
    // Comprobacion de que el protocolo actual es el correcto
    if (estado != protocolStates.TRANSACTION) {
        throw new InvalidProtocolStateException(this.aspectName, "Withdrawal");
    }

    // Comprobacion de las precondiciones asociadas
    if (quantity > this.money ) {
        throw new Exception("There aren't enough money to withdraw");
    }

    // Comprobación del estado anterior a la valuación
    // NO APLICABLE

    // Ejecución de la funcionalidad asociada
    this.money -= quantity;
    money = this.money;

    // Comprobación de la activación de Triggers
    // NO APLICABLE

    // Nuevo estado alcanzado
    estado = protocolStates.TRANSACTION;
    return null;
}
```

Sin embargo, esta implementación no soporta la evolución de la especificación de un aspecto. En el prototipo actual, la evolución de los aspectos se realiza mediante reconfiguración dinámica, ya que los componentes sí que permiten la sustitución de unos aspectos por otros. De esta forma, para cambiar la especificación de un aspecto en ejecución, deberá primero modificarse la especificación PRISMA del aspecto, compilarse y sustituir al aspecto en ejecución.

4.2.3.2 Tipos de Aspectos

Los tipos de aspecto en PRISMA no están limitados, sino que el desarrollador puede definir nuevos tipos en función de los requisitos del modelo a desarrollar. Por ello, la interfaz *IAspectType* define los atributos y servicios básicos que todo tipo de aspecto debe implementar:

- **AspectType**: Atributo que a través de la reflexión de .NET debe almacenar el objeto *Type* correspondiente al tipo del aspecto, con el objetivo de poder identificar fácilmente el tipo de un aspecto sin tener que recorrer toda la jerarquía de herencia.
- **IsSameTypeAs**: Servicio que comprueba si el aspecto que se le pasa como parámetro es del mismo tipo que el aspecto que lo implementa.
- **CheckPreconditions**: Servicio que define las condiciones que debe cumplir el componente que va a incorporar el tipo del aspecto. Por ejemplo, el aspecto funcional requiere que sea agregado a un componente, mientras que el de coordinación sólo puede agregarse a conectores.

- **ToString**: este servicio únicamente es utilizado para poder obtener una cadena que identifique el tipo del aspecto.

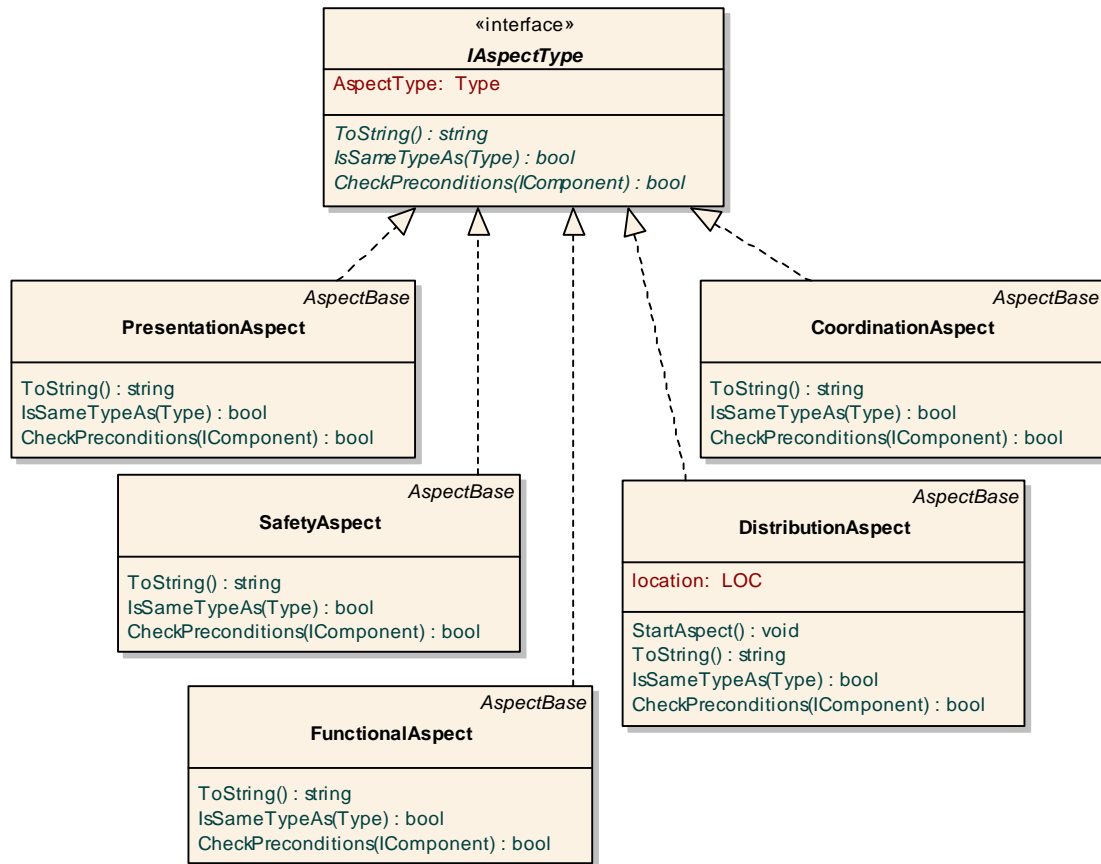


Figura 27 - Interfaz IAAspectType y definiciones de tipos de aspectos

Los distintos tipos de aspectos (*FunctionalAspect*, *DistributionAspect*, etc.) además de implementar la interfaz anterior para identificarse como un tipo de aspecto válido en el sistema, deben heredar su comportamiento de la clase *AspectBase*, que como se verá en el siguiente apartado, define el modelo de ejecución común a todos los aspectos. Por otra parte, si los tipos de aspectos requieren atributos adicionales específicos, pueden ser agregados a este nivel e inicializarse con valores por defecto sobrescribiendo el método *StartAspect()*, que es el encargado de iniciar la ejecución de un aspecto. A continuación se muestra cómo se ha implementado el aspecto funcional, con objetivo de ilustrar estos conceptos:

```

public class FunctionalAspect : AspectBase, IAAspectType {

    // Constructor del tipo "aspecto funcional"
    // El parámetro "aspectName" es el nombre del aspecto y es requerido
    // por el constructor de AspectBase
    public FunctionalAspect(string aspectName) : base(aspectName) {}

    // Devuelve el nombre del tipo del aspecto
    public override string ToString() { return "FunctionalAspect"; }

    // Devuelve el tipo del aspecto
    public Type AspectType { get { return typeof(FunctionalAspect); } }

    // Comprueba si dos aspectos son del mismo tipo o no
  
```

```

public bool IsSameTypeAs(Type obj) { return (this.AspectType.Equals(obj)); }

// Define las condiciones que debe cumplir el componente que va a incorporar
// este tipo de aspecto. El parámetro "comp" es una referencia al componente
// que va a incorporar este tipo de aspecto.
// Devuelve verdadero si cumple las precondiciones definidas.
public bool CheckPreconditions(IComponent comp) {
    // Comprobamos que el componente NO implementa la interfaz IConnector
    if (comp is IConnector)
        throw new Exception("FunctionalAspect can't be added to a Connector. "
            + "It must be a Component");
    return true;
}
}

```

4.2.3.3 Modelo de ejecución

El modelo de ejecución de los aspectos establece cómo se comportan los aspectos y cómo interactúan con los demás tipos del modelo. Como se ha visto en los apartados anteriores, la funcionalidad básica está definida en la clase *AspectBase*, que a su vez implementa la interfaz *IAspect*.

Esta interfaz define los atributos y servicios básicos que todo aspecto debe proporcionar. Permite, mediante el polimorfismo, que el *middleware* pueda tratar de la misma forma a todos los aspectos, o que el componente sepa que se le están agregando aspectos y no otros elementos. Estos atributos y servicios se describen a continuación:

- **aspectName**: es el nombre identificativo del aspecto.
- **aspectThread**: para el modelo de concurrencia es necesario poder obtener el hilo de ejecución asociado a un aspecto, y por ello es necesario un atributo que devuelva una referencia a dicho *thread*.
- **middlewareServer**: es necesaria una referencia al *middleware PRISMA* para aquellas ocasiones en que el aspecto pueda requerir sus servicios, como para la movilidad. Si el componente que posee el aspecto se mueve a otra máquina, dicha referencia debe actualizarse con la referencia del nuevo *middleware*.
- **componentLink**: el aspecto también puede requerir acceder al contexto del componente en el que está alojado, y lo hará a través de esta referencia.
- los servicios **startAspect**, **stopAspect** y **abortAspect**, que permiten iniciar, parar temporalmente (i.e. para moverse) o detener totalmente el aspecto, respectivamente.

Puesto que los aspectos se ejecutan concurrentemente, el modelo de ejecución debe proporcionar mecanismos para garantizar el procesado de la petición de un servicio mientras el aspecto está ocupado procesando otro, sin bloquear la ejecución del cliente que solicita el servicio. Para dar soporte a esta concurrencia, se ha implementado un sistema de colas en el cual se almacenan las distintas peticiones a medida que son recibidas, mientras el hilo de ejecución del aspecto las va procesando. Este mecanismo es

proporcionado por la clase *AspectBase* (ver Figura 28), de la cual todos los tipos de aspecto heredan su comportamiento.

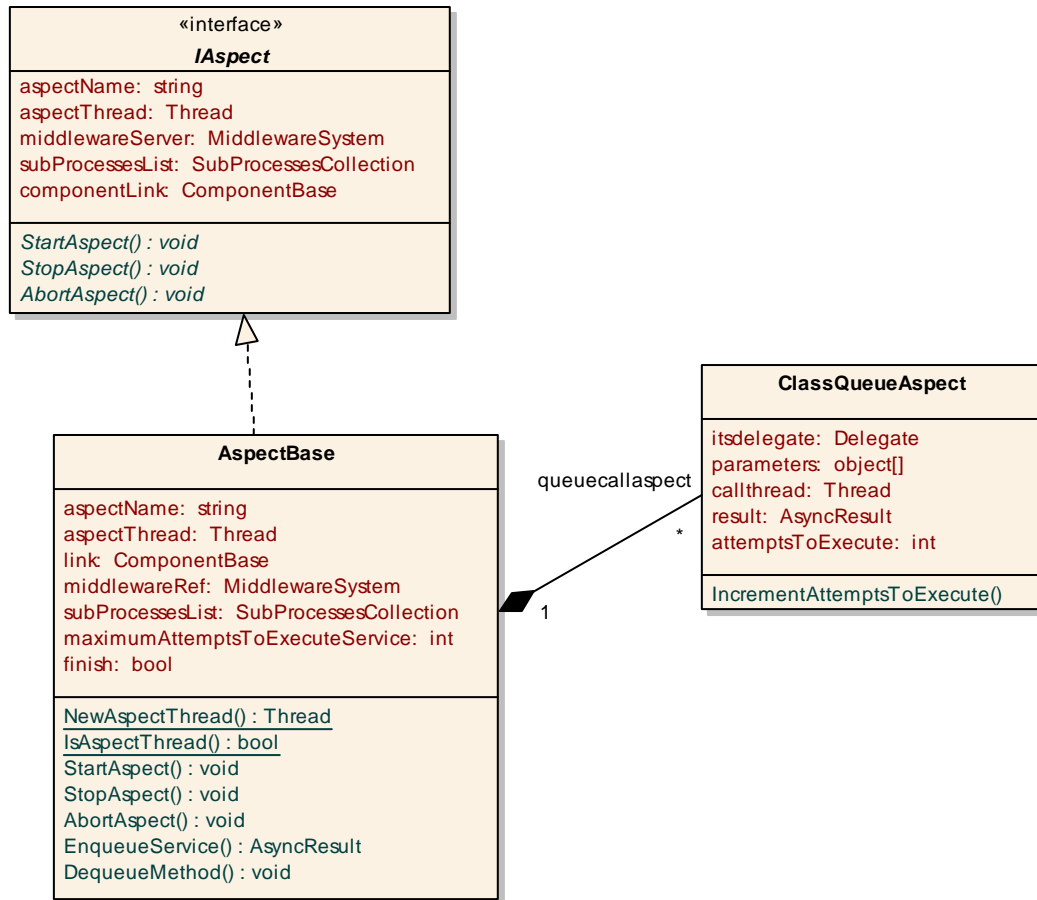


Figura 28 - Interfaz IAAspect y clase AspectBase

La cola del aspecto está sincronizada para tolerar el acceso concurrente de múltiples hilos, pues cada petición de servicio puede estar originada por un hilo distinto (por ejemplo, como se verá más adelante, por el hilo asociado a un *weaving*, por el hilo del componente para procesar una petición directa, etc.), y puede darse el caso que dos hilos intenten encolar a la vez. Los objetos que se almacenan en dicha cola son instancias de la clase *ClassQueueAspect*, que almacena la información necesaria para ejecutar cada servicio cuando le llegue su turno:

- **itsDelegate**: contiene un delegado que apunta al servicio que debe ejecutarse.
- **parameters**: parámetros que se requiere el servicio a ejecutar.
- **callThread**: referencia al hilo que ha originado la petición de servicio, para el caso en que la petición sea síncrona (por lo que el hilo del cliente habrá quedado suspendido en espera de la finalización del servicio) y por tanto necesite ser despertado.

- **result**: instancia de la clase *AsyncResult*, en la que se almacenarán los resultados de la ejecución del servicio, para que puedan ser convenientemente procesados por una ejecución asíncrona.
- **attemptsToExecute**: es un contador que indica las veces que se ha intentado poner un servicio en ejecución sin éxito.
- **IncrementAttemptsToExecute()**: este servicio incrementa el atributo anterior .

La política de planificación de servicios, cuya implementación se encuentra en el servicio *AspectBase.DequeueMethod()*, es compleja, pues tiene que preservar el orden de las peticiones de servicio recibidas, teniendo en cuenta que existen servicios con mayor prioridad que otros (tal como se define en los subprocesos del aspecto), y que hay servicios que tal vez no se puedan ejecutar a causa de que el estado del protocolo del aspecto no es válido para su ejecución. El proceso seguido ha sido ejecutar los servicios en el mismo orden en que han sido encolados, y si a causa del protocolo no se pueden ejecutar, se incrementa el contador *attemptsToExecute* descrito anteriormente y se pasa a ejecutar el siguiente servicio. Cuando se consigue ejecutar con éxito un servicio, la siguiente iteración (en la que tal vez se hayan añadido nuevos servicios al final de la cola) volverá a empezar la ejecución del servicio desde el principio, dando la oportunidad de ejecutar los servicios que pudieran haber fallado anteriormente, ya que el protocolo del aspecto puede haber cambiado. Sin embargo, si el atributo *attemptsToExecute* es mayor que una constante definida en *AspectBase.maximumAttemptsToExecuteService*, el servicio finalmente es descartado lanzándose una excepción que, a través del elemento *AsyncResult* es transmitida al cliente. Todo este proceso está debidamente protegido frente al acceso concurrente mediante mecanismos de sincronización y ejecuciones atómicas, ya que mientras se está procesando un servicio, la cola del aspecto puede ser modificada con nuevos servicios a procesar.

La estructura de un aspecto se divide en una parte pública y una parte privada. La parte privada contiene las acciones a realizar por el servicio, cuya estructura es la comentada en el apartado 4.2.3.1. Los servicios privados se nombran igual que en la especificación del aspecto añadiendo como prefijo el símbolo “_” para distinguirlos de los métodos públicos. De esta forma, el servicio privado *Withdrawal* tendría la siguiente signatura:

```
private AsyncResult _Withdrawal(decimal quantity, ref decimal money) { ... }
```

La parte pública, que es por donde llegan las peticiones de servicio, se compone de los mismos servicios que publica el aspecto, con idéntico nombre y signatura, cuya finalidad es encolar las peticiones recibidas en la cola interna mediante una llamada al método heredado *EnqueueService()*. Siguiendo con el ejemplo anterior, la definición del método público para el servicio *Withdrawal* se realizaría de la siguiente manera:

```
public AsyncResult Withdrawal(decimal quantity, ref decimal money) {
    return EnqueueService(withdrawalDelegate, quantity, money);
}
```

El método heredado *AspectBase.EnqueueService* es el encargado de añadir la petición de servicio en la cola. Para ello, necesita como parámetros un delegado del servicio privado a ejecutar y una lista variable de los argumentos requeridos, devolviendo un objeto del tipo *AsyncResult*. Internamente se encarga de crear una instancia *ClassQueueAspect*, inicializarla convenientemente y encolarla. De esta forma, el código de la parte pública, que deberá generarse automáticamente por cada servicio definido por un aspecto, será mínimo.

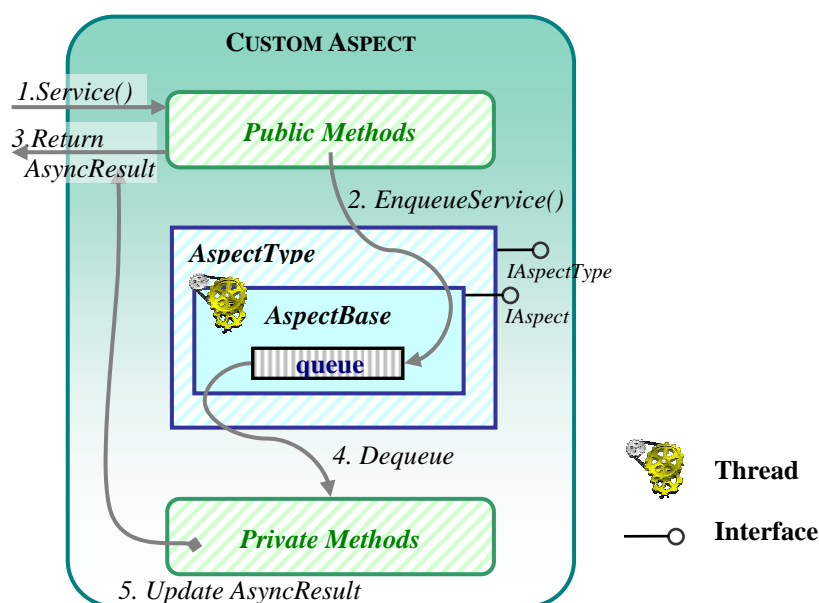


Figura 29 - Esquema de ejecución de un Aspecto

La creación de delegados en C# debe seguir tres pasos. En primer lugar deben declararse como tipos (con la misma signatura que el servicio al cual apuntan), en segundo lugar definir sus instancias, y en tercer lugar instanciarse con el método y el objeto donde se deben ejecutar:

```
// Definición del tipo de delegado WithdrawalDelegate
public delegate AsyncResult WithdrawalDelegate(decimal quantity,
    ref decimal money);
// Definición de una instancia del delegado
[NonSerialized] WithdrawalDelegate withdrawalDelegate;
// Instanciación del delegado anterior
withdrawalDelegate = new WithdrawalDelegate(this._Withdrawal);
```

Es por ello que también deberá generarse el código correspondiente a la definición de estos delegados por el compilador final. Además, como un delegado va siempre asociado a un servicio definido en una interfaz, se ha determinado que el lugar más apropiado para su generación es junto a la definición de las interfaces.

Finalmente, los servicios *startAspect*, *stopAspect* y *abortAspect* son necesarios para iniciar, parar o detener el hilo que procesa los servicios del

aspecto. Por ejemplo, cuando un componente va a ser movido junto con sus correspondientes aspectos, debe detenerse previamente la ejecución del aspecto. Además, debido a que en la cola de los aspectos se almacena el hilo del cliente en espera, y como los hilos no pueden ser serializados (y por tanto no pueden ser movidos), la movilidad de los aspectos queda limitada a que la cola de servicios a procesar esté vacía. Es por esta razón que el servicio *StopAspect* para detener el aspecto pasa por tres fases. En primer lugar no admite nuevas peticiones, por lo que éstas quedan almacenadas en el componente que las ha generado. En segundo lugar, termina de procesar las peticiones pendientes en la cola. En tercer lugar, el hilo se finaliza cuando todas las peticiones han sido procesadas.

Las clases que implementan los aspectos deben ser etiquetadas con el atributo *Serializable*, para permitir que su estado pueda distribuirse a otras máquinas mediante .NET Remoting. Sin embargo, debido a que en la versión actual de .NET los delegados no se comportan correctamente al ser serializados, éstos deben ser marcados como *NotSerialized* y vueltos a crear al llegar al destino.

Toda la funcionalidad asociada a un aspecto se empaqueta en un ensamblado y se almacena en la librería de tipos PRISMA, con el objetivo de facilitar la reutilización, la distribución por la red y la futura integración con un componente.

A continuación se muestra la estructura global de un aspecto en C#, omitiendo aquellas secciones de código no relevantes:

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Middleware;
using PRISMA.Attachments;
using PRISMA.Components.Ports;

namespace CuentaBancaria {
    // Definición del Aspecto Funcional "BankInteraction"
    [Serializable]
    Public class BankInteraction: FunctionalAspect, ICreditCardTransactions {

        Definicion de atributos PRISMA

        Definicion de los estados posibles del aspecto

        #region Definición de delegados
        [NonSerialized] WithdrawalDelegate withdrawalDelegate;
        // ...
        #endregion

        // Constructor del Aspecto: Inicialización
        public BankInteraction(decimal accountId) : base("BankInteraction") {

            // Estado inicial
            estado = protocolStates.BANKINTERACTION;

            // Inicialización de las vbles del aspecto
            this.numberId = accountId;
        }
    }
}
```

```
// ...

// Creación de la estructura de subprocesos para este aspecto
SubProcessClass subProcessICredit = new SubProcessClass("SERVER");
// ...

// Asignacion del nuevo estado alcanzado
estado = protocolStates.TRANSACTION;
}

#region Miembros Públicos de ICreditCardTransactions
public AsyncResult Withdrawal(decimal quantity, ref decimal money) {
    return EnqueueService(withdrawalDelegate, quantity, money);
}
// ...
#endregion

#region Miembros Privados de ICreditCardTransactions
private AsyncResult _Withdrawal(decimal quantity, ref decimal money) {
    ... }
// ...
#endregion

// Sobreescribimos el metodo StartAspect para inicializar los delegados
public override void StartAspect() {
    // Instanciacion de los delegados internos
    withdrawalDelegate = new WithdrawalDelegate(this._Withdrawal);
    // ...
    // Llamamos al padre
    base.StartAspect ();
}
}
}
```

4.2.4 Componentes y Conectores

Un componente PRISMA integra en su interior sus aspectos y ofrece una interfaz externa que es la agregación de las interfaces de sus aspectos. Esta interfaz es publicada al resto de elementos arquitectónicos a través de los puertos que forman el componente, que establecen qué interfaces serán publicadas. En .NET el componente se ha traducido por una clase que además de integrar dinámicamente en su interior los aspectos y los puertos de comunicación, también realiza dinámicamente los *weavings* entre los aspectos y redirecciona adecuadamente las peticiones de servicio (que pueden ser de entrada o de salida) entre puertos y aspectos. El funcionamiento de los conectores se ha implementado de la misma forma que los componentes.

El modelo de ejecución de los componentes proporciona los mecanismos necesarios para la gestión de la concurrencia, la gestión de aspectos, la gestión de *weavings* y la redirección de servicios desde y hacia los puertos. En los siguientes apartados se describe cómo se han implementado los distintos procesos. Finalmente, se muestra la correspondencia entre los componentes y conectores PRISMA y su implementación en C#.

4.2.4.1 Modelo de ejecución

De la misma forma que en los aspectos se ha definido una interfaz común con los servicios básicos que deben ofrecer, para los componentes se ha definido la interfaz *IComponent* (ver Figura 30), cuyos atributos y servicios se describen a continuación:

- **componentName**, **componentThread** y **middlewareServer**: Atributos que almacenan el nombre, una referencia al hilo del componente y una referencia al middleware, respectivamente.
- **isStopping**, **isMoving**: Atributos que permiten comprobar si el componente está en proceso de detenerse o de moverse, por lo que no aceptará nuevas peticiones.
- **inPorts**, **outPorts**: Atributos que almacenan las referencias a los puertos de entrada (recepción de peticiones de servicio) y a los puertos de salida (peticiones de servicio a otros componentes externos) que integra el componente.
- **Start**, **Stop**, **Abort**: Servicios que permiten iniciar, parar o detener al componente.
- **EnqueueService**: Método invocado por los puertos para añadir las peticiones de servicio entrantes en la cola del componente.

- **ProcessService**: Método privado, invocado por el hilo de ejecución del componente para procesar una petición de servicio añadida a la cola.
- **AddAspect, RemoveAspect**: Servicios para añadir o eliminar un aspecto.
- **GetAspect**: dado el tipo de un aspecto, este servicio comprueba si existe en el componente, y en caso afirmativo, devuelve su referencia.
- **AddWeaving, RemoveWeaving**: Servicios para añadir o eliminar un weaving entre dos aspectos, respectivamente.
- **IsWeaved**: Servicio de consulta que permite comprobar si un aspecto tiene un *weaving* con otro aspecto. Es necesario consultar previamente si un aspecto tiene weavings relacionados antes de eliminarlo.

Todos estos servicios son implementados por la clase *ComponentBase*, de la cual todos los componentes y conectores heredan su comportamiento.

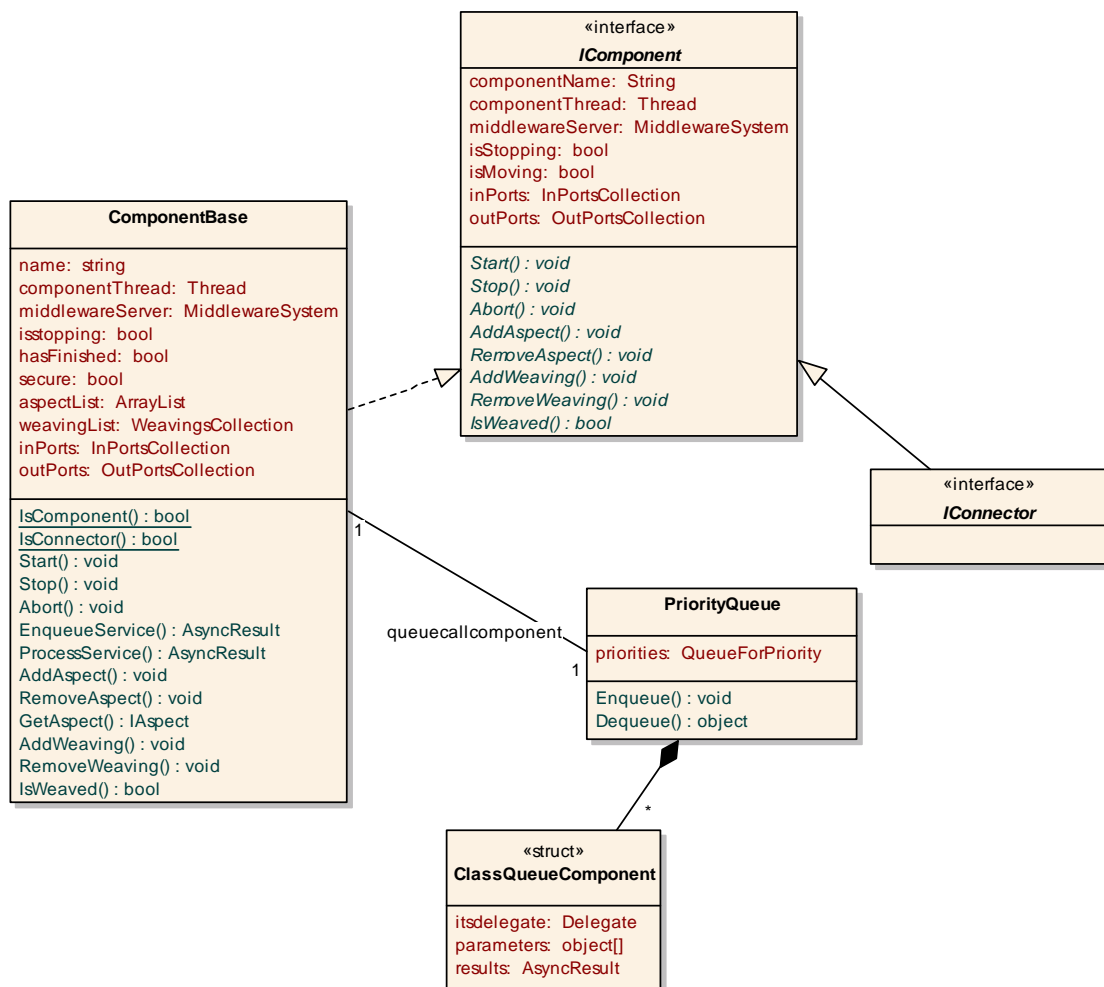


Figura 30 - Interfaces IComponent, IConector y clase ComponentBase

Los conectores heredan de *ComponentBase* debido a que su comportamiento es idéntico al de los componentes, salvo que en lugar de puertos tienen roles y en lugar de un aspecto funcional tienen uno de coordinación. Como no se han detectado atributos y servicios básicos adicionales a los de un componente, el conector se ha definido de la misma forma que un componente con la salvedad de que implementa la interfaz *IConnector*. Esta interfaz a su vez hereda de *IComponent* y tan sólo se utiliza como mecanismo de identificación frente a componentes. Por ello, se han añadido dos métodos de clase en *ComponentBase* que permiten comprobar cuándo una entidad que implementa *IComponent* es componente o conector. Estos servicios son *IsComponent()* y *IsConnector()*.

Las diferencias de comportamiento entre componentes y conectores se han resuelto de diferentes maneras. En la implementación realizada, los roles PRISMA se traducen a puertos, cuya implementación se verá más adelante, relegándose al *middleware* la tarea de comprobar que los conectores se comuniquen sólo con componentes (y viceversa). Por otra parte, la comprobación de que un conector deba tener un aspecto de coordinación (y un componente el aspecto funcional) se realiza en el momento de añadir un nuevo aspecto al componente o conector mediante la llamada al servicio *AddAspect*. Este método, antes de agregar el aspecto al componente comprueba que se cumplen las precondiciones establecidas en el tipo del aspecto, ejecutando el servicio *IAspectType.CheckPreconditions*, como se comentó en el apartado 4.2.3.2.

Un componente está formado por los puertos de entrada y de salida, por una cola de prioridades en la que se almacenan los servicios a procesar, un gestor de *weavings* y los aspectos (ver Figura 31).

- Los puertos de entrada (*inPorts*) son entidades que publican a los demás elementos arquitectónicos los servicios que ofrece el componente. Publican la interfaz que implementa un aspecto integrado en el componente y encolan las peticiones recibidas en la cola interna.
- Los puertos de salida (*outPorts*) son aquellos elementos que proporcionan los mecanismos necesarios para enviar peticiones de servicio a otros componentes. Cada puerto de salida únicamente admite servicios de una determinada interfaz y sólo se conectará con aquellos componentes cuyos puertos de entrada publiquen idéntica interfaz. Tanto los puertos de entrada como los de salida son seguros para el acceso multihilos.
- La cola de prioridades (*PriorityQueue*) almacena las diferentes peticiones de servicio recibidas hasta que puedan ser redirigidas por el componente hacia el aspecto correspondiente. En caso de conflicto, primero se entregará al aspecto el servicio más prioritario.
- El gestor de *weavings* (*WeavingsCollection*) se encarga de comprobar, por cada servicio solicitado a un aspecto, si tiene *weavings* o no. En

caso negativo, se redirige la petición al aspecto correspondiente. En caso afirmativo, se procesa el weaving de forma síncrona.

- Los aspectos se almacenan en una lista dinámica del componente, existiendo únicamente uno de cada tipo.

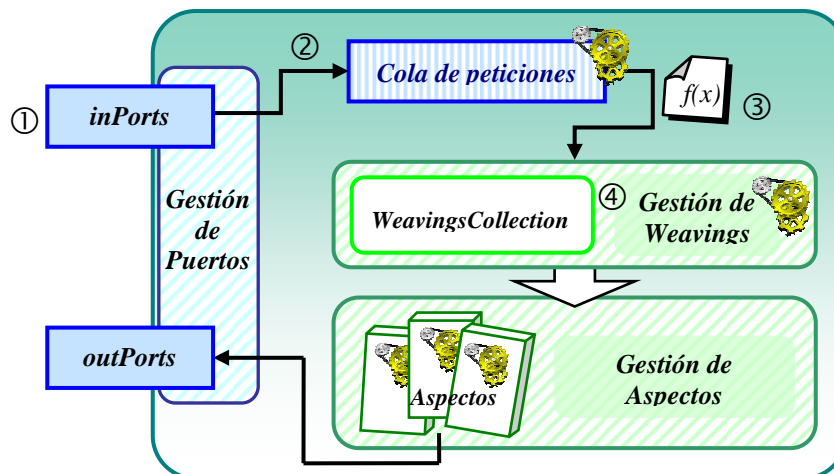


Figura 31 - Estructura interna de un componente en .Net

Cuando un componente externo solicita un servicio al componente, éste es solicitado al puerto que publica la interfaz de dicho servicio (paso 1, Figura 31), que se encarga de reenviarlo a la cola interna del componente (paso 2, Figura 31). Ésta es una cola ordenada por prioridades (clase *PriorityQueue*) en la que se almacenan los servicios a ejecutar por los aspectos. Dichas prioridades corresponden a las declaradas por los subprocesos de los aspectos, mediante los cuales puede establecerse una ordenación de los servicios a ejecutar. La información requerida por cada servicio a ejecutar se encapsula en objetos de tipo *ClassQueueComponent*. Estos objetos contienen un delegado hacia el método del aspecto a ejecutar, un vector con los parámetros necesarios y una referencia al objeto *AsyncResult* creado para devolver el resultado cuando el servicio sea procesado.

Debido a que a un componente le pueden llegar múltiples peticiones de servicio por los distintos puertos que publica, y todos confluyen en la cola interna, ésta se ha protegido para el acceso concurrente mediante el uso de monitores, de tal forma que únicamente un hilo pueda estar a la vez añadiendo servicios en la cola. El reenvío de las peticiones de servicio almacenadas a los aspectos correspondientes, y en su caso, la aplicación de los diferentes *weavings*, es realizada por un hilo de ejecución propio del componente (paso 3, Figura 31). Cuando un servicio va a ser procesado, en primer lugar se comprueba si tiene weaving asociado. Si no lo tiene la llamada es directa, ejecutándose el delegado de forma asíncrona, con la finalidad de que el hilo del componente pueda procesar las siguientes peticiones lo más pronto posible. Si lo tiene, el servicio es redirigido al gestor de weavings (paso 4, Figura 31), el cual se encargará de procesar todos los servicios relacionados, liberando al hilo del componente de esta tarea.

De la misma forma que en los aspectos, el hilo de ejecución del componente es puesto en ejecución cuando el método *Start* es invocado por el *middleware*, y es detenido cuando se invocan los servicios *Stop* (parada segura) o *Abort* (detención completa). La parada o detención de un componente implica la parada o detención de cada uno de los aspectos que lo forman, invocando secuencialmente los servicios *StopAspect()* o *AbortAspect()*, respectivamente.

La parada segura (*Stop*) de un componente es posible mediante una serie de flags internos que permiten sincronizar la parada de las distintas operaciones hasta alcanzar un estado seguro que permita posteriormente reanudar la ejecución en el punto que se había quedado. Un componente está en un estado seguro cuando no quedan peticiones en las colas internas de sus aspectos (pues dichas peticiones no son serializables, como se comentó), no existe ningún servicio en ejecución en los aspectos, ni tampoco se interrumpe ninguna inserción de servicios en la cola. El primero de ellos es el flag *secure*, el cual se utiliza para impedir que el componente sea parado o movido mientras está realizando operaciones atómicas que no pueden ser interrumpidas, como encolar un servicio. El flag *isStopping* indica que el componente va a detenerse y es activado cuando se invoca el servicio *Stop()*. El flag *isMoving* indica que el componente va a moverse y es activado por el aspecto de distribución cuando éste lo estime oportuno.

Cuando uno de estos dos flags está activo, se restringe el encolado de nuevas peticiones de servicio y únicamente son permitidas aquellas provenientes de los aspectos internos. Las peticiones de servicio rechazadas son notificadas al cliente mediante una excepción del tipo *IsMovingComponentException*, que normalmente se traducirá en un reintento por parte de los *attachments* hasta que el componente esté de nuevo activo. En cambio, si la petición es realizada por un aspecto interno, ésta es aceptada y encolada con la máxima prioridad, pues los servicios pendientes de ejecución de un aspecto deben finalizar para que el componente pueda moverse. El hilo de ejecución del componente también deja de reenviar peticiones a los aspectos, excepto los de máxima prioridad. Ésto es necesario para evitar interbloqueos, ya que cuando el componente va a pararse, ha de esperar a que los aspectos acaben de procesar los servicios pendientes. Sin embargo, si un servicio en ejecución de un aspecto requiere otro servicio del componente, y éste ya no admite más peticiones, el aspecto se quedaría a la espera indefinidamente, con lo componente y aspecto quedarían en espera mutua sin llegar nunca a poder detenerse en un estado consistente.

4.2.4.2 Gestión de Aspectos

Para unir los aspectos al componente se consideraron dos aproximaciones:

- **Estática**
La aproximación más sencilla y eficiente consiste en instanciar cada aspecto en una variable del tipo del aspecto (*FunctionalAspect*, *DistributionAspect*, etc.) en el momento de la creación del componente, de forma totalmente estática. Esto implica que los aspectos deberían crearse en el código generado a partir de las especificaciones PRISMA, sin utilizar la herencia para heredar el comportamiento. Por lo tanto, para añadir nuevos aspectos o eliminarlos, se debería emitir código o recompilar el componente. El inconveniente de esta aproximación es que proporciona muy poca flexibilidad al cambio.
- **Dinámica:**
supone la utilización de una lista dinámica (un *ArrayList*) y una serie de métodos para añadir o eliminar aspectos de dicha lista, cumpliendo la restricción de que no existan dos aspectos del mismo tipo instanciados en un componente. La invocación de servicios de un aspecto supone recorrer la lista dinámica para obtener la referencia a un aspecto. El problema de esta solución es el incremento del coste computacional que se produce cuando se realizan muchas llamadas a los aspectos. La principal ventaja es la gran flexibilidad que aporta para añadir/eliminar aspectos.

De las dos aproximaciones anteriores se ha implementado una solución intermedia, con la eficiencia de tener accesible en una variable la referencia hacia un aspecto y la flexibilidad proporcionada por las listas dinámicas. Como en PRISMA el número de tipos de aspectos no es limitado, éstos se han almacenado en una lista dinámica, de forma que puedan ser accedidos todos los aspectos de forma secuencial para realizar diferentes operaciones, como la detención de los aspectos, la comprobación de que un tipo no ha sido ya agregado, etc. Por otra parte, como los puertos de entrada publican únicamente una interfaz implementada por un aspecto, en cada puerto se han definido una serie de delegados que apuntan a cada uno de los servicios ofrecidos por el aspecto asociado. De esta forma, cuando se crea un *inPort* se instancian los delegados que apuntan a cada uno de los respectivos servicios de los aspectos, con lo que la invocación de un aspecto es directa y no necesita recorrer cada vez la lista dinámica de aspectos.

Los aspectos pueden ser agregados o eliminados en tiempo de ejecución sin la necesidad de detener el componente. La agregación de aspectos se realiza mediante la llamada al método *AddAspect(IAspect ref_aspect)*. Este método en primer lugar comprueba que el aspecto que va a agregar no sea de un tipo de aspecto de los que ya han sido agregados al componente. En segundo lugar, comprueba que el componente cumple las precondiciones especificadas en el tipo del aspecto. Después actualiza las referencias del

aspecto hacia el componente y el *middleware* y lo añade en la lista dinámica de aspectos. Finalmente, aunque no está implementado por no encontrarse entre los objetivos de este proyecto, se emitiría el código correspondiente en el constructor del componente para garantizar la persistencia de los cambios. Sin embargo, los aspectos no tendrán visibilidad hacia el exterior del componente hasta que no se creen los puertos relacionados que publiquen su interfaz. Para añadir el aspecto *BankInteraction* al componente *Account* se invocaría al método *AddAspect* pasándole como parámetro una instancia del tipo *IAspect*, como el aspecto que se definió anteriormente:

```
// Creación del aspecto funcional
AddAspect(new BankInteraction(numberID, customer, address, money));
```

La eliminación de aspectos se realiza mediante la llamada al método *RemoveAspect(Type AspectType, bool removeWeavings)*, que permite especificar el tipo de aspecto a eliminar y si se deben o no eliminar los *weavings* asociados. En primer lugar se eliminan los aspectos de la lista dinámica y los puertos correspondientes que estaban asociados al aspecto, con la finalidad de que no se puedan recibir nuevas peticiones. En segundo lugar, se realiza una detención segura del aspecto, dando un margen de actividad para finalizar los servicios que estuviera ejecutando. Si ese margen es superado, se obliga al aspecto a detenerse por completo. Finalmente, se debería emitir código para garantizar la persistencia de los cambios en el disco.

El componente también proporciona el método *GetAspect(Type AspectType)* que permite obtener la referencia del aspecto de un tipo concreto o comprobar si dicho tipo de aspecto ha sido agregado al componente. Por ejemplo:

```
// Obtenemos referencias a los aspectos
IAspect functionalAspect = GetAspect(typeof(FunctionalAspect));
```

4.2.4.3 Weavings

Como se ha comentado anteriormente, los *weavings* son mecanismos que permiten que la ejecución de un servicio de un aspecto concreto se asocie antes, después o en lugar de un servicio ofrecido por otro aspecto. Una diferencia importante respecto a otras tecnologías orientadas a aspectos es que esta funcionalidad es especificada en los componentes y no en los aspectos, de forma que los aspectos desconocen a qué otro aspecto pueden ser enlazados, con el objetivo de facilitar la reutilización de aspectos.

♦ Estrategias de implementación

No existen trabajos previos de implementación en .NET para dar soporte a los weavings a nivel de código fuente. Las aproximaciones estudiadas en el Capítulo 2 que trabajaban a este nivel únicamente realizaban un *weaving* estático, como se describe a continuación. En cambio, ninguna ha proporcionado un diseño capaz de soportar el *weaving* dinámico.

Por esta razón, para la implementación de los weavings en .NET se han diseñado cuatro estrategias distintas:

- *Inserción estática de código*

Los weavings pueden traducirse directamente en llamadas a los servicios de los aspectos requeridos, pero eso supone que cuando se desee insertar o eliminar un weaving no pueda modificarse fácilmente el código insertado, ya que está mezclado con el código dedicado a otras tareas. La ventaja de esta estrategia es el incremento del rendimiento del sistema en ejecución, frente a otras alternativas, ya que son simplemente llamadas a métodos.

No obstante, esta estrategia podría llevarse a cabo asociando a cada método un atributo que ofreciese las funciones *after*, *before* e *instead* y que contuviesen el código necesario para llamar a los métodos de los aspectos correspondientes. Con esto se conseguiría que los weavings de dicho método estuviesen aislados en el código del atributo. Sin embargo, a pesar de esta ventaja, su modificación seguiría haciéndose emitiendo código para el cuerpo de dichas funciones.

- *Intercepción de código mediante Proxies*

.NET permite mediante el uso de *ContextBoundObjects* y *ContextAttributes* utilizar mecanismos para separar la ejecución de una clase determinada de las demás clases, mediante la creación de un nuevo contexto de ejecución. Este contexto permite el filtrado de las peticiones de servicio dirigidas a dichas clases aisladas, es decir, el preprocesado y postprocesado de código previo a la ejecución de los servicios ofrecidos por dichas clases. De esta manera, cada aspecto debe heredar de *ContextBoundObject* y etiquetarse con un *ContextAttribute*, y el código del componente sería el encargado de rellenar adecuadamente dichas estructuras con los weavings a realizar. Esto se implementaría situando una llamada al aspecto destino en el preproceso (para el caso de weavings *before*) o en el postproceso (para el caso de weavings *after*), consiguiendo que cada servicio ofrecido por el aspecto sea interceptado y procesado adecuadamente. No obstante, esta aproximación supone interceptar todos los aspectos de un componente y si hay varios componentes, esto supone que cada clase se ejecute en un contexto diferente, con la sobrecarga que esto conlleva para el sistema.

- *Uso de eventos definidos en el código del aspecto*

Otra aproximación consiste en definir tres eventos asociados a cada servicio del aspecto: *before*, *after* o *instead*. En el código del servicio se comprueba si se ha suscrito algún delegado a dichos eventos, y en caso afirmativo, se ejecuta el correspondiente delegado. La tarea de añadir/eliminar weavings es realizada por el componente, suscribiendo eventos a dichos métodos mediante el uso de delegados que apuntan al servicio a ejecutar del siguiente aspecto. El principal inconveniente es que se ha de modificar el código de los aspectos para publicar los eventos, pero permite que ante cualquier petición de servicio externa se le pase el control al componente para realizar el procesado necesario.

- *Uso de una lista de delegados*

Esta aproximación supone que todas las peticiones de servicio se realicen a través del componente, y por tanto, el procesado de los weavings se hace previamente a la llamada al aspecto correspondiente. La idea general es utilizar una serie de listas dinámicas anidadas para finalmente almacenar los delegados asociados a cada tipo de weaving. Esta aproximación proporciona la suficiente flexibilidad para añadir/eliminar dinámicamente los weavings (sin emitir código), pero presenta la limitación de sólo aplicarse a nivel de componente; es decir, si en un aspecto se dispara un *trigger*, al no pasar por el componente, no se podría interceptar su llamada para aplicarle un weaving (a no ser que se modificase el código del aspecto, provocando que la ejecución de un *trigger* necesariamente deba redirigirse al componente). Por otra parte, proporciona una estructura para el almacenamiento conjunto de todos los weavings sin entremezclarse con el código del componente.

De entre las estrategias vistas, la inserción estática de código es la que presenta mejor rendimiento, y la intercepción de código la más flexible. Con el objetivo de encontrar un equilibrio entre flexibilidad y rendimiento, se ha implementado la aproximación que utiliza listas de delegados para la gestión de weavings (*WeavingsCollection*). El principal inconveniente de la implementación realizada es que la estructura para almacenar los diferentes delegados es compleja, formada una lista enlazada con tres niveles de profundidad, lo que provoca que para comprobar si un servicio tiene weaving se tengan que realizar hasta tres búsquedas por las diferentes listas. Esto se puede optimizar implementando algoritmos de búsqueda binaria o de búsqueda rápida.

♦ La estructura de almacenamiento de los weavings

En la definición de un weaving existen tres elementos clave por los cuales se puede clasificar la información de forma jerárquica, y con ello optimizar las búsquedas. En primer lugar, un weaving se asocia a un **tipo de aspecto**, y en concreto a un servicio de dicho aspecto. A este servicio se le denominará **método original** pues su invocación desencadenará la invocación de otros servicios asociados a través del weaving. El método original puede tener asociados hasta tres **tipos de weavings** distintos: *after*, *before* e *instead*. Se incluyen los tipos de weavings condicionales *afterIf* y *beforeIf* como un subtipo del weaving *after* y *before*, respectivamente. Al servicio a ejecutar como consecuencia de la aplicación de un weaving se le denominará **método destino**, pues puede reemplazar al método original (*instead*), modificar los parámetros de entrada (*before*) o modificar los parámetros de salida (*after*). Tanto el método original como el método destino se almacenarán en forma de delegados. Por tanto, y de forma general, navegar por la lista dinámica supone atravesar tres niveles hasta llegar finalmente al delegado que debe ejecutarse (método destino): el tipo de aspecto, el método original y el tipo de weaving. A continuación se describirá cómo se ha implementado esta estructura.

WeavingsCollection (ver Figura 32) es la clase que, además de proporcionar el comportamiento del gestor de weavings, contiene la lista dinámica descrita (*weavingList*) junto a las funciones necesarias para su manejo. *weavingList* almacena los distintos tipos de aspectos que tienen weaving asociado y representa al primero de los niveles descrito anteriormente. Cada nodo de esta lista es una instancia de la clase *AspectTypeNode*, que contiene el tipo de aspecto que representa (*aspectType*) y otra lista dinámica (*weavingAspectList*) con los distintos servicios del aspecto: esta lista representa el segundo nivel. Los elementos de esta segunda lista son instancias de *WeavingNode* y como información identificativa guardan el nombre del servicio que representan (el método original), un delegado a dicho servicio para posteriormente poderlo invocar dinámicamente, y tres listas dinámicas adicionales correspondientes a los weavings *after*, *before* e *instead*. Estas listas representan el tercer nivel de profundidad. En ellas se almacenan instancias de *WeavingMethod*, cuyos atributos y métodos son los siguientes:

- **methodDelegate**: Atributo que contiene el delegado que apunta hacia el método destino, es decir, el servicio cuya invocación será provocada por el weaving.
- **origMethod**: Atributo que contiene el delegado del método origen, es decir, el servicio que ha provocado la activación de los weavings.
- **weavingType**: Atributo que define el tipo de weaving que representa el nodo, y para el caso de los weavings condicionales, define la condición que debe cumplirse para aplicarse el weaving. Se describirá más adelante.

- **functions:** Atributo que contiene una instancia de *DelegateFunctionsCollection*, y define las funciones de transformación que deberán aplicarse a cada parámetro antes de ejecutar el método destino.
- **inputMethodParameters** y **outputMethodParameters:** Atributos que definen los tipos de parámetros que requiere el método destino para ejecutarse. Se describirá más adelante cuando se describan las funciones de transformación, pues son las que justifican la existencia de estos atributos.

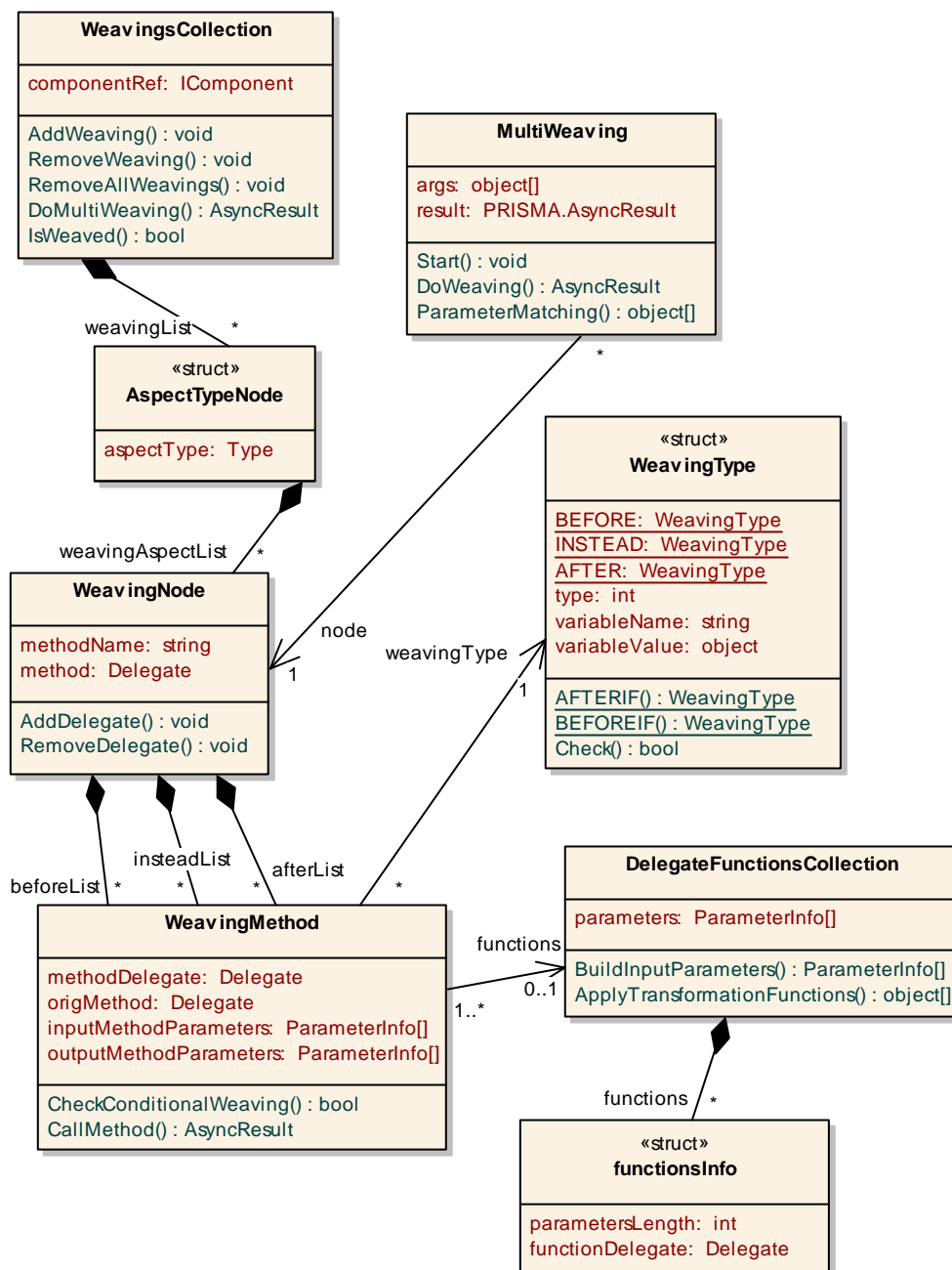


Figura 32 - Clases asociadas al Weaving

- **CheckConditionalWeaving(args: object[]):** Servicio que comprueba si se cumple la condición especificada por los weavings condicionales. Como parámetros requiere la lista de argumentos del método origen.
- **CallMethod(args: object[]):** Servicio que ejecuta el método destino, aplicándole previamente las funciones de transformación definidas. Como parámetros requiere la lista de argumentos del método origen.

La clase *WeavingType* permite representar el tipo de weaving especificado. Se compone de una serie de atributos de clase que contienen instancias de sí misma pero inicializadas adecuadamente para representar diferentes tipos de weaving: *after*, *afterIf*, *before*, *beforeIf* o *instead*. Internamente, un atributo numérico (*type*) permite diferenciarlos entre sí. Los weavings condicionales requieren como parámetros adicionales el nombre de la variable cuyo valor se ha de comprobar (*variableName*) y el valor que ha de tener dicha variable para que la condición se evalúe a cierta (*variableValue*). El método *Check(object variable)* permite comprobar que el valor de la variable pasada como parámetro es el que evalúa a cierta la condición. Sin embargo, ha quedado como tarea futura el implementar los demás operadores de comparación (<, >, ≤, ≥, ≠) en los weavings condicionales para aumentar la expresividad del modelo. Para ello, basta con añadir un parámetro adicional en la especificación de los weavings condicionales que defina el tipo de comparación a realizar.

◆ Creación y Eliminación de weavings

Una vez vistos los distintos elementos que forman un weaving, faltaría describir cómo los métodos *AddWeaving* y *RemoveWeaving* permiten definir, de forma intuitiva, la creación y eliminación de weavings. El método *AddWeaving* permite añadir nuevos weavings a la lista dinámica, creando un nuevo nodo *WeavingMethod* e insertándolo en la posición correspondiente dentro de la jerarquía. La signatura del método es la siguiente:

```
public void AddWeaving(IAAspect origAspectInstance, string origMethod,
    WeavingType weavingType, IAAspect destAspectInstance,
    string destMethod, DelegateFunctionsCollection functions)
```

El parámetro *origAspectInstance* es una referencia a un aspecto ya creado que contiene el método *origMethod*, al cual se le va aplicar un weaving de tipo *weavingType*. El servicio que se ejecutará (*destMethod*), según determine el weaving, pertenece al aspecto *destAspectInstance*. El parámetro *functions* permite especificar, de forma opcional, las funciones que se aplicarán para transformar los parámetros del método original a los requeridos por el método destino.

Sin embargo, como se puede observar, para definir un nuevo weaving no se proporciona el tipo del aspecto, sino un aspecto ya agregado al componente (y por tanto una instancia de la clase del aspecto). Esto es necesario debido a

que los delegados únicamente se pueden crear para instancias de clases, y por tanto para crear dinámicamente el delegado correspondiente al método *origMethod* se requiere la instancia de la clase del aspecto al cual pertenece el método. Esto presenta como limitación que los weavings no puedan existir si el aspecto correspondiente es eliminado. De esta forma, si un aspecto se elimina del componente y es sustituido por otro del mismo tipo con los mismos servicios, los weavings deberían volverse a crear. Esto queda como una tarea futura a resolver.

El método *RemoveWeaving* permite eliminar el weaving asociado a un método concreto. Los parámetros requeridos son el aspecto (*aspect*) al cual pertenece el método *origMethod* cuyo weaving se va a eliminar. El parámetro *weavingType* permite indicar qué tipo de weaving se va a eliminar y *destMethod* el método asociado a dicho weaving.

```
public void RemoveWeaving(IAAspect aspect, string origMethod,
    WeavingType weavingType, string destMethod)
```

A continuación se muestra como un weaving PRISMA se traduce a un weaving de la implementación. Por ejemplo, el siguiente weaving en PRISMA especifica que al retirar dinero de la cuenta (*Withdrawal*) se actualice adecuadamente el formulario asociado (aspecto de presentación, *ClientInteraction*) con el nuevo saldo restante:

```
Functional Aspect import BankInteraction;
Weaving
    BankInteraction.Withdrawal(quantity: decimal, money: decimal);
    before ClientInteraction.Show_money(money: decimal);
End_Weaving;
```

En la implementación, dicha especificación se correspondería con:

```
AddWeaving( functionalAspect, "Withdrawal", WeavingType.BEFORE,
    presentationAspect, "Show_money");
```

De forma similar, la especificación de un weaving condicional para especificar que si se alcanza el saldo cero en la cuenta el usuario sea alertado con un mensaje de advertencia correspondería con:

```
AddWeaving( functionalAspect, "Withdrawal",
    WeavingType.BEFOREIF("money", (decimal) 0),
    presentationAspect, "DisableWithdrawal");
```

Por otra parte, el método *IsWeaved* permite comprobar si el método de un aspecto tiene weaving asociado. Este método es invocado por el hilo del componente para comprobar si el servicio que va a ser redireccionado al aspecto tiene weaving asociado. La signatura del método es la siguiente:

```
public bool IsWeaved(IAAspect origAspect, string origMethod)
```

◆ El tratamiento de los parámetros

Por otra parte, también ha sido necesario definir una política adecuada para el paso de argumentos. Cuando se invoca el servicio de un aspecto (método original), se le proporcionan los argumentos necesarios para su ejecución. Si éste tiene weaving asociado, sus argumentos también serán proporcionados a los servicios cuya invocación desencadene el weaving (métodos destino).

El problema reside en que los métodos destino pueden tener distinto número y tipo de argumentos que el método original invocado. Esto se ha solucionado pasando a cada método destino los parámetros del método original que concuerdan en nombre y tipo.

Sin embargo, esta estrategia introduce una limitación para la definición de weavings. Si se desea definir un weaving de forma que uno o más parámetros del método origen deban ser proporcionados a otro método destino, éste deberá definirse de forma que el nombre del parámetro que vaya a ser proporcionado por el método origen coincida en nombre y tipo. Por ejemplo, nótese cómo en la definición del siguiente weaving los parámetros “*newAdd*” de los servicios *ChangeAddress* y *Move* concuerdan en nombre y tipo, debido a que al servicio *Move* se le proporcionará el valor que se le proporcione al servicio *ChangeAddress*.

```
Distribution Aspect Import ExtMbile;
    Weaving
        BankInteraction.ChangeAddress(newAdd: string)
            before ExtMbile.Move(newAdd: string);
    End_Weaving;
```

Otra estrategia posible podría no considerar el nombre del parámetro y realizar la comparación únicamente por el tipo y la posición. Sin embargo, no ha sido implementada ya que puede producir efectos indeseados si el método origen tiene varios argumentos del mismo tipo y su posición no coincide con la que define el método destino.

Por otra parte, independientemente de la estrategia empleada, si el método destino tiene más argumentos que el método original surge el problema de qué valores proporcionar a estos argumentos adicionales. Para resolverlo se han creado las funciones de transformación de parámetros.

Las funciones de transformación de parámetros permiten asociar a cada parámetro de un método destino una función. Dicha función toma como entrada uno o varios parámetros definidos para el método origen y devuelve un objeto del mismo tipo que el parámetro del método destino al cual está asociada. Por ejemplo, siguiendo con el ejemplo anterior, si se quisiera realizar un weaving entre el método *ChangeAddress(newAddress: string)* del aspecto funcional y el método *Move(newLoc: LOC)* del aspecto distribución, podría definirse una función con la siguiente signatura:

```
LOC ChangeAddressToLOC(newAdd: string);
```

El funcionamiento de dicha función sería el de transformar una cadena de texto al tipo LOC. Por tanto, la definición del weaving sería la siguiente:

```
Distribution Aspect Import ExtMbile;
  Weaving
    BankInteraction.ChangeAddress(newAdd: string)
      before ExtMbile.Move(ChangeAddressToLOC(newAdd: string));
  End_Weaving;
```

Obsérvese cómo los parámetros del método origen (*ChangeAddress*) y de la función definida concuerdan en nombre y tipo, y que el valor de retorno de ésta coincide con el tipo del parámetro del método destino (*Move*).

Las funciones de transformación de parámetros se almacenan en una instancia de la clase *DelegateFunctionsCollection*, cuyos atributos y servicios son los siguientes (ver Figura 32):

- **parameters:** Atributo que almacena la información relativa a los parámetros del método destino, es decir, el método cuyos parámetros van a ser proporcionados por los resultados de las funciones definidas.
- **functions:** Vector de instancias *functionsInfo* del mismo tamaño que el número de parámetros del método destino. Define la función que se aplicará sobre cada parámetro.
- **DelegateFunctionsCollection:** Servicio constructor de la clase. Se le proporcionan como parámetros el aspecto que contiene el método destino y el nombre del método destino. Inicializa adecuadamente los atributos *parameters* y *functions*.
- **this[“nombre_parámetro”]:** Indexador que permite asociar una función de transformación al parámetro cuyo nombre se especifica.
- **BuildInputMethodParameters():** Servicio que construye la lista de parámetros resultante de la agregación de los tipos requeridos por cada una de las funciones de transformación definidas. Es utilizado exclusivamente para inicializar la propiedad *inputMethodParameters* de la clase *WeavingMethod*.
- **ApplyTransformationFunctions(args: object[]):** Servicio que aplica las funciones de transformación a la lista de parámetros pasada como argumento, devolviendo la lista de parámetros que requiere el método destino.

La estructura *functionsInfo* representa a una función de transformación y únicamente contiene un delegado que apunta a la función a ejecutar (atributo *functionDelegate*) y el número de parámetros que requiere dicha función (atributo *parametersLength*). Siguiendo con el ejemplo anterior, el siguiente fragmento de código muestra cómo asociar la función de transformación *ChangeAddressToLoc* al parámetro *newLoc* del método *Move* del aspecto de distribución:

```
// Creamos la estructura para alojar func. de transform. asociadas a "Move"
DelegateFunctionsCollection functions =
```

```
new DelegateFunctionsCollection(distributionAspect, "Move");

// Asociamos las funciones de transformación a cada parámetro.
// Para ello, debemos proporcionar un delegado hacia la función a aplicar
functions["newLoc"] = new ExtMbile.ChangeAddressToLOCDelegate(
    ((ExtMbile) distributionAspect).ChangeAddressToLOC);

// Creamos el Weaving, especificando que se aplican func. de transform.
AddWeaving(functionalAspect, "ChangeAddress", WeavingType.BEFORE,
    distributionAspect, "Move", functions);
```

Una vez visto cómo se almacenan las funciones de transformación en la estructura de weavings, falta por describir cómo se aplican a los parámetros. Para ello, en primer lugar es necesario introducir el tipo de datos *ParameterInfo*. Éste se encuentra definido en el espacio de nombres de .NET System.Reflection, y proporciona información sobre los atributos de un parámetro, como el nombre, el tipo, si es de entrada o de salida, etc. La información de los parámetros de un método es almacenada en vectores *ParameterInfo* y se obtiene a través de la reflexión. Esta información es utilizada para poder realizar el filtrado de parámetros, como se describe a continuación.

Cuando el método destino va a ser invocado por el weaving, hay que proporcionarle la lista de parámetros necesaria para su ejecución. Ésta viene determinada por el método origen, que puede tener distinto número y tipo de argumentos, por lo que antes de invocar al delegado del método destino será necesario filtrar dicha lista de parámetros con los requeridos. Esta tarea es realizada por el método privado *ParameterMatching*, definido en la clase *MultiWeaving*, cuya signatura es la siguiente:

```
private object[] ParameterMatching(
    ParameterInfo[] origParameters, object[] origArgs,
    ParameterInfo[] targetParameters, object[] targetArgs);
```

Como parámetros de entrada toma la definición de los parámetros del método origen y del método destino (de tipo *ParameterInfo*), así como un vector con los argumentos del método origen (*origArgs*), que serán filtrados y copiados en el vector de argumentos del método destino (*targetArgs*), en el orden correcto que espera el método. Básicamente, esta función va comparando los parámetros del método origen en nombre y tipo con los del método destino, seleccionando únicamente los que necesita el método destino. Si algún parámetro requerido por el método destino no es encontrado, se deja el hueco correspondiente en el vector *targetArgs*, el cual será interpretado como el valor por defecto del tipo esperado (por ejemplo, si se esperaba un entero, el hueco se interpretará como el valor '0').

En cambio, si se definen funciones de transformación para el método destino, el filtrado deberá hacerse en función de los parámetros que especifiquen estas funciones, de tal forma que la lista de argumentos construida contenga los especificados por las funciones de transformación. En el atributo *inputMethodParameters* de la clase *WeavingMethod* se almacena esta información. En cambio, el atributo *outputMethodParameters* contiene el formato que tendrá la lista de argumentos del método destino,

sin tener en cuenta las funciones de transformación, para poder posteriormente recuperar los resultados obtenidos. El método *callMethod*, definido en *WeavingMethod*, tomando como entrada una lista de argumentos cuyo formato coincida con el especificado en *inputMethodParameters*, es el encargado de aplicar las distintas funciones de transformación a los argumentos proporcionados y de invocar el método destino con la lista de argumentos adecuadamente transformada.

Por ejemplo, si el método destino necesita como argumentos la lista: $\{string, decimal\}$, y se define una función de transformación tal que $f(x: int) \rightarrow decimal$, el atributo *inputMethodParameters* contendrá la lista: $\{string, int\}$, mientras que el atributo *outputMethodParameters* = $\{string, decimal\}$. Suponiendo que en ejecución la lista de parámetros del método origen es {"abba", 49.5, -5, "a"}, el filtrado de dicha lista gracias a la información contenida en *inputMethodParameters* generará la lista {"abba", -5}, y tras aplicar la función de transformación, podrá invocarse el método destino con los parámetros correctos. Suponiendo que los dos parámetros del método fuesen de retorno y que como resultado ha devuelto {"bb", 2.2}, gracias a la información contenida en *outputMethodParameters* puede filtrarse esta información y depositarse de nuevo sobre la lista de parámetros del método origen en la posición correcta: {"bb", 49.5, 2.2, "a"}.

♦ La aplicación de los weavings

La aplicación de un weaving es realizada por el hilo del componente tras comprobar previamente que el servicio a ejecutar por el aspecto tiene un weaving asociado, mediante la invocación al método *DoMultiWeaving*. Este método se encarga de buscar en la lista dinámica de métodos a los que se le aplica weaving, y en caso de encontrarlo, va recorriendo las listas *beforeList*, *afterList* e *insteadList* y ejecutando los delegados correspondientes en el orden adecuado. La signatura de dicho método es la siguiente:

```
public AsyncResult DoMultiWeaving(IAAspect aspect,
                                   string origMethodName, object[] args)
```

La aplicación de un weaving supone una ejecución secuencial de los distintos servicios implicados, es decir, del método origen y los distintos métodos destino, cuya invocación es desencadenada por la aplicación del weaving asociado al método origen. Esta ejecución es secuencial debido a que la propia definición de weaving establece un orden de ejecución de los distintos servicios, de lo que se deriva que hasta que un servicio no se haya procesado completamente no podrá procesarse el siguiente.

Para la ejecución de los weavings se consideraron dos posibles estrategias:

- **A través del hilo de ejecución del componente**

Esta estrategia no permite reenviar las nuevas peticiones a los aspectos hasta que el weaving finalice. En un primer prototipo se aplicó esta estrategia, cuya ventaja principal es una mayor sencillez y la conservación de un estado coherente en los aspectos, pues hasta que no ha finalizado un weaving por completo el aspecto no recibe las nuevas peticiones.

Sin embargo, el inconveniente viene cuando un servicio a ejecutar por el weaving no puede procesarse debido al estado actual del protocolo. En muchos casos la actuación esperada es dejar dicho servicio encolado en el aspecto (en espera, mediante el mecanismo descrito en el apartado de los aspectos) hasta la llegada de otro servicio que cambie el estado del protocolo a uno válido para la ejecución del servicio. No obstante, como el weaving requiere una ejecución síncrona, si el servicio está en espera el hilo del componente se encontrará en estado suspendido a la espera de la finalización del servicio, por lo que no podrá reenviar nuevas peticiones de servicio y por tanto nunca cambiará el estado del aspecto. Se produce por tanto un interbloqueo.

- **A través de un nuevo hilo de ejecución exclusivo**

Esta estrategia libera al hilo del componente de la ejecución de los weavings, permitiendo que reenvíe nuevas peticiones a los aspectos y por tanto, solucionar el problema descrito anteriormente. También incrementa el rendimiento del componente, pues mientras se está procesando un weaving, pueden reenviarse nuevas peticiones de servicio a otros aspectos.

Sin embargo, esta estrategia presenta el inconveniente de aumentar la probabilidad de perder el orden de ejecución de los servicios enviados a un aspecto, a causa de la concurrencia. En el caso de estar en un weaving en ejecución, y recibirse una petición de servicio cuyo aspecto estuviese implicado en el weaving, podría ocurrir que la nueva petición de servicio llegase antes al aspecto que por parte del weaving, por lo que el orden de ejecución se vería alterado.

De las dos estrategias descritas, se ha implementado la segunda, aunque el problema presentado no ha sido resuelto en el prototipo actual. Una posible solución al problema sería la creación de unos servicios de espera específicos en los aspectos, de forma que cuando el weaving se iniciase, detectase todos los aspectos relacionados e invocase dicho servicio. Este servicio, cuando fuese procesado por el aspecto, quedaría a la espera hasta que el weaving actualizase su estado, proporcionando el servicio concreto a ejecutar. De esta

forma, las peticiones posteriores al weaving no podrían ser procesadas hasta que el weaving finalizase.

El método *DoMultiWeaving* cuando es invocado crea una instancia de la clase *MultiWeaving*, proporcionándole los argumentos del método origen, el objeto *AsyncResult* donde depositar los resultados y el *WeavingNode* en el que se encuentra toda la información del weaving a aplicar. Esta instancia es la que crea un hilo de ejecución adicional para gestionar cada weaving. Toda la funcionalidad para el procesado de un weaving se encuentra definida en el método *DoWeaving()*, cuya estructura se procederá a describir.

La invocación de un servicio que tiene weaving asociado (método origen) desencadenará la ejecución de distintos servicios (métodos destino) en un orden definido por el weaving. Tras lanzarse a ejecución el método *DoWeaving*, en primer lugar se van ejecutando los distintos métodos destino almacenados en la lista *WeavingNode.afterList* (el método origen se ejecutará después de los definidos en esta lista). En segundo lugar se comprueba si existen métodos destino en la lista *WeavingNode.insteadList*, en cuyo caso se ejecutan secuencialmente, sustituyendo la ejecución del método origen. En caso negativo, se ejecuta el método origen. En tercer lugar, se ejecutan los distintos métodos almacenados en la lista *WeavingNode.beforeList*.

A continuación se muestra un extracto del código del método *DoWeaving*, el correspondiente a la ejecución de los weaving *after* y *afterIf*:

```
AsyncResult result = null;
object[] args = this.args;
WeavingNode nodo = this.node;

// Si hay un weaving AFTERIF y su condición no se evalúa a cierto, no se
// ejecutará el método origen
bool conditionalExecution = true;

// Obtenemos los parámetros del nodo origen
ParameterInfo[] origMethodParameters = nodo.Method.Method.GetParameters();

// Realizamos el weaving AFTER
if (nodo.AfterList != null && nodo.AfterList.Count > 0) {
    foreach (WeavingMethod nodoAfter in nodo.AfterList) {
        try {
            result = (AsyncResult)
                nodoAfter.CallMethod(ParameterMatching(origMethodParameters,
                    args, nodoAfter.InputMethodParameters,null));
            // Como el resultado del delegado anterior puede influir en el
            // siguiente, debemos esperar a que éste acabe (suspendido), tras lo
            // cual seremos despertados por el aspecto
            if (!result.HasResult())
                System.Threading.Thread.CurrentThread.Suspend();
        } catch (TargetInvocationException e) {
            Console.WriteLine(e.StackTrace); throw e.InnerException; }
    }

    // Obtenemos los resultados de la llamada síncrona.
    if (result.HasResult()) {
        object[] oldArgs = result.GetParameters();
        args = ParameterMatching(nodoAfter.OutPutMethodParameters, oldArgs,
            origMethodParameters, args);
    }
}
```

```
// Comprobamos si se ha de ejecutar el método origen
if (conditionalExecution)
    conditionalExecution = nodoAfter.CheckConditionalWeaving(oldArgs);
}
else {
    throw new Exception("Critical Error in Threading section");
}
}
}

// Si no hay miembros en la lista de instead, ejecutamos el METODO ORIGEN
if (nodo.InsteadList == null || nodo.InsteadList.Count == 0) {
    // Comprobamos si había una clausula AFTERIF con una condicion
    if (conditionalExecution) {

// ....
```

Como se puede observar, a través de un bucle se van procesando todos los weaving *after* (y *afterIf*, pues se almacenan en la misma lista) de forma secuencial. En primer lugar, mediante la invocación al método *ParameterMatching* se filtra la lista de argumentos del método origen generando la lista que espera recibir el método destino correspondiente. En segundo lugar, el método destino es invocado a través del método *CallMethod*, definido en la clase *WeavingMethod*, el cual aplica las funciones de transformación que hubiesen definidas. Tras esto, el hilo del weaving queda suspendido a la espera que la invocación del servicio finalice. Una vez despertado el hilo del weaving, obtiene los resultados de la estructura *AsyncResult*, y comprueba si se trataba de un weaving condicional (*afterIf*), invocando para ello el método *CheckConditionalWeaving()* también definido en la clase *WeavingMethod*. Finalmente, se filtran de nuevo los argumentos para que se ajusten a la signatura del método origen.

De esta forma, la creación de weavings es totalmente dinámica, sin más que especificar los aspectos a entretejer, los nombres de los métodos correspondientes y el tipo de weaving a aplicar. El método *AddWeaving* es el encargado, a través de la reflexión, de obtener la información de ambos métodos, crear dinámicamente los delegados y almacenar los datos adecuadamente en la estructura *WeavingsCollection*. Sin embargo, la creación de delegados en tiempo de ejecución no ha sido una tarea sencilla. La forma común de crear delegados es definiendo el tipo en tiempo de compilación e instanciándolo en tiempo de ejecución. Pero si los aspectos son agregados en tiempo de ejecución, el componente no conoce el tipo de los delegados necesarios para acceder a sus métodos, por lo que no puede instanciarlos.

Para solucionar este problema, pueden seguirse dos estrategias. La primera de ellas es la que se utilizó en un primer prototipo, y se basa en la creación de delegados de forma dinámica emitiendo código. Aunque en .NET existe una clase de la cual hereda todo delegado (las clases *Delegate* y *MulticastDelegate*), no existe ningún constructor para crear dinámicamente delegados, por lo que hay que recurrir a emitir código. El proceso a seguir es

generar el código intermedio correspondiente al tipo de delegado requerido, creándose un ensamblado en memoria, y posteriormente instanciar dicho delegado para que apunte al método requerido. El principal inconveniente, aparte de que debe emitirse código para cada delegado, es que si el componente es movido a otra máquina debe volverse a generar el delegado.

Otra alternativa posible al uso de los delegados sería hacer uso de la reflexión y el método *Invoke()* para invocar a los métodos, pero el rendimiento obtenido es inferior, como se demuestra en [Gun04], donde se puede encontrar un estudio sobre las distintas formas de invocar código dinámicamente en .NET. No obstante, en la versión de .NET *Framework* 2.0 han mejorado el rendimiento obtenido mediante el uso de la reflexión y han añadido mecanismos para crear dinámicamente delegados.

La segunda estrategia presupone que los tipos de delegados han sido definidos junto a las interfaces que definen los aspectos, por lo que para instanciar un delegado no hay más que buscar el tipo requerido en el ensamblado correspondiente. Esta estrategia es la que finalmente se ha implementado, puesto que los puertos también hacen uso de los delegados para encolar las peticiones de servicio. En la clase *Delegates* se encuentra la implementación de las dos estrategias.

4.2.4.4 Puertos

Los puertos son entidades agregadas al componente que, además de proporcionar los mecanismos necesarios para comunicarse con otros elementos arquitectónicos, facilitan la evolución de los componentes. El componente por sí solo (como instancia de la clase *ComponentBase*) únicamente proporciona el comportamiento para gestionar weavings y aspectos, y para redirigir servicios a los aspectos. Éstos, al ser agregados al componente, proporcionan la implementación de los servicios definidos en una serie de interfaces, y a través de los puertos, cada una de las interfaces que implementan los aspectos es publicada al exterior. Cuando un aspecto es eliminado de un componente, también son eliminados los puertos asociados, con lo que el componente deja de publicar los servicios asociados al aspecto. Esto se ha realizado de esta forma con dos objetivos: por una parte, que el componente sea totalmente independiente de las partes que lo componen, favoreciendo con ello la reutilización, y por otra parte, que la agregación/eliminación de aspectos, weavings o puertos pueda realizarse en tiempo de ejecución sin necesidad de recompilar el componente. Solamente deberá pararse el componente en aquellos casos en los que la modificación de alguna de sus partes afecte a un servicio en ejecución.

Los puertos también ofrecen mecanismos de reflexión, ya que pueden ser interrogados dinámicamente para obtener la lista de interfaces que publica el componente al cual están agregados, posibilitando comunicaciones totalmente dinámicas. Además, los puertos permiten aislar los servicios que

proporcionan los aspectos (los especificados por el desarrollador) de los métodos que publican los componentes para su manejo por parte del *middleware*. Por ejemplo, podría darse el caso de que en los aspectos también se definiesen servicios cuyo interfaz fuese *Start()* y *Stop()*, que entrarían en conflicto con los heredados de *ComponentBase*. También proporcionan una solución para el problema de idénticos servicios en distintas interfaces, al definirse un puerto diferente para cada interfaz.

Todo puerto hereda de una clase padre *Port* (ver Figura 33), que define los atributos básicos que tienen los puertos:

- **portName**: nombre identificativo del puerto, tal como ha sido definido en la especificación PRISMA de un componente.
- **interfaceName**: nombre de la interfaz que implementa el puerto.
- **subProcess**: nombre del subproceso vinculado al puerto.
- **attachmentRegistered**: es un vector que contiene información sobre los attachments que están comunicándose con el puerto, con el objetivo de que si el componente es movido, o detenido, el componente pueda notificar al *middleware* a qué attachments tiene que avisar.

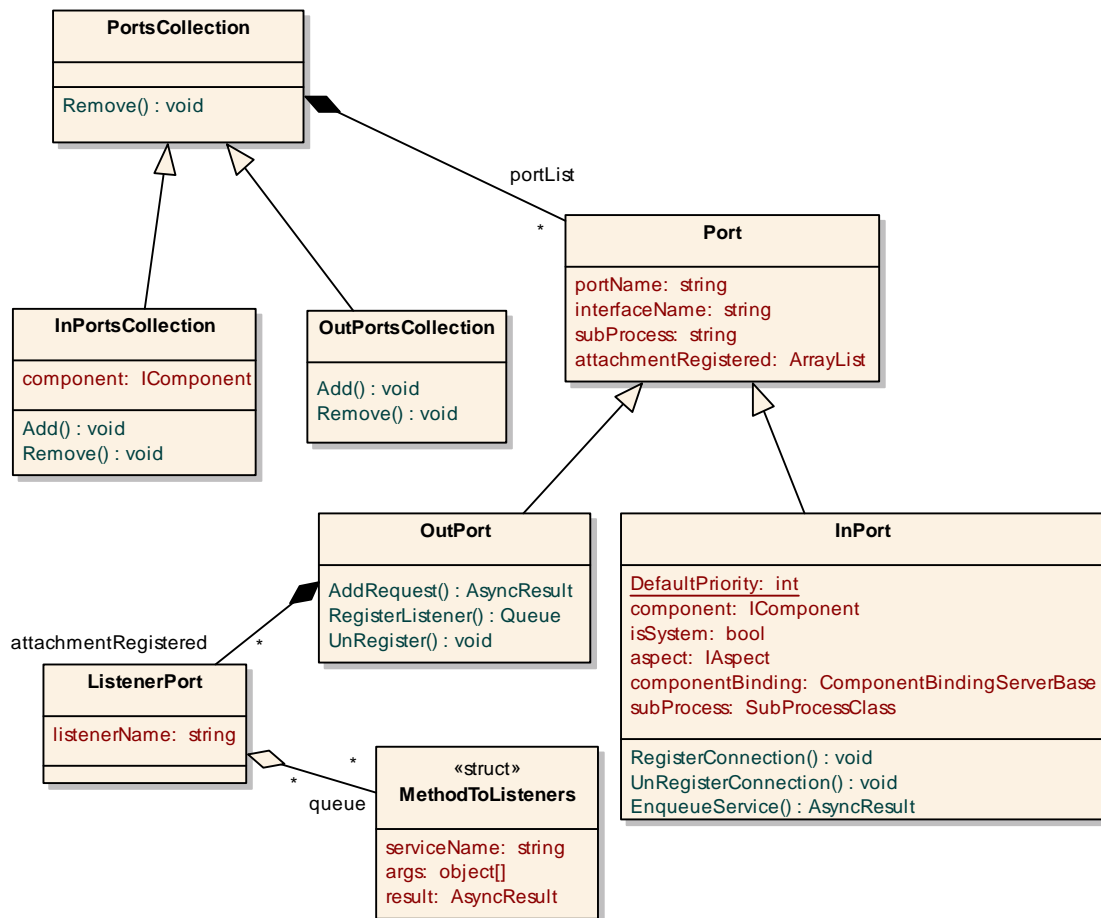


Figura 33 - Clases InPort, OutPort y ListenerPort

En implementación, los puertos se clasifican, por su función, en puertos de entrada y puertos de salida. Los de entrada publican los servicios que ofrece el componente y reciben las peticiones de otros componentes. Mientras que los de salida se encargan de entregar las peticiones generadas por el componente a los componentes externos a los cuales esté conectado.

Los puertos se agregan al componente a través de instancias de las clases *InPortsCollection* y *OutPortsCollection*, para los puertos de entrada y los puertos de salida, respectivamente. Estas clases son las encargadas de gestionar los puertos agregados al componente. Ambas heredan de la clase *PortsCollection*, que define como atributo básico, una lista dinámica (*portList*) y la signatura de los métodos para eliminar los puertos de dicha lista. La implementación de los métodos para añadir o eliminar puertos a dicha colección es realizada por sus descendientes, pues los puertos de entrada requieren parámetros de inicialización distintos a los de salida. Los métodos que permiten eliminar puertos tienen la siguiente signatura:

```
void Remove(string interfaceName, string subProcesses);
void Remove(string portName);
```

Las dos sobrecargas del método permiten eliminar un puerto, tanto de entrada como de salida, a partir de su nombre identificativo o por la interfaz y el nombre del subprocesso (*played_role*) que lo identifica.

La clase *InPortsCollection* es la encargada de gestionar la lista de puertos de entrada. Proporciona dos *indexers* para poder acceder a los puertos por nombre o por el par [nombre_interfaz, nombre_subproceso]. Ofrece la implementación de los métodos *Add* y *Remove*. La signatura del método *Add* se muestra a continuación:

```
void Add(string portName, string interfaceName,
        IAspect implementerAspect, string SubProcessName)
```

El método *Add* crea un puerto de entrada a partir del nombre de la interfaz que debe implementar, dándole un nombre identificativo (*portName*), la referencia al aspecto que implementa y el nombre del subprocesso que representa (si es aplicable).

De forma similar, la clase *OutPortsCollection* se encarga de gestionar la lista de puertos de salida. Proporciona también dos *indexers* para obtener los puertos por el nombre o por la interfaz+subproceso. También proporciona la implementación para los métodos *Add* y *Remove*. A diferencia de los puertos de entrada, los de salida no requieren un aspecto asociado, por lo que la signatura del método *Add* es distinta:

```
void Add(string portName, string interfaceName, string subProcessName)
```

◆ Puertos de entrada

El comportamiento de los puertos de entrada se define en la clase *InPort*, que a su vez hereda de la clase *Port*. Un puerto de entrada, desde el momento de su creación, está asociado al aspecto el cual implementa los servicios que él publica. Estos servicios pueden tener una prioridad asignada, que vendrá determinada por los subprocesos (*played_roles*) del aspecto. Los atributos y servicios que define esta clase son:

- **DefaultPriority**: atributo que define la prioridad por defecto para aquellos servicios que no tengan prioridad asignada.
- **component**: atributo que guarda la referencia hacia el componente al cual está agregado el puerto. Servirá para poder enviar las peticiones recibidas.
- **isSystem**: flag que permite conocer si el puerto está asociado a un sistema, en cuyo caso el funcionamiento es ligeramente distinto, pues puede que el puerto también tenga asociado un binding al que redirigirle la petición.
- **aspect**: atributo que guarda la referencia hacia el aspecto al cual está asociado el puerto. Es necesario para poder instanciar los delegados.
- **componentBinding**: atributo que almacena el binding al cual redirigir la petición de servicio recibida, en el caso de ser un puerto perteneciente a un sistema.
- **subProcess**: atributo que almacena la información relacionada con los subprocesos (*played_roles*) del aspecto y permite obtener la prioridad de cada servicio.
- **RegisterConnection()** y **UnRegisterConnection()**: servicios que almacenan o borran, respectivamente, el nombre del attachment que se está comunicando con este puerto en el vector heredado *attachmentRegistered*.
- **EnqueueService()**: método protegido que sólo podrá ser invocado por las clases que hereden de *InPort*. Se encarga de almacenar en la cola del componente o del sistema, según sea el caso, la petición de servicio, que consiste en un delegado y un vector con sus argumentos. También encapsula en un sólo método las diferencias entre puertos de sistemas y componentes.

De forma similar a los aspectos, la implementación de los puertos de entrada se encuentra separada, por una parte en el comportamiento básico, y por otra parte en la definición de la interfaz que publica el puerto. Mientras que la clase *InPort* define cómo el puerto de entrada se comporta e interactúa con el componente, la especificación de los servicios de las interfaces que publica el puerto se implementan en otras clases descendientes de *InPort*. De esta forma, cuando se compila una interfaz PRISMA, también se genera el puerto de entrada correspondiente que

publicará dicha interfaz. Cuando el puerto de entrada es agregado al componente, se carga el tipo de puerto correspondiente a la interfaz requerida y se instancia con los valores iniciales necesarios. La convención utilizada para que el *middleware* sea capaz de encontrar los puertos requeridos para cada interfaz es nombrar a los tipos generados como "InPort"+nombre_interfaz.

Para los componentes externos, acceder a un servicio publicado por un componente, se traduce a una línea de código bastante intuitiva:

```
Nombre_componente.InPorts[Nombre_interfaz].NombreServicio(argumentos);
```

A continuación se muestra un ejemplo de la sentencia que se ha de incluir en el código para que una instancia del componente *Account* accediese al servicio *Withdrawal*, cuya interfaz es *ICreditCardTransactions*.

```
Account1.InPorts["ICreditCardTransactions"].Withdrawal(quantity,money);
```

La plantilla de código que debería generar el compilador para especificar un puerto de entrada que publicase servicios de la interfaz *ICreditCardTransactions* sería la siguiente:

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Components;
using PRISMA.Components.Ports;

namespace CuentaBancaria {
    // Esta clase implementa el puerto de entrada correspondiente a la interfaz
    // "ICreditCardTransactions". Los componentes que tengan dicha interfaz como
    // servicio, usarán este puerto.
    [Serializable]
    public class InPortICreditCardTransactions: InPort,ICreditCardTransactions {
        // Constructor del InPort. Proporciona al padre los parámetros necesarios
        public InPortICreditCardTransactions(string inPortName,
            string subProcessName, IAspect aspect, IComponent component) :
            base(inPortName,"ICreditCardTransactions", subProcessName,aspect,
                component) {}

        #region Servicios ofrecidos por ICreditCardTransactions
        // Aquí se define cada uno de los servicios que publica la interfaz

        public AsyncResult Withdrawal(decimal quantity, ref decimal money) {
            lock(this) {
                // Creamos el delegado hacia el servicio correspondiente
                if (aspect!=null)
                    withdrawaldelegate = new WithdrawalDelegate(
                        ((ICreditCardTransactions)this.aspect).Withdrawal);
                money = 0;

                // Encolamos la petición
                return (AsyncResult) this.EnqueueService("Withdrawal",
                    withdrawaldelegate, quantity, money);
            }
        }
        // ...
    }
    #endregion
}
```

Como se puede observar, el código a generar es mínimo, pues la mayor parte del comportamiento es heredado de la clase *InPort*. Al igual que las demás entidades implementadas, los puertos de entrada deben ser etiquetados con el atributo *Serializable* para permitir que puedan moverse con el componente. El constructor del puerto únicamente requiere los argumentos declarados por el constructor del padre, como el nombre del puerto, el aspecto que implementa la interfaz, el subproceso (*played_role*) que representa y una referencia al componente. Por cada servicio especificado en la interfaz debe definirse un servicio con la misma signatura que, tras bloquear el puerto para permitir un acceso concurrente seguro, cree el delegado hacia el servicio del aspecto que lo implementa, y lo encole en el componente a través del método heredado *InPort.EnqueueService*.

◆ Puertos de salida

Los puertos de salida se han diseñado para permitir la petición de servicios por parte de los aspectos de un componente a los demás elementos arquitectónicos, sin que para ello los aspectos deban conocer el elemento arquitectónico concreto con el que se están comunicando, y favorecer con ello la reutilización. Esto es posible gracias a que los componentes y conectores son conectados entre sí a través de los attachments, que son los encargados de recoger las peticiones depositadas por los aspectos en los puertos de salida y reenviarlas a los puertos de entrada del otro extremo.

Un puerto de salida se encuentra implementado en la clase *OutPort*, y viene identificado por el nombre del puerto o bien por la interfaz que implementa y el subproceso (*playedRole*) que representa, de la misma forma que un puerto de entrada. Cuando los aspectos desean solicitar un servicio de otro componente, depositan sus peticiones en el puerto de salida. Éstas se almacenan en una cola, con la finalidad de que sean procesadas en el mismo orden que fueron enviadas. Sin embargo, como puede que varios attachments estén pendientes de un mismo puerto de salida, aparece el problema de quién o cuándo debe borrarse de la cola la petición de servicio. Si es borrada demasiado pronto, puede que algún attachment no hubiese recogido la petición. Si es descolada demasiado tarde, puede que sea recogida por un attachment que ya la había recogido previamente. El problema se ha solucionado creando un mecanismo de registro en los puertos de salida que ofrece a cada suscriptor (los attachments) un canal exclusivo en el que se depositarán las peticiones a él dirigidas. Como el suscriptor será el único que accederá a ese canal, podrá procesarlas a su ritmo de proceso, establecido por la latencia de las redes que tenga que atravesar.

Las peticiones de servicio son depositadas en el puerto mediante una invocación al método *AddRequest(method: string, args: object[])*. El aspecto únicamente proporciona el nombre del método y los argumentos necesarios,

siendo los attachments los encargados de traspasar la petición al servicio adecuado.

A continuación se muestra el aspecto de coordinación del apartado 3.4.1.2, en el que se define el protocolo a seguir cuando se reciba una petición *Withdrawal*:

```
COORD = (CLIENT.Withdrawal?(quantity, money) →
  SERVER.Withdrawal!(quantity, money) →
  SERVER.Withdrawal?(quantity, money) →
  CLIENT.Withdrawal!(quantity, money)).COORD
```

Este protocolo especifica que cuando se reciba una petición del servicio *Withdrawal* por el puerto cuyo *played_role* es CLIENT, deberá reenviarse dicha petición al puerto cuyo *played_role* es SERVER. Sin embargo, en la especificación de los aspectos PRISMA no se hace mención alguna a los puertos del componente a los cuales el aspecto puede solicitar peticiones de servicio externas. Esto es debido a que el aspecto no conoce los puertos que formarán parte del componente, con el objetivo de favorecer la reutilización de aspectos y componentes. Por este motivo, en la especificación únicamente se indica que un servicio se solicitará a una interfaz con un *played_role* específico. En la especificación de un componente, los puertos tienen un nombre identificativo, la interfaz y *played_role*, permitiendo establecer una relación entre los puertos del componente y los *played_roles* del aspecto.

En la implementación, esto se traduce en que cuando el método privado del aspecto *_Withdrawal* es invocado por un puerto cuyo subproceso (*played_role*) representado es CLIENT¹¹, debe invocar el mismo servicio por el puerto cuyo subproceso es SERVER. En el siguiente fragmento de código se muestra cómo se invocaría desde un aspecto al servicio *Withdrawal* por el puerto cuyo *played_role* es SERVER.

```
Object[] args = new object[] {quantity, money}
AsyncResult result =
  link.OutPorts["ICreditCardTransactions", "SERVER"].AddRequest(
    "Withdrawal", args);
```

El atributo *link* de los aspectos es la referencia que éstos tienen hacia el componente que los agrega, mediante la cual pueden acceder al atributo *OutPorts* que contiene la lista de puertos de salida que forman parte del componente. El aspecto obtiene la referencia del puerto en el que depositar la petición de un servicio a través de la interfaz que publica y el *played_role* que representa. Finalmente, la invocación del método *AddRequest* devolverá un objeto *AsyncResult* para procesar posteriormente los resultados obtenidos.

Una vez añadida una petición de servicio al puerto de salida, ésta debe ser proporcionada a los attachments que están suscritos al puerto. Mediante los

¹¹ La identificación del subproceso para servicios entrantes, es decir, proporcionados por el puerto de entrada, no ha sido implementada en el prototipo actual. En cambio, sí que ha sido implementada para los servicios salientes.

métodos *RegisterListener(name: string)* y *UnRegister(name:string)* los attachments pueden suscribirse o desuscribirse a un puerto de salida y obtener con ello el canal exclusivo, que consiste en una referencia a una instancia de la clase *ListenerPort*. Una instancia de esta clase es creada cuando el attachment se suscribe al puerto, borrada cuando se desuscribe y se almacena en la lista heredada *Port.attachmentsRegistered*. Contiene el nombre del suscriptor y una cola en la que dejar las peticiones destinadas a dicho suscriptor. Por tanto, cuando al puerto de salida le llega una nueva petición de servicio (como instancia de *MethodToListener*), ésta será copiada en las colas de los objetos *ListenerPort* asociadas a cada suscriptor. Periódicamente, como se verá en apartados posteriores, el suscriptor consultará dicha cola para comprobar si tiene peticiones pendientes de procesar.

4.2.4.5 Definición de Componentes y Conectores en .NET

Una vez vistos todos los elementos que forman parte de un componente, en esta sección se verá de forma global cómo se combinan para representar un componente especificado en PRISMA. A continuación se muestra el código correspondiente a la implementación del componente *Account*, cuya especificación PRISMA ha sido mostrada en el apartado 3.4.1.3.

```
using System;
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace CuentaBancaria {
    // Definición del Componente ACCOUNT
    [Serializable]
    public class Account: ComponentBase {

        // Constructor del componente Account
        public Account(string AccountName,
            <Atributos_requeridos_por_los_aspectos>
            decimal accountId, LOC initialLocation,
            <\Atributos_requeridos_por_los_aspectos>
            MiddlewareSystem middlewareSystem) :
            base (AccountName, middlewareSystem) {

            <Definicion_Aspectos>
            // Creación del aspecto funcional
            IAspect functionalAspect = new BankInteraction(accountId);
            AddAspect(functionalAspect);
            // Creación del aspecto distribución
            IAspect distributionAspect = new ExtMbile(initialLocation);
            AddAspect(distributionAspect);
            <\Definicion_Aspectos>

            /** Otra forma de obtener las referencias a los aspectos agregados: **
            IAspect functionalAspect = GetAspect(typeof(FunctionalAspect));
            IAspect distributionAspect = GetAspect(typeof(DistributionAspect));
            *****/

            <Definicion_Weavings>
```



```

// Creamos el Weaving entre el aspecto funcional y el de distribución
AddWeaving( functionalAspect, "ChangeAddress",
            WeavingType.BEFORE,
            distributionAspect, "Move");
<\Definicion_Weavings>

<Definicion_Puertos>
InPorts.Add( "AccountPort", "ICreditCardTransactions",
            functionalAspect, "SERVER");
OutPorts.Add( "AccountPort", "ICreditCardTransactions", "SERVER");
<\Definicion_Puertos>
    }
}
}
}
}

```

Como se puede observar, todo el código correspondiente a la definición de las propiedades del componente (aspectos, weavings y puertos) se define en el cuerpo del constructor de la clase. La clase *Account* representa al componente del mismo nombre especificado en PRISMA, y hereda de *ComponentBase* todo su comportamiento, como ya se ha descrito en las secciones anteriores. Si se estuviese definiendo un conector, la clase también debería especificar que implementa la interfaz *IConnector*, con el objetivo de que sea tratado como tal.

Los parámetros iniciales que requiere el constructor son el nombre identificativo que tendrá la instancia del componente creado, una referencia al *middleware*, y los atributos iniciales que necesiten los aspectos, tal como se define en la sección *Initialize* del ADL PRISMA. El cuerpo del constructor sigue una estructura genérica:

- *Definición de los Aspectos:* en primer lugar se crean los aspectos, se almacenan en una variable local para facilitar posteriormente su acceso, y se agregan al componente a través de invocaciones al método *AddAspect*.
- *Definición de los Weavings:* en segundo lugar se crean los weavings mediante invocaciones al método *AddWeaving*, tal como se vió en el apartado correspondiente.
- *Definición de los Puertos:* finalmente se crean los puertos de entrada y de salida, mediante las invocaciones respectivas a los métodos *Add*, con lo que las interfaces implementadas por los aspectos son publicadas al exterior y se habilita el medio de comunicación de los mismos hacia otros componentes externos.

El orden en que se realizan los procesos es importante, pues para crear un puerto o un weaving es necesario que previamente se haya agregado el aspecto correspondiente.

De este modo el componente, gracias a los distintos módulos que lo integran, permite la modificación de su estructura interna (puertos, aspectos y weavings) en tiempo de ejecución sin necesidad de detener sus operaciones por completo y recompilarse. Sin embargo, el prototipo actual únicamente

proporciona los servicios necesarios para que cada instancia de un componente, a modo individual, pueda ser modificada. En cambio, para que el prototipo soporte la evolución de forma completa aún tienen que ser introducidos una serie de mecanismos adicionales que permitan la coordinación entre las diferentes instancias y la propagación de los cambios a los distintos *middlewares*.

4.2.5 Sistemas

Una vez vista la implementación de los componentes PRISMA, a continuación se presentará el diseño y la implementación de los sistemas. Estas entidades permiten especificar componentes complejos, formados a partir de la agregación de componentes y conectores, interconectados entre sí mediante attachments, y conectados a los puertos del sistema a través de bindings.

4.2.5.1 Modelo de ejecución

Un sistema es un componente que puede tener aspectos, definir weavings entre ellos y publicar los servicios implementados por los aspectos a través de los puertos. Además, al ser una componente compleja, puede estar compuesto a su vez por componentes y conectores y definir bindings y attachments. Por tanto, un sistema es un componente que extiende su comportamiento con servicios adicionales para gestionar las relaciones de composición entre los elementos que lo forman.

Es por esto que el sistema se ha implementado como una clase especializada de *ComponentBase*, por lo que hereda el comportamiento de los componentes, pero que además define una serie de servicios específicos que caracterizan al sistema, agrupados en la interfaz *ISystem* (ver Figura 34). Dichos servicios permiten añadir o eliminar componentes (*AddComponent*, *RemoveComponent*), conectores (*AddConnector*, *RemoveConnector*), attachments (*AddAttachment*, *RemoveAttachment*) o bindings (*AddBinding*, *RemoveBinding*). La clase *SystemBase* es la que implementa esta interfaz.

Todas las entidades que forman parte de un sistema (componentes, conectores, attachments y bindings) son instanciadas, mantenidas y eliminadas por el *middleware*. El sistema únicamente comprueba que se cumplen las precondiciones (que el componente a agregar no ha sido definido ya, que un attachment se define para un componente y conector, etc.) e invoca los servicios correspondientes del *middleware*. El *middleware* se encarga de buscar el tipo necesario, cargarlo y ponerlo en ejecución (para los servicios de creación), de mantener las distintas entidades en un estado consistente (para los servicios de movilidad), y de eliminarlas. El sistema únicamente mantiene una serie de listas dinámicas con los nombres identificativos (y únicos) de las distintas entidades que lo forman. En tiempo

de ejecución, cuando el sistema requiera ejecutar un servicio del *middleware* relacionado con alguna de las entidades que lo forman (por ejemplo, mover un componente, eliminar un attachment, etc.), proporcionará su correspondiente identificador.

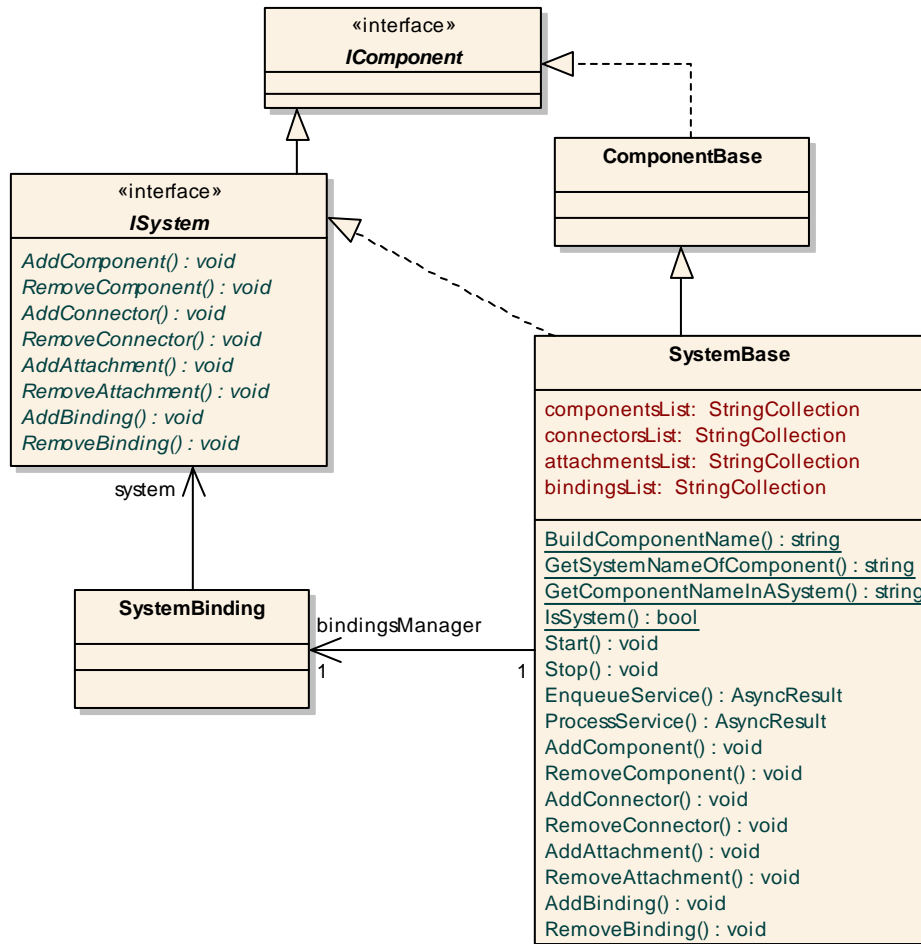


Figura 34 - Interfaz ISystem y clase SystemBase

En un diseño preliminar del Sistema, se intentó que las entidades que lo forman (componentes, conectores, attachments y bindings) estuviesen encapsuladas en su interior como objetos agregados, en lugar de guardar sólo el identificador. También se planteó la posibilidad de que la gestión (creación, modificación y borrado) de las distintas entidades fuese realizada completamente por el sistema. Sin embargo, en el caso de aquellos componentes y conectores internos de un sistema que residan en máquinas distintas, la gestión sólo podrá realizarse en la máquina correspondiente donde se alojen. Además, el sistema no podría tener la referencia remota de dichos elementos ya que, como se verá más adelante, los componentes y conectores no se publican para acceso remoto. En cambio, si la gestión es realizada parcialmente por el *middleware*, éste puede encargarse de propagar la solicitud de determinados servicios (i.e. mover un componente a otra ubicación) al *middleware* que contenga el componente remoto, de forma totalmente transparente para el sistema. Es preferible que el sistema se

ocupe de los requisitos funcionales (mayor nivel de abstracción) y dejar que el *middleware* realice las operaciones de más bajo nivel, como la creación y eliminación de objetos, acceso remoto, serialización, búsqueda y transferencia de tipos, etc.

Por esta razón, la clase *SystemBase* define los atributos *componentsList*, *connectorsList*, *attachmentsList* y *bindingsList* como listas dinámicas de cadenas, en las que se almacenan los nombres identificadores de cada uno de los elementos que forman parte del sistema. Es fundamental que dichos nombres sean únicos para todo modelo arquitectónico PRISMA en ejecución, para que el *middleware* pueda localizar de forma unívoca dónde se encuentran los elementos residentes en distintas máquinas. A nivel de implementación los componentes/conectores son tratados por el *middleware* como si estuviesen agregados de forma referencial. Esto es necesario para que el *middleware* pueda gestionar cada elemento de forma individual y, además de su creación y destrucción, poder moverlos a otras máquinas independientemente de la máquina donde se encuentre el sistema. En caso contrario, los componentes y conectores no podrían ser reubicados a distintas máquinas, pues el *middleware* no tendría acceso a ellos.

En la especificación PRISMA de un sistema, para cada componente y conector agregado, se asocia un nombre identificativo mediante el cual se puede hacer referencia a dichos componentes internos dentro del ámbito del sistema. Sin embargo, cuando un sistema es instanciado, debe utilizarse un nombre que identifique de forma única a cada componente interno agregado al sistema, y que sea válido a nivel de modelo arquitectónico. Esto es necesario debido a que cada componente interno es registrado en el *middleware* como si fuese un componente independiente. Por ejemplo, si se define un sistema del tipo “SystemType” formado por un componente cuyo nombre es “Component”, y se crean dos instancias de dicho sistema, “System1” y “System2”, los componentes internos no podrían registrarse en el *middleware* como “Component”, pues no habría forma de distinguir cuál de los dos pertenece a cada sistema.

Para solucionar este problema, se ha establecido que el nombre identificativo de un componente interno se forme por la concatenación del nombre de la instancia del sistema al cual pertenece, seguido del nombre del componente interno definido en la especificación. De esta forma, los nombres de los componentes internos del ejemplo anterior serían: “System1.Component” y “System2.Component”. El método estático *BuildComponentName* construye el nombre identificativo de un componente o conector interno tomando como entrada el nombre del sistema y el nombre del componente definido en la especificación. Los métodos estáticos *GetSystemNameOfComponent* y *GetComponentNameInASystem* permiten obtener a partir del nombre identificativo de un componente interno, el nombre del sistema al cual pertenece y el nombre del componente definido en la especificación, respectivamente.

Por otra parte, el método estático *IsSystem* permite, dado un tipo, comprobar si es un sistema. Es utilizado por el *middleware* en la carga de tipos. Los métodos de creación y borrado de componentes, conectores, attachments y bindings, de forma general, comprueban en primer lugar si se cumplen las distintas precondiciones, como que el elemento a crear no existe ya o que los tipos proporcionados como parámetros son los correctos. Después, invocan los servicios equivalentes del *middleware* (*CreateComponent*, *RemoveComponent*, *CreateAttachment*, *RemoveAttachment*) o del gestor de bindings (*CreateBinding*, *RemoveBinding*). Finalmente, si dicha invocación ha tenido éxito, los nombres identificativos de los elementos creados (o eliminados) son añadidos (o borrados) de las listas internas del sistema. A continuación se muestran las firmas de los distintos métodos que permiten modificar los elementos arquitectónicos que componen el sistema:

```
void AddComponent(string componentName, Type componentType, string targetURL,
                 object[] args);
void RemoveComponent(string componentName);

void AddConnector(string connectorName, Type connectorType, string targetURL,
                 object[] args);
void RemoveConnector(string connectorName);
```

Como se puede observar, los métodos para crear un componente o un conector requieren los mismos tipos de parámetros, pues su implementación es idéntica, como ya se ha visto anteriormente. Los parámetros requeridos son el nombre definido en la especificación, el tipo del componente/conector a agregar al sistema, la URL donde se ubicará dicho componente/conector inicialmente, y un vector con los parámetros adicionales que pueda requerir. El borrado de un componente o conector se realiza proporcionando el nombre identificativo.

```
void AddAttachment(string componentName, string portName,
                  string connectorName, string roleName);
void RemoveAttachment(string attachmentName);
```

La creación de attachments entre un componente y un conector se realiza de forma similar a como se define en la especificación PRISMA. En primer lugar se proporciona el nombre y el puerto del componente que se desea conectar, y en segundo lugar el nombre y el rol del conector. El borrado se realiza proporcionando el nombre del attachment a borrar.

```
void AddBinding(string systemPortName, string componentName,
               string componentPortName);
void RemoveBinding(string bindingName);
```

La creación de bindings se realiza proporcionando el nombre del puerto del sistema que se desea conectar con el componente o conector, el nombre del componente (o conector) y el puerto (o rol). Para borrar un binding debe proporcionarse el nombre identificativo del binding a eliminar.

Debido a que el sistema agrega una serie de elementos adicionales frente a los definidos por *ComponentBase*, la clase *SystemBase* reemplaza los métodos heredados *Start()*, *Stop()*, *EnqueueService()* y *ProcessService()* para definir el comportamiento adicional. Los métodos *Start* y *Stop* se encargan de indicar al *middleware* que inicie (o detenga) la ejecución de los componentes, conectores, attachments y bindings que lo componen. Tras esto, el sistema invoca el comportamiento heredado *Start()* o *Stop()* para que el hilo de ejecución propio del sistema sea iniciado (o detenido).

Los métodos *EnqueueService* y *ProcessService* han sido reescritos para contemplar la redirección de peticiones de servicio desde los puertos a los bindings asociados. Para el caso en que a un puerto se le ha asociado un binding, el método *EnqueueService* es sobrescrito para, además de insertar en la cola la petición de servicio recibida, añadir también un delegado hacia el binding que debe ejecutarse. Por su parte, el método *ProcessService* añade el comportamiento necesario para invocar adecuadamente dicho delegado.

4.2.5.2 Attachments

En el modelo PRISMA, la comunicación entre componentes y conectores se realiza a través de los attachments. Cada attachment conecta dos elementos (componente y conector) y se encarga de llevar peticiones desde el uno al otro, actuando como si fuera un canal de comunicación. De esta forma, los attachments añaden una capa de abstracción adicional, permitiendo que las comunicaciones distribuidas sean transparentes para componentes y conectores. La comunicación es bidireccional, ya que por una parte se encargan de hacer accesible al componente desde otras ubicaciones remotas (comportamiento servidor), y por otra, se encargan de establecer la comunicación hacia los componentes remotos (comportamiento cliente). Además, también permiten que los distintos elementos arquitectónicos interconectados funcionen de forma independiente, sin que se conozcan entre sí.

A nivel de implementación, la comunicación distribuida entre objetos se ha realizado mediante la tecnología .NET REMOTING, ya que proporciona implementados la mayor parte de los mecanismos para acceder remotamente a objetos así como para serializarlos y moverlos de una máquina a otra. REMOTING permite que una clase que herede de *MarshalByRefObject* sea accesible remotamente, creando automáticamente en el cliente un proxy del objeto remoto que se encargue de establecer las comunicaciones entre cliente y servidor de forma transparente para ambos. Para que un objeto pueda ser accesible remotamente debe ser publicado previamente. Esto se reduce a una llamada al método *RemotingServices.Marshal()*, para el que se le debe proporcionar la instancia del objeto a publicar, el URI a través del cual será accesible, y el tipo. Por otra parte, mediante la serialización un objeto puede ser convertido a un flujo de bytes, enviado por el canal de comunicación y al ser deserializado vuelto a

convertir en un objeto que mantiene su estado previo. Sin embargo, como limitación un objeto que herede de *MarshalByRefObject* siempre será accedido de forma remota y no podrá enviar una copia de sí mismo a otra máquina, porque lo que envíe será siempre un proxy hacia él mismo. Por tanto, un componente PRISMA no podrá tener la capacidad de moverse a la vez que también es accesible remotamente. La movilidad es la capacidad de un componente para moverse a otro equipo y seguir su proceso de ejecución en el punto que lo dejó, manteniendo su estado. Por esta razón, se han separado las propiedades de movilidad y acceso remoto: los componentes conservan la movilidad y los attachments se encargan del acceso remoto. De esta forma, todas las clases que forman parte de los componentes son serializables, mientras que los attachments que deban ser accesibles remotamente heredan de *MarshalByRefObject*.

De forma general, un attachment comunica un puerto de salida de un componente con el puerto de entrada de un conector y viceversa, proporcionando una comunicación bidireccional. Además, puede conectar un puerto y un rol con distinta interfaz, siempre y cuando una de ellas sea un subconjunto de la otra, pero esto obliga a que el attachment se diseñe para una combinación de interfaces específica. Los elementos arquitectónicos a conectar pueden encontrarse en la misma máquina o en distintas máquinas, en cuyo caso la comunicación es distribuida. Además, éstos pueden reconfigurarse dinámicamente, por lo que los attachments deben diseñarse de forma que puedan establecerse nuevas conexiones en tiempo de ejecución.

Por tanto, un attachment debe crearse de forma dinámica para una combinación de interfaces (con servicios comunes), definida por los puertos que conecta. La implementación se ha realizado a través de la herencia, de forma similar a los puertos de entrada. Por una parte, el comportamiento general del attachment se ha definido en una serie de clases genéricas. Por otra parte, el comportamiento específico, dependiente de la interfaz cuyas comunicaciones debe redirigir, es generado en tiempo de compilación a la vez que se generan las interfaces .NET correspondientes a las interfaces PRISMA. Para evitar la generación de un attachment por cada posible combinación de interfaces que pueda conectar, se generan fragmentos de código específicos para cada interfaz. Posteriormente, en tiempo de ejecución, el attachment se construye combinando los fragmentos correspondientes a las interfaces que conecta. De esta forma, por cada interfaz generada, también se genera el código necesario para construir dinámicamente el attachment correspondiente, cuyo comportamiento es heredado de las clases genéricas.

Un attachment tiene dos comportamientos claramente diferenciados: cliente y servidor. El comportamiento cliente consiste en escuchar periódicamente las peticiones depositadas en el puerto de salida que está escuchando, y es por ello que requiere un hilo de ejecución propio. El comportamiento servidor consiste en reenviar las peticiones recogidas al puerto de entrada. Como los elementos arquitectónicos que conecta el attachment pueden estar

en máquinas distintas, se ha subdividido al attachment en dos entidades, cada una de las cuales representa a un participante de la comunicación (componente o conector) y que se ubican en la misma máquina del que representan. Estas entidades son instancias de la clase *Attachment*.

Cada una de estas entidades podrá actuar como cliente o como servidora dependiendo de la dirección en que fluya la comunicación. La que actúe como cliente deberá, tras recoger una petición, reenviarla a la otra entidad, que actuará como servidora. La servidora deberá publicarse a través de Remoting para poder ser accedida de forma remota, por lo que también heredará de *MarshalByRefObject*. Para separar ambos comportamientos se ha optado por implementarlos en clases separadas: *AttachmentClientBase* para el comportamiento cliente, y *AttachmentServerBase* para el comportamiento servidor. Estas clases sólo definen el comportamiento genérico, que es heredado por los attachments específicos generados para cada interfaz en tiempo de compilación, como se ha comentado anteriormente.

Por tanto, un attachment PRISMA se corresponde con una pareja de instancias de la clase *Attachment*, una asociada al componente y otra asociada al conector, cada una de las cuales está formada por la agregación de las clases que implementan el comportamiento cliente y el comportamiento servidor. En la Figura 35 se muestra el esquema general de funcionamiento de los attachments y su composición interna. En cursiva se muestran los nombres de las clases que definen el comportamiento de cada módulo. El esquema muestra el attachment PRISMA definido para conectar dos puertos cuyas interfaces publicadas son “InterfazX” e “InterfazY”. Los módulos “Interface{X|Y}Client” e “Interface{X|Y}Server” heredan respectivamente de *AttachmentClientBase* y de *AttachmentServerBase* y son generados en tiempo de compilación para incorporar el comportamiento específico de los attachments.

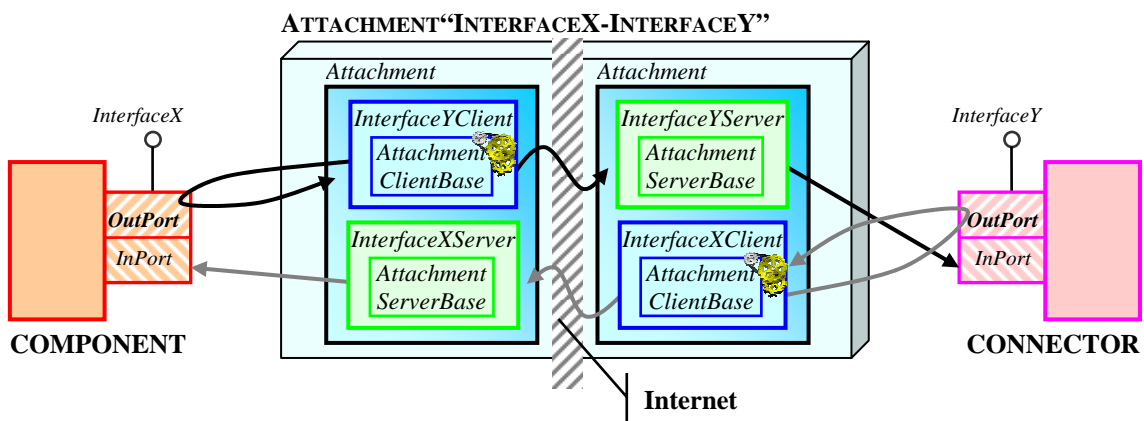


Figura 35 - Esquema de funcionamiento de los Attachments

La comunicación entre un componente y un conector se realiza creando una pareja de attachments, cargando los tipos (Client y Server) correspondientes a la interfaz de los puertos que se desea conectar, instanciándolos e

iniciando su ejecución mediante la llamada a *AttachmentStart()*, como se verá más adelante. Los servicios que el attachment acepta (cliente) y reenvía (servidor) son el resultado de la intersección de los servicios definidos en las interfaces de los puertos. Por ejemplo, si en "InterfaceX" se define un servicio *ChangeName* no definido en "InterfaceY", y se deja en el puerto de salida una petición de este servicio, el módulo "InterfaceYClient" no recogerá dicha petición pues no está contemplada en la interfaz que implementa, y por tanto tampoco podrá solicitarse en el destino dicho servicio. De esta forma, este diseño resuelve el problema de conectar interfaces compatibles sin tener que comparar entre sí los métodos definidos en las interfaces.

Sin embargo, el diseño puede ser optimizado para crear los attachments de forma totalmente dinámica, sin necesidad de generar código, utilizando la reflexión y los delegados. El inconveniente es que los tipos de delegados necesarios deben buscarse en los ensamblados. Por otra parte, el diseño también puede simplificarse si se desea que la comunicación sea unidireccional, eliminando para ello el comportamiento servidor del extremo que actúa exclusivamente como cliente y el comportamiento cliente del extremo que actúa solamente como servidor.

Una vez visto el modelo de ejecución de los attachments, a continuación se describirán los detalles de implementación. En la Figura 36 se muestra el diseño de clases para dar soporte a los attachments. Los attachments se almacenan en una colección del tipo *AttachmentsCollection*, que proporciona los métodos necesarios para añadir y eliminar attachments, y es gestionada por el *middleware*. Como ya se ha comentado, un attachment PRISMA se corresponde con dos instancias de la clase *Attachment*, cada una de las cuales contiene información sobre el componente o conector al cual representan, además de la información necesaria sobre su pareja para poder establecer la conexión remota. En caso de que el otro extremo de la conexión se mueva a otra máquina, el attachment deberá volver a conectarse con su pareja, y para ello necesitará el nombre de su pareja, la URL donde se encuentra y el tipo del objeto servidor que es accesible remotamente.

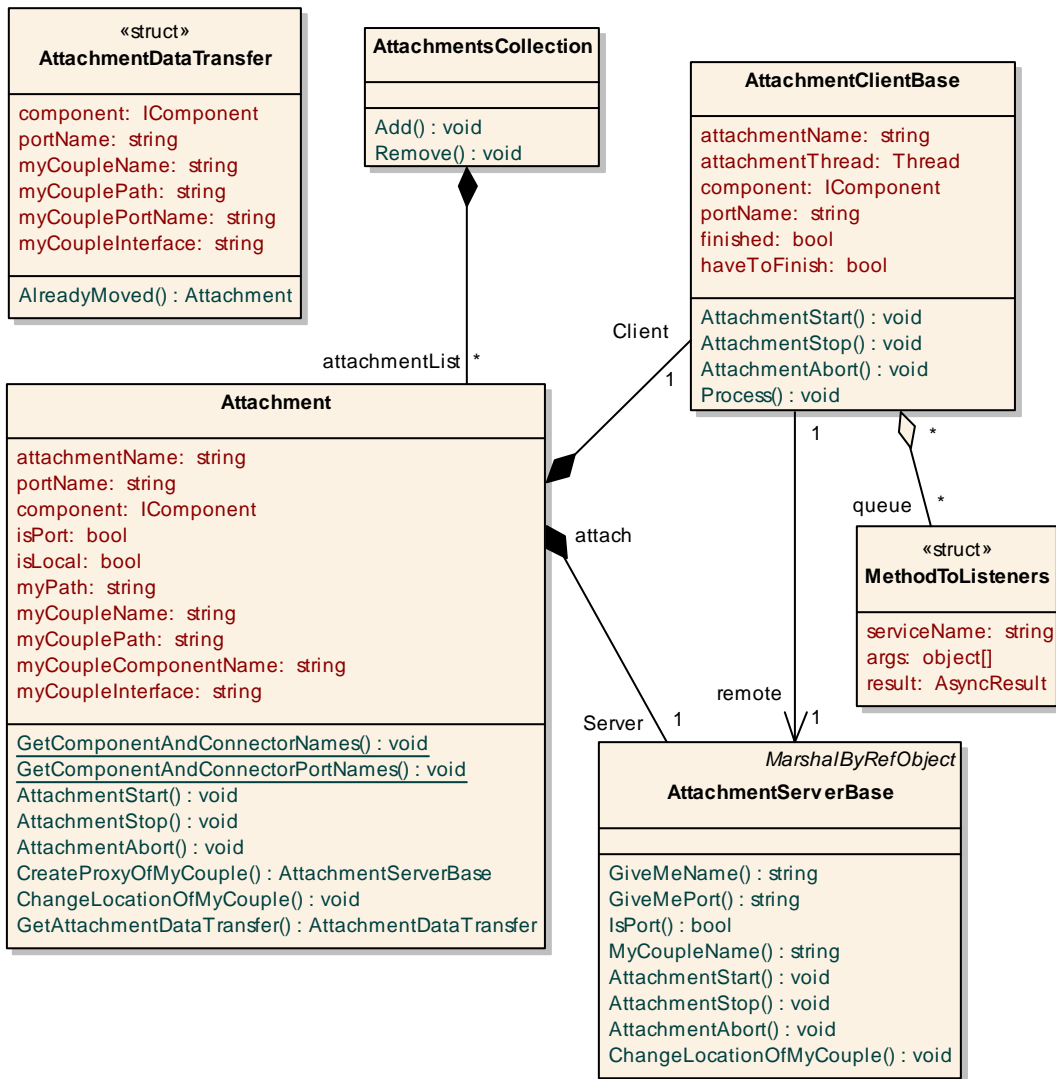


Figura 36 - Clases asociadas al Attachment

El nombre identificativo del attachment debe construirse de tal manera que no provoque ambigüedad en caso de haber más de un attachment conectado al mismo componente y conector. Una alternativa posible es que el nombre esté formado por un identificador numérico, pero el problema es que cada *middleware* generará uno propio, que tal vez coincida con el generado por otro *middleware*. Por esta razón, se ha optado por un mecanismo de identificación más sencillo y global: utilizar una cadena formada por el nombre identificativo del componente y conector que conecta el attachment, además de los puertos que conecta. De esta forma, el attachment queda identificado de forma unívoca por la conexión que establece. Sin embargo, el principal inconveniente de este mecanismo de identificación es que el nombre puede ser largo y por tanto la búsqueda de un determinado attachment en una lista puede ser más costosa que el uso de un identificador numérico.

La clase *Attachment* (ver Figura 36) define los siguientes atributos y servicios:

- **attachmentName**, **portName**, **myPath**, **component**: atributos básicos de un attachment, como su nombre identificativo, el nombre del puerto al cual escucha (comportamiento cliente) y al cual reenvía peticiones recibidas (comportamiento servidor), la URL donde se encuentra, y la referencia al componente al cual el *attachmentServer* actúa como proxy. A través de esta referencia el attachment podrá invocar los servicios del puerto de entrada y recoger las peticiones del puerto de salida.
- **isPort**, **isLocal**: flags que permiten conocer si la instancia actual representa a un componente (*isPort=true*) o a un conector (*isPort=false*), o si el par de attachments está en la misma máquina (*isLocal=true*) o en máquinas distintas (*isLocal=false*).
- **myCoupleName**, **myCouplePath**, **myCoupleComponentName**, **myCoupleInterface**: atributos que permiten conocer los datos del otro miembro de la pareja del attachment actual. Son utilizados para establecer la comunicación entre el Cliente y el Servidor.
- **Client**, **Server**: atributos que contienen las referencias a los objetos agregados *AttachmentClientBase* y *AttachmentServerBase*, respectivamente.
- **GetComponentAndConnector[Names|PortNames]()**: métodos estáticos que permiten obtener, a partir del nombre identificativo de un attachment, el nombre del componente y conector o el puerto y rol que une.
- **Attachment[Start|Stop|Abort]()**: servicios que inician, paran o detienen la ejecución de un attachment. El inicio de un attachment implica su registro en los puertos de entrada y de salida, y la interconexión entre los miembros de la pareja de attachments. De la misma forma, la parada implica la desuscripción de los respectivos puertos. A su vez, la invocación de estos servicios es propagada al objeto agregado *Client* para el inicio, parada o detención del hilo de ejecución responsable de escuchar las peticiones depositadas en el puerto de salida.
- **CreateProxyOfMyCouple()**: servicio para crear un proxy hacia el objeto *server* del otro attachment. Se utiliza para poder acceder al otro attachment cuando se encuentra en una máquina distinta.
- **ChangeLocationOfMyCouple()**: servicio que actualiza la ubicación donde se encuentra la otra pareja del attachment, con el objetivo de que vuelva a establecerse la comunicación entre los dos miembros de la pareja (el cliente se conecte al servidor remoto y el cliente remoto se conecte al servidor local).
- **GetAttachmentDataTransfer()**: servicio de creación de un objeto serializable (de tipo *AttachmentDataTransfer*) con la información relevante del attachment para que cuando éste vaya a ser movido a otra máquina pueda reconstruirse de nuevo manteniendo el estado.

La invocación del método *AlreadyMoved()* de este objeto crea de nuevo el attachment con los datos transferidos.

La clase *AttachmentClientBase* define el comportamiento cliente del attachment. Tiene los atributos necesarios para conectarse al puerto del componente (o conector) del cual recoge peticiones, y un atributo *queue* que es una referencia a la cola exclusiva proporcionada por el puerto de salida, en la que se van depositando las peticiones dirigidas al attachment. Para ello, el cliente tiene un hilo de ejecución propio que cada cierto tiempo comprueba el estado de dicha cola.

Esta alternativa proporciona la máxima independencia del attachment respecto al puerto de salida, ya que éste en ningún momento interactúa directamente con el attachment, tan sólo va depositando mensajes para que sean recogidos. Sin embargo, como desventaja dicho hilo de ejecución ocupa (pequeños) ciclos de CPU para comprobar si hay peticiones en la cola, que en caso de no haberlas, se convierte en una operación innecesaria. Otra posible alternativa podría ser que dicho hilo estuviese en estado suspendido hasta que el puerto de entrada dejase una petición en su cola y lanzase un evento que lo despertase. De esta forma, el hilo del cliente únicamente estaría activo cuando realmente tuviese peticiones que procesar. No obstante, el puerto de salida necesitaría acceder al hilo de ejecución del attachment para despertarlo, lo que conlleva una pérdida de independencia del puerto.

Los flags *finished* y *haveToFinish* son indicadores del estado de ejecución de dicho hilo y que permiten detenerlo cuando sea invocado el servicio *AttachmentStop*. El atributo *remote* contiene un proxy hacia el *Server*, que puede ser local o remoto, al cual se le reenvían las peticiones recibidas. El servicio *Process()* es invocado cada vez que se detecta una nueva petición y debe ser sobrecargado por el *AttachmentClient* específico generado para la interfaz concreta.

A continuación se muestra el código generado para el attachment específico de la interfaz *ICreditCardTransactions*, comportamiento cliente. El nombre del tipo generado es: *AttachmentICreditCardTransactionsClient*

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace CuentaBancaria {

    // Clase generada automáticamente al generarse el código de la interfaz
    // "ICreditCardTransactions"
    [Serializable]
    public class AttachmentICreditCardTransactionsClient: AttachmentClientBase {

        // Constructor - Llama al padre
        public AttachmentICreditCardTransactionsClient
            (IComponent component, string portName, string attachmentName)
            : base(component, portName, attachmentName) {}

        // Procesa una petición de servicio
    }
}
```

```
public override void Process(MethodToListeners service) {
    try {
        switch(service.serviceName) {
            case "Withdrawal": {
                decimal arg0 =(decimal)service.Args[0];
                decimal arg1 =(decimal)service.Args[1];
                AsyncResult result = ((ICreditCardTransactions) remote).
                    Withdrawal(arg0, ref arg1);
                service.Result.SetDerivedParameters(result);
                break;
            }
            case "Balance": { /*[Omitido]*/ }
            case "ChangeAddress": { /*[Omitido]*/ }
            case "Transfer": { /*[Omitido]*/ }
            default:
                break;
        }
    }
}
}
```

Como puede observarse, la mayor parte del código generado corresponde al cuerpo del método *Process*, en el que existe un “case” para cada servicio ofrecido por la interfaz. Cuando en la cola se reciba una petición de servicio, el cliente comprobará si el nombre del método pertenece a uno de los servicios de la interfaz, y en caso afirmativo, se invocará al servicio correspondiente del *Server*. Todos los servicios invocados en el *AttachmentServer* devuelven un proxy al objeto *AsyncResult* almacenado en la máquina destino, por lo que debe establecerse una relación entre el objeto *AsyncResult* local y el remoto, para que cuando el remoto obtenga los resultados, el local sea avisado convenientemente. Esto se realiza invocando el método *setDerivedParameters* del objeto *AsyncResult* local, almacenado en la cola del puerto de salida, y consultado por el aspecto que ha invocado el servicio externo.

Finalmente, la clase *AttachmentServerBase* es la que define el comportamiento servidor del attachment. Por una parte, actúa como objeto accesible remotamente que publica los servicios de una determinada interfaz y redirige las peticiones al componente/conector del cual actúa como proxy. Por otra parte, también actúa como proxy de la instancia *Attachment* a la cual pertenece. Esto se ha realizado para evitar publicar también la instancia de *Attachment*. Ésta debe ser accedida remotamente para la parada de los attachments o para actualizarla en caso de que el otro miembro de la pareja se haya movido a una máquina distinta.

A continuación se muestra el código generado para el attachment específico de la interfaz *ICreditCardTransactions*, comportamiento servidor. El nombre del tipo generado es: *AttachmentICreditCardTransactionsServer*.

```
using System;
using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;
```

```
namespace CuentaBancaria {

    // Clase generada automáticamente al generarse el código de la interfaz
    // "ICreditCardTransactions"
    [Serializable]
    public class AttachmentICreditCardTransactionsServer: AttachmentServerBase,
        ICreditCardTransactions {
        private ICreditCardTransactions InPortOfComponent;

        // Constructor - Llama al padre
        public AttachmentICreditCardTransactionsServer(Attachment attach):
            base (attach) {

            InPortOfComponent =
                (ICreditCardTransactions) Component.InPorts[attach.PortName];
            if (InPortOfComponent == null) {
                throw new Exception("ICreditCardTransactions's InPort does not
                    exists in " + Component.componentName + " component.");
            }
        }

        /* REDIRECCIÓN DE LOS SERVICIOS HACIA EL COMPONENTE DESTINO */

        public PRISMA.AsyncResult Withdrawal(decimal quantity, ref decimal money){
            try {
                return InPortOfComponent.Withdrawal(quantity, ref money);
            } catch (Exception e) {
                // Propagamos la excepción al cliente del servicio
                AsyncResult result = new AsyncResult(1);
                result.ThrowExceptionToClient(e);
                return result;
            }
        }

        public PRISMA.AsyncResult Balance(ref decimal money){/*[Omitido]*/}
        public PRISMA.AsyncResult ChangeAddress(string newAdd){/*[Omitido]*/}
        public PRISMA.AsyncResult Transfer(decimal quantity, ref decimal money) {
            /*[Omitido]*/ }
    }
}
```

Como puede observarse, el código generado también es mínimo. En el constructor, por motivos de eficiencia, se almacena en una variable de clase una referencia al puerto de entrada al cual el Server debe reenviar las peticiones recibidas. Por cada servicio definido en la interfaz, se define un método con la misma signatura cuya funcionalidad es reenviar la misma petición al puerto de entrada del componente o conector, y capturar las excepciones para propagarlas al cliente del servicio. Las excepciones son retransmitidas a través del objeto *AsyncResult*, mediante la invocación del método *ThrowExceptionToClient()*.

4.2.5.3 Bindings

En PRISMA, un binding es la entidad encargada de establecer la comunicación entre un componente/conector y el sistema que lo encapsula. Asocia un puerto de un sistema con un puerto de un componente o conector, actuando como un redirector de peticiones, de forma transparente para los componentes externos al sistema. Cuando un puerto de un sistema recibe

una petición de servicio, y éste tiene un binding asociado, la petición es reenviada al componente interno, que puede estar en otra máquina distinta a la del sistema. Por el contrario, cuando un componente/conector interno desea comunicarse con un componente externo, tras dejar la petición en su puerto de salida, el binding recoge dicha petición y la reenvía al puerto de salida del sistema.

Su diseño se ha realizado de forma similar a los attachments. Un binding está formado por dos partes, cada una de las cuales reside en la misma máquina que la entidad a la cual representa. La diferencia radica en que la parte que representa al sistema se ha diseñado de forma específica para que sea común para todos los bindings de dicho sistema. En la Figura 37 se muestra el esquema de funcionamiento de un binding PRISMA.

La parte que reside en el lado del componente/conector, a la que se ha denominado *ComponentBinding*, conserva el mismo diseño de los attachments: está formada por un objeto que implementa el comportamiento cliente (*ComponentBindingClient*) y otro objeto con el comportamiento servidor (*ComponentBindingServer*). El objeto cliente se encarga de recibir las peticiones de servicio del puerto de salida del componente y redirigirlas hacia el otro extremo de la comunicación, la parte residente en el lado del sistema. El objeto servidor se encarga de recibir las peticiones enviadas por la parte del binding residente en el sistema y redirigirlas al puerto de entrada del componente.

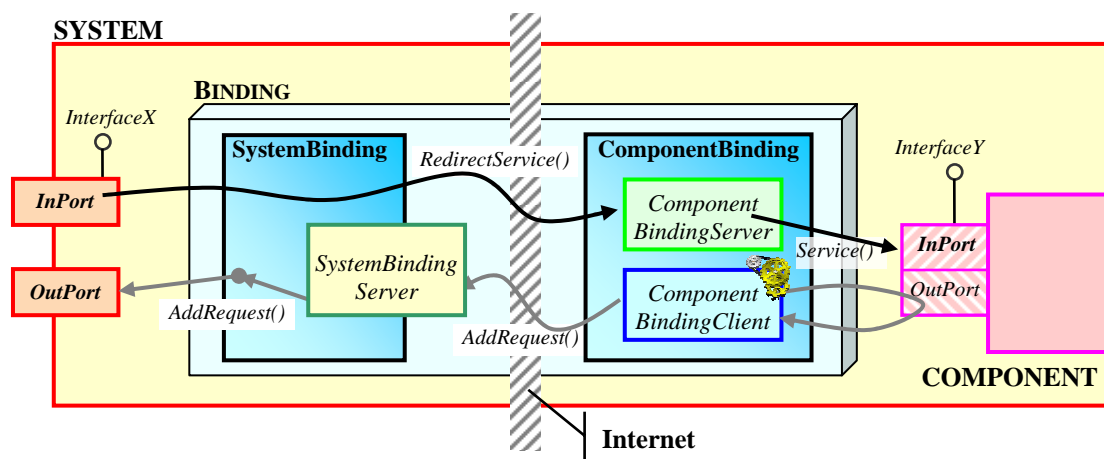


Figura 37 - Esquema de funcionamiento de un Binding

La parte que reside en el lado del sistema, a la que se ha denominado *SystemBinding*, realiza dos tareas:

- Realiza las funciones complementarias al *ComponentBinding*. Por una parte, recoge todas las peticiones de servicio salientes de los componentes internos, enviadas a través de los *ComponentBindings*, y las reenvía a los puertos de salida correspondientes. Por otra parte, inicializa adecuadamente los puertos de entrada para que redirijan las peticiones de servicio entrantes hacia los componentes internos.

- Gestiona los bindings del sistema, encargándose de la creación y eliminación de bindings. Existe sólo una instancia de *SystemBinding* por cada sistema, y es la encargada de crear los respectivos *ComponentBindings* por cada binding definido. También es la encargada de actualizar las referencias en caso de movilidad. Cuando detecta que existen *ComponentBindings* remotos, publica una parte de sí misma (*SystemBindingServer*) para ser accedida remotamente.

A diferencia de los attachments, los cuales requieren dos hilos de ejecución por cada módulo cliente (uno en cada máquina) para realizar la escucha de los puertos de salida, los bindings se han implementado de forma diferente para mejorar el rendimiento. La parte del binding residente en el sistema (*SystemBinding*) no requiere escuchar de forma activa los puertos de entrada y salida del sistema, pues no existen colas de espera. Las peticiones de servicio recibidas por el puerto de entrada pueden ser redireccionadas inmediatamente al *ComponentBinding*, y lo mismo ocurre con las peticiones enviadas por éste al *SystemBinding*: sólo hay que reenviarlas al puerto de salida correspondiente. Por esta razón, no ha sido necesario un hilo de ejecución propio para el *SystemBinding*, aunque los *ComponentBindings* sí que lo requieren para la escucha periódica de las peticiones generadas por el puerto de salida del componente interno.

Habrán tantos *ComponentBindings* como bindings haya definidos en el sistema, pero sólo un único *SystemBinding* para todos ellos. Esto se ha realizado de esta forma con la finalidad de encapsular en una única clase toda la funcionalidad necesaria para la gestión de los bindings definidos en un sistema, y para evitar la publicación de un *SystemBindingServer* por cada binding definido. De esta forma, todos los bindings remotos de un mismo sistema confluyen a un único objeto, que es el que gestiona la concurrencia para acceder a los puertos del sistema.

El modelo de ejecución de los bindings es semejante al de los attachments. Por una parte, cuando el *ComponentBinding* detecta una petición de servicio en el puerto de salida del componente al cual está escuchando, invoca el método *AddRequest()* del *SystemBinding* proporcionándole el nombre del *ComponentBinding*, el servicio a ejecutar y sus parámetros. Dicho método, tras comprobar que ha sido invocado por un *componentBinding* válido, obtiene el puerto del sistema al cual redirigir la petición e invoca al método *AddRequest()* del puerto de salida correspondiente del sistema, pasándole los parámetros recibidos desde el *componentBinding*.

Por otra parte, la redirección de las peticiones de servicio entrantes hacia los *ComponentBindings* se ha realizado a través de un atributo de los puertos de entrada: *ComponentBinding*. Cuando se asocia un binding a un puerto, el objeto *SystemBinding* inicializa su atributo *ComponentBinding* con una referencia al *ComponentBindingServer* (que puede ser remoto o local). En tiempo de ejecución, cuando llega una nueva petición a un sistema, el puerto de entrada crea un delegado hacia el método *RedirectService* del *ComponentBindingServer* y lo almacena en la cola del sistema, con la

finalidad de preservar el orden de llegada de las peticiones. Posteriormente, al ser ejecutado dicho delegado por el sistema, la petición de servicio será reenviada al puerto de entrada del componente asociado. Este diseño presenta el inconveniente de que ha precisado modificar el puerto de entrada del sistema para que también se almacenase el binding asociado, resultando en una pérdida de independencia por parte de los puertos. Sin embargo, esto ha sido necesario porque el único punto donde pueden interceptarse las peticiones de servicio para redirigirlas hacia los bindings es en los puertos de entrada, ya que tras ser insertadas en la cola del sistema se pierde la información sobre el puerto de la cual procedían. Además, tampoco pueden redirigirse directamente, sin pasar por la cola del sistema, las peticiones recibidas en los puertos de entrada hacia los *ComponentBindings*, ya que en ese caso se perdería el orden de los servicios recibidos.

Los bindings, a diferencia de los attachments, son creados dinámicamente sin necesidad de generar código, gracias a la reflexión y al uso de los delegados proporcionados por .NET. Cuando se crea un nuevo binding, el *SystemBinding* indica al middleware que cree un nuevo *ComponentBinding* en la ubicación del componente remoto, indicándole entre otra información, el nombre del sistema con el cual se comunicará y el puerto y nombre del componente al cual debe reenviar peticiones. El constructor del *ComponentBinding* crea a su vez las instancias *ComponentBindingClient* y *ComponentBindingServer*, proporcionándoles la interfaz del puerto de salida del componente y del puerto de entrada del sistema, respectivamente. A partir de la interfaz proporcionada, *ComponentBindingClient* obtiene dicho tipo a través del middleware y por reflexión obtiene los métodos que la forman, añadiéndolos a una lista dinámica. En tiempo de ejecución, los servicios recogidos del puerto de salida son comparados con los de dicha lista, y en caso de coincidir, son reenviados al *SystemBinding*. Por su parte, y de forma similar, *ComponentBindingServer* obtiene los métodos de la interfaz a través de la reflexión, pero en lugar de almacenar únicamente los nombres de los servicios, crea también un delegado hacia el método destino. En tiempo de ejecución, cuando el método *RedirectService* es invocado, al que se le proporciona el nombre del método a ejecutar, se obtiene el delegado correspondiente y se ejecuta con los parámetros proporcionados. Otra posible estrategia para la invocación del servicio solicitado podría ser, en lugar de utilizar delegados, utilizar el método *Invoke()* que se proporciona por la reflexión. Sin embargo, y como ya se discutió en el apartado 4.2.4.3, el uso de delegados es más eficiente que la invocación de métodos por reflexión ([Gun04]).

Por otra parte, los bindings también implementan mecanismos para el inicio o la parada segura, a través de los métodos *Start* y *Stop* implementados tanto en el *SystemBinding* como en los *ComponentBindings*. En el *ComponentBinding*, la invocación del método *Start* supone el inicio del hilo de ejecución del *ComponentBindingClient*, mientras que el *Stop* supone la parada de dicho hilo, con lo que no se recogen nuevas peticiones. Sin

embargo, debido a que *SystemBinding* y *ComponentBindingServer* no disponen de un hilo de ejecución propio, el inicio y la parada se indica a través de un flag interno, *stopped*, que impide que se procesen y reenvíen peticiones de servicio. Se contemplan dos tipos de paradas, la parada del sistema o la parada de los componentes internos. En caso de que el sistema deba pararse (p. ej. porque va a ser movido), deben detenerse también todos los bindings asociados, lo que implica la parada de todos los *ComponentBindings*. Esto es realizado por el método *Stop()* del *SystemBinding*, al igual que el método *Start()* se encarga de reanudar la ejecución de los *ComponentBindings* asociados. En el caso de la parada de un componente interno, también deben ser parados los bindings que tenga asociados, al igual que los attachments. Esta tarea es realizada por el middleware a través de la invocación de los métodos *Start()* y *Stop()* de los *ComponentBindings*. El inicio o la parada puede ser invocada de forma local o remota a través del *ComponentBindingServer*, que propaga la petición al *ComponentBinding* y éste a su vez también detiene el módulo cliente.

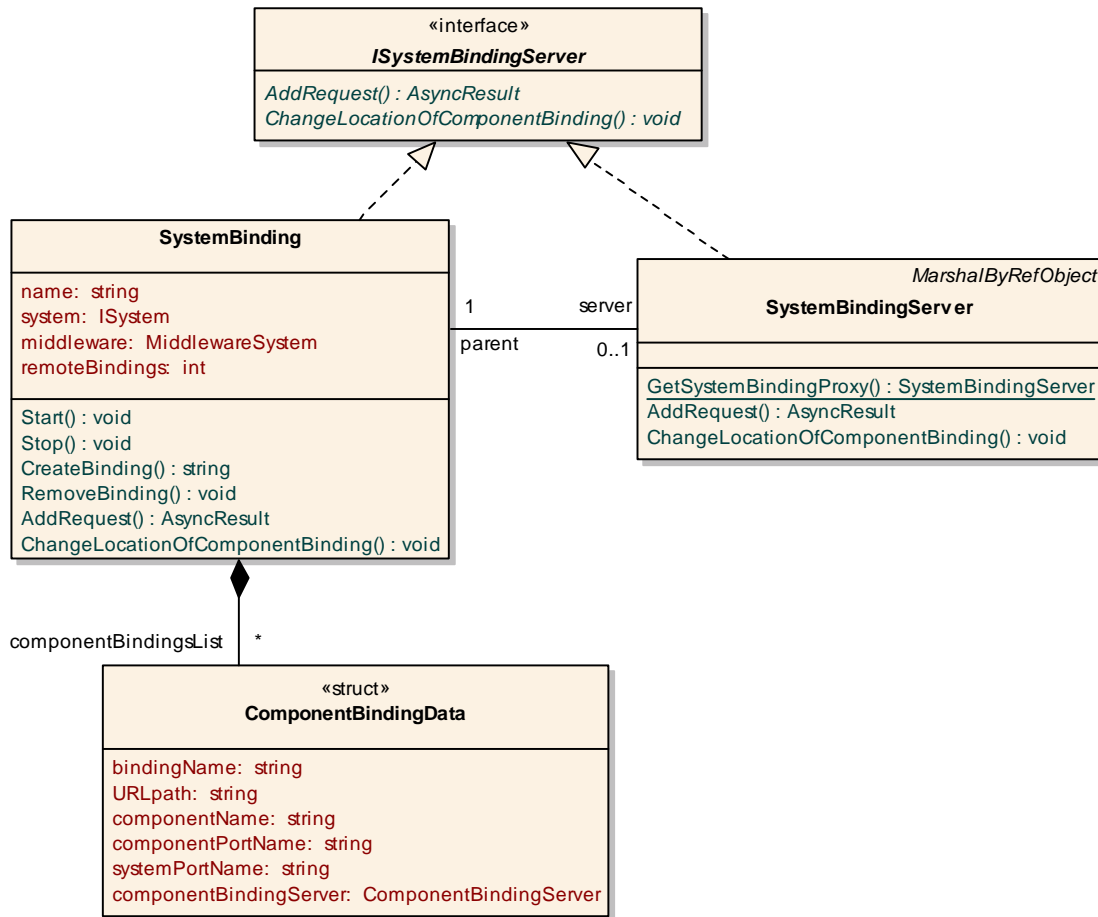


Figura 38 - Clases *SystemBinding* y *SystemBindingServer*

Una vez visto el diseño de los bindings, a continuación se describirán las características de las clases citadas anteriormente. La primera de ellas es la clase *SystemBinding* (ver Figura 38), cuyos atributos y servicios son:

- **name**: atributo que almacena el nombre identificativo de la instancia. Se construye a partir del nombre del sistema al cual pertenece, y se utiliza para crear el nombre de los *ComponentBindings*.
- **system, middleware**: atributo que guarda las referencias hacia el sistema al cual pertenece el gestor de bindings y hacia el middleware.
- **remoteBindings**: atributo que indica el número de bindings remotos que están utilizando el objeto *SystemBindingServer*. Este contador permite conocer cuándo debe publicarse o no dicho proxy.
- **server**: atributo para guardar la referencia hacia el *SystemBindingServer* creado.
- **componentBindingsList**: atributo que contiene una lista con la información sobre los distintos *ComponentBindings* creados. Permite conocer a qué puerto debe redireccionarse cada petición de servicio recibida desde los *ComponentBindings*, o volver a conectarse a ellos en caso de movimiento del sistema a otra máquina. Dicha información se almacena en tuplas *ComponentBindingData*, cuyos campos contienen el nombre identificativo del *ComponentBinding*, la URL donde se encuentra, el nombre del puerto y del componente al cual representa, el nombre del puerto del sistema, y una referencia a la parte servidora, que en caso de ser remota será un proxy.
- **Start()**, **Stop()**: servicios para iniciar o detener la ejecución de los bindings, respectivamente.
- **CreateBinding()**, **RemoveBinding()**: servicios para crear o eliminar un binding de forma dinámica, de la forma que se ha descrito anteriormente.
- **AddRequest()**: servicio que añade la petición de un servicio al puerto de salida del sistema, dado el nombre del *ComponentBinding* que invoca el método, el nombre del servicio requerido y sus argumentos.
- **ChangeLocationOfComponentBinding()**: servicio que actualiza con el nuevo valor el nodo correspondiente de la lista de *ComponentBindings*, dada una referencia a un objeto *ComponentBindingServer*. Se utiliza para reflejar el cambio de ubicación de un *ComponentBinding*, normalmente a causa de que su componente asociado se ha movido a otra máquina.

Por otra parte, los atributos y servicios de la clase *ComponentBinding* (Figura 39) se describen seguidamente:

- **name**: atributo que define el nombre del binding, para que pueda ser identificado de forma unívoca por el *middleware*. El nombre es creado a través del método estático *CreateName()*.
- **portName**, **myPath**, **componentRef**: atributos que contienen la información relativa al componente al que representa, como el nombre del puerto al que se debe escuchar y reenviar peticiones, la

URL donde se encuentra la instancia de *ComponentBinding*, y una referencia para poder acceder al componente.

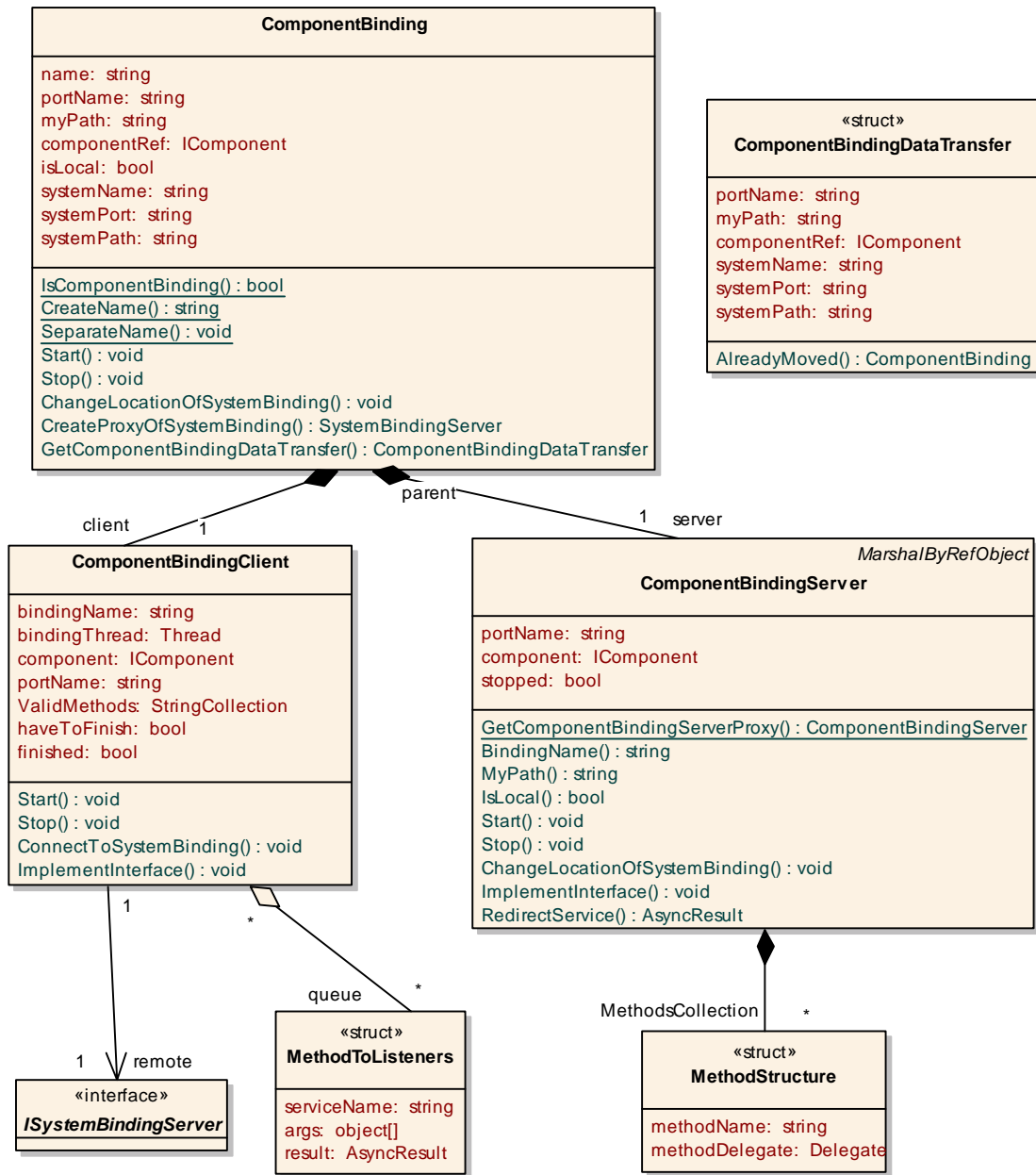


Figura 39 - Clases asociadas al ComponentBinding

- **isLocal, systemName, systemPort, systemPath:** atributos que definen información relativa al sistema al cual este *componentBinding* pertenece, como el nombre, el puerto y la URL donde se encuentra. El atributo *isLocal* indica si este *componentBinding* es local al *systemBinding*, por lo que no requiere la publicación del objeto *componentBindingServer* como objeto remoto.
- **Client, Server:** atributos que contienen las referencias al módulo servidor (*ComponentBindingServer*) y al módulo cliente (*ComponentBindingClient*).

- **IsComponentBinding()**: método estático que indica si es un *ComponentBinding* o no (en cuyo caso será un attachment), dado un nombre de un *listener* de un puerto de entrada o de salida.
- **CreateName()**, **SeparateName()**: métodos estáticos que definen cómo se construye un nombre identificativo de un *ComponentBinding*. Dicho nombre se construye mediante la concatenación del nombre y puerto del sistema y el nombre y puerto del componente/conector entre los cuales establece la comunicación, de forma similar a los attachments. El método *SeparateName* separa las partes que componen dicha cadena.
- **Start()**, **Stop()**: métodos para iniciar o parar la ejecución del *ComponentBinding* de forma segura, como se ha comentado anteriormente.
- **ChangeLocationOfSystemBinding()**, **CreateProxyOfSystemBinding()**: métodos para actualizar las referencias hacia el *SystemBinding*, o crear el proxy correspondiente para acceder remotamente.
- **GetComponentBindingDataTransfer()**: método que genera una estructura con el estado del *ComponentBinding*, cuyo tipo es *ComponentBindingDataTransfer*, de igual forma que los attachments. Esta estructura es utilizada para mover el *ComponentBinding* a otra máquina, y una vez allí, a través de la invocación del método *AlreadyMoved()* se vuelve a generar el *ComponentBinding*.

La clase agregada *ComponentBindingClient* es muy similar al *AttachmentClient*, pues tiene un hilo de ejecución propio (*bindingThread*) encargado de escuchar las peticiones de servicio del puerto de salida del componente, cuyo nombre se almacena en *portName*. La parada de dicho hilo se controla a través de los atributos privados *haveToFinish* y *isFinished*. El primero es activado cuando se invoca el servicio *Stop()*, y el segundo sólo es activado cuando finalmente el hilo se ha parado de forma segura. Las peticiones generadas por el puerto de salida son almacenadas en una cola, cuya referencia es almacenada en el atributo *queue* al registrarse el *ComponentBindingClient* a dicho puerto. Los servicios válidos que se aceptarán y serán reenviados hacia el *SystemBinding* (cuya referencia es almacenada en el atributo *remote*) son los definidos en la lista *ValidMethods*, obtenidos dinámicamente a partir de la interfaz del puerto de salida del componente (método *ImplementInterface*).

Por otra parte, la clase agregada *ComponentBindingServer* acepta las peticiones enviadas desde el *SystemBinding* y las redirecciona al componente al cual está asociada. Hereda de *MarshalByRefObject* para que pueda ser accedida remotamente si se encuentra en una máquina distinta a la del *SystemBinding*. Las peticiones de servicio son recibidas mediante la invocación remota del método *RedirectService*, proporcionando el nombre del servicio a ejecutar y un vector con sus parámetros. Tras obtener de la lista

MethodsCollection el delegado asociado al nombre del servicio, éste es invocado con los parámetros proporcionados. Esta lista de delegados es construida por el método *ImplementInterface* a partir de la interfaz del puerto de entrada del componente, como se ha comentado anteriormente.

4.2.5.4 Definición de Sistemas en .NET

Una vez visto el funcionamiento de los attachments y de los bindings, y los mecanismos de comunicación que proporcionan, en esta sección se muestra cómo éstos son definidos en un sistema para establecer las comunicaciones entre los distintos elementos arquitectónicos que lo componen. De igual forma, también se presentará cómo los componentes y conectores son agregados al sistema.

Los sistemas también utilizan la herencia para implementar los tipos PRISMA, de la misma forma que los componentes. En la clase *SystemBase* se define el comportamiento genérico de los sistemas, mientras que el comportamiento específico es generado en tiempo de compilación para cada tipo PRISMA definido. Para ello, se tomará como ejemplo la especificación del sistema *BankSystem*, mostrada en el apartado 3.4.1.4:

```
System_type BankSystem
  Ports
    CreditCard_port : ICreditCardTransactions;
  End_Ports

  Variables
    Var_ATM : ATM;
    Var_Account : Account;
    Var_ATMAccount : ATMAccount;
  End_Variables

  Attachments
    Var_ATM.AccountTrans_port ↔ Var_ATMAccount.ATM_role;
    Var_Account.Account_port ↔ Var_ATMAccount.Account_role;
  End_Attachments;

  Bindings
    Var_ATM.VISACreditCard_port ↔ BankSystem.CreditCard_port;
  End_Bindings;

  Initialize
    New() {
      Var_ATM = new ATM(accountId: decimal, initialLocation: LOC);
      Var_Account=new Account(accountId:decimal,initialLocation:LOC);
      Var_ATMAccount = new ATMAccount(initialLocation: LOC);
    }
  End_Initialize;

  Destruction
    Destroy() {
      Var_ATM.destroy();
      Var_Account.destroy();
      Var_ATMAccount.destroy();
    }
  End_Destruction;
```

```
End_System_type BankSystem;
```

A continuación se muestra el código generado para definir dicho sistema:

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace CuentaBancaria {

    // Definición del Sistema BankSystem
    [Serializable]
    public class BankSystem: SystemBase {

        public BankSystem (string name, MiddlewareSystem middlewareSystem,
            <Atributos_requeridos_por_los_componentes_internos>
            decimal ATM_Id, LOC ATM_initialLocation,
            decimal Account_Id, LOC Account_initialLocation,
            LOC ATMAccount_initialLocation
            <\Atributos_requeridos_por_los_componentes_internos>
            ) : base(name, middlewareSystem) {

            <Definicion_Aspectos>
            // No se han definido en el ejemplo
            <\Definicion_Aspectos>

            <Definicion_Weavings>
            // No se han definido en el ejemplo
            <\Definicion_Weavings>

            <Definicion_Puertos>
            // Creación de PUERTOS de ENTRADA y SALIDA
            InPorts.Add("CreditCardPort", "ICreditCardTransactions");
            OutPorts.Add("CreditCardPort", "ICreditCardTransactions", "");
            <\Definicion_Puertos>

            <Definicion_Componentes>
            // Var_ATM
            AddComponent("Var_ATM", typeof(CuentaBancaria.ATM),
                ATM_initialLocation, ATM_Id);

            // Var_Account
            AddComponent("Var_Account", typeof(CuentaBancaria.Account),
                Account_initialLocation, Account_Id);
            <\Definicion_Componentes>

            <Definicion_Conectores>
            // Var_ATMAccount
            AddConnector("Var_ATMAccount", typeof(CuentaBancaria.ATMAccount),
                ATMAccount_initialLocation);
            <\Definicion_Conectores>

            <Definicion_Attachments>
            AddAttachment("Var_ATM", "AccountTrans_port", "Var_ATMAccount",
                "ATM_role");

            AddAttachment("Var_Account", "Account_port", "Var_ATMAccount",
                "Account_role");
            <\Definicion_Attachments>

            <Definicion_Bindings>
            AddBinding("CreditCard_port", "Var_ATM", "VISACreditCard_port");
            <\Definicion_Bindings>
        }
    }
}
```

```
}  
  }  
}
```

Un sistema PRISMA se implementa a través de una clase que hereda su comportamiento de la clase *SystemBase*, y es *Serializable* para poder ser distribuida a otra máquina. De la misma forma que el componente, la estructura de un sistema se define en el cuerpo del constructor y también se encuentra separada en distintas secciones. Algunas de ellas son comunes a los componentes, como la definición de aspectos, weavings y puertos. En cambio, las que caracterizan a un sistema son las que permiten definir componentes, conectores, attachments y bindings. Como se puede observar, la conversión desde la especificación PRISMA a la implementación es casi directa.

Para agregar un componente o conector se invoca el método *AddComponent* o *AddConnector* proporcionando el nombre del componente o conector, el tipo, la localización inicial y los valores de inicialización adicionales requeridos por cada tipo. Puede observarse en el ejemplo que la localización inicial es proporcionada a través del tipo LOC, estructura creada para identificar la localización de un componente distribuido. Contiene un atributo de localización e incorpora métodos para comprobar que dicho valor es una URI bien definida. El *middleware* se encargará de ubicar en la localización especificada a los componentes/conectores creados.

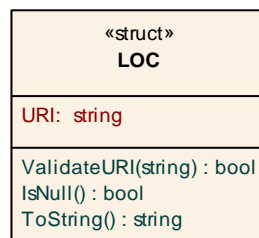


Figura 40 - Estructura LOC

Del mismo modo, la definición de los attachments se realiza especificando el nombre del componente y del puerto, junto al nombre del conector y del rol. La definición de los bindings se realiza especificando el puerto del sistema junto al nombre del componente/conector y del puerto/rol.

4.2.6 MiddlewareSystem

La clase *MiddlewareSystem* implementa la funcionalidad más dependiente de la plataforma de desarrollo, y proporciona distintos servicios a los elementos arquitectónicos PRISMA.

- Gestiona la creación y destrucción de objetos
- Permite establecer la comunicación distribuida, por lo que es el encargado de la creación de los attachments y los bindings.

- Proporciona los servicios de movilidad y de transferencia de componentes, conectores y sistemas entre los diferentes *middleware*.
- Proporciona servicios de localización de los distintos componentes, es decir, es el encargado de realizar las correspondientes búsquedas DNS entre los diferentes *middleware*
- Establece las comunicaciones necesarias con otros *middleware* para propagar las solicitudes de servicio que no puede atender.
- Se encarga de la búsqueda de los diferentes tipos solicitados o requeridos por los elementos arquitectónicos, bien cargando los ensamblados desde la máquina local o solicitándolos a otros *middleware*.
- Implementa un servicio de suscripción para enviar los mensajes de depuración a los suscriptores.

Además, también proporcionará una serie de servicios, de cara a soportar la evolución, para propagar los cambios a los distintos *middleware*, tanto de los elementos arquitectónicos en ejecución, como de los tipos (ensamblados) de los otros *middleware*, aunque queda propuesto como tarea futura.

En la Figura 41 se muestra el diagrama de despliegue del modelo de ejecución PRISMA, soportado por varios *middlewares* ubicados en distintos nodos. Cada uno de los *middlewares* tendrá una instancia en ejecución de la clase *middlewareSystem*.

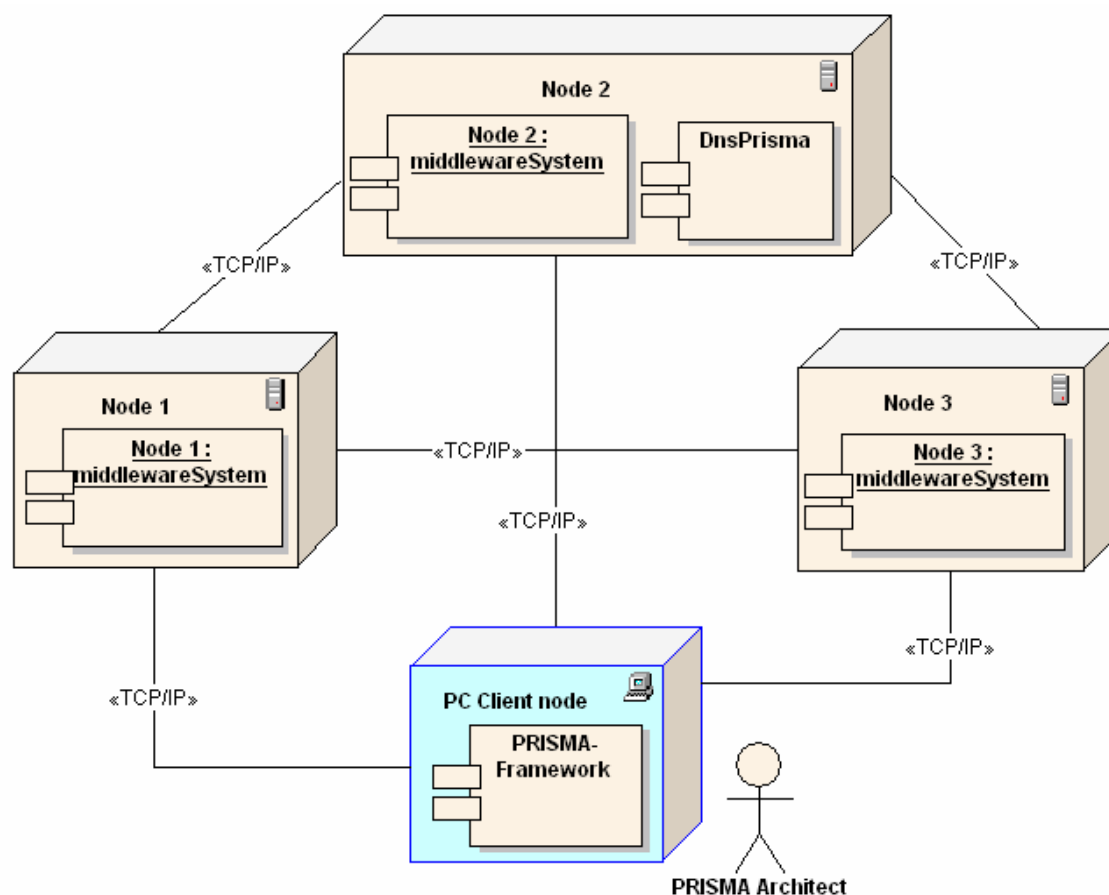


Figura 41 - Diagrama de despliegue del middleware

El nodo *PC Client* contendrá el framework de PRISMA y desde él se podrán controlar de forma centralizada los diferentes *middlewares*. Su cometido es el de administrar los modelos PRISMA en ejecución, cargarlos o detenerlos, así como reconfigurar sus componentes o hacerlos evolucionar, etc. Este nodo también es el que recibe los mensajes de depuración de los diferentes *middlewares*, con lo que puede controlar su estado, a través del mecanismo de suscripción que se comentará a continuación.

La clase *DnsPrisma* modela un DNS centralizado. Esta clase es publicada para su acceso remoto en una ubicación conocida y los distintos *middleware* se conectan a ella para añadir, borrar o consultar ubicaciones de componentes, conectores o sistemas. Sin embargo, en un futuro esta clase se implementará de forma descentralizada, manteniendo su interfaz intacta, de acuerdo con el modelo PRISMA. Cada *middleware* tendrá una instancia local de esta clase a la que solicitará servicios de resolución. Cada instancia se encargará de resolver la ubicación solicitada comunicándose con otras instancias remotas, consultando las localizaciones proporcionadas por los attachments y bindings.

En la Figura 42 se muestra el diagrama de clases correspondientes al diseño de la clase *MiddlewareSystem*. Se han omitido gran parte de los métodos privados y aquellos exclusivos de la implementación, con la finalidad de que el diagrama sea claro.

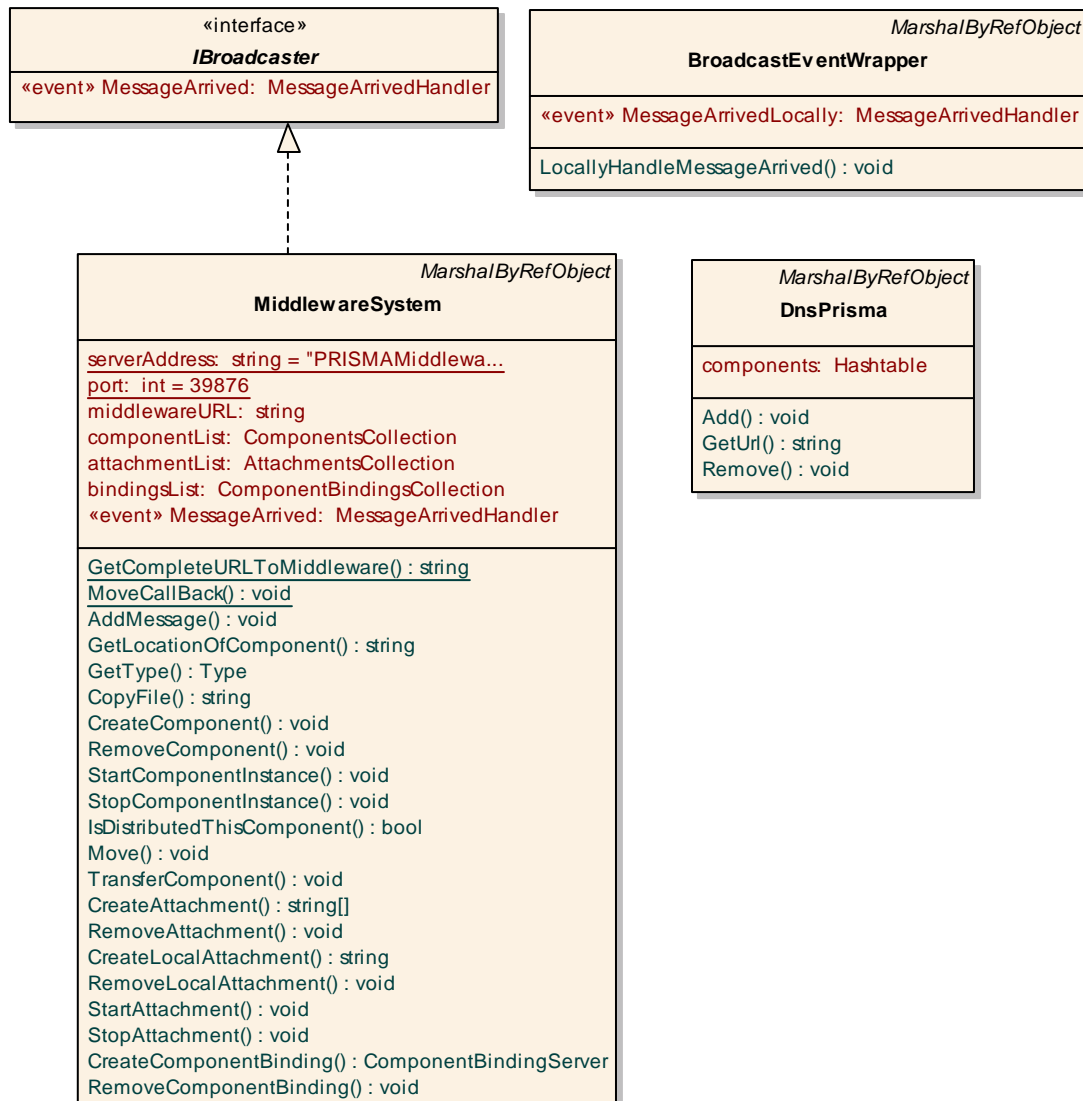


Figura 42 - Clase MiddlewareSystem

La interfaz *IBroadcaster* y la clase *BroadcastEventWrapper* permiten implementar el mecanismo de suscripción de eventos del *middleware*. Esto se ha realizado con la finalidad de que un único nodo pueda administrar la información de todos los *middleware* activos. La interfaz *IBroadcaster* define el evento que deben implementar los *middleware* para que se suscriban los posibles clientes. El *middleware* envía los mensajes de depuración a los suscriptores del evento *MessageArrived*, y para soportar el paso de eventos en entornos distribuidos, Remoting requiere de una clase auxiliar accesible remotamente (*BroadcastEventWrapper*) que reenvíe la petición a los suscriptores. De esta forma, la aplicación de administración de PRISMA (*Prisma-Framework*) cuando es creada se registra en todos los *middleware* PRISMA, con lo que recibe todos los mensajes de depuración para que el arquitecto de modelos PRISMA (o el administrador) realicen las operaciones pertinentes.

La clase *MiddlewareSystem* define como atributos estáticos los parámetros para conectarse a ella. Éstos son el puerto de la máquina en la que se registrará para su acceso remoto (*port*, por defecto 39876) y el nombre del servidor (*serverAddress*) por el que será accedida, que por defecto es “PRISMAMiddleware.soap”. Se ha implementado el método estático *GetCompleteURLToMiddleware* que, dada la URL del *middleware* al que se desea conectar, a partir de los parámetros anteriores construye la cadena de conexión necesaria para acceder remotamente a ella. Internamente, la clase almacena la URL donde se encuentra ubicada en el atributo *middlewareURL*.

La clase contiene tres listas: *componentList*, *attachmentList* y *bindingsList*. En la lista *componentList* se almacenan las referencias a todos los componentes, conectores y sistemas que son gestionados por este *middleware*. Del mismo modo, en *attachmentList* y *bindingsList* se almacenan las referencias correspondientes a los attachments y bindings que residen en este *middleware*. Los componentes, conectores y sistemas cuando son creados, son registrados en el DNS, para que otros *middleware* conozcan de su existencia. En cambio, los attachments y bindings no son registrados, ya que son entidades auxiliares para establecer la comunicación y únicamente deben conocerlos los *middleware* implicados. Sin embargo, a partir del nombre de un attachment o binding puede extraerse el nombre de los componentes que conecta y, como éstos si que están registrados en el DNS, puede obtenerse la localización de los attachments asociados.

A continuación se describen brevemente los servicios que proporciona el *middleware*, agrupados por la funcionalidad ofrecida. Los métodos ofrecidos por el *middleware* pueden ser invocados indistintamente desde cualquier máquina. Será tarea del *middleware* que ha recibido la petición el averiguar a qué *middleware* concreto se le debe redirigir dicha petición. Por ejemplo, puede invocarse desde un “nodo A” la creación de un attachment entre los componentes que residen en el “nodo B” y el “nodo C”. El *middleware* se encargará de reenviar las peticiones adecuadas a cada nodo para realizar la tarea solicitada.

El método *AddMessage(string message)* es invocado por los tipos PRISMA en ejecución para registrar cualquier evento a considerar, como por ejemplo, la creación de componentes, la inserción de aspectos, etc. La invocación de este método propaga el mensaje a los suscriptores del evento *messageArrived*.

El método *GetLocationOfComponent(string componentName)* devuelve la ubicación del componente, conector o sistema a partir del nombre proporcionado. Para ello, propaga la petición al DNS, el cual se encarga de obtener el resultado.

Por su parte, el método *GetType(string typeName)* se encarga de la búsqueda del tipo cuyo nombre se proporciona como parámetro. Por ejemplo, este método es invocado en la creación dinámica de puertos, attachments o bindings, ya que requieren un tipo específico a la interfaz para la que se

crean. Para ello, en primer lugar busca el tipo en los ensamblados locales, almacenados en la librería de tipos PRISMA que debe residir en una subcarpeta del directorio de trabajo del *middleware*. En segundo lugar, si no lo encuentra, va interrogando a los diferentes *middleware*, hasta encontrar uno que sí lo tenga. Es entonces cuando se invoca el método *CopyFile* para iniciar la transferencia del ensamblado que contiene el tipo hacia el *middleware* que lo ha solicitado.

La gestión de los componentes es realizada a través de varios servicios. El método *CreateComponent(Type componentType, object[] args, string URL)* es el encargado de crear un componente, conector o sistema. Para ello, se le debe proporcionar el tipo de componente a crear, los argumentos requeridos por el constructor y la URL donde se ubicará el componente. Si la localización se refiere a otra máquina distinta, la solicitud de creación es reenviada al *middleware* ubicado en dicha localización. En segundo lugar, el componente es creado a partir del tipo, registrado en el DNS y añadido a la lista interna de componentes. El método *RemoveComponent* realiza el proceso contrario. Sin embargo, un componente no inicia su funcionamiento hasta que es invocado el método *StartComponentInstance(string name)*. Este método crea un nuevo hilo para el componente y lo pone en ejecución, mediante la invocación del método *Start()*. Del mismo modo, la invocación del método *StopComponentInstance(string name)* detiene la ejecución del componente cuyo nombre se indica.

La gestión de los attachments es realizada por otro conjunto de servicios. La creación de un attachment se solicita mediante la invocación del método *CreateAttachment*, al que se le proporciona el nombre del componente y conector a conectar, así como el nombre del puerto y rol. Este método invoca al método *CreateLocalAttachment* en cada *middleware* donde residen los componentes a conectar, con la finalidad de que creen los dos attachments correspondientes. La eliminación de attachments se realiza de un modo similar mediante la invocación del método *RemoveAttachment*, que a su vez invoca los métodos *RemoveLocalAttachment* en los *middlewares* donde resida cada parte del attachment. Los attachments creados son puestos en ejecución a través de la invocación del método *StartAttachment(string name)* y son detenidos por el método *StopAttachment(string name)*.

Los bindings son creados a través de la clase *SystemBinding*, descrita en el apartado 4.2.5.3. Sin embargo, la creación remota de los *ComponentBindings* debe ser realizada por el *middleware*. Este método se limita a crear el *componentBinding* cuyos datos se proporcionan como parámetros y añadirlo a la lista interna de bindings, devolviendo un proxy hacia su parte servidora, para que pueda ser accedido remotamente. Por su parte, el método *RemoveComponentBinding* lo elimina y lo borra de la lista interna.

Los servicios de movilidad son proporcionados por los métodos *Move*, *MoveCallBack* y *TransferComponent*. El método *Move(LOC newLocation*,

string componentName) es invocado para mover un componente, conector o sistema en ejecución a otra máquina, sin perder el estado, y continuar su ejecución en la nueva ubicación. La ejecución de este método implica una serie de pasos antes de poder transferir al componente hasta su nueva ubicación, los cuales se describen aquí de forma general:

1. Se obtienen todos los attachments y bindings que están conectados a los puertos del componente a mover, para posteriormente procesarlos.
2. Se invoca el método *Stop* del componente para que realice una parada segura, lo que implica parar recursivamente también los aspectos que lo componen.
3. Una vez el componente ha sido detenido, son detenidos también los attachments y bindings a él conectados.
4. Se procesan los attachments y bindings locales con dos motivos: los que estaban publicados para su acceso remoto, son desregistrados, pues van a moverse a otra máquina. Por otra parte, si el attachment o binding que se va a mover era local (es decir, sus dos partes estaban en la misma máquina), es necesario publicar para acceso remoto la parte del attachment/binding que no se mueve, con la finalidad de que posteriormente pueda comunicarse con la parte remota (la que va a ser movida).
5. Se borra del DNS al componente que va a ser movido
6. Se transfiere al *middleware* remoto el componente a mover, junto a la lista de attachments y bindings asociados, mediante la invocación del método *TransferComponent*.
7. Se elimina el componente, los attachments y los bindings del *middleware* local que han sido transferidos a la nueva ubicación.
8. Si ocurre algún error durante el proceso, todas las acciones son deshechas.

El método *TransferComponent* también forma parte del proceso de movilidad, y realiza las operaciones necesarias en el *middleware* destino para poner en ejecución todos los elementos transferidos. En primer lugar deserializa el componente recibido y lo pone en ejecución. En segundo lugar deserializa los attachments, reestablece las comunicaciones entre los attachments recibidos y los remotos, y finalmente los pone en ejecución. Con los bindings recibidos se procede de igual manera. De esta forma, se consigue mover un componente en ejecución desde una máquina a otra y restablecer las comunicaciones, gracias a la parada segura proporcionada por los elementos encargados de las comunicaciones.

El método *MoveCallBack* se proporciona para permitir la llamada al método *Move* de forma asíncrona. Esto es necesario para aquellos casos en que un componente indique al *middleware* que desea moverse a petición propia (por ejemplo, como ocurre con el aspecto de distribución). Si la llamada no fuese asíncrona, cuando el componente invocase el servicio de movimiento, se produciría un interbloqueo: el método *Move* no puede continuar su proceso

hasta que el componente a mover sea parado, pero dicho componente no puede pararse hasta que el método que él ha invocado (*Move*) finalice su ejecución, por lo que se produce un ciclo del que no se finalizará nunca.

4.3 Resumen del Capítulo

Finalmente, a modo de resumen, se muestran las correspondencias entre los diferentes tipos PRISMA y las clases implementadas en C#:

MODELO PRISMA	IMPLEMENTACIÓN C#
<i>Interfaz</i>	Interfaz
<i>Aspectos</i>	Clase Aspecto (<i>AspectBase</i>)
<i>Componentes</i>	Clase Componente (<i>ComponentBase</i>)
<i>Conectores</i>	Clase Componente e interfaz específica (<i>IConnector</i>)
<i>Attachments</i>	2 instancias de la clase Attachment (<i>AttachmentBase</i>)
<i>Binding</i>	2 clases: una para el sistema (<i>SystemBinding</i>) y otra para el componente (<i>ComponentBinding</i>)
<i>Sistema</i>	Clase Sistema (<i>SystemBase</i>)

Tabla 7 - Resumen correspondencias PRISMA y la implementación C#

ASPECTO

MODELO PRISMA	IMPLEMENTACIÓN C #
<i>Tipo</i>	Representado a través de la herencia de una clase que representa al tipo deseado. Esta clase a su vez hereda el comportamiento básico del aspecto.
<i>Nombre</i>	Variable <i>aspectName</i>
<i>Aspecto</i>	Clase base (<i>AspectBase</i>) con lista de <i>played_roles</i> y un hilo de ejecución para procesar las peticiones de servicio. Los atributos y servicios no son contemplados, son implementados por el compilador.
<i>Atributos</i>	Variables
<i>Servicios</i>	Dos métodos, uno externo para encolar la petición de servicio y devolver el control al componente y otro interno para ejecutar el servicio solicitado.
<i>Valuaciones</i>	Implementación del método interno al que está asociada la valuación.
<i>Precondiciones</i>	Comprobadas a través de una condición dentro del cuerpo del método interno, previa a la ejecución de la valuación.
<i>Played_Roles</i>	Clase <i>SubProcess</i> , que contiene el nombre del <i>played_role</i> y el conjunto de métodos y sus propiedades.
<i>Protocolo</i>	El conjunto de estados se almacena en una enumeración. Es comprobado en el cuerpo del método interno antes que las precondiciones. El nuevo estado es asignado al final del método si la ejecución ha sido correcta.

Tabla 8 - Correspondencias Aspecto PRISMA y la implementación C#

COMPONENTE

MODELO PRISMA	IMPLEMENTACIÓN C #
<i>Nombre</i>	<i>Variable Name</i>
<i>Interfaz</i>	Conjunto de servicios que invoca el middleware para la gestión de componentes (<i>IComponent</i>)
<i>Componente</i>	Clase base con referencias a aspectos, puertos, weavings, un thread propio para la gestión de servicios, y una cola de peticiones de servicios. (<i>ComponentBase</i>)
<i>Puertos</i>	Colección de puertos del componente. Se divide en dos tipos: - Cliente (<i>OutPort</i>): deja las peticiones en una cola escuchada por el attachment - Servidor (<i>InPort</i>): redirecciona las peticiones a la cola del componente.
<i>Aspectos</i>	Lista dinámica de aspectos (<i>aspectList</i>)
<i>Weavings</i>	Clase <i>Weaving Collection</i> = Lista de Weavings asociada a una componente. Clase <i>Weaving Node</i> = Lista de weavings asociados al servicio que desencadena el weaving. Clase <i>Weaving Method</i> = Clase que contiene el servicio no desencadenante del weaving y las funciones de transformación necesarias para los parámetros del weaving. Clase <i>Weaving Type</i> = Tipos de weavings. Delegados hacia funciones que permiten la transformación de parámetros.

Tabla 9 - Correspondencias Componente PRISMA y la implementación C#

CONECTOR

MODELO PRISMA	IMPLEMENTACIÓN C #
<i>Nombre</i>	<i>Variable Name</i>
<i>Interfaz</i>	Conjunto de servicios que invoca el middleware para la gestión de conectores (<i>IConnector</i> , subclase de <i>IComponent</i>)
<i>Conector</i>	Clase <i>ComponentBase</i> que se identifica como conector por el hecho de implementar la interfaz del conector.
<i>Roles</i>	Colección de puertos (como el componente)
<i>Aspectos</i>	Colección de aspectos (como el componente)
<i>Weavings</i>	Colección de weavings (como el componente)

Tabla 10 - Correspondencias Conector PRISMA y la implementación C#

SISTEMA

MODELO PRISMA	IMPLEMENTACIÓN C #
<i>Nombre</i>	Variable <i>Name</i>
<i>Interfaz</i>	Conjunto de servicios que invoca el middleware para la gestión de sistemas (<i>ISystem</i> , subclase de <i>IComponent</i>)
<i>Sistema</i>	Clase <i>SystemBase</i> , subclase del componente, que contiene la lista de componentes, conectores y attachments que contiene, y el conjunto de bindings.

Tabla 11 - Correspondencias Sistema PRISMA y la implementación C#

ATTACHMENT

MODELO PRISMA	IMPLEMENTACIÓN C #
<i>Nombre</i>	Variable <i>attachmentName</i>
<i>Attachment</i>	Dos clases <i>Attachment</i> asociadas a cada extremo del canal de comunicación. Cada clase está formada por dos clases agregadas, una cliente y otra servidora para procesar por canales diferentes las peticiones y las recepciones.

Tabla 12 - Correspondencias Attachment PRISMA y la implementación C#

BINDING

MODELO PRISMA	IMPLEMENTACIÓN C #
<i>Nombre</i>	Variable <i>bindingName</i>
<i>Binding</i>	<p>Dos clases específicas, una para el sistema (<i>SystemBinding</i>) y otra para el componente o conector que lo forma (<i>ComponentBinding</i>).</p> <p>La clase <i>SystemBinding</i> actúa como servidor remoto para aquellos componentes o conectores que están en otra máquina. Gestiona las redirecciones desde los puertos del sistema hacia los componentes, tanto en un sentido como en el otro.</p> <p>La clase <i>ComponentBinding</i> implementa al conector o al componente y está compuesta por dos clases agregadas, una cliente y otra servidora, para procesar por canales diferentes las peticiones y las recepciones.</p>

Tabla 13 - Correspondencias Binding PRISMA y la implementación C#

CONCLUSIONES

Contenidos del capítulo:

5.1 Conclusiones	153
5.2 Trabajos futuros	154

CONCLUSIONES

5.1 Conclusiones

En este trabajo se ha realizado la implementación de un modelo orientado a aspectos y basado en componentes como PRISMA utilizando la tecnología .NET.

Para ello, se ha realizado un análisis preliminar de las diferentes aproximaciones existentes orientadas a aspectos, del que se han extraído las principales tendencias de desarrollo, así como las carencias de los prototipos existentes. Todas ellas tienen en común la separación clara del código de aplicación respecto del código de los aspectos, y la definición de los *weavings* de forma independiente al código a unir. Sin embargo, pocas de ellas aportan mecanismos para añadir o eliminar aspectos dinámicamente. Las que sí lo tienen en cuenta, lo hacen a costa de perder la reutilización, bien de aspectos, o bien del código de la aplicación. Además, ninguna de ellas consigue fusionar adecuadamente la programación orientada a aspectos (AOP) con el diseño basado en componentes (CBSD).

En cambio, el modelo PRISMA combina la reutilización proporcionada por AOP y la reconfiguración dinámica de los modelos CBSD. Los aspectos se definen de forma separada a los *weavings* y son altamente reutilizables. Los componentes se forman por la agregación dinámica e inclusiva de aspectos y pueden ser movidos a otras máquinas en tiempo de ejecución.

La propuesta de implementación realizada se ha materializado en un prototipo desarrollado con la tecnología .NET, del que se han identificado las plantillas de código que deberán ser utilizadas por el compilador para generar código C# a partir de especificaciones PRISMA. Para el diseño del prototipo se han intentado alcanzar los siguientes objetivos, que caracterizan al modelo PRISMA:

- Alta reutilización de los elementos arquitectónicos
- Ejecución concurrente de los distintos elementos
- Soporte a la comunicación distribuida
- Movilidad autónoma (por iniciativa propia) o inducida (a petición de otro elemento), de componentes, conectores y sistemas

- Reconfiguración dinámica de los modelos arquitectónicos para ajustarse a otros entornos
- Evolución dinámica del modelo

Algunos de los objetivos anteriores se han alcanzado completamente, como la reutilización, la ejecución concurrente, la comunicación distribuida y la reconfiguración dinámica. Otros objetivos se han alcanzado parcialmente, pero han sentado las bases de futuras versiones del prototipo desarrollado. Es el caso de la movilidad y de la evolución dinámica. Los componentes pueden ser movidos a otras máquinas, por iniciativa propia o a petición de otro elemento. Sin embargo, aún quedan algunos problemas por resolver para situaciones concretas, como el movimiento cuando la ejecución de un servicio no puede ser interrumpida. La evolución dinámica no ha sido implementada en su totalidad, pero se han definido los mecanismos que permiten modificar en tiempo de ejecución el comportamiento de cada elemento arquitectónico PRISMA, aunque sólo individualmente. Por ejemplo, el componente soporta la modificación en tiempo de ejecución de su estructura interna (puertos, aspectos y weavings), sin necesidad de detener sus operaciones por completo y recompilarse. Sin embargo, los cambios sólo afectan a la instancia a la que se le han solicitado. Para que el prototipo soporte la evolución de forma completa aún tienen que ser introducidos una serie de mecanismos adicionales que permitan la coordinación entre el tipo y sus diferentes instancias y la propagación de los cambios a los distintos *middlewares*.

Además, el prototipo desarrollado ha sido validado con la implementación de dos casos de estudio: la cuenta bancaria y el robot 4U4. El caso de estudio de la cuenta bancaria, cuya especificación se ha mostrado en el capítulo 0 para ilustrar los conceptos del modelo PRISMA, ha permitido validar la implementación de la movilidad y la comunicación distribuida. Por otra parte, el caso de estudio del robot 4U4 [Per04], cuya especificación modela un brazo robótico de ámbito industrial, ha permitido validar la implementación de arquitecturas complejas, concurrentes y altamente reutilizables.

5.2 Trabajos futuros

La plataforma de ejecución de modelos PRISMA aún se encuentra en una fase temprana y no se puede dar el trabajo realizado por concluido. Todavía quedan tareas pendientes por realizar y muchos aspectos que no han sido solucionados en esta primera versión. A continuación se citan los más importantes.

Uno de los trabajos más importantes que quedan por realizar es el desarrollo de un compilador que transforme las especificaciones PRISMA a .NET, cuyas plantillas de código se han obtenido como resultado de este

trabajo. El uso de dicho compilador permitirá ejecutar de forma transparente para el usuario los modelos PRISMA introducidos, así como realizar los cambios en la especificación del modelo, sin hacer uso de la evolución dinámica.

Otro aspecto a considerar, tal y como se ha comentado anteriormente, es que ha de dotarse al prototipo de los mecanismos necesarios para soportar la evolución dinámica a nivel global. Dichos mecanismos deberán permitir:

- Propagar los cambios a todas las instancias de un tipo en la máquina local.
- Almacenar los cambios para que futuras instancias del tipo los incorporen.
- Propagar el tipo modificado a los demás *middlewares* que lo utilicen, además de propagar los cambios a las instancias en ejecución de otras máquinas.

Por otra parte, también es necesario introducir mecanismos para mejorar la parada segura de componentes, conectores y sistemas, de cara a la movilidad. En la implementación actual, existen una serie de situaciones en las que la movilidad no es posible, a menos que finalicen o sean abortadas. Dichas situaciones se producen cuando un componente se encuentra ejecutando un servicio de larga duración, que no puede detenerse, o un servicio que implica la invocación de otros servicios externos.

Por último, también queda como tarea pendiente la implementación del DNS de forma descentralizada, como establece el modelo PRISMA.

BIBLIOGRAFÍA

- [Ali03] Ali N.H., Silva J., Jaen J., Ramos I., Carsi J.A., Perez J. *Mobility and Replicability Patterns in Aspect-oriented Component-Based Software Architectures*. In Proc. of 15th IASTED, Parallel and Distributed Systems, Acta Press, ISBN: 0-88986-392-X, ISSN: 1027-2658, pp. 820-826. Marina del Rey, C.A., USA, Noviembre 2003.
- [Aosd] *Aspect-oriented software development*. En: <http://aosd.net>
- [Ar02] Archer T., *Inside C#, 2nd Edition*. Microsoft Press, ISBN: 0-7356-1648-5, Redmond, Washington, 2002.
- [AsJ03] The AspectJ Team. *AspectJ Programming Guide*, 2003.
- [AsJ04] AspectJ Project, en <http://eclipse.org/aspectj/>
- [Bal85] Balzer R., *A 15 Year Perspective on Automatic Programming*. IEEE Transactions on Software Engineering, vol.11, num.11, págs. 1257-1268, Noviembre 1985.
- [Dng04] *AspectDNG Project*. En: <http://aspectdng.sourceforge.net/>
- [Gun04] Gunnerson E., *Calling Code Dynamically*. Microsoft Corp., Feb. 2004. En: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp02172004.asp>
- [Jack04] Jackson A., Clarke S., *SourceWeave.NET: Cross-Language Aspect-Oriented Programming*. In Proc. of Generative Programming and Component Engineering (GPCE 2004). Vancouver, Canada, 2004.
- [Kic97] Kiczales, G., et al., *Aspect-Oriented Programming*. In European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland, 1997, Springer-Verlag, pp.220-242.

- [Kim02] Kim H., *AspectC#: an AOSD implementation for C#*. M. Sc. Thesis, Comp. Sci., Trinity College, Dublin, Noviembre 2002.
- [Mil91] Milner, R., *π -Cálculo Poliádico: A tutorial*, 1991.
- [Mono] *The Mono Project*. En: <http://www.mono-project.com>
- [Laff03] Lafferty D., Cahill V., *Language-Independent Aspect-Oriented Programming*. In Proc. of Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003). Anaheim, California, USA, 2003.
- [Lam02] Lam J., *Cross Language Aspect Weaving*. Demonstration, AOSD 2002, Enschede, 2002.
- [Let98] Letelier P., Sánchez P., Ramos I., Pastor O., *OASIS 3.0: "Un enfoque formal para el modelado conceptual orientado a objeto"*. Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721- 663-0, 1998.
- [Ossh00] Ossher H., Tarr P., *Multi-Dimensional Separation of Concerns and The Hyperspace approach*. In Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.
- [Pas97] Pastor O. Et al., *OO-METHOD: A software production environment combining conventional and formal methods*. In proc. of 9th International Conference, CaiSE97, Barcelona, 1997.
- [Per03] Pérez, J., Ramos, I., *Oasis como Soporte Formal para la Definición de Modelos Hipermedia Dinámicos, Distribuidos y Evolutivos*. Informe Técnico DSIC-II/22/03, Universidad Politécnica de Valencia, octubre 2003.
- [Per04] Pérez, J., Cabedo R., et al., *Arquitectura PRISMA para el caso de estudio: Brazo Robot*. Actas del II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pags. 119-127, junto a Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Málaga, noviembre 2004.

- [Ramm02] Rammer, I., *Advanced .NET Remoting (C# Edition)*. Apress, 2002.
- [Raj03] Rajan H., Sullivan K., *EOS: Instance-Level Aspects for Integrated System Design*. In Proc. of Joint European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). Helsinki, Finland, Septiembre 2003.
- [RRose] Rational Software, *Rational Rose*, <http://www.rational.com/products/rose/>
- [Sch00] Schult W., Polze A., *Aspect-Oriented Programming with C# and .Net*. In Proc. of International Symposium on Object-oriented Real-time distributed Computing (ISORC) 2002, pp. 241-248, Crystal City, VA, USA, Mayo 2000.
- [Sch02] Schult W., Polze A., *Dynamic Aspect-Weaving with .NET*. Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen, TU Berlin, Germany, Noviembre 2002.
- [Schm02] Schmied, F. *AOP with .NET*. En: <http://wwwse.fhs-hagenberg.ac.at/se/berufspraktika/2002/se99047/contents/>
- [Schü02] Schüpany, M., Schwanninger, C., Wuchner, E., *Aspect-Oriented Programming for .NET*. In First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, (Enschede, The Netherlands, 2002), pp.59-64.
- [Scott03] McLean S., Naftel J., Williams K., *Microsoft .NET Remoting*. Microsoft Press, 2003.
- [SetP04] SetPoint! Project, en: <http://www.dc.uba.ar/people/proyinv/setpoint/>
- [Ser94] Sernadas A., Costa J.F., Sernadas C., *Object specifications trough diagrams: OBLOG approach*. INESC Lisbon, 1994.
- [Sparx] Sparx Systems, *Enterprise Architect*. En: <http://www.sparxsystems.com/>

- [Suv03] Suvee, D., Vanderperren, W. and Jonckers, V., *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development*. In Proc. of international conference on aspect-oriented software development (AOSD), Boston, USA, pp 21-29, ISBN 1-58113-660-9, ACM Press, Marzo 2003.
- [Vers03] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V., *JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services*. In Proc. of Second Nordic Conference on Web Services NCWS'03, Vaxjo, Sweden. Publicado en "Mathematical modelling in Physics, Engineering and Cognitive Sciences", Vol. 8, Noviembre 2003.

LA PLATAFORMA DE DESARROLLO .NET

La plataforma de desarrollo .NET es la apuesta de Microsoft para el desarrollo, el despliegue y la ejecución de la nueva generación de software y servicios Web. Está concebida para proporcionar independencia de la plataforma de desarrollo y para facilitar el desarrollo de aplicaciones distribuidas e interoperables, todo ello a través de un entorno de desarrollo multilinguaje que facilite la combinación de módulos desarrollados en distintos lenguajes (o la mera reutilización de código). Inicialmente los lenguajes soportados por .NET eran *C#*, *J#*, *C++* y *Visual Basic* pero, debido al soporte multilinguaje que ofrece, se han ido añadiendo nuevos lenguajes para aumentar el atractivo hacia desarrolladores diversos. Entre los nuevos lenguajes soportados se encuentran *Cobol*, *RPG*, *APL*, *Perl*, *Pascal*, *SmallTalk*, *Eiffel*, *Fortran*, *Haskell*, *Mercury*, *Oberon*, *Oz*, *Python*, *Scheme*, *Standard ML*, etc.

La arquitectura de la plataforma ha sido diseñada totalmente desde cero, con el objetivo de evitar los problemas que se venían arrastrando de las antiguas APIs (Application Programming Interfaces) de Windows, que por motivos de compatibilidad hacia atrás se habían ido manteniendo, arrastrando los problemas de diseño iniciales. Se ha simplificado el modelo de programación, y los servicios que antes eran ofrecidos por funciones encapsuladas en DLLs o mediante objetos COM, en .NET se proporcionan mediante un modelo de programación orientado a objetos. Además, para evitar los problemas derivados de la sobreescritura de nuevas versiones de librerías compartidas por varias aplicaciones, se han introducido mecanismos de versionado para que puedan coexistir sin problemas diferentes versiones de una misma librería.

Por otra parte, *.NET Framework* está basado en estándares, a diferencia de lo que venía ocurriendo con las alternativas anteriores de Microsoft. Las especificaciones del CLR (la máquina de ejecución de .NET) y de *C#* se han publicado entre los estándares ECMA y además parte del código fuente y de la biblioteca de clases está disponible públicamente. Aunque el entorno .NET se ha proclamado como independiente del sistema operativo, sólo se encuentra disponible en los sistemas operativos de Microsoft. No obstante, el proyecto MONO [Mono], liderado por Ximian y patrocinado por Novell, está consiguiendo con éxito la portabilidad del *Framework* a las plataformas GNU/Linux, Unix y Mac OS X.

A continuación se describirán las principales características del *Framework*, sin entrar en un alto nivel de detalle, pues el objetivo es dar a conocer algunos conceptos a los cuales se referirá en este proyecto en otros capítulos. Para más detalles puede consultarse [Ar02] que, aunque centrado en el lenguaje C#, muestra en profundidad los conceptos más importantes sobre la arquitectura de .NET.

A.1 Interoperabilidad entre lenguajes: el CTS y los Metadatos

La interoperabilidad entre lenguajes ha sido posible gracias a la definición de un Sistema de Tipos Común (CTS, *Common Type System*) y a la introducción de los metadatos. Mediante el CTS se ha podido establecer un marco común para definir los tipos básicos que todo lenguaje soportado por .NET puede tener. Con los metadatos se ha establecido un mecanismo para que los distintos lenguajes puedan obtener la descripción de los tipos y utilizarlos sin necesidad de recurrir a definiciones externas de las interfaces, como es el caso de CORBA, en el que deben generarse por separado las interfaces que publican sus tipos. De esta forma, clases programadas en diferentes lenguajes pueden combinarse sin problemas en una misma aplicación, favoreciendo por tanto la reutilización de código.

Las características del CTS son:

- especifica cómo se declaran, utilizan y gestionan los tipos en el marco de ejecución,
- proporciona un modelo orientado a objetos que soporta la implementación de muchos lenguajes de programación,
- define las reglas que los lenguajes deben seguir para su interoperabilidad.

El CTS se define en el *Common Language System (CLS)*, que define las reglas que deben cumplir todos los compiladores de .NET para generar código interoperable. Una de estas reglas es que el compilador debe soportar todos los tipos definidos en el CTS, con lo que se asegura que objetos o tipos creados en diferentes lenguajes sean compatibles entre sí. Por otra parte, el CTS define una jerarquía única de objetos, es decir, cualquier tipo definido deriva necesariamente del tipo base *System.Object*, y con ello se garantiza que todos los tipos tendrán una interfaz común derivada de este tipo base y por tanto una funcionalidad básica (los servicios *Equals*, *Finalize*, *GetHashCode* y *Tostring*).

Los metadatos son información binaria que describe los tipos implementados por una aplicación. Todos los compiladores de los diferentes lenguajes soportados por .NET, además de emitir el correspondiente código intermedio, están obligados a emitir los metadatos sobre cada tipo contenido

en los ficheros fuente. De esta forma, los ficheros así generados son autodescriptivos, manteniéndose siempre sincronizado el código con las descripciones de sus tipos y permitiendo que cualquier lenguaje pueda obtener la información de los tipos que contienen. Además, el desarrollador puede crear nuevos metadatos para añadir información adicional o incluso más funcionalidad semántica a los tipos, mediante el uso de los *atributos*, que serán descritos más adelante.

Ensamblados

Estas unidades autodescriptivas que agrupan varios tipos y sus correspondientes metadatos, se denominan *Ensamblados (Assemblies)*. Se definen como los bloques de construcción de las aplicaciones para la plataforma .NET, siendo la unidad fundamental de despliegue, de reutilización y de control de versiones, así como la unidad sobre la que se aplican las políticas de seguridad. Un ensamblado es una colección de tipos y recursos que constituyen una unidad lógica de funcionalidad, proporcionando la información que el CLR necesita sobre las implementaciones de dichos tipos.

A.2 El CLR: La máquina virtual de .NET

De igual forma que Java, la plataforma .NET proporciona un marco de ejecución para las aplicaciones que permite gestionar el código en ejecución, liberando al programador de cuestiones como la gestión de memoria, la recolección de basura, la gestión de excepciones, la gestión de los hilos de ejecución o los componentes remotos. Este marco de ejecución (que en Java sería la máquina virtual) es el CLR (*Common Language Runtime*) y podría considerarse como el núcleo del *framework*, pues todas las aplicaciones son gestionadas y supervisadas por él.

Con el objetivo de proporcionar una capa de abstracción adicional del hardware, favorecer una ejecución multiplataforma y la integración entre diferentes lenguajes, se ha diseñado un lenguaje intermedio: el MSIL (*Microsoft Intermediate Language*). Este lenguaje intermedio es independiente del juego de instrucciones de una CPU específica y proporciona un nivel de abstracción superior a los lenguajes nativos comunes pues incluye instrucciones para el tratamiento de objetos, de excepciones, tablas, etc. Los distintos compiladores de .NET se encargan de generar este lenguaje intermedio y el CLR es el encargado de verificar dicho código y de ejecutarlo, transformándolo a código nativo.

El proceso de ejecución de código pasa por dos etapas: la compilación a código intermedio, y la carga y ejecución de este código intermedio por el CLR. En la Figura 43 se muestra el proceso de transformación que sigue el código fuente hasta poder ser ejecutado. En la primera fase, a partir del

código fuente, el compilador genera un ensamblado que contiene los metadatos que describen los tipos definidos y el código intermedio.

En la segunda fase, cuando se requiera la ejecución de un ensamblado, el código IL es interpretado por el compilador JIT (*Just-In-Time*) que lleva el CLR, es convertido a lenguaje máquina y puesto en ejecución. Si dicho código tiene dependencias con otros ensamblados, gracias a los metadatos el CLR los va cargando sucesivamente, siguiendo el mismo proceso.

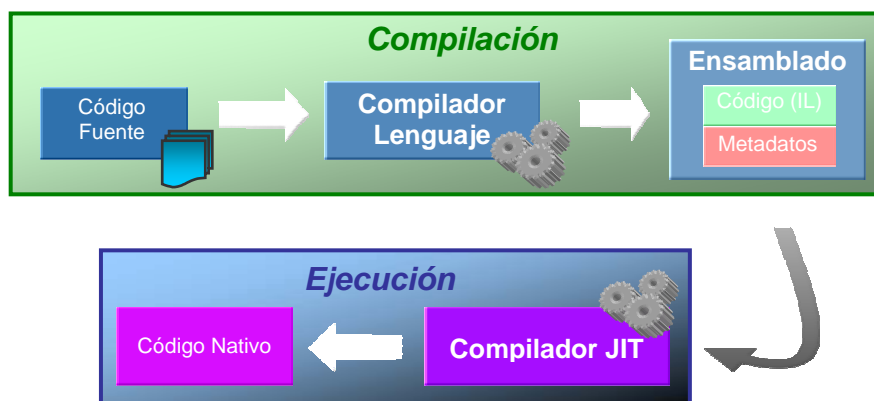


Figura 43 - Proceso de compilación en .NET

El compilador JIT va convirtiendo dinámicamente el código MSIL a ejecutar en código nativo según sea necesario. Únicamente convierte el código que se vaya a ejecutar, almacenándolo por si fuera necesario en futuras llamadas. Existe una versión del compilador, el *prejitter*, mediante la cual es posible compilar completamente cualquier ensamblado a código nativo, con lo que futuras ejecuciones no necesitarán de la compilación dinámica.

El CLR, además de encargarse de cargar y ejecutar el código, también se encarga de verificar la seguridad de la aplicación que está ejecutando, con el objetivo de evitar operaciones inválidas o que accedan a zonas de memoria no permitidas. Para ello, el CLR utiliza los siguientes mecanismos:

- Verifica la seguridad de los tipos, a través del CTS y aísla la memoria dedicada a la aplicación. Con estas medidas se pretende evitar los fallos comunes de *buffer overrun* (desbordamiento de búfer, a causa del cual se accede al área de memoria de otra aplicación), que los tipos están dentro de los rangos establecidos, etc.
- Comprueba que la aplicación en ejecución dispone de los permisos adecuados sobre los recursos que quiere utilizar, que dependerá de la confianza que el usuario haya depositado sobre dicha aplicación.
- Gestiona las excepciones provocadas, dotando a la aplicación de mecanismos para la gestión de sus errores.

Sin embargo, .NET permite generar código no administrado, es decir, no supervisado por el CLR, con la finalidad de permitir el desarrollo de *drivers* específicos que necesiten acceder a recursos de muy bajo nivel o para el desarrollo de aplicaciones cuya eficiencia sea crítica. El código no

administrado es código nativo, y por lo tanto, no goza de los servicios que ofrece el CLR en tiempo de ejecución. No obstante, también se proporcionan mecanismos para que el código administrado por el CLR pueda comunicarse con código no administrado, que incluye desde objetos COM hasta código incluido en DLLs.

A.3 La Librería de Clases

Con el objetivo de facilitar el desarrollo de aplicaciones, y de la misma forma que Java, .NET también proporciona una extensa librería de clases y tipos reutilizables que permiten al desarrollador invertir el mínimo tiempo posible en el desarrollo de la infraestructura de la aplicación, centrándose más en la lógica de negocio. Además, como estas librerías están construidas sobre la naturaleza orientada a objetos del CLR, su funcionalidad puede ser extendida a través de la herencia. Y como también están escritas en código intermedio, podrán ser utilizadas por cualquier lenguaje cuyo compilador genere código MSIL (es decir, lenguajes soportados por .NET).

Debido a la gran cantidad de clases existentes, éstas se agrupan de forma jerárquica en lo que se denominan *espacios de nombres (namespaces)*, los cuales agrupan clases de funcionalidades similares e independientes. En la Figura 44 se puede observar la estructura dividida en capas que constituye el entorno .NET.

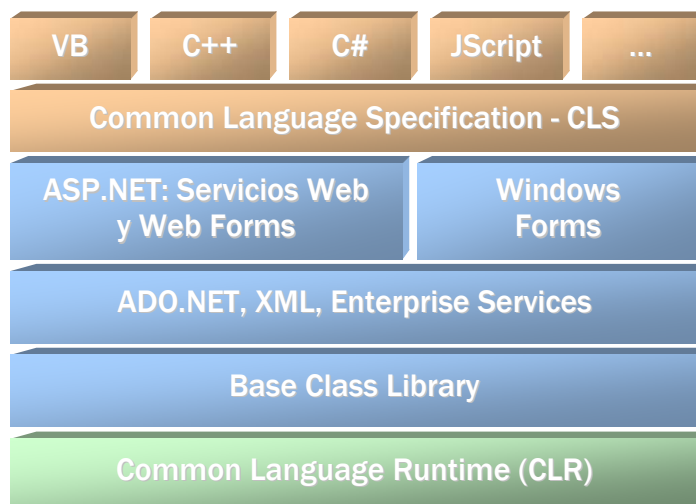


Figura 44 - .NET Framework

En primer lugar se encuentran los distintos lenguajes, que deben cumplir las reglas especificadas por el CLS. En último lugar se encuentra el CLR que es el encargado de cargar y ejecutar las aplicaciones de .NET en la máquina destino. Entre estas dos capas se encuentran una serie de niveles que representan los distintos servicios que ofrece .NET a través de sus librerías:

- *ASP.NET*: proporciona una infraestructura y un nuevo modelo de programación para los desarrollos basados en Web. Está formado por dos modelos de programación: las aplicaciones web y los Servicios Web.
- *Windows Forms*: proporciona los servicios necesarios para el desarrollo de interfaces de usuario de escritorio, introduciendo un único conjunto de clases para cualquier lenguaje de .NET.
- *ADO.NET*: forma el conjunto de servicios necesarios para el acceso a fuentes de datos, utilizando XML como formato de transmisión de datos. Basado en el modelo anterior de ADO, aumenta su escalabilidad e interoperabilidad, y permite el trabajo sobre conjuntos de datos desconectados (datos cuya procedencia no está disponible temporalmente).
- *XML*: ofrece clases para crear y formatear documentos XML .
- *Enterprise Services*: proporciona un conjunto de servicios para la compatibilidad con componentes COM+.
- *Base Class Library*: además de las clases y tipos básicos, contiene todo un conjunto de servicios para el soporte de las operaciones más comunes en el desarrollo de la mayoría de aplicaciones como son el manejo de excepciones, el tratamiento de fechas, la gestión de la entrada/salida, tablas, pilas y colas, etc.

A.4 .NET Remoting

De la misma forma que .NET ha venido a sustituir la tecnología COM para el desarrollo de componentes, .NET Remoting es la nueva tecnología que ha propuesto Microsoft para sustituir a DCOM, el método más extendido en plataformas Win32 para el desarrollo de aplicaciones distribuidas. .NET Remoting es una arquitectura orientada a objetos que facilita el desarrollo de aplicaciones distribuidas, permitiendo que dos objetos que se encuentren en máquinas distintas, comunicadas a través de una red cualquiera, puedan comunicarse entre sí. Además de ofrecer una gran cantidad de servicios y protocolos estándar ya implementados, permite usar y/o construir formatos de codificación o protocolos de comunicación distintos. Frente a otras tecnologías para el desarrollo de aplicaciones distribuidas, presenta como ventaja su facilidad de uso, pues no necesita la generación de interfaces en un lenguaje abstracto como en CORBA o DCOM, ni tampoco ciclos de generación de proxys/stubs como en Java/RMI. También proporciona los mecanismos básicos para los Servicios Web de .NET, implementando estándares como SOAP y WSDL. Para más detalles, puede consultarse [Ramm02] y [Scott03] donde se trata en profundidad la tecnología .NET Remoting.

A.5 Aspectos avanzados de .NET

El entorno .NET añade una serie de tecnologías que, utilizadas convenientemente, permite al desarrollador generar dinámicamente ensamblados, o incluso permitir la evolución dinámica del software desarrollado. A continuación se describirán brevemente algunos de los conceptos más relevantes, como los nuevos tipos añadidos (atributos y delegados) o tecnologías orientadas a la emisión dinámica de código (reflexión, emisión de código y CodeDom).

A.5.1 Atributos

Como se ha comentado anteriormente, los metadatos permiten describir los tipos implementados por una aplicación, pero además, el desarrollador tiene la posibilidad de añadir nuevos metadatos que aporten más información, a través de los *Attributes* de .NET. Los atributos se definen como construcciones del lenguaje que permiten añadir información adicional (metadatos) a los elementos del lenguaje y extender su funcionalidad. Pueden ser asociados a: clases, métodos, interfaces, ensamblados, parámetros, propiedades, etc.

De esta forma, pueden crearse metadatos que representen información de diseño (como documentación sobre los tipos a los que van asociados), información en tiempo de ejecución (como el nombre de una columna de una base de datos o de un campo) o hasta nuevas características de comportamiento en tiempo de ejecución (como indicar que un miembro es serializable, o que se ha de comportar de determinada forma).

El uso de los atributos puede separarse en tres etapas: en la primera se define el atributo como una clase derivada de la clase *Attribute*, utilizando para ello las construcciones comunes de cualquier lenguaje; en la segunda etapa se asocia el atributo a los elementos correspondientes; y en la tercera etapa, en tiempo de ejecución y a través de la reflexión, se obtienen los atributos asociados a un elemento del código para actuar de la forma que se estime oportuna. Además, como los atributos se definen como clases, pueden contener propiedades adicionales que codifiquen información relativa al estado del atributo.

A.5.2 Delegados

Una innovación del CTS es el tipo *Delegate* (delegados). Son tipos de referencia que encapsulan un método con una signatura específica y se utilizan para proporcionar procesamiento asíncrono de servicios, para inyectar código en llamadas a funciones genéricas (por ejemplo, para proporcionar la forma de ordenar elementos en una función de ordenación) y para el manejo de eventos de forma asíncrona. De forma similar a los punteros a funciones existentes en el lenguaje C, los delegados encapsulan

la referencia a un método y se comportan como variables, por lo que pueden ser pasados como parámetros en otras funciones.

A.5.3 Reflexión

Los servicios de reflexión permiten, a través de los metadatos, obtener en tiempo de ejecución información sobre los tipos definidos en una aplicación o de los módulos a los que acceda. De esta forma, puede obtenerse dinámicamente el nombre del ensamblado que se está ejecutando, interrogar a distintas clases por los métodos que ofrecen, los campos y propiedades públicos, etc. Para ello, mediante los servicios ofrecidos por el espacio de nombres *System.Reflection* puede obtenerse el tipo de cualquier instancia e interrogarlo para obtener todas sus propiedades.

Una primera aplicación que se puede derivar de este mecanismo es el “enlace tardío” (o *Late Binding*) mediante el cual las librerías dinámicas se pueden cargar y enlazar dinámicamente a la aplicación en función de si contienen clases específicas requeridas por la aplicación (por ejemplo, que herede de una determinada clase, o que implementen determinada interfaz). Además, la reflexión también es utilizada por otras librerías de .Net con diversos fines: por ejemplo para serializar objetos (guardar el estado de un objeto que se encuentra en memoria) se utiliza la reflexión para obtener los campos que pueden ser serializados o no.

A.5.4 Emisión de código

Sin embargo, la principal ventaja de los servicios de reflexión proporcionados por la librería es la posibilidad de emitir dinámicamente código. Para ello se utilizan los tipos y sus correspondientes servicios definidos en el espacio de nombres *System.Reflection.Emit* para crear primero un ensamblado en memoria, definir los tipos que lo formarán (junto a sus miembros) y proceder a emitir instrucciones MSIL. Posteriormente, el ensamblado podrá guardarse en disco o dejarse en memoria si no se necesita la persistencia del código generado.

El principal inconveniente que plantea este mecanismo es que requiere por parte del desarrollador un conocimiento profundo del código intermedio, a medio camino entre los lenguajes de alto nivel y el lenguaje máquina, lo que puede provocar numerosos errores de codificación.

A.5.5 CodeDom

Frente a la emisión de código intermedio, pueden utilizarse los servicios del espacio de nombres *System.CodeDom* para generar código dinámicamente a partir de plantillas de código o para la compilación dinámica de código. Para ello, utilizando tipos básicos definidos en dicha librería y que representan cada uno de los tipos disponibles en el *Common Type System*, se construye un grafo de objetos. Este grafo de objetos proporciona un modelo de objetos independiente de cualquier lenguaje de programación y representa la

estructura del código fuente en memoria. Este grafo podrá ser transformado a cualquier lenguaje utilizando para ello un generador de código *CodeDom* específico para el lenguaje de destino. Por otra parte, *CodeDom* también proporciona servicios para compilar dinámicamente ficheros de código fuente a un ensamblado binario.

Se ha de tener en cuenta que *CodeDom* no ha sido diseñado para soportar todas las estructuras y tipos que se pueden encontrar en los distintos lenguajes de programación soportados por .NET, sino tan sólo aquellos tipos básicos comunes a todos los lenguajes, es decir, los definidos en el CTS. Sin embargo, este aspecto ha sido solucionado a través de unas estructuras sintácticas que permiten añadir código específico de un lenguaje (los *codeSnippets*), pero no se garantiza la conversión de dicho código a otro lenguaje distinto.

La principal ventaja es que el desarrollador, tras aprender las estructuras sintácticas de *CodeDom* podrá generar código dinámicamente sin necesidad de conocer el lenguaje intermedio que se genera, utilizando para ello el mismo lenguaje en el que ha desarrollado la aplicación. Otra utilidad derivada de esta tecnología es transformar código fuente de un lenguaje a otro lenguaje, aunque en la actualidad aún no se encuentra totalmente soportada, pues el ritmo en el que se añaden nuevas estructuras al CLR no es el mismo al que se implementan dichos cambios en los distintos compiladores de *CodeDom*.