Marco de trabajo para el desarrollo de arquitecturas software orientado a aspectos

TESIS DOCTORAL

Departamento de Ingeniería de Sistemas Informáticos y Telemáticos
Universidad de Extremadura

Dirigida por:
D. Juan Manuel Murillo Rodríguez
Titular de Universidad
Área de Lenguajes y Sistemas Informáticos

Presentada por:
Dña. Amparo Navasa Martínez
para optar al grado de Doctor
en Ingeniería Informática

Cáceres, Diciembre 2008

Edita: Universidad de Extremadura
Servicio de Publicaciones

Caldereros 2. Planta 3ª
Cáceres 10071
Correo e.: publicac@unex.es
http://www.unex.es/publicaciones

D. Juan Manuel Murillo Rodríguez, Titular de Universidad en el Área de Lenguajes y Sistemas Informáticos del Departamento de Ingeniería de Sistemas Informáticos y Telemáticos de la Universidad de Extremadura

Certifica que

Dña. Amparo Navasa Martínez, Licenciada en Matemáticas, ha realizado en el Departamento de Ingeniería de Sistemas Informáticos y Telemáticos de la Universidad de Extremadura, bajo mi dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

Marco de trabajo para el desarrollo de arquitecturas software orientado a aspectos

Revisado el presente trabajo, estimo que puede ser presentado al tribunal que ha de juzgarlo y autorizo la presentación de esta Tesis Doctoral en la Universidad de Extremadura

Cáceres a 15 de Julio de 2008

Fdo.: Juan Manuel Murillo Rodríguez
Titular de Universidad
Área de Lenguajes y Sistemas Informáticos
Universidad de Extremadura

Agradecimientos

Esta tesis doctoral es la culminación de muchos años de trabajo, que no hubiera podido llevar a cabo sin el apoyo y la ayuda de mi familia, mis amigos y mis compañeros. A todos ellos quiero agradecer el haber estado siempre ahí, en los buenos y en lo malos momentos, que han sido muchos.

No puedo, ni debo dejar de mencionar a mi director de tesis, Juan Manuel, y la confianza que depositó en mí al empezar a trabajar juntos. Sólo él sabe la dedicación y el tiempo que ha supuesto dirigir mis pasos, dándome siempre libertad para aportar mis ideas a lo largo de estos años.

Una mención especial tiene que ser para mi familia, mi marido y mis hijos, que, casi sin comprender lo que hacía, han estado siempre cerca apoyándome, dándome ánimos y como no, soportando mi estrés, mis viajes y mi dedicación a la tesis, en lugar de a ellos. A mis padres, que veían con preocupación que el trabajo me quitaba el tiempo que tenía que dedicar a vivir. Gracias, ellos me han dado fuerzas para seguir luchando. Prometo compensar todas esas carencias que en ningún momento me han reprochado. Espero que no sea demasiado tarde.

Quiero nombrar aquí a mis amigos y compañeros de fatigas, Miguel Ángel, Marisol, Pedro, Pedro Luís y Myriam, que me han soportado estos largos años de angustias y de algunos triunfos. A pesar de todo, me quedo con los buenos ratos que hemos pasado, sus consejos y sus ánimos. Sin ellos, probablemente hubiera tirado la toalla hace tiempo. Quiero agradecer los comentarios y revisiones que han realizado sobre los borradores de esta tesis.

Un recuerdo merecen los alumnos que a lo largo de estos años han trabajado conmigo. De entre ellos, destacaría, por ser los últimos, a Maria, Iván y Cristina, que me han facilitado el trabajo y, tras el tiempo que hemos compartido, creo que puedo decir que somos amigos.

Esta tesis es fruto del tesón y del trabajo. Quiero dar gracias a Dios por darme fuerza para soportar todos los contratiempos y tantos años de sinsabores. Por fin, hoy presento este trabajo, que debe ser el punto de partida para afrontar nuevos retos.

A la memoria de mi padre

A mi fmilia

Resumen

La gestión de sistemas de software complejos es uno de los retos más importantes de la Ingeniería del Software (IS). El ingeniero de software debe afrontar el desarrollo de sistemas software cada vez más complejos y con requisitos más estrictos. Además, el mundo real cambia, evoluciona y los sistemas deben adaptarse a él para seguir siendo útiles. Por esta razón, la adaptación debe realizarse de manera sencilla; los ingenieros de software deben encontrar técnicas y herramientas que se lo permitan. Sin embargo, los métodos tradicionales (métodos estructurados o paradigmas orientados a objetos) sólo permiten una reconfiguración y adaptación parcial de los sistemas.

Se han considerado muchas soluciones a esta cuestión, siendo la separación de aspectos una de ellas. La *Programación Orientada a Aspectos* se propuso como una manera de aplicar la separación de aspectos, tratando de superar las limitaciones de la Programación Orientada a Objetos. Desde entonces, la *Programación Orientada a Aspectos* y otras técnicas centradas en la modularización del código transversal se agruparon bajo el nombre de "Separación avanzada de intereses (*concerns*)". La idea es extraer el código relacionado con los *crosscutting concerns* (un *concern* captura los requisitos que atraviesan múltiples módulos en un sistema), que está disperso a lo largo del código del sistema, en módulos independientes. Esta separación evita el código enmarañado y permite la reutilización de un aspecto. En muchos trabajos de investigación, los aspectos se consideran como una parte importante del proceso de desarrollo de los sistemas complejos y proponen técnicas para extraerlos de una manera efectiva. Muchos de estos métodos *Orientados a Aspectos* se centran principalmente en la especificación y extracción de éstos en el diseño detallado o en el nivel de implementación, prestando menos atención al impacto de los aspectos en las primeras fases del ciclo de vida. *Desarrollo de Software Orientado a Aspectos* (DSOA) es la denominación utilizada para referirse a las técnicas y tecnologías de modularización de los *crosscutting concerns* a lo largo del ciclo de vida.

Por otra parte, los nuevos métodos de la Ingeniería del Software consideran a la *Arquitectura del Software* (AS) como una parte importante de la etapa de diseño que ayuda a controlar la complejidad de los sistemas. En ella éstos se definen como un conjunto de componentes que describen su funcionalidad a alto nivel y se conectan a través de un conjunto de conectores arquitectónicos.

Durante el diseño arquitectónico se lleva a cabo la definición estructural del sistema. El interés de definir un diseño arquitectónico *Orientado a Aspectos* surge cuando se observa que los *crosscutting concerns* atraviesan también a los componentes arquitectónicos. Además, considerar los aspectos en esta fase facilita la gestión de la evolución al poder considerarlos artefactos software. Por ello, es necesario disponer de mecanismos que permitan identificar los aspectos durante la arquitectura del software.

A su vez, el *Desarrollo Software Basado en Componentes* (DSBC) es otro paradigma de la ingeniería del software que pretende facilitar la construcción de grandes sistemas software de calidad, que evolucionen fácilmente, reduciendo el coste y el tiempo de los desarrollos. La filosofía de DSBC puede provocar un cambio en cuanto a la forma de afrontar los desarrollos, pasándose de considerar líneas de código a la definición de modelos.

El uso combinado de los paradigmas y disciplinas mencionadas (DSOA, AS y DSBC) puede facilitar el desarrollo de sistemas software complejos, proporcionándose métodos, procedimientos y herramientas. El trabajo que aquí se presenta considera conjuntamente el *Desarrollo de Sistemas Orientado a Aspectos*, la *Arquitectura del Software* y las características generales del *Desarrollo de Sistemas Basado en Componentes*. Así, el objetivo de esta tesis es proporcionar un **marco de trabajo para el desarrollo de sistemas software orientado a aspectos** (*AOSA Space*), que está formado por un modelo arquitectónico: *AOSA Model*, una metodología de trabajo, un lenguaje de descripción arquitectónica (LDA) orientado a aspectos y una herramienta para su utilización.

El modelo propone hacer una especificación estructural de un sistema orientado a aspectos (OA) considerando los aspectos como entidades de primera clase, concretamente como componentes arquitectónicos. *AOSA Model* se basa en un LDA convencional y un modelo de coordinación. Sin embargo, surge un problema al tratar de representar sistemas OA pues los LDA convencionales no soportan conceptos de OA. Además se ha considerado fundamental que la inserción de los aspectos se realice observando el principio de inconsciencia. De este modo, los componentes que constituyen el sistema en el que se insertan los aspectos no cambian. Además, para resolver el problema de la ejecución coordinada de componentes y aspectos, en *AOSA Model* se ha considerado un modelo de coordinación; concretamente *Coordinated Roles* que se basa en protocolos de notificación de eventos.

Por otra parte, hay varias propuestas para desarrollar sistemas OA a lo largo del ciclo de vida, pero sólo algunas de ellas proporcionan un soporte lingüístico. A lo largo de este trabajo se ha considerado interesante dotar a los sistemas OA de un soporte formal, igual que se hace en el desarrollo de sistemas convencionales. Así, igual que los sistemas convencionales se representan en esta etapa mediante LDA, los sistemas OA deberían expresarse también mediante LDA adecuados. En este trabajo se muestra cómo es posible dotar al proceso de desarrollo de un sistema OA de un soporte lingüístico durante la fase de diseño arquitectónico.

En otro orden de cosas, los sistemas software tienen que adaptarse al mundo cambiante en el que se definen. Por ello, cuando se trabaja desde el punto de vista de la evolución de los sistemas, es especialmente interesante considerar que los componentes que forman el sistema a actualizar no cambien al incluir nuevos requisitos. En estos casos, tales modificaciones pueden considerarse como aspectos, por lo que se podrían aplicar los principios de DSOA. Por ello, cuando la evolución se trata desde la fase de diseño arquitectónico se puede aplicar *AOSA Space*, pues también facilita la evolución de los sistemas complejos, su representación formal y el chequeo de la arquitectura obtenida.

Una de las aportaciones de *AOSA Space*, el marco de trabajo que se propone, es la manera en la que se integran las aproximaciones en las que se basa, para aprovechar las ventajas que aporta cada una:

- La separación de aspectos se considera desde las primeras fases del desarrollo, aplicando los principios de DSOA.
- Mejora el mantenimiento y la evolución de los sistemas complejos, al tratar los cambios de sistemas existentes desde un punto de vista arquitectónico, considerándolos como aspectos, en aquellos casos en los que sea posible. Se aplican aquí conceptos de DSOA y de arquitectura del software.
- La definición de los aspectos como componentes arquitectónicos, que permanecen separados (no se tejen) a lo largo del desarrollo, facilita la reutilización y su sustitución. Se utilizan los tres paradigmas mencionados (DSOA, AS y DSBC).
- Tanto en tiempo de diseño como en tiempo de evolución se mantiene el principio de inconsciencia por lo que los componentes que forman el sistema en el que se insertan los aspectos no se modifican. En este caso, se integran conceptos de DSOA y DSBC.
- La utilización de un LDA-OA en la representación de los sistemas facilita la descripción formal de los mismos, permite ejecutar un prototipo del mismo y chequear la arquitectura, para determinar si el comportamiento del sistema es el esperado, utilizándose conjuntamente las características del DSOA y de la AS.
- Un juego de herramientas facilita la labor del arquitecto de software a lo largo del proceso de desarrollo de un sistema orientado a aspectos.

Sin embargo, podemos decir que la principal aportación de esta tesis doctoral es *tratar el problema de la separación de aspectos como uno de coordinación*.

Índice general

Índice de figuras

Indice de Tablas

CAPÍTULO 1
Introducción

El objeto de esta tesis titulada "Marco de trabajo para el desarrollo de arquitecturas software orientado a aspectos" es:

> presentar un modelo de desarrollo de sistemas orientado a aspectos, desde un punto de vista arquitectónico. Para la aplicación del modelo se dispondrá de una metodología, un Lenguaje de Descripción Arquitectónica Orientado a Aspectos y una herramienta que ayude al ingeniero de software a desarrollar sistemas software complejos.

Dado que los sistemas software tienen que mantenerse actualizados para adaptarse a los cambios del entorno en el que han sido definidos, se pretende así mismo que la propuesta

> permita que el desarrollador pueda diseñar sistemas fácilmente adaptables y de evolución sencilla. Para ello, los cambios se promoverán desde un punto de vista estructural.

En este capítulo se presenta el ámbito en el que se inscribe este trabajo y se detallan los objetivos marcados.

La estructura de este capítulo es la siguiente: la sección 1 enuncia el ámbito en el que se realiza el trabajo; la sección 2 resume los antecedentes que han motivado la realización del mismo y centra el universo del discurso; la sección 3 presenta la justificación de la estructura de esta memoria y el modo en el que se ha abordado el trabajo: por una parte la tarea de definir un modelo arquitectónico para sistemas orientados a aspectos; por otra, la definición de un lenguaje de descripción de arquitecturas orientado a aspectos; la sección 4 explica los principales objetivos de esta tesis doctoral. Finalmente, en la sección 5 se resume la estructura de los capítulos que la componen.

1.1. Contexto de la tesis

Esta tesis doctoral hay que encuadrarla dentro de los trabajos que el grupo *Quercus* de Ingeniería del Software de la Universidad de Extremadura ha venido desarrollando en los últimos años sobre la separación de aspectos y el desarrollo de sistemas basados en componentes. Así, las primeras tesis doctorales relacionadas con este trabajo están enfocadas a la separación de aspectos de sincronización [San99] y de coordinación [Mur01] en los lenguajes de programación orientados a aspectos. Las siguientes tesis doctorales del grupo estuvieron enfocadas a describir cómo representar el diseño de sistemas para su utilización en aplicaciones orientadas a aspectos [Her03] y en formalizar el aspecto de coordinación para validar sistemas coordinados [San04]. Por último, los esfuerzos se han encaminado hacia el desarrollo dirigido por modelos utilizando componentes y aspectos [Cle07].

En este contexto, esta tesis doctoral está enfocada hacia el estudio y desarrollo de arquitecturas de sistemas orientadas a aspectos, definiéndose un marco de trabajo para el estudio de la integración de aspectos durante el diseño de alto nivel. Este trabajo ha sido desarrollado junto a la tesis doctoral [Per08] que se centra en el estudio de cómo la inserción de aspectos afecta a los sistemas software, así como en la detección de posibles patologías en el código final obtenido.

1.2. Ámbito del estudio

El ámbito en el que se inscribe esta tesis doctoral se centra en el estudio de las *Arquitecturas Software* y el paradigma del *Desarrollo de Sistemas Orientado a Aspectos*; particularmente se desarrolla un modelo Arquitectónico Orientado a Aspectos. Así pues, la propuesta que se describe a lo largo de esta memoria define un marco de trabajo que facilita la inserción de aspectos en la definición arquitectónica de un sistema. El marco de trabajo, que se ha denominado *AOSA Space,* incluye:

- Un modelo arquitectónico orientado a aspectos, *AOSA Model*, que permite especificar arquitecturas software orientadas a aspectos. *AOSA Model*, que se describe en el capítulo 4, combina el uso de dos aproximaciones que facilitan la definición de arquitecturas software: el *Desarrollo de Sistemas Basado en Componentes* y el *Desarrollo de Sistemas Orientado a Aspectos*. El modelo se define a partir de la concepción de los aspectos como entidades de primera clase, en particular como componentes, y la utilización de un modelo de coordinación para coordinar la composición de los aspectos en la arquitectura existente. Se propone la definición de una arquitectura en dos niveles en la que el *nivel base* corresponde a la arquitectura inicial y un meta nivel denominado *nivel de aspecto* en el que se superponen los elementos necesarios para generar la arquitectura orientada a aspecto.
- Una metodología de trabajo, que se describe en el capítulo 5, permite obtener la estructura de un sistema orientado a aspectos a partir de uno inicial que se extiende.

La metodología abarca varias fases del ciclo de vida, que se ejecutan de modo iterativo, sirviendo de guía al arquitecto a través de los pasos que se deben dar para desarrollar los sistemas siguiendo *AOSA Model*. A partir de la descripción arquitectónica de un sistema, se facilita la inclusión de nuevos requisitos en forma de aspectos que se insertan en la arquitectura teniendo en cuenta las especificaciones del modelo. Los nuevos requisitos, para poder considerarlos como aspectos, deben ser ortogonales. El sistema obtenido se puede validar utilizando herramientas adecuadas, se puede generar un prototipo ejecutable de la nueva arquitectura. Además, la metodología está soportada por herramientas que automatizan el proceso.

- Un Lenguaje de Descripción Arquitectónica Orientado a Aspectos: *AspectLEDA*, que se ha definido para formalizar y analizar las arquitecturas modeladas:
  AspectLEDA se describe en el capitulo 6; se ha diseñado a partir de LEDA, un LDA basado en cálculo π que permite definir un prototipo de la arquitectura y generar código Java.
- Una herramienta: *AOSA Tool/LEDA,* que asiste al arquitecto y facilita la obtención de la arquitectura orientada a aspectos objetivo, así como la ejecución de un prototipo generado automáticamente tras el proceso. Una interfaz de usuario amigable ayuda al arquitecto del software a introducir la información necesaria para generar el sistema extendido en *AspectLEDA*.

1.3. Antecedentes

La gestión de sistemas complejos es uno de los retos que debe afrontar la Ingeniería del Software actual. Además, el mundo real cambia y los sistemas deben adaptarse a él para seguir siendo útiles. Por esta razón, los ingenieros de software deben encontrar técnicas y herramientas que le permitan adaptar los sistemas de manera sencilla. Sin embargo, los métodos tradicionales (métodos estructurados o paradigmas orientados a objetos) sólo permiten una reconfiguración y adaptación parcial de los mismos.

La Programación Orientada a Objetos (POO) es un paradigma de programación que realiza una descomposición funcional [Par72] pero tiene la limitación de que algunos temas de interés del problema no pueden localizarse en un único módulo sino que atraviesan varios elementos estructurales del sistema. La *separación de intereses* (*separation of concerns*), introducida en [Par72], es un principio de la ingeniería del software que trata de resolver el problema de modo que los diferentes *concerns* (temas de interés, asuntos) de un sistema se consideren individualmente. Dijkstra en [Dij76] demostró que esta división proporciona mejores resultados y tiene mayores ventajas, incluyendo la reducción de la complejidad del software y mejorando la modularidad, la reutilización y el mantenimiento de los artefactos software. En 1996, Kiczales [Kic+96] propuso la *Programación Orientada a Aspectos* –POA- (*Aspect Oriented Programming* –AOP-) como una manera de aplicar la s*eparación de aspectos* para resolver las limitaciones de la POO. Esta propuesta difiere de las anteriores en que la POA introduce una nueva definición de *concern*: *aquellos intereses que pertenecen al desarrollo del*

sistema, su operación o cualquier otro aspecto que sea importante para un usuario o desarrollador.

Desde entonces, la POA y otras técnicas, centradas en la modularización del código transversal, se agruparon bajo el nombre de *Separación Avanzada de Intereses* (o en su denominación inglesa *Advanced Separation of Concerns)*. La idea es extraer el código relativo a los *crosscutting concerns* (*concerns* –intereses o propiedades- que capturan requisitos y cruzan varios módulos del sistema [AOSD]) que está disperso a lo largo del mismo, en módulos independientes. Esta separación evita el código enmarañado (*tangled*) y permite la reutilización de los aspectos. En diversos trabajos de investigación, los aspectos se consideran como una parte importante del proceso de desarrollo de los sistemas complejos y proponen técnicas para extraerlos de una manera efectiva (como *Composition Filters* [BeAk01], *AspectJ* [Kic+01], *Adaptive Programming* [LiOrOv01], *HyperJ* [OsTa00], etc.). Muchas de estas aproximaciones orientadas a aspectos (OA) se centran en la especificación y extracción de aspectos durante el diseño detallado o en el nivel de implementación, prestando menos atención al impacto de los aspectos en las primeras fases del ciclo de vida. En 2002 se popularizó el nombre de "*Desarrollo de Software Orientado a Aspecto*" (*DSOA*) para referirse a las técnicas y tecnologías de modularización de los *crosscutting concerns* a lo largo del ciclo de vida.

Por otra parte, los nuevos métodos de la ingeniería del software consideran a la arquitectura del software como una parte importante de la etapa de diseño: *la arquitectura del software es una actividad que ayuda a los desarrolladores a controlar la complejidad de los sistemas y a definir sus estructuras de modo que su mantenimiento y evolución sean sencillos*. Esto es así porque durante esta etapa, se define la estructura de un sistema como un conjunto de componentes que describen la funcionalidad del sistema a alto nivel y se conectan a través de un conjunto de conectores arquitectónicos.

Durante la arquitectura del software se lleva a cabo la definición estructural del sistema; la conveniencia de definir un diseño arquitectónico orientado a aspectos surge cuando se observa que los *crosscutting concerns* atraviesan también los componentes arquitectónicos de modo que el diseño final es complejo. Además, considerar los aspectos en esta fase facilita la gestión de la evolución del sistema. Esto es así porque durante el diseño arquitectónico los aspectos se pueden considerar como artefactos software. En este sentido, diversas propuestas consideran los aspectos arquitectónicos como artefactos de distinto tipo: componentes, conectores, *slices*, vistas…

Una consideración a tener en cuenta en el desarrollo de las arquitecturas del software OA es cómo expresarlas. Según se documenta en el capítulo 3 de esta memoria, hay varias propuestas para desarrollar sistemas OA a lo largo del ciclo de vida y en particular durante la fase del diseño arquitectónico, pero sólo algunas de ellas proporcionan un soporte lingüístico. Es necesario dotar a los sistemas OA de un soporte formal, igual que se hace en las propuestas de desarrollo de sistemas convencionales, particularmente durante su definición estructural. Por tanto, igual que los sistemas ordinarios se describen en esta etapa mediante lenguajes de descripción arquitectónica (LDA), los sistemas OA deberían expresarse también mediante LDA adecuados. Sin embargo, los LDA convencionales no permiten al diseñador representar adecuadamente los sistemas orientados a aspectos. De ahí la necesidad de definir Lenguajes de

Descripción Arquitectónica Orientados a Aspectos (LDA-OA). En esta memoria se muestra cómo es posible dotar al proceso de desarrollo de un sistema OA de un soporte lingüístico durante la fase de diseño.

1.4. Motivación

Concretando la exposición realizada hasta el momento, en esta sección se establecen las motivaciones principales que han llevado a la realización de esta tesis doctoral.

El tratamiento de los intereses transversales (*crosscutting concerns*) a nivel arquitectónico y el modelado de aspectos son dos técnicas que, en los últimos años, han captado el interés de la comunidad investigadora. Prueba de ello es la gran cantidad de publicaciones y conferencias internacionales específicas sobre el tratamiento de aspectos en fases tempranas del desarrollo. Dos de los puntos de interés en este ámbito son el desarrollo de modelos que faciliten la construcción de Sistemas Orientados a Aspectos y la definición de Lenguajes de Descripción Arquitectónica Orientados a Aspectos.

1.4.1. Definición de modelos arquitectónicos OA

Dada la complejidad de los sistemas a desarrollar hoy día, es necesario que los ingenieros de software dispongan de técnicas que les permitan reducir dicha complejidad. Una de estas técnicas es el DSOA. Esta disciplina ayuda a mejorar el proceso de desarrollo software, mediante la utilización del concepto de *aspecto* a lo largo de todo el ciclo de vida[1]. El DSOA proporciona técnicas que permiten una identificación temprana de aspectos, su extracción, representación y posterior composición. En este paradigma, los aspectos se consideran como entidades de primera clase que pueden ser manipulados a lo largo del proceso de desarrollo de un sistema; además, la definición de aspectos es un mecanismo de abstracción por el que éstos pueden incorporarse a un sistema existente de tal modo que los elementos que se añaden no quedan dispersos a lo largo de diversos módulos, sino que permanezcan en módulos independientes.

Al modelarse las propiedades que atraviesan el sistema como artefactos software, se logra que grandes sistemas desarrollados siguiendo este paradigma se puedan adaptar a los cambios y a la evolución de sus propiedades de un modo más sencillo que siguiendo paradigmas tradicionales, en los cuales el código de las propiedades o características transversales queda distribuido a lo largo de varios módulos o componentes del sistema.

Es interesante considerar juntos los conceptos de AS y DSOA. Ello permitirá gestionar la complejidad de un sistema, desde un punto de vista estructural, siguiendo el paradigma de DSOA:

[1] En [Fil+05] un *aspecto* se define como "una unidad modular diseñada para implementar un *concern*".

- La AS considera la estructura de los sistemas cuando ésta se está definiendo y estudia su modularización, su composición y sus consecuencias. Este concepto es próximo al de separación de aspectos (*separation of concerns*) que establece que cada *concern* debería mantenerse en un módulo independiente. Después de que los módulos se hayan definido, surge un conjunto de estructuras que los combinan para estudiarlos todos juntos. Por tanto, es útil considerar la AS (como técnica para la modularización y separación de propiedades) de un sistema para reducir su complejidad.
- El DSOA facilita que el diseño de los sistemas sea claro y hace su evolución más sencilla: las propiedades transversales se implementan en unidades modulares, lo que mejora la posibilidad de reutilización y la adaptabilidad de los sistemas.

El concepto de *aspecto*, a nivel de arquitectura, introduce un nuevo tipo de modularización y composición del software que define nuevas estructuras que deben ser estudiadas por la AS, así como determina las características arquitectónicas de los aspectos. Recientemente, se han presentado varios estudios cuyos resultados muestran los beneficios de la aproximación arquitectónica en el marco de la orientación a aspectos [AOSD, Early]. Para ello, es necesario definir mecanismos que permitan identificar y especificarlos durante esta fase, así como gestionar su interacción con otros elementos arquitectónicos. Como consecuencia, el uso conjunto de ambos conceptos potencia las características de ambos.

AOSA Model considera los aspectos arquitectónicos como componentes arquitectónicos, siguiendo una aproximación asimétrica, en la que estos componentes permanecen independientes del contexto cuando se insertan en un sistema ya existente o durante el diseño de uno nuevo. El modelo propone realizar una especificación estructural de un sistema considerando los aspectos como entidades de primera clase, se basa en el uso de un lenguaje de descripción arquitectónica para expresar formalmente el diseño arquitectónico y en un modelo de coordinación. La aplicación de *AOSA Model* permitirá incluir aspectos en un sistema sin que resulten modificados los componentes del sistema en los que se realiza la inserción.

1.4.2. Definición de un LDA-OA

Uno de los puntos de interés de la investigación que se está realizando actualmente sobre aspectos tempranos (*early aspects*) es la propuesta de LDA orientados a aspectos. Esto es debido a la importancia que juegan, o deben jugar, los LDA en la expresión del diseño de los sistemas complejos:

- Como herramienta descriptiva: el DSOA proporciona mejoras en la reutilización, mantenimiento, evolución y calidad de los sistemas complejos, pero es necesario expresar adecuadamente sus especificaciones; en particular, las especificaciones arquitectónicas de los sistemas. En este sentido, los LDA-OA proporcionan el apoyo lingüístico necesario para expresar sus diseños y facilitar el paso a la implementación.
- Como soporte formal. Un LDA-OA debe proporcionar el soporte formal suficiente para asegurar la corrección de la arquitectura OA que describe: igual que un LDA convencional proporciona el adecuado soporte para razonar sobre las propiedades de las arquitecturas convencionales, los LDA-OA deben proporcionar el soporte

necesario para razonar sobre las propiedades de las arquitecturas OA en términos de componentes, conexiones, aspectos y sus vínculos con el sistema.

Como los aspectos se pueden incorporar a un sistema ya existente modificando así su comportamiento, y como esto se puede hacer de un modo transparente a los componentes que definen su arquitectura, el segundo punto adquiere especial relevancia en los objetivos de esta tesis. De hecho, cuando un aspecto se incorpora a una arquitectura existente, se produce un cambio en el comportamiento de los componentes afectados por la inserción.

En el Capítulo 6 de esta memoria se describe el LDA-OA, definido para expresar formalmente arquitecturas software de sistemas OA, obtenidas tras la aplicación del modelo que se propone.

1.5. Objetivos

El objetivo principal de esta tesis es

> *proporcionar un marco de trabajo para desarrollar sistemas complejos. Éste debe integrar la definición y especificación de un sistema software y facilitar su proceso de desarrollo, su mantenimiento y evolución. Para ello, el trabajo se basará en la utilización de técnicas y aproximaciones que permitan obtener los resultados esperados: utilizando el paradigma de DSOA, considerando el DSBC y las técnicas que proporciona la arquitectura del software como disciplina, en particular el uso de lenguajes de descripción arquitectónica.*

Este objetivo principal puede dividirse en varios objetivos específicos:

- Estudio de los trabajos relacionados en el entorno del DSOA, prestando especial atención a los realizados durante el diseño arquitectónico.
- Revisión de los principales LDA tanto convencionales como orientados a aspectos, estudio de sus características, ventajas e inconvenientes.
- Definición de un modelo que integre DSOA y DSBC durante el diseño arquitectónico de los sistemas software. El modelo obtenido debe permitir la inserción de aspectos durante el diseño y la ejecución de un prototipo de la arquitectura generada, partiendo de la especificación de los requisitos de un cierto sistema. Además, la inserción de los aspectos debe ser transparente al sistema en el que se incluyen, potenciándose así la reutilización de componentes y aspectos.
- Definición de un LDA orientado a aspectos que dote de soporte formal a las arquitecturas obtenidas tras la aplicación del modelo propuesto. Este LDA-OA debe disponer de suficiente expresividad para lograr una especificación completa del sistema. Además, se deben poder aplicar herramientas que permitan asegurar la corrección de la arquitectura.

- Definición de un metamodelo que especifique las características del modelo propuesto.
- Proposición de una metodología que sirva de guía al arquitecto del software a lo largo del proceso de desarrollo de los sistemas OA.
- Definición de una herramienta que ayude al arquitecto a realizar las tareas que define la metodología.
- El marco de trabajo desarrollado debe ser independiente de la tecnología (entendiendo por tecnología la relación hardware – sistema operativo) donde se ejecute.

1.6. Estructura de la tesis

El resto de esta memoria se organiza según los siguientes capítulos:

- **Capítulo 2. Arquitectura del software**

  En este capítulo se hace una introducción a la *Arquitectura del Software* como disciplina y su papel en el ciclo de vida del desarrollo software. Se ha creído conveniente dedicar un espacio a presentar los conceptos de este campo que se han empleado en el desarrollo de la tesis. En él se define la disciplina, sus características y los diferentes conceptos que giran en torno a ella. Además, se estudian diversos LDA.

- **Capítulo 3. Desarrollo software Orientado a Aspectos**

  Este capítulo proporciona una base conceptual para los diferentes conceptos relacionados con el paradigma de la orientación a aspectos. Se hace una revisión de diversos modelos de desarrollo que han sido propuestos siguiendo este paradigma, y resuelven el problema de la integración de aspectos en sistemas software desde las diferentes fases del ciclo de vida. El estudio se centra en aquellos trabajos realizados en el marco de la arquitectura del software; se incluye un estudio de varios LDA-OA.

- **Capitulo 4. *AOSA Model***

  Se definen en este capítulo el modelo propuesto, los conceptos y características del mismo.

- **Capítulo 5. Una metodología para *AOSA Model***

  La metodología definida para desarrollar el modelo es el objeto de este capítulo. Asimismo, se enuncia y desarrolla un caso de estudio que ayuda a aclarar los conceptos y las tareas a realizar. Además, se describe el metamodelo que representa a *AOSA Model*.

- **Capítulo 6. *AspectLEDA***

  En este capítulo se describe el LDA-OA desarrollado en el marco de esta tesis doctoral: *AspectLEDA*, que extiende un LDA convencional (LEDA). Por otra parte, se ha desarrollado una herramienta que asiste al arquitecto del software en las tareas de describir sistemas orientados a aspectos. El funcionamiento de la misma se resume convenientemente.

- **Capítulo 7. Conclusiones y trabajos futuros**

  En este capítulo se enumeran las conclusiones y contribuciones que se pueden extraer del trabajo desarrollado. Asimismo, se enumeran los trabajos en curso y los nuevos a realizar siguiendo la línea iniciada con esta tesis doctoral.

- **Anexo 1. LEDA**

  Resume el LDA convencional en el que se apoya el lenguaje orientado a aspectos que se define en este trabajo.

- **Anexo 2. EVADeS**

  Describe una herramienta diseñada para facilitar la creación, depuración y ejecución de arquitecturas descritas en LEDA. La herramienta permite realizar todas estas tareas desde un mismo entorno.

El trabajo llevado a cabo en esta tesis doctoral se ha presentado en diversos foros nacionales e internacionales, habiéndose generado varias publicaciones. Entre ellas destacan:

- [Nav+02] en la que se proponen las características que deberían tener los lenguajes de descripción arquitectónica orientados a aspectos, así como el interés de considerar modelos de coordinación para llevar a cabo las tareas de coordinación y tejido.
- En [NaPeMu04] se propone un método para diseñar un sistema gestionando independientemente los aspectos que intervienen en él. Se define una arquitectura de niveles que hace posible añadir aspectos a un sistema existente. El trabajo se centra en la definición arquitectónica de sistemas que pueden cambiar su comportamiento en tiempo de diseño cuando se añaden o eliminan restricciones, sin que cambien los componentes que lo forman. Mostramos cómo la separación de aspectos se puede gestionar durante la AS definiendo una aproximación metodológica al DSOA. Se define un estilo arquitectónico donde un meta nivel permite aplicar aspectos a componentes de diseño.
- En [Nav+04] se describe cómo afrontar el desarrollo de sistemas orientados a aspectos desde el punto de vista de la arquitectura del software, aportando dos propuestas metodológicas que se comparan.
- En [NaPeMu05a] se discute cómo se pueden tratar conjuntamente el desarrollo de sistemas durante la fase de diseño arquitectónico y los conceptos de OA. Para ello se presenta una propuesta que introduce conceptos de modelado de aspectos durante el diseño de la arquitectura de un sistema. La propuesta proporciona los pasos necesarios para permitir la evolución y el mantenimiento de los sistemas, cuando se

desea incluir nuevos requisitos en ellos, especificando una arquitectura OA. La arquitectura definida proporciona dinamismo arquitectónico permitiendo la incorporación de nuevos componentes e interacciones.

- [NaPeMu05b]. En este artículo se hace un estudio de la evolución de los sistemas, desde un punto de vista estructural considerando conceptos de separación de aspectos. Se muestran los pasos de una propuesta, en la que se aplica el modelado de aspectos a la evolución de los sistemas, durante el diseño arquitectónico, al introducir nuevos requisitos en los ya desarrollados. La propuesta es una herramienta que facilita la evolución y el mantenimiento de los sistemas, a través de la definición de una arquitectura orientada a aspectos que permite cambiar fácilmente su comportamiento, en tiempo de diseño, añadiendo restricciones sin que los componentes que lo constituyen resulten modificados (aprovechando los beneficios de aplicar el principio de inconsciencia). En el artículo se concluye que el paradigma de separación de aspectos y la arquitectura del software están relacionados directamente con la evolución, facilitando el desarrollo de los sistemas complejos. Igualmente, se puede decir que tratando ambos conceptos conjuntamente se potencian las ventajas de cada una, y que el problema de la evolución del software se puede abordar desde el punto de vista de la evolución de los requisitos, gestionándolos como un problema estructural.
- [NaPeMu07]. En este artículo se describe *AspectLEDA* como un LDA-OA en el que se propone considerar conjuntamente conceptos de arquitectura del software y de desarrollo de sistemas orientado a aspectos. Se estudian diversos modelos que tratan conjuntamente ambas disciplinas; sin embargo, no todas proponen LDA-OA. La definición del lenguaje se apoya en un modelo arquitectónico que permite incorporar aspectos observando el principio de inconsciencia.
- En [NaPeMu08] (en prensa) se describe formalmente *AspectLEDA*, el proceso de traducción a LEDA y posteriormente a Java, para la obtención, en tiempo de diseño, del prototipo de un sistema que se extiende con aspectos. Un juego de herramientas (*AOSA Tool/LEDA* y EVADeS) ayudan y guían al diseñador a lo largo de todo el proceso.

CAPÍTULO 2
Arquitectura del Software

En este capítulo se introduce el concepto de Arquitectura del Software y se definen brevemente los conceptos básicos en los que se apoya la disciplina: componentes, conexiones, estilos, abstracción, etc., para terminar detallando las características comunes de los lenguajes de descripción arquitectónica. El capítulo empieza con una serie de definiciones que aclaran estos conceptos y la evolución histórica de la disciplina. Además, se expone la relevancia de la Arquitectura del Software en el contexto del desarrollo software y de los campos en los que está enmarcada. Finalmente, se dedica un espacio a las arquitecturas dinámicas y sus características. Por su relación con el tema objeto de esta tesis se ha considerado interesante incluir, así mismo, un apartado con otros conceptos relacionados tales como la evolución de los sistemas, la reflexión arquitectónica y la norma IEEE 1471.

La inclusión de este capítulo está motivada por la necesidad de realizar un estudio en profundidad de la Arquitectura del Software y los conceptos relacionados, para desarrollar adecuadamente el modelo arquitectónico y el lenguaje que se proponen. Por tanto, el objetivo de este capítulo es presentar una base conceptual del campo y un resumen de los conceptos más interesantes.

2.1. Arquitecturas Software

La *Arquitectura del Software* (AS) es la parte de la ingeniería del software que se ocupa de la descripción y el tratamiento de un sistema como un conjunto de componentes, que facilite su organización en los diferentes subsistemas que lo forman. Esto es útil, entre otras cosas, para poder asignarlos a equipos de trabajo y que puedan llevar a cabo el desarrollo del sistema de una manera organizada y eficiente. Esta disciplina surge ante la necesidad de describir sistemas software complejos, descomponerlos en un conjunto de componentes y ver qué relaciones existen entre ellos.

El objetivo de esta sección es proporcionar una base de conocimiento mínima que sirva de apoyo al resto de la tesis. Para ello, se presenta un conjunto de definiciones de *arquitectura del software* propuestas por diversos autores [PeWo92, GaPe94, Per94, GaPe95, ShGa96]. Igualmente, se hace un resumen de su evolución histórica, desde los tiempos de Dijkstra, Parnas y Brooks, pasando por el nacimiento de esta disciplina, dónde fue denominada como tal (artículos de Perry y Wolf en 1989 [PeWo89] y publicados más tarde en 1992 [PeWo92]), hasta llegar a los conceptos más recientes: estilos arquitectónicos, lenguajes de descripción de arquitecturas y el concepto de vista arquitectónica, entre otros.

Especificar la arquitectura de un sistema software en etapas tempranas del ciclo de vida tiene muchas ventajas según la mayor parte de las metodologías de desarrollo del software actuales. La *arquitectura del software* puede definirse como el nivel del diseño del software en el que se definen tanto la estructura como las propiedades globales de un sistema, centrándose en aquellos aspectos del diseño y posterior desarrollo que no pueden tratarse adecuadamente dentro de los módulos o componentes que lo forman. Así pues, desde el punto de vista del proceso de desarrollo, la *arquitectura del software* se encarga de realizar el diseño preliminar o de alto nivel del sistema, de la organización del mismo, y de aspectos relacionados con su desarrollo y adaptación al cambio (composición, reconfiguración y reutilización); y se despreocupa, por tanto, del diseño detallado, del diseño de algoritmos o del diseño de las estructuras de datos. Se trata de una organización de alto nivel del sistema software como una colección de componentes, conexiones entre dichos componentes y cómo éstos interactúan entre sí; donde cada componente puede tomar decisiones acerca de los mensajes que envía a su entorno y reacciona cuando recibe de ese entorno un mensaje para el que está programado. Además, la AS trata de la descripción, verificación y reutilización de los sistemas software.

La descripción de la arquitectura de los sistemas software adquiere gran importancia al aumentar su complejidad. Es por ello que, dado que las expectativas de los usuarios aumentan cada vez más y la gestión de estos sistemas por parte de los diseñadores se hace más difícil, es necesario elegir una arquitectura adecuada para ellos, así como la posterior implementación de cada una de las partes. Esta definición arquitectónica es la que permitirá gestionar los sistemas complejos, centrando su estudio en los componentes que lo forman y en las relaciones que se establecen entre ellos, de modo que, al final, estas partes componentes puedan trabajar conjuntamente para resolver el sistema.

A continuación se presentan varias definiciones sobre AS que se han considerado interesantes; en las secciones siguientes se muestra un estudio general del campo de la *arquitectura del software*, así como de algunos conceptos relacionados que han sido fundamentales en el desarrollo del trabajo que aquí se presenta. Estudios más profundos y detallados se pueden encontrar en una creciente bibliografía ([ShGa96], [Sha01], [BaClKa03], [IES06], entre otras referencias).

2.1.1. Definiciones

Aunque intuitivamente el concepto de AS es bastante claro, no se ha dado hasta el momento una definición que satisfaga por completo a todos aquellos que trabajan en este campo. Quizás, para obtener una visión de conjunto, lo más adecuado sea considerar varias definiciones a la vez. A continuación se mencionan varias de ellas ordenadas cronológicamente:

* Posiblemente la primera definición fue la dada en 1993 por David Garlan y Mary Shaw [GaSh93]:

> *Más allá de los algoritmos y estructuras de datos de la computación, el diseño y especificación de la estructura global del sistema emerge como un nuevo tipo de problema. Los detalles estructurales incluyen: la organización general y la estructura de control global; los protocolos de comunicación, sincronización y acceso a datos; la asignación de funcionalidad a los elementos de diseño; la distribución física; la composición de los elementos de diseño; su escalabilidad y los aspectos de rendimiento; y la selección entre alternativas de diseño.*

En este párrafo, más que definir detalladamente qué es la AS, se da una justificación de la (en aquella época emergente) disciplina. Esta definición incluye la exposición de los objetivos y aspectos a tener en cuenta en la AS, aunque en muchos de los desarrollos realizados hasta ahora sólo se han tenido en cuenta de forma parcial.

* Se puede considerar una segunda definición, quizás la más breve y sencilla, y a la vez la más aceptada, dada por David Garlan y Dewayne Perry [GaPe95]:

> *La arquitectura del software está compuesta por la estructura de los componentes de un programa o sistema, sus interrelaciones, y los principios y reglas que gobiernan su diseño y evolución a lo largo del tiempo.*

* A continuación se presenta la definición dada por Clements [Cle96] que puede considerarse como una definición general, amplia y flexible, aunque por ello pueda parecer incompleta y algo ambigua. Sin embargo, es válida tanto para aquellos que desean una visión descriptiva de la AS (punto de vista representado por Garlan) como para los que utilizan una visión de proceso (representada por Perry):

> *La arquitectura del software es, a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se percibe desde el resto del sistema, y las formas en que los componentes*

> *interactúan y se coordinan para alcanzar el objetivo del sistema. La vista arquitectónica es una vista abstracta de un sistema, aportando el más alto nivel de compresión y la supresión del detalle inherente a la mayor parte de abstracciones*.

* En el año 2000 se acordó definir oficialmente la AS según figura en el documento IEEE Std 1471-2000 [IEEE]:

> *La arquitectura del software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el entorno, y los principios que dirigen su diseño y evolución.*
>
> Si se compara esta definición con la de Garlan y Perry, la de IEEE es más precisa.

* En el número de marzo/abril de 2006 de IEEE Software, en el artículo de Kruchten, Obbink y Stafford [KrObSt06] figura la siguiente definición, más elaborada:

> *La arquitectura del software captura y preserva las intenciones de los diseñadores sobre la estructura y el comportamiento de los sistemas, proporcionando un mecanismo de defensa contra el deterioro de los sistemas antiguos. Esta es la clave para lograr el control intelectual sobre la creciente complejidad de los sistemas….*
>
> *… Paradójicamente… aún no hay un consenso sobre cual es la respuesta a la pregunta de qué es la arquitectura del software y que sea ampliamente aceptada… Llegar a un acuerdo sobre la definición más adecuada para esta disciplina fue uno de los objetivos de definir el estándar IEEE. Sin embargo, esta falta de acuerdo no ha sido impedimento para que la arquitectura del software progrese.*

A partir de todas las definiciones que se han dado sobre AS, se ha llegado a una especie de consenso por el que esta disciplina se refiere a

> *la estructura de un sistema a grandes rasgos, formada por componentes y relaciones entre ellos* [BaClKa03].

Estas cuestiones se tienen en cuenta durante el diseño, puesto que la AS se refiere al diseño del software que se realiza en fases tempranas del desarrollo software y a alto nivel de abstracción.

> Resumiendo se puede decir que la AS
>
> *permite realizar el diseño de alto nivel del sistema, definir su organización como la descripción y análisis de propiedades relativas a su estructura, los protocolos de comunicación, su distribución física y de sus componentes, etc., además de los aspectos relacionados con el desarrollo del sistema y su adaptación al cambio. Es decir, su composición, reconfiguración y reutilización*.

Por otra parte, la AS puede considerarse como un puente entre los requisitos y el código: desempeña un papel fundamental entre la especificación de los requisitos de un sistema y la implementación del mismo. Al realizar la descripción abstracta del sistema se representan una serie de características, mientras otras se ocultan, proporcionando así una guía para que los arquitectos o diseñadores puedan sugerir un modelo de construcción en función de los requisitos.

En esta tesis, recogiendo las definiciones anteriores, la AS se considera como

un conjunto formado por la organización, la estructura y la infraestructura de un sistema software.

En este sentido, se describe un modelo arquitectónico para sistemas software, pudiéndose realizar una evaluación y un análisis del diseño obtenido. Además, asociado al modelo se proporciona una descripción lingüística formal del diseño. Por otra parte, el modelo arquitectónico que se propone ayuda a planificar y a gestionar la evolución de los sistemas.

2.1.2. Evolución histórica de la arquitectura del software

Cuando se estudia la evolución histórica de la AS pueden mencionarse varios precedentes. Se puede empezar citando a Dijkstra cuando, en 1968, propuso que

debería establecerse una correcta estructuración de los sistemas software antes de comenzar a programar [Dij68].

Él fue quién planteó considerar niveles de abstracción para resolver los problemas del software y cuya aplicación a la AS ha resultado de gran interés. Estos trabajos fueron referenciados después en la conferencia de la OTAN de 1969 [NATO70] sobre técnicas de ingeniería del software.

En 1969 Brooks e Iverson [BrIv69] definieron la arquitectura como *la estructura conceptual de un sistema*, pero desde la perspectiva de la implementación.

Durante la década de los 70 surge el diseño estructurado y las primeras investigaciones académicas en materia del diseño de sistemas complejos. Los trabajos que surgen en esta época sobre una consideración software de la arquitectura de un sistema se deben a Parnas [Par72] y Brooks [Bro95], entre otros. David Parnas demostró que los criterios que se siguen en la descomposición de un sistema tienen un gran impacto sobre la estructura de los programas, y propuso diversos principios de diseño que debían seguirse a fin de obtener una estructura adecuada. Él desarrolló temas como módulos con ocultamiento de información en los que aparecen conceptos como *encapsulación y abstracción,* y la idea de módulos como cajas negras de las cuales sólo se conoce la interfaz; todo esto, sin haber llegado aún al concepto de objeto. Parnas ha sido el introductor de algunos de los conceptos esenciales de la AS:

> *... Las decisiones tempranas en el desarrollo serían las que probablemente permanecerían invariantes en el desarrollo ulterior de una solución…*

Esas *decisiones tempranas* constituyen de hecho lo que hoy se llaman *decisiones arquitectónicas*. Como escribirían más tarde Clements y Northrop [ClNo96]:

> *La estructura es primordial, y la elección de la estructura correcta es crítica para el éxito del desarrollo de una solución*.

Según esto, no cabe duda que *la elección de la estructura correcta* representa la razón de ser de la arquitectura del software.

Durante la década de los ochenta, los métodos estructurados demostraron no ser suficientes para abordar el desarrollo de los sistemas que demandaban los usuarios, y estos métodos fueron dejando paso a nuevos paradigmas como la programación orientada a objetos. Paralelamente, hacia finales de la década, la AS empieza a aparecer en la literatura haciendo referencia a la *configuración morfológica* de una aplicación. Mary Shaw, en [Sha84] y [Sha89], vuelve a reivindicar las abstracciones de alto nivel, reclamando un espacio para esa reflexión y augurando que el uso de esas abstracciones en el proceso de desarrollo puede resultar en *un nivel de arquitectura del software en el diseño*, y por otro lado, que *el desarrollo de grandes sistemas requiere abstracciones de alto nivel*. En sus trabajos se refiere a un nivel de abstracción en el conjunto; aunque todavía no se tenían elementos de juicio que permitieran reclamar la necesidad de una disciplina particular.

A principio de los noventa se menciona por primera vez la AS en el sentido en el que se conoce actualmente, y surge realmente como disciplina cuando se publican los trabajos de Royce y Royce [RoRo91], Rechtin [Rec91], Kruchten [Kru91] y destacando el artículo de Perry y Wolf [PeWo92]. Este trabajo se suele considerar como punto de partida del actual auge de la AS. Puede decirse que estos autores fundaron la disciplina, siendo seguida en primera instancia por los miembros de lo que podría llamarse la *escuela estructuralista* de Carnegie Mellon, de la que forman parte David Garlan, Mary Shaw, Paul Clements y Robert Allen.

Para determinar lo que sucedió en esta década se cita una frase de Perry y Wolf, que ha quedado inscrita en la historia de esta disciplina:

> *... La década de 1990, creemos, será la década de la arquitectura del software. Usamos el término "arquitectura" en contraste con "diseño" para evocar nociones de codificación, de abstracción, de estándares, de entrenamiento formal (de los arquitectos del software) y de estilo. … Es tiempo de re-examinar el papel de la arquitectura del software en el contexto más amplio del proceso de software y de su administración, así como señalar las nuevas técnicas que se han adoptado…*

En 1995, la AS era ya una realidad y surgen numerosos trabajos y propuestas de desarrollo como SAAM [Kaz+94] o la de Kruchten [Kru95], con el modelo arquitectónico de 4+1 vistas, entre otros autores.

Durante toda la década, la actividad, el desarrollo y el interés de la AS siguieron aumentando. Muestra de ello es que la AS se ha integrado como una parte fundamental de

la Ingeniería del Software, incorporándose a las sucesivas ediciones de la Conferencia Internacional sobre Ingeniería del Software (ICSE). Además, en 1999 se creó una conferencia específica sobre arquitectura del software (WICSA). Sin embargo, aún se trata de una disciplina joven que se encuentra en un momento de auge al estar surgiendo continuamente nuevas técnicas y métodos, como los propuestos por Kazman, Klein, Bass y otros investigadores del SEI (Software Engineering Institute de Carnegie Mellon) [BaClKa03, ClKaKl02, Cle02].

El trabajo del Software Engineering Institute es el origen del desarrollo de nuevas metodologías ligadas al ciclo de vida en términos arquitectónicos. En este sentido, se está trabajando en establecer distintas modalidades de análisis, diseño, verificación, refinamiento, recuperación, diseño basado en escenarios, estudios de casos y hasta justificación económica. Durante los últimos años han surgido varios grupos de trabajo que realizan sus investigaciones en diversas áreas de la AS:

- Por una parte, a medida que la AS se ha ido afianzando como disciplina y popularizándose sus conceptos y técnicas de desarrollo, surge la necesidad de estandarizar las ideas del nuevo campo. En 1995, el IEEE comienza a trabajar en este tema, que culmina con la publicación en 2000 de la Norma Recomendada RP-1471 [IEEE]. Trata de homogeneizar y ordenar la nomenclatura de la descripción arquitectónica y homologa los estilos como un modelo fundamental de representación conceptual (sección 6 de este capítulo).
- Por otra parte, surgen proyectos, en particular en Estados Unidos, sobre todo en el ámbito de la investigación militar: el proyecto START (Software Technology for Adaptable, Reliable Systems) introdujo el concepto de arquitectura del software. Posteriormente, el proyecto EDCS (Evolutionary Design of Complex Systems) se centra en el concepto de arquitectura, y en el que participan empresas y universidades. Su objetivo es dar una visión de los sistemas complejos como entidades sometidas a continuos cambios. Así, la AS juega el papel de elemento unificador en este contexto.
- Con el fin de incrementar la práctica de la descripción arquitectónica, el Open Group publica la primera versión de una norma para la descripción arquitectónica, denominada ADML (Architecture Description Markup Language), basado en XML, (apartado 2.4).
- En Europa, diversos grupos también desarrollan sus investigaciones en el ámbito de la arquitectura del software.

Otros investigadores que se debe mencionar son Nenad Medvidovic, David Rosenblum y Richard Taylor en Irvine y Los Ángeles, Mark Moriconi y su grupo en el SRI (Computer Science Laboratory, SRI International) de Menlo Park, o Roy Fielding.

Como conclusión de esta breve introducción histórica se puede decir que, durante estos 40 años, la arquitectura del software se ha convertido en una auténtica disciplina dentro de la ingeniería del software cuya importancia ha ido incrementándose a medida que transcurría el tiempo. Además, se ha ido homogeneizando su terminología, se han establecido los distintos estilos arquitectónicos y se han descrito diversos lenguajes de descripción de arquitecturas (LDA). Finalmente, el concepto de vista arquitectónica se ha consolidado en los últimos años.

Figura 2.1 ha sido extraída de [IES06] y muestra los hitos más importantes en la evolución de la arquitectura del software desde el año 1992 cuando se publicó el artículo de Perry y Wolf.

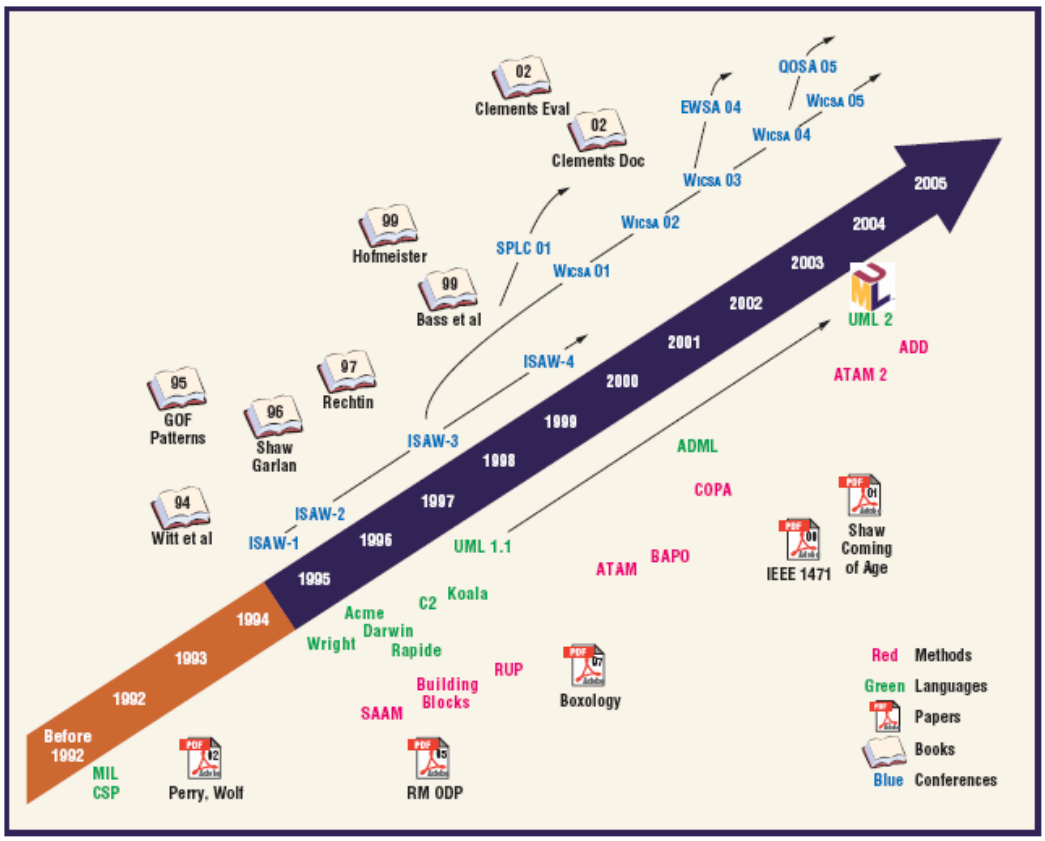


Figura 2.1. Resumen de la evolución de la Arquitectura del Software. Hitos relevantes.

- **Relevancia de la arquitectura del software**

La aplicación de los conceptos y técnicas de AS al desarrollo de sistemas software ha sido muy beneficiosa en la mayoría de los casos, reduciendo costos, evitando errores, encontrando fallos, reduciendo el esfuerzo en mantenimiento posterior, etc., de tal modo que hoy nadie duda de la necesidad de considerar la visión arquitectónica durante el desarrollo; si un proyecto no ha logrado definir una arquitectura para el sistema, el proyecto no debería continuar su desarrollo.

Las ventajas de la arquitectura del software [ClNo96, Gar+00] son:

· *Comunicación mutua*: la AS representa un alto nivel de abstracción común que la mayoría de los participantes en un desarrollo pueden usar como base para crear una vía de comunicación entre ellos. La descripción arquitectónica expone las restricciones de alto nivel sobre el diseño del sistema, así como la justificación de decisiones arquitectónicas fundamentales.

- *Decisiones tempranas de diseño*: la AS representa las decisiones de diseño más tempranas sobre un sistema; esos vínculos tempranos tienen un gran peso sobre el resto del desarrollo, y su posterior mantenimiento.
- *Restricciones constructivas*: una descripción arquitectónica proporciona diseños de alto nivel para el desarrollo, mostrando los componentes principales y las dependencias entre ellos.
- *Reutilización o abstracción transferible de un sistema*: la AS representa un modelo relativamente pequeño de la estructura de un sistema y refleja cómo se entienden entre sí sus componentes; este modelo es aplicable a otros sistemas que tengan requisitos parecidos y puede promover la reutilización a gran escala. El diseño arquitectónico soporta reutilización de grandes componentes o incluso de marcos de trabajo (*frameworks*) en los que aquellos se pueden integrar.
- *Evolución*: la AS permite planificar la dirección en la cual se puede esperar que evolucione un sistema. Esto ayuda a identificar las partes que pueden cambiar y las que no lo van a hacer.
- *Análisis*: las descripciones arquitectónicas ayudan a comprender los requisitos facilitando las verificaciones de consistencia del sistema, la conformidad con las restricciones impuestas por un estilo, la conformidad con atributos de calidad, el análisis de dependencias y análisis específicos de dominio y de negocio.
- *Administración*: la evaluación crítica de una arquitectura conduce a una comprensión más clara de requisitos, estrategias de implementación y riesgos potenciales.

El conocimiento adquirido sobre la AS ha quedado recogido en numerosos libros y artículos como los mencionados anteriormente. Por su especial interés y trascendencia hay que reseñar el de Mary Shaw y David Garlan *Software Architecture: Perspective on an Emerging Discipline* [ShGa96]. La obra sitúa a la AS como una auténtica disciplina dentro del marco de la ingeniería del software, distinta del diseño y la implementación.

La experiencia alcanzada tras los años de utilización de la AS permite afirmar que la realización de un estudio sobre la arquitectura software de un sistema facilita el éxito de los proyectos: al realizar una descripción abstracta del mismo se exponen ciertas características y se ocultan otras; esta descripción sirve de guía del sistema; con ella los diseñadores pueden razonar sobre ciertos requisitos y sobre qué modelo usar para la construcción del mismo.

2.1.3. Conceptos fundamentales de la arquitectura del software de un sistema

En este apartado se explican los conceptos fundamentales de la arquitectura del software. Desde el punto de vista lingüístico, una arquitectura viene determinada por unos *componentes* -elementos básicos caracterizados por una *interfaz* segmentada en *puertos* y *conectores* que la constituyen-, y por una serie de *conexiones* o *enlaces* específicos -que definen la relación de los componentes formando una estructura-. A esta estructura se le da el nombre de *configuración,* que a veces se organiza en *vistas*, cada una de las cuales se centran en un aspecto diferente. Cuando interesa obtener los patrones genéricos que

definen a una familia de sistemas y no una configuración concreta, se habla de *estilos arquitectónicos* ([Gar95, BaClKa03]).

La definición de todos estos conceptos es lo que da una visión amplia de lo que se entiende por arquitectura del software. En los siguientes apartados se describe cada uno de ellos, dedicándole mayor atención a los lenguajes de descripción arquitectónica.

2.1.3.1. Componentes arquitectónicos

El concepto fundamental de la *arquitectura del software* es el de *componente*: un *componente arquitectónico* es cada una de las partes o unidades de composición en las que se divide la funcionalidad de un sistema y cuya unión constituye el sistema completo.

Aunque la idea intuitiva de *componente* es fácil de comprender, es más difícil dar una definición que sea ampliamente aceptada. Sin embargo, parece que la siguiente, debida a Szyperski, sí ha alcanzado un cierto consenso [Szy98]:

> *Un componente es una unidad de composición de aplicaciones software, que tiene un conjunto de interfaces y un conjunto de requisitos, que ha de poder desarrollarse e incorporarse a un sistema, y componerse con otros de forma independiente, en tiempo y espacio.*

La definición es lo suficientemente general para poder aplicarla al contexto que nos ocupa. En este sentido, un *componente* es un elemento arquitectónico fuertemente encapsulado y con una interfaz bien definida, lo que le da independencia del resto del sistema en el que se integra mediante mecanismos de composición e interacción.

Un *componente*, que se relaciona con el resto del sistema mediante la interfaz que proporciona y que necesita, se puede tratar como una caja negra en la que la aplicación de los principios de encapsulamiento y ocultación de la información resultan especialmente interesantes. Este concepto facilita la evolución de los sistemas al ser sencillo cambiar un componente por otro que lo sustituya. Sin embargo, a veces es necesario considerar ciertos elementos que faciliten la adaptación de los componentes. Estos elementos adaptadores se suelen representar a nivel arquitectónico mediante conectores específicos.

2.1.3.2. Conectores o conexiones arquitectónicas

Se denomina *conector* a cualquier dispositivo capaz de comunicar dos o más elementos y permiten modelar las interacciones entre componentes. El concepto de *conector* arquitectónico fue introducido por Mary Shaw [ShGa94] quién propuso separar la funcionalidad de un sistema (*componentes*) de su interacción (*conectores*). Los primeros se encargaran de realizar su tarea sin preocuparse de cómo se relacionan, mientras que los segundos se encargan de resolver la comunicación de los primeros. De este modo se plantea una *separación de intereses* que eleva el nivel de abstracción durante el desarrollo de un sistema, a la vez que aumenta su modularidad. Shaw, en su propuesta, confiere a los conectores entidad propia a nivel arquitectónico (elementos de primera clase), lo que permite su utilización en diferentes contextos.

Esta definición independiente de los *conectores* respecto a los componentes que relaciona obliga a definir dos tipos de vínculos entre distintos elementos de una

descripción arquitectónica. Por una parte el propio *conector* que expresa la posibilidad de interacción entre varios componentes. Por otra parte es necesario establecer un vínculo entre un componente y el *conector* que se le asocia; este enlace se suele llamar conexión (*attachment*).

2.1.3.3. Puertos

El concepto de *puerto* es cercano al de conector, pero no se debe confundir. Se denomina *puerto* a cada uno de los puntos por los que un componente puede realizar cualquier tipo de interacción. Se puede decir que *es cada uno de los segmentos en los que se fragmenta la interfaz de un componente*. Hace referencia a un punto de entrada o de salida de la caja negra que se considera el componente. Según esto, los *puertos* se refieren tanto a los servicios que éste oferta como a los que precisa. Por tanto, los *puertos* configuran la naturaleza externa de los componentes, lo que a su vez condiciona la estructura de la arquitectura.

2.1.3.4. Vistas

El concepto de *vista,* que ha sido el último propuesto de manera explícita, permite que una arquitectura software pueda considerarse desde diferentes perspectivas, de manera que cada una de estas arquitecturas 'parciales' constituye una *vista;* la visión completa del sistema se adquiere por combinación de todas ellas. Muchas veces se compara este concepto con el de los diversos planos que, desde diversas perspectivas, permiten definir de modo completo un cierto edificio.

Históricamente, los primeros en sugerir un enfoque *multi-vista* fueron Perry y Wolf [PeWo92] que ya desde el principio propusieron que una arquitectura debería tener una *vista* de flujos, otra de control y otra de recursos. Pero sin duda la propuesta más difundida es el denominado *Modelo 4+1* de Kruchten [Kru95], que ha sido posteriormente popularizada por su adaptación a UML y su difusión dentro del campo de la orientación a objetos. Este modelo distingue entre cinco *vistas* que se pueden aplicar al diseño arquitectónico. Estas son: la vista lógica, la de desarrollo, la de proceso, la vista física, y la vista de casos de uso. Cada una se refiere a un conjunto de características: la vista lógica soporta los requisitos funcionales y describe el modelo de objetos; la vista de desarrollo soporta también la especificación de los requisitos funcionales y la organización estática del sistema; la vista de proceso y la vista física consideran los requisitos no funcionales y se refieren a aspectos como la concurrencia, la distribución o la tolerancia a fallos; la vista de casos de uso incluye los casos de uso y los escenarios para definir y validar la arquitectura.

La Norma Recomendada RP-1471 del IEEE [IEEE] destaca la importancia del concepto de *vista* adoptando explícitamente un enfoque *multi-vista* para la AS.

2.1.3.5. Abstracción

Los distintos significados del concepto de *abstracción* ayudan a identificar las distintas corrientes existentes dentro de la AS. Se puede considerar la *abstracción* como la extracción de las propiedades esenciales o la identificación de los aspectos importantes

de un problema, para poder examinar selectivamente ciertas características del mismo, posponiendo el estudio de aquellos detalles menos relevantes. En esta definición están de acuerdo el IEEE y autores como Rumbaugh y Shaw, entre otros. Además, investigadores como Bass, Clements y Kazman [BaClKa03] opinan que si una decisión debe posponerse hasta el momento de tratar las cosas a bajo nivel entonces no se trata de una decisión de arquitectura. Según Clements y Northrop [ClNo96], el estudio de Garlan y Shaw sobre estilos arquitectónicos muestra que, aunque los programas puedan combinarse de infinitas formas, se pueden obtener ventajas si el numero de combinaciones se restringe a un conjunto relativamente pequeño en cuanto se trata de cooperación e interacción. Estas ventajas son: mejor reutilización, mejor análisis, menor tiempo de selección y mejor operabilidad.

2.1.4. Campos de la arquitectura del software

La arquitectura del software abarca un conjunto de áreas de investigación teórica y de formulación práctica. Garlan y Perry [GaPe95] le asignan los siguientes campos de desarrollo:

- Lenguajes de descripción de arquitecturas.
- Fundamentos formales de la AS (bases matemáticas, caracterizaciones formales de propiedades extra-funcionales).
- Técnicas de análisis arquitectónico.
- Métodos de desarrollo basados en arquitectura.
- Recuperación y reutilización de arquitectura.
- Codificación y guía arquitectónica.
- Herramientas y entornos de diseño arquitectónico.
- Casos de estudio.

De entre todos ellos, el ámbito de la reutilización es uno de los fundamentales y que más justifican la arquitectura del software. Los estudios actuales tratan de conseguir un almacenamiento estructurado de este tipo de información reutilizable, que es información de diseño de alto nivel propia de una familia de sistemas.

La relación anterior se puede comparar con los cincos temas que propone Clements [Cle96] en torno a los que agrupa la disciplina:

- Diseño o selección de la arquitectura: indica cómo crear o seleccionar una arquitectura en función de los requisitos funcionales, de rendimiento o de calidad.
- Representación de la arquitectura: se refiere a cómo comunicar una arquitectura, utilizando recursos lingüísticos. También incluye la selección del conjunto de información a ser comunicada.
- Evaluación y análisis de la arquitectura: trata de cómo analizar una arquitectura para predecir el comportamiento del sistema. Un problema semejante es cómo comparar y escoger entre diversas arquitecturas posibles.
- Desarrollo y evolución de sistemas basados en arquitectura: relativo a cómo construir y mantener un sistema, dada una representación arquitectónica que se cree que es la que resolverá el problema.

· Recuperación de la arquitectura: define cómo hacer que un sistema evolucione cuando los cambios afectan a su estructura; para los sistemas de los que se carezca de documentación, esto supone realizar primero una *minería arquitectónica* que extraiga su arquitectura.

Con todo esto, Mary Shaw considera que [Sha01]:

Los campos más prometedores de la AS son los que tienen que ver con el tratamiento sistemático de estilos, el desarrollo de los lenguajes de descripción arquitectónica y la formulación de metodologías, y el trabajo con patrones de diseño. Se necesitan modelos precisos que permitan razonar sobre las propiedades de una arquitectura y verificar su consistencia y completitud.

Se pueden considerar distintos enfoques asociados a las descripciones arquitectónicas que han dado lugar a modelos arquitectónicos con distintas características. Algunos autores los clasifican de la siguiente manera:

1) *Modelos estructurales*: sostienen que la arquitectura del software está formada por componentes, conexiones entre ellos y, normalmente, otros elementos, como pueden ser configuración, estilo, restricciones, semántica, análisis, propiedades, requisitos, etc. El trabajo en este área se caracteriza por el desarrollo de lenguajes de descripción de arquitecturas (LDA).
2) *Modelos de marcos de trabajo*: son similares a los anteriores, pero se centran más en la estructura coherente del sistema completo que en su composición. Estos modelos a menudo se refieren a dominios específicos.
3) *Modelos dinámicos*: dinámico aquí se refiere a la posibilidad de cambiar la configuración del sistema o la dinámica involucrada en la computación. Se centran más en la conducta o el comportamiento.
4) *Modelos de proceso*: están centrados en la construcción de la arquitectura y en el proceso involucrado en dicha construcción. Aquí, la arquitectura es el resultado de seguir determinadas pautas.
5) *Modelos funcionales*: la arquitectura se considera como un conjunto de componentes funcionales, organizados en capas, que proporcionan servicios hacia las capas superiores.

En esta clasificación, los enfoques no son excluyentes entre sí.

2.1.5. Estilos arquitectónicos

Este es uno de los conceptos más importantes de la arquitectura del software. Se puede definir un *estilo arquitectónico* como

la descripción de una familia de sistemas, en términos de un patrón de organización estructural [ShGa94].

De ese modo, un *estilo* determina un vocabulario común de componentes y conectores que se pueden usar para el desarrollo de un sistema y las instancias de dicho

estilo, junto con una serie de propiedades sobre cómo se pueden combinar. En contraposición a este concepto, se define el de *configuración* como

> *la arquitectura del software de un sistema concreto; es decir, la estructura específica constituida por unos conectores y componentes específicos*.

De este modo, una *configuración* es una instancia de un *estilo*.

Por otra parte, un *estilo* describe las propiedades básicas de una arquitectura e impone los límites de su evolución (lo que se relaciona con el dinamismo de la arquitectura).

En 1999 Klein y Kazman [KlKa99] propusieron una definición según la cual:

> *Un estilo arquitectónico es una descripción del patrón de los datos y la interacción de control entre los componentes, ligada a una descripción informal de los beneficios e inconvenientes aparejados por el uso del estilo.*

Los *estilos* arquitectónicos, afirman,

> *Son artefactos de ingeniería importantes porque definen clases de diseño junto con las propiedades conocidas asociadas a ellos.*

Ofrecen evidencia basada en la experiencia sobre la forma en que se ha utilizado históricamente cada clase de diseño, junto con un razonamiento cualitativo para explicar por qué cada clase tiene esas propiedades específicas.

Según Garlan [GaSh93] la especificación de un *estilo* define:

- Un vocabulario de tipos de componentes y conectores de los que se dispone.
- Un conjunto de restricciones sobre cómo se pueden combinar los componentes y conectores del estilo.
- Opcionalmente un modelo semántico que determine cómo se pueden deducir las propiedades globales del sistema, a partir de las propiedades individuales de cada uno de sus elementos.

Actualmente se asume que cualquier descripción de un *estilo arquitectónico* se debe basar en la combinación de estos tres elementos, y sólo en ellos. Sin embargo, la mayoría de los LDA no capturan la idea de *estilo* arquitectónico; sólo Wright [All97] y Acme [GaMoWi00] permiten, en cierto modo, su definición de un modo general.

El estudio de los *estilos* ha sido abordado desde un punto de vista informal, identificándose un gran número de ellos y sus características. La primera taxonomía la propusieron Shaw y Garlan [GaSh93], considerando los siguientes tipos:

- Tuberías-filtros.
- Organización de abstracción de datos y orientación a objetos.
- Invocación implícita, basada en eventos.
- Sistemas en capas.
- Repositorios.
- Procesos distribuidos, ya sea en función de la topología (anillo, estrella, etc.) o de los protocolos entre procesos. Una forma particular de proceso distribuido es, por ejemplo, la arquitectura cliente-servidor.

- Organizaciones programa principal / subrutina.
- Arquitecturas software específicas de dominio.
- Sistemas de transición de estados.
- Sistemas de procesos de control.
- Estilos heterogéneos.

Posteriormente, el grupo de Buschmann denominó a los *estilos* como *patrones arquitectónicos*. Esos patrones

> *expresan esquemas de organización estructural fundamentales para los sistemas software. Proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen guías para organizar las relaciones entre ellos*.

Así, en [Bus+96] se enumeraron estos patrones, agrupándolos en: s*istemas estructurales, distribuidos, interactivos y sistemas adaptables*. Posteriormente, en *Software Architecture in Practice* [BaClKa03] se definió un conjunto de clases de estilo en los siguientes grupos: *flujo de datos*, *llamada y retorno componentes independientes*, *centrados en datos y máquina virtual.*

En los últimos años la relación de *estilos* se ha hecho más detallada y exhaustiva. Considerando solamente los estilos que contemplan alguna forma de distribución o topología de red, Roy Fielding [Fie00] propuso una nueva taxonomía: *estilos de flujo de datos, de replicación, jerárquicos, de código móvil, estilo peer-to-peer y transferencia de estado de representación (REST).*

Como se puede apreciar, hay una gran variedad en la clasificación de los estilos, según la época y el autor. Sin embargo, aquí no se profundiza más en este concepto ni se describen los estilos mencionados, ya que, aún siendo un concepto fundamental en la AS, su relación con el tema de esta tesis es, en cierto modo, indirecta.

2.2. Lenguajes de Descripción Arquitectónica

Desde el principio de la arquitectura del software surgió la necesidad de disponer de una notación específica para describirla, separando aspectos relativos a la estructura del sistema de otros detalles del desarrollo. Estas notaciones han ido evolucionando y recibiendo distintos nombres, siendo la denominación más utilizada actualmente la de *Lenguajes de Descripción Arquitectónica* o *Lenguajes de Descripción de Arquitecturas* (LDA) que se define como

> *Una notación que describe y analiza formalmente las propiedades observables de una arquitectura software, dando soporte a distintos estilos arquitectónicos a diferentes niveles de abstracción* [Sch99].

La gran ventaja de usar un LDA sobre otras notaciones informales es que permiten una mejor comunicación entre el diseñador, implementadores y lectores debido a que la formalidad no deja lugar a la ambigüedad. Además permite el análisis formal y temprano de las decisiones de diseño; es decir, una vez que el arquitecto conoce los

requisitos de un sistema, decide su estrategia de diseño. A partir ahí, debe expresar sus características y modelarlo aplicando una convención gráfica o algún lenguaje avanzado de alto nivel de abstracción. Los LDA son lenguajes que permiten modelar la arquitectura del sistema, analizar si es adecuada, determinar sus puntos críticos e, incluso, simular su comportamiento. Y todo esto antes de abordar la implementación.

En esta sección se presenta un breve estudio sobre algunos LDA y sus características; su interés reside en que proporcionan una sintaxis para definir los elementos arquitectónicos. Hasta las clasificaciones publicadas por Shaw y Garlan [ShGa94], Kogut y Clements [KoCl94], y Medvidovic y Taylor [MeTa97], existía poco consenso respecto al significado de un LDA, qué aspectos de la arquitectura se deben modelar con un LDA, y cuáles de ellos son más adecuados para representar qué clase de arquitectura o estilo. Por otra parte, la generación de prototipos ejecutables a partir de estas especificaciones es un objetivo para alguno de ellos.

Según Shaw y Garlan los LDA deberían cumplir una serie de propiedades [ShGa94]:

- *Composición*: un lenguaje debe permitir la representación de un sistema como composición de una serie de partes –componentes- y la comunicación entre ellas –conectores-. Igualmente deben permitir la definición de restricciones de diseño en la composición del sistema para dejar claro qué tipo de composiciones se permiten.
- *Configuración*: la descripción de la arquitectura debe ser independiente de la de los componentes que forman el sistema. Además, los LDA deben permitir realizar descripciones jerárquicas y encapsular subsistemas como componentes en sistemas grandes.
- *Abstracción*: mediante la cual se describen los roles o papeles abstractos que juegan los componentes dentro de la arquitectura.
- *Flexibilidad*: que permita la definición de nuevas formas de interacción entre componentes.
- *Reutilización*: tanto de los componentes como de la propia arquitectura.
- *Heterogeneidad*: que permita combinar descripciones heterogéneas.
- *Análisis* de la arquitectura y de los sistemas desarrollados a partir de ella; incluso con la capacidad de generar prototipos del sistema.

Además de estas propiedades, los LDA deberían permitir la representación de propiedades no funcionales para el análisis del sistema y la definición de estilos arquitectónicos.

Aunque hay otras notaciones y formalismos que también se utilizan como los LDA para la descripción de arquitecturas (CHAM, UML y Z), sólo los LDA proporcionan construcciones para especificar abstracciones arquitectónicas y mecanismos para descomponer un sistema en componentes y conectores; describen de qué manera estos elementos se combinan para formar configuraciones y definen familias de arquitecturas o estilos, evitando los detalles de la implementación de los módulos concretos. Si un arquitecto utiliza un LDA puede razonar sobre las propiedades del sistema de manera precisa, sin abandonar un nivel de abstracción que sea lo suficientemente alto. Su objetivo general es determinar los puntos críticos del desarrollo y, eventualmente, simular su comportamiento.

Las ventajas de un cierto LDA vienen dadas por su poder expresivo para especificar la estructura y el comportamiento, pero también por su forma de uso, la funcionalidad, el rendimiento, la flexibilidad, la capacidad de reutilización, la facilidad de comprensión y las restricciones tecnológicas. La mayoría de los LDA se complementan mediante herramientas: compiladores, comprobadores de restricciones, simuladores, etc.; herramientas casi siempre desarrolladas para trabajar con un LDA específico.

Por último, para que un LDA permita la representación de la arquitectura de los sistemas software debería incluir las siguientes características:

- *Dinamismo*: su objetivo es hacer explícitos los aspectos arquitectónicos del software para facilitar el desarrollo y evolución de sistemas complejos. Debido al carácter inherentemente dinámico y reconfigurable de muchos de estos sistemas, la capacidad de describir los aspectos que rigen la evolución de una arquitectura es un requisito básico de cualquier LDA. Las arquitecturas dinámicas admiten la replicación, inserción, eliminación y reconexión de sus componentes en tiempo de ejecución.
- *Verificación de propiedades*: la notación utilizada para describir un sistema en un cierto LDA debe basarse en un formalismo que le sirva de soporte para la comprobación de las propiedades de dicho sistema en las etapas iniciales del desarrollo, como es el diseño arquitectónico, reduciendo substancialmente el coste de los errores. Un modelo formal proporciona una definición precisa de una arquitectura de software.
- *Desarrollo del sistema y reutilización*: el proceso de desarrollo de cualquier sistema de software de cierta complejidad pasa normalmente por una serie de refinamientos sucesivos en los que el sistema se representa a diferentes niveles de abstracción, que lo van llevando progresivamente desde la especificación a la implementación. Un buen LDA debe proporcionar mecanismos para el refinamiento de la arquitectura y sus componentes que faciliten este proceso de desarrollo.

Wolf en [Wol97] propuso que, como mínimo, un lenguaje debe poder describir: componentes, conectores, configuraciones y restricciones, y además debe cumplir los siguientes requisitos:

- Ayudar en la definición de la arquitectura y en las tareas de refinamiento y validación.
- Definir las reglas sobre las que se construye una arquitectura completa.
- Tener la capacidad de representar la mayoría de los estilos arquitectónicos.
- Tener la capacidad de proporcionar vistas del sistema que expresen la información arquitectónica, a la vez que omita aspectos relacionados con la implementación del sistema.

2.2.1. Conceptos

Los conceptos fundamentales de los LDA son componentes, conectores y configuraciones. Los tres se describen brevemente a continuación.

2.2.1.1. Componentes

El concepto y la definición de *componente arquitectónico* dadas en el apartado 2.1.3 es válido en el ámbito de los LDA, como no podía ser de otra manera. Allí se decía que *un componente es un elemento arquitectónico fuertemente encapsulado y con una interfaz bien definida, lo que le da independencia del resto del sistema en el que se integra mediante mecanismos de composición e interacción*. Igualmente se decía que *un componente, que se relaciona con el resto del sistema mediante la interfaz que proporciona y que necesita, se puede tratar como una caja negra, en la que la aplicación de los principios de encapsulamiento y ocultación de la información resultan especialmente interesantes.*

Cada LDA modela sus componentes de una forma, pero siempre contienen una descripción de la funcionalidad del sistema. Esta funcionalidad se integra a través de las interfaces de los componentes. Se pueden considerar dos clases de componentes:
- *Primitivos*: encapsulan una parte del comportamiento y son las entidades básicas para la construcción de la aplicación. Su definición debe incluir la mayor información posible con respecto a su contenido.
- *Compuestos*: son una clase particular de componentes que no encapsulan funcionalidad propiamente dicha sino que contienen una configuración; es decir, las instrucciones relacionadas con la instanciación de componentes y sus relaciones expresadas como interconexiones.

El concepto de *semántica* de un componente es importante dentro del análisis de las arquitecturas y se puede definir como

> *Un modelo de alto nivel que define su comportamiento, que se necesita para realizar su análisis, hacer cumplir sus restricciones, y asegurar mapeos consistentes entre diferentes niveles de abstracción de una arquitectura*.

Cada LDA utiliza formas diferentes para definir la *semántica* de los componentes.

Una *interfaz* representa el conjunto de puntos de interacción de un componente con su entorno. Especifica los servicios que proporciona y los que necesita para realizar sus funciones. Estos servicios pueden ser mensajes, operaciones y/o variables. La definición de la *interfaz* de un componente depende del LDA utilizado: en unos cada punto del *interfaz* es un *puerto* (apartado 2.1.3.3) mientras en otros la *interfaz* entera es tratada como un *puerto*.

Por otra parte, para asegurar la cohesión de los componentes del sistema y los elementos internos del mismo es necesario definir ciertas *restricciones*. Éstas pueden ser propiedades del sistema completo o de parte de él y para determinarlas se usa la notación específica del LDA.

Como elementos de diseño que son, los componentes pueden *evolucionar* y se les puede añadir o eliminar funciones. Los LDA deben, por tanto, proporcionar esta capacidad utilizando técnicas tales como el subtipado y el refinamiento de las características de los componentes.

2.2.1.2. Conectores

Representan y modelan las interacciones entre los componentes. Además, definen las reglas que gobiernan esas interacciones. Al contrario que los componentes, pueden no corresponderse con unidades de compilación en la implementación final del sistema. Pueden ser variables compartidas, entradas de tablas, estructuras de datos dinámicas, llamadas a procedimientos, protocolos de cliente-servidor, etc.

Los conectores tienen definida una *semántica* propia que modela su comportamiento y permite especificar los protocolos de interacción (independiente del procesamiento). Para ello tienen una especie de *interfaz* que es el conjunto de puntos de interacción entre el conector y los componentes que relaciona y define los roles de los componentes participantes en la interacción. Esta *interfaz* es necesaria para permitir la conectividad de los componentes y su comunicación dentro de la arquitectura. Un conector debe *exportar* los servicios que ofrecen los componentes a los que está conectado. Esta *interfaz* recibe distintos nombres (como *roles* en Unicon o Acme, o *puertos* en C2 o Wright).

La *evolución y reutilización* de los conectores se consigue modificando o refinando, siempre que sea posible, los existentes y sus propiedades; los LDA deben proporcionar mecanismos para facilitar esta evolución.

2.2.1.3. Configuraciones

También llamadas *topologías*, se representan como grafos de componentes y conectores. Su definición es necesaria para determinar si los componentes están conectados adecuadamente, si permiten la comunicación y si el resultado describe el comportamiento deseado. En los LDA más avanzados, la *topología* del sistema se define independientemente de los componentes y conectores que lo constituyen. Además, los LDA deben facilitar los cambios, proporcionando dinamismo y facilitando la evolución de la *configuración* del sistema.

2.2.1.4. Algunos LDA representativos

A lo largo de los años han surgido diversas propuestas para describir y razonar en términos de AS a través de los LDA. Los distintos lenguajes constituyen un conjunto de propuestas con distinto grado de rigurosidad que han ido surgiendo paralelamente al desarrollo de la AS desde la década de 1990.

Existe un gran número de LDA, cada uno con diferentes características, funcionalidades y objetivos específicos. En este apartado se ha hecho una clasificación heterogénea e interesada con la única idea de agrupar algunos de estos lenguajes de un modo conveniente. En la Tabla 2.1 se muestran los que se han estudiado, ordenados cronológicamente. Sin embargo, en los próximos apartados se realiza una agrupación en función de la semántica que los soporta.

LDA	*Año*	*Creador/es*	*Características*	*Semántica*
Darwin	1995	Magee, D, Kramer, E.(ICL)	LDA centrado en el dinamismo.	Cálculo π
Rapide	1995	Luckham (Standford)	LDA y simulación.	Eventos (*poset*)
UniCon	1995	Shaw (CMU) y otros	LDA de propósito general con énfasis en conectores y estilos.	Eventos
ArchJava	1995	Aldrich, Chambers (CMU)	LDA con énfasis en la comunicación y descripción jerárquica.	Featherweight Java
C2SADL	1996	Taylor, Medvidovic (UCI)	LDA específico de estilo, que permite modelar arquitecturas en el estilo C2.	Eventos
Acme	1997	Monroe, Garlan (CMU), Wile (USC)	Lenguaje de intercambio de LDA.	Lógica de predic
Wright	1997	Allen (CMU)	LDA de propósito general centrado en la comunicación.	CSP
xArch	2001	Dashof, Garlan, van der Hoek, Schmerl (CMU)	LDA basado en XML permite definir la arquitectura de instancias de un sistema. Núcleo de otros LDA	XML
xADL 2.0	2001	Medvidovic (UCI), Taylor (UCLA)	LDA basado en XML, es un superconjunto de xArch.	XML
LEDA	1999	Canal, Pimentel, Troya (U. de Málaga)	LDA que permite especificar arquitecturas dinámicas y está basado en el cálculo-π.	Cálculo π
xAcme	2001	Schmerl (CMU)	LDA basado en XML que permite pasar de ACME a xArch y viceversa.	XML
ADML	2000	MCC, Spencer (OMG)	Intento de estandarización de descrip-ciones arquitectónicas basadas en XML	XML
PiLAR	2002	Cuesta et al. (U. de Valladolid)	LDA Reflexivo	Cálculo π

Tabla 2.1. LDA considerados en este capítulo.

2.2.2. LDA basados en álgebras de procesos

Todos los lenguajes agrupados en este apartado se basan en algún álgebra de procesos para definir su semántica. Las álgebras de procesos inicialmente se definieron para dotar de un marco semántico formal a los lenguajes concurrentes. Sin embargo, desde los años 80 se vienen utilizando como formalismos para describir y analizar sistemas software y determinar si satisfacen o no ciertas propiedades de interés.

Aunque hay un gran número de lenguajes basados en álgebras de procesos, por su interés, se ha decidido destacar los que figuran en este apartado.

Wright

Fue desarrollado por la Escuela de Ciencias Informáticas de la Universidad Carnegie Mellon [All97] y se puede definir brevemente como *una herramienta de formalización de conexiones arquitectónicas*. Su objetivo es la integración de una metodología formal con una descripción arquitectónica, la aplicación de procesos formales, tales como álgebras de procesos, el refinamiento de procesos y una verificación

automatizada de las propiedades de las arquitecturas software. Sin embargo, es uno de los que tiene menos capacidades dinámicas.

En *Wright* se define un conjunto de tipos de componentes y conectores, y un conjunto de restricciones. Cada una de las restricciones asociadas a un estilo representa un predicado que debe ser satisfecho por cualquier configuración de la que se declare que es miembro del estilo. La notación para las restricciones se basa en el cálculo de predicados de primer orden. Las restricciones se pueden referir a los conjuntos de componentes; conectores y asociaciones (*attachments)*; a los puertos y a las computaciones de un componente específico; y a los roles y las ligaduras de un conector particular. Así mismo, es posible definir sub-estilos que heredan todas las restricciones de los estilos de los que derivan. No existe, sin embargo, una manera de verificar la conformidad de una configuración con un estilo estándar.

Es uno de los lenguajes que tiene la sintaxis más formal y también es el más estático de los que aquí se estudian. Existe una versión dinámica de este lenguaje [AlDoGa98] pero que tampoco alcanza altos grados de dinamismo.

Una descripción de *Wright* convencional consiste en una configuración compuesta por una serie de componentes cuya interfaz está segmentada en *puertos*. Los distintos *puertos* se relacionan entre sí internamente a través de la computación definida por el componente. Por su parte, la comunicación entre componentes requiere la existencia de conectores cuya *interfaz* se segmenta en *roles*, relacionados entre sí mediante un código de unión (*glue code*). La combinación de *puertos* y computación especifica las posibilidades de comunicación de un componente; la de *roles* y el *glue*, el protocolo especificado por un conector. La unión del *puerto* y el *rol* adecuado recibe el nombre de asociación (*attachment*) y es lo que hace posible la comunicación. Todo esto descrito en el álgebra de procesos CSP. Además *Wright* es uno de los pocos lenguajes que permiten la especificación de estilos, mediante el uso de una variante de la lógica de predicados.

Wright dinámico

La especificación de *Wright dinámico* [AlDoGa98] es casi idéntica a la de *Wright*, sólo se añade que en todo sistema dinámico se asume la existencia de un componente especial denominado *configuror*, que realiza las reconfiguraciones requeridas por el sistema. Para ello, el lenguaje se amplía con las cuatro operaciones básicas de la reconfiguración (crear, destruir, enlazar y desligar). Estas operaciones sólo se pueden hacer desde el *configuror,* pues este elemento es el que contiene la especificación completa de los aspectos dinámicos del sistema.

Dado que el objetivo de *Wright* es traducir una descripción arquitectónica a una especificación formal en CSP, la introducción de dinamismo en el lenguaje plantea un problema ya que CSP sólo puede describir configuraciones estáticas de procesos. No tiene soporte para la movilidad, por lo que no es posible crear nuevos procesos y canales de comunicación. Para lograrlo, se introducen dos ideas. La primera es que se restringen los sistemas que se pueden representar: solamente se admiten aquellos para los que el número de configuraciones posibles es finito, aunque pueda ser grande. La segunda es que en la traducción de *Wright* a CSP se añade una etiqueta a todos los eventos, que

expresa en qué configuración ocurren los mismos. La restricción de finitud se da, por tanto, para poder disponer de un número finito de etiquetas. Así, el efecto de una acción de reconfiguración consiste únicamente en la selección de un subconjunto de las expresiones de CSP: las que estén señaladas con la etiqueta adecuada. De este modo, en cada momento se producen solamente las acciones de una configuración concreta, ya que aquellas que tienen una etiqueta diferente no se seleccionarán nunca de manera simultánea, y por tanto no pueden llegar a interactuar.

Este modo de definir el dinamismo en *Wright* es muy complejo, se restringe mucho el número de posibilidades y complica la especificación.

Finalmente, *Wright* no proporciona notación gráfica en su implementación nativa, y la generación de código no produce, al menos en forma directa, ningún ejecutable. Tampoco dispone de ninguna herramienta que lo soporte.

Darwin

Darwin es un lenguaje de descripción arquitectónica, desarrollado por Magee y Kramer [Mag+95, MaKr96], orientado al diseño de arquitecturas dinámicas. De este lenguaje existe una versión, podríamos decir que preliminar, que fue desarrollada por Dulay [Dul90], miembro del grupo de Kramer. La versión denominada *Darwin 2* supone una evolución de la anterior a nivel semántico y fue formalizado usando cálculo π [Mag+95]. Esta es la versión a la que se suele referir la literatura al hablar de *Darwin*. La tercera versión [Gia99] es una simplificación de la parte estructural de *Darwin 2*, que se combina con un modelo de comportamiento completamente nuevo basado en sistemas de transiciones etiquetadas.

Darwin se ha definido como *un lenguaje minimal orientado a objetos, concurrente, de configuración, de coordinación, de composición y de descripción arquitectónica*. Admite notación textual y gráfica. Sintácticamente, todo en *Darwin* es un componente. Según Kramer, *tener dos tipos de elementos -componentes y conectores- altera la estructura de un sistema* [MaKr96] por lo que todo está definido en función de este único concepto. Los componentes pueden ser primitivos o compuestos: los componentes primitivos se corresponden directamente con la implementación; los compuestos se construyen jerárquicamente a partir de otros componentes. Como no hay conectores, en *Darwin* tampoco se definen *configuraciones*. La arquitectura total de un sistema es simplemente un componente compuesto. Por tanto, los conectores no pueden tener nombre, no se pueden reutilizar ni se pueden describir patrones de interacción independientes de los componentes que interconectan.

Dentro de un componente se realizan dos tipos de enlaces: entre dos subcomponentes (instancias) del mismo nivel y entre distintos niveles (enlaces jerárquicos). El lenguaje facilita la definición de patrones entre componentes, por lo que se admite la parametrización de éstos, así como el uso de bucles, condiciones, estructuras tipo array e incluso especificaciones recursivas.

Todo componente tiene dos puntos de interacción bien definidos, que se corresponden con el concepto de *puerto* y pueden ser de dos clases: los servicios que proporciona (*provide*) el componente y los requisitos que necesita (*require*). Así pues, todo enlace en *Darwin* va desde un servicio requerido hasta un servicio ofrecido, aunque una vez establecido es bidireccional y admite una cardinalidad 1:N. Las *configuraciones* se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

Darwin soporta la descripción de arquitecturas reconfigurables dinámicamente a través de dos construcciones: *instanciación tardía* y *estructuración dinámica explícita* (instanciación dinámica). Mediante la *instanciación tardía* cada componente se declara por su nombre y se describe una configuración; luego se instancian los componentes pero sólo en la medida en que los servicios que ellos proporcionan vayan a ser utilizados por otros componentes. La *estructuración dinámica explícita* se realiza mediante construcciones de configuración imperativas; ni siquiera se da nombre al componente, al que sólo se hace referencia mediante su tipo. Se crea dinámicamente en el momento en el que otro componente intenta activar uno de sus enlaces; y él mismo genera un nuevo enlace, desde el que tomará la iniciativa de comunicación –los demás no pueden invocarlo, ya que no conocen su nombre-. Así pues, el dinamismo en *Darwin* consiste en un tipo de reconfiguración restringida que expresa unos patrones fijos, pero muy potentes, en los que se admite la creación de componentes y enlaces. Por otra parte, la evolución dinámica de los enlaces produce de manera habitual una reconfiguración trivial.

Uno de los aspectos más interesantes de *Darwin* es la especificación formal de la semántica de sus enlaces en cálculo π. Esto abarca tanto los enlaces estáticos como las reconfiguraciones dinámicas. A pesar del uso del cálculo π para las descripciones estructurales, *Darwin* no proporciona una base adecuada para el análisis del comportamiento de una arquitectura. Esto es debido a que el modelo no tiene herramientas para describir las propiedades de un componente o de los servicios que presta. Las implementaciones de un componente vendrían a ser cajas negras no interpretadas, mientras que los tipos de servicio son una colección dependiente de la plataforma cuya semántica también se encuentra sin interpretar en el marco de trabajo de *Darwin*. Cada servicio se modela como un nombre de *canal*, y cada declaración de enlace (*binding*) se entiende como un proceso que trasmite el nombre de ese *canal* a un componente que requiere el servicio. Este modelo se ha utilizado para demostrar la corrección lógica de las configuraciones de *Darwin.*

Dado que el cálculo π permite describir procesos móviles, su uso como modelo semántico confiere a las configuraciones de *Darwin* un carácter potencialmente dinámico.

En resumen, *Darwin* es uno de los LDA más potentes y flexibles. Se puede utilizar en varios niveles de descripción, tiene una *semántica* formalizada y un dinamismo limitado, aunque considerable en su versión más declarativa. En cuanto a herramientas que soporten el lenguaje, se puede utilizar *SAA*.

UniCon

UniCon (*Universal Connector Support*) es un LDA desarrollado por Mary Shaw et al. [Sha+95] y proporciona una herramienta de diseño para construir configuraciones ejecutables, basadas en tipos de componentes, implementaciones y *conexiones expertas* que soportan tipos particulares de conectores. *UniCon* se asemeja a *Darwin* en la medida en que proporciona herramientas para desarrollar configuraciones ejecutables de caja negra y tiene un número fijo de tipos de interacción, pero el modelo de conectores de ambos lenguajes es distinto.

Oficialmente se define como *un LDA para soportar estilos y para permitir la construcción de sistemas a partir de sus descripciones arquitectónicas*. Su propósito es generar código ejecutable a partir de una descripción adecuada de los componentes primitivos; además, tiene una gran capacidad para gestionar métodos de análisis de tiempo real a través de RMA (*Rate Monotonic Analysis*).

Los puntos de interfaz de los componentes se llaman *players*. Éstos tienen un tipo que indica la naturaleza de la interacción esperada y un conjunto de propiedades que detallan la interacción del componente en relación con esa interfaz. En el momento de definir la configuración, los *players* de los componentes se asocian con los *roles* de los conectores. En cuanto a la *semántica*, la de los componentes se especifica mediante trazas de eventos con listas de propiedades; la *semántica* de los conectores está implícita en los tipos de conector, cuya información puede darse también mediante lista de propiedades. Además, *UniCon* dispone de notación gráfica.

Este lenguaje genera código C, asociando los elementos arquitectónicos a construcciones de implementación, que en este caso serían archivos que contendrían el código fuente. Sin embargo, hay algunos problemas con esta forma de implementación, dado que presuponer que esa asociación vaya a ser siempre uno-a-uno puede resultar poco razonable (después de todo, se supone que los LDA deben describir los sistemas a un nivel de abstracción más elevado que el que es propio del código fuente). Por otro lado, carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos predefinidos. Los tipos de componente se definen por enumeración, no siendo posible definir sub-tipos y careciendo por lo tanto de capacidad de evolución.

En lo que se refiere herramientas, la que sirve de base a *UniCon* soporta una especificación activa, pues su editor gráfico previene errores durante la fase de diseño.

LEDA

LEDA es uno de los LDA más avanzados y con mayores cotas de dinamismo. Su autor principal es C. Canal [Can00] de la Universidad de Málaga, quién ha desarrollado tanto los aspectos sintácticos como los semánticos [CaPiTr99, CaPiTr01] incluyendo las nociones formales necesarias.

Su estudio es de especial interés puesto que este lenguaje ha servido de base para la definición de *AspectLEDA*, lenguaje que se propone en esta tesis doctoral. Por esta

razón se ha considerado conveniente realizar un estudio más detallado, que se presenta en el Anexo 1, exponiéndose aquí un resumen del mismo.

En su definición y concepción inicial se parte de cómo construir un LDA cuyas descripciones sean luego directamente traducibles a una especificación formal completa. Su descripción semántica se basa en el cálculo π. Se puede decir que *LEDA* es lo que *Wright* podría haber sido si en lugar de CSP hubiera utilizado cálculo π como formalismo base.

Una arquitectura se describe como un conjunto de componentes cuya interacción se indica formalmente en el álgebra de procesos. Ésta da al lenguaje su gran potencia con enormes capacidades dinámicas. En *LEDA* una arquitectura se expresa como un conjunto de componentes cuya interfaz se divide en *roles*. La estructura de cada rol se especifica con una descripción en cálculo π, en la que los nombres hacen referencia a los elementos del componente (sus roles) o son creados como variables auxiliares (*names*). Los componentes pueden ser primitivos o compuestos, y se pueden parametrizar. Cada componente se conecta con otros componentes por medio de asociaciones (*attachments*) directas. Es decir, en *LEDA* no existen conectores. Así mismo, *LEDA* permite definir 3 tipos de asociaciones: estáticas, reconfigurables y múltiples.

El lenguaje admite la extensión de componentes y roles con posterioridad a su definición y la definición de múltiples niveles de abstracción, vistos como un mecanismo de subtipado; existe pues un tipo de herencia (*extend*) y un tipo básico neutral (*any*). Este es uno de los aspectos más interesantes del lenguaje, en particular por su tratamiento formal [CaPiTr01], que lo vincula con la noción de compatibilidad. De hecho, se puede decir que *LEDA* tiene el desarrollo de subtipado más completo.

Finalmente, se puede definir la construcción de un componente como envolvente de otro ya existente de modo que capture las interacciones de sus roles, sin necesidad de redefinirlos como propios; estas construcciones se denominan *adaptadores*.

Las operaciones de reconfiguración no son necesarias: la manipulación de los enlaces es natural como paso de nombres en cálculo π. Los nuevos componentes (procesos) se pueden crear mediante un operador –*new*-; la destrucción de componentes y enlaces también se hace de forma natural al salir los nombres del ámbito del proceso.

La potencia expresiva de *LEDA* es enorme: al estar basado en le cálculo π tiene una gran flexibilidad y gran capacidad para el dinamismo. Las características propias del lenguaje, tales como el tratamiento de subtipado, la definición de adaptadores o los distintos tipos de asociaciones, son un valor añadido. En resumen, se puede decir que éste es uno de los LDA más elegantes y que muestra un mayor dinamismo.

En el marco de esta tesis doctoral se ha desarrollado una herramienta gráfica (*EVADeS*) que facilita la descripción de arquitecturas en *LEDA,* en un entorno amigable (Anexo 2).

En el Anexo 1 se detallan las características del lenguaje *LEDA*.

Acme y sus derivados

Acme [GaMoWi00] no fue diseñado originalmente como un LDA, sino como un Lenguaje de Intercambio Arquitectónico [GaMoWi97] que sirviese de pasarela entre múltiples lenguajes. Sin embargo, actualmente tiende a utilizarse como un lenguaje de arquitectura.

Acme proporciona un vocabulario que permite definir todos los conceptos propios de una AS y soporta restricciones de diseño que se pueden aplicar a cualquiera de los elementos que componen la arquitectura.

Inicialmente se pretendía que el lenguaje fuera un denominador común para poder expresar todas las descripciones arquitectónicas. Garlan lo consideraba un lenguaje de segunda generación porque es como *un metalenguaje que permite el entendimiento de dos o más LDA*. Pero con el tiempo esta dimensión metalingüística de *Acme* fue perdiendo prioridad para profundizar en su papel como LDA puro, aunque en su sitio web oficial [Acme] ya se advierte que este lenguaje no es apto para todo tipo de sistemas. *Acme* se define como un vocabulario no interpretado de 7 elementos: *componentes, conectores, puertos, roles, sistemas, representaciones y mapas de representación (rep-maps)* que se corresponden con los análogos de otros LDA: Los *componentes* son elementos computacionales y de almacenamiento de un sistema. Los componentes pueden tener múltiples *interfaces*, y los puntos de interfaz se llaman *puertos*. Los *conectores* representan interacciones entre componentes. Los conectores también tienen interfaces que están definidas por un conjunto de *roles*. Los *sistemas* son configuraciones; una *representación* es la descomposición jerárquica de un componente o conector; y un *rep-map* expresa la correspondencia entre las interfaces de un componente y una representación suya. También se pueden definir *enlaces* entre los puertos y roles de los componentes y conectores, respectivamente. Igualmente se pueden definir plantillas de componentes, que se resuelven de manera estática.

Acme, según [Gar95], permite definir cuatro conceptos arquitectónicos:

- La estructura del sistema (su organización), en función de las distintas partes que lo constituyen: componentes, conectores, sistema, representaciones,…
- Propiedades, que constituyen la información de un sistema o de las partes que lo forman, que describe su comportamiento y permite razonar sobre el comportamiento local o global, tanto funcional como no funcional.
- Restricciones de diseño, que determinan cómo puede evolucionar la arquitectura a lo largo del tiempo.
- Tipos y estilos, que definen clases y familias de arquitecturas.

En cuanto a la semántica, *Acme* no soporta de manera directa la especificación semántica, pero sí permite la utilización de modelos semánticos de otros LDA a través de listas de propiedades. Éstas no se interpretan y existen sólo a nivel de documentación. Además, *Acme* facilita la gestión de familias de sistemas o estilos. Esta capacidad se define de modo natural como una jerarquía de propiedades correspondiente a tipos.

Los elementos de los siete tipos citados pueden tener asociados una serie de especificaciones, que se denominan *propiedades*, que pueden venir expresadas en

cualquier lenguaje, ya que *Acme* sólo define lo que se denomina un *marco semántico abierto*. Este marco sólo supone que las propiedades se expresarán en términos de Lógica de Primer Orden y las relaciones predefinidas, y como pares de asignaciones atributo-valor. Sin embargo no se proporciona una definición cerrada de las mismas.

Para definir una arquitectura en *Acme* se deben seguir los siguientes pasos:

- Identificar los conceptos en el modelo fuente que se corresponden con los elementos que *Acme* permite definir: sistema, componentes, conectores, puertos, etc.
- Definir una familia o grupo de familias para la descripción en *Acme* (modelo destino) y describir en *Acme* los componentes, conectores, puertos, etc., identificados en el paso inicial.
- Definir la serie de propiedades que describen el comportamiento de los elementos del modelo.

Aunque *Acme* se basa en la idea de representación de arquitecturas como estructuras estáticas, también se puede utilizar para representar arquitecturas reconfigurables (dinámicas). De esta forma se pueden añadir propiedades y nuevas características a los sistemas en tiempo de ejecución.

Hasta el momento de redactar esta tesis (Julio de 2008) se han estudiado intercambios desde representaciones en *Aesop*, *C2*, *Meta-H* y *UniCon* hacia *Acme*, e intercambios desde *Wright* y *Rapide* a *Acme* y viceversa, de *Meta-H* hacia *Acme* y de *Acme* hacia *UML*. Sin embargo, cada vez se insiste más en utilizarlo para describir arquitecturas directamente, en lugar de intercambiarlas. En este sentido es en el que se deben entender los desarrollos siguientes, de los que se ha afirmado en algún momento ser dinámicos: *Armani*, *HOT Acme*, *AML*. Ninguno tiene fundamento algebraico, pero están vinculados a *Acme*; son extensiones o modificaciones de éste.

En cuanto a herramientas que asisten al arquitecto, para *Acme* el proyecto Isis y Vanderbilt proporcionan *GME* (*Metaprogrammable Graphical Model Editor*), un ambiente de múltiples vistas que, al ser meta-programable, se puede configurar para cubrir una rica variedad de formalismos visuales de diferentes dominios. Una segunda opción es *VisEd*, un visualizador arquitectónico y editor de propiedades de GA Tech. Otra herramienta es *AcmeStudio;* y finalmente, *Acme Powerpoint Editor* de ISI, que implementa COM. Por otra parte, *Acme* se puede probar con *Aladdin*, un analizador de dependencias creado por el Departamento de Ciencias de la Computación de la Universidad de Colorado.

Armani

Fue desarrollado por Monroe [Mon98] como una extensión de *Acme*, no pretende describir la arquitectura, sino abarcar la fase de diseño en su totalidad. Incorpora un Lenguaje de Predicados que permite expresar restricciones de la arquitectura y heurísticas de diseño de un modo más claro que *Acme*. Es una variante de la lógica de predicados a la que se incorpora una serie de funciones predefinidas sobre tipos, conjuntos, relaciones y conectividad de grafos; y a la que se añaden una serie de operadores y los cuantificadores *para todo* y *existe*, así como la posibilidad de definir funciones propias de usuario y de especificar invariantes. El lenguaje está definido de manera semiformal y se inspira en el OCL de UML.

HOT Acme

HOT Acme [PeSu98] consiste en la integración de *Acme* en un sistema de tipos de orden superior –HOT: *Hight Order Typed*- que pretende dar soporte para traducir una descripción *Acme* a una especificación verificable: un comprobador automático de tipos.

En este lenguaje se parte de la descripción de un determinado elemento arquitectónico (un componente o un puerto), con sus propiedades definidas en *Acme*. La descripción de cada elemento de una configuración se trata como la definición de un tipo polimórfico. Luego, el mecanismo de comprobación de tipos es capaz de determinar si la misma es consistente. Este tipo de verificación puede aplicarse a cualquier aspecto que se desee, en particular para la definición de arquitecturas dinámicas, aunque esta línea de trabajo parece que ha sido abandonada, ya que no hemos encontrado referencias recientes de ella.

AML

AML, *Architectural Meta-Language*, fue diseñado por Wile [Wil99] como un Lenguaje de Intercambio alternativo a *Acme*. Se planteó como un metalenguaje neutral en el que poder describir las características de los distintos LDA, para luego poder realizar intercambios de descripciones sobre una base común. Al considerarse como un lenguaje de intercambio nunca se ha considerado un uso directo.

AML se basa en el Modelo Entidad-Relación; por eso las arquitecturas se representan como un conjunto de elementos que mantienen una serie de relaciones topológicas que se describen estrictamente. El concepto fundamental del lenguaje es el de relación que se asocia a una serie de suposiciones, que expresan sus propiedades en una notación basada en la Lógica de Primer Orden y en la que las propias relaciones son predicados. Estas relaciones se definen entre elementos concretos y quedan identificados por definición; es decir, son instancias, no tipos. La construcción más potente de este lenguaje es la que permite definir elementos genéricos para expresar relaciones genéricas y permite además la creación de nuevos tipos definidos por el usuario, que luego pueden usarse como si fueran palabras clave en cualquier lugar de la descripción.

El autor del lenguaje dice que *uno de los objetivos del mismo es la especificación del dinamismo*. Éste se introduce a través del uso de las restricciones proporcionadas por el lenguaje.

ArchJava

Es un lenguaje de descripción de arquitecturas creado en la universidad de Washington por Aldrich, Chambers y Notkin [AlChNo02a, ArchJava]. *ArchJava* amplía Java para modelar arquitecturas como una jerarquía de instancias de componentes. La comunicación de los componentes es a través de conectores que pueden tener una semántica definida por el usuario. Es un lenguaje que verifica que un programa se ajusta a la arquitectura del sistema en un sentido técnico denominado *integridad de la*

comunicación: "los componentes en un sistema sólo se comunican a través de los canales de comunicación declarados en su arquitectura".

ArchJava es una extensión de Java que *permite a los programadores especificar la arquitectura del software dentro del programa.* Para describir arquitecturas software *ArchJava* añade nuevas construcciones que permiten la definición de componentes, puertos y conexiones. En *ArchJava*, la estructura jerárquica del software se expresa mediante *componentes compuestos* que están constituidos por un cierto número de subcomponentes conectados unos con otros; un *subcomponente* es una instancia de un componente anidado en otro componente.

Lo que hace interesante a *ArchJava* es que se apoya en arquitecturas dinámicas, pues algunas arquitecturas necesitan crear y conectar dinámicamente un número determinado de componentes. Además, el componente de nivel superior debe instanciarse al principio de la aplicación, incluso en programas con arquitecturas estáticas [AlChNo02b].

La *integridad de la comunicación* exige a cada componente definir el tipo de interacciones arquitectónicas que se permiten entre subcomponentes. Se usa un patrón de conexión (*connect pattern)* para describir el conjunto de conexiones que pueden instanciarse en tiempo ejecución.

Como Java no proporciona una manera explicita de *eliminar* objetos, *ArchJava* tampoco; en su lugar, los componentes se *almacenan como basura* cuando no se instancian a través de referencias directas o conexiones.

La semántica de *ArchJava* viene definida por una extensión de Featherweight Java, modelo formal, compacto y minimal de Java que permite probar extensiones de Java [BiAlHa06].

Por otra parte, *ArcJavaDoc* es una herramienta que se ha desarrollado para facilitar la documentación gráfica para arquitecturas *ArchJava.*

El estudio de este lenguaje ha tenido especial interés en el desarrollo de esta tesis doctoral, pues ha servido de base para definir un nuevo lenguaje OA –*AAJ*- aún en fase de desarrollo.

PiLAR

PiLAR [Cue02] es un LDA diseñado con el objetivo de servir de modelo para representar arquitecturas dinámicas. Para ello, se le ha dotado de una estructura reflexiva en la que los conceptos de reflexión se integran en el lenguaje. Es un LDA dinámico de tipo proceso-algebraico con su semántica basada en el cálculo π poliádico.

Su sintaxis es sencilla; en ella no existe el concepto de conector como elemento de primera clase; sólo hay componentes y enlaces entre ellos. Una descripción arquitectónica en *PiLAR* se estructura en tipos de componentes (arquetipos) e instancias. El comportamiento se especifica mediante restricciones.

Al estar basado en reflexión, la descripción de una arquitectura puede estar dividida en meta niveles. *PiLAR* introduce la noción reflexiva de *reificación* que es una relación estructural bidireccional que expresa una conexión causal entre los componentes (instancias) que conecta. La instancia *reificada* se denomina componente base (o *avatar*), mientras que la que *reifica* se denomina metacomponente. El conjunto de los metacomponentes forma el meta nivel de la arquitectura, y dentro de su meta nivel se comportan como cualquier instancia de componente. Por otra parte, tienen un acceso total a sus componentes base (*avatares*) en el nivel inferior (incluidos sus elementos internos y en particular a sus puertos). Además, pueden a su vez ser *reificados* desde un nivel meta meta lo que da lugar a una definición arquitectónica en varios meta niveles.

Con la definición de este LDA se demuestra que un enfoque reflexivo de la arquitectura del software proporciona un entorno tal que permite describir el dinamismo. Por tanto, este lenguaje es especialmente adecuado para la especificación de sistemas reflexivos y arquitecturas dinámicas.

PiLAR, sin embargo, no dispone, hasta el momento, de herramientas que le sirvan de soporte.

2.2.3. LDA basados en eventos

El modelo basado en eventos y en particular la invocación implícita [GaNo91] muestra una clara capacidad de dinamismo, ya que permite definir una arquitectura con un grado de acoplamiento débil. Los lenguajes que se pueden incluir en este modelo utilizan los eventos como concepto básico y además realizan una gestión global de los mismos durante la especificación.

Los modelos basados en eventos están relacionados en cierto modo con las álgebras de procesos ya que éstas, en la teoría de la concurrencia, se han planteado como la definición de patrones de emisión y recepción de eventos.

Rapide

Rapide se puede definir como *un lenguaje de descripción de sistemas de propósito general que permite modelar interfaces de componentes y su conducta observable*. Fue desarrollado por Luckham [Luc+95] basado en la idea de modelar las computaciones e interacciones del sistema mediante un conjunto de *eventos parcialmente ordenados* (*poset*). Estas construcciones permiten obtener un modelo ejecutable que puede simularse en tiempo de diseño. Por tanto, *Rapide* es un lenguaje concurrente de descripción de arquitecturas basado en eventos y orientado a objetos, que permite simular y analizar el comportamiento de arquitecturas de sistemas distribuidos. Los requisitos del sistema se expresan como restricciones y como patrones de eventos concurrentes.

La estructura de *Rapide* es muy compleja y se articula en cinco lenguajes:

- El *lenguaje de tipos* describe las interfaces de los componentes.
- El *lenguaje de arquitectura* describe el flujo de eventos entre componentes.

- El *lenguaje de especificación* describe las restricciones abstractas del comportamiento de los componentes.
- El *lenguaje ejecutable* describe los módulos ejecutables.
- El *lenguaje de patrones* describe los patrones de evento.

Sin embargo, el concepto de arquitectura de este LDA no coincide exactamente con el tradicional. En *Rapide*, los componentes no forman parte de la arquitectura, que tiene una estructura independiente. Esta estructura define los patrones de interacción entre unas *interfaces*, que son uno tipo de componentes, que en *Rapide* se definen como un conjunto de eventos de comunicación abstracta. Posteriormente, los componentes se integran en la arquitectura enlazándolos a una interfaz determinada. De un modo general, se podría decir que el concepto de arquitectura de *Rapide* y el estándar se corresponden si se asimila el concepto de *interfaz* de *Rapide* con lo que en otros lenguajes se entiende como tipo de componente. En realidad, una *interfaz* no representa un componente sino a los requisitos de interacción sobre un componente, aunque a veces se confunden estos términos. Desde un punto de vista práctico, esto tiene la ventaja de que un componente se puede sustituir por otro que cumpla con las especificaciones de la *interfaz*. Así son las *interfaces* las que forman parte de la arquitectura. Por tanto, se puede decir que las arquitecturas *Rapide* están formadas por:

a) Un conjunto de especificaciones de módulos, *interfaces*. Cada una tiene una conducta asociada que se define a través de *conjuntos de eventos parcialmente ordenados*.
b) Un conjunto de *reglas de conexión* que define la comunicación directa entre las *interfaces* y que se especifica mediante patrones de eventos.
c) Un conjunto de *restricciones formales* que define los patrones de comunicación.

Según esto, los elementos arquitectónicos principales en la sintaxis de *Rapide* son:

- *Componentes*: *Rapide* define tipos de componentes (*interfaces*) en términos de una colección de eventos de comunicación que pueden ser observados (*acciones externas*) o iniciados (*acciones públicas*); generan y observan la ocurrencia de eventos. Los eventos generados pueden ser *dependientes* o *temporizados*. Los componentes pueden ser objetos de *interfaz*, una arquitectura completa que implementa una *interfaz* o un módulo que implementa una *interfaz*. Mientras muchos LDA no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, *Wright* y *Rapide* sí permiten modelar el comportamiento de sus componentes.
- *Interfaces*: definen el comportamiento computacional de un componente vinculando la observación de acciones externas con la iniciación de acciones públicas. Pueden ser *funciones*, que son interfaces de comunicación síncrona requeridas o proporcionadas, o *eventos*, que son interfaces de comunicación asíncrona.
- *Conectores*: conectan interfaces *emisoras* con interfaces *receptoras*. Los componentes se comunican entre ellos a través de los *conectores* mediante llamadas a funciones de sus interfaces. No se puede poner nombre a los conectores, subtipar o reutilizar. Hay tres tipos de conexión: *conexión básica* (A to B), *conexión de tubería* o *pipe* (A => B), y *conexión de agente* o *agent* (A | | > B).
- *Restricciones*: se distinguen los patrones (*patterns*) que aparecen en las interfaces y/o en la definición de una arquitectura, y las restricciones secuenciales que aparecen en

el comportamiento a nivel de módulo y se aplican a tipos, parámetros, objetos y declaraciones.

- *Configuración*: es la arquitectura en sí, formada por componentes, conectores y restricciones.

Los *eventos* son complejos, admiten parámetros e incluso un valor de retorno; así se pueden considerar como si fueran métodos; aunque existen diferencias, la mayoría de las veces pueden obviarse. Se distinguen dos tipos de eventos: *acciones externas* que son observadas por la interfaz y *acciones públicas* que son generadas por ella. La computación se define relacionando la observación de *acciones externas* con la iniciación de *acciones públicas*, mediante la especificación de un orden parcial. Los *eventos* se difunden mediante *broadcast*. Sin embargo, cada interfaz sólo observa aquellos eventos que tiene definidos como *acciones externas*. La relación entre eventos localizados en distintos puntos de la especificación puede tomar dos formas: utilizando conexiones y mediante restricciones. Dos eventos que se han comparado se consideran equivalentes, esto es, son el mismo evento. Así, partes separadas de la arquitectura, que habrían podido incluso definirse independientemente, quedan ligadas de manera directa.

La unión mediante *restricciones* es más flexible. En este caso, se especifican relaciones causales entre dos eventos: uno es la causa del otro y la aparición del primero implica la iniciación del segundo; pero se consideran eventos distintos; en realidad, lo que se hace es definir un orden parcial entre ellos. Esto permite la descripción de patrones de interacción de eventos a lo largo de la arquitectura; sin embargo, dichos patrones se entremezclan con la definición de las interfaces y en general son poco reutilizables. Las conexiones son poco flexibles, siéndolo más las restricciones.

Las arquitecturas en *Rapide* se diseñan considerando que su objetivo es la simulación. Toda descripción *Rapide* se puede reducir a un *poset*, es decir, a una secuencia parcialmente ordenada de eventos. Sin embargo, la especificación del orden parcial normalmente no es determinista, lo que significa que una simulación concreta (la ejecución de la arquitectura) sólo tomará uno de los caminos posibles; el resto se ignora. Así, el *poset* resultante describe el encadenamiento de eventos en una ejecución determinada, lo que agota todas las posibilidades de la especificación. Por tanto, las herramientas de *Rapide* no permiten la verificación de la arquitectura, sino una simulación que prueba su corrección, pero no la demuestra.

Por otra parte, el orden parcial se describe en la especificación y por lo tanto es totalmente estático. El dinamismo en *Rapide* viene dado por una sentencia *where* que puede crear o no un cierto enlace –y desencadenar o no una serie de eventos o acciones-, según se cumplan o no las condiciones que ésta especifica. Por tanto, se tiene un tipo de reconfiguración programada, basada en restricciones, equiparable en potencia a las de *Darwin*.

Con posterioridad, Vera, Perrochon y Luckman [VePeLu99] añadieron a *Rapide* un conjunto de comandos de reconfiguración, más directos que los anteriores, que se mantienen integrados en el lenguaje. Estos comandos pueden invocarse interactivamente (reconfiguración *ad hoc*) desde una herramienta, o asociadas a una restricción para que se

disparen en el momento adecuado (reconfiguración programada). Estos comandos se definen en función de *posets*.

Como conclusión se puede decir que *Rapide* es uno de los lenguajes más versátiles y utilizados, no ha dejado de evolucionar y, además, dispone de un conjunto de herramientas de soporte: los diseños de *Rapide* también se pueden probar con *Aladdin*, un analizador de dependencia creado por el Departamento de Ciencias de la Computación de la Universidad de Colorado.

C2 /C2SADL

C2 es un estilo arquitectónico útil en el modelado de sistemas que requieren el paso intensivo de mensajes y que suelen tener una interfaz gráfica dominante. La comunicación está basada en eventos que es el único tipo de interacción posible: los componentes nunca se comunican directamente con otros elementos sino que emiten eventos. *C2SADL* (*Simulation Architecture Description Language*) es un LDA que permite describir arquitecturas en el estilo C2 [Tay+96]. Este lenguaje es predecesor de C2SADEL (*C2 Software Architecture Description and Evolution Language*) [MeRoTa99] que es una variante suya.

Toda descripción en *C2SADL* [Tay+96] consta de tres partes: un conjunto de *componentes*, otro de *conectores* y una especificación de la arquitectura (topología que forman). Los *componentes* tienen un comportamiento propio, que no tiene por qué ser conocido por la arquitectura. Los *conectores* son básicamente *buses de eventos* y tienen lo que se ha denominado *reflexión contextual*: no tienen una interfaz fija sino que ésta depende de los *componentes* que estén conectados a ellas, es decir, de su contexto. Además se permiten enlaces de *conector* a *conector* directamente (lo que lo diferencia de cualquier otro LDA).

C2SADL está dotado de una serie de herramientas software que facilitan su uso, incluso se ha creado una biblioteca Java que implementa *C2*, de modo que las especificaciones *C2SADL* se pueden utilizar para generar código. Se apoya en la definición de una teoría de tipos completa [MeRoTa99]. Ésta se basa en los tipos de comportamiento, pero adaptado al entorno de la arquitectura del software y, en especial, a la noción de componente. Utiliza el concepto de *conector* de tal modo que este concepto ha servido para la integración de componentes COTS en arquitecturas generales. También se ha trabajado en la conversión de sus especificaciones a un formato intermedio basado en XML (apartado 2.2.4).

Se puede decir que *C2SADL* es uno de los lenguajes con mayores capacidades para la expresión de la evolución de sistemas a través de la AS [Med99, MeRoTa99], además de contar con alguno de los mejores estudios sobre el tema específico de la AS dinámica [Med96]; aunque tiene la limitación de haber sido diseñado para representar únicamente el estilo *C2*. Esta característica ha sido fundamental a la hora de diseñar las herramientas, plantear la integración de los COTS o traducirlo a otros entornos.

C2SADL es "más dinámico" que otros LDA al no estar predeterminada la forma de sus conectores y porque en cada momento sólo se activan algunos de los enlaces; pero

la configuración no puede cambiar en ningún momento, ya que todas las conexiones son fijas y explícitas. Por tanto, el lenguaje en sí no es dinámico.

Como herramienta de soporte, *Saage* es un entorno para diseño arquitectónico que utiliza *C2SADL* y requiere Visual J++ (o Visual J#), COM y eventualmente Rational Rose (o DRADEL). También existe una extensión de *Visio* para *C2*.

2.2.4. LDA basados en XML

XML es una herramienta muy flexible y extensible, adecuada para lenguajes de descripción de arquitecturas, que se está empleando cada vez más en un amplio conjunto de herramientas de soporte. Esto hace que los lenguajes basados en *XML* partan con ventaja frente a los demás, desde el punto de vista de la compatibilidad y del uso de herramientas de ese tipo. Además, el esquema de *XML* proporciona un metalenguaje que se adapta perfectamente para el desarrollo de notaciones modulares y extensibles.

Es usual asociar *XML* con documentos en el sentido tradicional. Sin embargo, la flexibilidad que proporciona presenta la capacidad de describir modelos de información no orientados a documentos. En concreto, tiene muchas ventajas como estándar para la representación de LDA. Se ha creído conveniente realizar un estudio de los LDA basados en *XML* ya que suponen una novedad en este campo.

El uso de *XML* como estándar para representar información permite que algunos LDA basados en él puedan compartir esquemas o modelos, y pueda haber más compatibilidad y semejanzas entre unos LDA y otros. Los siguientes puntos explican cómo *XML* se adapta a las características que requiere un LDA [Pru+98]:

1) *XML* proporciona claridad suficiente para representar el vocabulario mediante el cual un LDA describe componentes, conexiones y restricciones de un sistema. Además, su extensibilidad es adecuada para soportar los cambios que se puedan dar en los LDA.
2) A la hora de realizar el análisis, es posible que un diseñador necesite realizar más de uno sobre su correspondiente diseño y tener que representarlo usando diferentes LDA, que proporcionen las características deseadas para cada análisis. La transformación a cada LDA se puede facilitar mediante el uso de un estándar como *XML*.
3) Desde el punto de vista de las trazas, es importante asociar de forma directa los requisitos del sistema con aquellos elementos que los satisfacen. Si los requisitos se capturan dentro de *objetos* de forma que puedan direccionarse mediante una URL, las facilidades de enlace –*linking*- que proporciona *XML* pueden hacer que un elemento de la arquitectura asocie el *objeto* con el requisito correspondiente. Esos *objetos* de requisitos pueden contener escenarios, diferentes casos de prueba, etc.
4) El arquitecto puede presentar la arquitectura de un cierto sistema desde distintos puntos de vista, mostrando la información de modo diferente, en función, por ejemplo, del destinatario. Esto facilita la comunicación durante el proceso de desarrollo. La extensibilidad de *XML* permite capturar gran variedad de información para expresar el sistema desde las distintas vistas de la arquitectura.

Esta información se puede encapsular en ella usando las etiquetas *XML* o mediante enlaces hacia o desde la arquitectura por medio de las facilidades de enlace que ofrece *XML*. En cualquier caso, la información de una vista determinada es transparente para las demás vistas, siendo posible la comunicación entre ellas. Las presentaciones visuales y ayudas gráficas en general facilitan esta labor.

5) En el ámbito de la colaboración, las facilidades que proporciona *XML,* en lo que a enlaces se refiere, posibilita que un modelo arquitectónico se pueda dividir en partes para, posteriormente, recomponerlo. Esto soporta de manera significativa las necesidades que requiere un desarrollo de software distribuido.

6) Por último, *XML* se está erigiendo como el estándar para los creadores de repositorios comerciales. Los repositorios son muy útiles para almacenar modelos, de modo que se pueda hacer uso de ellos posteriormente en otras aplicaciones. Esta característica quizás en un futuro permita el intercambio de modelos arquitectónicos mediante la definición de repositorios de diseños arquitectónicos expresados en este lenguaje.

Finalmente, para todos los lenguajes basados en *XML* se puede usar cualquier herramienta de soporte de este lenguaje (como *XML Spy*), lo que facilita la creación de una arquitectura.

Como conclusión se puede decir que la extensibilidad y flexibilidad de *XML* lo hacen adecuado para describir modelos de LDA. Sin embargo, lo más importante es que *XML* está recibiendo importante apoyo comercial, cuestión a tener en cuenta para, desde el punto de vista industrial, adoptar un LDA u otro.

En los últimos años se han desarrollado varios lenguajes basados en *XML*. Entre ellos: *xArch*, *xADL 2.0*, *xAcme* y *ADML.* Además, y como ya se ha dicho, *XML* permite que esos LDA puedan ampliarse y adaptarse fácilmente. Los cuatro están relacionados entre sí, ya que se puede pasar de la representación en uno de ellos a la representación en cualquier otro lenguaje de ese grupo (Figura 2.2).

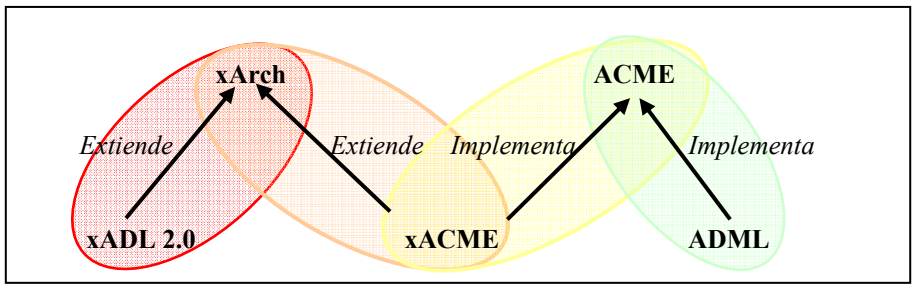


Figura 2.2. Relación entre los diferentes lenguajes basados en *XML*.

xArch

xArch [xarch, DaHoTa01] surgió con el objetivo de obtener un LDA más abierto y flexible que los LDA específicos que existían en aquel momento para conseguir un tipo de representación estándar de arquitecturas software. Está formado por un conjunto de

esquemas XML que permiten representar la arquitectura de instancias de un sistema; es decir, la representación de su arquitectura ejecutable –el modelo de ejecución-. Para describir la estructura de la arquitectura en tiempo de diseño (representación previa al sistema ejecutable) hay que utilizar una extensión del lenguaje como la realizada por los desarrolladores del lenguaje *xADL*. Las extensiones se pueden organizar jerárquicamente, de forma que se pueden importar extensiones de otros diseños en *xArch*, e incluso extenderlas. Pueden crearse traductores hacia y desde otros LDA basados en XML, así como herramientas que pueden usarse para analizarlos y manipularlos.

Los esquemas de *xArch* proporcionan los elementos comunes a la mayoría de los LDA; para ser lo más genérico posible, no presenta ningún tipo de restricción sobre el comportamiento y la organización de dichos elementos. *xArch* es, a su vez, el núcleo de otro LDA más completo, *xADL 2.0.*

xArch, al proporcionar un núcleo con notación XML, ofrece las siguientes posibilidades:

- Puede servir de punto de partida para otras notaciones arquitectónicas (basadas en XML), ya que *xArch* se puede reutilizar como núcleo y extenderlo para dar soporte a otros LDA.
- *xArch* se puede utilizar como un mecanismo de intercambio de representaciones de arquitecturas. Los traductores pueden construirse para traducir desde y hacia XML.
- Al estar basado en XML permite el uso de herramientas para dicho lenguaje.

Las características de *xArch* a tener en cuenta son:

- En *xArch* no hay distinción entre roles y puertos.
- No hay propiedades asociadas o sujetas a ningún elemento.
- La descomposición jerárquica de componentes y conectores sólo se puede realizar de una manera, a diferencia de otros LDA que proporcionan más posibilidades.
- En *xArch* las interfaces no tienen representación (se hace de forma externa e independiente a la descripción en *xArch*).
- Facilita la creación de grupos, esto es, agrupaciones lógicas de elementos de la arquitectura.
- *xArch*, como cualquier LDA, permite hacer una descripción global de una arquitectura, pero no realiza representaciones internas de los elementos que la componen. Esas representaciones deberán realizarse de la forma que se considere más adecuada, y en un lenguaje de implementación concreto.

Los elementos incluidos en el esquema de instancias son los siguientes: instancias de componentes, de conectores, de interfaces, de enlaces y de grupos generales (agrupaciones lógicas de componentes, conectores, interfaces e, incluso, otros grupos).

Las herramientas de soporte para *xArch* se encuentran en *ArchStudio 3*, paquete de aplicaciones de diferente tipo que permiten describir arquitecturas fácilmente, aplicar tests a las descripciones, trabajar con los tipos definidos en una arquitectura, etc. Se trata de herramienta muy completa desarrollada por la *UCI* (University of California, Irvine).

xADL 2.0

Fue creado, al igual que *xArch*, por la UCI [DaHo01] e igualmente define esquemas XML para la descripción de familias arquitectónicas, o sea, estilos, siendo fácilmente extensible. Permite realizar un modelado tanto en tiempo de diseño como en tiempo de ejecución, así como gestionar la configuración de la arquitectura. Se definen esquemas genéricos, pues su objetivo principal es "ser un lenguaje estándar para representar arquitecturas".

xADL 2.0 presenta ciertas características de las que pueden beneficiarse los LDA que deriven de él:

- Separación de los modelos de diseño y de ejecución, dentro del sistema software.
- Implementación del mapeado que permite pasar de la especificación del LDA a código ejecutable.
- Capacidad para modelar ciertos aspectos de la evolución y la generación de arquitecturas.

Al igual que sucede con otros LDA, el núcleo de este lenguaje modela los cuatro elementos arquitectónicos: *componentes, conectores, interfaces y configuraciones.*

La filosofía de los creadores de *xADL 2.0* era reunir los mejores conceptos y características de otros LDA en un *super* LDA adecuado para todos los dominios de aplicación. Sin embargo, el concepto de arquitectura es muy amplio y los diferentes puntos de vista que se tienen sobre ella en los distintos proyectos y/o marcos de trabajo, así como el tipo de datos que debe contener cada una, son muy diferentes. Por ello, la conclusión fue que era difícil que un LDA pudiera ajustarse a todos los proyectos y dominios de aplicación, aunque la sintaxis de *xADL 2.0* y sus herramientas se creasen con intención de poder extenderlas y dar soporte a necesidades específicas de los diversos proyectos. La idea fundamental era que donde las metodologías divergen *xADL 2.0* tiende a ser neutral, no optando por ninguna forma específica de representación.

La principal ventaja de *xADL 2.0* es su capacidad de extensión y que puede actuar como base para el desarrollo de LDA específicos de un dominio determinado, al igual que para *xArch*. Además, *xADL 2.0* presenta un conjunto de herramientas de infraestructura que soportan la creación, manipulación y compartición de documentos *xADL 2.0,* lo que supone una ventaja frente a otros LDA que no presentan herramientas de soporte. Por otra parte, los elementos incluidos en los esquemas de instancias de *xADL* son los mismos que para *xArch*.

Con *xADL 2.0* se puede usar también *ArchStudio 3;* para trabajar con arquitecturas de líneas de producto se usa *Menaje,* para el mantenimiento y gestión de arquitecturas cambiantes en tiempo de ejecución. También existe una extensión de *Visio* para dar soporte a este lenguaje que se utiliza como la interfaz de usuario de preferencia para *data binding* de las bibliotecas de *ArchEdit* para *xADL*.

Como conclusión, sobre estos dos lenguajes, se puede decir:

- *xADL* y *xArch* están muy relacionados. *xADL 2.0* es una extensión de la representación genérica que proporciona *xArch* o, dicho de otra forma, *xArch*

representa el núcleo de *xADL 2.0*, concretamente el núcleo de instancias junto con las extensiones de estructura y tipos.
- Desde el punto de vista de la semántica, *xArch* se considera neutral, ya que no define restricciones o reglas sobre los elementos o su comportamiento. Estas cuestiones deben especificarse en las extensiones.
- *xArch* y *xADL* están formados por un conjunto de esquemas XML que se pueden usar como base para el desarrollo de nuevos LDA basados también en XML.

ADML

ADML (*Architecture Description Markup Language*) [adml, Spe00] es un lenguaje de representación de arquitecturas y constituye un intento de estandarizar las descripciones arquitectónicas basadas en XML. Empezó a desarrollarse a finales de los 90 por el Consorcio de Microelectrónica y Tecnología de Computadores (*MCC*) y está promovido por el *Open Group* [OGR]; *ADML* es una representación en XML de *Acme*. Este lenguaje se eligió como base debido a su capacidad como lenguaje de intercambio de LDA, y porque se considera un estándar para la interoperabilidad y reutilización de arquitecturas. Se suele usar en el ámbito de las líneas de producto y las arquitecturas de empresa. *ADML* añade a *Acme* lo siguiente:

- Una representación procesable por cualquier *parser* de XML.
- Capacidad para definir enlaces a objetos externos a la arquitectura, como componentes, diseños, etc.
- Capacidad para trabajar con repositorios comerciales.
- Extensibilidad transparente.

Desde el *Open Group* se ha tratado de concienciar a la industria de las ventajas que supone adoptar *ADML*, de manera que los desarrolladores de herramientas de COTS lo utilicen cada vez más. Las posibilidades que ofrece el lenguaje son:

- Llegar a ser un estándar para las arquitecturas de líneas de producto, objetivo original para el que se creó.
- Ser la base para un nuevo tipo de herramientas de soporte para los desarrollos arquitectónicos.
- Convertirse en estándar para el intercambio de modelos arquitectónicos entre herramientas de arquitectura, y entre herramientas de arquitectura y herramientas usadas en otras fases del ciclo de vida.
- Representar los modelos en diferentes vistas: vista lógica, física, de organización, etc.
- Dividir los modelos para permitir la colaboración entre grupos de trabajo dispersos geográficamente (desarrollo distribuido).
- *ADML* se utiliza principalmente para intercambiar modelos arquitectónicos entre distintas herramientas. La idea de los creadores es que desarrolladores de herramientas y arquitectos colaboren en la evolución del lenguaje. Sin embargo, es necesario que *ADML* se extienda aún más y se consolide como estándar para que se lleguen a cumplir los puntos anteriores, ya que, de hecho, los arquitectos de empresa

no usan directamente *ADML*, sino que se benefician de él a través de herramientas que utilizan para crear arquitecturas más fácil y rápidamente.

- Con este lenguaje se puede usar *TOGAF*, herramienta de especificación con muchas posibilidades.

xACME

Los esquemas de *xAcme* fueron definidos en la Carnegie Mellon Univeristy [xacme, Sch01] y extienden el núcleo estructural de *xArch*. Fue propuesto inicialmente como lenguaje de intercambio entre *xArch* y *Acme*, pues representa conceptos de *Acme* usando el conjunto de esquemas de *xArch*. Es por ello que *xAcme* también se considera como una extensión *Acme* para *xArch*. Estas extensiones que forman *xAcme* se representan mediante cinco esquemas y cada una de ellas proporciona diferentes funcionalidades de *Acme*. Su objetivo es permitir, a las distintas herramientas que describen arquitecturas con *Acme,* el intercambio de dichas descripciones arquitectónicas con herramientas que usan *xArch*. Por tanto, la idea de *xAcme* es la de servir de representación intermedia de arquitecturas para pasar de *xArch* a *Acme*, y viceversa.

Una característica de *xAcme,* tomada de *Acme,* es que permite definir restricciones de diseño que se pueden aplicar a cualquiera de los elementos que forman la arquitectura. Sin embargo, al igual que *Acme*, *xAcme* se concibió únicamente como una notación de modelado y no proporciona un soporte directo para la generación de código.

Las cinco extensiones que forman *xAcme* son: extensión de propiedades, de familias, de restricciones, de propiedades *Acme*, y de restricciones *Acme*. Como base de todas estas extensiones se encuentra *xArch*, que representa la estructura de la arquitectura.

Para *xAcme* no existe una herramienta específica, sino que se puede usar *AcmeStudio*, herramienta propia de *Acme* que, en sus últimas versiones, permite importar y exportar descripciones en *xAcme.*

La comparación entre los dos últimos lenguajes descritos permite decir que:

- Tanto *ADML* como *xAcme* se basan en *Acme* y, por tanto, están muy relacionados; *xAcme* se considera a como una evolución de *ADML*.
- Las similitudes se pueden observar en el esquema de ambos lenguajes pues, en muchas ocasiones, los nombres de etiquetas para representar ciertos elementos en cada lenguaje coinciden.
- *ADML* codifica la representación correspondiente a *Acme 1.0*, mientras que *xAcme* adapta *xArch* a la representación de *Acme 3.0*.

2.2.5. LDA: conclusión y resumen

En este apartado se comparan los LDA estudiados y presentados en los apartados anteriores, considerando sus principales características, los conceptos que definen, su facilidad de uso, etc. A modo de resumen, en las tablas 2.2 y 2.3 que se presentan al final

de esta sección, se destacan las características generales de cada LDA. De los lenguajes estudiados:

- *Wright, Darwin, LEDA, PiLAR y Acme*, se basan en álgebra de procesos; *Rapide* y *UniCon* en eventos; y *xArch, xADL 2.0, xAcme* y *ADML* en esquemas XML.
- *Wright* está basado en CSP, lo que limita sus capacidades dinámicas; sin embargo, *LEDA, Darwin y PiLAR,* al utilizar el cálculo π, y, en menor medida *ArchJava,* basado en Featherweight, disponen de mayor capacidad de dinamismo.
- Los lenguajes basados en eventos son "más complejos", pero permiten definir arquitecturas con bastante dinamismo: *Rapide* permite representar modelos ejecutables (en C/C++, en ADA o en VHDL) en tiempo de diseño, que pueden simularse antes de empezar la implementación. *LEDA* permite generar un prototipo ejecutable en Java también durante el diseño.
- XML es un lenguaje sencillo, esquemático y fácil de entender. *xArch* y *xADL 2.0*, basados en esquemas XML, son, en principio, los más genéricos y no generan código; pero, mediante aplicaciones de *ArchStudio 3,* y asociando implementaciones a los distintos elementos de una descripción, se puede instanciar una arquitectura y comprobar así su funcionamiento. Dentro de este grupo, *xAcme* y *ADML*, que se basan en *Acme*, son, de momento, meramente descriptivos y tampoco generan código ejecutable (pero se les pueden asociar implementaciones a ciertos elementos de las arquitecturas). Para conseguir esta funcionalidad sería necesario crear las herramientas correspondientes y extender esos LDA.
- Los lenguajes basados en XML tienen como característica fundamental la facilidad de extensión, así que, de un modo sencillo, se pueden extender con nuevas funcionalidades. Pero para trabajar con las nuevas características añadidas sería necesaria la modificación del código fuente de las herramientas de soporte del LDA extendido.
- Por otra parte, los lenguajes *xArch*, *xADL 2.0*, *xAcme* y *ADML* están muy relacionados entre sí, de manera que se puede pasar de una representación en uno de ellos a una representación en cualquiera de los otros.
- Uno de los objetivos de los LDA basados en XML es llegar a ser un LDA estándar para representar cualquier arquitectura, de forma que se puedan extender fácilmente para adaptarlos a cada dominio de aplicación. Por ejemplo, *xArch* y *xADL 2.0* permiten describir arquitecturas *C2*. *xADL 2.0* tiene, además, muchas funcionalidades para usarse en arquitecturas de líneas de producto. *ADML* también es útil para las arquitecturas de líneas de producto y para arquitecturas *enterprise* (de empresa), todo ello promovido por el Open Group.

Se puede concluir que, a la hora de hacer el diseño arquitectónico de un sistema, no es sencillo decantarse por un LDA determinado por encima del resto. La elección depende de sus características y las circunstancias que rodean al sistema que se desea describir. Cada LDA tiene ventajas e inconvenientes y, además, éstos pueden variar según el dominio de aplicación en el que se use el lenguaje. Un ejemplo de la dificultad para elegir un LDA sobre otro es el elevado número de éstos que existe, y la falta de un lenguaje estándar que reúna todas las características deseables. Conseguir que un LDA se

convierta en un estándar, o al menos popular en el mundo de la Arquitectura del Software, es una tarea complicada que, de momento, no se ha conseguido.

Desde el punto de vista de la simplicidad y de sus posibilidades en el futuro, parece lógico decantarse por un LDA basado en XML. Entre ellos, *xArch* y *xADL 2.0* son, en principio, más genéricos y simples, y se pueden usar como punto de partida para otros. Además, se puede pasar de una representación en esos lenguajes a una representación en *xAcme* y *Acme* y, a partir de este último, a otros LDA, como es el caso de *ADML*, *Wright* y *Rapide*. Sin embargo, los lenguajes basados en XML carecen de una base formal fuerte que en ciertos casos es necesaria.

Por otra parte, a la hora de seleccionar un lenguaje u otro habría que considerar la posibilidad de que éste ofrezca una serie de características como son:

- Representación de sistemas de diferentes características.
- Generación código y obtención de prototipos ejecutables en tiempo de diseño.
- Soporte al análisis y validación de la arquitectura, concepto relacionado, en general, con la definición semántica del lenguaje basada en métodos formales o no.
- Adecuada gestión de la configuración de los sistemas descritos, en particular del dinamismo. Esta característica es especialmente importante en multitud de sistemas, donde la parada y reinicio para realizar modificaciones puede llevar a retrasos, riesgos inaceptables y a un incremento de los costes. En este sentido, la cualidad de dinamismo en un LDA se refiere a que éste debe proporcionar características específicas para los cambios en el modelo dinámico de los sistemas y técnicas para efectuarlo mientras éste se ejecuta.

Los LDA presentados en este capítulo pretenden ser un resumen del estado actual de esta herramienta de representación formal de las arquitecturas software. No se puede dejar de mencionar aquí la publicación de Medvidovic [MeTa00] en la que hace una revisión de los LDA desde un punto de vista histórico. En ella se propone un completísimo *framework* que sirve para comparar y clasificar diversos LDA. Por otra parte, Medvidovic en [MeDaTa07] presenta una nueva perspectiva sobre la arquitectura del software y los LDA: revisa los lenguajes denominados de primera generación y considera tres farolas (*lamppost*) como aspectos adicionales a tener en cuenta en la definición de los LDA de segunda generación. Esta nueva perspectiva proporciona una base más fuerte para el tratamiento de los LDA.

Del estudio de los lenguajes que se han presentado en este capítulo se puede concluir que cada uno dispone de un juego de características que hace interesante su utilización en determinados ámbitos de trabajo. Para cubrir los objetivos propuestos en esta tesis doctoral se debería elegir un LDA con, al menos, las siguientes propiedades:

- Generación de código y ejecución de prototipos.
- Una definición semántica que lo dote de fuerte apoyo formal.
- Dinamismo.

Según esto, y entre los LDA estudiados, se ha escogido LEDA como el lenguaje más adecuado para definir a partir de él un LDA Orientado a Aspectos, uno de los objetivos fundamentales de esta tesis doctoral.

LDA	*Basado en otro*	*Descripción*	*Dominios de aplicación*	*Herramientas*	*Generación de código*	*Dinamismo*
Rapide	Estructura compleja. 5 lenguajes subyacentes	LDA de propósito general. Concurrente y O.O. Modela computaciones e interacciones de sistemas mediante un conjunto parcialmente ordenado de eventos	Arquitecturas dinámica y de sistemas distribuidos. Modelado y simulación de sistemas	Aladdin	Permite construir un modelo ejecutable que puede simularse en t. de diseño (C++, Ada, VHDL)	Dinamismo restringido. Aunque los comp se pueden sustituir, la arquitectura no cambia. Se tiene reconfiguración programada basada en restricciones *ad-hoc*
Darwin	No	Lenguaje concurrente y O.O. Admite notación textual y gráfica. Todo es un componente, no hay conectores	Sistemas distribuidos	Visuales e interactivas. SAA	Configuración ejecutable de caja negra. Código C++	Dinamismo restringido
Wright	No	Sintaxis muy formal. Elementos: componentes, conectores y restricciones Fácil gestión de estilos	Sistemas estáticos	No	No	Limitado en Wright dinámico
ArchJAva	Java	Amplía Java. Las AS sor una jerarquía de instancias de componentes. Elementos componentes, puertos y conexiones	De propósito general. Verifica integridad de la comunicación	*Plataforma ECLIPSE. Archjava/doc* se ha desarrollando para facilitar la documentación gráfica	Genera Código Java	Facilita la definición de arquitecturas dinámicas. Creación dinámica de componentes
Acme	No	Propuesto inicialmente como lenguaje de intercambio. Actualmente es LDA	Para todo tipo de sistemas	*AcmeStudio* y otras	Se pueden construir config ejecutables pero no proporciona soporte directo para generar código	No. Representa las arquitecturas como estructuras estáticas, también se puede usar para representar arq. reconfigurables
Unicon	No	Soporta estilos y construcción de S a partir de descripciones arquitectónicas. Permite generar métodos de análisis en t. real	En principio cualquiera	Herramientas de modelado grráfico	Genera código C	No. Carece de capacidad de evolución
C2SADL	No	Basado en el paso intensivo de mensajes	Integración de componentes COTS en arquitecturas generativas	Saage	Genera código Java	Si. Inserción de elementos, borrado y reescritura
Leda	No	Todo es un componente. No hay conectores. Componentes parametrizables y extensibles. Definición de adaptadores	Arquitecturas dinámicas	EVADeS: Entorno Visual de Asistencia al Desarrollo de Sistemas para LEDA	Si. Genera prototipo del S en Java, en t. de diseño	Restringido. No son necesarias operaciones de reconfiguración. Creación dinámica de componentes
PiLAR	No	Lenguaje reflexivo y dinámico	Arquitecturas dinámicas y entornos reflexivos	No	No	Sí. Lenguaje altamente dinámico
xArch	No. Es el núcleo de xADL 2.0	Define la arquitectura de instancias de un sistema (t. de ejecución). Su extensión permite definir la estructura del sistema (t. de diseño)	Muy genérico para poder describir cualquier tipo de arquitectura, en particular en C2	*ArchStudio 3.* Para definir arquitecturas, analizarlas y aplicarles tests	Se pueden instanciar arquitect usando *ArchStudio 3.* Se puede representar el modelo de ejecución. No generan código	Usando ArchStudio se pueden instanciar arquitecturas y comprobar su funcionamiento
xADL2.0	Extensión de xArch	Define la estructura de un sistema (t. de diseño) y su arquitectura de instancias (t. de ejecución). Facilita la definición de estilos. Gran capacidad de extensión	Genérico para poder describir cualquier tipo de arquitectura, en particular C2.Adecuado para arquitect de líneas de producto	*ArchStudio 3.* Para definir arquitecturas, analizarlas y aplicarles tests. *Mènage* para líneas de producto	Se pueden pasar las especific en xADL a código ejecutable. Permite modelado en t. de diseño y de ejecución. Gestiona la config de la arquitectura. No generan código	Usando ArchStudio se pueden instanciar arquitecturas y comprobar su funcionamiento
xAcme	Inicialmente leng de intercambio Acme-xArch	Extiende xArch para adaptarlo a Acme. Permite definir restricciones de diseño aplicable a cualquier elemento de la arquitectura de un S	En principio, para todo tipo de sistemas	*AcmeStudio*, permite importar y exportar descripciones xAcme	No proporciona soporte directo para la generación de código	Son sólo descripciones arquitectónicas
ADML	Representación en XML de Acme	Lenguaje de representación de arquitectura. Pretende introducirse en la industria. Pretende ser un estándar para las arquitecturas de líneas de producto	Arquitect de líneas de prod y de empresa. Adaptable a la representación de S mediante *building blocks*. Usa reposit comerciales	*TOGAF*, facilita la descripción de sistemas en ADML	Sólo son descripciones arquitectónicas. No generan código ejecutable. Necesario usar las herramientas adecuadas	No

Tabla 2.2. Características generales de los LDA estudiados.

LDA	*Interfaces*	*Tipos*	*Comunicación interna*	*Vistas del sistema*	*Representación Textual/Grafica*
Rapide	Mediante reglas de conexión	Definición de tipos arquitectónico	Broadcast y eventos	Multiples vistas	T, G
Darwin	Mediante los servicios proporcionados y requeridos	Sistema de tipos extensible y parametrizables	Mediante canales	Una sola vista	T, G
Wright	Mediante Roles. Los roles comunican los componentes a través de sus interfaces	Tipos de componentes extensible	Relación Port-Rol	-	T
ArchJava	Puertos	-	Mediante conectores	Una sola vista	T
Acme	Mediante Roles. Los roles comunican los componentes a través de sus interfaces	Tipos extensibles. Definición de plantillas	Mensajes entre componentes	Una sola vista	T
Unicon	Mediante Roles. Los roles comunican los componentes a través de sus interfaces	Predefinidos	Eventos	-	T, G
C2SADL	Puertos de E/S	Tipos extensibles	Eventos	Una sola vista	T, G
Leda	Los roles comunican los componentes a través de sus interfaces proporcionadas y requeridas	No permite la definición de tipos nuevos	Eventos	Una sola vista	T, G
PiLAR	Puertos	El lenguaje define tres tipos de componentes	Eventos	Representación multicapa del sistema. Reflexión	T
xArch	Los puntos de interacción son las interfaces de tipo *in*, *out* o *inout*	Posibilidad de definir tipos de elementos arquitectónicos incluida en la extensión	Flujo de información a través de las interfaces	Vista del modelo de ejecución y existe una extensión que permite la vista en tiempo de diseño	T
xADL 2.0	Los puntos de interacción son las interfaces de tipo *in*, *out* o *inout*	Permite definir tipos de elementos arquitectónicos. Se pueden describir grupos de tipos y de elementos	Flujo de información a través de las interfaces. Proporciona funcionalidad para el envío de mensajes	Permite la vista del modelo de diseño y del modelo de ejecución	T
xAcme	Las interfaces se definen como roles (para conectores) y puertos (para componentes)	Definición de tipos de elementos arquitectónicos. Definición de familias (conjuntos de tipos y otros elementos)	Flujo de información a través de las interfaces. No permite definir mensajes ni eventos	Permite varias vistas (característica tomada de Acme)	T
ADML	Las interfaces se definen como roles (para conectores) y puertos (para componentes)	Permite definir tipos y familias (como conjuntos de tipos y otros elementos)	Flujo de información a través de las interfaces. No permite definir mensajes ni eventos	Varias vistas (característica tomada de Acme) y varias opciones de diseño	T

Tabla 2.3. Representación de conceptos arquitectónicos en los LDA estudiados.

2.3. Arquitectura del software dinámica

La complejidad de los sistemas a desarrollar crece continuamente, así como la variedad de temas que se tratan en esos desarrollos. Muchos de estos sistemas son altamente dinámicos (sistemas abiertos, distribuidos, concurrentes, en red, etc.) y deben tener una estructura que permita que cambien y evolucionen fácilmente. Estos cambios deben considerarse como elementos estructurales durante el diseño del sistema, no una cuestión coyuntural: cuando la evolución de una arquitectura es continua y los cambios en su interior siguen un patrón bien definido, ese patrón debe considerarse que forma parte intrínseca de la estructura.

Por otra parte, como el objetivo de la arquitectura del software es describir la estructura de los sistemas, ésta debe incluir tanto sus partes fijas -estáticas- como las cambiantes –dinámicas. De hecho, en ciertos sistemas, los aspectos dinámicos son más importantes que los estáticos, siendo necesario realizar la especificación arquitectónica con patrones dinámicos. Además, también es necesario poder expresar el dinamismo en la AS de un modo formal. Sin embargo, los LDA específicos para arquitecturas dinámicas son pocos. Se pueden citar *Darwin, Rapide, C2*, la versión dinámica de *Wright, LEDA,* o *PiLAR*.

En esta sección se hace una revisión de los conceptos y modelos de las arquitecturas del software dinámicas. Algunas veces la *arquitectura del software dinámica* se denomina *reconfiguración dinámica,* concepto que se ha usado en el campo de la configuración de sistemas distribuidos. De un modo general, se entiende por *reconfiguración dinámica* como aquellos cambios que se producen en la topología de un sistema compuesto. Es decir, cualquier alteración en el número de los nodos y enlaces que lo definen. Este es el grado de dinamismo que actualmente se está alcanzando en los LDA. Sin embargo, según Kramer [KrMa90], el nombre de *arquitectura dinámica* debería reservarse para referirse a los sistemas en los que incluso el tipo de componentes implicados puede cambiar (como por ejemplo en *PiLAR*).

El estudio de las *arquitecturas del software dinámicas* permitirá, por una parte, tener un conocimiento preciso de las características de los sistemas dinámicos, desde el punto de vista de su arquitectura; y por otra, establecer cómo definir modelos arquitectónicos que los soporten, como el que se propone en esta tesis doctoral.

2.3.1. Tipos de dinamismo

Todo sistema software, cuya estructura se describe mediante una arquitectura, tiene siempre algún tipo de comportamiento dinámico: algunos sistemas muestran una estructura inalterable, mientras que otros se disponen según una topología interna que cambia continuamente, variando incluso el número de elementos que lo componen. Así, los sistemas dinámicos se pueden clasificar por el tipo de dinamismo.

Arquitectónicamente se pueden identificar tres tipos, según la clasificación realizada por [Cue+01], lo que permite reflexionar sobre el tipo de actividad de cada tipo de sistema. Atendiendo a su potencial ante los cambios, se consideran los distintos tipos:

- *Dinamismo interactivo*. Este término se refiere al movimiento de datos entre componentes. Es el dinamismo que se produce normalmente en la comunicación e interacción entre los elementos de cualquier sistema. Está relacionado con cuestiones tales como la sincronización y los protocolos de comunicación, y afecta al comportamiento de un sistema, no a su estructura. Todo sistema tiene este tipo de dinamismo, pero resulta especialmente obvio en los *concurrentes* y *paralelos*.
- *Dinamismo estructural*. Se refiere a los cambios que afectan a los componentes de un sistema y sus interacciones. Este tipo de dinamismo se suele expresar en la creación y destrucción de instancias y enlaces entre componentes. Por tanto, lo que cambia es la estructura: la arquitectura del sistema se ve alterada. Es característico de los sistemas *distribuidos*.
- *Dinamismo arquitectónico*. Se refiere a los cambios que se producen en los tipos de componente y de interacción, modificándose así hasta el significado y el "vocabulario" del sistema. Se refiere también a la inserción de nuevos elementos de componente (y de conector) y el borrado de algunos existentes. En este caso, lo que cambia es la infraestructura, la base en la que las arquitecturas definen su meta nivel. Este es el mayor nivel de dinamismo y el más difícil de utilizar. Este tipo de dinamismo se da en los *sistemas abiertos*.

Cada nivel incluye a los anteriores. El primero existe entre los elementos arquitectónicos del sistema, por lo que su arquitectura no se ve afectada. El segundo manipula componentes arquitectónicos, produciéndose variaciones (distintas configuraciones) de una única descripción arquitectónica (estilo). El tercero modifica las arquitecturas como un todo y crea otras arquitecturas nuevas. Cualquier sistema presenta dinamismo interactivo; los sistemas compuestos, incluidos los distribuidos y los reconfigurables, tienen dinamismo estructural; finalmente el dinamismo arquitectónico se refiere a aquellos sistemas con capacidades reflexivas.

La mayoría de los LDA actuales permiten expresar el dinamismo del primer tipo y pocos permiten expresar el dinamismo estructural, entre ellos, *Darwin*. En lo referente al tercer tipo, *Rapide, LEDA* y *PiLAR* son, al menos, parcialmente compatibles con él.

2.3.2. Tipos de reconfiguración

Se pueden distinguir varios tipos de reconfiguración, que representan problemáticas diferentes:

1) *Según el tiempo*: propuesta por Wermelinger [Wer99] que considera que uno de los aspectos críticos en un cambio es *cuándo* se produce. La dificultad para realizarlo varía según el momento del desarrollo en el que se realice:
   - *Antes de la compilación*. Los cambios en la estructura prevista del sistema se producen antes de su ejecución, por lo que sólo hay que asegurar su consistencia.

- *Antes de la ejecución*. El sistema ha sido construido, pero aún no se ha puesto en funcionamiento; introducir un cambio sólo requiere asegurar la consistencia. Este caso es similar al anterior.
- *Durante la ejecución*. Ésta se puede considerar como la auténtica *reconfiguración dinámica*: los cambios se producen con el sistema en funcionamiento, los componentes tienen un estado y sus enlaces soportan un cierto número de transacciones.

2) *Según el origen.* Clasificación propuesta por Endler [EnWe92] y se refiere a *cómo* se inicia el proceso de cambio; depende de quién tome la iniciativa:
   - *Reconfiguración programada*. Se denominan así las operaciones que inicia el propio sistema, a partir de una serie de reglas que forman parte de la especificación. También se ha denominado *reconfiguración restringida* en tiempo de ejecución y es el nivel que se debería alcanzar en la AS.
   - *Reconfiguración ad hoc* o *reconfiguración evolutiva*. Se refiere a los cambios que introduce manualmente el usuario y requiere algún tipo de herramienta que haga corresponder la arquitectura con un sistema real. Estos cambios no se pueden prever, pero en general sí pueden controlarse.

2.3.3. Modelos de dinamismo arquitectónico

En este apartado se presenta, de modo resumido, una taxonomía de los enfoques existentes para la descripción de arquitecturas de los sistemas dinámicos basada en cómo se expresa el dinamismo. Esta clasificación ha sido realizada por C. Cuesta y presentada en [Cue02] y [Cue+01] con distinto nivel de detalle, y, según el propio autor, no debe considerarse estricta. Los sistemas se consideran a alto nivel y su objetivo es facilitar la comparación con sus equivalentes reflexivos, realizada también en [Cue02]. Este estudio concluye que existe una interesante relación entre el dinamismo arquitectónico y la reflexión, como se muestra en el apartado 2.4.2.

Los modelos de dinamismo arquitectónicos identificados pueden agruparse en tres grandes bloques:

1) *Supervisor*: Este grupo caracteriza a aquellas arquitecturas en las que hay algún tipo de entidad *omnisciente* que gestiona la dinámica del modelo y decide los cambios relevantes. Este enfoque es el más utilizado y se puede observar desde diferentes perspectivas:
   - *Configurador*: El supervisor se modela de manera explícita y se incluye como un componente especial de la arquitectura, cuya única función es modelar y gestionar la propia arquitectura. Ejemplos de este tipo son el *configuror* en la versión dinámica de *Wright* o el tipo *architecture* en *HOT Acme*.
   - *Coordinación*: En este modelo se supone la existencia de un "espacio" externo con ciertas reglas que gobiernan la evolución dinámica de la arquitectura. En este caso el supervisor (que se encarga de hacer cumplir las reglas) es implícito y no se puede identificar un único elemento que ejerza esta función. Se puede dividir a su vez en:

  - Espacio común: El esquema de coordinación se realiza sobre un ámbito común, *memoria compartida*; ejemplo es el modelo de tuplas de Linda [Gel85].
  - Control externo: Se refiere a aquellos modelos en los que no existe un supervisor explícito, pero sí se modela el dinamismo que es controlado por una entidad externa. Son aquellos que son controlados de manera activa. Es el caso de Manifold [ArHeSp93].
- *Oráculo*. En este enfoque la entidad externa es completamente ajena al sistema y no puede ser modelada. Puede intervenir en cualquier momento pero es impredecible. Este grupo reúne todas las soluciones *ad hoc* basadas en intervención humana. Es el caso de *C2SADL* y *Darwin*, junto con las herramientas software que permiten la interacción del usuario.

2) *Componente inteligente.* En este bloque, la definición del LDA proporciona a los componentes una serie de primitivas que les permiten alterar la estructura del sistema o parte de ella. Así, los componentes realizan su propia computación y las tareas de reconfiguración. Ejemplo son algunos de los LDA con mayor capacidad de dinamismo como *Darwin* o *LEDA*.

3) *Conector inteligente*. Este grupo reúne las propuestas que provocan cambios en los sistemas utilizando o definiendo alguna característica de su medio de interacción, en particular de sus conectores. Aquí se pueden incluir los sistemas de eventos, como *Rapide*, y los sistemas con bus común, como los conectores de *C2SADL*.

Esta clasificación es similar a la aplicada en el estudio de los modelos reflexivos (apartado 2.4.2) y permite justificar la afirmación: *todo sistema con dinamismo arquitectónico se puede describir en términos de reflexión* [Cue02].

A lo largo de este apartado se han revisado diversos enfoques relacionados con la arquitectura dinámica. Igualmente se han presentado dos clasificaciones para ellos, según su origen y según su estructura interna, mostrándose someramente su gran diversidad. Finalmente, se ha presentado una taxonomía para modelos de dinamismo arquitectónico que será útil para estudiar los modelos reflexivos.

2.4. Conceptos relacionados

En esta sección se comentan brevemente dos conceptos relacionados con la arquitectura del software como son la evolución de los sistemas y la reflexión.

2.4.1. Arquitectura del software y evolución

Es un hecho que el mundo real evoluciona y los sistemas software tienen que adaptarse a unos requisitos que cambian a lo largo del tiempo, bien sea por modificación y adaptación de los requisitos del sistema, o porque es necesario añadir nuevas funcionalidades no previstas inicialmente. Por esta razón, es necesario diseñar los

sistemas de modo que se puedan capturar los cambios tanto del entorno como de los requisitos. Estos cambios muchas veces suponen la redefinición del diseño del sistema añadiendo, eliminando o cambiando sus elementos estructurales, y/o redefiniendo la interconexión entre ellos. Sin embargo, la arquitectura software de un sistema debería ser esencialmente estable a pesar de los cambios. Considerar el punto de vista arquitectónico en la evolución de los sistemas puede reducir muchas de sus dificultades.

Dado que esta tesis doctoral se incluye en el ámbito de la *arquitectura del software,* no se puede dejar de comentar aquí la evolución desde este punto de vista, pues el modelo arquitectónico que se propone puede facilitar la evolución de los sistemas, cuando se desea incluir nuevas características que se puedan considerar ortogonales a las existentes en el sistema actual.

La evolución del software es un concepto ampliamente debatido en los últimos 30 años; concretamente, el problema de la evolución del software ha sido extensamente estudiado por Lehman [Leh96], quién definió la Primera Ley de la evolución del software:

> *Un programa (sistema) que se usa debe ser adaptado continuamente para que su ejecución sea satisfactoria y se acomode a los cambios del entorno*.

Actualmente la evolución es uno de los mayores retos que afectan a la confiabilidad de los sistemas, así como a las actividades de la ingeniería del software y su entorno [Fel03]. Frecuentemente se piensa que:

> *... La evolución es un fenómeno que debería evitarse; sin embargo, la evolución debe tratarse como una característica propia de los sistemas*... [Fel03].

Esta necesidad de que el software cambie y evolucione continuamente supone un importante desafío para los ingenieros de software que deben buscar técnicas y herramientas que faciliten el cambio de los sistemas. Por ello, los paradigmas que se utilicen en su desarrollo tienen que considerar la evolución como una parte del ciclo de vida ya que, por una parte la evolución es inevitable y, por otra, los sistemas se van degradando, reduciéndose su confiabilidad. Además, hay que tener en cuenta que la evolución puede ocurrir durante diversas fases del ciclo de vida, desde etapas tempranas (evolución de requisitos) hasta su implementación y explotación (mantenimiento correctivo y perfectivo). Por otra parte, en [CaPiAn05] la evolución del software se define como:

> *Un tipo de mantenimiento del software que tiene lugar después del desarrollo inicial de un sistema.*

De cualquier modo, su objetivo es *adaptar un sistema a los cambios que se produzcan, tanto de los requisitos de usuario como del entorno operativo, casi siempre de un modo no previsto*. De aquí se podría deducir que frecuentemente la evolución del software implica el rediseño del sistema completo, el desarrollo de nuevas características y su integración en el sistema existente. Por eso, la evolución no debe restringirse sólo al nivel de implementación. Diversos trabajos presentados en talleres y conferencias internacionales, como en el *workshop* sobre coordinación y técnicas de adaptación (WCAT de 2004 [WCAT04] y siguientes) o la conferencia europea sobre mantenimiento

del software y reingeniería (CSMR) proponen elevar el nivel de abstracción en el que se realiza o se produce la evolución, de modo que se pueda prescindir de detalles de implementación. Además, no considerar que pueda haber diferentes tipos de evolución puede dar lugar a problemas prácticos. Según estos trabajos, parece evidente la necesidad de mejorar la adaptabilidad del software, sin que ello suponga un gran impacto en el sistema mismo. La habilidad de predecir cómo será la evolución de un sistema es crucial en la definición de éstos, ya que representa la posibilidad de extenderlos. Aproximaciones tradicionales (programación estructurada o el paradigma OO) sólo permiten una reconfigurabilidad y reutilización parcial del sistema.

De cualquier modo, la evolución del software y el mantenimiento se caracterizan por su alto costo (por encima del 60% del coste total del desarrollo el sistema) y su baja velocidad de implementación, por lo que *sería especialmente interesante considerar la evolución a lo largo del ciclo de vida de los sistemas* [Fel03]. Desde este punto de vista evolutivo, el espacio de evolución se puede considerar:

- *La propia evolución del software* al cambiar el entorno.
- *Evolución de los requisitos*. Los requisitos expresan las necesidades de los usuarios por lo que parece lógico que sea en este punto donde se capture la información de la evolución de los sistemas. Además, la evolución de los requisitos se ha considerado principalmente como un problema de gestión, en lugar de considerarse como una característica propia de los sistemas. Hasta hace poco, en la ingeniería de requisitos, en general, se hacía escaso énfasis en las características evolutivas del software, a pesar de que se sabe que es imposible congelar los requisitos. Sin embargo, sí es posible analizarlos para determinar cuales evolucionan, establecer cuales son los más estables y cuales los que más pueden cambiar. Así, se puede estudiar la evolución de los requisitos atendiendo a diversos aspectos: tipo de requisito, dependencias entre ellos, tipos de cambios (añadirlos, cambiarlos, modificarlos) y trazabilidad de los requisitos.
- *Evolución de la arquitectura*. Describe cómo se percibe la evolución a nivel de diseño. Es complicada ya que puede suponer cambiar la definición estructural del sistema. Por tanto, es necesario controlar la variación de los puntos específicos que pueden caracterizar los sistemas.
- *Otro tipo de evolución* considera los factores humanos y enfatiza en la interacción entre el sistema y su entorno.

La necesidad de tener en cuenta la evolución en la arquitectura del software ha sido reconocida en la norma IEEE 1471 2000 (sección 2.7) donde se define la arquitectura como:

> *La organización fundamental de un sistema descompuesto en sus componentes, sus relaciones (entre sí y el entorno) y los principios que guían su diseño y evolución (configuraciones)*,
>
> y además:
>
> *Proporciona una base fuerte para soportar la evolución del software*.

Por otra parte, la descripción arquitectónica de un sistema debe definir cada vista que contiene. Desde el punto de vista de la evolución, las vistas que aportan mayor información sobre cómo cambia el software son las vistas conceptual, lógica y dinámica; la *vista conceptual* actúa como un límite sobre las decisiones de diseño para asegurar, entre otras cosas, un desarrollo económico; la *vista lógica* transforma las características del sistema (visto por el usuario) en una vista de alto nivel de áreas del software; la *vista dinámica* considera las características de interacción.

En el 4º *workshop* sobre evolución arquitectónica [WCAT04] desarrollado en el marco de ECOOP 2004 y para garantizar que una arquitectura software sea robusta respecto de la evolución, de modo que cambie lo menos posible, se propuso la siguiente definición que enfatiza estos requisitos:

> *Una arquitectura software es una colección de clases o categorías de elementos que comparten la misma probabilidad de cambio. Cada categoría contiene elementos software que muestran características de estabilidad similares*.

Además, una arquitectura software contiene un núcleo central que representa el nivel más estable frente al cambio. En él se identifican las características (restricciones) que no pueden cambiar sin reconstruir el sistema completo. Sin embargo, según se ha dicho, a lo largo del tiempo las restricciones arquitectónicas también pueden cambiar. Para que la evolución de la AS se realice de un modo correcto, la arquitectura tiene que especificarse previamente de un modo adecuado, por ejemplo mediante lenguajes de descripción arquitectónica. Los tres elementos fundamentales de la AS (componentes, conectores y configuraciones) pueden evolucionar independientemente; hasta ahora, la mayor atención se ha puesto en el estudio de la evolución de los componentes y las configuraciones.

Entre los trabajos relativos a la evolución se pueden mencionar aquellos presentados en los *workshops* sobre el tema que, desde 2001, se celebran asociados a las conferencias del ECOOP y otras conferencias internacionales como el ICSE. En ellos se proponen diversas formas de elevar el nivel de abstracción para soportar la evolución del software. Se pueden citar:

- Andrade [And+01] propone desacoplar computación, configuración y coordinación a nivel de programación, de modo que las reglas de negocio se puedan expresar en dos niveles para la evolución arquitectónica, en la vista de comportamiento de un sistema: un nivel para gestionar las interacciones y otro para manejar las computaciones. Esto facilita la evolución de los sistemas porque los cambios del comportamiento pueden referirse a modificaciones sobre cómo los componentes interactúan y no sobre el cambio de los propios componentes.
- Heckel [HeMeWe02] define, sobre un metamodelo, un nivel extra de abstracción. Esto tiene un efecto adicional que puede ayudar a proporcionar un soporte dominio-independiente para la evolución del software.
- Zhao [Zha02] eleva el nivel en el cual se ejecuta el análisis del impacto del cambio (del nivel de implementación al nivel de arquitectura). El objetivo es no sólo llegar a tener una mejor idea conceptual del problema sino fijarse en los niveles en los que los cambios pueden tener un mayor impacto sobre el sistema.

Como conclusión a lo expuesto en este apartado, se puede decir que el problema de la evolución de los sistemas software tiene que abordarse desde el principio del desarrollo, considerando que los cambios son inevitables e incluso anticipándose a las modificaciones no previstas, de modo que éstas no supongan cambios estructurales en el sistema construido. El interés de su estudio en el marco de esta tesis doctoral se debe a que el tema se aborda considerando la evolución de los requisitos, gestionándolo como un problema estructural. La evolución se trata desde un punto de vista arquitectónico y cuando se produce una variación en los requisitos (ya sea añadiendo nuevos o modificando los actuales).

2.4.2. Reflexión

El campo de la *reflexión computacional* es muy amplio por lo que este apartado sólo contiene algunos aspectos generales del mismo, pues sólo se pretende dar una visión reducida del tema, en particular en su relación con la arquitectura del software y el modelo arquitectónico propuesto en esta tesis.

El concepto de *reflexión computacional* (o sólo *reflexión*) fue definido por Smith [Smi82] y se refiere a los sistemas que tratan y actúan sobre sí mismos. Definiciones posteriores, como la de Maes [Mae87] y la de Watamnabe y Yonezawa [WaYo88] vienen a decir que la *reflexión* es:

> *La capacidad de un sistema computacional para razonar y actuar sobre sí mismo, así como para ajustar su comportamiento a las condiciones de cambio*.

En este sentido, como la *reflexión* se refiere a la capacidad de un sistema para observar y opcionalmente modificar su estructura de alto nivel, se podría aplicar para resolver el problema genérico de la creación de aplicaciones flexibles, fáciles de mantener, de usar y capaces de ser extendidas y/o modificadas.

El dominio computacional de un sistema *reflexivo* –el conjunto de información que computa- añade al dominio de un sistema convencional la estructura y el comportamiento de sí mismo. Así, un sistema *reflexivo* es capaz de acceder, analizar y modificarse a sí mismo, como si se tratase de una serie de estructuras propias de un programa.

La *reflexión* produce la división implícita de un sistema en un *nivel base,* que realiza la actividad normal, y uno o varios *niveles meta* que observan y modifican al inferior:

- El *sistema base* es el motor de computación de un programa.
- El *meta-sistema* es el sistema de computación que tiene por dominio de computación a otro sistema.

Si se supone que el prefijo *meta* hace referencia al nivel de abstracción en el que se definen las reglas que gobiernan el sistema, suponiendo también que éste adopta igualmente la forma de un sistema, entonces, un *meta-sistema* es aquel que actúa sobre otro sistema (*sistema base* o *sistema objeto*). El paso a la reflexión a partir de estos conceptos es sencillo [Mae87]:

> *Un meta-sistema conectado causalmente con un sistema objeto que es parte de sí mismo se denomina sistema reflexivo, y el acto de razonar y actuar sobre sí mismo se denomina computación reflexiva*.

La característica esencial que define a un sistema *reflexivo* y lo distingue de cualquier otro *meta-sistema* es el concepto de *conexión causal*:

> *Un sistema tiene conexión causal si su dominio computacional contiene el entorno computacional de otro sistema y, al modificar la representación de éste, los cambios realizados causan un efecto en la computación del sistema base* [Mae87].

Según esto, se debe entender por *conexión causal* a la conexión existente entre el *sistema base* y el *meta-sistema*. A su vez, todo cambio realizado por el *meta-sistema* sobre la meta-representación ha de tener un efecto instantáneo sobre el *sistema base*. Dicho de otro modo, la *conexión causal* expresa la conexión entre un sistema reflexivo y la parte de él mismo que lo utiliza como objeto de su computación. Para ello, un sistema *reflexivo* ha de tener un modelo interno en el que se represente a sí mismo, y de este modo está conectado causalmente con él. Esto significa que cualquier cambio en el modelo se reflejará en el sistema y viceversa. Además, se puede adaptar a condiciones cambiantes, lo que proporciona una gran flexibilidad.

Una de las razones para incorporar *reflexión* a los sistemas de computación es facilitar que la computación del *nivel base* se lleve a cabo de la forma adecuada. Para ello, es necesario que la organización interna del sistema pueda cambiar, de modo que se facilite el acceso a esta estructura interna como si fuesen datos; disponiendo de los mecanismos necesarios para razonar sobre su estado actual y permitiendo su modificación, de modo que se adecue a las nuevas condiciones.

En los últimos años, la *reflexión* se ha aplicado a los modelos y marcos de integración de componentes, como CORBA, .NET o el entorno de Java; a los sistemas de *middleware* reflexivo; o a los modelos basados en aspectos. También se ha aplicado a UML de un modo explícito.

2.4.2.1. Arquitectura de Meta-Nivel

Hay tres descripciones asociadas con la infraestructura de un sistema *reflexivo*: meta-arquitectura, arquitectura de meta-nivel y arquitectura reflexiva. En [Mae87] se describen como sigue:

a) Se denomina *meta-arquitectura* a la arquitectura o estructura que se utiliza para dar soporte a la definición y construcción de otras arquitecturas. Este es el sentido etimológico del término y podría considerarse como sinónimo de infraestructura. Se puede referir a aspectos físicos, organizativos o conceptuales, y no tiene por qué ir asociada a un sistema reflexivo. Sin embargo, en este contexto, se refiere al contenido del nivel meta –o de todos los niveles meta-.
b) La expresión *arquitectura de meta-nivel*, de un modo literal, describe la estructura –arquitectura- de los elementos del nivel meta. Sin embargo, no se debe entender así este concepto. Realmente el término *arquitectura de meta-nivel* representa la

estructura –arquitectura- que describe la relación entre el *nivel base* y el *meta*, que se podría interpretar como los mecanismos que utiliza un sistema *reflexivo* para proporcionar la reflexión.

c) Se denomina *arquitectura reflexiva* a la estructura –arquitectura- de un sistema *reflexivo*; es decir, al conjunto formado por el *nivel base*, el *nivel meta*, y la relación entre ellos (Figura 2.3).

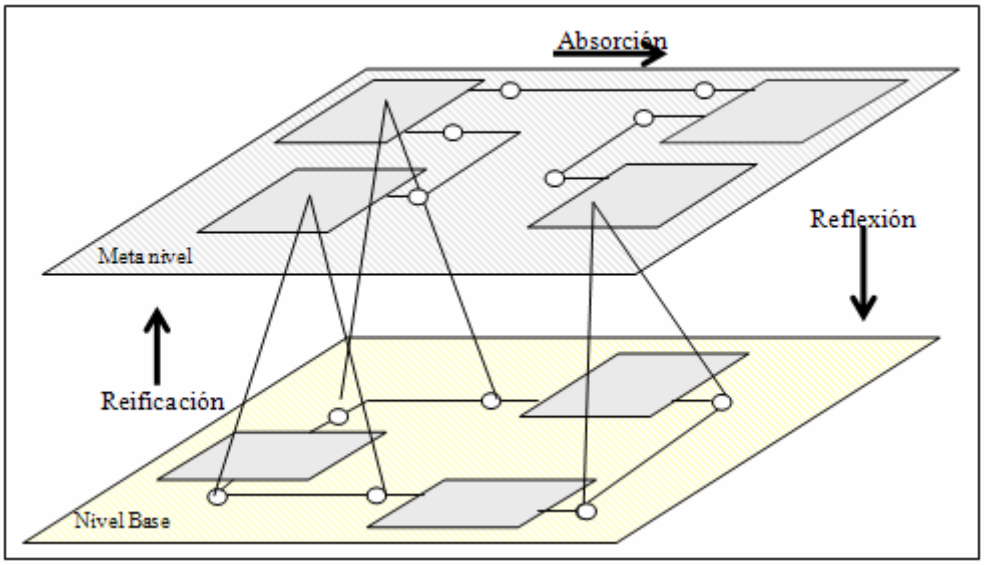


Figura 2.3. Modelo de reflexión de 2 niveles.

Estrictamente hablando existe una diferencia entre los dos últimos términos. En general se puede decir que una *arquitectura de meta-nivel* es aplicable a todo meta-sistema, mientras que la *arquitectura reflexiva* sólo es aplicable a los sistemas *reflexivos*. Maes en [Mae87] dice que:

> *La diferencia principal entre una arquitectura de meta-nivel y una arquitectura reflexiva es que la arquitectura de meta-nivel sólo proporciona acceso estático a la representación del sistema computacional, mientras que la arquitectura reflexiva también proporciona un acceso dinámico.*

Es decir, una *arquitectura de meta-nivel* sólo proporciona *introspección*, mientras que la *arquitectura reflexiva* da soporte a cualquier tipo de reflexión. Ambas expresiones se vienen usando en los dos sentidos. Se suele utilizar el término *arquitectura de meta-nivel* para referirse al soporte genérico de reflexión en un cierto sistema, mientras que una *arquitectura reflexiva* se considera como la estructura de una instancia de ese sistema, y se refiere al uso concreto que hace de dicho soporte genérico.

Por definición, un sistema es meta-sistema de otro (*sistema base*), cuando opera sobre éste o más exactamente sobre su modelo. Cuando un sistema actúa como su propio meta-sistema, se dice que es *reflexivo*. Sin embargo, las dos vistas del mismo permanecen separadas conceptualmente como niveles, lo que permite manipularlas y razonar sobre ellas de manera independiente.

La relación que expresa la conexión causal entre los dos niveles se asimila a una operación de *reificación*[2] que asciende del *nivel base* al *meta*. En el sentido contrario, se llama *reflexión* a la relación que desciende del *nivel meta* al *base*.

2.4.2.2. Reificación

Este concepto relacionado con el de reflexión fue introducido por Wand y Friedman en [WaFr88]. Su definición queda expresada según el siguiente párrafo propuesto por Malenfant [MaJaDe96]:

> *La* reificación *es el proceso por el cual un programa de usuario P, o cualquier aspecto de un lenguaje de programación L, que estaban implícitos en el programa traducido y en el sistema de ejecución, se hacen explícitos, utilizando una representación (estructuras de datos, procedimientos, etc.) que se expresa en el propio lenguaje L y se pone a disposición del programa P, que podrán inspeccionarlos como si se tratase de datos ordinarios. En los lenguajes reflexivos, los datos obtenidos mediante la* reificación *están conectados causalmente con la información reificada correspondiente, de modo que una modificación en uno de ellos afecta al otro. De este modo, los datos obtenidos por la reificación son siempre una representación fidedigna del correspondiente aspecto reificado.*

Así, la *reificación* se refiere a la posibilidad de acceder desde un *sistema base* a su *meta sistema*, en el que se puede manipular el *sistema base* como si de datos se tratase. Si se puede *reificar* un sistema, se puede acceder y modificar su estructura y su comportamiento y, por tanto, se podrá añadir a su dominio computacional su propia semántica. Se estará trabajando entonces con un sistema *reflexivo*. Por tanto:

> *La reificación* es *la capacidad de un sistema para acceder al estado de computación de una aplicación, como si se tratase de un conjunto de datos propios de una aplicación de usuario* [Smi82].

Dicho de otro modo, la *reificación* es el proceso que permite a una cierta entidad "subir" del *nivel base* al *nivel meta*; y dado que todo cambio en el *nivel meta* ha de reflejarse en el *nivel base*, se dispone de un proceso inverso que permite "bajar" del uno al otro: *reflexión* . La *conexión causal* realiza el conjunto de ambas operaciones (Figura 2.3). El conjunto de operaciones realizadas en el *nivel meta* se puede denominar *absorción*, así se expresa que una parte de la computación ha sido absorbida por la meta-computación.

Teniendo en cuenta esto, cada operación reflexiva puede verse como un proceso en tres etapas: la *reificación* de la entidad, la *absorción* de una tarea dada, y la *reflexión* final que permite reflejar los resultados en el *sistema base*. A veces el *nivel meta* no existe como una estructura persistente, definida explícitamente como un fragmento del sistema,

[2] Reification: Este término se ha traducido por *reificación*, concretización, materialización o cosificación. Aquí se utiliza *reificación*.

sino que sólo se produce como un esquema o proceso de *reificación*, *absorción* y *reflexión* que realiza, en definitiva, una meta-computación intercalada en la ejecución del sistema.

2.4.2.3. Clasificación de la reflexión

Existen varias clases de *reflexión*, de modo que se puede dividir considerando diversos criterios. En este apartado se comentan someramente algunas de las clasificaciones que se refieren a la *reflexión* en el nivel de implementación. Se puede consultar una clasificación más completa en [Ort02]:

1) Por su efecto

Atendiendo al efecto que tiene el meta-programa sobre el programa, se pueden distinguir dos grandes grupos: *Introspección* e *Intercesión*. A su vez, la *Intercesión* puede ser *reflexión estructural* (lingüística) y de *comportamiento*:

- *Introspección* es la capacidad que tiene un sistema para observar y razonar sobre los objetos que lo forman, pero no los modifica. Esta capacidad tiene la ventaja de poder tomar decisiones en función de la información (meta-información). Este tipo de reflexión permite acceder al estado del sistema en tiempo de ejecución: el sistema ofrece la capacidad de conocerse a sí mismo.
- *Intercesión* es la capacidad de un programa para modificar su propio estado de ejecución – comportamiento- sin alterar su código.
  - Dentro de este grupo se dice que hay *reflexión estructural* cuando se modifica la estructura del *nivel base*. Mediante el lenguaje del *sistema base* se puede modificar el propio lenguaje [Ort02]. Malenfant [MaJaDe96] define de un modo formal la *reflexión estructural* como:

    La habilidad de un lenguaje para proporcionar una reificación completa del programa que se está ejecutando y de sus tipos abstractos de datos.

    También se denomina *reflexión lingüística*, pues es capaz de generar nuevos fragmentos de programa en funcionamiento e integrarlos en su ejecución.

  - Se habla de *reflexión de comportamiento* cuando lo que se altera es la semántica del *nivel base*, su modelo de ejecución; en definitiva, su comportamiento. Malenfant [MaJaDe96] define este concepto como:

    La reflexión de comportamiento se define como la habilidad del lenguaje para proporcionar una reificación completa de su propia semántica e implementación, así como una reificación completa de datos y la implementación del propio sistema de ejecución dentro del cual él mismo se ejecuta.

Ambos tipos de reflexión (*introspección* e *intercesión*) no son mutuamente excluyentes.

2) Por tiempo

Según el momento en el que se aplique la reflexión, se puede distinguir entre:

- *Reflexión en tiempo de compilación.* En este caso, un meta-programa M altera el modo en que se compila el programa P correspondiente, de modo que el código ejecutable que se genera es distinto del que inicialmente se había creado. Con distintos meta-programas se puede lograr que el mismo programa P tenga comportamientos diferentes, sin que se haya introducido ningún cambio en su código fuente. Que el acceso al *meta sistema* se produzca en tiempo de compilación implica que el sistema tiene que prever su flexibilidad antes de ser ejecutado. Una vez en ejecución no puede acceder a su *meta sistema* de un modo no contemplado en su código fuente.
- *Reflexión en tiempo de ejecución.* Esta es la auténtica *reflexión*. Mientras se está ejecutando un sistema sus operaciones son interceptadas por un meta-programa, que puede ser independiente, o bien una parte de su propio código[3]. En los sistemas con este tipo de reflexión, el acceso al *meta sistema,* su manipulación y el reflejo producido por un mecanismo de conexión causal se realiza en tiempo de ejecución. En este caso, el sistema es flexible de forma dinámica, es decir, se puede adaptar a eventos no previstos (en tiempo de compilación) cuando se esté ejecutando.

2.4.2.4. Reflexión y Arquitectura del Software

Algunos de los conceptos mencionados en el apartado anterior sobre reflexión computacional son compatibles con la arquitectura del software como el de arquitectura de *meta-nivel*, o *reflexión estructural*.

Las *arquitecturas reflexivas* proporcionan un nuevo modelo de computación en el que existe una representación de la arquitectura del sistema que puede ser observada y manipulada. En términos más generales se define como [Tis+01]:

La computación que realiza un sistema sobre su propia arquitectura software.

El propósito de definir una *arquitectura reflexiva* es garantizar el funcionamiento adecuado de un sistema y proporcionar un medio para implantar la computación reflexiva de manera modular, lo que hace que el sistema sea más manejable, comprensible y fácil de modificar.

Un sistema de *arquitectura reflexiva* está estructurado en dos niveles, llamados niveles arquitectónicos: un *nivel base* y un *nivel meta*:

- *Nivel base*: en [Tis+01] se dice que es el sistema *ordinario*. Es el que realiza una computación para resolver problemas y devolver los resultados al exterior. Las entidades que trabajan en el *nivel base* se denominan *entidades base.*

[3] En un sistema secuencial, el meta-programa se inserta como parte del programa y altera la ejecución del modo conveniente, capturando sus interacciones. En un sistema concurrente, los dos programas pueden ser independientes, pero uno de ellos supervisa el comportamiento del otro y le obliga a seguir una evolución determinada. Esto coincide con el concepto de superposición de Katz [Kat93], definido en la teoría de concurrencia y que no partía de consideraciones reflexivas.

- *Nivel meta* o *parte reflexiva*: en [Tis+01] se establece que es el que mantiene conectado causalmente objetos reflejando la arquitectura del *nivel base*; el dominio del *nivel meta* es la arquitectura software del *nivel base*; mantiene información y describe cómo se lleva a cabo la computación en el *nivel base*. Su tarea es solucionar problemas y devolver información sobre el propio sistema y su computación. Para ello, se tienen estructuras de datos que exponen las actividades y estructuras de las entidades que trabajan en el nivel inferior y sus acciones se reflejan en tales estructuras. Cualquier cambio en ellas modifica el comportamiento de la entidad.

Ambos niveles están conectados en una forma causal de manera que cambios en el *nivel base* se reflejan en el *nivel meta*. Idealmente, el *nivel meta* tiene acceso al *base*, pero el *base* no tiene conocimiento de la existencia del *nivel meta*. Esta característica permite a los desarrolladores aislar comportamiento de los elementos del *nivel base* de las propiedades ortogonales asignadas debido a las necesidades del ambiente computacional. Los mecanismos de *reflexión* no permiten la manipulación directa del sistema, sino a través de un modelo o representación restringida del sistema que se puede cambiar [Alv00].

A continuación se mencionan algunas propuestas que relacionan de distintos modos la *reflexión* con la *arquitectura del software* [Cue02]:

1) *Arquitectura con reflexión*. Se refiere a modelos de AS que utilizan y describen la *reflexión* de manera explícita. En este apartado se pueden considerar modelos y LDA que incluyen de manera explícita soporte para conceptos reflexivos, su descripción y manipulación:
   - En [Cue02] se describe un modelo reflexivo (*MARMOL*) para la arquitectura del software y un LDA (*PiLAR*) dinámico y reflexivo.
   - Los trabajos de Cazzola y Tisato [Tis+01] en el contexto de la reflexión arquitectónica describen tanto un modelo como un LDA.
2) *Reflexión de la arquitectura*. Se refiere al uso de técnicas reflexivas en modelos de AS, sin que éstos desarrollen su potencial descriptivo. En este apartado se incluyen modelos arquitectónicos que utilizan alguna técnica reflexiva, pero con el objetivo de resolver algún aspecto concreto como la implementación de conectores. Se pueden considerar:
   - La definición de conectores reflexivos para facilitar la implementación de una arquitectura. Ejemplos son los conectores abiertos de Asmann [AsLuPf99], los conectores instrumentados de Balzer [Bal97] o el modelo para conectores ejecutables [DuRi97].
   - La definición de meta-actores dentro del modelo de actores para aportar flexibilidad [Ast99].
   - Los modelos relacionados con *middleware* reflexivo como [Bla+00].
3) *Arquitectura de la reflexión*. Se basa en el uso de modelos de AS para la descripción de la estructura de esquemas reflexivos. La estructura de los sistemas reflexivos suele ser compleja, lo que hace más difícil su utilización. Por ello, sería interesante tener una descripción precisa de estas estructuras en términos de la AS, pero esta tarea ha sido poco estudiada.

4) *Reflexión con arquitectura*. Se refiere al tratamiento específico de la reflexión en un contexto que también considera aspectos arquitectónicos. En este bloque se incluyen aquellos trabajos que, utilizando reflexión de manera explícita, consideran también elementos relacionados con la AS, de manera que ambos conceptos se combinan.
   - En [AcNi00] se elabora el lenguaje de composición (*Piccola*) que utiliza cálculo π. *Piccola* es un lenguaje que considera la coordinación, configuración y adaptación de componentes, dando lugar a la construcción de AS. Además, su estructura interna hace uso de la reflexión a través de la definición de un *meta nivel* explícito.
   - En [Mar01] se hace un estudio sobre la construcción reflexiva de patrones de diseño; aunque no utiliza un enfoque estrictamente arquitectónico, sí se consideran conceptos análogos, pero ligados a aspectos de implementación. Los trabajos de Prior [PrBaCa00, Val+01] son continuación de los anteriores y próximos a la orientación a aspectos.

5) *Reflexión sin arquitectura*. Se hace un tratamiento específico de la reflexión, donde se manipulan conceptos análogos a los de AS. En este grupo se incluyen los trabajos que, a pesar de concentrarse en el dominio de la reflexión, dan especial importancia a aspectos estructurales o se relacionan estrechamente con campos cercanos a la AS. Todos ellos manejan conceptos relevantes para la perspectiva arquitectónica de la reflexión. Se pueden mencionar trabajos relacionados con:
   - Sistemas de programación concurrente orientada a objetos que definen un modelo subyacente reflexivo [WaYo88].
   - Desarrollos vinculados a los Sistemas Operativos reflexivos [McA96].
   - Los sistemas de *middleware* reflexivos, de gran difusión en los últimos años, en parte gracias al marco proporcionado por CORBA, que en su concepción básica es ya en parte reflexivo, aunque el primer sistema plenamente reflexivo fue OpenCorba [Led99]. Otros trabajos son los del grupo de Fabre sobre tolerancia a fallos y el trabajo de Blair [Bla+00] sobre OpenORG.
   - Los trabajos sobre persistencia ortogonal del grupo de la Universidad de St. Andrews.
   - En los últimos años se han producido diversas adaptaciones del concepto en el contexto del lenguaje Java, completando el enfoque de API de reflexión que sólo proporciona *introspección*.

Como conclusión de este apartado se puede decir que la *reflexión* se ha perfilado como un mecanismo de computación flexible, que permite añadir, evolucionar y/o modificar funcionalidades en un sistema. Aquí se han comentado someramente algunos conceptos relativos a este tema, dando una perspectiva general, incidiéndose de un modo especial en los conceptos estructurales y en los relacionados con la AS. En este sentido, habría que concluir que hay puntos de contacto entre los conceptos y modelos de *reflexión* y la *arquitectura del software*, pero, sin embargo, son pocos los trabajos que se refieren a la integración de ambos campos. Sí lo hacen los trabajos de Cuesta [Cue02] y de Álvarez [Alv00].

2.4.2.5. Arquitectura del software dinámica con reflexión

En apartados anteriores se ha comentado la relación entre la *arquitectura del software* y la *reflexión*. Sin embargo, en ningún momento se ha relacionado el concepto de *reflexión* con el de *arquitectura dinámica*; o dicho de otro modo, no se ha mostrado el dinamismo en las AS desde el punto de vista reflexivo. En la tesis doctoral de Cuesta [Cue02], cuyo título es *Arquitectura de software dinámica basada en reflexión,* se plantea y se discute que

cualquier tipo de dinamismo arquitectónico se puede expresar mediante reflexión.

El trabajo se propone

... justificar que cualquiera de los modelos de dinamismo arquitectónico (como los descritos en el apartado 2.3.3) *en los que se clasifican estas arquitecturas se puede expresar como "una relación reflexiva" entre una o varias meta entidades y los elementos de la arquitectura…*

Y se comprueba que

todos estos modelos pueden describirse sin cambios en términos de reflexión.

Además, aunque no mediante una demostración formal, sí queda suficientemente argumentado que

la reflexión es suficiente para expresar el dinamismo en cualquiera de sus formas.

Finalmente una conclusión interesante a la que llega el trabajo es que

introducir reflexión en la arquitectura facilita la extensibilidad del sistema, de modo que, en un modelo arquitectónico reflexivo, es más sencillo introducir seguridad, múltiples vistas, e incluso la descripción de sistemas orientados a aspectos.

Por último, y después de mostrar las características y conceptos de AS dinámica y de reflexión, se puede decir que una se puede explicar en función de la otra, como propone Cuesta, que prueba que los modelos de descripción de arquitecturas dinámicas se pueden ver desde un punto de vista reflexivo.

2.5. Desarrollo software basado en componentes

El paradigma del *Desarrollo Software Basado en Componentes* (DSBD) o *Component Based Software Development* (*CBSD*) en su denominación en inglés, supone adoptar un enfoque diferente al tradicional, ya que su objetivo es evitar abordar el desarrollo de los sistemas desde cero, sino hacerlo mediante la aplicación de técnicas de reutilización de partes del software ya desarrolladas y probadas. Estos elementos reutilizables son los llamados *componentes*.

A partir de una discusión inicial sobre el término componente, qué es y qué no es, se ha llegado a una definición comúnmente aceptada, debida a Szypersky [Szy98] quien afirma que:

> *Un componente software es una unidad de composición binaria que especifica un conjunto de interfaces que el componente ofrece al entorno y un conjunto de dependencias que el componente tiene del entorno*.

Además, un *componente software* en general se desarrolla de forma independiente, para ser compuesto posteriormente con otros componentes.

Hoy se considera que un componente es una unidad de composición reutilizable. Su reutilización en diferentes contextos hace necesario que se tenga que prestar especial atención a la descripción de sus interfaces. Esto es así porque mediante ellas el componente se comunica con los otros con los que se relaciona, proporcionando servicios al resto y recibiendo del entorno los servicios que ofrecen otros componentes. Los lenguajes de definición de interfaces deberían proporcionar información que permita manejar los componentes como cajas negras que encapsulan una cierta funcionalidad a la que no sería necesario acceder.

Bajo este paradigma, un sistema se define como un conjunto de componentes que deben conectarse adecuadamente. La fase que permite definir la *arquitectura del sistema* final, estableciendo las conexiones entre los componentes que la forman, se denomina *fase de ensamblado*, de especial importancia en este tipo de desarrollos. Sin embargo, para definir la *arquitectura de un sistema* no basta con definir las interfaces de los componentes, sino que es necesario que todos se hayan desarrollado siguiendo ciertas convenciones relativas a cómo debe componerse cada uno. Es decir, se deben haber construido siguiendo el mismo modelo de componentes.

Considerando que una de las razones para aplicar el DSBC es hacer más flexible la estructura del sistema final al aplicar un desarrollo basado en la composición de elementos, el último objetivo de un componente no es ejecutarse de forma aislada, sino conectarse a otros componentes en un cierto contexto. Así, el *diseño de la arquitectura* es una fase esencial en el DSBC, al entenderse que los componentes son una parte intrínseca de la arquitectura de la que forman parte.

En lo que se refiere al trabajo que aquí se presenta, el modelo que se propone trabaja en torno a la definición de un modelo basado en componentes, sin que estos tengan que estar definidos bajo ningún modelo de componentes concreto.

2.6. Desarrollo software dirigido por modelos

En los últimos años ha surgido una aproximación al desarrollo software que pretende facilitar la evolución de los sistemas software: el *Desarrollo Software Dirigido por Modelos* (*DSDM*), o en su denominación inglesa *Model-Driven Software Development* (*MDSD*), que trata de mejorar la adaptabilidad de los sistemas ante los cambios, no sólo de los requisitos, sino también los tecnológicos mediante la definición

de modelos a diferente nivel. Es una aproximación basada en el modelado de sistemas software, la transformación de modelos y la generación final del sistema a partir de ellos. Sin embargo, al ser sólo una aproximación, no define técnicas a utilizar ni fases del proceso ni ningún tipo de guía metodológica, sólo proporciona una estrategia general a seguir en el desarrollo del software. En este paradigma los modelos se consideran elementos de primer orden: un modelo se crea con un propósito específico y contiene sólo la información necesaria para llevar a cabo dicho propósito. Así, distintos modelos contemplan el desarrollo y manipulación de diversos aspectos del sistema en estudio. La transformación empieza en etapas tempranas y el ingeniero de software completa cada modelo obtenido de un modo más o menos manual. El proceso de transformación de un modelo en otro juega un papel fundamental a lo largo del desarrollo de un sistema bajo este paradigma. El *DSDM* soporta también distintos niveles de abstracción, relacionados en mayor o menor medida con modelos dependientes de la plataforma de implementación. La gran ventaja de la utilización de esta aproximación es que los modelos no sólo son documentos, sino que realmente son los elementos centrales del desarrollo que finalmente se transforman en el producto final.

Prueba del interés que está despertando el *DSDM* en la comunidad investigadora es el auge de la aproximación *Model-Driven Architecture*:

Model Driven Architecture (*MDA*) [OMG-MDA], propuesta del OMG (Object Management Group [OMG]), es un marco de trabajo para el desarrollo software dirigido por modelos. *MDA* se basa en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad; le da mayor importancia al modelado conceptual y al papel de los modelos como elementos de primer orden en el diseño, desarrollo e implementación del software y a la definición de las transformaciones entre ellos. En función del nivel de abstracción, *MDA* considera diferentes tipos de modelos:

- Los requisitos del sistema son recogidos en los *Modelos Independientes de Computación* (conocido como *CIM*, *Computation Independent Models*). En este nivel se modela la lógica de negocio o la especificación de los requisitos de un modo independiente de la computación.
- Los *Modelos Independientes de la Plataforma* (PIM, *Platform Independent Models*) representan la funcionalidad del sistema abstrayéndose de la plataforma final. En este nivel se modela el sistema independientemente de cualquier plataforma software/hardware. El modelo presentado en el *CIM* es refinado en este nivel; para ello, se transforma un modelo independiente de la computación en otro independiente de plataforma. El nivel *PIM* permite centrarse en la especificación de la funcionalidad del sistema, olvidándose de todo detalle relativo a la plataforma en la que será implementado.
- Los *Modelos Específicos de la Plataforma* (PSM, *Platform Specific Models*) combinan las especificaciones contenidas en un *PIM* con los detalles de la plataforma elegida. A partir de los distintos *PSM* se pueden generar automáticamente distintas implementaciones del mismo sistema. En este nivel se introducen los detalles específicos de una plataforma en los modelos que han sido especificados en niveles anteriores. *PSM* es un refinamiento del nivel *PIM* anterior.

La definición de los niveles supone una mejora en el desarrollo de sistemas complejos; sin embargo, a veces, los modelos y las transformaciones entre ellos se vuelven excesivamente grandes.

En lo referente al ámbito de esta tesis doctoral, la definición arquitectónica de un sistema, desde el punto de vista del *DSDM* o, más concretamente, en el de *MDA*, corresponde a la definición de modelos a alto nivel e independientes de la plataforma, por lo que se estaría hablando del nivel de abstracción definido como *Modelo Independiente de la Plataforma* (*PIM)*: tras realizarse la especificación del sistema a partir de los requisitos y después del proceso de análisis se define el o los modelos independientes de la plataforma que correspondan, para, en transformaciones sucesivas, ir bajando el nivel de abstracción.

2.7. Norma IEEE 1471

Esta es la *Norma* que el IEEE *Computer Society* [IEEE] propuso en 2000 como estándar para desarrollar sistemas software en términos de *descripciones arquitectónicas* (*DA*). En ella se establece un marco de trabajo o *framework conceptual* para la descripción arquitectónica. Igualmente se define el contenido y las características de esas *descripciones arquitectónicas*.

Como se sabe, la AS tiene una gran influencia en el desarrollo de un sistema. Sin embargo, en general, los conceptos de arquitectura no se han aplicado adecuadamente. Además, hasta la definición de la *Norma* no había un marco de trabajo comúnmente aceptado para configurar el pensamiento arquitectónico que facilitara la aplicación y evolución de las prácticas arquitectónicas. En la actualidad se empiezan a aplicar conceptos arquitectónicos al desarrollo de los sistemas para lograr beneficios como reducción de costes e incremento de la calidad de los sistemas. Así pues, el objetivo genérico del *estándar 1471* es facilitar la comunicación de las arquitecturas y sentar las bases para desarrollos de calidad y coste reducido, a través de la estandarización de elementos de la descripción arquitectónica. Para ello, la *Norma* contiene recomendaciones sobre cómo realizar la descripción arquitectónica de un sistema durante su desarrollo. Sin embargo, no se trata de la propuesta de una arquitectura estándar, un proceso arquitectónico o un método, sino de un conjunto de recomendaciones prácticas para la definición de arquitecturas.

La *Norma* incluye las *descripciones arquitectónicas* que se usan en: la descripción del sistema y su evolución, la comunicación entre usuarios y desarrolladores, la evaluación y comparación de arquitecturas, las actividades de planificación y gestión del desarrollo, la expresión de características persistentes y guía para el cambio, y la verificación de la implementación relacionada con la descripción arquitectónica.

Actualmente existe un consenso general sobre la importancia del nivel arquitectónico en el desarrollo de los sistemas, ya que este nivel consta de las decisiones tempranas tomadas sobre el conjunto del diseño, los objetivos, los requisitos y las estrategias de desarrollo. Sin embargo, no hay una definición precisa de cómo se debería

describir la arquitectura de un sistema para que esa descripción sirva dónde y cuándo se debe realizar.

Por estas razones, la *Norma* de IEEE trata de reflejar tendencias generalmente aceptadas para la descripción arquitectónica y proporcionar un marco o *framework* técnico para la evolución en este área. Además, se establece un marco o *framework* conceptual con conceptos y términos de referencia. La *Norma* se centra en aquellos elementos sobre los que hay consenso, especialmente en el uso de múltiples vistas, especificaciones reutilizables para modelos en las vistas y la relación de la arquitectura con el contexto del sistema.

2.7.1. Definiciones

El estándar *IEEE 1471* propone varias definiciones, entre las que destacamos las siguientes:

- *Descripción arquitectónica* (*Architectural Description)- DA*: conjunto de productos que permiten documentar la arquitectura. No detalla su formato sino que declara ciertos contenidos mínimos que deben especificarla.
- *Arquitectura*: organización fundamental de un sistema compuesto por componentes, sus relaciones y el entorno, así como los principios que guían su diseño y evolución.
- *Vista* (*View*): representación de un sistema desde la perspectiva de un conjunto de aspectos relacionados. Una *descripción arquitectónica* puede contener una o más *vistas;* una *vista* puede contener uno o más *modelos arquitectónicos*, permitiendo así la utilización de múltiples notaciones para representarla. Por otra parte, las *descripciones arquitectónicas* permiten detectar inconsistencias entre las *vistas* que contiene.
- *Punto de vista* (*Viewpoint*): es una especificación para construir y usar una *vista*. Una *vista* está bien definida si corresponde exactamente a un *punto de vista*. Un *punto de vista* es un patrón para construir vistas; los *puntos de vista* definen reglas sobre las *vistas*.

2.7.2. Marco conceptual

Este marco de trabajo (Figura 2.4) se define para establecer los términos y conceptos relativos al contenido y uso de las *descripciones arquitectónicas*:

- *Sistema*: este término se refiere a todas aquellas aplicaciones, sistemas, subsistemas, líneas de producto, familias de productos o cualquier otra entidad de interés. Todo *sistema* existe en un *entorno* que influye en él.
- El *entorno* o contexto determina el conjunto de circunstancias de todo tipo que influyen en el *sistema*. El *entorno* se puede referir también a otros sistemas que interactúen con él y determina sus límites.
- Los *concerns* (o *competencias*) son aspectos o temas de interés relacionados con el *sistema* en desarrollo, sus operaciones o cualquier otro aspecto interesante para algún *actor* del sistema. El concepto *concern* incluye consideraciones del sistema tales

como *ejecutabilidad*, fiabilidad, seguridad, distribución, o capacidad de evolución. Cada *concern* está dirigido por una *vista* arquitectónica

- *Stakeholders*[4] (*Actor*): Un *sistema* tiene uno o más *stakeholders.* Cada uno tiene interés en el *sistema* desde algún *punto de vista*.
- Una *misión* es un uso o una operación en la que está interesado al menos un *actor* del sistema para lograr algún objetivo. Un *sistema* existe para llevar a cabo una o más *misiones* de su entorno.
- Por otra parte, todo *sistema* tiene una *arquitectura,* y una *arquitectura* se gestiona por una *descripción arquitectónica*. La *Norma IEEE 1471* distingue la arquitectura del sistema (que es conceptual) de las descripciones particulares de esa *arquitectura* (que son productos o artefactos software concretos). Así, las *descripciones arquitectónicas* son el objeto de esta *Norma*.

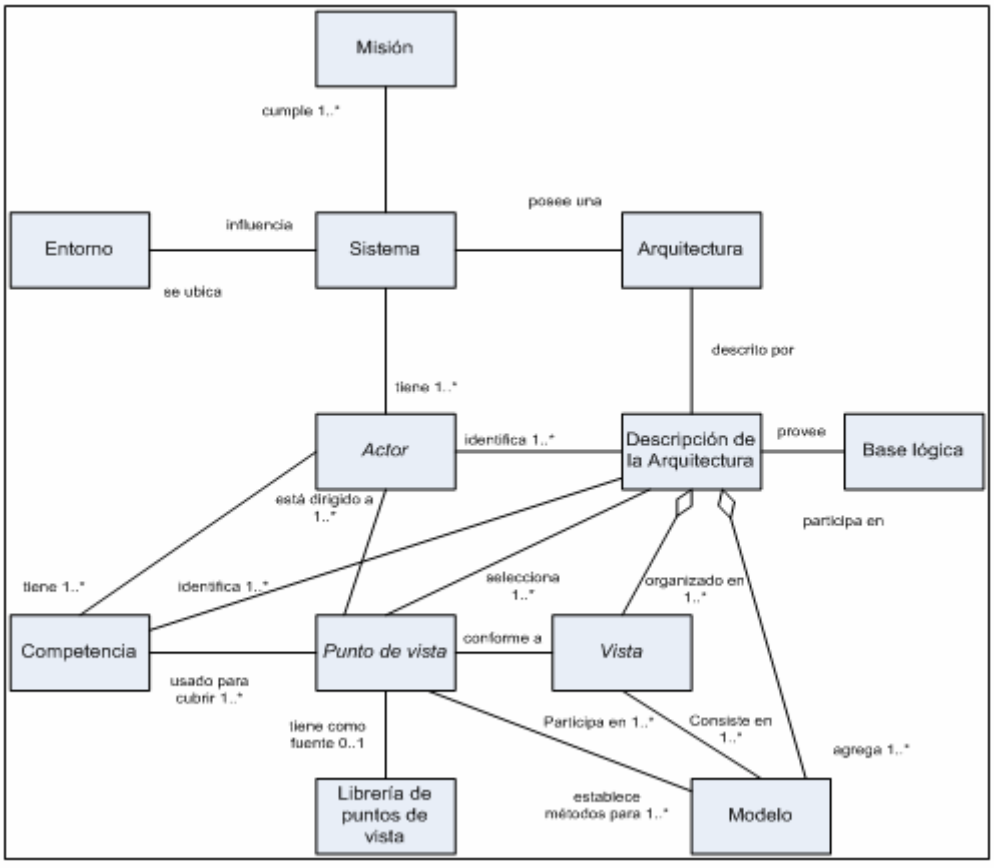


Figura 2.4. Marco conceptual definido por la *Norma* IEEE.

[4] *Stakeholder*: es una persona relacionada de algún modo con el sistema en desarrollo o a desarrollar, tanto usuarios (en sus distintas categorías: clientes, propietario, usuario del sistema final….) como miembros del equipo de desarrollo a lo largo de todas las fases (ingeniero de sistemas, arquitecto, diseñador, *mantenedor*, proveedor…). Por ello se suele usar la palabra *actor* como traducción de la misma. Aquí se usarán ambas indistintamente.

- Una *descripción arquitectónica* se divide en una o más *vistas*. Cada *vista* se refiere a uno o más *concerns*. Una *vista* se usa para referirse a la expresión de una *arquitectura* de un sistema con respecto a un *punto de vista* particular.
- Un *punto de vista* establece las convenciones por las cuales se crea una *vista*, se representa y se analiza; y una *vista* está formada por varios *puntos de vista*. El *punto de vista* determina el lenguaje que se usa para describir la *vista*, así como cualquier otro método de modelado o técnica de análisis que se use para representarla. Una *descripción arquitectónica* usa uno o más *puntos de vista*. La selección de un *punto de vista* se basa en consideraciones de los actores o *stakeholders* a los que van dirigidas las *descripciones arquitectónicas* y en sus *concerns*. Un *punto de vista* se puede definir en relación a una *descripción arquitectónica* o haberse definido en otro lugar (se obtendría entonces de una biblioteca de *puntos de vista*).
- Por otra parte, una *vista* puede estar formada por uno o más *modelos arquitectónicos*. Cada *modelo* se desarrolla usando los métodos establecidos por sus *puntos de vista* asociados. Un *modelo arquitectónico* puede formar parte de una o más *vistas*.

2.7.3. Arquitectura y evolución

La *Norma IEEE 1471* también hace algunas consideraciones sobre el interés que tienen que tener en cuenta las *DA* a lo largo del desarrollo y en particular durante la evolución.

Arquitectura en el ciclo de vida

Ya se ha comentado en apartados anteriores el interés de tener en cuenta las consideraciones arquitectónicas desde el principio del desarrollo y durante todo el ciclo de vida del sistema. El estándar no presupone el uso de un modelo de desarrollo concreto, sino que hace sugerencias para aplicar durante el mismo.

Arquitectura para sistemas que evolucionan

Un sistema y su arquitectura se definen y desarrollan en función de las necesidades y los requisitos de los usuarios. Por ello, la arquitectura de los sistemas tiene que evolucionar en respuesta a esas nuevas necesidades. En este sentido, las *descripciones arquitectónicas* se usan para guiar cada sistema a lo largo del proceso de desarrollo, y las sucesivas versiones de las *DA* permiten evaluar cada entrega del sistema. Las arquitecturas desarrolladas bajo este escenario son más flexibles y adaptables que los sistemas definidos sin considerar las posibilidades de evolución. Las *DA* para la evolución de los sistemas se desarrollan siguiendo un patrón iterativo de cambios de versiones. Las necesidades de los diversos actores o *stakeholders* se van refinando en cada iteración de la arquitectura.

Si el desarrollo de los sistemas se ha hecho sin considerar las *descripciones arquitectónicas* o bien cuando no se dispone de ellas, el arquitecto puede llegar a obtenerlas usando técnicas de reingeniería. De este modo, se reconstruye la *descripción*

arquitectónica del sistema que se usará para desarrollar un nuevo sistema, para guiar las actividades de evolución o para establecer una arquitectura evolutiva.

Evaluación arquitectónica

El propósito de la evaluación es determinar la calidad de las *DA* o predecir la calidad del sistema. La calidad en las *DA* se refiere a su capacidad para descubrir las necesidades de los usuarios para los que se ha construido el sistema. También se pueden determinar temas de interés (*concern*) para los usuarios como comprensibilidad, consistencia, completitud, eficiencia o capacidad de análisis de las *DA*.

2.7.4. Usos de las descripciones arquitectónicas

Las *descripciones arquitectónicas* son aplicables a muchos usos a lo largo del desarrollo. Entre otros podemos citar: el análisis de arquitecturas alternativas, la planificación de la transición de una arquitectura heredada a una nueva, la comunicación entre *stakeholders* o la gestión de la documentación tanto para desarrollar actividades posteriores como durante la evolución del sistema. Para llegar a definir la arquitectura de un sistema en función de las *DA*, hay que determinar los siguientes elementos:

- Identificar las *descripciones arquitectónicas*.
- Identificar los *actores* del sistema y los *concerns* relevantes para la arquitectura.
- Especificar cada *punto de vista* que se haya seleccionado para organizar la representación de la arquitectura.
- Obtener las *vistas*.
- Determinar las posibles inconsistencias.

2.8. Resumen y conclusiones del capítulo

En este capítulo se han revisado los conceptos en los que se fundamenta el modelo arquitectónico propuesto en esta tesis:

- Se han comentado las características, definiciones y elementos fundamentales de la arquitectura del software, fase del desarrollo de un sistema en la que se enmarca el estudio realizado.
- Se han revisado los lenguajes de descripción arquitectónica que se han considerado más interesantes y se han descrito aquellos que tienen mayor relación con el propuesto en esta tesis.
- Igualmente, se ha hecho una somera revisión de los conceptos de dinamismo, dinamismo en la arquitectura del software, reflexión, reflexión arquitectónica, DSBC y DSDM que también se relacionan con el tema de esta tesis.
- Finalmente, el apartado dedicado a la *Norma* IEEE 1471 revisa la propuesta de dicho organismo para la estandarización de la definición de la arquitectura de los sistemas.

El estudio realizado a lo largo de este capítulo ha mostrado el interés de la AS y su importancia en el desarrollo de los sistemas software. En este sentido, se han propuesto mecanismos para describir formalmente las arquitecturas, en concreto, mediante la definición de los diferentes LDA; se ha mostrado la importancia de la AS durante la evolución y el interés de considerar las AS dinámicas en el desarrollo de sistemas complejos. Sin embargo, no se aborda el problema de modelar adecuadamente las propiedades transversales no modularizables que presentan muchos sistemas.

El trabajo realizado durante el desarrollo de esta tesis doctoral prevé una solución a la cuestión relativa a las propiedades transversales, consideradas desde un punto de vista arquitectónico. Para ello, se propone un modelo, una metodología y un LDA que permiten desarrollar sistemas orientados a aspectos, desde la perspectiva de la arquitectura del software. Sin embargo, para abordar esta tarea con suficientes garantías, fue necesario realizar antes un estudio en profundidad del paradigma del Desarrollo de Sistemas Orientado a Aspectos, razón por la cual se incluye en esta memoria el siguiente Capítulo.

CAPÍTULO 3
Desarrollo software orientado a aspectos

Como se concluyó en el capítulo anterior, para cumplir los objetivos de esta tesis, era necesario estudiar en profundidad las propiedades transversales no modularizables que se encuentran presentes en muchos sistemas software y en su desarrollo. Por ello, se abordó el estudio del paradigma del desarrollo de sistemas orientado a aspectos. En este capítulo se introducen los conceptos de aspecto, separación de aspectos y desarrollo orientado a aspectos. Se empieza con una breve reseña histórica y las definiciones de los conceptos más importantes. Igualmente se estudian los distintos modelos desarrollados en los últimos años, mostrándose agrupados según la fase de desarrollo en la que se centran, prestando especial atención a la fase del diseño arquitectónico. Finalmente se comentan varios de los Lenguajes de Descripción Arquitectónica Orientados a Aspectos de los que se exponen sus ventajas e inconvenientes.

3.1. Introducción al paradigma

Esta sección describe primeramente, de modo general, las características del *Desarrollo de Sistemas Orientado a Aspectos* (DSOA); en apartados subsiguientes, por una parte se hace una breve reseña histórica del tema; y por otra, se describen detalladamente los diferentes conceptos relacionados con esta aproximación, y que es necesario conocer para comprender el resto del capítulo.

El DSOA es un paradigma en auge que se está convirtiendo en una alternativa para mejorar el desarrollo de sistemas complejos. Primero, surgió la Programación Orientada Aspectos (OA), centrada en la fase de implementación y, posteriormente, se planteó el desarrollo de sistemas OA, que propone la utilización del concepto de aspecto (apartado 3.1.2) a lo largo de todo el ciclo de vida. Así, los aspectos identificados en la ingeniería de requisitos o la especificación y el análisis se utilizan en el diseño o la implementación.

Esta aproximación proporciona técnicas para la identificación y separación de los aspectos, así como su composición posterior. Igualmente permite identificar los que son *concerns*[5] aspectuales y los que no lo son, concretamente durante las fases tempranas del desarrollo. Además, permiten mantener la trazabilidad de tales *concerns* (materias de interés) en etapas posteriores. De este modo, se consigue una mejora en la modularización de los sistemas, obteniéndose un código menos enmarañado y evitándose la mezcla entre funcionalidad del sistema y los aspectos extra-funcionales, lo que, además, facilita el mantenimiento y la evolución. Para ello, las técnicas para desarrollar sistemas OA amplían las técnicas tradicionales, permitiendo la encapsulación de los aspectos en módulos independientes. Es decir, encapsulan las propiedades que atraviesan varios componentes de un sistema.

Por otra parte, en el capítulo 2 se ha puesto de manifiesto que

> *la arquitectura del software es una disciplina que tiene en cuenta las decisiones de diseño en etapas tempranas del proceso de desarrollo software, el efecto de estas decisiones sobre el análisis de requisitos y cómo se propaga a las demás fases del ciclo de vida. Además, la AS es una disciplina que facilita el estudio de la modularización, la composición en un sistema y sus consecuencias* [Cue+05].

Esta definición de AS es una representación del principio de la Separación de Intereses (*Separation of Concerns*), que establece que cada interés (*concern*) se debe expresar como un módulo. Después de definir los módulos, aparecen las estructuras que los combinan, y esto debe ser estudiado junto con sus consecuencias.

Recientemente varios autores han publicado resúmenes y diversas consideraciones sobre la identificación de los aspectos a lo largo del ciclo de vida; entre ellos cabe destacar los trabajos llevados a cabo en el marco del proyecto AOSD-Europe [AOSDEur], en particular el estudio realizado en [Chi+05]. En [Cue+05] se incluye la

[5] *Conern* se define en [Fil+05] como *algo en lo que se tiene interés a lo largo del desarrollo de un sistema*.

enumeración de diferentes aproximaciones y modelos desarrollados en el marco de la Arquitectura del Software Orientada a Aspectos. Estos documentos, entre otros, se han tenido en cuenta para redactar este capítulo.

3.1.1. Reseña histórica

Los conceptos de orientación a aspectos nacieron en 1996, cuando Kiczales [Kic+96] propuso una nueva idea: la *Programación Orientada a Aspectos* (POA) o *Aspect Oriented Programming* en su denominación inglesa. A partir de entonces, la POA y otras técnicas y tecnologías, que se centraban en la modularización del código que atraviesa varios componentes para realizar una cierta función (*crosscutting code*), se agruparon bajo el nombre de *Advanced Separation of Concerns*[6]. Después de varios años, en 2002, se adoptó la denominación actual *Desarrollo de Sistemas Orientado a Aspectos* para referirse a las técnicas y tecnologías que pretenden la modularización de los *crosscutting concerns*[7] a lo largo del ciclo de vida. Desde entonces, los investigadores que trabajaban en torno a la orientación a aspectos comenzaron a estudiar las relaciones entre aspectos así como las propiedades de la composición aspectual.

Originalmente, la identificación, especificación e implementación de los aspectos se realizaba en la fase de implementación, viéndose los programadores en la necesidad de ser ellos los encargados de rediseñar el sistema para evitar el código entrelazado (*crosscutting code*). En la actualidad, la responsabilidad de identificación y especificación de aspectos se ha desplazado a fases iniciales del ciclo de vida: durante la especificación de requisitos y la arquitectura del software, dejando a los programadores la responsabilidad exclusiva de su implementación.

El enfoque OA ha demostrado ser una tecnología potente para manejar la separación de propiedades. Esta separación permite la construcción de un software más modularizado, facilitando la reutilización, el mantenimiento y la evolución de los sistemas. El problema principal para la adopción de estas técnicas en proyectos a gran escala es la falta de un proceso de desarrollo claro. Por otra parte, una de las principales ventajas de las propuestas de DSOA es que permiten a los desarrolladores reaccionar fácilmente ante los cambios no previstos.

3.1.2. Conceptos de separación de aspectos

Este apartado se ha denominado *separación de aspectos*, si bien proviene de la expresión inglesa *separation of concerns* que también se ha traducido como separación de

[6] Se ha mantenido la expresión inglesa por ser de uso común y posiblemente más adecuada que su traducción como *Separación avanzada de intereses o de propiedades*.

[7] Como en el caso anterior, se ha mantenido la expresión inglesa por ser de uso común y, posiblemente, más adecuada que su traducción como *intereses o propiedades transversales*.

intereses (*concern*: algo de interés, asunto) o separación de propiedades. Por eso, antes de definir un aspecto (*aspect*), conviene definir el concepto de *concern*[8]*:*

- Según [Fil+05] un *concern* es

  Algo en lo que se tiene interés a lo largo del desarrollo de un sistema.

  Se puede decir, pues, que un *concern* es un tema de interés, una parte del problema a resolver, o una característica o propiedad que se debe considerar. Este concepto se utiliza para referirse tanto a propiedades funcionales como a las no funcionales de un sistema. Ejemplo de *concerns* que expresan estas propiedades son: seguridad, tolerancia a fallos o sincronización.

  Si se siguen aproximaciones tradicionales, el comportamiento correspondiente a un *concern* queda entrelazado (*tangled*) con el de otros *concerns* del sistema. Por otra parte, para llegar a expresar sus objetivos, este comportamiento queda o puede quedar disperso (*scattered*) a lo largo del mismo.

- No existe una definición consensuada de *aspecto* que sirva a lo largo de las diferentes fases del desarrollo pues existen *aspectos* a nivel de ingeniería de requisitos y diseño arquitectónico, a nivel de diseño o de implementación. En general, un *aspecto* puede considerarse como

  un elemento con comportamiento extra que se ejecuta en un momento dado durante de la ejecución de una aplicación.

  Un *aspecto* según [Fil+05] es

  Una unidad modular diseñada para encapsular el comportamiento de un concern específico, siendo la naturaleza de esta descomposición diferente de otras descomposiciones usadas anteriormente en aproximaciones convencionales.

  También se ha definido un *aspecto* como

  una unidad para modularizar una propiedad transversal en el sistema (crosscutting concern).

- Relacionado con el principio de la separación de intereses (*separation of concern*s) está el problema de los *intereses o propiedades transversales* (*crosscutting concerns*). Una *propiedad transversal* es [Fil+05]

  Aquella cuya implementación queda dispersa (scattered) a lo largo de varios elementos del sistema.

  Además, esta implementación dispersa supone o puede suponer la obtención de un código enmarañado (*tangled*). Por eso, el término *propiedades transversales* (*crosscutting concern)* se suele describir en términos de dispersión (*scattering*) y enmarañamiento (*tangling*). Es decir, el código transversal se refiere a la dispersión y el enmarañamiento del código de los intereses o propiedades del sistema que surge debido a una pobre modularización del mismo. En [BeCoHe06] y [BeCoHe07] se discuten ampliamente estos tres conceptos (código transversal, enmarañamiento y

[8] En estos apartados se mantienen algunos términos en inglés por considerarse éstos más adecuados que la expresión en castellano, sobre los que, en algunos casos, no hay consenso.

dispersión) y su interrelación. En trabajos posteriores de estos autores ([Con+07] y [CoHe08]) se establece de un modo formal la relación entre ellos.

Los *crosscutting concerns* han sido considerados como una de las principales características responsables de la complejidad del software. El término se refiere a propiedades del software que no pueden ser efectivamente modularizadas usando técnicas tradicionales, como las orientadas a objetos (por *crosscutting* se puede entender que rompe las barreras de encapsulación).

Finalmente, el propósito de la definición de *aspectos* es llegar a modularizar el comportamiento de una propiedad específica (*concern*), evitando los problemas de dispersión (*scattering*) y enmarañamiento (*tangling*).

Para cada fase de desarrollo hay diferentes propuestas para realizar la separación de propiedades que resultan más o menos adecuadas en función de las características del sistema a construir. Las propuestas varían en función de cómo y cuándo se identifican los aspectos; algunas siguen el modelo simétrico y otras el asimétrico (apartado 3.1.3); unas definen los aspectos como componentes o conectores; y otras definen conceptos nuevos como vistas (*view*) asociados a los aspectos o los *concerns*. Además, diferentes propuestas varían en el modo en que se realiza la composición de los aspectos: se puede hacer de forma estática -en tiempo de compilación-, de forma dinámica -en tiempo de carga- o en tiempo de ejecución. Por otra parte, los *puntos de intercepción* pueden hacer referencia a elementos internos de clases y componentes (modelos invasivos) o sólo al comportamiento externo (modelos no invasivos), siendo un *punto de intercepción* o *punto de enlace* (*join point*) un punto en la ejecución de un módulo del sistema en el que se puede insertar un cierto *concern*. En este caso, el hilo de ejecución es interceptado para que se ejecute un módulo aspectual (*aspecto*), si se dan ciertas condiciones.

3.1.2.1. Aspectos tempranos o early aspects

Este concepto empezó a usarse a partir del primer *workshop* sobre *Early Aspects* organizado dentro de la primera conferencia sobre Desarrollo de Sistemas Orientados a Aspecto en 2002 [Early02], aunque antes hubo algunos trabajos en este sentido, pero sin usar tal denominación. Por ejemplo en [Men+98] se habla de identificar *intereses (concerns) durante el análisis de requisitos y el análisis del dominio*. Desde entonces, se vienen organizando anualmente *workshops* en diversas conferencias como OOPSLA, SPLC, ICSE o AOSD, con una creciente participación en el número de investigadores y de los trabajos presentados.

Según esto, en cada fase del ciclo de vida habría que identificar los diferentes intereses o propiedades (*concerns*) del sistema. Estos *concerns* se convertirán en módulos en fases posteriores. Sin embargo, pueden aparecer *concerns* en estas etapas que no se puedan incluir en un único módulo, sino que queden dispersos o distribuidos entre varios. Se denominan *Aspectos Tempranos* o *Early Aspects* a los aspectos (*concerns*) identificados en las primeras etapas del ciclo de vida para distinguirlos de los que se identifican en fases posteriores. El término se refiere, pues, a las propiedades que atraviesan el sistema (*crosscutting concerns*) cuando éste se estudia a lo largo de la especificación de requisitos o el diseño arquitectónico. Por tanto, los *aspectos tempranos* tienden a estar dispersos (*scattered*) entre varios módulos, que en fases iniciales pueden

ser requisitos, casos de uso, componentes arquitectónicos o diagramas de interacción; esto conllevaría una mala modularización de tales propiedades, que se podría propagar a otros artefactos software durante proceso de desarrollo del sistema. Además, estas propiedades afectan (o pueden afectar) de algún modo a otros requisitos o componentes arquitectónicos del sistema, de modo que si no se identificaran, no se podría conocer el efecto que tienen (o pueden llegar a tener) sobre el sistema. En este grupo se pueden considerar incluidas propiedades como la seguridad, movilidad, disponibilidad o restricciones de tiempo real [Early]. La identificación temprana de aspectos mejora la calidad del producto final y reduce los costos de adaptación, evolución y mantenimiento. Figura 3.1 tomada de [Early] representa la conversión de los diversos intereses o propiedades (*concerns*) en módulos a lo largo de las distintas fases.

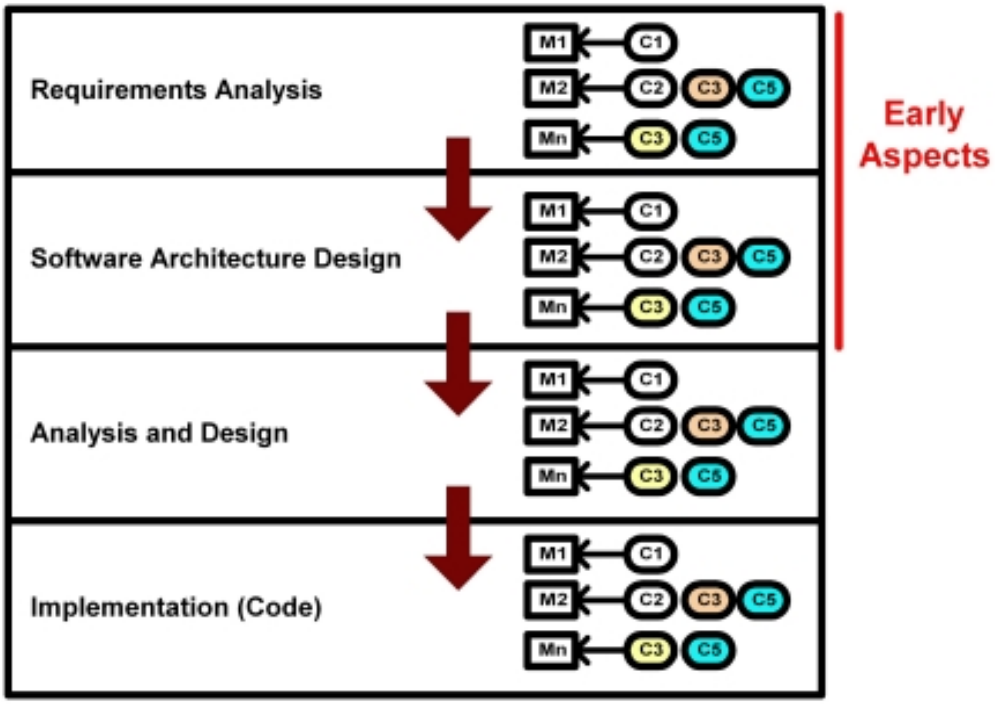


Figura 3.1. Transformación de propiedades o intereses (*concerns*) en módulos a lo largo del ciclo de vida e identificación de *aspectos tempranos.*

La identificación y gestión de los *aspectos tempranos* a lo largo de las fases iniciales [Early]:

- Incrementa la consistencia entre los requisitos y la arquitectura, con la implementación.
- Asegura que los *crosscutting concerns* de las fases iniciales se identifican como aspectos y que se propagarán adecuadamente a lo largo de la arquitectura y el diseño sobre la implementación.
- Proporciona técnicas y métodos para razonar y gestionar la influencia de esos aspectos.

- Proporciona técnicas para ayudar a los desarrolladores a identificar aspectos.
- Facilita la trazabilidad[9] de los aspectos a lo largo del ciclo de vida.

3.1.2.2. Aspectos arquitectónicos

La arquitectura del software es una de las fases del ciclo de vida del desarrollo de software en la que se toman decisiones tempranas de diseño, incluyendo la identificación y especificación de los componentes de alto nivel, que influirán en el diseño y la implementación posteriores. Por ello, es importante que el diseño arquitectónico cumpla con los criterios de calidad requeridos por los usuarios para los sistemas a desarrollar. Para asegurar que se cumplen estos criterios de calidad, es importante identificar los aspectos (*concerns*) relevantes a nivel arquitectónico.

Sin embargo, algunos de los *concerns* que se identifican durante el diseño arquitectónico no se localizan sobre un único componente arquitectónico, según se refleja en Figura 3.1. Estos *concerns* se denominan *aspectos arquitectónicos*[10]. De este modo, es necesario aplicar abstracciones específicas de orientación a aspectos para identificar, especificar y evaluar los aspectos a nivel de arquitectura del software. No identificar los *concerns* que atraviesan los elementos arquitectónicos puede suponer que posteriormente, a nivel de implementación, se genere código disperso. Esto supone que los factores de calidad del sistema en desarrollo pueden no cumplirse. Sin embargo, la representación de los aspectos arquitectónicos no es una tarea sencilla porque suele ser necesario utilizar mecanismos de representación explícita de algunas propiedades tales como la gestión de errores o los protocolos de seguridad. Estos *concerns* arquitectónicos transversales pueden interactuar con los módulos afectados de diferentes formas; además, los aspectos pueden interactuar entre ellos en puntos determinados. Por tanto, se debe proporcionar a los arquitectos de software medios para llevar a cabo una representación modular de los elementos aspectuales.

> Mens en [Men02] considera los *aspectos arquitectónicos* como
>
> *aquellos elementos arquitectónicos que cruzan el diseño. Además, los aspectos arquitectónicos definen los puntos de corte (pointcuts) en los que se debe incorporar el código adicional*.
>
> En [Gar+06] se define un *aspecto arquitectónico* como
>
> *un concern que atraviesa las unidades modulares arquitectónicas y no pueden modularizarse de un modo efectivo usando modelos y Lenguajes de Descripción Arquitectónica tradicionales*.

[9] La *trazabilidad* se puede definir como la relación que se puede establecer entre dos o más artefactos del proceso de desarrollo. La *trazabilidad* facilita la comprensión del sistema relacionando un artefacto software con una representación en una fase y la que se utiliza en la siguiente, junto con la información que permite razonar sobre las decisiones tomadas.

[10] El término *aspecto arquitectónico* (*architectural aspect*) fue definido de un modo formal por Tekinerdogan en [Tek04].

En general, los LDA permiten expresar las propiedades (*concerns*) que se puedan modularizar como elementos arquitectónicos. Si no se dota de elementos que permitan evitarlo, la descripción arquitectónica de las propiedades transversales (*crosscutting concerns*) en un LDA puede dar lugar a dispersión y entremezclado de los *concerns*. Para soportar esta modularización se han propuesto algunos LDA orientados a aspectos (sección 3.4). En ellos varía el modo en el que los elementos del lenguaje pueden verse afectados por los *crosscutting concerns.*

3.1.3. Modelos simétricos y asimétricos

El concepto de aspecto surge para tratar los intereses transversales (*crosscutting concerns*) de un sistema. La técnica de descomposición aspectual introduce un tipo de módulo especial que permite representarlos de modo que un *aspecto* resulta definido finalmente como un módulo. Esto supone que se rompe la encapsulación de los módulos convencionales y sus contenidos. Sin embargo también supone admitir que existen ciertos módulos que definen las barreras de la encapsulación. Se trata de un tipo de descomposición asimétrica porque los *crosscutting concerns* se modularizan en ese tipo de módulo que se ha denominado *aspecto*. Por el contrario, si se utilizara el mismo tipo de módulo para representar todos los *concern* de un sistema, los que atraviesan y los que no, se estaría considerando un tipo de descomposición simétrica [Har+02]. La mayoría de los modelos orientados a aspectos son asimétricos ya que están fuertemente influenciados por *AspectJ* (que es asimétrico).

- Un modelo asimétrico se basa en una descomposición dominante (considerando la modularización funcional). Este tipo de modelos supone que los *aspectos* son *concerns* que atraviesan (*crosscut*) los módulos funcionales del sistema. En ellos, un *aspecto* se compone (*woven*) con los componentes que definen el sistema inicial. Así, componentes y *aspectos* tienen diferente estructura. Estos paradigmas soportan composición del tipo componente-*aspecto*. Algunos también soportan la composición *aspecto-aspecto*. Los modelos asimétricos son los más utilizados al ser más fáciles de integrar en las aproximaciones de desarrollo actuales que los modelos simétricos.

  Los modelos asimétricos promueven la representación y la encapsulación de los *crosscutting concerns* de modo que pueden evolucionar y reutilizarse, al considerarlos como una unidad. En ellos, los componentes definen la estructura inicial y los aspectos la completan. En estos modelos, la evolución de los sistemas es más sencilla.

- Sin embargo, en los modelos simétricos no existe una descomposición dominante y cada elemento del modelo se considera un *concern*. Así, en modelos como *MDSoC* (apartado 3.3.1) o *Prisma* (apartado 3.3.2), la noción de código transversal (*crosscutting*) no tiene sentido ya que un *concern* es una dimensión y cada módulo puede cruzar a otros, uniéndose posteriormente. Todos los componentes se tratan como elementos de primera clase, bloques con idéntica estructura, no siendo ninguno más importante que otro; sólo hay componentes que son los módulos básicos. La mayoría de estos modelos no usan el término *aspecto* (no necesitan diferenciar entre entidades que son *aspecto* y las que no lo son), pero requieren otros elementos para

representar este concepto. Por esta razón, los modelos simétricos se suelen considerar más flexibles y abstractos. Algunas aproximaciones, como en [NuKrFi94], se diseñan para capturar requisitos; similar es el concepto de *subject* [HaOs93]. Esta noción evoluciona a su vez hacia dos tipos de modelos: el propuesto por Clarke [Cla02, BaCl04a] que define el concepto de *theme* (apartado 3.3.1), y aquellos que utilizan el concepto de *dimensión* para estructurar el software, siendo ambos elementos de primera clase. Las dimensiones se denominan *hyperslices*, que se pueden definir como *aspectos* simétricos [OsTa00].

3.2. DSOA y evolución

El mantenimiento de los sistemas software ha sido un problema cuya resolución ha interesado a los desarrolladores a lo largo de la historia de la informática. Desde las conferencias de la OTAN en 1968 [NATO69] y 1969 [NATO70], en las que se planteó el problema del mantenimiento de los sistemas (surgiendo el término *crisis del software*), hasta hoy, se ha tratado de reducir el coste y el esfuerzo que supone el mantenimiento del software. Recientemente y dada la complejidad que van adquiriendo los sistemas a desarrollar, se ha hecho necesario dar a la evolución del software un tratamiento más cuidadoso, pues es un hecho que todos los sistemas están sujetos a cambios. El uso de diversos paradigmas ha reducido dicho coste y ha permitido que la evolución de los sistemas suponga la aplicación de un menor esfuerzo.

Uno de los paradigmas que se ha utilizado para facilitar la evolución de los sistemas y reducir su costo es el DSOA: la orientación a aspectos aporta simplicidad al proceso de evolución del software, ya que cuando se desea cambiar propiedades que afectan a un único interés (*concern*) del producto software, sólo habría que considerar las propiedades del *aspecto* que especifica el *concern* que cambia. Por eso, el DSOA, además de proporcionar mecanismos para separar los diferentes *aspectos* o *concerns* del sistema durante el desarrollo software, se puede considerar como una aproximación que permite afrontar de un modo diferente la resolución de algunos de los problemas relativos a la evolución del software. En este contexto, un *aspecto* se puede definir como

> *un mecanismo de abstracción que, de algún modo, se puede añadir a un sistema existente, de forma que el elemento incorporado no quede distribuido (disperso) entre varios módulos del sistema, sino que se mantenga independiente de ellos.*
>
> También se podría decir que un *aspecto* (*concern*) puede ser
>
> *cualquier criterio que permita separar partes de un sistema que tengan diferentes probabilidades de cambio o que tenga diferente impacto sobre la evolución.*

De este modo, y dado que el paradigma del DSOA permite modelar como artefactos software las propiedades que atraviesan los sistemas, los desarrolladores pueden adaptarlos más fácilmente a los cambios y a la evolución de estas propiedades, pues al modelarlas como aspectos (elementos independientes en el desarrollo) se pueden modificar (añadir, cambiar, eliminar) sin que los componentes del sistema resulten afectados. Así, durante la evolución, resulta sencillo incorporar nuevos elementos y los

problemas de la adaptación se resuelven más fácilmente, frente a la utilización de paradigmas tradicionales en los que este código transversal queda disperso por los diferentes módulos o componentes del sistema.

En los próximos párrafos, la exposición se centra en la evolución de los sistemas desde un punto de vista arquitectónico dentro del paradigma de DSOA. Se relacionan los conceptos de evolución y adaptabilidad, arquitectura del software y el desarrollo de sistemas orientado a aspectos.

Es interesante tratar estos tres conceptos conjuntamente y estudiar la evolución de los sistemas durante la arquitectura del software desde una perspectiva de la orientación a aspectos [NaPeMu05b].

Nuevos requisitos como aspectos

La evolución de los sistemas a veces está motivada por cambios en ciertos requisitos. Los cambios se pueden introducir durante el diseño arquitectónico, modificando únicamente las interacciones entre los componentes del sistema inicial, sin que dichos componente resulten modificados:

- En el capítulo 2 se ha mostrado la conveniencia de tratar la evolución de los sistemas a lo largo de las distintas etapas del ciclo de vida, en particular durante la arquitectura del software.
- Por otra parte, en esta sección se ha comentado que la *separación de aspectos* facilita la evolución de los sistemas, al permitir la incorporación al mismo de propiedades que se tratan como entidades software independientes sin que los componentes que forman parte del sistema se vean afectados, siguiendo el principio de inconsciencia propuesto por Filman y Friedman en [FiFr00].

A partir de estas premisas se puede razonar sobre la conveniencia de tratar la *incorporación* de nuevos requisitos desde un punto de vista arquitectónico como si fueran *aspectos* y asociarlos al sistema de un modo transparente al mismo. Si esto fuera así, en tiempo de mantenimiento se podría abordar la evolución de los sistemas de un modo sencillo. Sin embargo no todos los requisitos que se tuvieran que incorporar al sistema en un momento dado pueden tratarse de este modo. Si los requisitos a incorporar resultan ser ortogonales a los anteriores, sí sería interesante aplicar modelos arquitectónicos OA; en particular, se podría aplicar el modelo descrito en el Capítulo 4. Éste propone considerar los nuevos requisitos como *aspectos*, de modo que resulten incorporados al sistema considerando el principio de inconsciencia y haciendo que la evolución se acometa de un modo sencillo, sin tener que realizar cambios estructurales en la arquitectura del sistema original, sino cambiando aquello que sea necesario modificar. Si por el contrario los nuevos elementos cruzan a otros *aspectos* o requisitos ya incorporados siguiendo esta filosofía, el resultado no sería tan beneficioso, pues la arquitectura del sistema se vería degradada. En estos casos habría que plantearse retirar el elemento incorporado con anterioridad, modificarlo con las nuevas características y volver a incluirlo aplicando el modelo propuesto.

Aproximaciones en DSOA y evolución

A continuación se mencionan algunos trabajos que consideran la evolución de los sistemas desde el punto de vista de la arquitectura, la orientación a aspectos y/o el uso de LDA, si bien afrontan el tema desde una perspectiva distinta a la propuesta en el Capítulo 4 de esta tesis doctoral:

- El grupo ERCIM [MeMeTo04] trabaja en torno a la evolución del software, teniendo como objetivos identificar un conjunto de técnicas formales y definir herramientas que ayuden a los desarrolladores a llevar a cabo la evolución de sistemas complejos. En concreto están trabajando en torno a tres actividades: *aspect mining*, identificación de aspectos y evolución de aspectos (en el sentido de evolución del software OA).
- En [PeSeDu04] y [Pes+06] se considera que el Desarrollo Basado en Componentes (*DBC*), la POA y los LDA permiten representar la evolución desde puntos de vista complementarios. Proponen crear un *framework* integrando los tres conceptos. Por su relación con el modelo que se presenta en esta tesis, este trabajo se describe con mayor detalle en el apartado 3.3.5 *Aproximaciones al modelado OA*.
- En [Bar+04] se presenta *TranSAT Method*, un *framework* para especificar la evolución de la arquitectura software usando conceptos de OA, definiendo los pasos a considerar y describiendo los sistemas incrementalmente. Las especificaciones de la arquitectura se traducen al LDA *ArchJava*. Como en el caso anterior, y por su relación con el modelo que se presenta en esta tesis, se describe con mayor detalle en el apartado de *Aproximaciones al modelado OA* de este capitulo.
- En [And+01] se define un *framework* que extiende un modelo de componentes y un LDA basado en XML con conceptos de OA. Presenta, asimismo, una aproximación arquitectónica de tres niveles para llevar a cabo la evolución del software basándose en la separación entre computación, coordinación y configuración. Se define una primitiva de modelado para encapsular las interacciones entre componentes de modo transparente a ellos.
- DAOP [Pin+02] es una plataforma dinámica distribuida para sistemas OA en la que componentes y aspectos se consideran entidades de primera clase que se componen en tiempo de ejecución mediante un nivel de *middleware* que almacena la información de la arquitectura y los componentes. La estructura de composición de este modelo facilita la evolución. Por su relación con el modelo que se presenta en esta tesis, este trabajo también se describe con mayor detalle en el apartado 3.3.5.
- PRISMA [Per+05] es un modelo que soporta la evolución dinámica de las arquitecturas mediante la definición de un meta nivel y de las propiedades reflexivas de su LDA. La evolución se puede clasificar como evolución de los tipos y evolución de la configuración (o reconfiguración). La evolución de los tipos se obtiene invocando servicios de evolución que no afectan a la estructura de la arquitectura o a la comunicación entre elementos arquitectónicos. Por el contrario, la configuración de la arquitectura evoluciona cuando se invocan los servicios de evolución que actualizan los elementos arquitectónicos, la comunicación entre ellos y la estructura de la arquitectura. Los servicios de evolución afectan a la vista externa de los elementos arquitectónicos del modelo. Por otra parte, tanto la creación como la destrucción de nuevas instancias de componentes, conectores y sistemas produce un cambio en la arquitectura. La evolución de las arquitecturas se puede realizar en

tiempo de ejecución, lo que permite la reconfiguración dinámica de las arquitecturas mediante la ejecución de los servicios de evolución en tiempo de ejecución.

Como conclusión de esta sección se puede decir que los conceptos de *orientación a aspectos* y de *arquitectura del software* están relacionados con el de *evolución*, de modo que facilitan el desarrollo y mantenimiento de los sistemas complejos. Tratando ambos conceptos conjuntamente, se potencian las ventajas de cada uno. La propuesta que se presenta en el Capítulo 4 es una muestra de ello: *AOSA Model* es un modelo arquitectónico OA que facilita la evolución y el mantenimiento de los sistemas al permitir, en tiempo de evolución, la incorporación de nuevos requisitos, considerando éstos como aspectos (si son ortogonales) [NaPeMu05b]. Los nuevos requisitos se insertan en el diseño del sistema, sin tener que modificar el código de los componentes.

3.3. Modelado de aspectos a lo largo del ciclo de vida

A lo largo del proceso de desarrollo de un sistema software, los aspectos y *concerns* surgen en diferentes etapas y se refieren a requisitos funcionales y no funcionales. Dependiendo de la fase de desarrollo, las propiedades transversales (*crosscutting concerns*) pueden tener mayor o menor granularidad, así como variar en su número.

El DSOA pretende llevar a cabo una identificación de aspectos en las etapas iniciales del ciclo de vida y mantener su trazabilidad a lo largo de todo el proceso. Ésta es importante para mantener la consistencia del sistema y su capacidad de evolución. Los mecanismos de trazabilidad determinarían qué aspectos identificados en etapas iniciales serán o deberían ser aspectos en etapas posteriores.

A continuación se muestra de modo resumido cómo se pueden considerar las diferentes fases del ciclo de vida desde la perspectiva de la orientación a aspectos. Para ello, se comentan los modelos OA más interesantes. En particular, algunos se estudian en relación al trabajo que se presenta en esta tesis doctoral.

3.3.1. Ingeniería de requisitos orientada a aspectos

La ingeniería de requisitos orientada a aspectos proporciona soporte para la separación de las propiedades transversales (*crosscutting concerns*) identificadas durante la ingeniería de requisitos. Las técnicas de ingeniería de requisitos OA reconocen la importancia de considerar estos *crosscutting concerns* ya que facilitan la detección de los conflictos que puedan surgir debido al enmarañamiento que puedan producir los requisitos identificados como transversales. La identificación temprana de estos conflictos facilita su resolución también en fases iniciales. Ejemplos de *concerns* a considerar en las etapas iniciales son tiempo de respuesta, calidad, seguridad, etc.

Dentro de este apartado se pueden distinguir dos grandes grupos: a) aquellas aproximaciones que, desde el punto de vista de la ingeniería de requisitos, consideran los módulos aspectuales de granularidad fina y b) las que los consideran unidades modulares de granularidad gruesa:

a) Una de las técnicas que reconocen la importancia de considerar los *crosscutting concern* en fases iniciales es la llamada *Ingeniería de Requisitos Orientada a Aspectos* (o en su denominación inglesa *Aspect Oriented Requirements Engineering, AORE*). Las aproximaciones que se han desarrollado dentro de este grupo pretenden obtener una representación modular de los requisitos que resulten ser transversales para ese desarrollo. De este modo, los desarrolladores podrían asegurar la trazabilidad de los *crosscutting concern* a lo largo del ciclo de vida. Los modelos en AORE adoptan el principio de la *separación de aspectos* desde la fase de especificación de requisitos, lo que ha venido en llamarse separación temprana de intereses (*early separation of concerns*), así como proponen una representación de los *crosscutting concerns* como artefactos propios de la fase de requisitos.

   Las aproximaciones desarrolladas dentro de este grupo se plantean también cómo realizar la composición de los aspectos con otros elementos del sistema. Algunos trabajos destacados en este marco se mencionan a continuación:

   - *MDSoC* son las siglas con las que se ha popularizado el modelo desarrollado por Ossher y Tarr [OsTa01] cuyo nombre completo es *Multidimensional Separation of Concerns*. Es un modelo simétrico en el que los aspectos son elementos de primera clase; se basa en la idea de que todos los artefactos de desarrollo software están constituidos por múltiples aspectos superpuestos, y en que el desarrollo software se puede beneficiar de ello si los sistemas se pudieran componer y descomponer de un modo flexible, teniendo en cuenta las distintas combinaciones posibles de los aspectos. El modelo introduce el concepto de *hyperslice* que permite encapsular y manipular las interacciones entre aspectos y su integración en el sistema usando *HyperJ* como herramienta; cada componente y *concern* tiene su propia dimensión (*hyperslice*). En [MoRaAr05] se propone la descomposición de los requisitos con independencia de su naturaleza funcional o no funcional. Éste es un modelo muy expresivo donde cada propiedad se representa como la cara de un *hipercubo*, de forma que se puede escoger cualquier conjunto de propiedades como base para proyectar la influencia de otro conjunto de propiedades sobre ellas. Este trabajo propone un mecanismo para la composición de propiedades y el análisis de la influencia de unas propiedades sobre otras. Finalmente, la estructura arquitectónica sólo llega a crearse en aquellas dimensiones que se van a mezclar.
   - El modelo *Cosmos* de Sutton [SuRo02] sigue la misma filosofía (modelo simétrico); está basado igualmente en *HyperJ* que se aplica durante la especificación de requisitos. Esta aproximación define un *espacio de aspectos* en términos de *concerns*, relaciones, predicados y temas (*topics*). Los *concerns* se consideran como entidades de primera clase, clasificándolos en dos categorías: físicos y lógicos. Los *concerns* físicos se refieren a los elementos del sistema inicial y los lógicos se refieren a entidades conceptuales. Las relaciones reflejan

cómo interactúan los *concerns*. Los predicados representan las condiciones que deben cumplir las relaciones.

- *Arcade*: presentada en [Ara+02] y desarrollada en [Ras+03] es una de las primeras aproximaciones que utilizaron la orientación a aspectos en la etapa de especificación de requisitos. De hecho, los autores introdujeron el término *Early Aspects* para referirse a los aspectos identificados durante la ingeniería de requisitos y diseño arquitectónico; propusieron así el uso de aspectos como primitivas de modelado. *Arcade* es un modelo que soporta la aproximación AORE y propone una técnica para separar requisitos aspectuales y no aspectuales, así como las reglas para llevar a cabo su composición. Por otra parte, *Arcade* pretende llevar a cabo la modularización y composición de los aspectos a nivel de requisitos asegurando su consistencia mediante la detección de conflictos; sigue un modelo asimétrico y usa el concepto de *puntos de vista* para identificar los puntos en los que los requisitos funcionales son atravesados (*crosscut*) por *concerns* no funcionales. Finalmente utiliza XML para la representación de los artefactos que define.

b) Las aproximaciones de AORE, en general, no especifican la granularidad de los módulos aspectuales. Sin embargo, las técnicas basadas en componentes OA sí consideran como unidad cada aspecto en un sistema basado en componentes. Estos aspectos son módulos de granularidad gruesa, evitándose el mezclado (*weaving*) y usando reflexión en tiempo de ejecución:

   - La *Ingeniería de Componentes Orientada a Aspectos* (o en su notación inglesa *Aspect Oriented Component Engineering, AOCE*) es una metodología de desarrollo software debida a Grundy [Gru00] que considera los aspectos como un conjunto de servicios que proporcionan y que requieren los componentes de un sistema. AOCE extiende el modelo de componentes con el modelo de aspectos; además, componentes y aspectos se componen en tiempo de ejecución.

     La *Ingeniería de Requisitos de Componentes OA (Aspect Oriented Component Requirement Engineering, AOCRE)* debida también a Grundy [Gru99] se puede considerar como una primera etapa de AOCE. AOCRE se centra en la identificación y especificación de requisitos funcionales y no funcionales, relativos a los requisitos de los componentes del sistema, en función de los servicios que cada uno proporciona o requiere, y que se puedan especificar a alto nivel en un modelo de diseño; así, AOCRE se puede incluir también en el apartado de diseño de alto nivel. Como AOCRE se puede considerar como continuación de AOCE, los aspectos se definen en ambos de la misma manera, siendo considerados como componentes. La definición de un componente de aspecto se hace separadamente de la especificación de los componentes regulares, para que sean independientes y reutilizables. Sin embargo, se introduce un elemento denominado *Aspect Manager* que lleva a cabo la coordinación de componentes regulares y de aspecto. Aunque en esta aproximación no hay conectores, el *Aspect Manager* sí es considerado como tal para realizar el tejido entre componentes y aspectos en tiempo de ejecución. En esta aproximación se consideran aspectos: la persistencia, la distribución, la configuración, entre otros.

Se puede decir que AOCRE facilita el diseño y la implementación de los componentes ya que, además, soporta un lenguaje de diseño basado en una plataforma de implementación OA dinámica (*JViews* [GrMu98]) y basada en componentes, en la que los aspectos se especifican también como tales. Una herramienta, *JComposer* [GrMu98], ayuda a desarrollar sistemas siguiendo este modelo.

- *Theme* [ClBa05, Theme] es un modelo simétrico que surgió a partir de los trabajos de S*ubject-Oriented Programming* [HaOs93]. Se centra en el concepto de tema (*theme* en el modelo) que es una unidad representativa de funcionalidad cohesiva y que se define como *una característica, propiedad o requisito de interés que debe manejar el sistema*. Un *theme* es un módulo que describe un *concern* desde una perspectiva concreta; la arquitectura de cada *concern* se define como una vista independiente que se relaciona posteriormente. Según esto, en el modelo no se define una entidad aspectual explícita, sino el concepto de *theme* que es más general. Esta aproximación expresa los *concerns* como construcciones conceptuales y de diseño llamados *módulos theme* (que capturan los *concerns)* que atraviesan o no los elementos del sistema durante el diseño. Cuando un *theme* tiene muchas interacciones con otros se pueden superponer en un *patrón de composición*. Esta es una estructura genérica que se puede parametrizar para combinarla más fácilmente con otras.

  Una parte de este modelo es *Theme/Doc* [BaCl04b] que realiza el análisis de los documentos en los que se especifican los requisitos, y soporta la identificación y análisis de aspectos en la documentación de los requisitos [BaCl04b], donde los aspectos representan descripciones de los comportamientos que intervienen y se mezclan. Utiliza el concepto de *theme* para analizar las relaciones entre los comportamientos descritos en un documento de requisitos e identificar los aspectos. Los *theme* se clasifican en *theme* base y *theme* transversales. Los *theme* transversales son aspectos en *Theme/Doc*. Así, esta parte del modelo se dirige a las últimas etapas de la ingeniería de requisitos, cuando ya hay al menos un documento de requisitos inicial para poder analizarlo léxicamente.

- El enfoque basado en *Casos de Uso* (*Aspect-Oriented Software Development with Use Cases, AOSD/UC)* [Jac03] extiende el modelo de casos de uso añadiendo los conceptos de puntos de corte (*pointcuts*)*, use case slices* y *use case modules.* El modelo propone representar las funcionalidades propias del sistema como *casos de uso base* que pueden tener *puntos de extensión* (*extension points*). Estos son puntos concretos dentro de un determinado caso de uso en los que se puede añadir comportamiento, que se especificaría en casos de usos denominados extensiones (*use case extension*). Este modelo sigue la aproximación asimétrica en tanto en cuanto, por una parte está fuertemente influenciada por el concepto de *aspecto,* y por otra distingue dos tipos de casos de uso: los caso de uso base y las extensiones. Los primeros son independientes unos de otros y se refieren a los requisitos base. Los casos de uso *extension* son características adicionales del sistema que se superponen a los casos de uso base y se definen independientemente de éstos.

- Una aproximación similar a ésta es la denominada *Aspectual Use Case Driven* [ArMo03] que extiende el modelo de los casos de uso para soportar requisitos no funcionales e identificar el solapamiento de los casos de uso funcionales. Para ello se definen casos de uso denominados *inclusion* y *extension*. Igualmente introduce los atributos de calidad como casos de uso (que no estarían asociados a los casos de uso base).

3.3.2. Arquitectura software Orientada a Aspectos

Los trabajos a nivel de arquitectura se dirigen hacia la identificación, especificación y evaluación de aspectos durante la fase de diseño arquitectónico. Desde una perspectiva de DSOA, la arquitectura pretende proporcionar un modo de identificar y modelar aspectos a alto nivel de abstracción. Esto es importante porque el diseño arquitectónico incluye la toma de decisiones tempranas sobre el diseño, que tienen especial impacto en fases posteriores.

La transformación de un modelo de requisitos en un modelo de diseño permite obtener, por una parte los requisitos que serán elementos del sistema, y por otra, los requisitos que se transformarán en aspectos. Ciertas propiedades funcionales o no funcionales del sistema se pueden convertir en aspectos a nivel de diseño, teniendo entidad propia. Sin embargo, hay otras propiedades (funcionales o no) que no tendrán su correspondiente entidad de primera clase en esta fase. Además, no hay que olvidar que las reglas de composición de requisitos deben tener su equivalencia en la fase de diseño.

En las aproximaciones de diseño arquitectónico OA, un aspecto arquitectónico (apartado 3.1.2) es un *módulo arquitectónico que tiene influencia sobre varios módulos arquitectónicos del sistema*; es decir, es una entidad que permite la modularización y encapsulación de especificaciones arquitectónicas. En este sentido, también hay que definir cómo los aspectos se relacionan con el resto de los elementos de la arquitectura, descritos como componentes (en la mayoría de los modelos) y conectores (como en [Cue+04b] o [Bat+06b]). Igualmente, habrá que establecer cómo son estas nuevas relaciones; diversos trabajos ([Kan03], [PiFuTr05], [KaKa03], [BaMeDu06], entre otros) estudian la relación entre los aspectos y las propiedades de la composición aspectual. Seguridad, autorización, autentificación, encriptado/desencriptado son ejemplos de aspectos arquitectónicos. Aproximaciones no-OA pueden llevar a no considerar estas características aspectuales durante el diseño de la arquitectura de un sistema y no ser resueltas hasta el nivel de programación, lo que puede provocar la obtención de código enmarañado. Por el contrario, las aproximaciones de diseño arquitectónico OA proporcionan mecanismos para identificar aspectos arquitectónicos, quedando éstos definidos de un modo explícito en esta fase.

Las aproximaciones en esta fase se pueden considerar divididas en varias categorías según se propone en [Cue+05] y en [Chi+05]. A modo de resumen, hemos seleccionado algunos de los trabajos realizados en esta etapa y se han organizado en los siguientes grupos, sin pretender que ésta sea una clasificación definitiva:

a) Aproximaciones de *modelado arquitectónico*. En este grupo se pueden considerar los siguientes modelos:

   - *Perspectival Concern-Space (PCS) Method* es un *framework* desarrollado por Kandé [Kan03] que consideran el DSOA desde una perspectiva estructural. En él se define una técnica para describir *concerns* de múltiples dimensiones en una vista arquitectónica formada por uno o más modelos y diagramas. En esta aproximación se define una *perspectiva* como un "modo de ver" en un espacio multidimensional de aspectos, desde un punto de vista específico. Se consideran explícitamente los *concerns* en la definición interna de las abstracciones arquitectónicas. En este modelo no hay una entidad aspectual explícita sino que su definición es fija. El comportamiento relativo a los aspectos (*concerns*) se puede definir como una parte de la especificación. Este es un modelo simétrico basado en el modelo MDSoC y en el estándar IEEE-1471, y usa UML para especificar las arquitecturas OA. Como resultado se tiene un LDA propio con características de composición.

   - *CAM/DAOP* [PiFuTr05] es una aproximación basada en componentes y aspectos que soporta la separación de aspectos desde el diseño a la implementación. Esta aproximación define el llamado Modelo de Componentes y Aspectos (*Component-Aspect Model, CAM*), el lenguaje *DAOP-ADL* y la Plataforma *DAOP*. Este modelo se estudia más detalladamente en el apartado 3.3.5. *Aproximaciones al modelado OA*.

   - PRISMA [Per06, Per+06]. Este modelo incluye la definición de aspectos (*concerns*) como una entidad explícita a nivel arquitectónico. Es un modelo simétrico en el que los aspectos son parte de la definición de un elemento arquitectónico que se describe como un conjunto de aspectos, un tejido (*weaving*) explícito y una interfaz común. Además, permite definir arquitecturas de sistemas complejos y se caracteriza por la integración de DSBC y de DSOA, así como por sus propiedades reflexivas. Esta integración se realiza definiendo como aspectos las características relevantes de un elemento arquitectónico (distribución, coordinación, seguridad, etc.), los *concerns*. Por su relación con el trabajo objeto de esta tesis, se estudia con mayor detalle en el apartado 3.3.5.

b) Aproximaciones para la evolución arquitectónica. En este grupo se pueden considerar:

   - *TranSAT Method* [BaDuMe05, BaMeDu06], como ya se comentó en el apartado 3.2.2, es un *framework* para la especificación de la evolución del software. Se centra en facilitar la evolución de la arquitectura mediante la aplicación de los principios de programación OA en el contexto arquitectónico. La especificación del sistema se transforma en una arquitectura integrando los *concerns arquitectónicos*. *TranSAT* se propone resolver tres problemas: i) la integración de un nuevo *concern*; ii) un *concern* debe ser lo suficientemente genérico para reutilizarlo en varios contextos; y iii) la integración de un nuevo *concern* no debe modificar la consistencia de la descripción arquitectónica existente. Para lograr estos objetivos, el modelo introduce el concepto de patrón arquitectónico

(*architecture pattern*). Este modelo se estudia con más detalle en el apartado 3.3.5.

- AO-Rapide [Pal04] es un modelo que facilita la inserción de aspectos en una arquitectura previamente diseñada. Los aspectos se tratan como componentes siendo necesaria la definición de un conjunto de elementos que facilitan el entrelazado. El modelo se apoya en la definición de AO-Rapide como lenguaje arquitectónico OA y en el uso de un modelo de coordinación (tambien se estudia en 3.3.5).

c) Se puede identificar otro bloque denominado *evaluación arquitectónica*. Este grupo se puede considerar representado por el método de análisis de arquitecturas de aspecto (*Aspectual Architecture Analysis Method* conocido por sus siglas *ASAAM)*, entre otros:

- *ASAAM* [Tek04] es una aproximación de diseño arquitectónico que propone identificar y especificar explícitamente aspectos arquitectónicos en etapas tempranas del ciclo de vida usando reglas que permiten deducirlos. Esta aproximación se crea sobre el método de análisis de arquitectura basado en escenarios (*Scenario-Based Architecture Analysis Method*) y por ello debe considerarse como una aproximación complementaria a este método, al que extiende y refina. La ventaja de *ASAAM* es el soporte simétrico que ofrece para la gestión de los aspectos arquitectónicos, así como para identificar los componentes que atraviesan (*crosscut*) el diseño mediante el uso de escenarios. Los artefactos en *ASAAM* son, por tanto, escenarios (que pueden ser a su vez directos, indirectos y aspectuales) y componentes arquitectónicos (que son cohesivos, indefinidos, enmarañados *–tangled-* y compuestos). La principal diferencia entre esta aproximación y otras es que en ella se evalúa una arquitectura orientada a aspectos en lugar de especificar e implementar una arquitectura software. Así, este modelo se puede considerar como un modelo de evaluación arquitectónica que permite evaluar si una arquitectura OA ha considerado los aspectos adecuados, y si la factorización de los aspectos es también la adecuada.

d) *Ligadura arquitectónica*. En este grupo se puede considerar:

- El trabajo de Katz [Kat93] sobre *superposición* define un nuevo tipo de ligadura que se considera como un concepto de primera clase: "*… la ligadura es una abstracción arquitectónica…"*. El concepto de *superposición* propuesto por Katz se define como una *relación que define una estructura composicional de alto nivel*. En [ShKa03] los conceptos de *superposición* y de aspecto se tratan conjuntamente para definir una aproximación OA. Una *superposición* implementa un algoritmo que se puede aplicar a distintos sistemas iniciales sobre los que se superponen los aspectos. Las *superposiciones* se pueden declarar, especificar y verificar independientemente porque no están relacionadas con ningún sistema básico concreto. La *superposición* se introduce como un conjunto de aspectos parametrizados y con una especificación formal.

e) *Modelos de aspecto*. Dentro de este bloque se puede considerar:

- *Fac* [PeSeDu04] es una extensión de *Fractal* que integra conceptos de programación basada en componentes y de orientación a aspectos a nivel arquitectónico. Incluye el concepto de componente de aspecto y de ligadura de aspecto (*aspect binding*) que conecta componentes base con un componente de aspecto, pero no hace distinción entre el comportamiento de un aspecto y un componente del sistema. Este modelo se considera más detalladamente en el apartado 3.3.5.
- *JasCo* [SuVaJo03] es un modelo de programación que combina conceptos de desarrollo basado en componentes y de programación OA. El modelo extiende el de Java Beans definiendo *Aspect Beans* y conectores. El modelo de conector de esta propuesta es su mayor aportación.
- *FuseJ* [SuFrVa06] define una aproximación que relaciona aspectos y componentes, pero no define ninguna construcción específica para representar los aspectos; se basa en la definición de componentes de aspecto como componentes arquitectónicos; define un concepto de interfaz que expone la funcionalidad interna de un componente y ofrece los puntos de acceso para la interacción con otros componentes. El modelo de composición define un tipo especial de conector que extiende el concepto tradicional incluyendo construcciones que permiten especificar el comportamiento transversal. *FuseJ* no proporciona soporte explícito para definir la interacción entre aspectos ni modularización de aspectos heterogéneos. Sólo permite cuantificación sobre la llamada a métodos y no tiene la noción de configuración en la que se declare un conector y sus ligaduras; en *FuseJ* la conexión se define dentro de los conectores. Es un lenguaje de programación para la definición de arquitecturas software basadas en componentes sobre el modelo de componentes de *Java Beans,* al igual que *JasCo,* que es el trabajo previo del mismo grupo; es, por tanto, una aproximación dependiente del lenguaje en la que no se define ningún concepto nuevo para representar aspectos, sino que los implementa como componentes. Las tareas de coordinación se ejecutan usando conectores como primitivas OA para definir los puntos de corte (*pointcuts*) y los *advices* (código que especifica los aspectos). Una herramienta facilita la definición de las arquitecturas OA definidas bajo esta aproximación. Tanto *FuseJ* como *JasCo* han sido propuestos para trabajar con sistemas orientados a aspectos a un nivel de abstracción más bajo por lo que también podrían haberse incluido en el apartado 3.3.3.
- *Asbaco* [MeBu05] (*Aspect-Based Controller*) utiliza un modelo basado en componentes para capturar la estructura de la parte controladora de un componente (según se propone en *Fractal*); basándose en este modelo después define un modelo basado en aspectos para especificar las extensiones de la parte controladora. Utiliza el término *microcomponente* (*microcomponent)* para referirse a la parte controladora del componente, reservando el término componente para referirse a los componentes generales del sistema. Un *microcomponente* permite modelar las características no funcionales de los componentes; así, introduce los aspectos en los componentes. *Asbaco* define el

concepto de componente de aspecto como un conjunto de extensiones de la parte controladora de los componentes para realizar la intercepción y los *advices*. Cada *microcomponente* tiene un conjunto de interfaces cliente y servidor, permitiendo su relación con otros componentes. Basado en este concepto *Asbaco* define los componentes de aspecto que permiten modelar las características no funcionales de los componentes. Las extensiones de las partes controladoras se integran aplicando selectivamente los componentes de aspecto a la jerarquía de componentes de una aplicación. Por otra parte, usan un lenguaje de descripción de arquitecturas para los *microcomponentes* basado en XML, que luego se traduce a Java.

Estos modelos (*JasCo, FuseJ, Asbaco*) no se estudian en detalle pues se trata de modelos y lenguajes que consideran los sistemas y su adaptación mediante la inclusión de aspectos a más bajo nivel que el arquitectónico.

f) *Modelos de múltiples dimensiones*. Dentro de este bloque se puede considerar:

- La propuesta de Katara [KaKa03] está basada en *superposición* en la que cada *concern* se define a nivel arquitectónico como una vista independiente que posteriormente se compone. Trata los aspectos independientes unos de otros, pero considera que cuando más de un aspecto implementa un *concern* existe solapamiento entre ellos. Este solapamiento significa que los aspectos están relacionados y no son completamente independientes, aunque sí sean entidades independientes. Los autores definen el concepto de *sub-aspecto* como aspectos que no son completamente ortogonales, pero que se pueden componer para formar un aspecto compuesto que representa un único *concern*. Dada la posible dependencia entre aspectos, para asegurar que éstos trabajan adecuadamente una vez compuestos, es importante tener en cuenta el orden en que se aplican los *sub-aspectos*. Finalmente se puede decir que en este modelo los aspectos se describen como una composición de elementos que proporcionan y requieren servicios.
- También se pueden incluir en este apartado los trabajos ya mencionados de Clarke a nivel arquitectónico [BaCl04b] y *MDSoC* de Ossher y Tarr [OsTa00, Tar+05] en los que cada componente tiene su propia dimensión.

3.3.3. Diseño Orientado a Aspectos

Durante el diseño detallado se confirma la corrección de la arquitectura y que se cumplen los requisitos. Las decisiones que toma el diseñador durante el Diseño OA (DOA) se basan en las decisiones tomadas durante la ingeniería de requisitos o el diseño arquitectónico; de esta manera, cuando los requisitos o la arquitectura cambian, las decisiones tomadas tienen que revisarse y posiblemente haya que alterarlas. Por tanto, el objetivo del DOA, como cualquier actividad de diseño del software, es caracterizar y especificar el comportamiento y la estructura de los sistemas.

Los *concerns*[11] de un sistema, que quedan dispersos y se mezclan cuando se aplican aproximaciones de desarrollo tradicionales, pueden modularizarse aplicando técnicas de DOA. De este modo, aumenta la cohesión de los módulos y se reduce el acoplamiento de los mismos. Así pues, el DOA soporta la especificación de la modularización y composición de los *concerns,* y se centra en la representación explícita de los *crosscutting concern* utilizando lenguajes de diseño OA que especifican los aspectos y cómo se componen; asimismo dispone de un conjunto de reglas de composición semántica para describir cómo integrar los aspectos. Los *concerns* que surgen en esta etapa son debidos a la tecnología escogida para llevar a cabo el diseño y la posterior implementación. La gestión de excepciones, *catching*, *buffering*, sincronización son algunos de ellos.

Los modelos de DOA se pueden considerar ligados a extensiones de UML o independientes de este lenguaje de modelado:

- Extensiones de UML orientadas a aspectos. Los trabajos desarrollados en este ámbito representan los conceptos de orientación a aspectos a nivel de diseño. Este tipo de modelos se incluye dentro del llamado *Modelado Orientado a Aspectos* o *Aspect Oriented Modeling* [Ald]. Una de estas extensiones es la propuesta de Stein: *Modelado de Diseño Orientado a Aspectos* o *Aspect-Oriented Design Modeling (AODM*) [StHaUn02] que extiende UML con una notación de diseño para soportar conceptos OA; estos elementos están próximos a *AspectJ*. El modelo soporta la especificación del comportamiento y la estructura transversal del sistema; ésta se expresa en los diagramas de clases y con una plantilla parametrizada de diagramas de colaboración.

  Theme proporciona un lenguaje basado en UML, *Theme/UML* [Cla02], que extiende el meta-modelo de UML, y permite modelar la estructura y el comportamiento de los aspectos usando UML. Cada tema *–theme-* se encapsula en un paquete UML conteniendo todos los elementos que lo modelan. Los temas *–theme-* parametrizados son plantillas UML que contienen una serie de elementos como parámetros. La característica que lo distingue es que proporciona un tipo especial de interacción. El modelo final de la aplicación se obtiene especificando relaciones de ligadura que instancien los parámetros entre los temas *–theme-* parametrizados y los no parametrizados.

- *CoCompose* [WeBe02] es un lenguaje de diseño no orientado a UML. Soporta la representación de aspectos reutilizables de alto nivel como "*feature"*: una *feature* es un aspecto de alto nivel que atraviesa los límites de las aplicaciones. Las *features* son también construcciones abstractas que describen patrones de diseño, estando por tanto semánticamente bien definidos. *CoCompose* es un lenguaje de diseño gráfico que se puede usar para realizar DOA; también soporta diseños ejecutables. Finalmente, esta aproximación se centra en la automatización de la transformación desde modelos de diseño de alto nivel a plataformas de implementación específicas.

[11] *Concerns*: se mantiene este término por resultar más claro que su traducción: propiedades, intereses o características.

A alto nivel, la principal aportación del DOA ha sido proporcionar medios a los diseñadores para modelar sistemas OA, razonar sobre los aspectos (*concerns*) de un modo independiente y capturar las especificaciones de modularidad de los aspectos (*concern*) de diseño. Donde hay modularización también debe haber un modo de especificar cómo se deben componer esos módulos con el resto del diseño del sistema. Asimismo, el DOA proporciona un mecanismo para resolver los conflictos entre *concerns* y para especificar cómo cooperan; también el DOA contribuye a la trazabilidad de los *concerns* a lo largo del ciclo de vida, incrementando así la comprensibilidad y la mantenibilidad de los sistemas.

3.3.4. Programación Orientada a Aspectos

Para la implementación de las aplicaciones OA es recomendable utilizar lenguajes que soporten el concepto de aspecto: los Lenguajes de Programación Orientados a Aspecto (LPOA). Dependiendo del modelo de diseño que se haya utilizado para representar el sistema, así convendrá utilizar un LPOA u otro. Hay varios lenguajes de programación OA: *AspectJ*, que es una extensión de Java, es el más importante y el más popular. Además se pueden mencionar los siguientes:

- *CaesarJ* [MeOs03] es un modelo de programación POA con su propio lenguaje de programación; es un modelo dependiente de la tecnología, y se caracteriza por desarrollar módulos de alto nivel para expresar los aspectos independientemente de los puntos de unión (*join points*).
- *DemeterJ* [Lie+99] fue concebido inicialmente como un modelo de componentes; su evolución llevó hacia la definición de componentes aspectuales (*aspectual components*) en los cuales, con una mínima modificación, se pueden especificar aspectos.
- *JAC* [Pau+04] y *JBOSS AOP* [JBoss] que están relacionados con *AspectJ*.

3.3.5. Aproximaciones al modelado OA

Los apartados anteriores muestran diversos trabajos realizados en el marco del DSOA, desde el punto de vista de las distintas fases del ciclo de vida. Próximos al trabajo presentado en esta tesis están aquellos modelos mencionados en los apartados 3.3.1 y 3.3.2, en los cuales los aspectos se definen como elementos relevantes de un cierto sistema, pero de un modo diferente a como se consideran en esta memoria. Algunos de ellos se comentan más abajo. Para que sea más evidente la diferencia entre el modelo que se presenta en el Capítulo 4 de esta tesis y los modelos reseñados a continuación se ha considerado interesante destacar aquí las siguientes características del modelo que se propone:

· Se define una arquitectura para sistemas complejos integrando las ventajas del desarrollo de sistemas basados en componentes y el DSOA.
· El modelo se refiere al uso de componentes convencionales para contener los aspectos. Esto es, los aspectos se definen como componentes arquitectónicos

regulares, en lugar de proporcionarse una abstracción adicional para definirlos (como pueden ser vistas -*views*- o *slices*). Esto tiene la ventaja fundamental de dar uniformidad al modelo.

- El modelo arquitectónico propuesto sigue un modelo asimétrico en el que se distinguen los componentes que describen el sistema de aquellos que representan otro tipo de características: los aspectos.
- Además, se define una estructura de coordinación de 2 niveles para soportar el mezclado (*weaving*) de aspectos. La aplicación de propiedades reflexivas del modelo permite la modificación del comportamiento del sistema inicial al incorporar los aspectos en un meta nivel.
- El modelo se formaliza mediante la definición de un LDA orientado a aspectos (*AspectLEDA*) que extiende un lenguaje de descripción de arquitecturas ya existente (LEDA), dotado de una fuerte base formal (cálculo π). Esto permite, por un lado generar de un modo sencillo un prototipo ejecutable del sistema, ya que el lenguaje en el que se apoya lo permite; y por otro, asegurar la consistencia del sistema, y analizar y validar la arquitectura.
- Un juego de herramientas facilita al arquitecto de software la tarea de definir el sistema extendido con aspectos.

3.3.5.1. Aproximaciones al modelado OA en ingeniería de requisitos

Dentro de este bloque interesa resaltar especialmente el trabajo de Jacobson, por su relación con el modelo que se presenta en esta tesis doctoral:

Aspect-Oriented Software Development with Use Cases (*AOSD/UC*) [Jac03] extiende el modelo de casos de uso tradicional añadiendo los conceptos de *extension points* en los casos de uso, *use case slices* (que representan la especificación de un caso de uso en una cierta etapa del desarrollo) y *use case modules* (que contienen la especificación de un caso de uso a lo largo de todas las etapas) para agrupar artefactos del desarrollo.

En esta aproximación se distinguen dos tipos de casos de uso: casos de uso base (*peer*) y casos de uso *extension*. Los casos de uso base son independientes entre sí y representan el comportamiento base del sistema. Los casos de uso *extension* se refieren a características adicionales a aplicar sobre los casos de uso base. Aunque la definición de los casos de uso *extension* se puede hacer independientemente de los casos base, normalmente se usan junto con ellos. Los casos de uso *extension* pueden ser referenciados desde los casos de uso base a través de los puntos de extensión (*extension points*).

Jacobson propone que los *crosscutting concerns*[12] se consideren como casos de uso *extension*, en el sentido en que la realización de cada caso de uso afecta o puede afectar a varios módulos; igualmente propone representar la funcionalidad propia del sistema como *casos de uso base*, que tienen o pueden tener *puntos de extensión*. Estos son puntos concretos dentro de un determinado caso de uso en los que se puede

[12] *Crosscutting concerns*: propiedades o intereses transversales.

añadir comportamiento, que se especificara en los *use case extension* que representa un aspecto y se ejecutará dependiendo del valor de las condiciones expresadas en el *punto de extensión* del/de los casos de uso al/a los que extiende. El diagrama de casos de uso final representa los diferentes *concerns* separadamente.

Finalmente, se puede decir que *AOSD/UC* se aplica a lo largo de todo el desarrollo software, desde la especificación de los requisitos hasta la implementación. La captura de requisitos sigue un proceso iterativo de modo que se realiza en dos fases: una primera en la que se modela la funcionalidad básica del sistema mediante casos de uso base; y una segunda fase en la que se completan con requisitos que representan los aspectos. Estos casos de uso que extienden el sistema (*use case extension*) pueden ser genéricos y tomarse de un repositorio para su reutilización. En fases posteriores la propuesta considera *use case slices* y *use case modules* para representar y agrupar los diversos artefactos software que surjan a lo largo del desarrollo.

Se puede concluir que este método proporciona un proceso sistemático a través del cual la separación de aspectos (*concerns*) que se establece en el diagrama de casos de uso del sistema se mantiene a lo largo de todo el ciclo de vida.

La propuesta que se presenta en el Capítulo 4 de esta memoria está relacionada con la descrita en este apartado, en tanto en cuanto en *AOSA Model* la especificación de requisitos está inspirada en *AOSD/UC*:

- Primero, se hace una especificación de los requisitos base del sistema y se expresa mediante el correspondiente diagrama de casos de uso.
- Por otra parte, los aspectos identificados, en una segunda vuelta del proceso iterativo que también sigue *AOSA Model*, se representan mediante *use case extension*, que se incorporan al diagrama inicial.
- Los casos de uso obtenidos se representan mediante diagramas de secuencia que muestran las interacciones entre componentes.

A partir de aquí, el modelo propuesto y *AOSD/UC* divergen pues *AOSA Model* incide sobre todo en el diseño arquitectónico, mientras que la propuesta de Jacobson utiliza conceptos como: paquetes (para describir los casos de uso), clasificadores (para representar los aspectos) y *slices* para representar el diseño orientado a aspecto del sistema. Por otra parte, el modelo que se propone en esta memoria se apoya en la utilización de un modelo de coordinación, una definición arquitectónica de dos niveles, la definición de un LDA orientado a aspectos y dispone de la posibilidad de generación de un prototipo del sistema.

3.3.5.2. Aproximaciones al modelado OA en diseño arquitectónico

CAM (*Modelo de Componentes y Aspectos*) [Pin+02]

Es un modelo basado en componentes y aspectos que considera la separación de aspectos a nivel arquitectónico para definir la arquitectura de una aplicación en términos de componentes, aspectos y la composición entre ellos. En *CAM* se modela

independientemente el comportamiento básico de la aplicación y las propiedades extra-funcionales. De esta forma se facilita la reutilización de los componentes en otras aplicaciones y se permite mayor extensibilidad, adaptabilidad y escalabilidad. Los componentes representan la funcionalidad básica del sistema que es atravesada por aspectos (*concerns*) no funcionales, siendo los aspectos un tipo especial de componente. En *CAM* los componentes y los aspectos son entidades encapsuladas que se definen mediante una serie de interfaces. Éstas establecen una relación entre ellas y definen un conjunto de dependencias de contexto explícitas. La composición con los componentes se ejecuta interceptando los servicios que llegan o parten ellos y añadiendo el nuevo comportamiento. *CAM* no introduce un nuevo concepto arquitectónico para modelar los aspectos, sino que éstos son una visión del concepto de componente que simula el comportamiento de aspectos y la composición de aspectos y componentes. El modelo no proporciona conectores para especificar la arquitectura del sistema.

CAM actúa como base para la *plataforma de componentes y aspectos (DAOP)* [PiFuTr05] en la que los componentes se describen mediante un conjunto de interfaces. El proceso de tejido se define a través de ellas; así se preserva la reutilización de componentes y aspectos. Las interfaces se definen en XML y son parte del lenguaje *DAOP-ADL* (sección 3.4). Los componentes interaccionan entre sí mediante el envío/recepción de mensajes y de eventos. Un aspecto de coordinación determina los destinatarios de los eventos.

La arquitectura de *DAOP* (*Plataforma Dinámica Orientada a Aspectos*) se organiza en torno a un núcleo, que almacena información compartida entre todos los usuarios de una aplicación y la parte del cliente *DAOP*, en donde se ejecuta una instancia de la plataforma que gestiona la participación de un usuario dentro de la aplicación distribuida. *DAOP* es una alternativa para aplicaciones de tipo *peer-to-peer* (igual a igual), característica importante porque hay muchas plataformas que no dan soporte para dichas aplicaciones; también es útil para aplicaciones cliente-servidor. Los componentes y los aspectos instanciados se comunican entre sí a través de la plataforma sin ninguna interacción con el núcleo.

La composición, mezcla o *weaving* de componentes y aspectos se realiza dinámicamente a través de la plataforma *DAOP* en base a un conjunto de reglas que se definen externamente a los componentes y los aspectos. En el modelo también se establece el orden de composición de los aspectos en un punto de corte determinado, teniendo en cuenta que la evaluación de un grupo de aspectos puede ser concurrente o secuencial.

En la plataforma *DAOP* tanto los componentes como los aspectos se instancian como entidades de primer orden y permanecen separados incluso en tiempo de ejecución: el mecanismo de composición dinámica posterga la composición de componentes y aspectos hasta el momento en el que un aspecto intercepta el código de los componentes. Por tanto, esta propuesta es dinámica. Aunque la composición estática es más eficiente, hay situaciones en las que es necesario adaptar la arquitectura de una aplicación en tiempo de ejecución. En esos casos, la composición estática no es adecuada, pues cualquier cambio implicaría parar y recompilar la

aplicación. Por ello, en este sentido el modelo parece especialmente adecuado para desarrollar sistemas con altos requisitos de adaptabilidad dinámica. En definitiva, con este tipo de composición, se puede realizar cualquier cambio en la arquitectura de la aplicación durante la ejecución de la misma.

Los aspectos se componen con los componentes que se tratan como cajas negras, por lo que sólo tienen acceso a la interfaz pública de éstos. *CAM* permite que los aspectos puedan *atravesar* el comportamiento de los componentes antes, después o en lugar de la ejecución de los mensajes o la recepción de eventos, así como antes o después de la creación y destrucción de instancias de componentes.

La información que proporciona *CAM* durante el diseño se expresa mediante un LDA orientado a aspectos (*DAOP-ADL)* que usa esquemas XML [PiFuTr03] para describir componentes, aspectos y sus interacciones (sección 3.4), lo cual tiene varias ventajas; sin embargo, hasta el momento, XML adolece de una base formal fuerte y esto puede provocar problemas de corrección, seguridad e inconsistencia. Tiene limitaciones para validar propiedades o generación de código sin ambigüedades. *DAOP-ADL* es independiente de la tecnología y de los lenguajes de programación.

La plataforma *DAOP* está implementada en Java, lo que proporciona un *middleware* para soportar la ejecución de aspectos y componentes, y realizar un tejido dinámico entre ellos a través de Java. Por su relación con el lenguaje *AspectLEDA* propuesto en esta tesis*, DAOP-ADL* se estudia en la sección 3.4.

Ventajas e inconvenientes

La propuesta que se ha descrito en los párrafos anteriores es una de las más completas que hemos encontrado en el entorno de las arquitecturas software OA. Está constituida por un modelo de componentes propio (*CAM*), un lenguaje de descripción de arquitecturas (*DAOP-ADL*) y una plataforma de *middleware* propia que da una gran adaptabilidad al sistema, pero que no puede exportarse a otras plataformas.

Como gran aportación, presenta la posibilidad de realizar el proceso de tejido en tiempo de ejecución lo que dota al modelo de gran flexibilidad al permitir la incorporación y eliminación de componentes y/o aspectos en tiempo de ejecución. No obstante, a pesar de ser esta una ventaja, también es un inconveniente ya que el tejido dinámico reduce el rendimiento del sistema y la escalabilidad del mismo.

Un aspecto interesante del modelo es la utilización de datos compartidos entre componentes y aspectos, las denominadas *propiedades*. Esta característica es especialmente interesante en entornos colaborativos, uno de los objetivos del modelo.

Esta propuesta está relacionada con la que se presenta en el Capítulo 4 de esta tesis doctoral en que ambas:

- Inciden sobremanera en el aspecto arquitectónico del desarrollo de sistemas OA.
- Los componentes del sistema se tratan como cajas negras.
- Los aspectos son considerados como componentes.

Sin embargo, difiere de nuestra propuesta en los siguientes puntos fundamentales:

a) El lenguaje *DAOP/ADL* se basa en esquemas XML, lo que proporciona una base formal limitada, mientras que *AspectLEDA*, el LDA propuesto, dispone de una base formal más fuerte al definirse ésta en cálculo π.
b) En *CAM/DAOP* se define un aspecto que realiza las tareas de coordinación entre componentes y aspectos. *En AOSA Model* se define una estructura arquitectónica de dos niveles que facilita las tareas de coordinación de un sistema OA, al definirse un meta nivel que contiene los aspectos y otros elementos coordinadores.
c) Finalmente, *AOSA Model* no se define asociado a un modelo de componentes propio lo que en principio aumenta las posibilidades de reutilización.

PRISMA [Per+06]

En *PRISMA* se propone un modelo formal para definir arquitecturas software de sistemas complejos, distribuidos y reutilizables, e integra el DSBC y DSOA para describir modelos arquitectónicos. En este modelo, desde el principio de la especificación, todo el comportamiento se modela en aspectos pues integra las características transversales de la arquitectura como *aspectos* (distribución, coordinación, seguridad…). Así, un *aspecto* representa una propiedad o característica (*concern*) específica que atraviesa la arquitectura software y es compartido por un conjunto de componentes del sistema. Esto significa que aquellos *concerns* que no atraviesan la arquitectura no se van a considerar *aspectos*. Para evitar los *crosscutting concern*, en *PRISMA* se define un *elemento arquitectónico* formado por un conjunto de *aspectos* que lo describe en forma de diferentes *concerns* de la arquitectura. El tipo de *aspectos* que forman un *elemento arquitectónico* depende de los *concerns* del sistema que se esté especificando: i) los *aspectos* pueden ser funcionales, para especificar la computación de los puntos de enlace (*join points*) del sistema; ii) un *aspecto* especial es el de coordinación que especifica la sincronización entre los diferentes *elementos arquitectónicos* del sistema; iii) otros *aspectos* son seguridad, distribución, calidad, etc.

Los *aspectos* se definen como elementos independientes de la arquitectura, siendo los *elementos arquitectónicos* del sistema los que importan los *aspectos*. Además, así pueden ser reutilizados en cada *elemento arquitectónico* que requiera esas características. De este modo se evita el mezclado (*tangled*) del código en un *elemento arquitectónico*.

Según lo anterior, un *elemento arquitectónico* en *PRISMA* está formado por un conjunto de *aspectos* que lo describe; se puede ver desde dos puntos de vista, uno interno y otro externo. La vista interna (caja blanca) representa un *elemento arquitectónico* como un prisma, en el que cada cara es un *aspecto* de ese elemento y representa una característica del mismo. Está formado simétricamente por un conjunto de *aspectos* de distinto tipo y por las relaciones de sincronización entre los *aspectos* que conforman el *elemento arquitectónico* (que representan el tejido). La composición de diferentes *aspectos* de un *elemento arquitectónico* se define externamente a los *aspectos* para aumentar su reutilización, y se define en la especificación de cada *elemento arquitectónico*.

Desde el punto de vista externo, un *elemento arquitectónico* encapsula su funcionalidad como una caja negra y publica un conjunto de servicios que ofrece al resto de los *elementos arquitectónicos* del sistema. Éstos (siguiendo DSBC) pueden ser componentes, conectores o sistemas. Los tres están constituidos por un conjunto de *aspectos* y sus relaciones de sincronización (*weaving*).

- Un componente *PRISMA* es un *elemento arquitectónico* que captura la funcionalidad del sistema y no actúa como coordinador entre otros *elementos arquitectónicos*. Está formado por un identificador, un conjunto de *aspectos* y sus relaciones de sincronización (*aspect weaving*), y uno o más puertos, que representan los puntos de interacción con otros *elementos arquitectónicos*, que están definidos como interfaces (que publican un conjunto de servicios que pueden ser requeridos o proporcionados por el componente).
- Un conector en *PRISMA* es el *elemento arquitectónico* que actúa como coordinador entre otros *elementos arquitectónicos*. Está formado por un identificador, un conjunto de aspectos y sus relaciones de sincronización (*aspect weaving*), y uno o más puertos que representan los puntos de interacción entre los *elementos arquitectónicos,* que se definen como interfaces (que publican un conjunto de servicios que pueden ser requeridos o proporcionados por el conector).
- Un sistema es un *elemento arquitectónico* complejo que tiene asociados componentes, conectores y sistemas que se conectan entre ellos, y un conjunto de puertos que publican los servicios requeridos o proporcionados por el sistema. La definición de un sistema debe especificar las relaciones de conexión (*attachments*) y las relaciones de composición (*bindings*) entre los *elementos arquitectónicos* que contiene. 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 los pueros del sistema y los puertos de los *elementos arquitectónicos* que contiene el sistema.

Finalmente, se puede decir que el modelo da la misma relevancia a los requisitos funcionales y a los no funcionales, especificándolos del mismo modo, mediante el concepto de *aspecto*.

En *PRISMA*, la combinación de DSBC y DSOA permite tener diferentes niveles de reutilización y adaptación que mejoran los tiempos de desarrollo y de mantenimiento. Además, *PRISMA* tiene propiedades reflexivas mediante la definición de un meta nivel que permite la evolución de los *elementos arquitectónicos* en tiempo de ejecución. Por otra parte, un mecanismo permite realizar la reconfiguración dinámica del sistema ya sea de forma inducida o *ad hoc* (por invocación directa de los servicios correspondientes) o bien de forma programada al detectarse ciertas condiciones. Para este tipo de reconfiguración se incorpora un nuevo *aspecto* (*configuration aspect*) que proporciona los servicios necesarios.

PRISMA se basa en *OASIS* [Let+98] para definir de un modo formal la semántica de los modelos arquitectónicos y mantener sus ventajas; esto es, la validación y verificación de los modelos arquitectónicos y la generación automática de código a

partir del LDA. Como *PRISMA* está basado en *OASIS*, los *aspectos* se definen como atributos, servicios, precondiciones, evaluaciones, disparos y protocolos.

Uno de los objetivos de *PRISMA* es generar automáticamente el código de sus arquitecturas software OA sobre la plataforma *.NET*. Para ello, se ha desarrollado un *middleware* que permite la ejecución de arquitecturas dinámicas y orientadas a aspectos.

Ventajas e inconvenientes

PRISMA es un modelo orientado al desarrollo software basado en componentes y aspectos desde un punto de vista arquitectónico, siendo ésta una de sus principales características. El modelo propone la definición del concepto de *elemento arquitectónico* constituido por componentes y *aspectos* en un modelo de estructura simétrica. También se propone un modelo de componentes específico que no puede integrarse con modelos de componentes genéricos, lo que limita su aplicabilidad. Por otra parte, los *aspectos* son independientes del sistema en el que se utilizan lo que promueve la reutilización de los mismos. Está dotado de un LDA y una herramienta que facilitan al arquitecto de software el desarrollo de los sistemas basados en este modelo.

PRISMA se relaciona con *AOSA Model* en que ambas propuestas definen un modelo arquitectónico, un LDA y una herramienta para desarrollar sistemas OA. Sin embargo, ambas afrontan la resolución del problema desde perspectivas completamente diferentes:

a) *PRISMA* define un modelo simétrico, mientras que *AOSA Model* sigue una aproximación asimétrica.
b) *PRISMA* define un modelo de componentes propio, difícilmente integrable con otros modelos. Por el contrario, *AOSA Model* no define ningún modelo de componentes propio, sino que se basa en la especificación genérica del modelo de componentes de UML 2.0.
c) *PRISMA* se ha definido orientado a la especificación y el diseño de sistemas desde el principio de su proceso de desarrollo, si bien también *PRISMA* puede aplicarse en tiempo de evolución (para sistemas desarrollados bajo este modelo). *AOSA Model,* por su parte, está definido para facilitar la evolución de los sistemas, permitiendo la incorporación de nuevos requisitos que se puedan considerar como aspectos. Esto no es óbice para que *AOSA Model* también permita el diseño de sistemas OA desde su inicio; para ello, se propone un proceso iterativo, en el que primero se especifican los requisitos base del sistema y luego, en una iteración posterior, se incorporan los requisitos que se hayan extraído de los iniciales y se puedan considerar aspectos.

AO-Rapide [Pal04]

AO-Rapide es un modelo que permite expresar arquitecturas orientadas a aspectos centrándose en la especificación formal de las interacciones entre ellos. La autora es K. Palma [Pal04] y su desarrollo pretende ser una solución al problema de la separación de los intereses transversales (*crosscutting concerns*). Éste se considera como un problema de coordinación, que puede resolverse a nivel de arquitectura software describiendo el sistema mediante un LDA y el uso de modelos de coordinación [Nav+02]. En el caso de *AO-Rapide* [PaEtMu05] el lenguaje es *Rapide* y modelo de coordinación *Reo*[13] [Arb04] que permite especificar la coordinación de los componentes en entidades separadas de esos componentes. Se trata de un modelo en desarrollo, por lo que no se dispone de mucha información al respecto.

En *AO-Rapide* los aspectos se definen como componentes. Para llevar a cabo la comunicación entre componentes del sistema y los aspectos, observando el principio de inconsciencia [FiFr00], el modelo propone la definición de unos elementos que envuelven a cada componente funcional afectado por un aspecto. La misión de un envolvente (*wrapper*) es comunicar a los conectores la ocurrencia de eventos. Estos elementos permiten la definición de una arquitectura orientada a aspectos de un modo transparente a los componentes funcionales, así como incorporar o eliminar aspectos sin impacto sobre dichos componentes funcionales.

El modelo define un tipo especial de conectores que son los encargados de establecer las reglas de coordinación que determinan si la descripción de la funcionalidad y la de los aspectos debe entrelazarse de forma síncrona o asíncrona para obtener el comportamiento deseado, y determinan también cuándo debe realizarse el entrelazado. Se definen además reglas de coordinación entre aspectos, que son necesarias cuando dichos aspectos están asociados al mismo punto de unión o punto de corte, y hay dependencias entre ellos.

Los conectores se componen de tres partes:

a) El *coordinador:* es un conector *Reo* y permite coordinar la ejecución de los componentes a través de las operaciones de entrada/salida sobre conectores. Se especifica como una *interface Rapide*.
b) El *enlace* o *wrapper linker* es el enlace entre el envolvente (*wrapper*) del componente funcional y el coordinador que solicita los *join points* correspondientes del *wrapper* y ejecuta operaciones de entrada/salida en el coordinador. El *wrapper linker* conoce: el *wrapper* del que debe solicitar los *join*

[13] *Reo* es un modelo de coordinación exógeno basado en canales, donde coordinadores complejos llamados conectores se construyen a través de la composición de otros más simples. Cada conector en *Reo* impone un patrón de coordinación específico sobre los componentes que realizan operaciones de E/S a través de él, sin el conocimiento de estos componentes. Este modelo permite definir un conjunto de operaciones para especificar los protocolos de coordinación a través de un conjunto de canales con un comportamiento bien definido.

points, los *join points*, el coordinador sobre el que debe actuar y las operaciones asociadas a los *join points*. Siempre debe haber un *wrapper linker* en cada conexión entre un coordinador y el envolvente de un componente.

c) El *aspect linker* es un enlace entre el coordinador y el componente de aspecto. Contiene información que identifica al coordinador y al aspecto asociados, así como los eventos que se lanzan para llamar a los servicios del aspecto; por tanto, debe haber un enlace de este tipo por cada conexión entre un coordinador y un aspecto. Su función es traducir las operaciones de entrada/salida ejecutadas en el coordinador a las correspondientes llamadas a los servicios del aspecto. Este enlace permite sustituir el aspecto correspondiente, siempre que el nuevo siga el mismo patrón de coordinación.

Esta división de los conectores proporciona al diseñador el uso de coordinadores genéricos que encapsulen patrones de coordinación OA de uso frecuente, y que no se refieran a ningún componente en concreto; también permite la reutilización de los coordinadores. En Figura 3.2, tomada de [PaEtMu05], se muestra la conexión entre un aspecto y un componente. En el esquema se observa el envolvente del componente y la división del conector en tres partes. Para un cierto sistema, del que se han identificado sus componentes funcionales y los aspectos que lo constituyen, su arquitectura se especifica del siguiente modo:

- Cada componente funcional y de aspecto se especifica como un componente (un componente *Rapide*).
- Se identifican aquellos componentes funcionales a los que se asocie algún aspecto.
- Para cada uno de ellos, se determinan los *join point* específicos a los que se asocia el o los componentes de aspecto.
- Se determinan las reglas de composición (coordinación) que gobernarán las interacciones entre un punto de enlace (*join point*) y el componente de aspecto asociado a él.
- A partir de las especificaciones *Reo* y las reglas de composición, se construyen los *componentes coordinadores* que implementan las reglas de coordinación.
- Se conecta cada *wrapper* de un componente funcional a su coordinador, a través del correspondiente *wrapper linker*.
- Cada coordinador se conecta al aspecto asociado mediante su *aspect linker.*

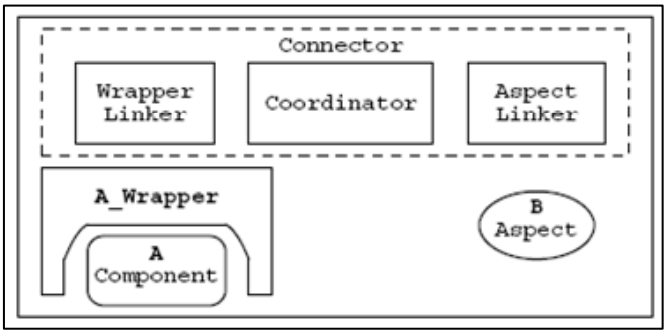


Figura 3.2. Arquitectura AO-Rapide.

Ventajas e inconvenientes

AO-Rapide es un modelo en desarrollo que se basa en la definición de aspectos como componentes, que se integran en un sistema mediante la definición de conectores. Éstos llevan a cabo la coordinación entre componentes y aspectos de un modo transparente a aquellos. Las labores de coordinación se realizan siguiendo la filosofía que propone un modelo de coordinación (*Reo*). *AO-Rapide* está fuertemente relacionado con la estructura del LDA en el que se basa su definición. La principal desventaja de este modelo es que, al menos a día de hoy, no dispone de ninguna herramienta que asista al arquitecto de software en las tareas del diseño arquitectónico de los sistemas.

En relación con el modelo que se presenta en esta memoria, se puede decir que, si bien ambos trabajan sobre la idea de que se separación de aspectos se puede tratar como un problema de coordinación, ambas abordan el tema de distinto modo:

a) *AO-Rapide* trabaja sobre la base que le proporciona el modelo de coordinación *Reo*, mientras que *AOSA Model* aprovecha las ventajas de otro modelo de coordinación, *Coordinated Roles*.
b) *AO-Rapide*, se apoya en las características del LDA *Rapide*, tanto para definir el modelo como el lenguaje OA que propone, lo que limita su aplicablidad, a pesar de las ventajas que tiene el lenguaje para la descripción arquitectónica de los sistemas. Por el contrario, *AOSA Model*, en su descripción, es independiente de ningún LDA, si bien se ha definido un lenguaje OA que responde a las características que el modelo requiere.
c) *AO-Rapide* no dispone de ninguna herramienta, al contrario que *AOSA Model*.

TranSAT Method

TranSAT [BaDuMe05, Bar+06], *Transformation Software Architecture Technology,* es, según sus autores, un marco de trabajo o *framework* que permite gestionar la evolución de las arquitecturas software aplicando los principios del DSOA en un contexto arquitectónico. Se ha desarrollado para reducir la complejidad de la integración de nuevas propiedades (*concerns*) en una arquitectura software. Para ello, Barais et al. proponen la definición de un *framework* para la gestión de la evolución de las especificaciones de las arquitecturas software, de modo que trata el ciclo de desarrollo software como una evolución guiada por la modificación constante y gradual de la arquitectura. La arquitectura software se describe paso a paso, de modo que los aspectos no se consideran desde el principio, sino como extensiones que se van incorporando uno a uno a las funcionalidades existentes, manteniéndose la integridad del sistema y considerando los componentes como cajas negras. Desde una arquitectura inicial (*core architecture*), que contiene sólo características de negocio (*business concerns)*, *TranSAT* refina las especificaciones

arquitectónicas añadiendo incrementalmente las características técnicas (*technical concerns),* como persistencia, seguridad o gestión de las transacciones, y nuevas características de negocio, para llegar a la arquitectura final. La integración de las características técnicas actúa sobre la especificación de la arquitectura realizando cambios arquitectónicos, estructurales y de comportamiento, apoyándose en conceptos de DSOA.

TranSAT aísla la descripción de la funcionalidad de cada *concern* en una estructura arquitectónica independiente, un patrón (*software pattern*), que se integra automáticamente en la arquitectura inicial (*core*) mediante un mezclador (*weaver*). *Integrar* un nuevo elemento en una arquitectura existente supone definir cómo son las nuevas interacciones necesarias. Un patrón (*software pattern*) es un fragmento arquitectónico que contiene una descripción de dónde se aplicará una secuencia de reglas de transformación y contiene toda la información necesaria (condiciones) para integrar el *concern* en la arquitectura. Se puede desarrollar independientemente de cualquier arquitectura específica y reutilizarse en diferentes arquitecturas. Un patrón está formado por un *nuevo diseño* (*new plan* o fragmento arquitectónico que debe integrarse), una máscara de puntos de unión (*join points mask*) y un conjunto de reglas de transformación que describen las modificaciones a realizar sobre la arquitectura inicial:

- El *nuevo diseño* (*new plan*) es un conjunto de componentes que describen la estructura y el comportamiento del nuevo *concern*. Los puntos de comunicación de este conjunto pueden no definirse inicialmente. Éstos serán los puntos a los que se asocien los conectores para llevar a cabo la integración en la arquitectura original. El *new plan* se describe en el LDA.
- La máscara de puntos de enlace (*join point mask*) define los requisitos estructurales y de comportamiento que la nueva arquitectura debe cumplir para llevar a cabo la integración.
- Las reglas de transformación especifican cómo llevar a cabo la integración. Para asegurar que ésta es correcta, *TranSAT* introduce un conjunto de restricciones en un *lenguaje de transformación* (que proporciona abstracciones para realizar las actualizaciones de un modo consistente), así como verificaciones estáticas y dinámicas. *TranSAT* simula estáticamente la ejecución de las reglas de transformación sobre varios elementos de la arquitectura. Al final de la verificación estática tiene que haber sido probado que cada puerto de la interfaz de un componente tiene, al menos, una operación. Para completar el análisis, *TranSAT* realiza una verificación dinámica a fin de asegurar que el comportamiento de los componentes de la nueva arquitectura se sincroniza adecuadamente tras la inclusión de los nuevos elementos. Así, el *lenguaje de transformación* y las verificaciones estáticas de *TranSAT* aseguran propiedades *a priori*, antes de que la arquitectura inicial sea modificada. Otras propiedades se verifican dinámicamente durante el proceso de transformación. Finalmente, si se produjeran errores, *TranSAT* facilita el proceso de deshacer los pasos dados.

Al llevar a cabo la integración de un nuevo *concern* en una arquitectura existente pueden surgir problemas de consistencia. Sin embargo, el marco (*framework*) definido por *TranSAT* asegura que los nuevos elementos introducidos son

consistentes con la arquitectura en la que se insertan. Para ello, los autores construyeron un modelo arquitectónico denominado *SafArchie* [BaDu05] que proporciona un modelo jerárquico de componentes de la arquitectura inicial y las relaciones entre ellos, donde un componente viene definido por una interfaz estructural y su especificación de comportamiento externo; las interfaces son un conjunto de operaciones representadas en puertos (*ports*); la especificación del comportamiento externo identifica las trazas de los mensajes que los componentes envían o reciben. Las especificaciones arquitectónicas de *SafArchie* pueden ser analizadas y transformadas en especificaciones *Fractal* o *ArchJava*. Además, se añade una especificación de comportamiento externo usando un subconjunto de *FSP*, que se usa en *Darwin* para analizar algunas propiedades de composición. El uso de un LDA garantiza una definición formal de la arquitectura y su independencia de la plataforma.

Los conceptos de modularidad en la aproximación de separación de aspectos supone que: i) los *concerns* deben definirse independientemente de la arquitectura en la que vayan a emplearse; ii) en general, las características técnicas (*technical concerns*) son transversales a los componentes que describen las características de negocio; iii) las reglas de integración de una característica (*concern*) en la arquitectura deben especificarse cuidadosamente. Para abordar estos principios, el modelo de componentes definido por *TranSAT* incorpora dos nuevos elementos:

- Un adaptador (*adapter*) que define las reglas de integración de los componentes técnicos sobre una arquitectura genérica y es independiente del contexto. Se definen tres tipos de reglas de integración: reglas estructurales, de comportamiento y de transformación arquitectónica (éstas actualizan las interacciones entre los componentes). Las reglas se definen a partir de las etiquetas asociadas a los puntos de corte.
- Un mezclador (*weaver*) que describe la integración entre una arquitectura específica y un *concern* técnico. El *weaver* identifica el lugar donde la arquitectura tiene un punto de corte, para definir la interacción.

Los aspectos se modelan como componentes, y el tejido (*weaving*) se define utilizando los adaptadores y los *weavers* (mezcladores o tejedores), que permiten definir respectivamente las reglas de integración y de coordinación.

Para especificar la evolución del software primero hay que seleccionar un aspecto que represente la evolución que se desea integrar en la arquitectura inicial y su adaptador, identificando los *puntos de enlace* en el centro (*core*) de la arquitectura. Además, hay que definir las etiquetas de *puntos de corte* (*pointcut label*) en la arquitectura objetivo, y hay que dar alguna información del contexto que necesita el adaptador para llevar a cabo la transformación: definición de los *puntos de corte* en los adaptadores y tejedores. Finalmente, se relacionan las definiciones de *puntos de corte* con las etiquetas. Estas relaciones representan el tejido entre el conjunto de reglas de transformación y la arquitectura software. Para incorporar las *características técnicas*, que representan la evolución de la arquitectura, se realizan varios pasos, cada uno de los cuales lo lleva a cabo un *tejedor* determinado, teniendo que especificarse manualmente el orden en le que se incluyen.

Un servicio de una *característica técnica* se define a través de un conjunto de componentes y sus interacciones; esta integración modifica la arquitectura en construcción. La integración en *TranSAT* se hace declarando primero unas reglas de interacción independientes del contexto y después configurándolas específicamente para la arquitectura final concreta.

TranSAT utiliza una herramienta: *SafArchie Studio* que es un conjunto de extensiones para *Argo UML*. La relación Aspecto-Componente se lleva a cabo mediante un conector UML; de este modo, se pueden definir las especificaciones de composición y su semántica.

A alto nivel (nivel de arquitectura) soporta la aproximación simétrica, pero a bajo nivel soporta un modelo asimétrico de diseño OA.

Ventajas e inconvenientes

Es un *framework* definido para facilitar la evolución de las arquitecturas software, lo que supone tanto una ventaja como un inconveniente: es una ventaja en tanto en cuanto, partiendo de una arquitectura inicial, en la que no se consideran los aspectos, éstos se van incorporando uno a uno. Precisamente esta es una desventaja de esta aproximación, pues sólo se puede aplicar sobre sistemas ya existentes que sufren alguna modificación a lo largo de su vida y los aspectos sólo se consideran en la fase de evolución, no desde el principio del desarrollo.

Otras aportaciones de *TranSAT* son: proporciona escalabilidad mediante la definición de patrones arquitectónicos, potenciando la reutilización de los aspectos; la integración de nuevos requisitos preserva la consistencia de la arquitectura original, manteniéndose la separación de los aspectos en la arquitectura extendida; se mantiene la trazabilidad de los elementos del sistema a lo largo del proceso; y el LDA que se define dota a la arquitectura de formalidad y de independencia de cualquier tecnología.

Este modelo es semejante al propuesto en el Capítulo 4 de esta memoria:

a) Ambos facilitan la evolución de los sistemas permitiendo la integración de nuevos requisitos una vez que el sistema ya ha sido creado, aunque *AOSA Model* también proporciona un procedimiento para que los sistemas se puedan definir desde el principio, extrayendo los requisitos que se puedan considera como aspectos y siguiendo un proceso iterativo de desarrollo del mismo.
b) Los aspectos se definen como componentes y se mantienen separados de los componentes del sistema a lo largo del desarrollo.
c) Ambos disponen de un LDA dotado de formalidad suficiente.
d) Disponen de herramientas para facilitar la tarea del arquitecto de software.

Sin embargo, difieren en el modo en el que se componen componentes y aspectos: *AOSA Model* considera un meta nivel que contiene los elementos necesarios para realizar esta tarea y se apoya en un modelo de coordinación para llevarla a cabo,

mientras que el proceso de composición en *TranSAT* es diferente, según se ha explicado.

FAC Model

FAC [Pes+06] es una extensión de *Fractal* [PeSeDu04] que integra conceptos de programación basada en componentes y de orientación a aspectos a nivel arquitectónico. La extensión consiste en superponer sobre un conjunto inicial de componentes un nivel asociado a un conjunto de elementos que corresponde al dominio del *código transversal*. Para ello, se definen unas nuevas relaciones o ligaduras (*binding*) que permiten llevarla a cabo. Se necesitan dos elementos nuevos para definir la extensión: un elemento que representa un componente que contenga la definición y el comportamiento de las *propiedades transversales,* y un mecanismo de *tejido* que mezcle dichas propiedades con la arquitectura original:

- El elemento que representa la definición de una propiedad transversal (*crosscutting concern*) es el que representa la definición del concepto de componente de aspecto (*Aspect Component*). Éste está compuesto, a su vez, por dos elementos: uno que implementa la intercepción asociada al *concern* (*punto de corte*); el otro es el *concern* propiamente dicho (implementa el *advice*[14]). Los primeros son los que implementan los métodos interceptores (*methods calls)*.
- Mecanismo de *tejido*: en *FAC* no se hace distinción entre el comportamiento de un aspecto y de un componente regular del sistema; por ello, el comportamiento de los aspectos se implementa como elementos regulares del sistema (que proporciona al menos una interfaz), que se tejen con los componentes mediante un nuevo mecanismo de composición: *conector de aspecto*. Está compuesto por:
  a) Una ligadura de aspecto (*aspect binding*) que conecta componentes base con un componente de aspecto (los componentes regulares exponen un conjunto de *puntos de enlace* a los cuales se asocian los aspectos). El modelo de *puntos de enlace* de *FAC* se centra en las llamadas entrantes y salientes sobre las operaciones de las interfaces cliente y servidor.
  b) Una *interfaz de tejido* (*weaving interface*), es una interfaz de control *Fractal* que se define para la gestión de los *puntos de enlace.*
  c) Un componente *aspectizable,* que es un componente que soporta la *interfaz de tejido*. Este componente contiene los *advice* de los aspectos.
  d) Una *interfaz de advice* que es la interfaz sobre la cual se asocia un *aspect binding*; puede ser de tipo *before*, *after* o *around*.
  e) Un componente de aspecto que es un componente *Fractal* que soporta la *interfaz de tejido.*
  f) Un dominio de aspecto (*aspect domain*) es un componente *Fractal* que *reifica* el dominio de un componente de aspecto.

Los componentes de aspecto se tejen y destejen en tiempo de ejecución de un modo semejante a como lo hacen los componentes regulares. Además, este modelo

[14] Se mantiene *advice* por considerarse más adecuado que su traducción *aviso*.

mantiene la idea de *Fractal* por la que se considera que un componente consta de una parte funcional y una parte controladora. Se centra en la intercepción de las interfaces funcionales y proporciona un lenguaje de arquitectura basado en XML para describir arquitecturas con *características transversales* (*crosscutting concerns),* implementándose finalmente los componentes en Java.

La metodología de *FAC* se basa en tres pasos:

a) Identificar y diseñar los *crosscutting concerns* como un conjunto de componentes.
b) Escribir los componentes de aspecto para integrar los *crosscutting concerns.*
c) Combinar o tejer los componentes de aspecto usando *Fractal ADL*.

Ventajas e inconvenientes

FAC es un *framework* basado en el modelo de componentes de *Fractal* y proporciona un LDA para describir arquitecturas software considerando los *crosscutting concern*. Añadir un nuevo comportamiento en *FAC* consiste en considerarlo como un componente de aspecto (*aspectual component*) y definir dónde hay que aplicar el nuevo comportamiento: definir los *puntos de corte*, que puede ser estructural o de comportamiento. El comportamiento estructural se puede usar como si fuera un *punto de enlace* del sistema, y cada operación de un componente (en una relación normal de tipo proporciona-requiere) puede ser interceptada por las nuevas características. Así, en *FAC* se permite que un aspecto se dispare sobre una secuencia de interacciones externas al componente: el aspecto captura la interacción de cada componente y el aspecto se dispara si la secuencia de interacción coincide con su definición de interacción.

FAC y *AOSA Model* son modelos que presentan algunas similitudes:

a) Ambos consideran los aspectos como componentes, que describen su funcionalidad, y un mecanismo externo a ellos que permite llevar a cabo la composición con los elementos del sistema en el que se insertan.
b) Ambos modelos definen un nivel de aspecto que contiene toda la información relativa a ellos y a su interacción.
c) Los lenguajes asociados a ambas generan código Java.

Sin embargo difieren en el modo en el que se definen los elementos relativos a la interacción de los aspectos (*AOSA Model* se apoya en un modelo de coordinación) y en que el lenguaje asociado a cada modelo tiene una base formal diferente: el LDA asociado a *FAC* está basado en esquemas XML, mientras que *AspectLEDA* tiene una base formal más fuerte.

3.3.6. Conclusiones sobre el modelado de aspectos

A lo largo de esta sección se han mostrado los modelos y aproximaciones al desarrollo de sistemas OA que se han considerado más relevantes. De los modelos estudiados, se puede destacar lo siguiente:

1) En cuanto a la simetría, *PRISMA* es un modelo simétrico, lo que hace que el modelo de componentes que propone no pueda integrarse en otros modelos genéricos. El resto de los modelos considerados son asimétricos.
2) En relación con los LDA asociados a los modelos, tanto *CAM/DAOP* como *PRISMA* y *FAC* definen lenguajes propios, si bien en ninguno de ellos se introducen nuevas abstracciones para soportar aspectos, sino que se extienden conceptos existentes. Sin embargo, consideramos que en la actualidad existe un amplio número de LDA, con diferentes características de dinamismo y formalidad, que pueden utilizarse para, tras las extensiones correspondientes, describir arquitecturas OA, como es el caso de *AO-Rapide*.
3) En lo que se refiere a herramientas, todos, excepto *AO-Rapide*, disponen de alguna que, en diferente medida, facilita las tareas de la descripción arquitectónica.
4) Atendiendo a la facilidad de evolución, prácticamente todos los autores indican la posibilidad de hacer evolucionar los sistemas desarrollados bajo sus respectivos modelos. Especial mención hay que hacer en este apartado de *TranSAT,* que se define con este objetivo, para sistemas ya existentes.
5) En cuanto al dinamismo, todos presentan cotas de dinamismo en distinto grado.

Como resumen se ha realizado una tabla comparativa de los modelos arquitectónicos estudiados (Tabla 3.1). Se ha considerado oportuno incluir *AOSA Model* en la última fila de la tabla, aunque el modelo aún no ha sido descrito. Las características que se presentan en la tabla, a modo de criterios de comparación se refieren:

1) A la simetría o asimetría de los aspectos al integrarse en la propuesta.
2) A la inclusión o no de conectores para modelar la arquitectura.
3) Al concepto arquitectónico que se utiliza para representar los aspectos.
4) Al proceso de *weaving* o mezclado de aspectos y componentes del sistema. Hay que considerar si este proceso es externo o interno a los aspectos, y si es estático o dinámico.
5) Al LDA asociado al modelo, si lo tiene, y sus características en cuanto a formalidad. De ella depende la posibilidad de verificar las propiedades de la arquitectura.
6) A la facilidad de evolución de la arquitectura mediante la inclusión de nuevos aspectos o su eliminación.
7) A la posibilidad de utilizar herramientas que asistan al arquitecto durante la definición de la arquitectura.
8) A la tecnología de la que depende.
9) Al uso de modelos de coordinación.

Tras el estudio realizado se echa en falta un modelo arquitectónico que se apoye en un lenguaje formal (como los modelos 2, 3, y 4 en la tabla 3.1) que extienda uno de los LDA ya existentes (como 3) y que siga un modelo asimétrico (como 1, 3, 4, y 5), en el que los aspectos se consideren como componentes (modelos 1, 2, 3, 4, y 5) y la interacción se defina externa a ellos para incrementar su reutilización (1, 2, 3, 4, y 5).

Además, como se propone en esta tesis doctoral, consideramos de especial interés tratar el problema de la separación de aspectos como uno de coordinación (como hace 3) que además genere un prototipo ejecutable y disponga de las herramientas necesarias (-). Es por ello que se define *AOSA Space*, cuyas características se pueden resumir en el siguiente párrafo y cumple con los requisitos expresados:

> *AOSA Space* es un marco de trabajo que consta de un modelo arquitectónico [NaPeMu05a], una metodología de trabajo y un LDA-OA que extiende un LDA convencional [NaPeMu08], y permite definir la arquitectura de sistemas OA. Los aspectos se consideran componentes regulares y su composición e integración se lleva a cabo externamente de ellos a aplicando un modelo de coordinación. Una herramienta, asociada al LDA definido, facilita el proceso de traducción del modelo AO a una arquitectura convencional y la ejecución de un prototipo. Además, facilita la evolución de los sistemas incorporando los nuevos requisitos como aspectos que se coordinan con la arquitectura base. Finalmente el modelo diseñado es independiente de la plataforma.
>
> Este marco de trabajo, objetivo de esta tesis doctoral, se estudia detenidamente en los capítulos siguientes.

Modelos OA	*Etapa*	*Modelo simétrico/ asimétrico*	*Definición de conectores*	*Representación de aspectos*	*Proceso weaving*	*LDA asociado*	*Evolución de la arquitectura*	*Herramienta*	*Tecnología de la que depende*	*Modelos de Coordinación*
Use Cases	Ing de Requisitos	Asimétrico	Sin conectores	Casos de Uso/Slice			Incorporación de U.C. Extension		Independiente	
(1) CAM/ DAOP	Arquitectura del software	Asimétrico	Con conectores	Como componente	Externo. Mediante interfaces	DAOP/ADL. Basado en XML	Weaving dinámico pero no inclusión de aspectos en runtime	De validación y modelado DAOP-ADTool	Plataforma DAOP, sistemas p2p	
(2) Prisma	Arquitectura del software	Simétrico	Con conectores	En los elementos arquitectónicos	Externo a los aspectos	Prisma ADL. Lenguaje propio	En tiempo de ejecución. Aspecto de configuración	Prisma Modeling EVG Prisma	PrismaNet	
(3) AO-Rapide	Arquitectura del software	Asimétrico	Sin conectores	Como componente	Externo. Mediante coordinador	AO-Rapide. Basado en Rapide	Weaving dinámico	No existen	Independiente	Reo
(4) TranSAT	Arquitectura del software	Asimétrico a alto nivel. Simétrico a bajo nivel	Sin conectores	Como componente	Externo. Usa weavers o adapters	SafArchie. Lenguaje propio	Fw para gestionar la evolución arquitectónica	SafArchie Studio	Independiente	
(5) Fac	Arquitectura del software	Asimétrico	Sin conectores	Como componente	Externo. Usa weavers o adapters	Extiende Fractal ADL Basado en XML	Weaving dinámico	--	Independiente	
AOSA Model	Arquitectura del software	Asimétrico	Sin conectores	Como componente	Externo. Mediante coordinador	AspectLEDA, extiende LEDA	Weaving dinámico	AOSA Tool/LEDA	Independiente	Coordinated Roles

Tabla 3.1. Tabla comparativa de los modelos de desarrollo OA.

3.4. Lenguajes de descripción arquitectónica orientados a aspectos

Dentro del paradigma de DSOA parece lógico utilizar también lenguajes de descripción arquitectónica para capturar las características de las arquitecturas software. Sin embargo, habría que determinar si los LDA existentes son adecuados para expresar las características de las *propiedades transversales* (*crosscutting concerns)* a nivel de arquitectura o bien si es necesario añadir nuevos elementos que permitan capturar las nuevas características. La integración de los LDA y DSOA puede realizarse si, de algún modo, se añaden los nuevos conceptos del paradigma a los lenguajes, ya sea como componentes o conectores aspectuales, entre otras posibilidades, ya que los *crosscutting concerns* no pueden expresarse adecuadamente en los LDA convencionales. Hasta hoy, se han propuesto algunos LDA orientados a aspectos (LDA-OA) que extienden LDA existentes integrando en ellos las abstracciones de orientación a aspectos (*AO-Rapide*, *AspectualAcme*, o *AspectLEDA*); otros (*PRISMA-ADL*, *DAOP-ADL*, *FAC-ADL*,…) se han desarrollado empleando las abstracciones de OA desde su concepción y a partir de ciertas características o modelos iniciales. Los LDA-OA varían en el modo en el que los *crosscutting concerns* se representan como elementos del lenguaje.

En [Na+02] propusimos un conjunto de requisitos que deberían cumplir los LDA para permitir la gestión de los *crosscutting concerns* usando abstracciones arquitectónicas. Estos requisitos son:

- *Especificar los componentes funcionales con sus interfaces y la interconexión entre ellas. Esto ya lo proporcionan los LDA convencionales, sin embargo también es necesario que los LDA (para representar las propiedades transversales) puedan especificar los puntos de enlace (join points) en esos componentes funcionales. De este modo, debería definirse un conjunto de nuevas primitivas para gestionar la especificación de los puntos de enlace. Estas primitivas soportarían la especificación de todos sus tipos, no estando restringido a un conjunto predefinido de ellos.*
- *Especificar los aspectos. La morfología de éstos es diferente de la de los componentes funcionales en el sentido que ellos no proporcionan servicios ni interfaces. Sin embargo, para llevar a cabo una modularización de los aspectos, éstos deberían especificarse como un tipo especial de componente, descritos mediante nuevas primitivas del lenguaje.*
- *Finalmente, especificar las conexiones entre los puntos de enlace y los aspectos.*

Por otra parte, en [Bat+06a] se presentaron 7 características relativas a la integración de DSOA y los LDA que posteriormente se recogieron en [Bat+06b] bajo un *framework* para la evaluación de los LDA-OA. En ese trabajo se discute cómo y por qué es necesario extender o no los elementos de interconexión de los lenguajes. Se concluye que *los LDA promueven la separación de aspectos al separar los componentes de las interacciones (que gestionan los conectores)*, lo que por otra parte ya se había comentado

en el apartado 3.3.2, al mencionar la relación general que existe entre el DSOA y la arquitectura del software. Según los autores del estudio, *las extensiones, que parece necesario realizar para utilizar los LDA convencionales en el contexto del DSOA, pueden referirse a la definición de conectores o componentes aspectuales*. Las siete características mencionadas, que relacionan los conceptos de los *crosscutting concerns* y abstracciones de conexión arquitectónica, se pueden resumir del siguiente modo:

- *Elemento base*. Describe qué elemento arquitectónico (en una descripción arquitectónica) puede contener las *propiedades transversales* (*crosscutting concerns*).
- *Composición aspectual.* Se refiere a las características de la composición aspectual que debe soportar un LDA-OA, para permitir la composición entre los elementos base y los aspectos. Aunque no hay consenso en cómo modelar la composición aspectual, hay algunos trabajos que se refieren al uso de conectores para ello [KaSt01, Cue+04b, Bat+06b]. Otros lenguajes consideran la composición de diferentes modos, como queda reflejado en el apartado correspondiente a cada uno de los estudiados.
- *Quantification*[15]. Se refiere a que un LDA-OA puede o no soportar mecanismos de *quantification*[16] (condicionamiento) sobre los *puntos de enlace* (*join points*). Este mecanismo propone que, para evitar referirse a cada *join points* explícitamente en la descripción arquitectónica, haya una única instrucción que permita localizar varios. Este mecanismo se puede definir en la parte de configuración. Si los LDA soportan la *quantification*, esta característica define cómo y cuándo se especifica.
- *Interfaces de aspecto.* Se refiere a la especificación de los interfaces de los aspectos en el LDA-OA.
- *Exposición de los join points*. Un LDA-OA debe soportar la exposición de los *join points* como *puntos de enlace arquitectónicos*. En [Bat+06a] se considera que el concepto general de interfaz lo proporcionan los LDA con suficiente expresividad y no necesita ser extendido.
- *Ampliación de las interfaces*. Esta característica se refiere a la ampliación de la interfaz de los componentes con nuevos elementos como servicios o atributos. Un LDA-OA puede soportar esta característica o no. Según Batista ésta no es una característica necesaria de los LDA-OA para capturar las *propiedades transversales* a nivel de especificación arquitectónica.
- *Descripción y representación de los aspectos*. Un LDA-OA debe soportar la descripción de estos elementos. Esta característica se refiere a si en los lenguajes se propone la definición de una nueva abstracción que represente a los aspectos a nivel de arquitectura. Piénsese que los aspectos representan "trozos" de funcionalidad que con los LDA convencionales quedaría disperso en los componentes regulares

[15] Se mantiene el término *quantification* en lugar de usar su traducción *condicionamiento* o *imposición de condiciones*. *Quantified statements* se traduce por sentencias condicionadas.

[16] La *quantification* se refiere al hecho de que algunas unidades software afectan a otras unidades software de un sistema cuyo código extienden o sustituyen. Significa que se pueden definir declaraciones (*statements*) unitarias y separadas en diversos lugares, no locales, de un sistema. La *quantification* puede ser estática o dinámica [FiFr00]. La *inconsciencia* (*obliviousness*) se refiere a que los lugares en los que se aplica la *quantification* no tiene que estar especialmente preparados para recibirlas (no tienen conocimiento de ellas); es un invocación implicita a esos lugares.

De esta propuesta podemos concluir, de acuerdo con Batista, que de los LDA-OA estudiados se deduce que no se necesitan nuevas abstracciones arquitectónicas para representar aspectos, si bien sí es necesario ampliar o modificar la definición de algunos elementos arquitectónicos ya existentes como conectores (a *conectores aspectuales*) o componentes (a *componentes aspectuales*), y extender el concepto de composición en esos LDA.

En los próximos aparatados se describen varios LDA orientados a aspectos (*DAOP-ADL, PRISMA ADL*, *AO-Rapide, AspectualAcme*, *FAC-ADL* y Pilar-OA), en su mayoría asociados a modelos de diseño arquitectónico (descritos en la sección anterior). De su estudio se puede afirmar que las características de los lenguajes están muy relacionadas con las de los modelos a los que están asociados. Esto nos ha llevado a conclusión de que, para expresar formalmente el modelo que se propone en el Capítulo 4, lo más adecuado era definir un lenguaje que se adapte a sus requisitos, pues ninguno de los estudiados cumplía con ellos. En particular, se iban buscando características como:

- Definición de aspectos como componentes.
- Utilización de modelos de coordinación para gestionar la integración de los aspectos.
- Posibilidad de generar código.
- Ejecución de un prototipo.

Consecuencia de esto es la definición de *AspectLEDA*, descrito en el Capítulo 6 de esta memoria.

3.4.1. DAOP-ADL

Este lenguaje fue desarrollado por Pinto y su grupo de trabajo [PiFuTr03] como complemento al modelo *CAM* (*Component Aspect Model*) y la plataforma *DAOP* (*Plataforma Dinámica Orientada a Aspecto*) [PiFuTr05]. *DAOP-ADL* es un LDA que permite describir arquitecturas software basadas en componentes y aspectos cuando se sigue el modelo *CAM*. Posteriormente, *DAOP-ADL* debe ser interpretado por la plataforma *DAOP*.

Es un LDA dinámico que permite analizar y verificar la corrección del diseño, así como evaluar diferentes posibilidades de diseño, incluso antes de la implementación. Existen dos niveles de validación:

- Utilizando un procesador XML (*parser*) para comprobar si el documento XML es válido y está bien formado según el esquema de definición de *DAOP-ADL*[17].
- Hay ciertas restricciones que deben cumplir las aplicaciones y que no pueden expresarse directamente con esquemas XML. Son las que constituyen el segundo nivel de validación. Ésta supone validar las descripciones de componentes y aspectos, las reglas de composición de los componentes y las reglas de evaluación de los aspectos.

[17] Un documento está bien formado si es correcto con respecto a la estructura y las restricciones de XML, y es válido si es correcto respecto a un esquema XML que dicho documento tenga asociado.

Para evitar el desacoplamiento que tiene la mayoría de LDA con respecto a los lenguajes de implementación (lo que causa problemas de análisis, implementación, evolución, etc.), uno de los objetivos de *DAOP-ADL* es disponer de la descripción de la arquitectura en tiempo de ejecución. Para ello, el lenguaje utiliza XML y XML *Schemas* por lo que se puede beneficiar de las herramientas disponibles y de sus ventajas. Además, proporciona la posibilidad de extender el lenguaje con nuevas características.

Los elementos básicos de *DAOP-ADL* son:

a) Los componentes y los aspectos de la aplicación.
b) Las propiedades que utilizan (para separar accesos a datos compartidos por varios componentes).
c) Las restricciones (*constraints*) de composición entre ellos que determinan cómo conectar los componentes entre sí, y cómo y cuándo hay que aplicar los aspectos a los componentes para extender el comportamiento del sistema con propiedades aspectuales (*weaving* (tejido) entre componentes y aspectos).
d) La información de despliegue que indica dónde se instancia cada entidad.
e) El contexto inicial que define los componentes y las propiedades globales que se crean al inicio de la ejecución.

Componentes y aspectos son elementos de primer orden del lenguaje. Los aspectos pueden interceptar mensajes dirigidos a la interfaz de los componentes mediante la definición de una *interfaz evaluada* –que determina los mensajes que los aspectos pueden interceptar- y un *interfaz de eventos* -que describe los eventos que los aspectos pueden capturar-. Los aspectos se pueden evaluar secuencialmente o concurrentemente. Se define de forma separada el comportamiento de un aspecto y la información de los puntos de intercepción de los componentes.

DAOP-ADL identifica a cada componente y aspecto con un nombre de rol (el mismo que tiene en el modelo CAM); y describe los atributos que definen su estado en términos de sus interfaces, las propiedades que usan y una lista de posibles implementaciones. Un componente viene descrito por tres interfaces: *proporcionada*, *requerida* y de *eventos* (lanzados por el componente). Un aspecto viene descrito por la *interfaz evaluada* que define los eventos que es capaz interceptar y cuándo. Un aspecto especial es el de coordinación, que, además de la interfaz evaluada implementa la *interfaz requerida* y un *protocolo de interacción*; este protocolo no es fijo, sino que varía al cambiar las necesidades de coordinación de cada aplicación.

Las restricciones de composición están formadas por las reglas de composición de componentes y las reglas de evaluación de aspectos (o reglas de composición entre componentes y aspectos). Las primeras resuelven posibles incompatibilidades entre el nombre de rol que aparece en la interfaz requerida de un componente y el nombre de rol del componente destino, y la incompatibilidad entre las propiedades compartidas.

En *DAOP-ADL* los aspectos se ejecutan en diversos momentos de la ejecución del sistema: cuando un componente envía o recibe un mensaje (los mensajes se envían entre componentes), cuando se dispara un evento (que es un mensaje sin destinatario conocido), o cuando se crea o se elimina un componente. En las situaciones anteriores se pueden ejecutar uno o más aspectos. En el caso del lanzamiento de eventos, un aspecto de coordinación (específico para cada sistema) resuelve el o los destinatarios de los mismos.

Tras describir la arquitectura en *DAOP-ADL* hay que realizar un proceso de análisis y validación de su contenido para comprobar si es correcta. Se han de validar:

- la descripción de los componentes y los aspectos, y
- las reglas de composición de los componentes y las de evaluación de los aspectos.

Por medio de la descripción en *DAOP-ADL*, se puede analizar qué componentes se pueden componer entre sí, viendo si coincide la interfaz requerida de uno con la proporcionada de otro. Hay veces que esta comprobación es directa, cuando la interfaz requerida de uno coincide con la proporcionada del otro. Sin embargo, esto no siempre es así, pues dos componentes se pueden componer también cuando la interfaz requerida de uno es un subconjunto de la interfaz proporcionada del otro. Estos casos necesitan un mecanismo de validación más complejo.

Finalmente, conviene decir que este lenguaje no soporta *quantification* a nivel arquitectónico ni una modularización explícita de aspectos arquitectónicos heterogéneos.

En [FuGaPi06] se presentó una extensión del lenguaje xADL, denominada *xDAOP-ADL,* que incorporaba el concepto de aspecto. Este lenguaje muestra cómo se puede añadir la noción de separación de aspectos a un LDA convencional. Sin embargo, hemos de decir que parece más potente *DAOP-ADL* que el nuevo lenguaje.

3.4.1.1. Otras características del lenguaje

En este apartado se presentan otras características interesantes de este lenguaje.

Extensibilidad de DAOP-ADL

Dado que *DAOP-ADL* usa esquemas XML y éste proporciona mecanismos para extender fácilmente los esquemas, el lenguaje se puede extender en varios sentidos. Por ejemplo, se puede añadir una nueva sección que proporcione información al validador. El mecanismo de extensión también se puede usar para definir nuevos aspectos o extender la descripción de los componentes y los aspectos (esta capacidad es la que se ha usado para definir el aspecto de coordinación). El lenguaje es extensible también con respecto a los tipos de datos utilizados en las interfaces, los atributos de estado, y las propiedades de los componentes y aspectos. También se puede extender mediante un esquema que defina en XML todos los tipos de datos utilizados en la descripción de la arquitectura; pero, en ese caso, se deben proporcionar los mecanismos para manejar estas diferencias en los tipos de datos. La última forma de extender *DAOP-ADL* consiste en extender el esquema XML que describe los protocolos de interacción encapsulados en los aspectos de coordinación. Se pueden añadir nuevas transiciones (atómicas y compuestas) así como nuevas funciones predefinidas que faciliten la definición del protocolo.

Dominio de aplicación

DAOP-ADL se define según [Pin04], sobre todo, para arquitecturas *peer-to-peer*, arquitecturas cliente-servidor y entornos virtuales colaborativos (una aplicación de entorno compartido integrado que se encuentra distribuida entre varias localizaciones, y todas ellas se comunican y colaboran entre sí).

Herramientas

DAOP-ADL permite la utilización de herramientas de validación que aseguren la corrección, desde el punto de vista arquitectónico, de la composición de componentes y la evaluación de aspectos. Por otra parte una herramienta (*DAOP-ADT tool*), creada por el grupo de trabajo de Pinto, facilita la definición de la arquitectura al proporcionar un modelo visual y una descripción en XML.

Generación de código

DAOP-ADL es un LDA que se define independiente de cualquier lenguaje de implementación.

Comunicación interna

El lenguaje permite que los componentes se envíen mensajes entre sí y que puedan lanzar eventos, que aquí se entienden como mensajes sin destinatario conocido, asunto que se encarga de resolver el aspecto de coordinación.

3.4.1.2. Conclusiones sobre DAOP-ADL

Con respecto a la descripción de *DAOP-ADL*, la autora, en [Pin04], destaca las siguientes conclusiones:

1. *DAOP-ADL* define la arquitectura de la aplicación de forma independiente a cualquier lenguaje de programación o plataforma de componentes, basado en XML.
2. La definición de la información sobre los aspectos es opcional, con lo cual se puede describir la arquitectura de cualquier aplicación basada en componentes, aplicando o no separación de aspectos.
3. Componentes y aspectos se consideran como entidades de primer orden. *DAOP-ADL* no considera conectores y configuraciones como elementos, sólo considera componentes. La abstracción de aspecto se define de un modo similar a componentes normales; se diferencian en cómo se componen los aspectos. Las dependencias entre los componentes y los aspectos se definen de forma explícita.
4. *DAOP-ADL* expone los *join points* de los componentes a través de sus interfaces requeridas.
5. *DAOP-ADL* permite la utilización de herramientas de validación que aseguren la corrección, desde el punto de vista arquitectónico, de la composición de componentes y la evaluación de aspectos.
6. El uso de XML y esquemas XML, así como la gran cantidad de software disponible para trabajar con este metalenguaje, facilita la construcción de herramientas que soporten la utilización de *DAOP-ADL*. Define un nuevo elemento XML: un conjunto de reglas que describen la composición. Sin embargo, carece del formalismo debido a su definición basada en XML. La extensibilidad proporcionada por los esquemas XML hace que el lenguaje sea fácilmente extensible, pudiéndose incorporar nuevas características no previstas inicialmente.

7. En *DAOP-ADL* los aspectos son elementos reutilizables, con la excepción del aspecto de coordinación (cuya naturaleza es diferente, pues se trata de un aspecto que se encarga de resolver los destinatarios de los eventos que lanzan los componentes), que es específico de cada arquitectura y, por tanto, difícilmente se podrá reutilizar.
8. Utiliza una plataforma dinámica que realiza el proceso de *weaving* (tejido) en tiempo de ejecución.
9. Como inconveniente de este lenguaje se puede decir que, al definirse sobre un modelo propio de componentes (*CAM*), no se puede utilizar en entornos más genéricos.

3.4.2. PRISMA ADL

PRISMA ADL es un LDA que extiende el lenguaje OASIS 3.0[18] [Let+98] para describir los conceptos de arquitecturas software y de aspecto proporcionado por OASIS. Al basarse en este modelo conceptual se puede definir su semántica de un modo formal y preservar sus principales ventajas: validación y verificación de los modelos arquitectónicos, y generación automática de código. Los *elementos arquitectónicos* (componentes y conectores), *interfaces* y *aspectos* se especifican en el LDA de *PRISMA* como definiciones de tipo siendo elementos de primer orden del lenguaje. Cada uno de estos elementos (tipos) se almacena en una biblioteca *PRISMA* para facilitar su reutilización. Además de la base de *PRISMA* sobre OASIS, el lenguaje ha sido extendido para representar elementos arquitectónicos, asociaciones (*attachments*), ligaduras (*binding*) y entrelazado de aspectos (*weaving*), entre otros.

A partir de los elementos proporcionados por *PRISMA ADL* se puede especificar la arquitectura de un cierto sistema definiendo su configuración. Ésta se define importando los tipos necesarios desde la biblioteca *PRISMA* e instanciándolos después. La descripción de la arquitectura se completa definiendo las asociaciones (*attachments*) entre componentes y conectores.

Una de las características más interesantes de *PRISMA ADL* es el modo en el que se realiza la definición de los elementos arquitectónicos: el lenguaje se divide en dos niveles, el nivel de tipo (que usa un *lenguaje de definición de tipos*) y el nivel de configuración (que usa el *lenguaje de configuración* de *PRISMA*). Esta separación tiene varias ventajas; en particular para la descripción de sistemas distribuidos, que dependen fuertemente de la configuración de las arquitecturas y de la ejecución de las instancias distribuidas, debido a su entorno dinámico y cambiante en tiempo real (reconfiguración). A continuación se describen los dos niveles:

A) *El lenguaje de definición de tipos* permite definir elementos arquitectónicos a nivel de tipo, lo que facilita combinar DSOA y DSBC; especifica las interfaces, aspectos y

[18] OASIS es un lenguaje formal para la definición de modelos conceptuales de sistemas OO que permite la validación y generación automática de aplicaciones.

elementos arquitectónicos a nivel de tipo y describe comportamientos independientemente del contexto, lo que potencia la reutilización.

- Las *interfaces* publican los servicios de los elementos arquitectónicos. Todos los servicios de una interface tienen que pertenecer al mismo tipo de aspecto (por ejemplo, a un aspecto de distribución o a un aspecto de replicación, pero no a ambos). En la definición de un *aspecto* se especifican las propiedades del aspecto y las interfaces cuya semántica define. Además, la definición de un aspecto contiene los servicios públicos de las interfaces que usa el aspecto y aquellos que son privados.

- La definición de los *aspectos* en *PRISMA* consta de:
  a) Una cabecera, en la que se especifica el tipo de propiedades que define.
  b) Una sección de atributos, para describir los atributos necesarios para especificar las propiedades del aspecto.
  c) Una sección de servicios que contiene los servicios publicados de las interfaces y que usa el aspecto, y los no publicados. En *PRISMA* una operación es un servicio no elemental y no atómico. Las operaciones atónicas se denominan transiciones que también se han de especificar dentro de la definición de un aspecto. Un protocolo es una secuencia de acciones que pueden ocurrir. La definición semántica del conjunto de servicios de los aspectos es diferente para los comportamientos cliente (*in*) y servidor (*out*).
  d) Una sección de evaluaciones (*valuation*) especifica cómo la ejecución de un servicio cambia el valor de los atributos y, por tanto, cambia el estado del aspecto y del elemento arquitectónico.
  e) En la sección de restricciones (*constraints*) se definen fórmulas basadas en el estado del aspecto y que deben satisfacerse para que el servicio se ejecute. Pueden ser estáticas o dinámicas.
  f) Las precondiciones establecen las condiciones a ser satisfechas para que se ejecuten los servicios.
  g) Un elemento disparador (*trigger*) que permite que un servicio definido en un aspecto de un elemento arquitectónico sea ejecutado siempre y cuando el aspecto que define el *trigger* alcance un cierto estado.

  Cuando se define un aspecto, éste se almacena en un repositorio *PRISMA* para que pueda ser reutilizado por otros elementos arquitectónicos del mismo o de diferentes modelos arquitectónicos. En *PRISMA* los requisitos no funcionales son aspectos, pero también la funcionalidad se especifica como un aspecto para facilitar la reutilización. La reutilización de las propiedades funcionales es independiente del modelo arquitectónico que lo contenga porque la funcionalidad se especifica como un aspecto.

  Una vez definidos, los aspectos se pueden mezclar para formar elementos arquitectónicos. Sin embargo, en la descripción de un aspecto no se mencionan dependencias entre aspectos y elementos arquitectónicos. Es decir, la definición de tipos permite describir comportamientos independientemente del contexto. Esto facilita la reutilización de un aspecto en diferentes elementos arquitectónicos después de haberlo definido y de que sus datos se almacenen en el repositorio de

tipos. Los elementos arquitectónicos pueden reutilizar estas descripciones, si son compatibles con los requisitos de comportamiento.

- La plantilla de un *elemento arquitectónico* se divide en cuatro partes: cabecera, puertos o roles, los aspectos y el *weaving*. La cabecera contiene la palabra clave *component_type* o *connector_type*, dependiendo del tipo de elemento arquitectónico que se esté definiendo. Los puertos (*ports*) o roles se especifican asignando a cada puerto o rol un tipo de interfaz. Se importan los aspectos que constituyen el elemento arquitectónico. El apartado *weaving* define la sincronización de los aspectos (*after, before o around*) y las dependencias entre los aspectos, adaptando esas sincronizaciones al comportamiento del elemento arquitectónico.

B) *El lenguaje de configuración* define un modelo arquitectónico (*Architectural Model*) que permite asociar instancias de elementos arquitectónicos, describiendo así la topología de la arquitectura a través de las asociaciones (*attachments*) y las ligaduras (*binding*) necesarias. En este nivel, las instancias de un elemento arquitectónico se asignan a una localización física y se registran en el modelo arquitectónico para ofrecer sus servicios remotamente. El modelo arquitectónico conoce las localizaciones de las instancias definidas en las conexiones de las asociaciones.

En la definición de un modelo arquitectónico se especifican los tipos de elementos arquitectónicos:

i) Componentes, conectores y sistemas que se importan de la biblioteca de *PRISMA*, en la que se almacenaron al definirlos con el *lenguaje de definición de tipos*.
ii) Una sección de instancias, de los tipos definidos anteriormente, instancia los tipos importados.
iii) Las conexiones entre los elementos instanciados que establecen la topología de la arquitectura.

La especificación de un sistema es similar a la de un elemento arquitectónico en cuanto a las secciones que contiene, entre las que ahora se incluye una sección de ligadura (*binding*).

El lenguaje soporta la modularización de las propiedades transversales (*crosscutting concerns*) en la que los aspectos son nuevas abstracciones del lenguaje, que se usan para definir la estructura o el comportamiento de los elementos arquitectónicos (componentes y conectores). Estos contienen información sobre la especificación de composición (*weaving*), que define cómo debe ser la ejecución de un aspecto y contiene operadores que describen el orden en que se realizará el proceso de mezclado (*after*, *before, around*).

Para la construcción de sistemas complejos en *PRISMA*, un marco de trabajo proporciona soporte para el modelo y su LDA-OA, así como para definir y generar arquitecturas software. El marco de trabajo se compone de un editor gráfico (*EGV-PRISMA*), una infraestructura (*middleware) .NET* y un compilador que automáticamente genera código fuente de las aplicaciones. El *middleware .NET* implementa una infraestructura que soporta diferentes tecnologías. El meta nivel de *PRISMA* contiene los servicios que proporcionan la creación del nivel base, la evolución y la reconfiguración

dinámica de la arquitectura software, en tiempo de ejecución. Estos servicios están implementados en la infraestructura de *PRISMA*. Actualmente estas funcionalidades están expresadas en C#. Los servicios de esta infraestructura son de dos tipos: los relativos a la cooperación necesaria entre los elementos arquitectónicos de la arquitectura y la infraestructura; y los servicios relativos a las alarmas que la infraestructura envía a los elementos arquitectónicos sin que haya una petición previa. Así, la infraestructura puede verse como un nivel abstracto entre la especificación del modelo arquitectónico de *PRISMA* y las diferentes plataformas. Es decir, es un *middleware* abstracto que esconde ciertas funcionalidades complejas de la aplicación final.

PRISMA es un lenguaje textual que podría usarse directamente para modelar arquitecturas software, pero proporciona una herramienta visual de modelado para representar las especificaciones. Esta herramienta se basa en UML y permite capturar las características más relevantes del modelo. Así, se ha definido un perfil UML que permite representar gráficamente las especificaciones de *PRISMA*.

3.4.2.1. Otras características del lenguaje

En este apartado se presentan otras características interesantes de este lenguaje.

Facilidad de extensión.

Como se ha comentado, al ser un lenguaje creado sobre el modelo *PRISMA* y ser él mismo una extensión de OASIS, no parece que sea complicada su extensión, sin embargo no parece necesario.

Dominio de aplicación

Ha sido diseñado para facilitar la construcción de grandes sistemas, complejos y distribuidos.

Herramientas

PRISMA dispone de herramientas de modelado y de representación gráfica basadas en UML. *EGV-PRISMA* es la herramienta gráfica que soporta la reutilización y el mantenimiento de los sistemas creados con *PRISMA*.

Generación de código

Los autores han desarrollado un compilador para generar código C# a partir de la especificación gráfica. Al extender OASIS, permite la generación automática de código de las aplicaciones en C++. Además, la especificación formal de este lenguaje facilita la generación de código.

Crear un modelo *PRISMA* y ejecutarlo supone seguir tres pasos:

i) A través de una herramienta visual se especifica el modelo arquitectónico.
ii) A partir del modelo arquitectónico se genera su código intermedio, dependiente de la plataforma de desarrollo (en C#).

iii) El código generado lo puede ejecutar el *middleware* de PRISMA que implementa el modelo, proporciona servicios de distribución y facilita la evolución.

Comunicación interna

La comunicación interna entre elementos del sistema se realiza a través de las interfaces. La comunicación se define externa a los componentes y aspectos, de modo que es independiente del contexto.

3.4.2.2. Conclusiones sobre PRISMA ADL

Con respecto a *PRISMA ADL*, se puede concluir lo siguiente:

1. Sigue un modelo simétrico.
2. El lenguaje se define en dos niveles: *nivel de definición de tipos* y *nivel de configuración*. En el LDA, al igual que en el modelo, se definen *elementos arquitectónicos* (que pueden ser componentes y conectores) que se comunican a través de interfaces y están formados por un conjunto de aspectos.
3. Facilita la reutilización de componentes y aspectos al definirse fuera de ellos las dependencias contextuales (*tejido* externo). Los aspectos se consideran como entidades de primera clase del lenguaje, pueden ser funcionales o no funcionales, y se usan para definir la estructura o el comportamiento de los elementos arquitectónicos. Los aspectos tienen acceso directo a las propiedades de un componente o conector para definir el comportamiento de ese elemento. Un aspecto especial es el de coordinación.
4. Una plataforma .NET (*Prisma.net*) facilita la descripción de sistema as distribuidos.
5. *PRISMA* tiene reconfiguración dinámica, no tiene *quantification* y facilita la evolución de los sistemas en tiempo de arquitectura.
6. Dispone de una herramienta de modelado (*EGV-PRISMA*) que facilita mantenimiento de los sistemas desarrollados bajo el modelo y su LDA.
7. En *PRISMA,* la composición de los aspectos se define dentro de los componentes o conectores, en la especificación del tejido (*weaving*). Esto puede añadir cierta complejidad a la descripción arquitectónica del sistema ya que la información de *weaving* queda dispersa dentro de los elementos arquitectónicos que contienen a dichos aspectos.

3.4.3. AO-Rapide

AO-Rapide es un LDA que extiende el lenguaje Rapide e integra dicha extensión con el modelo de coordinación exógeno *Reo*. El lenguaje se extiende para dar soporte a las especificaciones arquitectónicas de un sistema OA; es decir, para dar soporte a la separación de las *propiedades transversales*. De esta manera el nuevo LDA soporta la especificación de puntos de enlace (*join points*) en los componentes, la especificación de aspectos y la especificación de conectores, formalizando la interacción entre componentes regulares y de aspecto. Para ello se apoya en *Reo*, basándose en él las primitivas de

coordinación que extienden Rapide, lo cual puede complicar la comprensión del nuevo LDA.

En *AO-Rapide* las arquitecturas se especifican mediante componentes, aspectos y conectores. Este tipo conector es diferente de los conectores del lenguaje Rapide y, por tanto, no deben confundirse. Los conectores de *AO-Rapide* son conectores de *Reo* que realizan una función de coordinación entre los aspectos y los *join points*. Tanto los componentes del sistema como los aspectos se especifican como *interfaces* Rapide. La diferencia entre ambos es que los servicios de los aspectos no pueden invocarse por parte de los componentes. Así, los aspectos se definen como elementos de primera clase sin tener que añadir nuevos elementos al LDA. Por otra parte, la relación entre los aspectos y el sistema se puede hacer explícita. Los conectores definen la coordinación entre componentes y aspectos. Para conseguir una total integración entre Rapide y el modelo *Reo*, en *AO-Rapide* los conectores de *Reo* se especifican como *interfaces* Rapide.

Todo aquello que esté relacionado con los aspectos se define de forma separada a los componentes regulares del sistema, de manera que la evolución de *AO-Rapide* a Rapide no afecte a la representación que se hace de los componentes en Rapide.

Los tipos de puntos de enlace que puede tratar el lenguaje son: llamadas a función, retornos de función y acciones de componentes (*eventos* en Rapide). La ocurrencia de estos eventos es especialmente observada por los conectores.

En *AO-Rapide* se observan los principios de *quantification* e *inconsciencia*. Para ello se define un conector entre un aspecto y el punto de enlace (*join point*) de un componente. Este *join point* se incluye en un envolvente (*wrapper*) del componente para mantener el principio de inconsciencia. El conector contiene un coordinador en el que se usa *Reo* y, por tanto, este conector realiza una labor de coordinación. Así, en *AO-Rapide* cada componente tiene un envolvente (*wrapper*) cuya misión es comunicar a los conectores la ocurrencia de eventos, de modo transparente a ese componente, no siendo necesario especificar los *join points* en los componentes. Esto permite añadir y eliminar aspectos al sistema sin que los componentes resulten afectados. En *AO-Rapide* los *wrappers* se definen como *interfaces* Rapide.

Se pueden especificar puntos de corte (*pointcuts*) complejos pertenecientes al mismo componente usando *patrones de eventos* de Rapide. También se pueden especificar *pointcuts* complejos pertenecientes a componentes diferentes usando patrones de coordinación correspondientes a los conectores de *Reo*, pues *Reo* permite especificar patrones de coordinación exógenos, que se pueden representar como expresiones regulares sobre operaciones de entrada/salida.

Los conectores en *AO-Rapide* permiten la comunicación entre componentes y aspectos, y entre aspectos definiendo cuándo y cómo se ejecutan éstos. Cuando varios aspectos están asociados al mismo *join point* o *pointcut* y hay dependencias entre ellos, son necesarias reglas de coordinación. Los conectores se componen de tres partes, según se explico en la definición del modelo (apartado 3.3.5.2).

Especificación de la arquitectura

Identificados los componentes y los aspectos del sistema, se puede definir su arquitectura:

- Cada componente y aspecto se especifica como una *interfaz* Rapide.
- Se identifican los componentes que se deben asociar con los aspectos.
- Para cada componente identificado se determinan las reglas de coordinación que controlan la interacción entre dicho componente y sus aspectos asociados.
- Se construye y especifican los coordinadores como conectores *Reo*, que implementan las reglas de coordinación o bien se usan los coordinadores proporcionados por *AO-Rapide.*
- Se especifican el *wrapper* y los *linkers* asociados al aspecto.
- El *wrapper* de cada componente se conecta con el correspondiente coordinador a través del *wrapper linker* apropiado, donde se indican los *join points* para el componente.
- Cada coordinador se conecta con el correspondiente de aspecto a través del *aspect linker* adecuado.

Para determinar las reglas de composición para un *join point* o un *pointcut*, si esto implica a varios aspectos, pueden ocurrir dos cosas:

- Si no hay dependencias entre los aspectos implicados se generan varias conexiones asociadas al mismo *join point* o *pointcut*, cada una con su propio conector.
- Si hay dependencias entre los aspectos hay tres opciones: i) se pueden combinar todos los aspectos en uno solo; ii) se puede construir una conexión compleja con un coordinador que implemente todas las reglas de coordinación, incluyendo la coordinación entre los aspectos; y iii) se pueden coordinar los aspectos de forma separada, conectando cada aspecto implicado en el mismo *join point* o *pointcut* a los *wrappers* de los componentes, como si se tratara de aspectos independientes y, a continuación, se define un *componente de coordinación* que especifique la coordinación requerida entre los aspectos implicados, lo cual da lugar a un modelo más simple que en las opciones anteriores.

Finalmente, la formalidad en *AO-Rapide* es una característica que mantiene del lenguaje que extiende.

3.4.3.1. Otras características del lenguaje

En este apartado se presentan otras características interesantes de este lenguaje.

Facilidad de extensión.

Por las características del lenguaje, las extensiones en *AO-Rapide* no parecen sencillas.

Dominio de aplicación

En principio, *AO-Rapide* se puede utilizar en los mismos dominios que Rapide, con la aportación de que ahora se puede usar en el desarrollo de sistemas orientado a aspectos.

Herramientas

No hemos encontrado referencias a herramientas para llevar a cabo la definición de arquitecturas en *AO-Rapide*. De existir, tendría que mantenerse la compatibilidad con Rapide.

Generación de código

AO-Rapide genera código (en C/C++, en ADA o en VHDL) a partir de las descripciones en Rapide para representar modelos ejecutables en tiempo de diseño, los cuales pueden simularse antes de comenzar la implementación. Esta característica es útil pues permite comprobar el comportamiento de un sistema en la fase de diseño.

Comunicación interna

En *AO-Rapide*, los componentes lanzan mensajes y eventos para comunicarse entre sí, característica tomada de Rapide.

3.4.3.2. Conclusiones sobre AO-Rapide

De lo anterior, se pueden extraer las siguientes conclusiones:

1. Es una extensión de Rapide para soportar desarrollos orientados a aspectos en la que éstos se consideran entidades de primer orden. Todo lo relacionado con los aspectos se define de forma separada a los componentes, de manera que la evolución de *AO-Rapide* a Rapide no afecte a la representación que se hace de los componentes en Rapide.
2. Para la extensión de Rapide se ha usado un modelo de coordinación basado en canales, *Reo*, lo cual puede complicar la comprensión de este nuevo LDA. Cada aspecto tiene asociado su coordinador definido de forma externa al aspecto, lo que facilita la independencia y reutilización de los aspectos.
3. En *AO-Rapide* se puede generar un prototipo ejecutable del sistema en tiempo de diseño y llevar acabo el análisis de la arquitectura. Sin embargo, no existen herramientas que faciliten la especificación de las arquitecturas.

3.4.4. AspectualAcme

AspectualAcme [Bat+06b] es un LDA-OA que extiende Acme [GaMoWi00] para soportar la representación modular de los componentes arquitectónicos y su composición. Según sus autores,

se ha elegido Acme como lenguaje base para definir un LDA OA debido, entre otras razones, a su simplicidad, ser independiente del dominio y disponer de herramientas que facilitan el diseño y la generación de código.

En *AspectualAcme* los aspectos se definen como componentes (*componentes de aspecto*) que tienen la misma estructura que los componentes base, salvo que representa una propiedad en una interacción transversal a varios componentes base (*crosscutting concern*).

La única extensión necesaria para integrar aspectos en Acme se hace introduciendo el concepto de *Conector Aspectual (Aspectual Connector –ACon)* y el de *quantification* a nivel de configuración. Un *conector aspectual* es un conector Acme con una nueva interfaz que se define por una parte para distinguir entre los elementos que intervienen en una interacción que hace *crosscut* (es decir, que afecta a componentes base y a componentes de aspecto); y por otra, para capturar cómo se interconectan ambas categorías de componentes. Como consecuencia, un *conector aspectual* relaciona un componente de aspecto con un componente base. De este modo, el nuevo lenguaje soporta la integración de los *crosscutting concerns* en un LDA y los mecanismos de composición del DSOA a nivel de arquitectura. El lenguaje se centra en un mecanismo de composición construido en torno al concepto de *conector aspectual*.

Los autores justifican la definición de este concepto pues, para ellos, el de conector tradicional no es suficiente para modelar la interacción transversal porque el modo en el que un *componente aspectual* se compone con un componente regular es diferente a la composición entre dos componentes normales. Así, un *crosscutting concern* viene representado por los servicios que proporciona un componente aspectual y puede afectar a los servicios que proporcionan o requieren otros componentes, los cuales, por su parte, se pueden considerar como puntos de enlace (*join points*) arquitectónicos.

Para expresar la interacción transversal, el *ACon* tiene definido un nuevo tipo de interfaz (*interfaz transversal*) que permite distinguir entre los elementos que juegan distintos roles en ella; es decir, afecta a componentes base y a componentes de aspecto. Además, la nueva interfaz determina cómo se interconectan dichos componentes. Así, la interfaz del *ACon* contiene 2 roles: un *role base* que se conecta a un puerto del componente base (de entrada o salida) y un *role crosscutting concerns* que se conecta a un componente aspectual. Por otra parte, los *ACon* tienen una cláusula *glue* (heredada de Acme) que especifica los detalles de la composición entre componentes base y de aspecto; es decir, que especifica cómo un *ACon* afecta a los componentes regulares. Hay 3 tipos de *glue aspectual*: *after*, *before* y *around* cuya semántica es similar a la de los *advice* de aspecto. Los *ACon* pueden ser binarios (la cláusula *glue* es una declaración de tipo) o puede ser que más de un *role base* o *role crosscutting concerns* forme parte de la declaración del *conector aspectual*. En este caso, la cláusula *glue* es más compleja.

AspectualAcme permite la especificación de la interacción arquitectónica entre dos o más conectores aspectuales que tengan *join points* comunes. Esta interacción se declara en la descripción de la configuración, en la sección de *Attachment* (que es en donde se definen los *join points),* por lo que no es necesario definir nuevas primitivas para describir la composición, que se modela en la descripción de los conectores y de la

configuración. Por otra parte, este lenguaje soporta dos tipos de composición de operadores: precedencia y XOR [Gar+06].

El modelo de composición se centra en el concepto de *conector aspectual*, que se basa en los conceptos generales de conector arquitectónico (conectores y configuraciones). De este modo, para realizar la definición arquitectónica de los *crosscutting concerns* sólo hay que hacer ciertas extensiones, a fin para soportar la definición de algunas características de composición, como: a) interfaces transversales definidas a nivel de conector; b) un pequeño conjunto de declaraciones de interacción de aspectos como *attachments,* a nivel de arquitectura; y c) un mecanismo de *quantification* para asociar las descripciones arquitectónicas que se representan en la sección de *Attachments*, que es donde se define la configuración del sistema. Esto, por otra parte, facilita la reutilización de los conectores.

3.4.4.1. Otras características del lenguaje

A continuación se presentan otras características interesantes de este lenguaje.

Facilidad de extensión.

Por definición, el lenguaje no incluye ninguna nueva abstracción para representar la arquitectura de sistemas orientados a aspectos, por lo que no parece necesaria su extensión.

Dominio de aplicación

AspectualAcme, al igual que Acme, es independiente del dominio y proporciona estructuras que permiten describir un amplio rango de sistemas.

Herramientas

AspectualAcme dispone de herramientas y librerías heredadas de Acme que facilitan el diseño. Por otra parte, en [SaChCh06] se propone un modo de pasar descripciones en *AspectualAcme* a UML 2.0. Sin embargo, hasta el momento no se han desarrollado herramientas para soportar la creación de descripciones arquitectónicas en *AspectualAcme* y su transformación en descripciones de diseño.

Generación de código

AspectualAcme hereda de Acme la característica relativa a la construcción de configuraciones ejecutables, pero sin soporte dierecto para la generación de código.

Comunicación interna

La comunicación entre elementos del sistema se hereda de Acme, incluyéndose sólo el nuevo tipo de interfaz para los conectores aspectuales y considerando los *join points*.

3.4.4.2. Conclusiones de AspectualAcme

De lo anterior, se pueden extraer las siguientes conclusiones:

1. Se mantiene la simplicidad y expresividad de Acme.
2. Los aspectos se consideran entidades de primer orden, componentes; sin embargo no es necesario definir nuevas abstracciones para representar el concepto de aspecto, sino que se basa en enriquecer la semántica de composición soportada por los conectores arquitectónicos.
3. No hay una distinción explícita entre componentes regulares y de aspecto. La representación de la interacción transversal se hace mediante conectores (se introduce el concepto de *conector aspectual*) lo que facilita la reutilización de los componentes de aspecto.
4. Para modelar la composición se usa la descripción de los conectores y la configuración; no se necesitan cambios en la interfaz de los componentes regulares.

3.4.5. FAC ADL

Pessemier et al. en [Pes+06] definen *Fractal Aspect Component,* un modelo que extiende Fractal y su LDA en [PeSeDu04]. *FAC* es un modelo para relacionar componentes y aspectos que promueve la integración de POA y DSBC, por lo que se puede decir que su objetivo es capturar las propiedades que atraviesan (*crosscut*) el sistema. Es un marco de trabajo basado en el modelo de componentes de Fractal y proporciona un LDA para describir arquitecturas software con propiedades transversales (*crosscutting concerns*).

La extensión que propone *FAC* para considerar los *crosscutting concerns* sobre Fractal consiste en superponer, sobre un conjunto inicial de componentes, un nivel asociado a un conjunto de elementos que corresponde al dominio del código transversal. Para llevar esto a cabo, se definen nuevas ligaduras (*binding*). Además, es necesario disponer de un mecanismo que permita redireccionar la salida de un cierto componente hacia el nuevo entorno.

FAC propone tres abstracciones que definen esta extensión:

a) *Componente aspectual* (*Aspectual Component: ACom*) que especifica las *propiedades transversales* (*crosscutting concerns*) de la arquitectura. Está compuesto por dos elementos, uno es la propiedad propiamente dicha (*concern*) que implementa los servicios de un componente regular como un *advice*; y otro, que implementa la intercepción asociada al *concern* (*pointcut*) y los métodos interceptores (*method calls*). De este modo, los *componentes aspectuales* pueden afectar a los componentes mediante esta interfaz especial, *interfaz de intercepción*, que permite definir la ligadura (*binding*) entre los componentes regulares y de aspecto. Así, los *ACom* modifican o extienden el comportamiento de los componentes regulares a través de los *join points.* Es decir, un *crosscutting concern* se implementa en *FAC* como un componente Fractal que proporciona al menos una interfaz (de tipo *server*).

b) Un mecanismo de tejido (*weaving*) que permite relacionar las *propiedades transversales* (*aspectual component*) con la arquitectura original (componentes regulares). Esto se puede hacer a través del lenguaje o de la API. La ligadura (*binding*) de un *ACom* es similar a una ligadura entre los componentes regulares, pero en este caso un componente es el *ACom* y el otro tiene que proporcionar un método interceptor (*interceptor controller*). Se consideran dos tipos de ligadura entre componentes regulares y *ACom*. El primero, la *ligadura de intercepción* (*Direct Crosscut Binding*) que se define entre un componente regular y un *ACom*; todos los métodos del componente son interceptados por el *ACom*. El segundo, *crosscut binding,* que usa expresiones de *pointcut* (que se definen a partir de los nombres de los componentes), los nombres de las interfaces y los nombres de los servicios (métodos) sobre los que se aplicará el *ACom*. Los servicios que coinciden (*match*) son "aspectizados" por el *ACom*. Los puntos de intercepción (*pointcuts*) se definen a través del *crosscut binding*. Un *pointcut* selecciona los componentes, interfaces y servicios sobre el que se aplica el componente de aspecto. Esto permite expresar la *quantification*. Por otra parte, las ligaduras pueden ser entre dos componentes (*primitive binding*) o entre varios componentes (*composite binding*), característica heredada de Fractal. Las ligaduras además de ser directas pueden ser recursivas a través de una jerarquía de componentes compuestos.

c) *Dominios de aspecto* (*aspect domain*) que representan la *reificación* de los componentes regulares afectados por los componentes de aspecto. La *ligadura de aspecto* define una asociación (*link*) entre un componente regular y uno de aspecto. Este puede modificar el comportamiento del primero si resultan afectados sus *join points*.

Las interfaces juegan un papel importante en la definición de los sistemas con *FAC* (característica heredada de Fractal). Hay que distinguir entre *interfaces de negocio* y *de control*. *Las interfaces de negocio* son puntos de acceso externo a los componentes. *FAC* hereda de Fractal la posibilidad de definir interfaces clientes o servidoras: una interfaz servidora recibe invocaciones de operaciones emitidas por interfaces cliente. *Las interfaces de control* proporcionan un cierto control sobre los componentes a los que están conectadas. Estas interfaces se encargan de llevar a cabo ciertas operaciones no funcionales del componente o la gestión de sus enlaces (*binding*) con otros componentes.

Con estas características *FAC* permite la definición, configuración y separación de *concerns* funcionales y no funcionales. Además, tanto componentes como interfaces y *bindings* son dinámicos y los *ACom* se tejen y destejen en tiempo de ejecución, de modo semejante a como lo hace Fractal con las interfaces funcionales.

Por otra parte, los componentes pueden ser reflexivos: la *introspección arquitectónica* permite monitorizar el sistema y la *introspección dinámica* dota al sistema de reconfiguración dinámica.

Fractal Aspect Component Model es recursivo en el sentido en que sus componentes pueden ser simples o compuestos. Los componentes pueden ser compartidos en el sentido de que un componente puede formar parte de varios componentes compuestos.

En *FAC* se propone un modelo multinivel que captura los cambios estáticos y dinámicos en un sistema basado en componentes en el contexto de un LDA. Propone el uso de técnicas de POA para realizar la adaptación funcional y no funcional. La evolución/adaptación de los sistemas construidos con *FAC* es sencilla; basta con añadir el nuevo comportamiento considerándolo como un *ACom* y definir dónde hay que aplicarlo. El modo en el que se va a disparar el nuevo comportamiento se expresa mediante especificaciones estructurales o de comportamiento (*pointcuts*). Los elementos estructurales (como la signatura de un método, una interfaz funcional o un componente) se pueden usar como un *join point* del sistema. Cada operación de un componente (ya sea proporcionada o requerida) puede ser interceptada por las nuevas características.

3.4.5.1. Otras características del lenguaje

En este apartado se presentan otras características interesantes de este lenguaje.

Facilidad de extensión.

FAC es un modelo de alto nivel que se implementa bajo Fractal que es un modelo de componentes extensible y modular, por lo que, en principio, la extensión no supone ninguna dificultad, aunque no parece necesaria.

Dominio de aplicación

Se ha definido orientado al desarrollo de sistemas basados en componentes.

Herramientas

La única referencia que hemos encontrado sobre herramientas para *FAC* (probablemente porque aún está sin completar el desarrollo del modelo) es que utiliza *AOP Aliance* para la definición de los *ACom.* Esta es una iniciativa *open-source* que facilita la definición de un API común para marcos de trabajo (*frameworks*) en el entorno de POA. El API está implementado en Spring y JAC.

Generación de código

FAC, al extender Fractal, mantiene la representación de su LDA basado en XML que permite relacionar componentes regulares y aspectos del mismo modo que dos componentes regulares.

Comunicación interna

La comunicación interna entre elementos del sistema se realiza a través de las *interfaces de intercepción*. La interacción de los aspectos se define externa a los componentes, ya que cada aspecto captura la interacción de cada componente al que está asociado. El aspecto se dispara si la secuencia de intercepción coincide (*match*).

3.4.5.2. Conclusiones sobre FAC

Los siguientes puntos son conclusiones que se pueden extraer sobre *FAC*:

1. La aproximación de Pessemier soporta componentes y configuraciones pero no conectores. Los aspectos se consideran como entidades de primera clase del lenguaje al definirse como componentes.
2. El lenguaje está basado en XML y modela la ligadura entre componentes y aspectos del mismo modo que se hace para componentes normales. La *quantification* se expresa mediante la definición de las ligaduras.
3. La composición de los aspectos se define mediante la *interfaz de intercepción*. El *weaving* es externo a los componentes regulares e independiente de los aspectos. El componente sobre el que se aplica el aspecto no tiene dependencia de éste.
4. Tiene reconfiguración dinámica, la evolución de los sistemas es sencilla, y propone un modelo que captura los cambios dinámicos y estáticos para realizar la adaptación de los sistemas.

3.4.6. PiLAR orientado a aspectos

El lenguaje *PiLAR* ha sido extendido [Cue+04c] a fin de mostrar sus posibilidades en la descripción de arquitecturas software orientadas a aspectos y para representar los aspectos arquitectónicos de un modo directo en lugar de utilizar la visión reflexiva [Cue+04a]. La extensión, que se basa en el concepto de superposición de Katz [Kat93] se lleva a cabo sin alterar la semántica del lenguaje base, pues la base reflexiva de éste lo permite. La tabla 3.2a tomada de [Cue+04c] muestra la correspondencia entre los conceptos reflexivos y aspectuales del lenguaje.

La adaptación de *PiLAR* al paradigma de orientación a aspectos se realiza sólo introduciendo nuevos elementos sintácticos (relativos a los conceptos que se muestran en la tabla 3.2b (extraída de [Cue+04b]) y añadiendo lógica temporal (cálculo μ modal) en forma de aserciones.

3.4.6.1. Otras características del lenguaje

En este apartado se presentan otras características interesantes de este lenguaje.

Facilidad de extensión.

PiLAR OA ha sido extendido en su sintaxis para admitir la representación de aspectos por lo que no parece necesarias nuevas extensiones.

Dominio de aplicación

Por definición del lenguaje base, facilita la representación de sistemas dinámicos.

Nuevo Concepto	Noción Aspectual *Análoga*	Concepto Reflexivo Anterior
Punto de Vista	Interés *(Concern)*	Categoría de Reificación
Vista Arquitectónica	Corte Transversal *(Crosscut)*	(Subconjunto de) Nivel Meta
Componente Multidimensional	Hipermódulo	Meta-Espacio (Extendido)
Fragmento Arquitectónico Componente Parcial	Componente Aspectual Aspecto, Hipertrozo	Componente-Meta Compuesto (Subconjunto de) Meta-Espacio
Exterfaz	Interfaz de Aspecto	Metafaz (Meta-Interfaz)
Aserción de Vínculo	Designador de Punto de Corte Conector (de Aspectos)	-
Superposición (Destino de)	Punto de Corte *(Pointcut)*	Reificación (Destino de)
Superposición (Relación)	Tejido Dinámico	Reificación (Relación)
Combinación (Superpostura)	*Sistema* Tejido	Reificación (Conjunto)
β-Componente	*Módulo* Base	Componente Base, Avatar
σ-Componente	Aspecto (Parte de)	Meta-Componente, Rohatar
σ-Restricción	Consejo	Meta-Restricción
Componente en un Fragmento	Envolvente Aspectual	Componente Meta, Niyatar
Componente-Rol	Punto de Corte (Destino de)	Reificación (Destino de)
β-Acción Vinculada	Punto de Unión	Sincronización con Avatar

Tabla 3.2a). Conceptos reflexivos y aspectuales en PiLAR.

Palabra Clave	Noción	Estructura Básica
\fragment	Fragmento Arquitectónico	Análoga a la de un componente compuesto, incluyendo componentes-rol.
\rolecomp	Componente-Rol	Declaración de una instancia que va a actuar como v-componente; hueco en un fragmento.
\exterface	Exterfaz (Interfaz Externa)	Interfaz no superpuesta, que un a-componente reserva para su propio uso.
bcomp	β-Componente	Prefijo para los elementos de un β-componente.
scomp	σ-Componente	Prefijo para los elementos no superpuestos.
\impose	Superposición	Operador dinámico que superpone un fragmento sobre varios β-componentes.
\del	Destejido *(Unweaving)*	Borrado (destrucción) de una superposición.
assertion	Aserción de Vínculo.	Sintaxis para la selección de puntos de unión.

Tabla 3.2b). Elementos sintácticos en PiLAR extendido.

Herramientas

No hemos encontrado ninguna herramienta de modelado para este lenguaje.

Generación de código

Al igual que *PiLAR*, este es un lenguaje teórico y no soporta la generación de código.

Comunicación interna

La comunicación interna entre elementos del sistema se realiza a través de puertos de entrada y salida definidos en las interfaces de los componentes.

3.4.6.2. Conclusiones sobre PiLAR

1. La extensión de *PiLAR* para la OA soporta componentes y configuraciones, pero no conectores.
2. Es un lenguaje dinámico que permite expresar arquitecturas dinámicas OA. El lenguaje está basado en cálculo π; el dinamismo arquitectónico se logra combinando lógica temporal (que permite expresar la *quantification*) y la superposición de aspectos.
3. Utiliza la capacidad reflexiva del lenguaje base para implementar aspectos de manera directa [Cue+04c].
4. La definición de este lenguaje muestra que es posible expresar aspectos con ADL existentes, sin alterar su semántica.

3.5. LDA-OA. Conclusiones

Después de haber estudiado los lenguajes de descripción arquitectónica orientados a aspectos que se incluyen en esta sección se pueden extraer varias conclusiones:

1. Por una parte, se puede decir que no hay consenso sobre las características relativas a la integración de las abstracciones de orientación a aspectos y la arquitectura del software; tampoco hay un acuerdo en cómo abordar la descripción de los aspectos arquitectónicos, aunque en [Nav+02] propusimos un conjunto de requisitos que deberían cumplir los LDA para permitir la gestión de los aspectos usando abstracciones arquitectónicas y en [Bat+06b] se propone un *framework* que representa las características que deberían soportar estos lenguajes.
2. Se han propuesto algunos LDA-OA como extensión de LDA existentes (*AspectualAcme, AO-Rapide*); otros son nuevos lenguajes definidos para integrar los conceptos de OA a nivel arquitectónico (*DAOP-ADL, PRISMA ADL*).
3. Por otra parte, del estudio de los diversos LDA-OA se puede concluir también que la introducción de los conceptos de OA en un LDA, para resolver el problema de modelar los *crosscutting concerns,* se ha realizado de varias formas:

- Algunos lenguajes no definen nuevas abstracciones arquitectónicas para representar aspectos, si bien es necesario ampliar o modificar la definición de ciertos elementos arquitectónicos ya existentes como conectores (*conectores aspectuales*) o componentes (*componentes aspectuales*), y extender el concepto de composición en los nuevos lenguajes.
- En general, los LDA-OA que introducen nuevas abstracciones en LDA existentes son aproximaciones pesadas (*AO-Rapide*). Por otra parte, aproximaciones más ligeras tratan de aprovechar abstracciones proporcionadas por los LDA convencionales, con adaptaciones para soportar un modelado efectivo de los *crosscutting concerns*, sin introducir complejidad adicional en la especificación arquitectónica (*AspectualAcme*).

4. Otra conclusión que se puede extraer del estudio anterior es que hay varias formas en la que los LDA-OA representan los aspectos, aunque todos los consideran como entidades de primer orden. Los aspectos se pueden considerar como componentes (*FAC ADL, DAOP ADL, AO-Rapide*), como conectores (*AspectualAcme*), o como componentes y conectores formando parte de elementos arquitectónicos (*Prisma ADL*).
5. Algunos son ejecutables mediante la generación de un prototipo (*AO-Rapide*).
6. En el caso de la extensión aspectual de *PiLAR*, un lenguaje reflexivo y dinámico, se ha mostrado cómo es posible expresar conceptos de orientación a aspectos apoyándose en conceptos reflexivos y en el de superposición.

Las Tablas (3.3, 3.4, 3.5 y 3.6) expresan, a modo de conclusión y resumen, algunas de las características de los LDA-OA estudiados, y en las que aparecen sombreadas las características del lenguaje que se presenta en esta tesis (*AspectLEDA*), y se desarrolla en el Capítulo 6 de esta memoria.

3.6. Conclusiones del capítulo

A lo largo de este capítulo se ha hecho una revisión de los conceptos relacionados con el desarrollo de sistemas OA y de las actividades a realizar a lo largo del ciclo de vida dentro de este paradigma; igualmente se han comentado algunas aproximaciones al modelado orientado a aspectos relacionadas, de algún modo, con la aproximación que se presenta en esta memoria. Se ha considerado interesante dedicar una sección a tratar el tema de la evolución de los sistemas en relación con el DSOA y se han estudiado con detalle los Lenguajes de Descripción Arquitectónica Orientados a Aspecto más relevantes para esta tesis.

Teniendo en cuenta lo expuesto, se puede concluir que la orientación a aspectos es una disciplina con un creciente interés y que todas las fases del desarrollo software se pueden tratar desde este paradigma, en el que, sin embargo, cada modelo aporta su propio punto de vista.

Se han estudiado varias aproximaciones OA; sin embargo, éstas no proporcionan soporte para cubrir todas las fases del desarrollo, incluso el mantenimiento; podía incluirse el modelo *PRISMA,* pero sigue una aproximación simétrica. Así, se echa de menos una propuesta que proporcione un modelo orientado a aspectos para modelos asimétricos, utilizando para su formalización un LDA regular, que deberá extenderse para soportar aspectos. Tal modelo debería incluir:

1. Una semántica adecuada para soportar aspectos.
2. Permitir la especificación de arquitecturas software que se puedan validar.
3. Facilitar la evolución de sistemas existentes para adaptarlos a nuevas situaciones o a cambios no previstos, considerando los nuevos requisitos como aspectos.
4. Soporte técnico para facilitar el trabajo del arquitecto de software en la especificación de arquitecturas software orientadas a aspectos.
5. Un LDA que permita la formalización de las arquitecturas, definido a partir de LDA convencionales.
6. Obtención de prototipos ejecutables, desde fases tempranas del desarrollo, concretamente, desde el nivel arquitectónico, para validar la arquitectura a partir de las especificaciones expresadas en el LDA OA.

En esta tesis, se presenta *AOSA Space* como una aproximación definida para la creación de sistemas orientados a aspectos desde un punto de vista arquitectónico y para cumplir con estos objetivos. Así,

> AOSA Space *es un marco de trabajo que integra un modelo asimétrico,* AOSA Model; *una metodología; y un LDA-OA,* AspectLEDA*, definido a partir de un LDA regular, que permite ejecutar prototipos y validar la arquitectura.*

Además, el arquitecto de software dispone de una caja de herramientas que le asiste a lo largo del proceso.

La descripción de *AOSA Space* se distribuye en esta memoria del siguiente modo:

- *AOSA Model* se describe en el Capítulo 4.
- La metodología asociada y el metamodelo en el Capítulo 5.
- El lenguaje y una herramienta en el Capítulo 6.
- Una segunda herramienta en el Anexo 2.

Lenguaje	*Año*	*Autor*	*Principal característica*	*Basado en*	*Modelo base*	*Formalismo*
DAOP-ADL	2003	Pinto (U. de Málaga)	Basado en XML. Lenguaje propio	CAM	CAM/DAOP	XML
Prisma ADL	2005	Pérez (U. P. de Valencia)	Lenguaje propio	OASIS	PRISMA Model	OASIS
AO-Rapide	2005	Palma (U. Católica de Chile)	Extiende Rapide	Rapide y Reo	AO-Rapide Model	El de Rapide
AspectualAcme	2006	Batista et al. (U. Río Grande, U. Lancaster, U. Bahía)	Extiende Acme	Acme	Ninguno concreto	El de Acme
Fac	2004	Pessemier (INRIA)	Extiende Fractal ADL, basado en XML	Extiende Fractal ADL	FRACTAL	XML
PiLAR OA	2004	Cuesta (U. de Valladolid, UEM)	Extiende PiLAR. Reflexivo y dinámico	PiLAR, superposición, cálculo π	-	El de PiLAR
AspectLEDA	2005	Navasa (U. de Extremadura)	Extiende LEDA s	LEDA y CR	AOSA Model	El de LEDA

Tabla 3.3. LDA OA estudiados y *AspectLEDA*.

LDA OA	*Descripción*	*Interfaces de componentes*	*Tipos*	*Vistas del sistema*	*Invasivo*	*Evolución*
DAOP-ADL	Utiliza el modelo CAM de componentes y aspectos. Una plataforma dinámica realiza el tejido de componentes y aspectos en tiempo de ejecución	Los componentes tienen interfaces, que pueden ser proporcionadas, requeridas o de tipo evento	Se pueden asociar tipos a ciertos elementos de una arquitectura. Pueden ser tipos XML o bien tipos de un lenguaje determinado	Una sola vista	No	Sencilla
PISMA ADL	Se define sobre OASIS. Define elementos arquitectónicos como elementos del lenguaje	Se definen como definiciones de tipo	Los elementos arquitectónicos son tipos. Lenguaje de definición de tipos	Vista interna y externa de los elementos arquitect.	No	Sencilla
AO-Rapide	Extiende Rapide para describir sistemas orientados a aspecto, utilizando el modelo Reo (en desarrollo)	Las interfaces pueden ser proporcionadas, requeridas, action y sevicia	Permite definir tipos arquitectónicos	Permite múltiples vistas	No	Sencilla
AspectualAcme	Extiende Acme. Independiente del modelo arquitectónico	Interfaz en los AC determina cómo es la comunicación entre componentes y aspectos	Como en Acme	Una sola vista	No	-
FAC ADL	Extiende Fractal ADL superponiendo un nivel de aspecto	La interfaz de intercepción relaciona comp. y aspectos. Inter. de negocio: acceso externo los componentes	No permite definir tipos	Una sola vista	No	Sencilla
PiLAR OA	Extiende PiLAR para soportar aspectos, aprovechando sus características reflexivas	Los componentes tienen interfaces en las que se definen las restricciones	No se definen tipos	Meta niveles	No	-
AspectLEDA	Extiende LEDA para soportar aspectos. Las nuevas instrucciones se traducen a LEDA	Se definen roles en los componentes, que se enlazan para comunicarse entre sí	No se pueden definir tipos	Una sola vista	No	Sencilla

Tabla 3.4. Características generales de los LDA OA estudiados y el propuesto.

LDA-OA	*Aspectos*	*Composición*	*Quantification*	*Incons-ciencia*	*Interfaz de Aspecto*	*Exposición de JP*	*Mejora de interfaces*	*Coordinación*
DAOP ADL	Como componentes. Reutilizables.	Mediante un conjunto de reglas Puntos de intercepción y de corte definidos de forma separada a los componentes	No lo soporta	Si	Interfaz evaluada y de eventos	En la interfaz de los componentes de caja negra	No lo soporta	Aspecto de coordinación
Prisma ADL	Un aspecto es un *concern*. Reutilizables. Son parte de los elementos arquitectónicos	En componentes y conectores en la especificación de *weaving*. Externa a aspectos y componentes. Definida en los elementos arquitectónicos y al especificar la configuración	No lo soporta	Si	En la definición de los aspecto	Los aspectos tienen acceso a propiedades de un componente o conector	Aspectos pueden refinar propiedades de componentes y conectores	Aspecto de coordinación
AO-Rapide	Componentes de aspecto. Interfaces Rapide. Existen conectores entre los componentes y los aspectos	A través de un elemento coordinador y un conjunto de reglas. Por conectores e interfaces Rapide. Externa a aspectos y componentes	A nivel de configuración, en los tejedores de los componentes	Si	En los tejedores de aspecto	En la interfaz de los componentes de caja negra. JP y PC externos al componente (wrapper)	No lo soporta	Coordinadores Reo. Elementos independientes
Aspectual Acme	Componentes de aspecto: Representan los CC. Reutilizables	Semántica de composición dirigida por los *aspectual connectors* en la definición de configuración	Expresada en la sección de configuración	Si	Interfaces Acme extendidas	Componentes y conectores pueden exponer sus eventos internos	No lo soporta	En la definición de la configuración
FAC ADL	Define Aspectual component para expresar los CC	Dos tipos de ligadura: Direct binding y crosscut binding (usando pointcuts).Externa a los componentes. Por interfaz de intercepción	Expresada en términos de ligaduras	Si	Interfaz de intercepción permite la ligadura entre aspectos y componentes	Se pueden representar las llamadas entrantes y salientes	No lo soporta	En el *weaving*
PiLAR OA	Fragmento Arquitectónico	Utiliza el concepto de composición (superpostura)	Concepto de aserción de vínculo, para seleccionar los JP, basado en calculo π	Si	Denominada extremas (meta interfaz del meta componente -Aspecto-)	Representado por la aserción de vínculo	-	A través de la relación de superposición definida
Aspect LEDA	Existe un coordinador asociado a cada aspecto, y un MetaCoordinador para todos los coord.	A través de un elemento coordinador y un conjunto de reglas	Mediante parámetros de entrada, reglas y CR en la definición de config.	Si	Interfaz que proporciona el comp. de aspecto	En los Componentes. Componentes de caja negra	No lo soporta	Componentes coordinadores

Tabla 3.5 Representación de conceptos arquitectónicos en los LDA-OA estudiados y el propuesto.

LDA-OA	*Extensibilidad*	*Dominios de aplicación*	*Generación de código*	*Prototipo ejecutable*	*Dinamismo*	*Comunicación interna*	*Herramientas*	*Análisis y validación*
DAOP ADL	Si. Proporcionada por XML	Entornos virtuales colaborativos – distribuidos. Aplicaciones p2p y cliente-servidor	Independiente del lenguaje	Interpretado por la plataforma	Composición dinámica de aspectos	Envío de mensajes a través de las interfaces y lanzamiento de eventos	DAOP ADT y de validación	Si. 2 pasos
Prisma ADL	Parece sencilla pero no necesaria	Sistemas complejos distribuidos	C++ C#	---	Reconfigurable en tiempo de ejecución	A través de los interfaces. Externa a componentes y aspectos	EVG Prisma	Como OASIS
AO-Rapide	No es sencilla	Mismos dominios que Rapide. Independiente del dominio. Amplio rango de sistemas	C/C++ VHDL	Ejecutables en tiempo de diseño	El proporcionado por Rapide	A través de eventos. Hereda de Rapide	No disponibles	Si
Aspectual Acme	No perece necesario	Independiente del dominio. Orientado al DSBC	Hereda de Acme	----	----	A través del interfaz de los AC	Las de Acme de diseño. No de desarrollo arq.	Como Acme
FAC ADL	No parece necesario	Orientado al desarrollo de sistemas basados en componentes	---	---	Binding Dinámico. Tejido en tiempo de ejecución. Reconfig dinámica	A través de las interfaces de intercepción	AOP Aliance	---
PiLAR OA	No perece necesario	Arquitecturas Dinámicas	No	No	Si. Reconfiguración dinámica	Definición de ligaduras (binding) y definición de puertos	No	-
Aspect LEDA	Sencilla, pero no perece necesaria	Debería usarse en los mismos dominios que LEDA	Generación de código Java	Si, mediante EVADeS	Composición dinámica de aspectos	A través de Roles	ASOA Tool/LEDA	Si, como LEDA

Tabla 3.6. Resumen de conceptos en los LDA-OA estudiados.

CAPÍTULO 4
AOSA Model: un modelo arquitectónico orientado a aspectos

El propósito de este capítulo es la definición de un modelo que integre dos aproximaciones: el desarrollo de software basado en componentes (DSBC) y el desarrollo software orientado a aspectos (DSOA). El modelo descrito, AOSA (Aspect Oriented Software Architecture) Model, forma parte del marco de trabajo AOSA Space para la descripción de arquitecturas software de sistemas complejos. En este capítulo se estudia el proceso de desarrollo de los sistemas, durante la fase de diseño arquitectónico, que se extienden con aspectos. Para ello, éstos se consideran componentes arquitectónicos que se insertan en el sistema aplicando reglas de composición.

La estructura del capítulo es la siguiente: en la primera sección se describen los elementos que forman AOSA Model: los componentes y su interacción, así como las características del proceso de tejido; en la sección 2 se describe la estructura meta nivel que define el modelo; y en la sección 3, algunos casos que se pueden distinguir.

4.1. Introducción a *AOSA Model*

AOSA Model es un modelo para la descripción de arquitecturas software de sistemas complejos, tanto en tiempo de especificación y diseño como de mantenimiento y evolución. Sigue el paradigma del DSOA considerando los aspectos como entidades de primera clase a lo largo del desarrollo. En particular, se consideran como componentes arquitectónicos, sin que sea necesario definir nuevas abstracciones para la integración de los aspectos, sino que basta aplicar los conceptos arquitectónicos tradicionales (componentes y conectores) al desarrollo de una arquitectura orientada a aspectos.

En *AOSA Model*, las materias o propiedades de interés que cruzan varios módulos arquitectónicos de un sistema (*crosscutting concerns*) son considerados aspectos y encapsulan su comportamiento. Estos aspectos pueden ser funcionales o no funcionales identificados en etapas tempranas del desarrollo, durante su mantenimiento o la evolución del sistema.

Desde el punto de vista orientado a aspectos, *AOSA Model* sigue el modelo asimétrico (apartado 3.1.3) en el que los aspectos son entidades que se incorporan a un sistema cuya funcionalidad principal ha sido o está siendo definida. Los aspectos van a ser los requisitos transversales a las especificaciones iniciales, que se extraen de la definición inicial o bien se van a incluir como nuevos requisitos. Seguir el modelo asimétrico aporta ventajas, pues resulta más fácil de aplicar a sistemas en fase de diseño o que se han diseñado siguiendo aproximaciones no orientadas a aspectos. Además, se facilita la reutilización de los aspectos en diferentes contextos al definirse independientemente de los componentes del sistema en el que serán insertados. Por el contrario, modelos simétricos sólo pueden ser aplicados a sistemas en fase de diseño o ya diseñados con esa característica respecto de la simetría.

Finalmente, en *AOSA Model* los componentes del sistema se caracterizan tanto por las interfaces que proporcionan como por las que requieren; mientras que los aspectos, como componentes arquitectónicos, se caracterizan sólo por las interfaces que proporcionan. Su interacción con los componentes de la arquitectura es transparente a éstos.

4.1.1. Elementos estructurales. Su interacción

En *AOSA Model*, los elementos estructurales del sistema son componentes, tanto los que representan la funcionalidad básica del *sistema inicial*[19] como los aspectos que se insertarán en él para ampliarlo, completarlo o actualizarlo. Ambos tipos de componentes

[19] En adelante se denomina *sistema inicial, sistema base* o *sistema básico* a un sistema existente (ya diseñado) que se va a extender con nuevos requisitos que se puedan considerar aspectos, o bien a un sistema en fase de especificación en el que sólo se consideran los requisitos que representan su funcionalidad básica, habiéndose extraido aquellos requisitos transversales que se puedan considerar como aspectos, que se incluyen en una iteración posterior. En ambos casos, el sistema que se obtiene al añadir los aspectos se denominará *sistema extendido* o *sistema final.*

se consideran entidades de primer orden. Son entidades encapsuladas de granularidad alta que se pueden componer y vienen definidos por sus interfaces para establecer las relaciones entre ellos.

Componentes

- Los *componentes funcionales* del sistema son cajas negras que contienen la semántica del sistema mediante la definición de su comportamiento, encapsulando su funcionalidad, publicando los servicios que ofrecen a través de sus interfaces y requiriendo, a su vez, los servicios de otros componentes. Se entiende por servicio una operación que ejecuta una serie de acciones en un componente y produce un resultado. Los componentes son independientes entre sí e interaccionan con otros componentes intercambiando mensajes o lanzando eventos.
- Los *componentes de aspecto* definen la estructura y el comportamiento de un cierto *concern* que atraviesa varios componentes arquitectónicos del sistema encapsulando su operabilidad. Estos componentes vienen definidos por una interfaz que contiene los servicios que ofrecen, sin requerir ninguno. Además, un aspecto tiene asociada una semántica que especifica si deben o pueden ejecutarse los servicios que proporciona y cuándo hacerlo. Esta semántica está definida por un conjunto de precondiciones de ejecución. En *AOSA Model*, un aspecto se ejecuta de un modo independiente a las condiciones que definen su comportamiento (que son dependientes del contexto). Como consecuencia, los aspectos son independientes de la arquitectura donde se vayan a integrar y por tanto reutilizables.
  Como la aplicación de los aspectos se realiza sobre componentes caja negra, los puntos de corte tienen que estar localizados en la interfaz pública de dichos componentes. Los aspectos se aplican en virtud de la intercepción de mensajes que envían o reciben los componentes del sistema en el que se insertan.

Interacción

La interacción entre los elementos de la arquitectura se representa de dos formas:

- Conectores definidos entre los (interfaces de los) componentes del *sistema inicial*. Estas conexiones permanecen igual al extender el sistema con aspectos, salvo aquellos conectores asociados a los puntos donde se realizarán las inserciones que se rompen teniendo que definirse una nueva interacción.
- Nuevas conexiones que se tienen que establecer para relacionar los componentes del *sistema inicial* con los de aspecto. La relación de estos componentes con los del sistema se lleva a cabo mediante la especificación de una interacción compleja que define una relación de composición (proceso de tejido o *weaving*) e incluye varios elementos (componentes y conectores). Esta composición lleva a definir un conjunto de reglas, que se explican en el apartado 4.3.1.3.

4.1.2. Proceso de tejido

- El *proceso de tejido* o *weaving* en *AOSA Model* se define del siguiente modo:
  - Un punto definido en la interfaz de un componente al que se asocie un aspecto es un punto de unión o punto de enlace (*join point*).
  - Un conjunto de puntos de unión se denomina punto de corte (*pointcut*)*;* los puntos de corte permiten definir predicados o condiciones que se asocian a los puntos de unión. De este modo, los puntos de corte representan las condiciones que se tienen que satisfacer en los puntos de unión para que los aspectos se ejecuten.
  - El comportamiento asignado a los componentes de aspecto son los denominados *advices*. También indican el momento en el que debe ejecutarse ese aspecto respecto al punto de corte.

    AOSA Model considera tres tipos de tejido:
    - *After*: el servicio que define el comportamiento del aspecto se ejecuta *después* de la acción asociada al punto de enlace.
    - *Before*: el servicio del aspecto se ejecuta *antes* que la acción asociada al punto de enlace.
    - *Around*: el servicio del aspecto se ejecuta *en lugar de* la acción asociada al punto de enlace.
- Las *condiciones de tejido* se especifican fuera de los componentes y de los aspectos, para preservar su independencia del contexto. Esta especificación determina cuándo la detección de un evento sobre un componente debe disparar la ejecución de un aspecto. Un mismo evento puede disparar la ejecución de más de un aspecto.
- La *semántica del tejido* de cada aspecto viene definida por un componente *coordinador* que propone el modelo. Este componente intercepta la invocación de un servicio y, en tiempo de ejecución, determina si se cumplen las condiciones para la ejecución del aspecto al que está asociado, sincronizando la ejecución de los elementos que comunica.

  Como consecuencia, y según se ha mencionado:
  - Los aspectos son independientes del contexto y reutilizables.
  - Los elementos *coordinadores* son dependientes del contexto del sistema en el que los aspectos se van a insertar. Su estructura y características las determina el arquitecto del software al especificar las características del *sistema extendido*.
- La *composición* se realiza dinámicamente en tiempo de ejecución en función de las reglas que definen el comportamiento del aspecto en el contexto, pero de forma externa a él. Componentes y aspectos se mantienen siempre independientes, no hay una composición previa de ambas entidades antes de la ejecución del sistema. El componente *coordinador* gestiona esa composición al conocer en tiempo de ejecución el valor de las condiciones que determinan la ejecución del aspecto.

  La composición de más de un aspecto en un mismo punto de corte hace las relaciones más complejas, debiendo determinarse la prioridad de ejecución de los aspectos

cuando más de uno actúa, o debe actuar, sobre el mismo punto. Es responsabilidad del ingeniero de software establecer adecuadamente la prioridad en cada caso. Una asignación errónea de prioridad puede provocar resultados no deseados en la ejecución final del sistema.

- En esta propuesta se supone que los aspectos que se aplican sobre un mismo punto de corte son independientes entre sí.
- Además, *la separación de aspectos*, durante el diseño arquitectónico, *se gestiona como un problema de coordinación*, habiéndose considerado un modelo de coordinación exógeno como es Coordinated Roles [Mur01]. En la siguiente sección se justifica esta afirmación y se describe someramente el problema de coordinación así como algunos modelos considerados.

4.2. Separación de aspectos a nivel arquitectónico: un problema de coordinación

Una de las características distintivas de *AOSA Model* y que constituye una de las principales aportaciones de esta tesis doctoral es que *el problema de la separación de aspectos, a nivel de arquitectura, se aborda como uno de coordinación*.

En efecto, en [Nav+02] presentamos algunas consideraciones metodológicas sobre cómo afrontar la separación de aspectos en el diseño arquitectónico abordando el problema desde un punto de vista estructural. El estudio concluye que

> *la separación de aspectos, a nivel arquitectónico, tiene algunas similitudes con los problemas de coordinación, que se resuelven aplicando modelos de coordinación.*

Esta relación se hace patente si se consideran los siguientes puntos:

- Por una parte, la funcionalidad de los sistemas se representa, a nivel de arquitectura, como un conjunto de componentes y la interacción entre ellos mediante conectores arquitectónicos.
- Por otra parte, la especificación de los aspectos identificados en el sistema es transversal a la especificación de los componentes, por lo que la interacción entre ambos elementos tiene que tratarse de un modo diferente. Aplicando el paradigma de la separación de aspectos y extrayendo los aspectos de la funcionalidad básica del sistema surgen nuevas conexiones entre los componentes y los aspectos extraídos. Establecer la nueva interacción supone:
  - Identificar en los componentes los puntos en los que se han extraído, o se van a insertar, los aspectos: especificación de los puntos de enlace (*join point*).
  - Especificar las conexiones entre los aspectos y los puntos de enlace. Estos conectores han de describir, además, las condiciones relativas a la aplicación de cada aspecto (cuándo y cómo debe aplicarse).

Estos nuevos conectores son los que gestionan la ejecución del sistema con los aspectos insertados. Sin embargo, y como se menciona en [Nav+02], *éste es un problema típico de coordinación* y se sugiere su uso para gestionar la interacción entre componentes funcionales y de aspecto.

En particular los modelos de coordinación exógenos [Arb96, Mur+99] especifican el código para determinar cómo componentes funcionales coordinan desde entidades separadas aquellas otras que van a ser coordinadas. En estas propuestas, unos *componentes coordinadores* gestionan la ejecución global del sistema.

El mismo esquema se puede usar en el contexto de la orientación a aspectos para especificar conectores entre aspectos y componentes. En este caso, hay dos tipos de componentes: funcionales y de aspecto; *los conectores entre ellos serían los componentes de coordinación*. La función de éstos es especificar cómo y cuándo los componentes de aspecto deben ser tratados.

Por su relación con el modelo que aquí se propone, en los próximos párrafos se resume el concepto de *coordinación* y de *modelos de coordinación*, con objeto de justificar la elección de uno de ellos para resolver el problema de la separación de aspectos a nivel arquitectónico. En concreto se resume *Coordinated Roles*, modelo en el que se apoya *AOSA Model*.

Modelos de coordinación

El término *coordinación* se utiliza para designar a un tipo de modelos, formalismos y mecanismos para la descripción de actividades concurrentes y distribuidas. Básicamente la coordinación se logra mediante comunicación generativa, a través de un espacio de datos compartido [Gel85] o manipulando de manera dinámica las interconexiones entre procesos, como consecuencia de la observación de un cambio de estado [ArHeSp93].

En el ámbito que nos ocupa, hay una proximidad conceptual entre *coordinación* y *arquitectura software* y parece natural comparar los modelos que representan. Se pueden considerar análogos en muchos sentidos [Cue+02] y a veces han sido descritos en ambos contextos. De este modo, muchos modelos propuestos originalmente para coordinación son igualmente aplicables a AS y viceversa.

Se pueden distinguir dos tipos de modelos principales de coordinación, atendiendo al criterio de la forma en la que se expresan las restricciones de coordinación: *modelos de coordinación endógenos* y *modelos exógenos*. Aunque se puede hacer un estudio mucho más extenso de los modelos de coordinación, aquí sólo se comenta brevemente por su relación con la AS y la separación de aspectos.

Modelos endógenos

Los modelos de este tipo proporcionan primitivas que han de ser integradas dentro del código funcional para su coordinación. Estos modelos están basados en la comunicación generativa que se define como una comunicación asíncrona entre procesos basada en una estructura de datos compartida. Se caracteriza por el hecho de que cuando

un proceso inicia una comunicación, no sabe ni a quién se dirige ni cómo llegar hasta él. Es el propio mensaje el que sabe encontrar (*generar*) este camino. La propuesta más representativa dentro de este grupo es *Linda* [Gel85].

> ***Linda*** se basa en la definición de un *espacio de tuplas* que hace de medio de coordinación, y es una estructura de datos compartida que contiene registros de múltiples campos que se denominan *tuplas*; estas pueden ser pasivas (datos) o activas (procesos). Las *tuplas pasivas* son secuencias finitas de campos; el acceso se realiza mediante un mecanismo de correspondencia de patrones. Las *tuplas activas* se dan cuando alguno de los campos es una llamada a función. Además, el modelo propone la definición de cuatro primitivas que manipulan las tuplas del espacio; las primitivas permiten *crear* una *tupla* en el *espacio de tuplas*, *eliminarla*, *hacer una lectura sin eliminación* y *crear una tupla activa*. Como las operaciones son atómicas, no se pueden producir inconsistencias en el *espacio de tuplas*. Al ser de naturaleza no determinista, permite simular el determinismo, nombrando a los procesos involucrados. Actualmente *Linda* dispone de una semántica basada en el álgebra de procesos [CiGoZa96], aunque hay varias versiones que amplían el modelo.

Modelos exógenos

Más interesante en relación al tema de esta tesis doctoral y en particular con el modelo que se describe en este capítulo son los modelos exógenos. Éstos proporcionan primitivas que soportan la coordinación de entidades desde fuera de las que han de coordinarse. En estos modelos aparece una entidad coordinadora que establece las políticas de coordinación a aplicar sobre las entidades que se tienen que coordinar. *Manifold* [PaAr98] es quizás el modelo más citado dentro de este tipo; se basa en el modelo de comunicación IWIM:

> El **Modelo IWIM** (*Idealized Worker –Idealized Manager**)*** [Arb96] es un modelo abstracto de comunicación diseñado para construir sistemas en los que haya poca dependencia entre los procesos. Pretende emular una situación en la que cada proceso intenta cumplir su tarea sin preocuparse de la organización. Además, una serie de gestores se encargan de manipular las relaciones entre los procesos, es decir, sólo se ocupan de la organización. Este modelo hace posible la separación estricta entre computación y coordinación, y que dos procesos puedan comunicarse sin saber nada el uno del otro (*comunicación anónima*). La interacción se realiza por difusión de eventos o mediante un canal punto a punto establecido por un gestor.
>
> Por su parte, ***Manifold*** [ArHeSp93, PaAr98] es un lenguaje desarrollado específicamente para construir los gestores del modelo. Éste es un lenguaje fuertemente tipado, estructurado en bloques y orientado a eventos. Sólo reconoce cuatro tipos de entidad: *procesos* (son los trabajadores -*workers*), *puertos* (que expresan sus entradas y salidas), *eventos* (que son la base del sistema) y *flujos* – *streams* (definidos como canales privados entre los puertos de dos procesos). No tiene estructura de control (al menos inicialmente) por lo que ha de realizar toda su tarea mediante una dinámica de transición de estados dirigida por eventos, en la que se crean con total libertad nuevos procesos y conexiones entre ellos. De este modo,

en respuesta a cada evento se produce una reconfiguración en el sistema especificada por uno de los gestores, denominados *coordinadores*. En *Manifold*, un proceso es una caja negra con puertos. Desde el punto de vista de un proceso no es posible distinguir a un *coordinador* de un trabajador –*worker*-; por tanto, un gestor –*coordinador*- puede estar coordinando a otros gestores; es decir, actuando como meta-coordinador. Así se pueden llegar a establecer protocolos de gran nivel de complejidad y a múltiples niveles de abstracción.

Coordinated Roles [Mur+99, Mur01] es un modelo de coordinación que también sigue IWIM que propugna la separación de los aspectos funcionales y de coordinación dentro de un sistema. Parte del concepto de sistema software coordinado como

> *un conjunto de componentes que desarrollan funciones determinadas y que son controlados por un conjunto de componentes que realizan las funciones de coordinación del sistema*.

Los componentes encargados de la coordinación –*coordinadores*- describen un patrón de coordinación en el que se definen los distintos roles que pueden desempeñar los componentes del sistema. Cuando un componente va a ser controlado por un *coordinador* debe adoptar uno de los roles definidos en éste. Los componentes son controlados por los *coordinadores* gracias a los *Protocolos de Notificación de Eventos*.

Mediante estos protocolos, un *coordinador* puede tener constancia de los eventos que se producen en los objetos que coordina. Cada protocolo puede ser solicitado de forma síncrona o asíncrona y está asociado a uno de cuatro posibles tipos de evento: recepción de un mensaje (*RM*), comienzo de procesamiento (*BoP*), fin de procesamiento (*EoP*) o consecución de un estado (*RS*) por parte del objeto coordinado. Mediante este mecanismo, el *coordinador* determina las acciones a realizar por los objetos coordinados cuando reciba la notificación de los eventos asociados a los roles que define, y que son desempeñados por los objetos coordinados.

En el *coordinador* no hay ninguna referencia a componentes concretos, sino que es en tiempo de ejecución cuando se asocian los componentes con los roles definidos en un determinado *coordinador*. Igualmente, en los componentes que deben coordinarse no existen acciones explícitas de comunicación con el *coordinador*. La coordinación es transparente a los objetos coordinados. Esto permite que un determinado componente pueda jugar varios roles con distintos *coordinadores*, y que, a su vez, diferentes instancias de la definición de un *coordinador* se puedan utilizar para controlar grupos diferentes de componentes que responden a un mismo patrón de coordinación. De esta forma, gracias a que ni los *coordinadores* ni los componentes coordinados se hacen referencia unos a otros, se consigue que sean totalmente reutilizables.

Del estudio de los modelos de coordinación en general, y de Coordinated Roles en particular, se puede concluir que *la separación de aspectos durante el diseño arquitectónico se puede gestionar como un problema de coordinación*, desde el punto de vista de los modelos exógenos, en el que una entidad coordinadora realizaría las tareas de coordinar la ejecución de los aspectos identificados en un cierto sistema cuando se

integran en él. De este modo, se ha definido en *AOSA Model* una estructura arquitectónica de dos niveles en la que, como se explica en la siguiente sección, se mantienen separados, por una parte los componentes que constituyen el sistema y su interacción; y por otra, los aspectos, las entidades coordinadoras y las nuevas interacciones.

4.3. Estructura meta nivel

En esta sección se describe la estructura meta nivel que se ha definido para *AOSA Model*. Se describe su estructura y las conexiones entre los elementos de la arquitectura que se propone. En *AOSA Model* se define la arquitectura de un sistema en dos niveles:

- El *nivel base* o *nivel de componente* formado por los componentes y las interacciones entre ellos que definen el *sistema inicial*, en el que se van a insertar los aspectos.
- El *meta nivel* o *nivel de aspecto* contiene los componentes de aspecto a ser insertados así como los elementos que propone *AOSA Model* para definir la nueva interacción: componentes *coordinadores* y un *MetaCoordinador* o *SuperCoordinador*, y la interacción entre ellos.

La estructura arquitectónica propuesta por *AOSA Model* se muestra en Figura 4.1. En ella se representa esquemáticamente la interacción compleja entre el *sistema inicial*, en el denominado *nivel base* y los aspectos, en el *meta nivel* (sólo aparecen los componentes de aspecto).

La estructura meta nivel de la arquitectura se explica en los apartados siguientes. Sin embargo, en Figura 4.2 se representan, de un modo detallado, los elementos del *nivel de aspecto*. En ella se muestra la interacción representada en Figura 4.1 como un conector, haciéndose explícitos ahora los componentes *coordinadores* que gestionan esa comunicación.

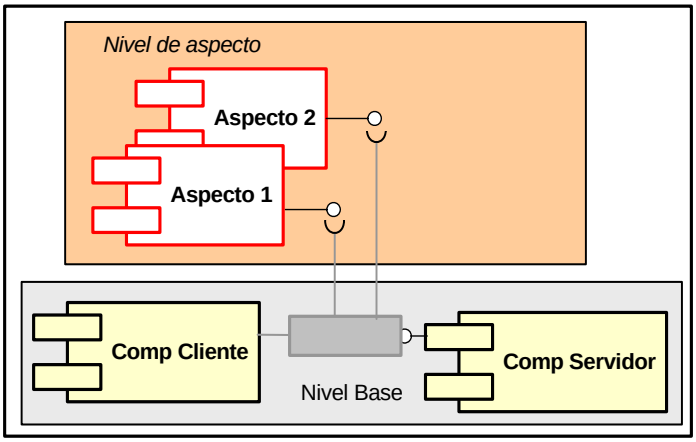


Figura 4.1. Estructura arquitectónica de *AOSA Model.* Representación esquemática.

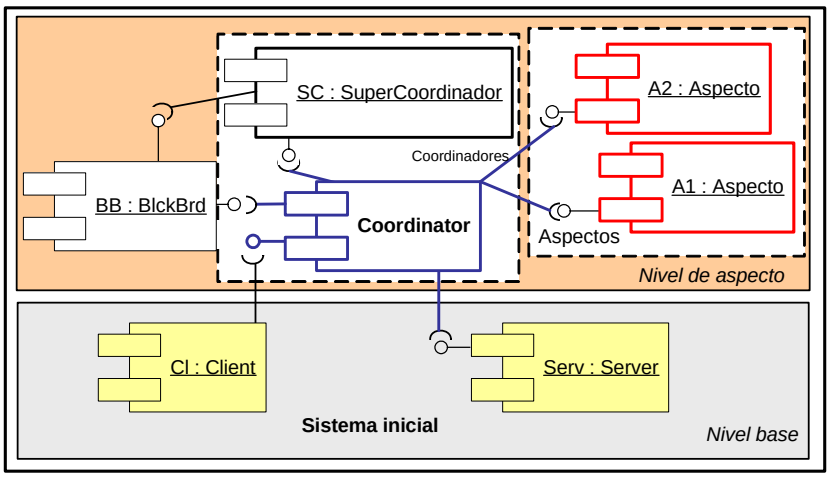


Figura 4.2. Representación detallada del nivel de aspecto.

Se ha diseñado una disposición en estrella para representar las nuevas interacciones y los elementos del *nivel de aspecto*. En esta representación, el centro es el elemento *coordinador* siendo el que gestiona la ejecución de todos los componentes implicados en la inclusión de cada aspecto.

La configuración final del sistema viene definida por los componentes del mismo y las asociaciones entre ellos, que establecen las conexiones entre los elementos arquitectónicos. El lenguaje en el que se represente la estructura arquitectónica del sistema establecerá de modo más concreto estas asociaciones.

En cualquier caso, *AOSA Model* propone que la estructura de los *sistemas extendidos* sea siempre igual, que de un modo general se representa de la siguiente forma:

$$\mathcal{S} = \{ \mathcal{S}_i, \mathcal{CA}, \mathcal{Co}, \mathcal{BB}, \mathcal{I} \}$$

Donde:

$\mathcal{S}_i$, $\mathcal{CA}$, $\mathcal{Co}$ y $\mathcal{BB}$, son los componentes que constituyen la definición estructural,

$\mathcal{I} = \{ \mathcal{I}_i, \mathcal{I}_e \}$ representan las interacciones.

A continuación se describen estos elementos:

A Definición estructural

La definición estructural de un *sistema extendido* con aspectos está formada por los siguientes tipos de componentes fundamentales:

A1) Un componente compuesto, *sistema inicial* ($\mathcal{S}_i$) que está formado, a su vez, por el conjunto de componentes que lo constituyen y las conexiones entre ellos. Así, la arquitectura del sistema se representa genéricamente como [ShGa96]:

$\mathcal{S}_i = \{ \mathcal{Comp}, \mathcal{I}_i \}$. Siendo:

$Comp = \{ C \}$ y $C = \{ \mathcal{P}, \mathcal{R}, \mathcal{At} \}$ son los componentes de diseño, donde:

$\mathcal{P}$ representa la interfaz que proporciona,

$\mathcal{R}$ es la interfaz que requiere, y

$\mathcal{At}$ son los atributos o parámetros.

$I_i = \{ (C_i, C_j) \}$ es el conjunto de las interacciones entre dos componentes C_i y C_j, que desempeñan los roles de cliente y servidor respectivamente en esa interacción. Éstas son las relaciones establecidas entre los componentes que se describen en la especificación del *sistema inicial* y expresadas en el diagrama de casos de uso y los diagramas de secuencia.

El modelo supone que este *sistema inicial* se especifica (o ha sido especificado) antes de superponer los aspectos.

A2) Un conjunto de componentes de aspecto ($\mathcal{CA}$) a insertar en el *sistema inicial*.

$\mathcal{CA} = \{ \mathcal{A} \}$ y $\mathcal{A} = \{ \mathcal{P}, \mathcal{At} \}$ donde:

$\mathcal{P}$ representa la interfaz que proporciona, y

$\mathcal{At}$, son los atributos o parámetros.

La interfaz de los componentes de aspecto que se incorporan se define en función sólo de las operaciones que proporciona ($\mathcal{P}$).

A3) Los componentes coordinadores ($\mathcal{Co}$) que gestionan la interacción entre los componentes del *sistema inicial* y los componentes de aspecto.

A4) $\mathcal{BB}$ es un componente que se ha denominado *BlackBoard* y proporciona los servicios de leer y escribir en una pizarra a la que acceden los *Coordinadores* y el *MetaCoordinador*. En ella se escriben, en tiempo de ejecución, fundamentalmente, el valor de la condición que ha de evaluarse para determinar si el aspecto se ejecutará o no.

La descripción detallada de estos componentes se realiza en el siguiente apartado.

B Definición de la interacción

La interacción I entre los elementos estructurales tiene que venir expresada en la definición de configuración. Esto es así pues la información que permite establecer las nuevas relaciones tiene que formar parte de la especificación del sistema en extensión. Por tanto, además de las conexiones definidas entre los componentes del *sistema inicial* (I_i) es necesario definir nuevas conexiones (I_e) entre (Figura 4.2):

B1) Los componentes del *sistema base* (C_i) en los que se haya definido un *join point* y el componente *coordinador* que tiene asociado el o los aspectos a insertar en ese punto ($\mathcal{Co}_i$).

B2) El *coordinador* ($\mathcal{Co}_i$) y el o los aspectos ($\mathcal{A}_i$) para el/los que se define.

B3) El componente *MetCoordinador* ($\mathcal{Co}_s$) y cada *coordinador* ($\mathcal{Co}_i$).

B4) El *BlackBoard* y cada *coordinador* (Co_i) y el *SuperCoordinador* (Co_s) a los que proporciona sus servicios.

De modo que la nueva interacción se puede expresar como:

$$I_e = \{ (C_i, Co_i), (Co_i, \mathcal{A}_i), (Co_s, Co_i), (\mathcal{BB}, Co_i), (\mathcal{BB}, Co_s) \}$$

Con esta estructura, el meta nivel actúa sobre el *nivel base* de modo que el comportamiento del *sistema inicial* resulta modificado. La ejecución de los servicios de los elementos del meta nivel se ve reflejado en el comportamiento del sistema sobre el que actúa. Esta organización da al sistema reconfiguración *ad hoc* en tiempo de diseño y una reconfiguración dinámica de la arquitectura.

4.3.1. Descripción detallada del meta nivel

En los siguientes apartados se describe cada uno de estos elementos que forman el *nivel de aspecto o meta nivel* (Figura 4.3):

- El conjunto de componentes de aspecto $\mathcal{CA}$.
- Una estructura estática a la que se ha denominado *Common Items –CI-*.
- Un conjunto de reglas *R* que especifican la interacción entre el *nivel base* y el *meta*.
- El *proceso de control*.
- El componente *BlackBoard –$\mathcal{BB}$-*.

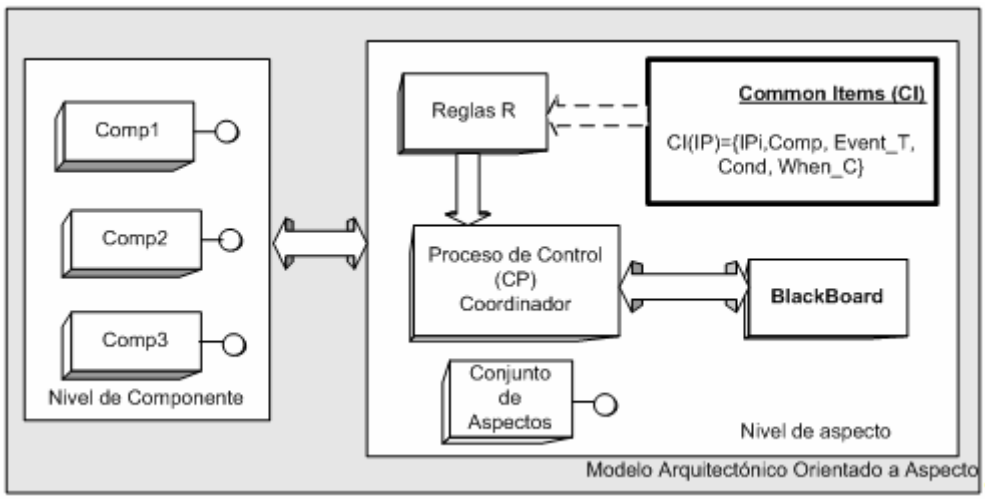


Figura 4.3. Elementos del nivel de aspecto.

4.3.1.1. Conjunto de aspectos CA

Los aspectos a insertar en el sistema se describen como componentes de diseño que tienen una interfaz con los servicios que proporcionan y determinan su operatividad. Además, podrían realizar operaciones internamente, así como tener algunos atributos.

Cómo se realiza la inserción de los aspectos en el *sistema inicial* se muestra detalladamente en el Capítulo 5. Sin embargo, para comprender la descripción de los elementos que constituyen el modelo, es necesario enunciar siquiera unas notas:

- Los aspectos se incluyen en el sistema que se va a ampliar desde la fase de especificación, considerándolos como un tipo especial de caso de uso: *use case extension* [JaNg05], que se relacionan con los casos de uso base a través de los puntos de extensión (*extension points* [JaNg05]).
- Para realizar posteriormente de una manera efectiva la inclusión de los aspectos es necesario almacenar toda la información relativa a éstos y a su ejecución (tipo de evento, condiciones de ejecución y cuándo debe aplicarse). Esta información forma parte de la documentación asociada al caso de uso que se extiende y está formada por:
  - nombre del caso de uso que se extiende,
  - nombre del caso de uso extendido (*use case extension*) y su descripción,
  - relación de *extension points* identificados,
  - tipo de evento que dispara la ejecución del aspecto,
  - condiciones que deben satisfacerse para su ejecución, y
  - cuándo debe ejecutarse cada aspecto.

Concepto de Insertion Point

Al desarrollar los casos de uso del sistema en estudio en los correspondientes diagramas de secuencia, cada *extension point* identificado permite definir una relación entre el componente de diseño que ejecuta la operación que representa el *extension point* y el aspecto que la intercepta. Así, se define un *Insertion Point* por cada terna:

{*Extension Point*, Aspecto, Componente de diseño}

Este concepto es cercano al de punto de unión (*join point)*, pero de más alto nivel.

En capítulos posteriores se desarrolla un ejemplo que facilita la comprensión del modelo. Sin embargo, se ha considerado oportuno describir primero los elementos que constituyen el nivel de aspecto.

4.3.1.2. Estructura Common Items -CI-

Common Items es una estructura que contiene la información que se necesita conocer para obtener la especificación y el diseño del *sistema extendido.* Esta información se deduce durante el proceso de construcción del nuevo sistema; en ella se

reflejan las asociaciones entre los componentes del *sistema base* y de aspecto, así como sus condiciones de aplicación. La estructura *CI* contiene una fila por cada *Insertion Point* y cada una de ellas contiene la siguiente información (Tabla 4.1):

- *Insertion Point –IP-:* representa una operación definida en la interfaz de un componente de diseño que va a ser interceptada por el aspecto.
- *Extensión Point –ExP-*: punto de extensión identificado en un caso de uso y que permite la definición del *IP*.
- *Componente Servidor -Comp-*: nombre del componente que proporciona los servicios asociados al *IP* anterior.
- *Nombre del Aspecto -Asp_C-*: nombre del componente de aspecto asociado.
- *Operación del Aspecto -Aspect_Op-*: es el nombre de la operación que realiza el componente de aspecto.
- *Evento*: se refiere al tipo de evento que dispara la aplicación del aspecto. El tipo de evento considerado es *recibir un mensaje síncrono o asíncrono*.
- *Condiciones de Aplicación -Cond-*: para la aplicación de los aspectos suele ser necesario considerar el cumplimiento de ciertas condiciones. En este caso, la ejecución del componente de aspecto depende de si en el momento de su ejecución se satisfacen o no las condiciones. Puede tomar los valores *true* o *false*.
- *Cláusula when*: indica cuándo el aspecto puede o debe aplicarse. Sus valores posibles son *antes*, *después* o *en lugar de* (*before, after* o *around*) la ejecución de la operación que disparó el evento asociado.
- *Componente Cliente -C_Cli-*: nombre del componente que requiere los servicios asociados al *IP* anterior.

Elemento	Descripción
Insertion Point – *IP*	Cada punto de corte identificado en el diagrama de secuencia para un componente de diseño y un aspecto. Similar a *Join Point* en AOP (*IP*).
Extension Point	Punto de inserción identificado en el caso de uso (*ExP*).
Componente Serv	Nombre del componente que proporciona los servicios asociados al IP (*Comp*).
Aspect Name	Especifica el nombre del aspecto asociado a esta fila de *Common Items (Asp_C)*.
Aspect Operation	Nombre de la operación que ejecuta el aspecto a aplicar (*Aspect_Op*).
Event Type	Tipo de evento que dispara la aplicación del aspecto (*ev*).
Application Condition	Condición que debe de satisfacerse para permitir la ejecución del aspecto (*Cond*).
When Clause	Cuándo el aspecto debe aplicarse (*When_C*).
Componente Cliente	Nombre del componente que requiere los servicios asociados al IP (*C_Cli*).

Tabla 4.1. Elementos de cada fila de la estructura *Common Items*.

Así, de la especificación detallada del sistema (expresada de esta manera) se puede obtener su estructura completamente definida. El diseñador deduce esta información cuando estudia las características de los aspectos a añadir y los diagramas de secuencia. En el Capítulo 5 se describe el procedimiento para lograrlo.

4.3.1.3. Conjunto de reglas

Al describir la arquitectura del *sistema extendido* se define un conjunto de reglas que describen las características de la interacción entre el/los componente/s de aspecto, en el nivel de aspecto, y los componentes del nivel inferior. Las reglas indican las acciones a ejecutar para aplicar los aspectos, considerando las condiciones que se tienen que cumplir. Las reglas se deducen de la información de la estructura *CI*:

- Para cada fila de *CI* se define una regla.
- La parte izquierda de las reglas especifica las condiciones que se tienen que cumplir para ejecutar las acciones de la parte derecha (y que se deducen de la información que define cada aspecto).
- La expresión canónica de una regla es:

```
IF ev | IP of Comp | Cond THEN
        DO Aspect_Op of Asp_C WHEN When_C
```

Siendo **ev**, **IP**, **Comp**, **Cond**, **Aspect_Op**, **Asp_C** y **When_C** valores de una fila de la estructura *CI*.

4.3.1.4. Proceso de control

El *proceso de control* (*CP*) gestiona la ejecución coordinada del *sistema extendido* considerando la información de las reglas. En concreto, *CP* coordina la ejecución de los componentes afectados por los eventos que se han disparado y cada aspecto asociado a ellos. Para afrontar el problema de coordinar la ejecución de los aspectos y los componentes del *sistema inicial,* según se comentó en 4.2, se ha utilizado un *modelo de coordinación,* que resuelve el problema de realizar la ejecución coordinada de los componentes del *sistema inicial* y los aspectos incorporados. Además, las interacciones que se tienen que establecer entre el *sistema inicial* y el *meta nivel* no son triviales, sino que representan un protocolo de comunicación compleja entre ellos.

Teniendo en cuenta estos argumentos, se ha decidido considerar el modelo *Coordinated Roles* (*CR*) [Mur+99] que es un modelo de coordinación exógeno y orientado a control. Se ha elegido *CR* (frente a otros modelos de coordinación) porque éste se adapta a las características de *AOSA Model:*

> *CR* se basa en protocolos de notificación de eventos, y hace posible que un componente coordinador "pregunte" por la ocurrencia de un evento sobre un componente del sistema a ser coordinado con un cierto aspecto. Así, cada coordinador está monitorizando el sistema de modo inconsciente (transparente) a los componentes. Cuando un coordinador (en el sentido de *CR*) detecta la ocurrencia de un evento, ejecuta las acciones de coordinación correspondientes. Los eventos pueden tratarse en un modo síncrono (el componente afectado por el evento queda bloqueado hasta que dicho evento ha sido convenientemente

tratado) o en modo asíncrono (el componente afectado continúa su ejecución mientras el evento está siendo tratado).

Dado que el *sistema extendido* puede ser complejo debido al número de aspectos a incluir, la definición de *CP* está formada por:

- Varios coordinadores (en el sentido de CR). Se define un coordinador para cada componente de diseño que tiene un *Insertion Point*. Por su parte, un coordinador puede gestionar varios aspectos asociados a un mismo IP. El comportamiento de un componente coordinador (apartado 4.3.1.6.) viene definido por las reglas que son chequeadas por otro componente: *MetaCoordinador*.
- El *MetaCoordinador* o *SuperCoordinador* se define como un componente arquitectónico, único para todo el sistema; realmente es un coordinador de coordinadores; su definición es necesaria, entre otras razones, para facilitar la ejecución coordinada de los coordinadores cuando se incluyen varios aspectos que se aplican al mismo o a diferentes puntos de inserción (*Insertion Points*).

4.3.1.5. BlackBoard

Este componente gestiona, en tiempo de ejecución, el almacenamiento, en una pizarra, de la información asociada a los eventos disparados y que aún no han sido tratados. La información que se almacena para cada evento es:

- El tipo de evento (*ev*).
- El punto de inserción (*IP*).
- Nombre del componente afectado por el *IP* -componente receptor del evento- (*Comp*).
- El o los valores de las condiciones que figuran en la parte izquierda de las reglas (*Cond*).

En tiempo de ejecución esta información se compara con aquella guardada en la estructura *CI*.

Cuando un *coordinador* detecta un evento, escribe su información asociada en la pizarra de este componente *(BB)*. Luego, el *SuperCoordinador* lee esta información para compararla con la información de los *CI* que define las Reglas. Después de que un evento haya sido tratado, el mismo *coordinador* que lo apuntó borra de la pizarra del componente *BlackBoard* la información correspondiente.

4.3.1.6. Descripción detallada de los componentes del meta nivel

En este apartado se describen los componentes de diseño del *meta nivel* o *nivel de aspecto*. Dado que el modelo es una propuesta para especificar arquitecturas software, sólo se hace la descripción de los componentes a alto nivel. Esto supone que sólo es necesaria la definición de la interfaz pública de cada componente, ya que lo que se desea es conocer su comportamiento, no codificarlo. Por otra parte, si se quisiera reutilizar componentes, esta descripción bastaría (o debería bastar) para localizarlos. Por tanto, en este punto se describen los componentes de diseño a alto nivel.

Para dotar de mayor expresividad a los componentes del modelo, aún siendo genérico, se considera la propiedad estado, que puede tomar los valores sin estado o con estado, siguiendo la idea de modelos de componentes empresariales como EJB o CCM. Por defecto, si no se indica nada, los componentes son sin estado.

Descripción de los componentes Cliente y Servidor

En los siguientes párrafos se describen los componentes del *sistema inicial* que, si bien no forman parte del *meta nivel*, sí interactúan con él. En este sentido, conviene decir que para incorporar nuevos requisitos a un sistema existente, y que se puedan considerar como aspectos arquitectónicos, el modelo sigue la filosofía cliente-servidor (Figura 4.4a): un componente de diseño se considera un componente cliente si requiere servicios proporcionados por otro que ejecuta el rol de servidor.

Figura 4.4a representa varios componentes relacionados entre sí de un cierto sistema ya diseñado o en fase de diseño -*sistema inicial*-. Entre ellos interesa considerar los componentes etiquetados como *Cl:Cliente* y *Serv:Servidor,* que representan a dos componentes del *sistema inicial* que tendrán alguna relación con los aspectos a añadir. Estos componentes de diseño forman parte del nivel de componente -o *nivel base*- definido en el modelo. Pueden ser reutilizados o de nueva creación, siendo la definición de las operaciones que necesitan o requieren independientes del sistema en el que se integren.

Cuando se extiende el sistema, la asociación entre los componentes cliente y servidor (*Cl:Cliente* y *Serv:Servidor*) resulta interceptada para incluir uno o varios aspectos. Un componente *coordinador* intercepta dicha comunicación. Esta intercepción es transparente a los componentes cliente y servidor. Figura 4.4b representa parcialmente el *sistema extendido* (sólo aparecen los componentes implicados directamente en la extensión). En ella se representan los nuevos componentes según el modelo propuesto:

- Un componente *coordinador* que intercepta la comunicación entre el componente que solicita un servicio –*Cliente*- y el que lo proporciona –*Servidor*-. El nuevo componente –*coordinador*- determina si el aspecto se ejecutará o no. De esta forma, los componentes Cliente y Servidor ignoran la existencia del *coordinador*.
- El componente de aspecto que origina la extensión.
- El componente *MetaCoordinador* o *SuperCoordinador* (*SC*).

En Figura 4.4b se muestra una representación canónica de los elementos que se incluyen. Sólo se pretende mostrar los nuevos elementos y sus interconexiones cuando se añade un aspecto. La inclusión de más aspectos supone la definición de nuevos componentes coordinadores y nuevas interconexiones.

Figura 4.5 representa una descripción genérica, en pseudicódigo, de un componente cliente y Figura 4.6, la de un componente servidor.

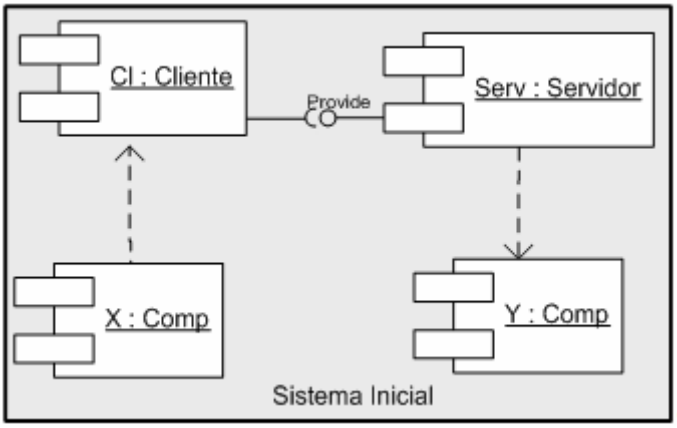


Figura 4.4a). Relación Cliente-Servidor. Sistema Inicial.

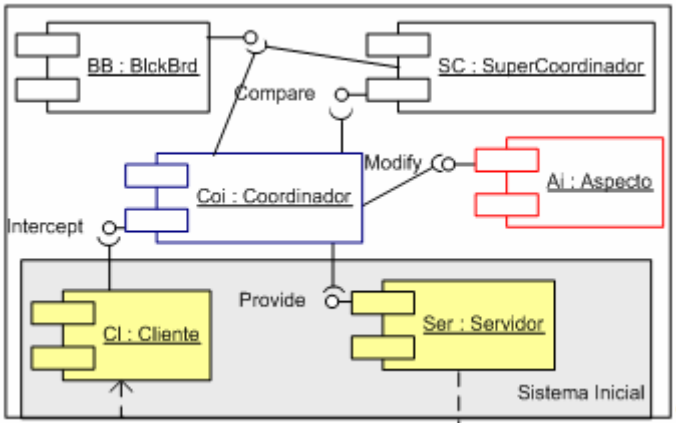


Figura 4.4b). Relación Cliente-Servidor. *Sistema Extendido.* Inclusión de un aspecto.

```
Component Cli       // Cli: Nombre del componente
Begin
… …
Send Op.serv(resp) // Op operación requerida por el cliente
… …
Send OtherOp       // OtherOp: otras operaciones que requiere
… …
End
```

Figura 4.5. Representación en pseudocódigo de un componente cliente.

```
Component Serv            // Serv: Nombre del componente
Begin
… …
When Op.serv(R) do Op(R) // Op operación que proporciona su
interfaz
… …
When OtherOp.serv do OtherOp // OtherOp: otras operaciones
que proporciona
… …
End
```

Figura 4.6. Representación en pseudocódigo para un componente servidor.

Descripción del componente Coordinador

Un componente *coordinador* se define asociado a uno o más aspectos, a un componente cliente y a otro servidor. Realmente, un *coordinador* se define asociado a un evento que es disparado por un componente *cliente* (por ejemplo solicitar un servicio) o por un componente *servidor* (ejecutar un servicio). Cuando se dispara el evento, es posible que se ejecute el aspecto al que se asocia; la ejecución depende de que se satisfagan ciertas condiciones. Por otra parte, el comportamiento del *coordinador* varía si el evento disparado es síncrono o asíncrono, según se explica a continuación.

Cuando el sistema en extensión ha sido completamente especificado, toda la información de interés está en la estructura *CI* y en el conjunto de *reglas*. A partir de ellos se establece el comportamiento del *coordinador* que, además de aplicar los protocolos indicados, ejecuta las reglas bajo las condiciones de tipo *when*. Se define un *coordinador* para cada regla; aunque en ciertos casos es posible que el comportamiento del *coordinador* esté definido por más de una regla (sección 4.4).

A) Descripción del comportamiento del Coordinador - mensaje síncrono-

Cuando se recibe un mensaje síncrono el comportamiento del componente *coordinador* varía dependiendo de las distintas situaciones que pueden darse (Figuras 4.7 y siguientes)[20], según se explica en los párrafos siguientes. Cuando el *coordinador* detecta la ocurrencia de un mensaje síncrono (lanzado por un componente *cliente* al que ha sido asociado para coordinar la ejecución de uno o más aspectos), (1) en Figura 4.7, actúa del siguiente modo:

- Escribe en la pizarra del componente *BlackBoard* la información producida en tiempo de ejecución asociada al evento detectado.
- Notifica esta ocurrencia al componente *SuperCoordinador* (2).

[20] Para dar mayor claridad a la explicación, en las figuras sólo se representa un aspecto asociado al *coordinador*.

- Permanece a la espera de la respuesta (3). Ésta depende del valor de las condiciones asociadas al aspecto cuando el evento fue disparado. Se pueden dar cuatro situaciones:
  - No se cumple ninguna regla (el valor de la condición, en tiempo real, no coincide con el valor *Cond* en la fila correspondiente de los *Common Items*).
  - Se cumple alguna regla y el valor de la cláusula *when* es *before*.
  - Se cumple alguna regla y el valor de la cláusula *when* es *after*.
  - Se cumple alguna regla y el valor de la cláusula *when* es *around*.

a) No se cumple ninguna regla -no hay Matching-

El *coordinador* recibe de *SuperCoordinador* la notificación de que no se dan las condiciones de aplicación del aspecto (3) y, por tanto, no tiene que ejecutarse. En este caso, el *coordinador* sólo ha de comunicar al componente *servidor* que ejecute la operación asociada al evento interceptado (4, y 5). Además, ha de borrar la información sobre este evento que había escrito en *BB*, devolver el control al componente *cliente* (6) y concluye su actividad.

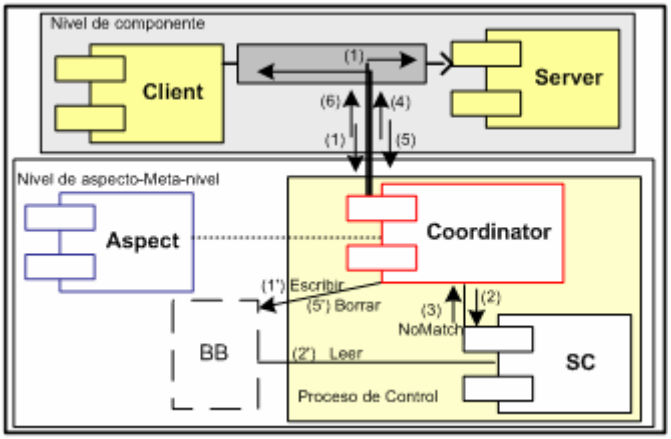


Figura 4.7. Comportamiento del *coordinador,* no se aplica el aspecto.

b) Se cumple alguna regla -hay Matching-

Cuando el *coordinador* recibe de *SuperCoordinador* la notificación de que se cumple una regla (3 en Figura 4.7) y siguientes se deben realizar las tareas de coordinación para la ejecución del aspecto y la operación del componente *servidor* que ha sido solicitada. El *coordinador* estudia el contenido de las condiciones de tipo *when* que definen la aplicación del o de los aspectos con objeto de indicar al *servidor* y al aspecto cuándo deben ejecutarse.

Además, ha de borrar de *BB* la información relativa al evento dado que ya ha sido atendido.

b1) when = before

La ejecución del aspecto debe ser anterior a la ejecución de la operación del *servidor*; así, la actividad del *coordinador* implica lo siguiente (Figura 4.8):

- Notifica al aspecto que debe ejecutarse (4).
- Permanece a la espera de que éste le informe del fin de la ejecución (5).
- Notifica al componente *servidor* (*Server*) que ejecute la operación desencadenante del evento (6 y 7).
- Borra de la pizarra del *BlackBoard* la información sobre este evento.
- Devuelve el control al componente *cliente* (*Client*) (8).
- Concluye su actividad.

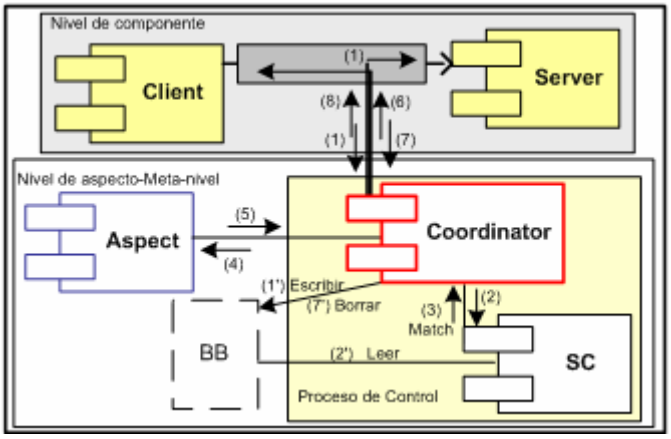


Figura 4.8. Actividad de *coordinador* si *when = before.*

b2) when = after

En este caso, la ejecución de la operación del componente *servidor* debe ser anterior a la del aspecto, por lo que la operación del *coordinador* implica lo siguiente (Figura 4.9):

- Notifica al *servidor* que debe ejecutar la operación desencadenante (4).
- Permanece a la espera de que le informe del fin de la ejecución (5).
- Notifica al aspecto que debe ejecutarse (6).
- Permanece a la espera del fin de ejecución (7).
- Borra de la pizarra la información sobre el evento correspondiente.
- Devuelve el control al componente *cliente* (8).
- Concluye su actividad.

b3) when = around

La operación del aspecto en este caso se debe ejecutar en lugar de la operación del componente *servidor* solicitada por el componente *cliente*. Así, la ejecución del *coordinador* implica lo siguiente (Figura 4.10):

- Notifica al aspecto que debe ejecutarse (4).
- Permanece a la espera de la notificación de fin de ejecución (5).
- Borra de de la pizarra la información sobre el evento.
- Devuelve el control al componente *cliente* (6).
- Concluye su actividad.

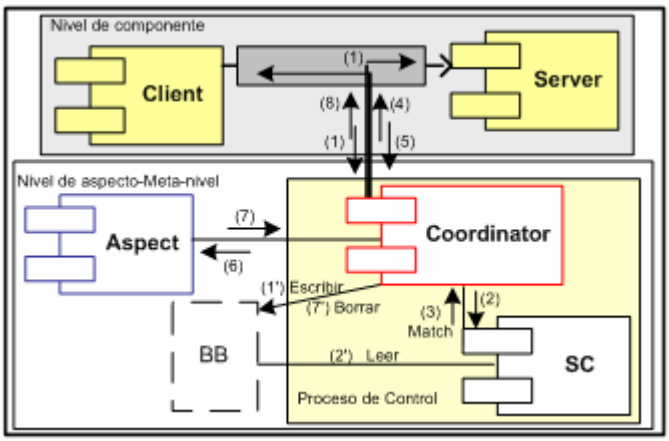


Figura 4.9. Actividad de coordinador si *when = after*.

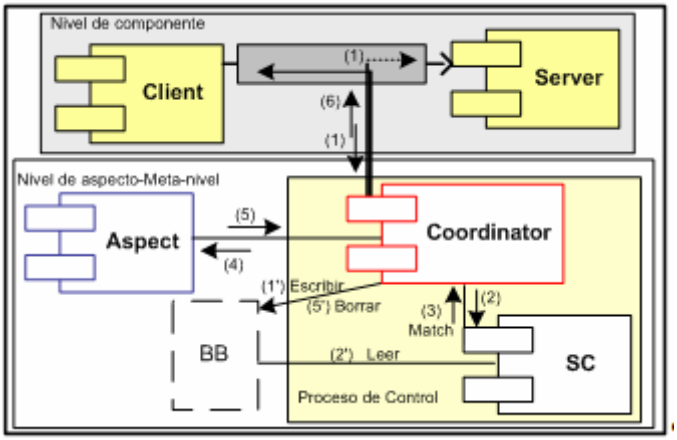


Figura 4.10. Actividad de coordinador si *when = around*.

B) Descripción del comportamiento del Coordinador -mensaje asíncrono-

Cuando se recibe un mensaje asíncrono, el comportamiento del componente *coordinador* varía dependiendo de las distintas situaciones que pueden darse (Figuras 4.11 y 4.12) según se explica en los párrafos siguientes. Cuando el *coordinador* detecta la ocurrencia de un mensaje asíncrono lanzado por un componente *cliente* -al que ha sido asociado para coordinar la ejecución de uno o más aspectos-, la ejecución del aspecto no debe suponer ningún retraso en la actividad del componente *servidor*, (1) en la Figura 4.11. En este caso, el *coordinador*, al detectar este evento, actúa del siguiente modo:

- Permite que el evento interceptado llegue a su destino (2 en Figura 4.11): el componente *servidor* continúa su actividad normal.
- Escribe en la pizarra del componente *BlackBoard* la información producida en tiempo de ejecución asociada al evento detectado.
- Notifica esta ocurrencia al componente *SuperCoordinador* (3).
- Permanece a la espera de la respuesta (4). Ésta depende del valor de las condiciones asociadas al aspecto cuando el evento fue disparado. Se pueden dar dos situaciones:
  - No se cumple ninguna regla.
  - Se cumple alguna regla.

a) No se cumple ninguna regla -no hay Matching-

El *coordinador* recibe de *SuperCoordinador* la notificación de que no se dan las condiciones de aplicación del aspecto (4) y, por tanto, no tiene que ejecutarse. En este caso, el *coordinador* sólo ha de borrar la información que de este evento había escrito en *BB* y concluye su actividad.

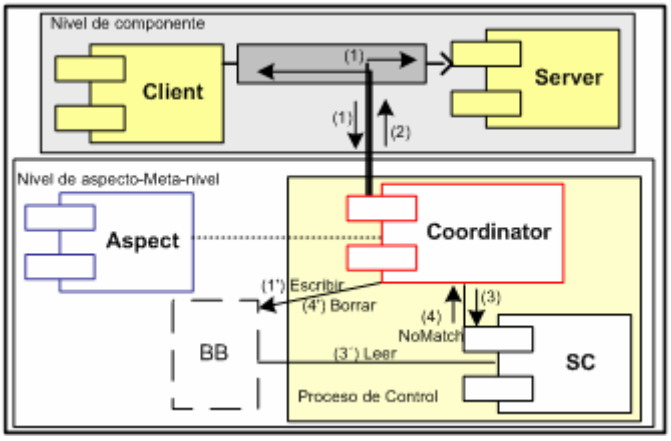


Figura 4.11. Mensaje asíncrono, si no hay *matching*.

b) Se cumple alguna regla -hay Matching-

Cuando el *coordinador* recibe de *SuperCoordinador* la notificación de que se cumple una regla, se deben llevar a cabo las tareas necesarias para la ejecución del aspecto (Figura 4.12):

- Borra de la pizarra la información relativa al evento dado, que ya ha sido atendido.
- Notifica al aspecto que debe ejecutarse (5).
- Permanece a la espera de que éste le informe del fin de la ejecución (6).
- Concluye su actividad.

Nótese que en este caso la cláusula *when* no tiene efecto, por lo que no se evalúa.

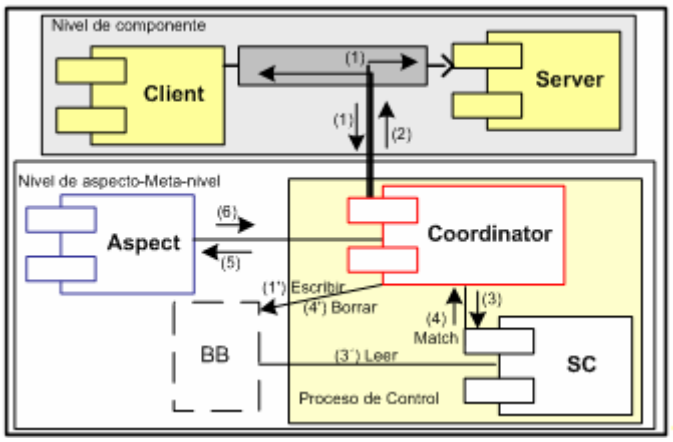


Figura 4.12. Mensaje asíncrono, si hay *matching*.

C) Descripción de la interfaz del componente Coordinador

Interfaz que proporciona

Está formada por una operación que denominaremos de un modo genérico *co.op,* siendo *op* el nombre de la operación solicitada por el componente *cliente* al componente *servidor* e interceptada por el *coordinador.*

Operaciones que necesita

Las operaciones que requiere el componente *coordinador* de los otros componentes con los que se relaciona son:

- *Comprobar_Reglas*, operación proporcionada por el *MetaCoordinador*.
- Ejecución del servicio que proporciona el aspecto: *asp.op,* siendo *op* el nombre de la operación que ejecuta el aspecto, y *asp* el nombre del componente de aspecto, que corresponde a la segunda parte de la regla asociada al *coordinador* y se ejecuta si se cumple la primera parte.
- Ejecución de la operación del componente *servidor* que desencadenó la ejecución del *coordinador*: *serv.op,* siendo *op* el nombre de la operación solicitada por el *cliente* e interceptada por el *coordinador*, y *serv* el nombre del componente proveedor de dicha operación.
- Ejecución de las operaciones *escribir* y *borrar* proporcionadas por el componente *BlackBoard*.

Operaciones internas

- Una operación *main* que controla la ejecución de todas las operaciones que requiere y es la que define su funcionalidad.

La descripción en pseudicódigo del *coordinador* se representa en Figura 4.13, para el caso en el que tenga asociado un aspecto.

```
Begin
    //se intercepta el evento de solicitud de ejecución de una
operación del componente servidor
     Obtener valores de CI
     Asignar valores a BB
     Llama Sc.comprobar_reglas(BB,Match)
           Otener (Match)
     Borra de  BB
     When CI.RMS_event DO            //evento sincrono
                If Match = False then
                      Begin
                            Llama Server.Op(resp)
                               Obtener (resp)
                      End
                 If Match = True then
                      Begin
                            If CI.when = before then
                               Begin
                                  Llamar Aspect.op()
                                  Llamar Server.op(resp)
                                  Obtener (resp)
                                End
                             If CI.when = after then
                                Begin
                                 Llamar Server.op(resp)
                                 Obtener (resp)
                                 Llamar Aspect.op()
                                End
                            If CI.when = around then
                                  Llama Aspect.op()
                                End
    Return (resp)
    EndDO
    When CI.RMA_event DO              //evento asincrono
                If Match = False then
                               Llamar Server.Op(resp)
                If Match = True then
                      Begin
                            If CI.when = before then
                                Begin
                                  Llamar Aspect.op()
                                  Llamar Server.op(resp)
                                End
                             If CI.when = after then
                                Begin
                                  Llamar Server.op(resp)
                                 Llamar Aspect.op()
                                End
                            If CI.when = around then
                                Llamar Aspect.op()
                                End
    EndDO    End
```

Figura 4.13. Pseudocódigo del componente *coordinador*.

Descripción del componente MetaCoordinador (SC)

El componente *MetaCoordinador* o *SuperCoordinador* (*SC*) se ha definido al ser necesaria una entidad que monitorice el estado global del sistema y sea capaz de chequear de forma íntegra las condiciones. Además, en sistemas complejos, la información asociada a varios eventos disparados y que ha sido escrita en la pizarra del *BlackBoard* por distintos *coordinadores* permanece en esta estructura hasta que *SC* trate la información que se ha escrito previamente.

Durante su ejecución analiza la información que se produce en tiempo de ejecución asociada con los eventos disparados, evalúa las condiciones y los valores de los parámetros públicos de los componentes del sistema y determina si hay coincidencia con el valor de las condiciones que definen las políticas de coordinación (expresadas en las reglas) asociadas a un evento y aspecto.

Descripción del comportamiento del MetaCoordinador

SC actúa del siguiente modo (Figura 4.14):

- Recibe de un *coordinador* la notificación de que un evento se ha disparado (1).
- Lee la informacion que está escrita en la pizarra del *BlackBoard* (2) y comprueba si coincide con la parte izquierda de alguna regla (fundamentalente el valor de la condición en tiempo de ejecución se compara con el valor de *Cond* en la estructura *Common Items*).
- Devuelve el control al *coordinador* que hizo la invocación que acaba de tratar, notificándole el resultado de su ejecución (3): si hay coincidencia con una regla (*Match = True*), el *coordinador* la ejecuta (es decir, se ejecuta el aspecto). En otro caso (*Match = False*), no se ejecuta ninguna regla (el aspecto no se ejecuta).

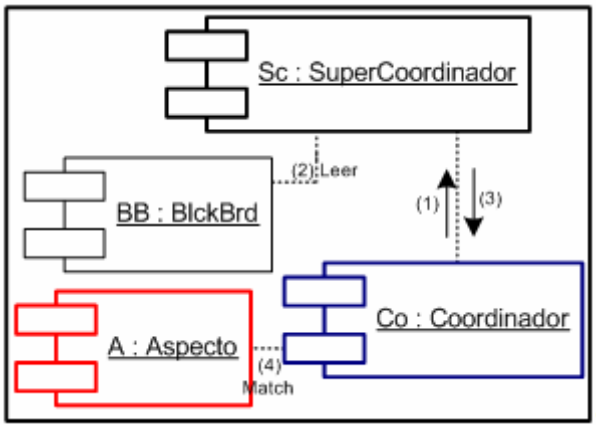


Figura 4.14. Descripción de *MetaCoordinador*.

Descripción de la interfaz del MetaCoordinador

Interfaz que proporciona

Está formada por la operación *Comporbar_Reglas* que es solicitada por un *coordinador*.

Operaciones que necesita
SC requiere a operación *Leer* del componente *BlackBoard.*

Operaciones internas
Después de leer la información correspondiente de la pizarra (escrita en tiempo de ejecución por el *coordinador* que lo invoca), *SC* compara su contenido con el de las reglas que definen su comportamiento para determinar si se cumple alguna. La descripción en pseudicódigo de *SC* se representa en Figuras 4.15.

```
Begin
//al recibir la solicitud de ejecución de la operación
Comprobar_reglas.
Leer BB
Match=False
Repetir
i:=i+1
Obtener Regla[i]
OK=Compara(Regla[i],BB)
If OK then Match =True
Hasta Match=True or i=n
Return (Match)
End
```

Figura 4.15. Descripción en pseudocodigo de *MetaCoordinador*.

Descripción del componente BlackBoard (BB)

Este componente gestiona la información que se produce en tiempo de ejecución. A él acceden los *coordinadores* y *SC*: los *coordinadores* para anotar en la estructura *pizarra,* que contiene la información asociada a un evento detectado por él; y *SC* para leer dicha información y determinar si en tiempo de ejecución se dan las condiciones especificadas para que se ejecute un cierto aspecto.

Descripción del comportamiento del BlackBoard

Su comportamiento se resume del siguiente modo:
- Ejecuta la operación *Escribir* en la pizarra cuando la invoca un *coordinador*.
- Ejecuta la operación *Leer* de la pizarra cuando la invoca *SC*.
- Ejecuta la operación *Borrar* de la pizarra cuando la invoca un *coordinador*.

Descripción de la interfaz del BlackBoard

Interfaz que proporciona
Está formada por las operaciones *Escribir, Borrar* que es solicitada por los *coordinadores* y *Leer* solicitada por *SC*.

Operaciones internas
Su operatividad viene determinada por las operaciones necesarias para gestionar la pizarra adecuadamente.

4.3.1.7. Conclusiones del apartado

En este apartado se han descrito los elementos que definen el modelo que se propone:

- Los componentes del *nivel base* que constituyen el *sistema inicial* y actúan siguiendo la filosofía cliente-servidor.
- Los elementos del *meta nivel* o *nivel de aspecto* que permiten la extensión del *sistema inicial*:
  - Conjunto de aspectos $\mathcal{CA}$.
  - Estructura de *Common Items* $\mathcal{CI}$.
  - Conjunto de Reglas $\mathcal{R}$.
  - El *BlackBoard* $\mathcal{BB}$.
  - El *Proceso de Control,* que a su vez está constituido por:
    . Los componentes *coordinadores* $\{ Co_i \}$.
    . El *MetaCoordinador* Co_s .

    De este modo, el *Proceso de Control,* se puede representar como:

    $$\mathcal{PC} = \{ Co_s, \{Co_i\}, \mathcal{New\_I} \}$$

    donde $\mathcal{New\_I}$ representa las nuevas interacciones definidas entre los componentes del *Proceso de Control,* y los componentes cliente y servidor implicados en la inclusión de cada aspecto.

  El *Proceso de Control* así definido gestiona las nuevas interacciones para el *sistema extendido.*

Finalmente, se ha descrito cada uno de los elementos y cómo interaccionan entre sí, considerando las distintas posibilidades relativas al cumplimiento de las condiciones de ejecución y de las reglas, asicomo si se trata de resolver eventos síncronos o asíncronos.

4.3.2. Diagramas de actividad para los elementos del *meta nivel*

Como resumen de lo que se ha expuesto en este apartado en el que se explica la estructura propuesta para el *meta nivel* o *nivel de aspecto*, en Figuras 4.16 y 4.17 se representan los diagramas de actividad para los elementos del *nivel de aspecto*, cuando se reciben mensajes síncronos y asíncronos respectivamente. Para mayor claridad, sólo se representa un aspecto.

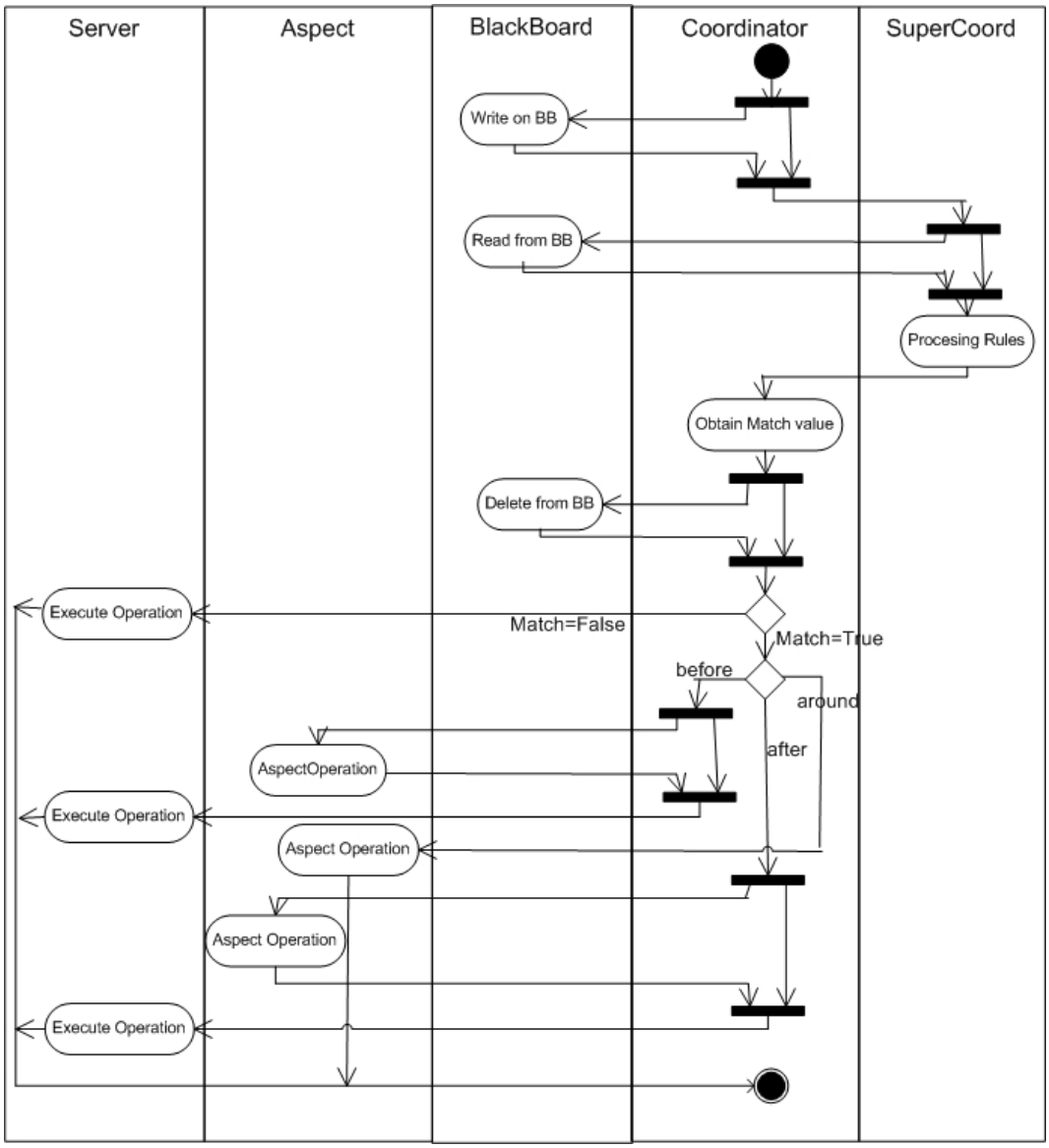


Figura 4.16. Diagramas de actividad. Recibir mensaje síncrono.

4.4. Inclusión de varios aspectos

Una vez realizada la descripción genérica del modelo, en esta sección se presenta una descripción complementaria. Se describen detalladamente algunos casos representativos de situaciones que se pueden dar, dependiendo del número de aspectos a incluir en el sistema y el punto de inserción.

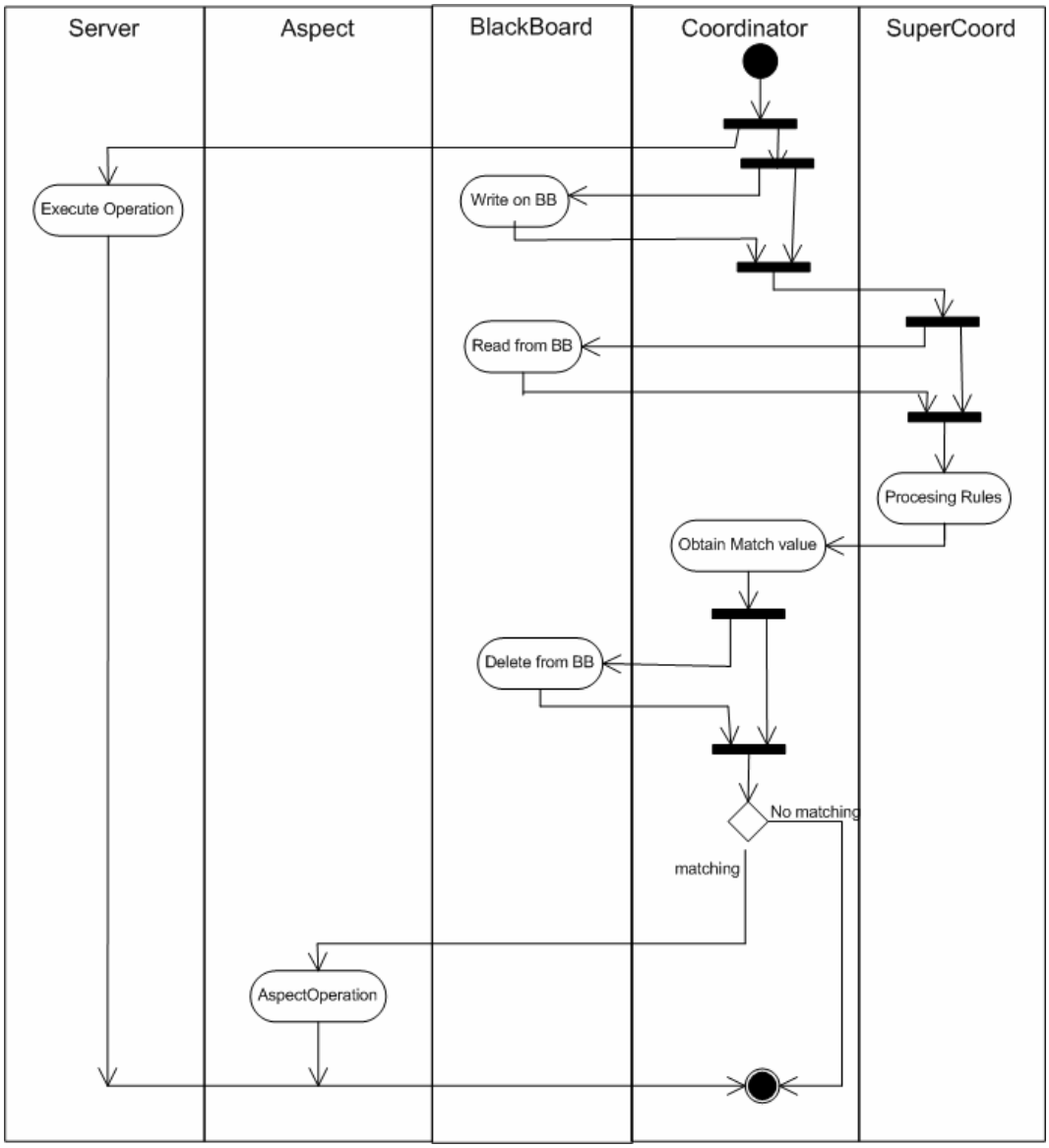


Figura 4.17. Diagramas de actividad. Recibir mensaje asíncrono.

Para cada caso se indican sus características: número de aspectos, de *coordinadores* y puntos de inserción que lo definen. En función de ello, se obtiene una regla que permite al *MetaCoordinador* determinar si, en tiempo de ejecución, ésta se cumple o no. Para facilitar su comprensión, cada caso se representa gráficamente.

Tabla 4.2 representa esquemáticamente las características de los casos que se muestran a continuación.

CASO	Aspectos	Comp	IP	Coord
1	1	1	1	1
2	≥2	1	≥ 2	≥2
2*	≥2	1	≥2	≥2
3	1	≥2	≥2	≥2
4	≥2	1	1	1
general	≥2	Variable, depende del caso		

Tabla 4.2. Elementos característicos de los casos descritos.

Leyenda: **CASO**: Número identificativo del caso.
Aspectos: Número de aspectos que se estudian en ese caso.
Comp: Número de componentes a los que se asocia cada aspecto.
IP: Número de Puntos de Inserción para cada aspecto.
Coord: Número de coordinadores que se definen.

4.4.1. Caso 1

Definición: "*Inclusión de un aspecto actuando sobre un componente (servidor)*".

Supone:

- La identificación de un *Punto de Inserción* (*IP*).
- La definición de un único componente *coordinador* que gestiona la inclusión de la nueva funcionalidad.
- Éste es el caso más sencillo y su estudio permite comprender mejor *AOSA Model* y su aplicación; se describe aquí para resaltar las semejanzas y diferencias con los que se muestran a continuación.

En este caso, el *coordinador* intercepta el mensaje dirigido al componente *servidor* procedente del componente *cliente*, toma las decisiones correspondientes y, en respuesta a ellas, dispara eventos hacia los componentes *MetaCoordinador (SC)*, aspecto y *servidor*. Figura 4.18 muestra la representación de la estructura de niveles para este caso: se parte de un *sistema inicial* formado por varios componentes, entre los que se destaca un componente *cliente* (*Cli:Cliente*) que solicita la ejecución de una operación que proporciona un componente *servidor* (*Serv:Servidor*). Como el meta nivel se define de un modo transparente a ellos, sólo es necesario redefinir la interacción entre los componentes *cliente* y *servidor* implicados en la inserción del aspecto, e incorporar los elementos del meta nivel y las nuevas relaciones.

Reglas

Para caso 1, la inserción de los aspectos supone la definición de una regla:

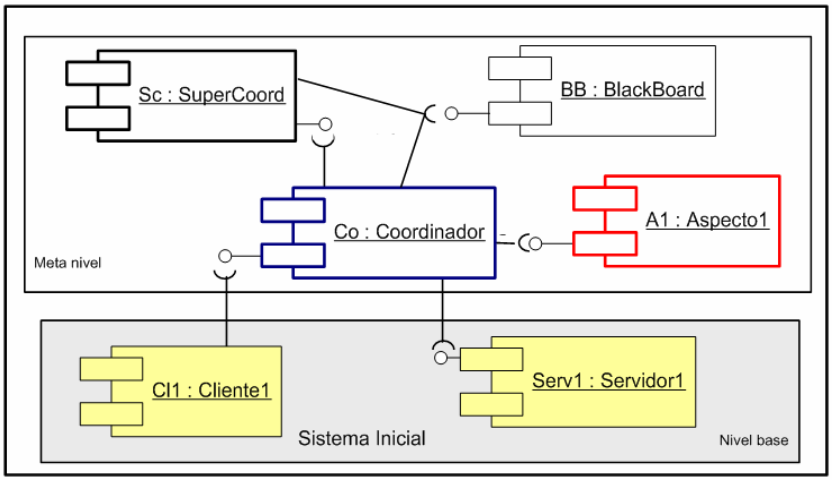


Figura 4.18. Representación esquemática de caso 1.

```
IF ev | IP OF Comp | Cond THEN
      DO Aspect_Op OF Asp_Com WHEN When_C
```

Donde: `ev`, `IP`, `Comp`, `Cond`, `Aspect_Op`, `Asp_Com`, `When_C` son los valores de la única fila que forma la estructura CI para este caso.

El *coordinador* se define asociado a `IP, Asp_Com`, y `Comp`.

4.4.2. Caso 2

Definición: *"Inclusión de dos o más aspectos actuando cada uno sobre un componente"***.**

Supone:

- La identificación de dos o más *Puntos de Inserción* (IP), uno sobre cada componente *servidor*.
- Es la generalización del caso 1 y supone la inclusión de varios aspectos actuando independientes sobre distintos componentes del sistema. En este apartado, sólo se consideran dos aspectos para mantener la claridad en la explicación.
- La definición de un *coordinador* por cada aspecto a insertar.

Cada *coordinador* intercepta los eventos dirigidos al componente *servidor* procedentes del componente *cliente* asociado, toma las decisiones correspondientes y, en respuesta a ellas, lanza eventos hacia el *MetaCoordinador* y los componentes de aspecto y/o servidor correspondientes. Figura 4.19 muestra la representación de la estructura de niveles para este caso: se parte de un *sistema inicial* formado por varios componentes, entre los que destacamos dos componentes cliente (*Cl1*:*Cliente* y *Cl2:Cliente*) y dos componentes servidor (*Serv1*:*Servidor* y *Serv2:Servidor*) con el que cada uno está asociado.

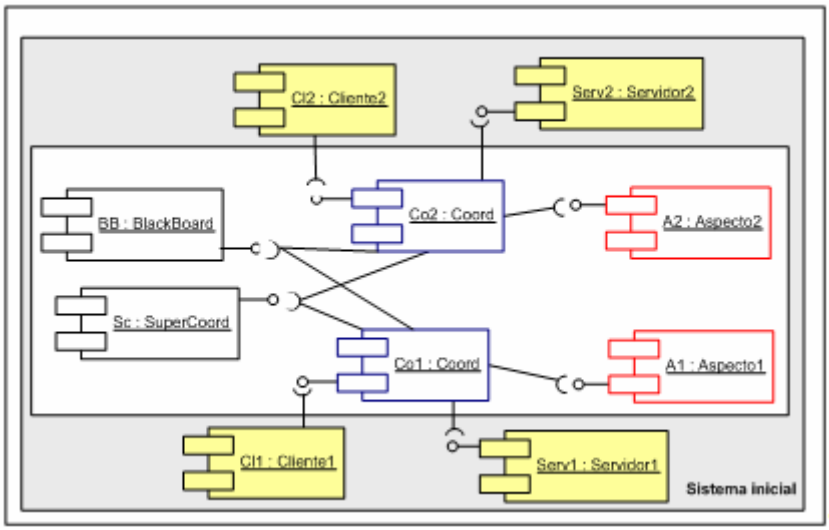


Figura 4.19. Representación esquemática del caso 2. Dos aspectos.

Para completar la definición del *sistema extendido* sólo hay que redefinir la interacción entre los componentes *cliente* y *servidor* implicados en la inserción de cada aspecto, e incorporar los elementos del meta nivel y las nuevas relaciones.

Reglas

La inserción de los aspectos supone la definición de dos reglas, que se deducen de la información de la estructura *Common Ítems,* que para este caso tiene dos filas:

```
IF ev1 | IP1 OF Comp1 | Cond1 THEN
            DO Aspect_Op1 OF Asp_Com1 WHEN When_C1
IF ev2 | IP2 OF Comp2 | Cond2 THEN
            DO Aspect_Op2 OF Asp_Com2 WHEN When_C2
```

El primer coordinador (*Co1*) se define asociado a IP_1, $Comp_1$ y Asp_Com_1. El segundo (*Co2*), a IP_2, $Comp_2$ y $Aspect\_Op_2$.

Aquí se han representado dos aspectos; similar sería el tratamiento para más aspectos que se aplican a distintos componentes.

Caso 2*

Una situación particular del caso anterior es la que se describe a continuación:

Definición: *"Inclusión de dos o más aspectos actuando sobre un único componente servidor; los aspectos se insertan en distintos puntos (IP) del componente"***.**

Supone:

- Que se identifica un *Punto de Inserción* (IP) para cada aspecto –todos en el mismo componente *servidor*-.

- Que se define un *coordinador* por cada *IP*, cada uno de ellos gestiona la inserción del aspecto al que está asociado.

Cada *coordinador* intercepta los eventos dirigidos al componente *servidor*, procedentes de los componentes *cliente* asociados, toma las decisiones correspondientes y, en respuesta a ellos, lanza eventos hacia el *MetaCoordinador* y los componentes de aspecto y/o *servidor* asociados (Figura 4.20).

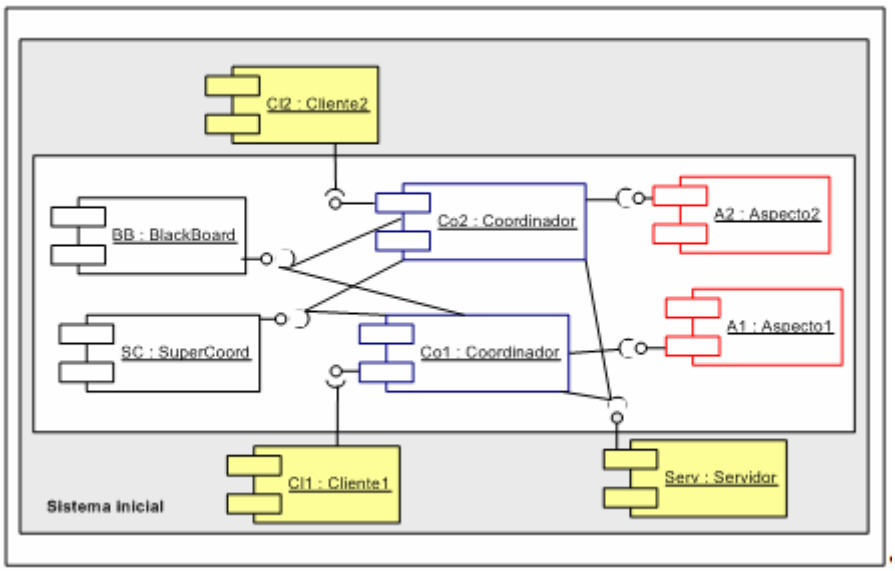


Figura 4.20. Representación esquemática del caso 2*. Dos aspectos.

Reglas

La inclusión de los aspectos en el sistema supone, como antes, la definición de dos reglas, pero ahora $\texttt{Comp}_1$ es igual a $\texttt{Comp}_2$.

4.4.3. Caso 3

Definición: *"Inclusión de un aspecto que tiene que actuar sobre dos o más componentes".*

Supone:

- Que se identifican varios *Puntos de Inserción* (IP), uno sobre cada componente *servidor*.
- Se define un *coordinador* por cada *IP*: cada uno de ellos gestiona la inserción del aspecto en el componente *servidor* correspondiente.
- Se puede considerar como un caso particular del caso 2 cuando un mismo aspecto se aplica a distintos componentes *servidor* (esto es, en distintos IP).

Cada *coordinador* intercepta los eventos dirigidos al componente *servidor* procedentes del componente *cliente* asociado, toma las decisiones correspondientes y en respuesta a ellos lanza eventos hacia el *MetaCoordinador* y los componentes de aspecto y/o *servidor* asociados. Figura 4.21 muestra una representación gráfica de este caso.

Como en los casos anteriores, la definición del *sistema extendido* se completa redefiniendo la interacción entre los componentes *cliente* y *servidor* implicados en la inserción del aspecto, incorporando los elementos del meta nivel y las nuevas relaciones.

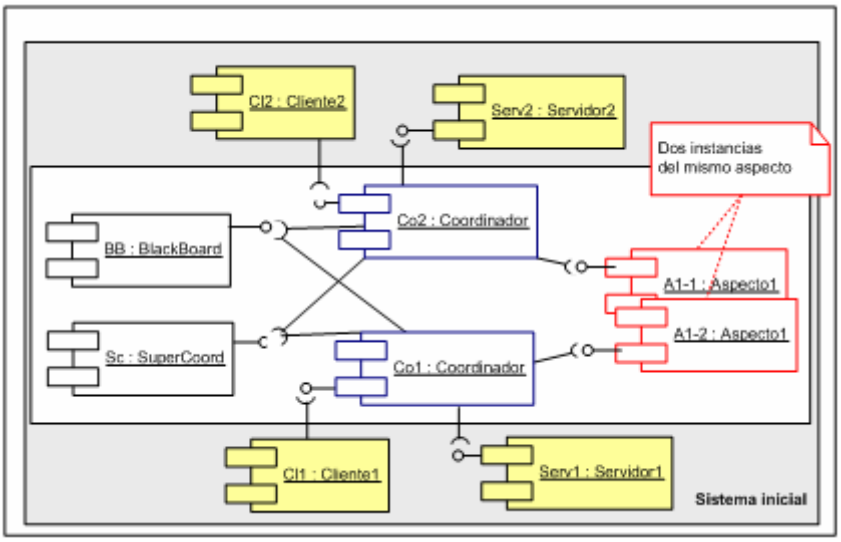


Figura 4.21. Representación esquemática de caso 3. Dos aspectos.

Reglas

La inserción del aspecto supone la definición de dos reglas, que se deducen de la información de la estructura *Common Items* que para este caso tiene dos filas:

```
IF ev1 | IP1 OF Comp1 | Cond1 THEN
        DO Aspect_Op OF Asp_Com WHEN When_C1
IF ev2 | IP2 OF Comp2 | Cond2 THEN
        DO Aspect_Op OF Asp_Com WHEN When_C2
```

El primer coordinador (*Co1*) se define asociado a IP_1, $Comp_1$ y `Asp_Com.` El segundo (*Co2*) asociado a IP_2, $Comp_2$ y `Aspect_Op`.

Como en los epígrafes anteriores, aquí se ha representado un aspecto que se aplica sobre dos componentes *servidor*; similar sería el tratamiento para un aspecto a insertar en más componentes. Éste es un ejemplo en el que se observa la posibilidad de reutilización de un aspecto.

4.4.4. Caso 4

Definición: *"Inclusión de dos o más aspectos actuando sobre un único Punto de Inserción (en un único componente servidor)".*

Este caso es el más complejo (Jacobson evita su estudio en [JaNg05]). Supone:

- Identificar un único *Punto de Inserción* (*IP*).
- Se define un único *coordinador* que gestiona la inserción de los aspectos.
- Es necesario considerar la prioridad en la aplicación de los aspectos.
- Aquí se muestra, a modo de ejemplo, la inserción de dos aspectos.

El *coordinador* intercepta los eventos dirigidos al componente *servidor* y procedentes del componente *cliente* asociado, toma las decisiones correspondientes y lanza eventos en respuesta a ellas hacia el *MetaCoordinador* y los componentes de aspecto y/o *servidor* (Figura 4.22).

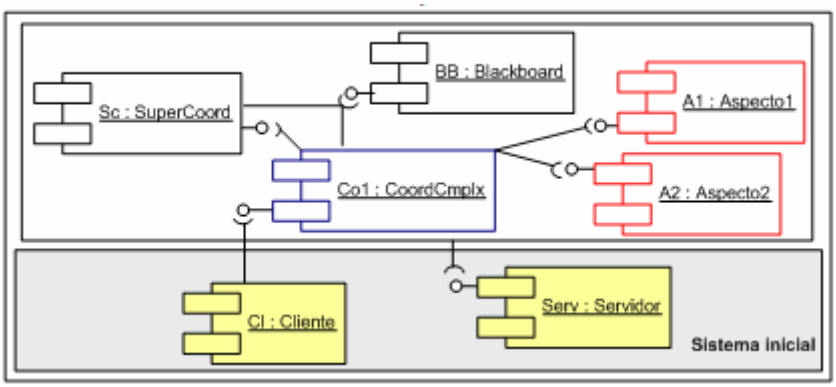


Figura 4.22. Representación esquemática de caso 4. Dos aspectos.

Relación de prioridad

En este caso, los aspectos se aplican en el mismo *IP* siendo responsabilidad del ingeniero de software determinar en *tiempo de diseño* la prioridad de su ejecución. Para ello, hay que establecer una *relación de prioridad* que determine el orden en que se insertan.

Reglas

La inclusión de los aspectos supone la definición de una regla compuesta que dirige el comportamiento del componente *coordinador*. Su formato genérico (para dos aspectos) es:

```
IF  ev |  IP OF Comp THEN
     IF (Cond1) THEN DO (Aspect_Op1 OF Asp_Com1) WHEN When_C1
     IF (Cond2) THEN DO (Aspect_Op2 OF Asp_Com2) WHEN When_C2
```

En este caso, el comportamiento del *coordinador* tiene que ser capaz de responder a esta regla compuesta. Esto lleva a la definición de un componente *coordinador* distinto al descrito para los casos anteriores y más complejo (*coordinador*

complejo). A modo de ejemplo, en la Tabla 4.3 se representan las alternativas que hay que considerar en la descripción del comportamiento de este *coordinador* (para dos aspectos).

Cuando en los aspectos coincida el valor de la condición *when*, se ejecutan según la prioridad establecida en el momento de diseñar el *sistema extendido*. En este caso, a Aspecto 1 ($Cond_1$, $Aspect\_Op_1$, Asp_Com_1, $When\_C_1$) se le asigna mayor prioridad.

Subreglas	Match1	Match2	When_C1	When_C2
C1	F	F	-	-
C2	T	T	Before	Before
C3	T	T	Before	After
C4	T	T	Before	Around
C5	T	T	After	Before
C6	T	T	Around	Before
C7	T	T	After	After
C8	T	T	After	Around
C9	T	T	Around	After
C10	T	T	Around	Around
C11	T	F	Before	-
C12	T	F	After	-
C13	T	F	Around	-
C14	F	T	-	Before
C15	F	T	-	After
C16	F	T	-	Around

Tabla 4.3. Comportamiento de *coordinador complejo*. Ejemplo caso 4, dos aspectos.

El pseudocódigo asociado a este componente *coordinador complejo* se muestra en Figura 4.23. En ella, aunque hay alternativas de la instrucción Case que suponen ejecutar las mismas acciones se han mantenido todas (sin agrupar) para una mayor claridad en la exposición:

i) Si *MetaCoordinator* devuelve el valor *Match=False* (no se cumple ninguna regla) para ambos aspectos (C1), no se ejecuta ninguno.
ii) Si devuelve *Match=False* para uno de ellos, sólo se considera el otro (C11 a C13, C14 a C16).
iii) En otro caso, hay que aplicar los aspectos cuando corresponda (dependiendo del valor de la cláusula *when*) y atendiendo a la prioridad entre ellos si *when* coincide.

Finalmente, en estos párrafos se ha considerado la inserción de dos aspectos sobre un componente (en el mismo *IP*). La generalización de este caso permite la inclusión de más aspectos, si bien la complejidad de tratamiento crece.

4.4.5. Caso general

La aplicación del modelo a situaciones complejas supone la combinación de los casos anteriores. Su aplicación se puede hacer de forma incremental para facilitar el diseño del *sistema extendido* o insertando todos los aspectos de una vez. En el Capítulo 5 se muestra un ejemplo.

4.5. Eliminación de aspectos

En secciones anteriores se ha mostrado cómo el modelo permite incorporar aspectos a un sistema. Para ello, se han determinado los *extension points* asociados a los aspectos que se quiere insertar y se han definidos los elementos que han permitido llevar a cabo dicha integración (estructura *Common Items*). Para eliminar un requisito, incorporado como un aspecto siguiendo *AOSA Model* hay que extraer del *nivel de aspecto* los elementos correspondientes así como la información asociada y reestructurar las conexiones interceptadas. Igualmente, hay que eliminar del código asociado a la arquitectura del *sistema extendido* las líneas de código correspondientes o bien regenerar el sistema.

4.6. Conclusiones

En este capítulo se ha presentado *AOSA Model,* un modelo para definir arquitecturas de sistemas orientados a aspecto. El modelo permite describir arquitecturas software de sistemas complejos de modo que es sencilla la inserción de nuevos requisitos que se puedan considerar como aspectos. Además, se facilita su mantenimiento, adaptabilidad y evolución, a la vez que se obtienen sistemas con código más limpio y menos enmarañado que aplicando metodologías tradicionales.

Las principales aportaciones del modelo se pueden resumir en los siguientes párrafos:

> Por una parte, el *sistema extendido* se construye combinando la filosofía de DSBC y DSOA, para aprovechar las ventajas de ambos paradigmas. Durante el proceso, los componentes del *sistema inicial* no resultan modificados; sólo es necesario redefinir las conexiones asociadas a los puntos de enlace de los componentes. Además, los aspectos se definen como componentes arquitectónicos. La interacción entre los componentes del sistema y los de aspecto es gestionada por unos componentes *coordinadores*.
>
> Para llevar a cabo la extensión se define una estructura arquitectónica de dos niveles: el *nivel base o nivel de componente* contiene los componentes y conexiones que definen el *sistema inicial* que se desea extender; y un *nivel meta* o *nivel de aspecto* que contiene los componentes de aspecto y los componentes

coordinadores que propone *AOSA Model* para llevar a cabo las acciones de coordinación y tejido de los aspectos con el *sistema base*.

Tanto los componentes del *sistema inicial* como los aspectos son reutilizables, al ser independientes del contexto. Sin embargo, los elementos *coordinadores* se definen dependiendo del contexto en el que se integran, y las operaciones de tejido las realizan de modo transparente tanto para los componentes del *sistema inicial* como para los aspectos que se inyectan (modelo no invasivo). La composición final de los componentes es dinámica en tiempo de ejecución y se realiza a través de las interfaces de los componentes y los aspectos, no existiendo una fase intermedia de mezclado.

El número de aspectos a insertar en el sistema no está limitado. El peor de los casos se da cuando más de un aspecto se tiene que aplicar sobre el mismo punto de corte (*IP*), en cuyo caso la estructura del componente *coordinador* asociado se hace más compleja. Además, en este caso, es necesario que el diseñador especifique la *relación de prioridad* que indique en qué orden han de insertarse (ejecutarse) los aspectos.

Al contrario que en otras propuestas, no se define un *aspecto "coordinador"* que realice las tareas de coordinación entre componentes del *sistema inicial* y los aspectos, sino que es un componente arquitectónico el que determina las interconexiones necesarias y ejecuta los protocolos de coordinación. Ésta es una de las *principales aportaciones* del modelo: *se utiliza un modelo de coordinación exógeno para llevar a cabo la separación efectiva de los aspectos y su composición*; los componentes *coordinadores* establecen la interconexión entre los componentes del sistema y los de aspecto de un modo transparente a ellos. Los componentes implicados desconocen que van a ser coordinados siendo el *coordinador* el que determina la ejecución de uno u otro en función de las condiciones establecidas para los aspectos (políticas de coordinación).

Otras características de *AOSA Model* que se pueden mencionar son: dota al sistema de reconfiguración *ad hoc* en tiempo de diseño y reconfiguración dinámica de la arquitectura; la adaptación de los sistemas se hace en tiempo de diseño, por lo que el mantenimiento es más sencillo; y *AOSA Model* sigue un modelo asimétrico, sin que los componentes del sistema tengan que estar definidos en un modelo de componentes concreto, pues se basa en una especificación genérica de modelo de componentes como UML 2, o en modelos de componentes empresariales como EJB o CCM.

Finalmente, una vez definido el modelo y para obtener las máximas ventajas es necesario proporcionar una metodología que facilite al ingeniero de software las pautas a seguir. En el Capítulo 5 se describe la metodología desarrollada para dar soporte al modelo, así como se describe el metamodelo asociado a *AOSA Model*.

```
Begin
// Coordinador intercepta el evento de
solicitud de una operación del componente
servidor por parte de un componente
cliente y lo intercepta.
   Obtener CI[1] // valores de Aspecto1
   Obtener CI[2] // valores de Aspecto2
//    Para Aspecto1
   Asignar valores Aspecto 1 a BB
   Llama Sc.comprobar_reglas(BB,Match1)
   Obtain (Match1)
//   Para Aspecto 2
   Asignar valores Aspecto2 a BB
   Llama Sc.comprobar_reglas(BB,Match2)
   Obtain (Match2)
   Borrar de BB
   Determinar OpcionCase
// ---------------------
When CI[1].RMS_event do
Case R1 :Begin
        Llama Server.op(resp)
        Obtain (resp)
        End
Case R2: Begin
        Llama Aspect1.op1
        Llama Aspect2.op2
        Llama Server.op (resp)
        Obtain (resp)
        End
Case R3: Begin
        Llama Aspect1.op1
        Llama Server.op (resp)
        Obtain (resp)
        Llama Aspect2.op2
        End
Case R4: Begin
        Llama Aspect1.op1
        Llama Aspect2.op2
        End
Case R5: Begin
        Llama Aspect2.op2
        Llama Server.op (resp)
        Obtain (resp)
        Llama Aspect1.op1
        End
Case R6: Begin
        Llama Aspect2.op2
        Llama Aspect1.op1
        End
Case R7: Begin
        Llama Server.op (resp)
        Obtain (resp)
        Llama Aspect1.op1
        Llama Aspect2.op2
        End
Case R8: Begin
        Llama Aspect2.op2
        Llama Aspect1.op1
        End
Case R9: Begin
        Llama Aspect1.op1
        Llama Aspect2.op2
        End
Case R10: Begin
        Llama Aspect1.op1
        Llama Aspect2.op2
        End
Case R11: Begin
        Llama Aspect1.op1
        Llama Server.op (resp)
        Obtain (resp)
        End
Case R12: Begin
        Llama Server.op (resp)
        Obtain (resp)
        Llama Aspect1.op1
        End
Case R13: Begin
        Llama Aspect1.op1
        End
Case R14: Begin
        Llama Aspect2.op2
        Llama Server.op (resp)
        Obtain (resp)
        End
Case R15: Begin
        Llama Server.op (resp)
        Obtain (resp)
        Llama Aspect2.op2
        End
Case R16: Begin
        Llama Aspect2.op2
        End
 Return (resp)
Enddo
//----------------------
When CI[1].RMA_event do
Case R1 :Begin
        Llama Server.op(resp)
        End
Case R2: Begin
        Llama Aspect1.op1
        Llama Aspect2.op2
        Llama Server.op (resp)
        End
Case R3: Begin
        Llama Aspect1.op1
        Llama Server.op (resp)
        Llama Aspect2.op2
        End
Case R4: Begin
        Llama Aspect1.op1
        Llama Aspect2.op2
        End
Case R5: Begin
        Llama Aspect2.op2
        Llama Server.op (resp)
Llama Aspect1.op1
        End
Case R6: Begin
        Llama Aspect2.op2
        Llama Aspect1.op1
        End
Case R7: Begin
        Llama Server.op (resp)
        Llama Aspect1.op1
        Llama Aspect2.op2
        End
```

```
Case R8: Begin
       Llama Aspect2.op2
       Llama Aspect1.op1
       End
Case R9: Begin
       Llama Aspect1.op1
       Llama Aspect2.op2
       End
Case R10: Begin
       Llama Aspect1.op1
       Llama Aspect2.op2
       End
Case R11: Begin
       Llama Aspect1.op1
       Llama Server.op (resp)
       End
Case R12: Begin
       Llama Server.op (resp)
       Llama Aspect1.op1
       End
Case R13: Begin
       Llama Aspect1.op1
       End
Case R14: Begin
       Llama Aspect2.op2
       Llama Server.op (resp)
       End
Case R15: Begin
       Llama Server.op (resp)
       Llama Aspect2.op2
       End
Case R16: Begin
       Llama Aspect2.op2
       End
Enddo
End
```

Figura 4.23. Pseudocódigo del *coordinador* cuando se aplican dos aspectos.

CAPÍTULO 5
Una metodología para AOSA Model

Cada modelo de desarrollo tiene una metodología que se debe seguir para obtener los mejores resultados en su aplicación, potenciar su calidad, y reducir el coste y el tiempo invertido en el desarrollo de un sistema. AOSA Space define una metodología para desarrollar sistemas siguiendo AOSA Model. En este capítulo se describe detalladamente esta metodología aplicándola al caso de estudio que se propone. En la última sección se describe el metamodelo para AOSA Model.

La metodología se divide en las siguientes etapas:

1) *Especificación del sistema inicial.*
2) *Creación del modelo arquitectónico correspondiente.*
3) *Identificación y especificación de los aspectos a incluir.*
4) *Definición de la especificación arquitectónica del sistema extendido, en un LDA-OA, en la que se incluyen los aspectos identificados en el paso anterior.*
5) *Traducción de la arquitectura obtenida a un LDA convencional y generación de código para obtener un prototipo del sistema.*

Cada una de estas etapas se describe detalladamente a lo largo de las siguientes secciones. Como se muestra en este capítulo, la inserción de los aspectos en el sistema inicial se hace siguiendo un proceso iterativo e incremental.

5.1. Etapas de la metodología

Antes de desarrollar las etapas en las que se ha dividido la metodología que se describe en este capítulo, en Figura 5.1 se representa el punto de vista externo que permitiría desarrollar una arquitectura orientada a aspectos (OA) cuando se sigue *AOSA Model*. En ella, los casos de uso representan las etapas de la metodología.

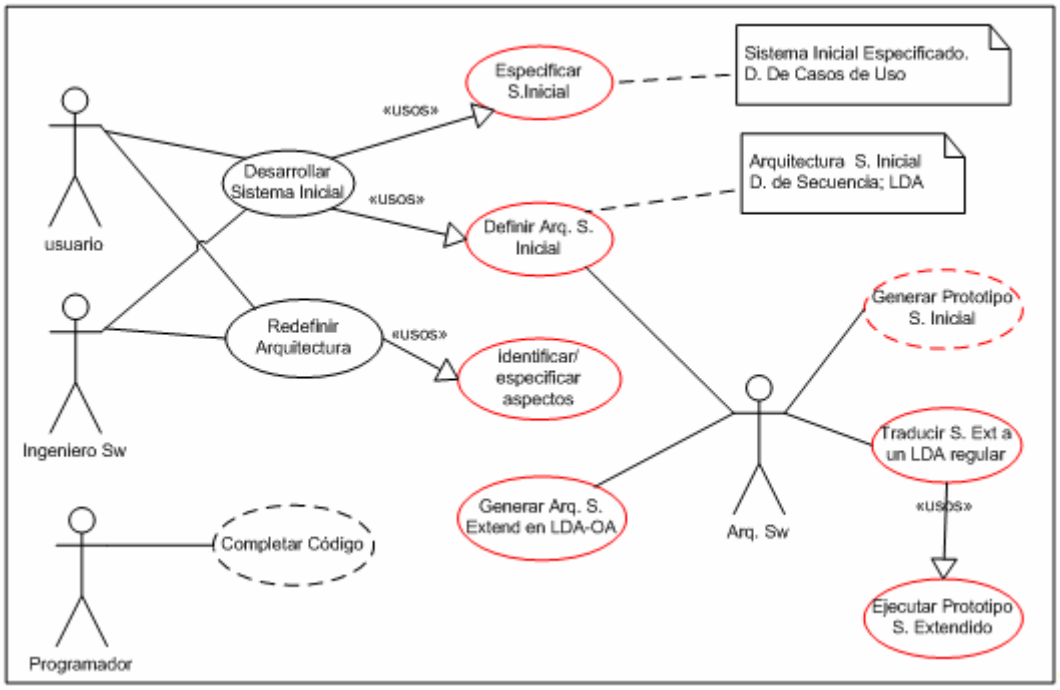


Figura 5.1. Representación esquemática del modelo.

Los actores implicados a lo largo del desarrollo de un sistema siguiendo la metodología son:

- *Usuario*, quien propone el *sistema inicial* y su extensión.
- *Ingeniero de software* que desarrolla el *sistema inicial* en sus primeras etapas y lo redefine para incluir las modificaciones (*sistema extendido*).
- *Arquitecto de software*, quien define la arquitectura del sistema (*sistema inicial*), la especifica en un LDA y genera el *sistema extendido* para obtener su prototipo.
- *Programador*, quien completa la implementación del prototipo para obtener el sistema final y completo.

A lo largo de este capítulo, la metodología se describe utilizando un *caso de estudio* descrito en la siguiente sección.

Las etapas de la metodología, según se enunciaron más arriba y ahora se detallan, son los siguientes (Figura 5.2):

1. Especificación detallada del sistema, sin considerar aspectos: *sistema inicial*.
2. Creación de un modelo de diseño y definición de la arquitectura del sistema, mediante un LDA, para obtener una especificación de diseño de alto nivel.
3. Incorporación de los nuevos requisitos, considerados como aspectos, para obtener el *sistema extendido* observando el principio de inconsciencia [FiFr00].
4. Especificación de la arquitectura del *sistema extendido* en un LDA-OA.
5. Traducción de la arquitectura obtenida a un LDA convencional y posterior generación de código para obtener un prototipo del sistema. Una herramienta asiste al arquitecto de software en estas tareas.

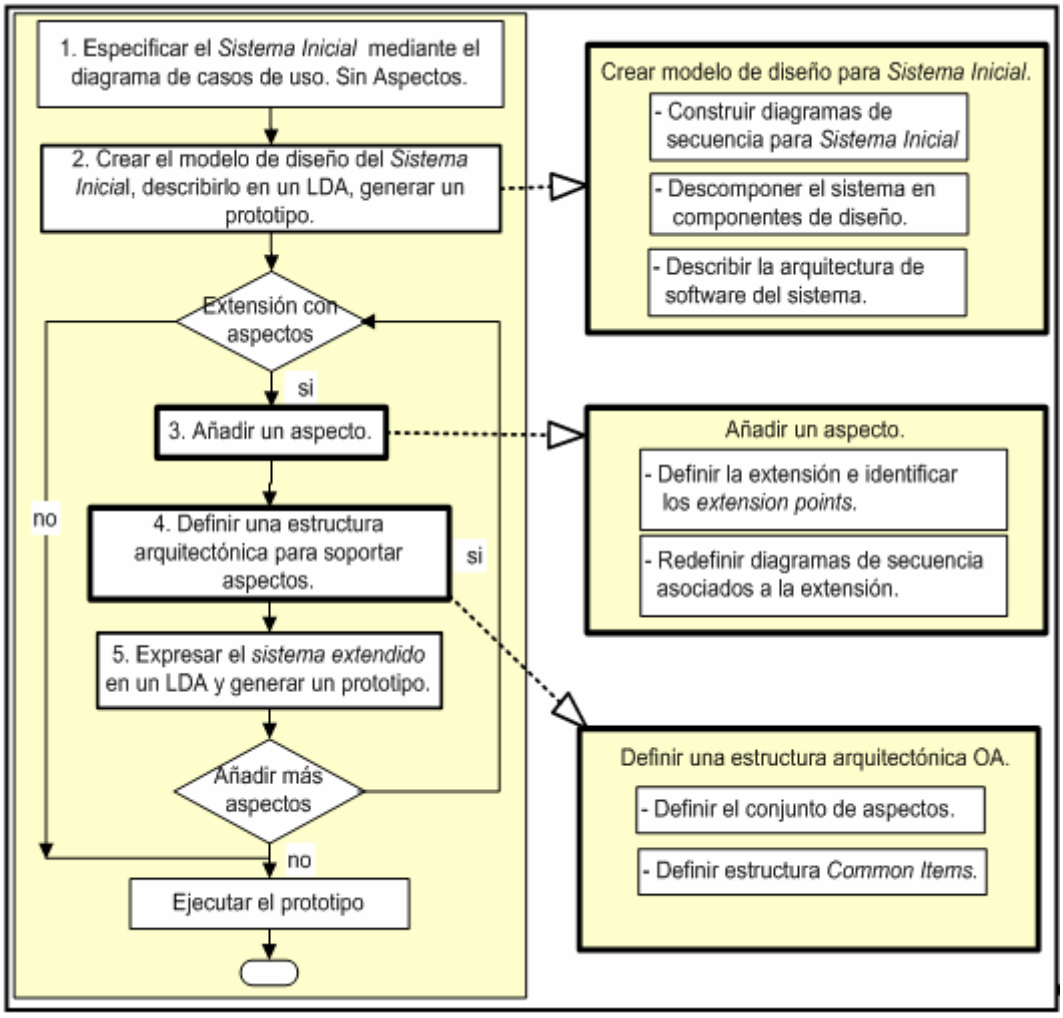


Figura 5.2. Etapas de la metodología.

Para expresar formalmente la arquitectura del *sistema extendido* en un LDA orientado a aspectos, se ha propuesto un Lenguaje de Descripción de Arquitecturas Orientado a Aspectos: *AspectLEDA*, obtenido extendiendo LEDA. La descripción del *sistema extendido* en *AspectLEDA* se traduce a LEDA, de tal manera que es posible, por las características de este lenguaje, comprobar la corrección de la arquitectura del nuevo

sistema y ejecutar un prototipo, ya que, las especificaciones en LEDA se traducen a Java. El objetivo final es obtener la ejecución de un prototipo del *sistema extendido*.

Las etapas de la metodología se han descrito considerando el proceso de desarrollo desde la especificación inicial de un sistema, que, en este caso concreto, se quiere extender con nuevos requisitos que se pueden considerar como aspectos. La inclusión se hace de un modo transparente a los componentes del *sistema inicial*. También es posible aplicar la metodología a un sistema en fase de diseño (no desarrollado por completo), del que se han extraído (o no se han considerado) los aspectos. En este caso, la aplicación de la metodología empieza en la etapa de especificación arquitectónica. La metodología igualmente se puede utilizar en sistemas en fase de evolución o mantenimiento, durante la cual se desea incorporar nuevos requisitos (considerándolos como aspectos). En este caso, la metodología se aplica también desde su tercera etapa.

En las siguientes secciones se describen las etapas de la metodología. Para dar mayor claridad a la explicación cada una de ellas se acompaña con un caso de estudio.

5.2. Especificación del *sistema inicial*

En la primera etapa de la metodología se propone considerar los sistemas desde la fase de especificación de requisitos, en la que se realiza la especificación detallada del sistema (*sistema inicial).* Ésta se hace sin considerar los aspectos que atraviesan el sistema, que se incorporarán posteriormente: en *AOSA Model* se ha optado por extraer los aspectos durante la especificación de requisitos y formalizar su definición durante la arquitectura del sistema, considerándolos como componentes arquitectónicos.

El *sistema inicial,* que puede ser de nueva creación o existir previamente, se especifica mediante el diagrama de casos de uso correspondiente (Figura 5.3). Los aspectos se han extraído del sistema en desarrollo o son nuevos requisitos a incluir en él.

Por su parte, cada caso de uso debe describirse adecuadamente para mantener el sistema convenientemente documentado (Tabla 5.1). Para ello se pueden utilizar diversas plantillas como las propuestas por Coleman [Col98] o Jacobson [JaNg05], entre otros, o la tabla que se propone aquí.

Caso de estudio. Especificación del sistema inicial

El caso de estudio está inspirado en el propuesto por Jacobson en [JaNg05]. Este ejemplo ha sido escogido con la idea de referenciar un caso de estudio ampliamente conocido, en lugar de utilizar uno más próximo a la realidad y, por tanto, más complejo. El ejemplo además sirve para mostrar que existen ciertas interferencias entre los aspectos:

> *Considérese un sistema que proporciona la funcionalidad necesaria para llevar a cabo la gestión de un hotel; especialmente se consideran los requisitos relativos al subsistema de reserva de habitaciones que usan los clientes del hotel y la gestión del restaurante para la reserva de mesas para clientes*.

La descripción de este *sistema inicial*, en función de los casos de uso, es la siguiente (Figura 5.3):

- *Cuando un cliente desea reservar una habitación se comprueba su disponibilidad. Si está disponible, se hace la reserva. En cualquier caso se notifica al cliente del resultado de su solicitud.*
- *Por otra parte, cuando un cliente del hotel desea reservar una mesa en el restaurante, hace una consulta a través de una aplicación de cliente que le permite hacer la correspondiente reserva.*

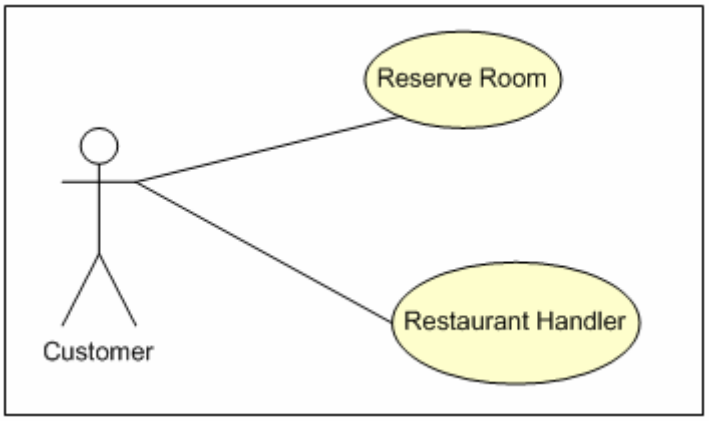


Figura 5.3. Caso de estudio. D. de casos de uso para el *sistema inicial*.

	Caso de uso 1	**Caso de uso 2**
Caso de uso	Reservar habitación	Gestionar Restaurante
Nombre	*Reserve Room*	*Restaurant Handler*
Actor	Cliente (customer)	Cliente (customer)
Relaciones	Ninguna	Ninguna
Descripción	*Reserva habitaciones del hotel.* *Se comprueba la disponibilidad.* *Si está disponible, se hace la reserva.* *Se notifica al cliente del resultado.*	*Reserva mesas en el restaurante para los clientes del hotel.*

Tabla 5.1. Descripción de los casos de uso.

5.3. Creación del modelo de diseño para el *sistema inicial*

El objetivo de esta etapa es especificar el modelo de diseño para el *sistema inicial* y describir su arquitectura mediante un LDA. La descripción arquitectónica del sistema se ha dividido en los siguientes pasos:

5.3.1. Creación de los diagramas de secuencia asociados al *sistema inicial*

Para cada caso de uso del diagrama obtenido en la primera etapa se crea el correspondiente diagrama de secuencia (Figura 5.4a y 5.4b). En cada uno aparecen los componentes de diseño que constituyen el *sistema inicial* así como la interacción entre ellos.

Los componentes pueden ser reutilizados o de nueva creación. En este caso no es necesario especificarlos completamente; sólo tiene que estar definido su comportamiento externo para que se puedan utilizar durante el diseño, pudiéndose retrasar su descripción completa hasta la implementación.

Caso de estudio. Diagramas de secuencia del sistema inicial

Figura 5.4 representa los diagramas de secuencia obtenidos a partir del diagrama de casos de uso del caso de estudio, considerado el escenario más favorable. Los componentes de diseño de los diagramas se asocian a los casos de uso según se muestra en Tabla 5.2.

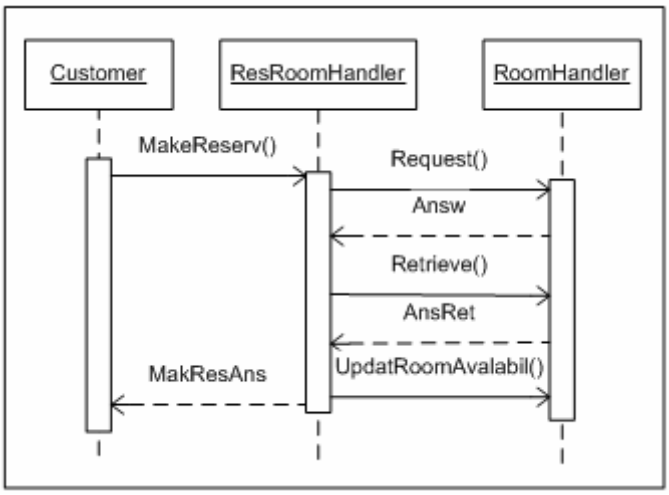


Figura 5.4a). Diagramas de secuencia, CU Reserva de habitación.

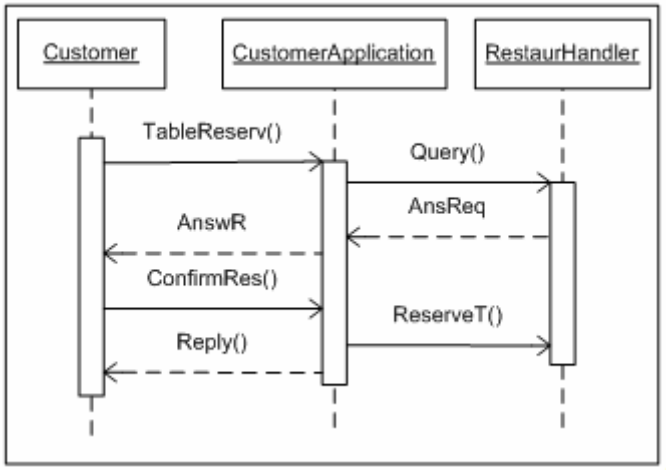


Figura 5.4b). Diagramas de secuencia. CU Gestión del restaurante.

Caso de uso 1	**Caso de uso 2**
Customer (actor)	Customer (actor)
ResRoomHandler	*CustomeApplication*
RoomHandler	*RestaurantHandler*

Tabla 5.2. Componentes de diseño en los diagramas de secuencia.

5.3.2. Descomposición del sistema en componentes de diseño

Tras la especificación detallada del sistema se obtiene su arquitectura. Los componentes de diseño representados en los diagramas de secuencia se deben especificar describiendo sus interfaces y considerando las relaciones entre los componentes.

Caso de estudio. Descripción de los componentes del sistema inicial

De los dos diagramas de secuencia de Figura 5.4 y de los componentes identificados (Tabla 5.2) se deduce la información relativa a las interfaces de los componentes identificados (Tablas 5.3a y 5.3b).

ResRoomHandler	
MakeReserv()	*Solicitud de reserva de una habitación*
RoomHandler	
Request()	*Consulta la disponibilidad*
Retrieve	*Recupera información de las habitaciones disponibles*
UpdateRoomAvailabil()	*Si hay disponibilidad, actualiza los valores*

Tabla 5.3a). Interfaz de los componentes del d. de secuencia *Reserva de habitación*.

CustomerApplication	
RequestTable()	*Solicitud de reserva de una mesa del restaurante*
CofirmReserv()	*Confirma la reserva solicitada*
RestauranteHandler	
Query()	*Recupera información de las mesas disponibles*
ReserveT()	*Si hay disponibilidad, reserva la mesa*

Tabla 5.3b). Interfaz de los componentes del d. de secuencia *Gestión del restaurante.*

5.3.3. Descripción de la arquitectura

Para formalizar el modelo de diseño obtenido (expresado como un conjunto de componentes y sus interacciones) se usa un Lenguaje de Descripción de Arquitecturas. Para la aplicación de *AOSA Model* se ha elegido el lenguaje LEDA. En el Anexo 1 de esta memoria se describen las características generales del lenguaje. Su elección está motivada por las siguientes razones:

- Su fuerte base formal.
- Es dinámico.
- Se traduce a Java.
- Permite la ejecución de un prototipo a partir de la especificación arquitectónica de un sistema, aunque los componentes no estén completamente definidos.

Caso de estudio. Descripción del sistema inicial en un LDA

Figura 5.5 muestra la descripción arquitectónica del *sistema inicial* en LEDA. En ella se puede apreciar que el sistema (*Basesystem*) está formado por los cuatro componentes que se describieron en el apartado anterior. Dos desempeñan el rol de cliente y dos el de servidor. En la sección de *attachments* se definen las relaciones entre los componentes a través de sus roles *ProvideM* y *RequireM*. La instancia del sistema se llama *basesys.*

Tras la realización de estos tres pasos de la etapa 2, se obtiene la descripción arquitectónica del *sistema inicial*, en la cual los aspectos (que representan los nuevos requisitos a insertar) no han sido considerados aún.

Esta etapa puede obviarse si el sistema ya existe, y por tanto su especificación y diseño se han realizado con anterioridad. En este caso, habría que estudiar la información generada.

```
Component Basesystem {
interface none;
composition
resroomhandler,customerapp: Client;
roomhandler,restauranthandler: Server;
attachments
resroomhHandler.requireM(retrieve,typeroom,cant,total)<>
roomhandler.provideM(retrieve,typeroom,cant,total);
customerapp.requireM(query,resp)<>restauranthandler
.provideM(query,resp);
}
component Client {
interface
requireM:RequireM;
}
component Server {
interface
provideM:ProvideM;
}
Role ProvideM(operation,param){
spec is
operation?(answer).(value)answer!(value).
ProvideM(operation,param);
}
Role RequireM(operation,param){
spec is
(answer)operation!(answer).answer?(value).
RequireM(operation,param);
}
instance basesys:Basesystem;
```

Figura 5.5. *Sistema inicial* en LEDA.

5.4. Inclusión de un nuevo requisito como un aspecto

Después de describir la arquitectura del *sistema inicial* se incorporan los aspectos (nuevos requisitos) para obtener un nuevo sistema aplicando el paradigma de DSOA. Para ello se propone redefinir la arquitectura del *sistema inicial.*

Los aspectos que atraviesan el sistema especificado pueden considerarse como unidades modulares; en particular, *AOSA Model* propone que estos requisitos se conviertan en aspectos arquitectónicos y se describan como componentes arquitectónicos.

5.4.1. Especificación de los aspectos

Creado el modelo de diseño para el *sistema inicial*, en una segunda iteración del proceso evolutivo que se propone, se redefine la especificación del sistema para incluir

los aspectos. Para ello, primero se deben describir y especificar, y después incorporarlos al diagrama de casos de uso del *sistema inicial.*

Nuevos requisitos como aspectos

El *sistema extendido* se define insertando los aspectos en el *sistema inicial* y manteniendo el principio de inconsciencia propuesto por Filman y Friedman en [FiFr00]. Según esta propuesta, el código asociado a los nuevos elementos permanece independiente del resto del sistema manteniendo un bajo acoplamiento con los componentes del *sistema inicial*. Esto lleva a la definición de los párrafos que siguen para completar definición de la arquitectura:

1. Inicialmente un aspecto se considera en la fase de especificación como un tipo especial de caso de uso: *use case extension* [JaNg05].
2. Hay que identificar los puntos donde interacciona un aspecto que va a ser incorporado con los casos de uso del *sistema inicial*. Estos puntos son los puntos de extensión o *extension points* [JaNg05] (denominados *match points* en [BrMo03]).
3. Es necesario especificar cómo interactúa cada aspecto con el sistema (sus condiciones de aplicación). Así, la definición de cada *extension point* supone determinar la siguiente información:
   - Eventos que desencadenan la ejecución de un aspecto (recibir un mensaje síncrono o asíncrono).
   - Las precondiciones o condiciones de ejecución que tienen que satisfacerse.
   - Cuándo (*when*) debe ejecutarse cada aspecto (antes, después, o en lugar de la acción que desencadena el evento).

   Esta información forma parte de la documentación asociada al caso de uso que se extiende y da lugar a la estructura *Common Items* (apartado 4.3.1.2).

Caso de estudio. Inclusión de nuevos requisitos

- Especificación de los aspectos

Los nuevos requisitos que se desea incluir en el sistema se modelan como aspectos y se pueden enunciar del siguiente modo:

1) *Un aspecto* Contador *que se encargue de contar el número de veces que se realiza una solicitud (de habitación o de mesas en el restaurante) y no ha sido satisfecha por no haber disponibilidad*.

2) *Se propone la inclusión del aspecto* Gestionar una lista de espera *asociado tanto a la reserva de habitaciones como a la gestión del restaurante del hotel y facilita a los clientes su inclusión en una lista de espera cuando hace una solicitud y no hay disponibilidad*.

3) *Un aspecto para* Encontrar habitación *si no hay disponibilidad en el hotel que realiza una búsqueda para localizar habitaciones disponibles en otros hoteles de la misma cadena*.

Nótese que, para que el sistema se comporte adecuadamente, el aspecto *Encontrar habitación* debe aplicarse antes que el aspecto *Contador* para evitar que se cuenten como solicitudes realizadas y no atendidas aquellas que finalmente sí se llevaron a cabo en otro hotel. Sin embargo, el arquitecto de software puede no ser consciente de este hecho cuando se especifiquen los requisitos del sistema, porque lo olvide, no se dé cuenta o bien porque ambos requisitos fueron introducidos en diferente momento y/o por diferentes personas. Además, el principio de inconsciencia contribuye a ocultar este hecho. Sin embargo, sería deseable que este problema se pudiera detectar lo antes posible para evitar arrastrarlo hasta la fase de implementación.

La metodología que se propone permite detectar esta situación de modo que el arquitecto de software, en tiempo de diseño, pueda asignar la prioridad de ejecución adecuada a los aspectos que, como en este caso, se han de aplicar en el mismo punto.

- Inclusión de aspectos en el diagrama de casos de uso

Cada aspecto a insertar en el sistema se considera como un caso de uso que extiende el comportamiento del *sistema inicial*. El diagrama de casos de uso de Figura 5.3 se amplía con los *use case extension* que se definen asociados a los aspectos descritos como requisitos en el párrafo anterior (Figura 5.6).

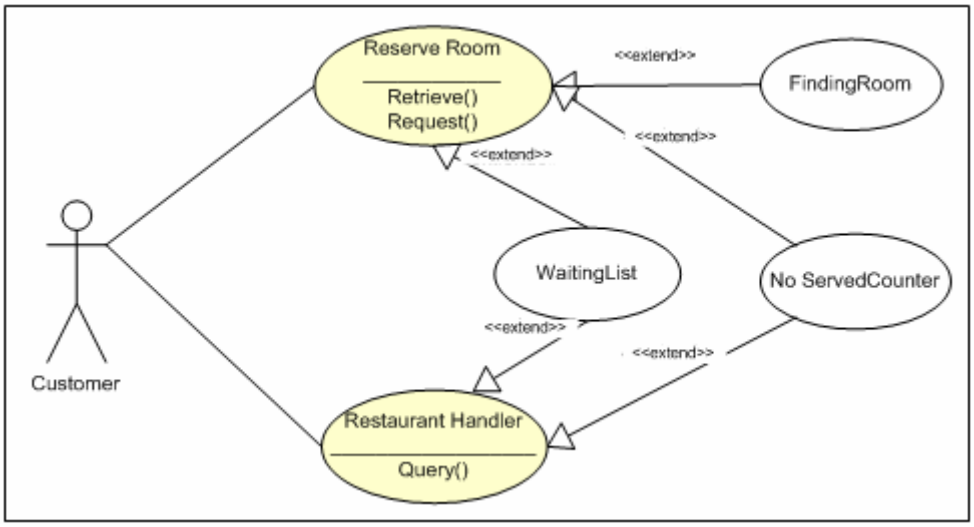


Figura 5.6. Extensión del diagrama de casos de uso con aspectos.

- Documentación asociada a los casos de uso

La información asociada a cada caso de uso extendido (Tabla 5.4a) se incorpora a la Tabla 5.3 (descripción de los casos de uso). La documentación de cada uno se muestra en la Tabla 5.4b.

	Caso de uso 1	Caso de uso 2
Caso de uso	Reservar habitación	Gestionar Restaurante
Nombre	*Reseve Room*	*Restaurant Handler*
Actor	Cliente (customer)	Cliente (customer)
Relaciones	Tres relaciones tipo *extend*	Dos relaciones tipo *extend*
Descripción	*Reserva habitaciones del hotel.* *Se comprueba la disponibilidad.* *Si está disponible, la reserva.* *Se notifica al cliente el resultado.*	*Reserva mesas en el restaurante para los clientes del hotel.*
Use cases extension	*NoSevedCounter,* *FindingRoom,* *WaitingList*	*NoSevedCounter,* *WaitingList*
Descripción	*Descripción de cada uno **	*Descripción de cada uno **
Extension points	*Request(), Retrieve()*	*Query()*
Tipo evento	*Recepción de mensaje síncrono*	*Recepción de mensaje síncrono*
Cond. de aplicación	*No disponibilidad (para los tres)*	*No disponibilidad (para los dos)*
Cuando (*when*)	*After(los tres)*	*After (los dos)*

Tabla 5.4a). Ampliación de la descripción de los casos de uso.

*** Descripción de los *use case extension***	
NoServedCounter	*Cuenta las veces que se realiza una solicitud (de habitación o de mesas en el restaurante) y no sido satisfecha.*
WaitingList	*Permite incluir a los clientes en una lista de espera cuando hace una consulta y no hay disponibilidad.*
FindingRoom,	*Si no hay disponibilidad de habitaciones en el hotel, busca una habitación disponible en otros.*

Tabla 5.4b). Descripción de los *use case extension* asociados a los nuevos requisitos.

5.4.2. Redefinición de los diagramas de secuencia asociados a la extensión

Como consecuencia de la extensión del sistema con aspectos y la inclusión de nuevos casos de uso que extienden el diagrama inicial, el diagrama de secuencia asociado a cada caso de uso que se extiende resulta modificado. Esto es así pues estos diagramas deben incluir el componente asociado al aspecto que se incorpora. Además, también se tienen que representar las nuevas interacciones, considerando que deben ser transparentes a los componentes del *sistema inicial*. *AOSA Model* propone la inclusión de un nuevo componente, *Aspect-Manager*, que contiene al componente de aspecto y la gestión de su

interacción[21], es decir, coordina y gestiona la interacción entre el aspecto añadido y el resto de los componentes que participan en ese escenario.

El comportamiento de *Aspect-Manager* se describe como sigue:

- *Aspect-Manager* está esperando la ocurrencia de un evento.
- Cuando detecta uno (solicitud de ejecución de una operación por parte de un tercer componente), lo intercepta.
- *Aspect-Manager* estudia si se cumplen las condiciones de aplicación del aspecto y cuándo se debe ejecutar éste.

La definición de *Aspect-Manager* de este modo permite que la incorporación de nuevos requisitos considerados como aspectos sea transparente a los componentes de diseño del *sistema inicial*. Por otra parte, y considerando las descripciones realizadas en el Capítulo 4, *Aspect-Manager* se define formado por una parte del proceso de control y el o los componentes de aspecto cuya interacción gestiona, esto es

Aspect-Manager = $\{ \mathcal{PC}^{\star}, \mathcal{CA}, \mathcal{New\_I} \}$; siendo:

$\mathcal{PC}^{\star} = \{ \mathcal{BB}, \mathcal{Co}_s, \mathcal{Co}, \mathcal{New\_I} \}$

$\mathcal{CA} = \{ \mathcal{A} \}$ es el conjunto de *Aspectos* que gestiona ese *Aspect-Manager*.

$\mathcal{New\_I}$ las nuevas interacciones definidas entre los componentes del *Proceso de Control* y los componentes *cliente* y *servidor* implicados en la inclusión de cada aspecto que gestiona.

$\mathcal{BB}$, $\mathcal{Co}_s$ son únicos para el sistema.

$\mathcal{Co}$ coordina las interacciones asociadas a $\mathcal{CA}$ para ese *Aspect-Manager*.

Finalmente, la inclusión de varios aspectos en distintos *Insertion Points* supone la definición de varios componentes *Aspect-Manager*.

Caso de estudio. Desarrollo

En el resto de la sección se describe detalladamente el caso de estudio considerando distintas situaciones:

- En los epígrafes A, B, y C se muestran los diagramas de secuencia obtenidos al insertar sólo un aspecto en el *sistema inicial*. Su estudio supone aplicar las consideraciones del caso 1 descrito en la sección 4.4.
- Los epígrafes D, E, y F describen la inclusión de dos aspectos: D es un ejemplo de la aplicación del caso 3; E representa la aplicación del caso 2 y E* de caso 2*; F aplica finalmente el caso 4.

En cada epígrafe se describe lo siguiente:

[21] Este elemento inicialmente se considera como un componente abstracto pero se refiere al *Nivel de aspecto* que define el modelo (apartado 4.3).

1. Redefinición de los diagramas de secuencia.
2. Definición de la estructura *CI* a partir de la información obtenida de los diagramas de secuencia.
3. Descripción de las reglas a aplicar.

La descripción de los *CI* y las reglas debe hacerse durante el desarrollo de la siguiente etapa de la metodología, según se describe en la siguiente sección. Sin embargo, se ha optado por presentar esta descripción en los epígrafes que siguen para que la explicación de cada caso sea más comprensible.

A) Caso de estudio: inclusión del aspecto Contador

A1) Redefinición de los diagramas de secuencia

El caso de uso identificado como *NoServedCounter* en Figura 5.6 representa un requisito que extiende el comportamiento del *sistema inicial* y, a partir de este momento, se va a considerar como el aspecto *Contador (Counter)*.

Figura 5.7 representa el diagrama de secuencia para el escenario más favorable del caso de uso relativo a la aplicación que realiza reservas de habitaciones (*Reserve Room*) cuando se extiende con un aspecto (en este caso, el aspecto *Contador*) y se sigue una aproximación no orientada a aspectos. Se puede observar que el componente *Gestor de Reservas* (*ResRoomHandler* en la figura) accede al componente *Contador* (*Counter* en la figura). Si se actúa de este modo, el código del componente *ResRoomHandler* tiene que ser modificado para que se pueda acceder desde él al componente *Counter*. En este caso no se observa el principio de inconsciencia.

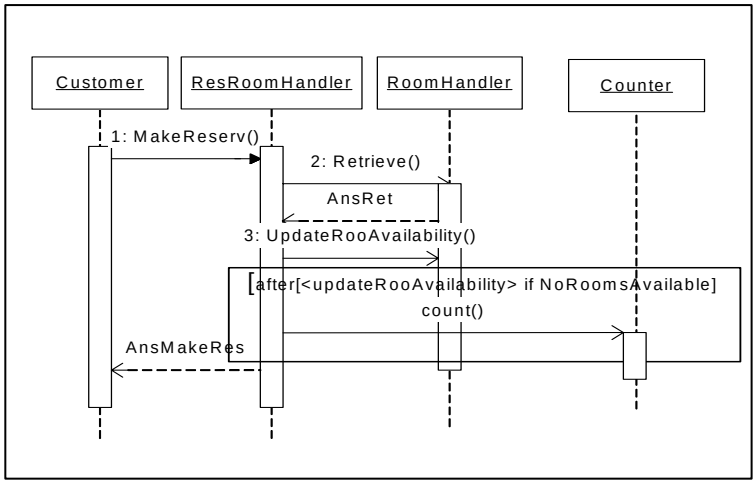


Figura 5.7. D. de secuencia para el aspecto *Counter*, siguiendo una aproximación No-OA.

En Figura 5.8 se muestra el diagrama de secuencia para el mismo escenario usando la metodología que se describe en este capítulo. Sin embargo, la interacción entre los componentes de aspecto aún no ha sido definida. Para representar la naturaleza de las nuevas interacciones *AOSA Model* propone la definición de *Aspect-Manager*.

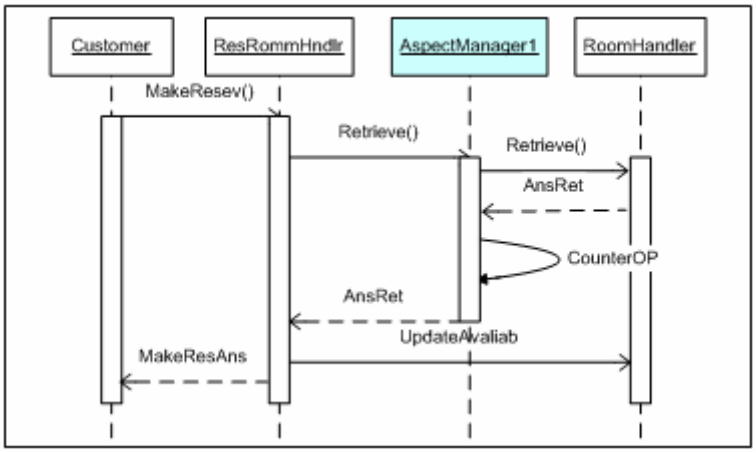


Figura 5.8. D. de secuencia para el aspecto *Counter* en el caso de uso *ResRoomHander* siguiendo *AOSA Model*, una aproximación OA.

En Figura 5.8 la llamada al método *Retrieve()* invocado por el componente *ResRoomHander* es interceptada por *Aspect-Manager*. Este componente analiza las condiciones de aplicación y cuándo debe aplicarse el aspecto (antes, después o en lugar del método llamado). En este caso, el aspecto debe ejecutarse después del método interceptado; por tanto, *Aspect-Manager* tiene que reenviar la llamada del método solicitado al componente *servidor, Gestor de Habitaciones (RoomHandler),* y chequear si la respuesta (disponibilidad-*availability*) es positiva o negativa. En caso de una respuesta negativa (no hay disponibilidad) se cumple la condición de ejecución del aspecto y debe ser ejecutado. Después de esto, el control del sistema vuelve al componente *ResRoomHandler* con el resultado de la operación solicitada (*Retrieve*). En esta figura la invocación del aspecto se representa por el símbolo de autodelegación en la línea de vida del componente *Aspect-Manager,* y se realiza después de la ejecución de la operación *Retrieve()* del componente *RoomHandler.*

Finalmente, interesa resaltar que los dos componentes que definen el comportamiento del sistema en este escenario son inconscientes de la actividad de *Aspect-Manager.*

A2) Estructura Common Items

Para el aspecto *Contador* se define la estructura *Common Items* que contiene su documentación asociada (Tabla 5.5).

Elemento	Descripción	Para *Contador*
Insertion Point - IP	Se refiere a cada puno de corte identificado en el diagrama de secuencia.	*Retrieve.*
Extension Point	Punto identificado en el caso de uso.	Llamada a *Retrieve*
Componente Serv	Nombre del componente servidor que proporciona el servicio asociado al IP.	Gest. de Habitación *–RoomHandler-*
Nombre del aspecto	Especifica el nombre del aspecto asociado a esta fila de la estructura de *Common Items.*	Contador *–Counter-*
Operación de aspecto	Nombre de la operación que ejecuta el aspecto.	Contar no atendida *–CounterOp-.*
Tipo Evento	Tipo de evento que dispara la aplicación del aspecto.	Recibir Mensaje Síncrono.
Cond. de aplicación	Condiciones que deben satisfacerse para que el aspecto se ejecute.	NoAvailable=True.
Cláusula When	Cuándo el aspecto debe o tiene que aplicarse.	Después (*After*).
Componente Cliente	Nombre de componente que requiere el servicio asociado al IP.	Gestor de Reservas *–ResRoomHandler-.*

Tabla 5.5. Tabla de *Common Items* para el aspecto *Counter*.

A3) Reglas a aplicar

Este caso, en el que se inserta un único aspecto en el *sistema inicial*, representa la situación descrita en el caso 1 de la sección 4.3: se aplica un aspecto (*Counter*) a un punto de corte (llamada a *Retrieve*) sobre un componente (*RoomHandler*). Es el caso más sencillo y su resolución supone aplicar una regla:

```
IF ev | IP OF Comp | cond THEN
     DO Aspect_Op OF Asp_Comp WHEN When_C
```

Donde:

ev : Recibir Mensaje Síncrono,
IP: R*etrieve*,
Comp: *RoomHandler*,
cond : NoAvailable=True,
Aspect_Op: Contar no atendida,
Asp_Comp: *Counter*,
When_C: *After*.

B) Caso de estudio: inclusión del aspecto Lista de Espera

B1) Redefinición de los diagramas de secuencia

Considérese ahora la extensión del sistema con el aspecto definido para gestionar la *Lista de Espera* (*WaitingList*) cuando se define asociado a la aplicación que realiza la gestión del restaurante (caso de uso *Restaurant Handler*). Estudiando el correspondiente caso de uso (Tabla 5.3a y 5.3b) se puede obtener un diagrama de secuencia similar al anterior (Figura 5.9). Sin embargo, ahora se incluye un componente *Aspect-Manager* diferente (*Aspect-Manager2*), que gestiona el aspecto *WaitingList*.

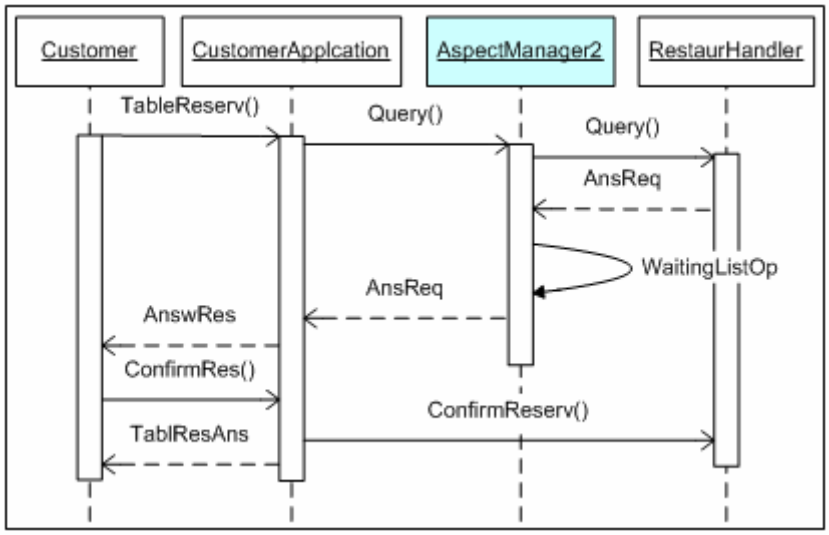


Figura 5.9. D. de secuencia para el aspecto *WaitingList* siguiendo *AOSA Model.*

B2) Estructura Common Items

La Tabla 5.6 muestra la estructura de *Common Items* para el aspecto *WaitingList*.

Elemento	Para *Lista de Espera*
Insertion Point -IP	*Query*.
Extension Point	Llamada a *Query*.
Componente Serv	Gestor de Restaurante -*RestaurantHandler*-.
Nombre del aspecto	GestionarListaEspera -*WaitingList*-.
Operación del aspecto	Actualizar Lista –*WaitingListOp*-.
Tipo Evento	Recibir Mensaje Síncrono.
Cond. de aplicación	Lleno=True.
Cláusula When	Después (*After*).
Componente Cliente	Aplicación de Cliente –*CustomerApplication*-.

Tabla 5.6. Tabla de *Common Items* para el aspecto *WaitingList*.

B3) Reglas a aplicar

Ahora, como en el caso anterior, se ha considerado nuevamente la inclusión de un aspecto. Representa igualmente la situación descrita en el caso 1: se aplica un aspecto *WaitingList* a un punto de corte (Llamada a *Query*) sobre un componente (*RestaurantHandler*). Su resolución supone aplicar una regla:

```
IF ev | IP OF Comp | cond THEN
     DO Aspect_Op OF Asp_Comp WHEN When_C
```

Donde:

ev : Recibir Mensaje Síncrono,
IP: *Query*,
Comp: *RestaurantHandler*,
cond : Lleno=True,
Aspect_Op: *WaitingListOp*,
Asp_Com: *WaitingList*,
When_C: *After*.

C) Caso de estudio: inclusión del aspecto Buscar Habitación

C1) Redefinición de los diagramas de secuencia

Para este tercer aspecto el proceso a seguir es el mismo y se obtendría el diagrama de secuencia de Figura 5.10.

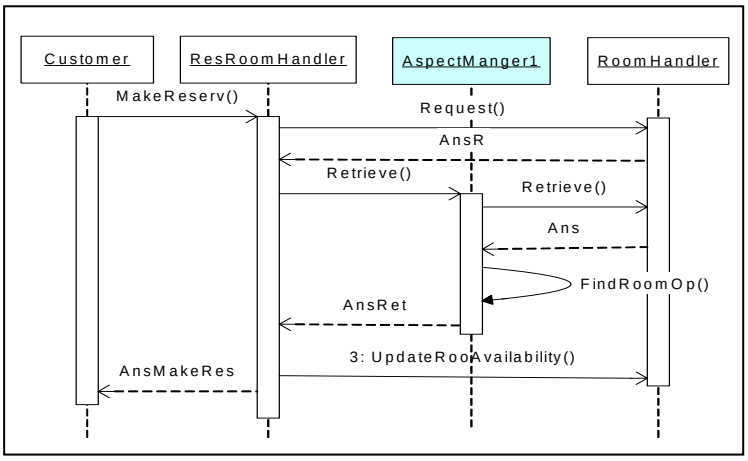


Figura 5.10. Diagrama de secuencia para el aspecto *FindRoom*.

C2) Estructura Common Items

La Tabla 5.7 muestra la estructura de *Common Items* para el aspecto *Buscar Habitación (FindRoom)*.

Elemento	Para *Buscar Habitación*
Insertion Point - IP	*Retrieve.*
Extension Point	Llamada a *Retrieve.*
Componente Serv	Gestor de Habitaciones -*RoomHandler*-.
Nombre del aspecto	Buscar Habitación - *FindRoom* -.
Operación del aspecto	EncontrarHabitación –*FindRoomOp*-.
Tipo Evento	Recibir Mensaje Síncrono
Cond. de aplicación	NoAvailable=True.
Cláusula When	Después (*After*).
Componente Cliente	Gestor de Reservas –*ResRoomHandler*-.

Tabla 5.7. Tabla de *Common Items* para el aspecto *FindRoom*.

C3) Reglas a aplicar

Este tercer caso también supone aplicar la regla:

```
IF ev | IP OF Comp | cond THEN
     DO Aspect_Op OF Asp_Comp WHEN When_C
```

Donde:

ev : Recibir Mensaje Síncrono,
IP: *Retrieve*,
Comp: *RoomHandler*,
cond : NoAvailable=True,
Aspect_Op: *FindRoomOp*,
Asp_Comp: *FindRoom*,
When_C: *After*.

Similar a estos tres casos es el estudio para las dos situaciones del caso de estudio no descritas: incluir *Lista de Espera* (*WaitingList*) para extender el caso de uso *Reserva de Habitación (ReserveRoom)* e incluir *Contador* (*Counter*) de solicitudes al *Restaurante* no satisfechas.

D) Caso de estudio: reutilización de un aspecto

Considérese ahora la inclusión de un aspecto *Counter* que actúa sobre los casos de uso *Reserve Room* y *Restaurant Handler*. Este caso supone reutilizar la especificación del aspecto como componente arquitectónico.

D1) Redefinición de los diagramas de secuencia

Se obtienen los diagramas representados en Figuras 5.8 (epígrafe A) y 5.11 (que se muestra a continuación de estas líneas). Son dos diagramas independientes pues los casos de uso afectados también lo son y el aspecto se define independiente del contexto. En este caso, el componente *Aspect-Manager* de Figura 5.8 (*Aspect-Manager1*) es distinto del que gestiona la inclusión del segundo aspecto en Figura 5.11 (*Aspect-Manager3*).

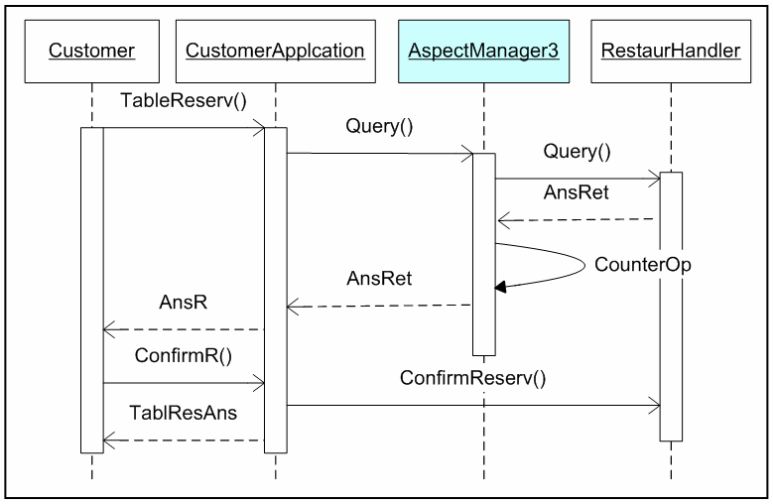


Figura 5.11. D. de secuencia para *Counter* en el C.U. *RestaurantHandler*.

D2) Estructura Common Items

La estructura *Common Items* en este caso tiene dos filas, una por cada uno de los *IP* en los que se inserta el aspecto (Tabla 5.8).

Elemento	**Para *Contador* *CU Reserve Room***	**Para *Contador* *CU Restaurant Handler***
Insertion Point - IP	*Retrieve.*	Consulta –*Query*-.
Extension Point	Llamada a *Retrieve.*	Llamada a *Query.*
Componente Serv.	Gestor de Habitación -*RoomHandler*-.	Gestor de Restaurante -*RestaurantHandler*-.
Nombre del aspecto	Contador –*Counter*-.	Contador –*Counter*-.
Operación del aspecto	Contar no atendida –*CounterOp*-.	Contar no atendida-*CounterOp*-
Tipo Evento	Recibir Mensaje Sinc.	Recibir Mensaje Sinc.
Cond. de aplicación	NoAvailable=True.	Lleno=True.
Cláusula When	Después (*After*).	Después (*After*).
Componente Cliente	Gestor de Reservas -*ResRoomHandler*-.	Aplicación de Cliente –*CustomerApplication*-.

Tabla 5.8. Tabla de *Common Items* para el aspecto *Counter* aplicado a dos casos de uso.

D3) Reglas a aplicar

Este caso supone la aplicación del caso 3: se aplica el aspecto *Counter* a un punto (Llamada a *Retrieve)* sobre un componente (*RoomHandler)*; y sobre otro punto (Llamada a *Query*) en otro componente *(RestaurantHandler)*. Su resolución supone aplicar dos reglas definidas para cada uno de los *IP* donde se aplica el aspecto:

```
IF ev1 | IP1 OF Comp1 | cond1 THEN
      DO Aspect_Op OF Asp_Comp WHEN When_C1
IF ev2 | IP2 OF Comp2 | cond2 THEN
      DO Aspect_Op OF Asp_Comp WHEN When_C2
```

Donde:

ev_1: Recibir Mensaje Síncrono,
IP_1: *Retrieve*,
$Comp_1$: *RoomHandler*,
$cond_1$: NoAvailable=True,
Aspect_Op: *Contar no atendidas*,
Asp_Comp: *Counter*,
When_C $_1$: *After*.

ev_2: Recibir Mensaje Síncrono,
IP_2: *Query*,
$Comp_2$: *RestaurantHandler,*
$cond_2$: Lleno=True,
Aspect_Op: *Contar no atendidas*
Asp_Comp: *Counter*,
When_C $_2$: *After.*

Similar a este caso es la inserción *Lista de Espera (WaitingList)* que extiende los casos de uso *Reserva de Habitación* (*ReserveRoom*) y *Gestión del Restaurante (RestaurantHandler)*.

Nótese que el componente de aspecto se reutiliza, pero se tienen dos instancias distintas. El componente de aspecto es un componente con estado.

E) Caso de estudio: inclusión de dos aspectos, en dos IP

La propuesta que se realiza en esta memoria es incremental e iterativa, entendiendo que los incrementos no tienen por qué ser unitarios. El resultado, por ejemplo, de dos incrementos unitarios es equivalente a un incremento doble (inclusión de dos aspectos). En los casos que se produzcan conflictos (inclusión de más de un aspecto en el mismo IP) es necesario establecer la prioridad de aplicación de cada uno en relación a los otros, como se discute en el epígrafe F.

A modo de ejemplo se muestra cómo sería el proceso de hacer un incremento no unitario, en el que los aspectos que se incorporan son independientes: considérese la inclusión del aspecto *Counter* sobre el caso de uso *ReserveRoom* y de *WaitingList* sobre el caso de uso *Restaurant Handler*.

E1) Redefinición de los diagramas de secuencia

En este caso, al insertar dos aspectos a la vez en el *sistema inicial* se obtienen dos diagramas de secuencia independientes, pues los casos de uso afectados lo son y se corresponden con los que se muestran en Figuras 5.8 y 5.9 descritos en los epígrafes A y B. Nótese que el componente *Aspect-Manager* de Figura 5.8 (*Aspect-Manager1*) es diferente del que gestiona la inclusión del segundo aspecto en Figura 5.9 (*Aspect-Manager2*).

E2) Estructura Common Items

Ahora esta estructura tiene dos filas, una por cada uno de los aspectos (Tabla 5.9).

Elemento	Para *Contador*	Para *Lista de Espera*
Insertion Point - IP	*Retrieve.*	*Consulta –Query-.*
Extension Point	Llamada a *Retrieve.*	Llamada a *Quero.*
Componente Serv.	Gestor de Habitación *-RoomHandler-.*	Gestor de Restaurante *-RestaurantHandler-.*
Nombre del aspecto	Contador *–Counter-.*	GestionarListaEspera *-Waiting List-.*
Operación del aspecto	Contar no atendida *-CounterOp-.*	*-WaitingListOp-.*
Tipo Evento	Recibir Mensaje Sinc.	Recibir Mensaje Sinc.
Cond. de aplicación	NoAvailable=Trae.	Lleno=True.
Cláusula When	Después (*After*).	Después (*After*).
Componente Cliente	Gestor de Reservas *-ResRoomHandler-.*	Aplicación de Cliente *-CustomerApplication-.*

Tabla 5.9. Tabla de *Common Items* para los aspectos *Counter* y *WaitingList*.

E3) Reglas a aplicar

Este caso supone la aplicación del caso 2. Esto es: i) insertar el aspecto *Counter* en un punto de corte, Llamada *Retrieve*, sobre el componente *RoomHandler*; ii) insertar un segundo aspecto, *Waiting List,* en otro punto, Llamada a *Query*, sobre el componente *RestaurantHandler*. Su resolución supone aplicar las dos reglas definidas anteriormente (en los epígrafes A y B) para cada uno de los aspectos pues son independientes uno del otro.

```
IF ev1 | IP1 OF Comp1 | cond1 THEN
      DO Aspect_Op1 OF Asp_Comp1 WHEN When_C1
IF ev2 | IP2 OF Comp2 | cond2 THEN
      DO Aspect_Op2 OF Asp_Comp2 WHEN When_C2
```

Donde:

ev_1: Recibir Mensaje Síncrono,
IP_1: R*etrieve*,
$Comp_1$: *RoomHandler*,
$cond_1$: NoAvailable=True,
$Aspect\_Op_1$: Contar no atendida,
Asp_Comp_1: *Counter*,
$When\_C_1$: *After*.

ev_2 : Recibir Mensaje Síncrono,
IP_2: *Query*,
$Comp_2$: *RestaurantHandler*,
$cond_2$: Lleno=True,
$Aspect\_Op_2$: *WaitingListOp,*
Asp_Comp_2: *WaitingList*,
$When\_C_2$: *After*.

Similar a este caso es el estudio para la inserción de *Lista de Espera*, que extiende el caso de uso *Reserva de Habitación,* junto a la inclusión del *Contador* de solicitudes al *Restaurante* no satisfechas; o la situación de la inserción de *Encontrar Habitación* en otro hotel, para extender el caso de uso *Reserva de Habitación*, junto a la inclusión del *Contador* o la *Lista de Espera* en el caso de uso de *Gestión del Restaurante*.

E) Caso de estudio: inclusión dos aspectos en dos IP (del mismo componente)*

Esta situación es un caso particular del descrito en el epígrafe anterior por lo que se muestra resumidamente:

Considérese la inclusión de un aspecto *Counter,* que actúa sobre el caso de uso *Reserve Room,* y del aspecto *WaitingList* sobre el mismo caso de uso. La aplicación de los dos aspectos es independiente pues están asociados a dos operaciones distintas (*Retrieve, Request*) sobre el componente *RoomHandler*. Los diagramas de secuencia que se obtienen se muestran en Figura 5.12a y 5.12b. Como los *IP* son distintos, los componentes *Aspect-Manager* también lo son: en Figura 5.12a, *Aspect-Manager1* gestiona la inclusión del aspecto *Counter*, mientras que en Figura 5.12b, *Aspect-Manager4* es el que gestiona *WaitingList*.

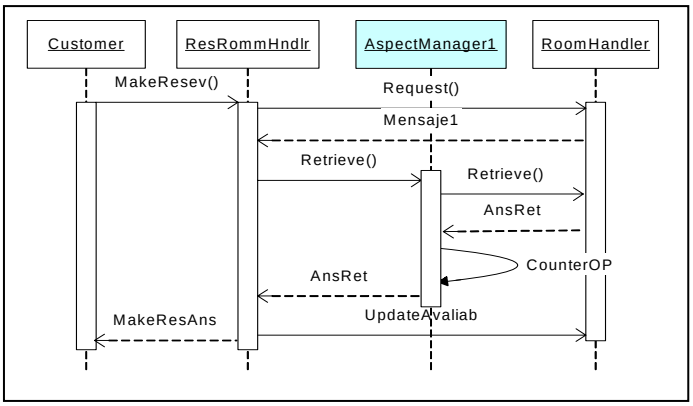


Figura 5.12a). D. de secuencia de aspecto *Counter,* CU *ReserveRoom.*

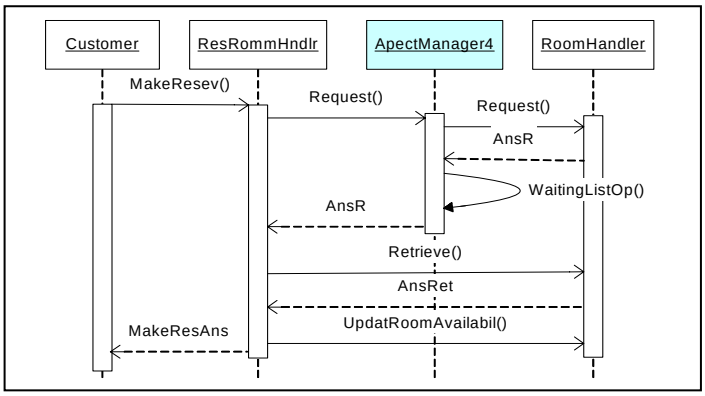


Figura 5.12b). D. de secuencia de aspecto *WaitingList,* CU *ReserveRoom.*

Como antes, la estructura *Common Items* tiene dos filas, una por cada uno de los *IP*; igualmente hay que aplicar dos reglas para la resolución del caso.

F) Caso de estudio: inclusión de dos aspectos sobre el mismo IP

Considérese ahora la inclusión de dos aspectos, *Counter* y *FindRoom,* que actúan sobre el caso de uso *Reserve Room* y sobre el mismo punto *IP* (Llamada a *Retrieve)*. La aplicación de ambos aspectos ahora no es independiente, pues están asociados a la misma operación (*Retrieve*) sobre el componente *RoomHandler*.

Este caso, la situación es más compleja que las descritas anteriormente debido a varias **causas**:

1) Puede que el arquitecto de software, al insertar un aspecto, no se percate que ya hay otro asociado a ese punto, introducido en un incremento anterior.
2) Puede que se de cuenta que se ha introducido un aspecto, pero que al introducir el segundo no tenga en cuenta el orden de ejecución entre ambos.
3) Sepa que hay un aspecto insertado ya en ese *IP* y tenga en cuenta el orden de ejecución.
4) Quiera introducir ambos aspectos en el mismo incremento.

Discusión:

1) En la primera situación, el *sistema extendido* se genera ignorando la prioridad aplicable en este caso, por lo que los resultados de la ejecución del prototipo pueden ser incorrectos. Es en ese momento cuando, al detectarse el error, si es que se descubre, el arquitecto se de cuenta del mal funcionamiento del sistema y corregirlo.
2) En el segundo caso, al igual que en el anterior, es responsabilidad del arquitecto indicar la prioridad de ejecución de los aspectos. Si ésta no se indica, al generarse el *sistema extendido* se pueden producir resultados inesperados, detectándose o no el error.
3) y 4) El arquitecto debe ser consciente de la situación y, por tanto, debe tener en cuenta que ambos aspectos tienen que ejecutarse en un orden determinado.

Solución:

1) y 2) Al apreciarse que el *sistema extendido* tiene un funcionamiento inadecuado y estudiar sus causas, la solución es generar el *sistema extendido* de nuevo, de modo que se considere la situación que ha llevado al error; esto es, la inclusión de los dos aspectos en el mismo *IP* y asignarles la prioridad adecuada. Para ello, se estudiará la información en la estructura *CI*.
3) y 4) El arquitecto, a la vista de los *CI*, debe asignar la prioridad de ejecución adecuada a cada aspecto.

Conclusión:

En cualquiera de los casos anteriores el componente *coordinador*, cuya definición propone el modelo (Capítulo 4) tiene que ser capaz de evaluar los aspectos insertados en el mismo punto *IP*. Como se indicó en 4.4.4, la resolución de las situaciones descritas supone aplicar una regla compuesta, y, según se dijo, esta circunstancia la resuelve un *coordinador complejo*, pues un *coordinador regular*, por construcción (aplicable en los demás casos), sólo atiende a un aspecto sobre un *IP*.

Si se consideraran dos *coordinadores regulares* para ejecutar los dos aspectos insertados sobre el mismo punto (incluso convenientemente ordenados), podría darse la siguiente paradoja: se ejecuta el primer aspecto (al que se asignaría mayor prioridad), siendo su condición *when* igual a *before*, y luego el segundo aspecto (el que correspondería a la prioridad menor) se ejecutará después, siendo su condición *when* igual a *before*. En este caso, ambos aspectos se ejecutarían antes que la operación interceptada, pero como cada coordinador (*coordinador regular*) atiende a un único aspecto, la operación interceptada se ejecutaría dos veces, ambas después de la ejecución del aspecto.

En este momento estamos trabajando en realizar la detección automática de esta situación (inserción de más de un aspecto en un punto) para alertar al arquitecto de software antes de que siga con el proceso de generación del *sistema extendido*.

F1) Redefinición de los diagramas de secuencia

En este caso se obtiene el diagrama de secuencia representado en Figura 5.13. Ésta representa el escenario más favorable del caso de uso relativo la reserva de habitaciones (*Reserve Room*) cuando se extiende con dos aspectos (*Counter* y *FindRoom*), pero cuando se sigue una aproximación no-OA. Se puede observar que el componente *ResRoomHandler* accede al componente *Counter* y a un gestor de Búsqueda Externa de Habitaciones (*RoomPublicsDB* en la figura). En este caso, el código del componente *ResRoomHandler* tiene que ser modificado para que se pueda acceder desde él al componente *Counter* y a *RoomPublicsDB*; no se observa el principio de inconsciencia.

Figura 5.14 muestra el diagrama de secuencia para el mismo escenario siguiendo la metodología que se describe en este capítulo. Como se puede observar en el diagrama y estudiando la tabla de *Common Items* (Tabla 5.9), ambos aspectos afectan al mismo punto; por tanto, hay que determinar el orden en el que deben operar. En la figura se representa que el aspecto *Counter* se ejecuta antes que *FindRoom.* En este caso, un mismo componente *Aspect-Manager* gestiona ambos aspectos.

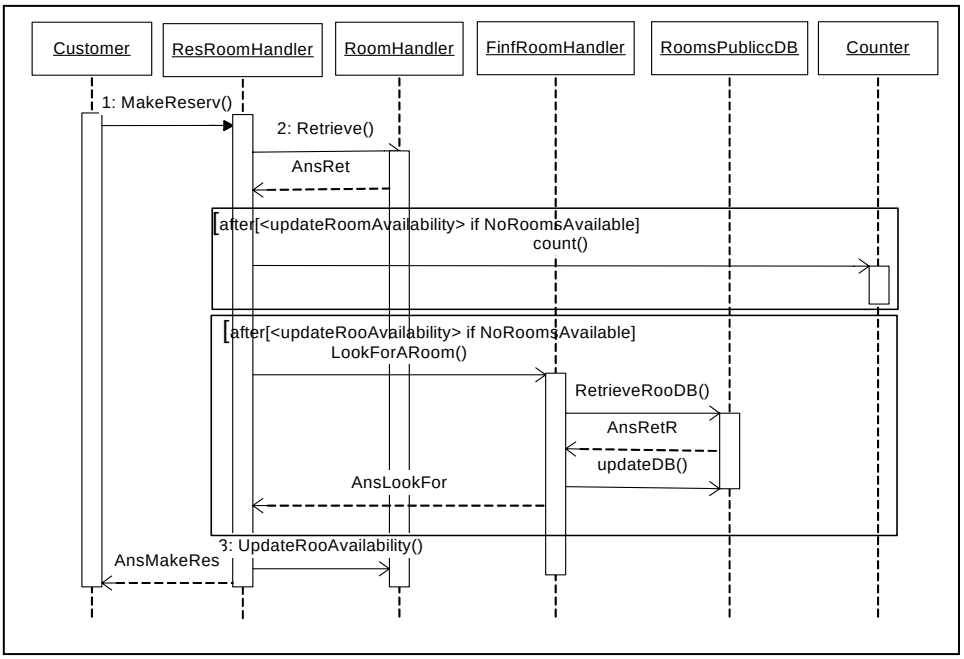


Figura 5.13. D. de secuencia para los aspectos *Counter y FindRoom* en el caso de uso *RoomHandler*. Aproximación No-OA.

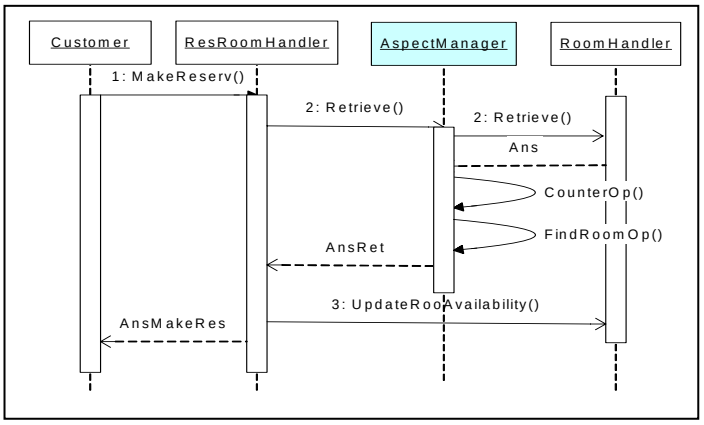


Figura 5.14. D. de secuencia para *Counter* y *FindRoom* en el caso de uso. *RoomHandler* Siguiendo *AOSA Model*.

Nótese que el orden en el que se están aplicando los aspectos no es el correcto. Este error se detectará posteriormente cuando, aún en fase de diseño, se observe que el comportamiento del sistema no es el esperado. Esto es así pues la metodología proporciona la posibilidad de ejecutar un prototipo del *sistema extendido* con aspectos (Sección 5.6). De este modo, se pueden descubrir errores que de otra forma sólo surgirían cuando el sistema estuviese totalmente implementado.

F2) Estructura Common Items

La estructura *Common Items* tiene dos filas, una por cada uno de los aspectos a insertar (Tabla 5.10).

Elemento	Para *Contador*	Para *Buscar Habitación*
Insertion Point - IP	*Retrieve.*	*Retrieve.*
Extension Point	Llamada a *Retrieve.*	Llamada a *Retrieve.*
Componente Serv.	Gestor de Habitación *–RoomHandler-.*	Gestor de Habitación *-RoomHandler-.*
Nombre del aspecto	Contador *–Counter.*	Buscar Habitación *-FindRoom -.*
Operación del aspecto	Contar no atendida *-CounterOp-.*	BuscarHabitación *–FindRoomOp-.*
Tipo Evento	Recibir Mensaje Sinc.	Recibir Mensaje Sinc.
Cond. de aplicación	NoAvailable=True.	NoAvailable=True.
Cláusula When	Después (*After*).	Después (*After*).
Componente Cliente	Gestor de Reservas *-ResRoomHandler-.*	Gestor de Reservas *–ResRoomHandler-.*

Tabla 5.10. Tabla de *Common Items* para los aspectos *Counter* y *FindRoom*.

F3) Reglas a aplicar

Este caso supone la aplicación del caso 4: se inserta el aspecto *–Counter-* en el punto Llamada a *Retrieve*, sobre el componente *RoomHandler;* y se inserta otro aspecto *–FindRoom-* en el mismo punto –Llamada a *Retrieve-* (en el mismo componente). Su resolución supone aplicar una regla compleja:

```
IF  ev |  IP OF Comp THEN
     IF (Cond1) THEN DO (Aspect_Op1 OF Asp_Com1) WHEN When_C1
     IF (Cond2) THEN DO (Aspect_Op2 OF Asp_Com2) WHEN When_C2
```

Donde:

ev: Recibir Mensaje Síncrono,
IP: R*etrieve*,
Comp: *RoomHandler*,

$cond_1$: NoAvailable=True,	$cond_2$: NoAvailable=True,
$Aspect\_Op_1$: Contar no atendida,	$Aspect\_Op_2$: Encontrar Habit,
Asp_Comp_1: *Counter*,	Asp_Comp_2: *FindRoom*,
$When\_C_1$: *After*.	$When\_C_2$: *After*.

G) Caso de estudio: inclusión de todos los aspectos

La aplicación del modelo a situaciones complejas supone la combinación de los casos anteriores. La extensión del sistema se puede hacer de forma incremental para facilitar el diseño del *sistema extendido* o insertando todos los aspectos de una vez; de cualquier modo, su estudio se hace uno a uno. El caso

que se desarrolla a continuación supone la inclusión de **tres aspectos** aplicados sobre **dos casos de uso** sobre un total de **5 *puntos de inserción*** *(IP)*. Se insertan, pues, todos los aspectos descritos en el enunciado del caso de estudio.

Figura 5.15 muestra el despliegue del s*istema extendido* para el caso de estudio completo. En ella se puede observar que resultan incluidos cuatro componentes *coordinadores*, para gestionar los tres aspectos propuestos en el enunciado. Igualmente aparecen las nuevas interacciones.

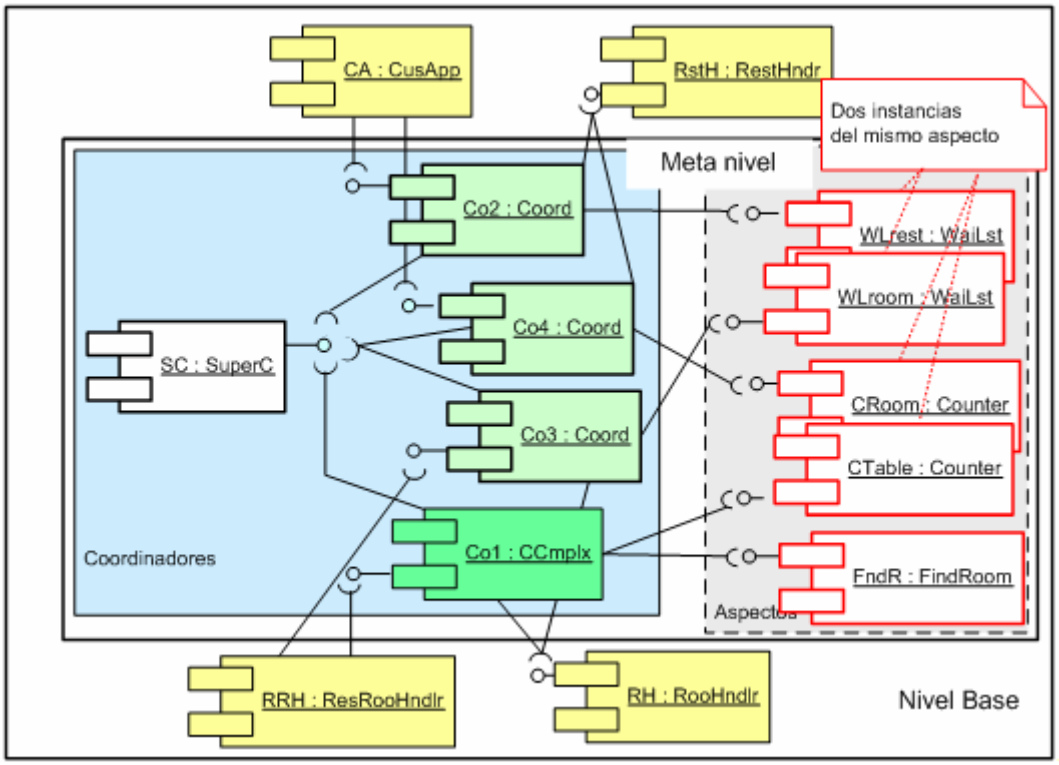


Figura 5.15. Despliegue final del *sistema extendido* para el caso de estudio.

Leyenda:

SuperC es representa el componente *SuperCoordinador*.
CA y **RRH** representan instancias los componentes *cliente*: *CustomerApplication* y *ResRoomHandler*.
RstH y **RH** son instancias de los componentes *servidor, RestaurantHandler* y *RoomHandler*.
Co2, **Co3**, **Co4** son instancias de la clase de componente *coordinador regular* (que atiende a un aspecto);
Co1 es una instancia de la clase *coordinador complejo* (que atiende a dos aspectos).
Se puede observar que hay 5 instancias de la clase de componente *aspecto*: dos del aspecto *Waitinglist* (*WLrest* y *WLroom*), dos del aspecto *Counter* (*CRoom* y *CTable*) y una del aspecto *FinRoom* (*FndR*).
Se ha omitido la presencia del componente *BlackBoard* para no complicar más la figura ya que no aporta nada en ella.

G1) Redefinición de los diagramas de secuencia

El desarrollo del caso de estudio completo supondría la definición de cuatro diagramas de secuencia, los correspondientes a:

- Figura 5.12b para representar la gestión asociada al *coordinador3*: inserción del aspecto *WaitingList* en el caso de uso *ReserveRoom* (epígrafe E*).
- Semejante sería la gestión para el *coordinador2*: inserción del aspecto *WaitingList* en el caso de uso *RestaurantHandler* (Figura 5.9 en el epígrafe B).
- Una figura similar a 5.8 llevaría a la definición del *coordinador4* para gestionar la inserción del aspecto *Counter* en el caso de uso *RestaurantHandler* (epígrafe A).
- Figura 5.14 muestra la gestión asociada al *coordinador1* que representa la inserción de dos aspectos *Counter* y *FindRoom* en el caso de uso *ReserveRoom* (epígrafe F).

G2) Estructura Common Items

La estructura *Common Items* tiene cinco filas, una por cada uno de los aspectos a insertar en el *IP* correspondiente (Tabla 5.11).

G3) Reglas a aplicar

En este caso habría que aplicar varias reglas:

- 3 reglas sencillas, que se corresponden a la aplicación de los apartados tres primero puntos de G1, en los que se aplica un aspecto.
- Reglas correspondientes al caso 4, que corresponde al caso en el que se aplican dos aspectos al mismo IP.

Elemento	**Para *Lista de Espera*** (para RoomHandler)	**Para *Lista de Espera*** (para RestaurantHandler)	**Para *Contador*** (para RoomHandler)	**Para *Buscar Habitación*** (para RoomHandler)	**Para *Contador*** (para RestaurantHandler)
Insertion Point - IP	*Request*	Consulta –*Query*	*Retrieve*	*Retrieve*	*Retrieve*
Extension Point	Llamada a *Request*	Llamada a *Query*	Llamada a *Retrieve*	Llamada a *Retrieve*	Llamada a *Retrieve*
Componente Serv	Gestor de Habitación (*RoomHandler*)	Gestor de Restaurante (*RestaurantHandler*)	Gestor de Habitación (*RoomHandler*)	Gestor de Habitación (*RoomHandler*)	Gestor de Restaurante (*RestaurantHandler*)
Nombre del aspecto	GestionarListaEspera (*Waiting List*)	GestionarListaEspera (*Waiting List*)	Contador (*Counter*)	BuscarHabitación (*FindRoom*)	Contador (*Counter*)
Operación del aspecto	Actualizar Lista (*WaitingListOp*)	Actualizar Lista (*WaitingListOp*)	Contar no atendida (*CounterOp*)	BuscarHabitación (*FindRoomOp*)	Contar no atendida (*CounterOp*)
Tipo Evento	Recibir Mensaje Sinc.	Recibir Mensaje Sinc.	Recibir Mensaje Sinc.	Recibir Mensaje Sinc.	Recibir Mensaje Sinc.
Cond. de aplicación	Lleno=True	Lleno=True	NoAvailable=True	NoAvailable=True	NoAvailable=True
Claúsula When	Después (*After*)	Después (*After*)	Después (*After*)	Después (*After*)	Después (*After*)
Componente Cliente	Gestor de Reservas (*ResRoomHandler*)	Aplicacion de Cliente (*CustomerApplication*)	Gestor de Reservas (*ResRoomHandler*)	Gestor de Reservas (*ResRoomHandler*)	Aplicacion de Cliente (*CustomerApplication*)

Tabla 5.12. Tabla de *Common Items* para los tres aspectos en los distintos IP.

5.5. Especificación arquitectónica del *sistema extendido*

La siguiente etapa de la metodología consiste en describir el *sistema extendido* en un Lenguaje de Descripción Arquitectónica adecuado. Para ello, se ha extendido un LDA convencional *LEDA* a fin de obtener un LDA-OA que permita al arquitecto de software expresar de un modo formal los conceptos del Desarrollo de Sistemas Orientado a Aspectos a nivel de arquitectura: la inclusión de los nuevos elementos para obtener un sistema orientado a aspectos (*sistema extendido)* ha llevado a considerar un conjunto de instrucciones de orden superior. Esto ha supuesto la definición de un LDA-OA: *AspectLEDA*, según se explica en el Capítulo 6.

Por otra parte, los nuevos elementos (*nivel de aspecto*) que se proponen en *AOSA Model* para ampliar la descripción del *sistema inicial* han de expresarse de un modo formal en el LDA. En concreto, los aspectos, los *coordinadores* y el *MetaCoordinador,* se definen como componentes del lenguaje *LEDA*, describiéndose también su interfaz y las nuevas interacciones.

Para diseñar el *sistema extendido* en un LDA-OA, a partir de la descripción arquitectónica del *sistema inicial* (en *LEDA*), el arquitecto de software ha de considerar la información almacenada en los *Common Items* y en el conjunto de reglas que se obtienen al seguir los pasos del modelo. Igualmente, para obtener la descripción, se ha desarrollado una herramienta (descrita en el Capítulo 6) que asiste al ingeniero de software a lo largo del proceso. Una vez descrito el *sistema extendido* en el LDA-OA, la herramienta traduce esta descripción arquitectónica al LDA en el que se apoya.

Caso de estudio. Especificación arquitectónica en AspectLEDA

La descripción del *sistema extendido* que se obtiene se muestra en el Capítulo 6.

5.6. Generación del prototipo

En el Capítulo 6 se muestra cómo se realiza la traducción de la descripción arquitectónica del *sistema extendido* expresada en el LDA-OA al LDA regular en el que se apoya. Asimismo, se ha desarrollado otra herramienta –*EVADeS*- (que se describe en el Anexo 2 de esta memoria) que facilita la generación automática de código Java para el nuevo sistema. También es posible ejecutar el código generado y obtener, en tiempo de diseño, la simulación del comportamiento del sistema OA.

Por otra parte, dada la base formal en la que se apoya *LEDA*, es posible demostrar las propiedades de la arquitectura del *sistema extendido,* así como comprobar su corrección.

Al realizar la traducción de LEDA a Java siguiendo un enfoque generativo [Can00], en las clases Java que se obtienen, el código de los métodos que expresan la funcionalidad del sistema están vacíos. Es en tiempo de implementación cuando se debe completar la descripción de los métodos de los componentes de diseño, pues inicialmente éstos no tienen que estar completamente especificados.

5.7. Validación de la arquitectura

La validación de un sistema consiste en asegurar que éste se comporta como el diseñador espera que lo haga. El comportamiento del sistema debe ser comparado con las expectativas que tiene el que lo está validando, razón por la cual la validación es un proceso inherentemente subjetivo [Val98].

A lo largo del proceso que se ha presentado, que se materializa en el Capítulo 6 al concretarse en un lenguaje OA, se realizan tareas de validación en distintos momentos:

a) Primeramente, para validar el modelo arquitectónico es necesario chequear la composición de cada uno de los elementos que intervienen en él, de manera que no existan discordancias en su declaración. Este análisis es necesario para impedir situaciones en las que un componente invoque métodos que no estén definidos en el componente destino del mensaje:

- En este caso, los analizadores de los lenguajes que se usan para especificar la arquitectura (LEDA para el *sistema inicial* y *AspectLEDA* para el *sistema extendido*), descritos en el Capítulo 6 realizan validaciones de tipo sintáctico y semántico para comprobar la corrección del código de entrada. Los analizadores avisan al usuario de la existencia de errores leves o graves o, en su caso, que no los ha habido. Dependiendo de la calificación del error, el proceso puede continuar o se detiene.
- Además, se realiza un análisis de la coherencia de los datos necesarios para producir el sistema final (apartado 6.5.2).
- La arquitectura final es gramaticalmente correcta pues la herramienta la construye a partir de una plantilla válida de la estructura de una arquitectura en LEDA.
- La arquitectura generada se valida por medio de un análisis de la compatibilidad de cada una de las conexiones dadas en la especificación LEDA [Can00]. Igualmente se realiza una verificación de las relaciones de herencia y extensión de los roles y componentes. De esta forma, se valida la arquitectura del *sistema extendido*, ya en LEDA, lo que permite comprobar que la especificación satisface determinadas propiedades.

b) Comprobado este primer tipo de coherencia, hay que comprobar la ejecución del sistema:

- La especificación arquitectónica en LEDA se traduce a Java de modo que se conservan las propiedades de la arquitectura, en particular de compatibilidad entre componentes.

- Tras la traducción a Java, es posible ejecutar un prototipo del sistema y comprobar si su comportamiento es el esperado. Hay herramientas de *model checking* que usan Java como "formalismo" [YAH].
- Por otra parte, se pueden generar especificaciones en cálculo π del código LEDA del *sistema extendido*, que se pueden simular utilizando un intérprete del cálculo o bien aplicar herramientas para estudiar el modelo construido (como las propuestas que figuran en la base de datos de Yahoda [YAH]). La especificación en cálculo π se puede usar como entrada para este tipo de herramientas, lo que permite ejecutar un chequeo más completo de la corrección del sistema que con la ejecución del prototipo.

5.8. Metamodelo para *AOSA Model*

En esta sección se describe el metamodelo diseñado para representar *AOSA Model*.

> "*Los metamodelos son modelos que hacen afirmaciones sobre el modelado. Más concretamente, un metamodelo describe la posible estructura de un modelo. De un modo abstracto, define las construcciones de un lenguaje de modelado y sus relaciones, así como las restricciones y las reglas de modelado. Sin embargo, un metamodelo no proporciona la sintaxis concreta del lenguaje de modelado. Así se dice que un metamodelo define la sintaxis abstracta y la semántica estática de un lenguaje de modelado*" [StVo06] (Figura 5.16).

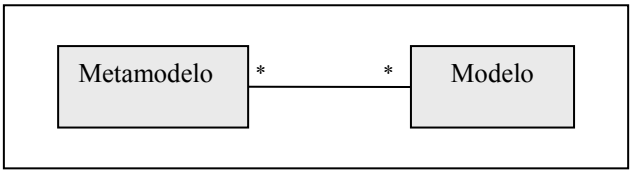


Figura 5.16. Relación entre modelo y metamodelo.

El metamodelo de *AOSA Model* se muestra en Figura 5.17; se ha representado mediante un conjunto de metaclases relacionadas entre sí, usando el diagrama de clases de UML 2.0. Las metaclases y la relación entre ellas definen la estructura y la información necesaria para describir modelos arquitectónicos siguiendo *AOSA Model*, tanto textuales (como el LDA-OA presentado en el Capítulo 6 de esta memoria) como visuales que incluyen la notación gráfica que se ha usado a lo largo de los Capítulos 4 y 5).

A continuación se muestra el metamodelo y los elementos que lo forman.

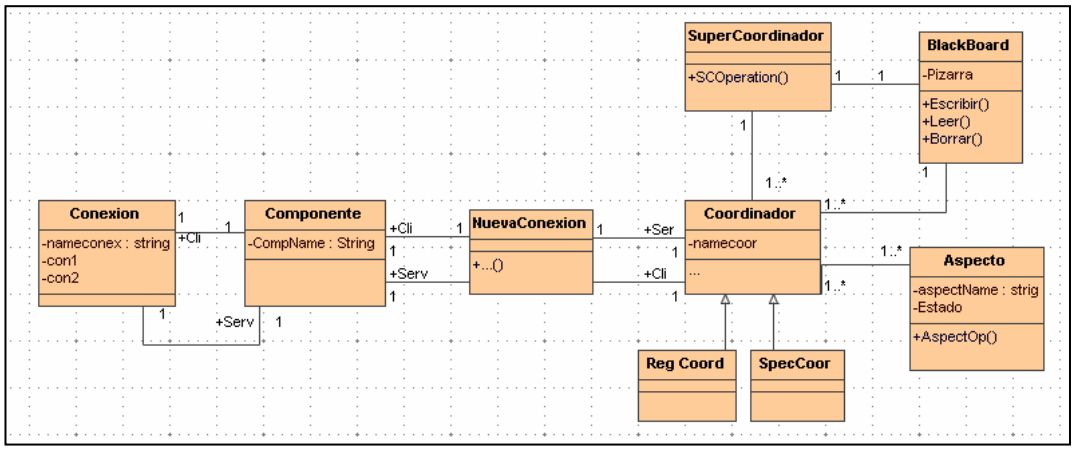


Figura 5.17. Metamodelo que define *AOSA Model*.

5.8.1. Metaclase *Aspecto*

Los principales elementos que definen esta metaclase son (Figura 5.18):

- Un atributo que permite designar al aspecto. Además, los aspectos pueden necesitar almacenar información para llevar a cabo su ejecución de un modo efectivo. Por eso la metaclase *Aspecto* tiene una relación de agregación con la metaclase *Atributo* de UML. Esta relación permite agregar/definir los atributos que componen un aspecto.
- Cada aspecto pueden ejecutar varias operaciones que, a su vez, pueden ser privadas o públicas. Las públicas son aquellas que el aspecto muestra en su interfaz; por ello, se define una asociación entre la metaclase *Aspecto* y la metaclase *Interface* de UML. En la figura esta operación se ha representado con el nombre *AspectOp*().

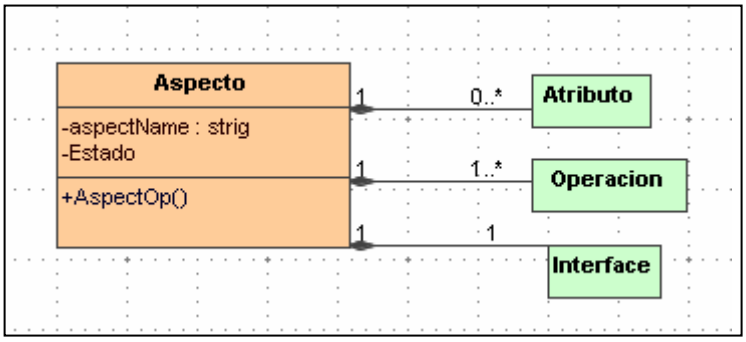


Figura 5.18. Metaclase *Aspecto*.

Restricciones:

- Todo aspecto proporciona al menos un servicio que representa un *advice* a ejecutar.
- Los aspectos no requieren operaciones de los componentes del *sistema inicial*.
- Un aspecto está sujeto a un conjunto de restricciones, que controla el elemento *coordinador*, que se define para gestionar su ejecución coordinada con los

componentes del *sistema inicial*. La información relativa a las condiciones de ejecución se almacena externamente a cada aspecto.
- Los servicios de un aspecto sólo son solicitados por el o los *coordinadores* que se definen asociados a él.
- Los aspectos son componentes arquitectónicos con estado.
- Al ser componentes heredan sus características generales de la metaclase *Component* de UML.

5.8.2. Metaclase *Coordinador*

Los *coordinadores* son los elementos encargados de realizar las tareas de coordinación y tejido entre los aspectos y los componentes del *sistema inicial*. Son componentes arquitectónicos por lo que heredan sus características generales de *Component*.

La metaclase *Coordinador* (Figura 5.19) se considera una clase abstracta que tiene dos especializaciones que permiten, a su vez, definir las características de los dos tipos fundamentales de coordinadores: *RegCoord* y *SpecCoor*. *RegCoord* es una metaclase que representa aquellos *coordinadores* que se definen asociados a un aspecto y un componente con un *IP* (*coordinador regular*). *SpecCoord* representa los *coordinadores* que tienen asociados más de un aspecto sobre el mismo *IP*, por lo que su comportamiento es más complejo (*coordinador complejo*).

Los principales elementos que definen esta metaclase son (Figura 5.19):
- Un atributo que permite asignarle nombre (*namecoor*).
- El comportamiento del *coordinador* viene definido por una operatividad que es diferente en las dos especializaciones. Sin embargo, todo *coordinador* tiene cuatro operaciones en su interfaz: tres requeridas y una proporcionada.
  - (*oprequired*) define la asociación con el componente que tiene un punto de enlace al que está asociado.
  - Una (o varias) operación requerida que define la conexión con el o los aspectos cuya ejecución coordina.
  - Una operación requerida define la asociación con el *MetaCoordinador* o *SuperCoordinador* (*calltoSC*).
  - Una operación proporcionada (*opprovided()*) que especifica el redireccionamiento de la operación interceptada (*oprequired*) para devolver el control al *sistema inicial*.

5.8.3. Metaclase *MetaCoordinador*

Este elemento se ha definido en *AOSA Model* para reducir la carga de trabajo de los *coordinadores* en tiempo de ejecución, realizar la evaluación de las condiciones de un modo centralizado y mantener la integridad. *MetaCoordinador* o *SuperCoordinador* es un componente arquitectónico por lo que hereda de *Component* sus características generales (Figura 5.19).

- Realiza la operación *SCOperation* para determinar si coinciden las condiciones de ejecución de un aspecto (en tiempo de ejecución) con las especificadas en las reglas que lo definen.

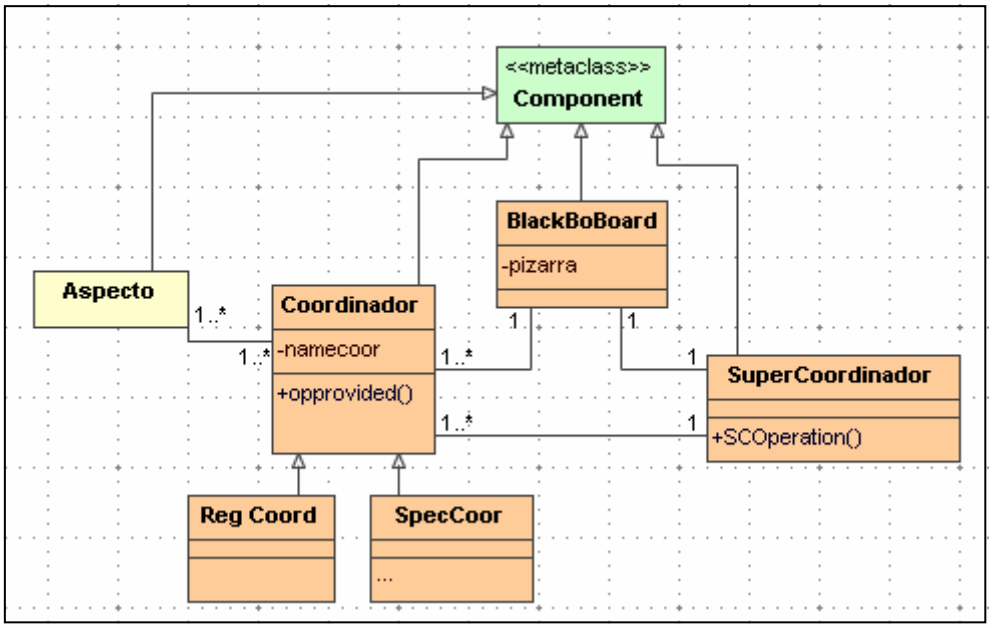


Figura 5.19. Metaclases Coordinador, SuperCoordinador y BlackBoard.

5.8.4. Metaclase *BlackBoard*

Este elemento almacena la información que se produce en tiempo real. Es utilizado por los *Coordinadores* y el *SuperCoordinador*. *BlackBoard* es un componente arquitectónico por lo que hereda de *Component* sus características generales (Figura 5.19).

- Realiza las operaciones *Escribir, Leer y Borrar* en la estructura Pizarra que es su atributo fundamental, cuando lo solicitan *SuperCoordinador* y *Coordinador*.

5.8.5. Metaclase *NuevaConexion*

Esta metaclase define las nuevas interacciones entre componentes del *sistema inicial* y los nuevos elementos de la arquitectura que se va a especificar, en particular los *coordinadores*.

Al tratarse de conexiones entre componentes arquitectónicos, la metaclase que define una nueva conexión hereda de la metaclase *Connector* de UML (Figura 5.20). Por tanto, la metaclase *NuevaConexion* tiene los atributos y operaciones que hereda de *Connector*.

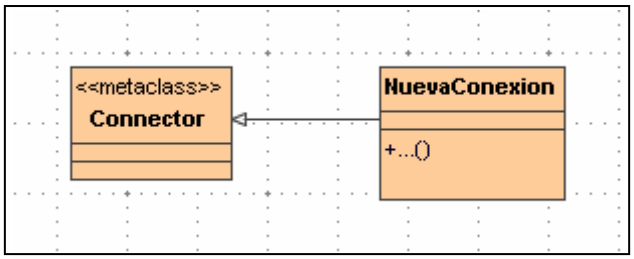


Figura 5.20. Metaclase *NuevaConexion*.

5.8.6. Sistema base

El *sistema base o sistema inicial* está formado por un conjunto de *componentes* y *conexiones*. Este elemento no aparece como tal en el metamodelo, aunque sí se muestran las metaclases para resaltar las relaciones existentes entre los componentes y las conexiones entre ellos, y los componentes y las nuevas conexiones definidas (Figura 5.17).

- Metaclase *Componente*
  - Esta metaclase tiene un atributo que permite asignarle nombre (*NameComp*).
  - Los componentes pueden realizar varias operaciones (Figura 5.21) que, a su vez, pueden ser privadas o públicas. Las públicas son aquellas que el componente publica en su interfaz; por ello se define una asociación entre la metaclase *componente* y la metaclase *interface,* y las metaclases *componente* y *operación.*

  Un componente puede ser simple o compuesto, y los sistemas son componentes compuestos por lo que heredan todas sus propiedades.

- Metaclase *Conexión*
  - Esta metaclase (Figura 5.22) tiene dos atributos que identifican los extremos de la conexión (*con1* y *con2*). Puede tener un nombre que la identifica (*nameConex*)
  - Proporciona una operación para crearlo (*newConex).*

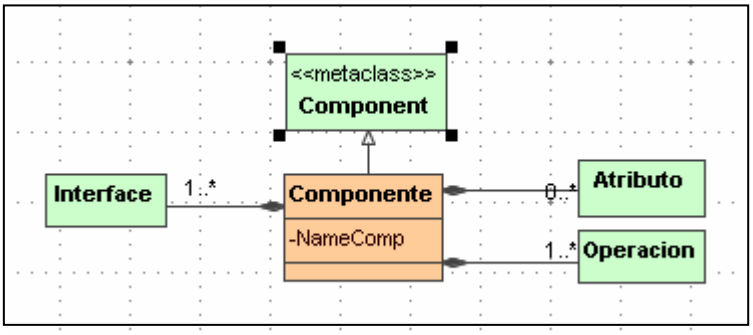


Figura 5.21. Metaclase *Componente*.

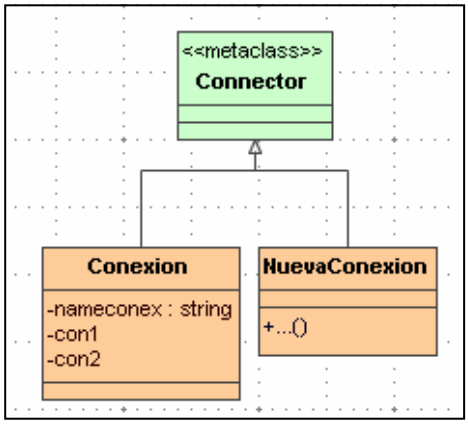


Figura 5.22. Metaclase *conexión* y conexiones en el modelo.

5.9. Conclusiones del capítulo

En este capítulo se ha presentado una metodología para desarrollar sistemas orientados a aspectos siguiendo *AOSA Model*. La metodología permite al ingeniero de software llevar a cabo el desarrollo de sistemas software orientado a aspectos, haciendo especial hincapié en la arquitectura del sistema. Particularmente, la metodología facilita la inserción de nuevos requisitos en sistemas ya diseñados o en fase de diseño, cuando dichos requisitos se pueden considerar como aspectos. Los aspectos se describen como componentes arquitectónicos que se insertan en el sistema de un modo inconsciente a los componentes del mismo, según propone *AOSA Model*. Así, los aspectos se mantienen independientes del contexto, por lo que se potencia su reutilización.

Para expresar la arquitectura de sistemas OA se ha definido un LDA-OA, que se explica detalladamente en el siguiente capítulo. Además, la metodología es fácil de aplicar gracias al desarrollo de una herramienta que asiste al ingeniero de software a lo largo del proceso. La herramienta facilita la generación de código en un LDA regular y su posterior traducción a Java, de modo que la metodología, en su último paso, permite obtener, en tiempo de diseño, un prototipo del sistema generado, sin que los componentes del *sistema inicial* estén completamente definidos, actividad que se puede realizar durante la implementación.

La ventaja de utilizar esta metodología es que la redefinición del sistema se hace en tiempo de diseño: insertar nuevos requisitos como aspectos sólo supone redefinir el diseño incorporando nuevos elementos, sin modificar los del sistema existente.

Realizar modificaciones no supone cambiar el código; nuevamente, sólo hay que redefinir el diseño, incluyendo las actualizaciones y generando automáticamente otra vez el código del sistema. Esto facilita el mantenimiento, al no tener que recodificar; sólo hay que incluir los nuevos elementos modificando el diseño, mientras que el resto del sistema permanece inalterado.

Por otra parte, se promueve la reutilización de componentes y aspectos, ya que la definición de estos elementos es independiente del contexto. Sólo los *coordinadores* dependen de él y su estructura es generada por la herramienta.

La metodología propone un proceso incremental e iterativo, aprovechando además las características generativas del lenguaje regular en el que se basa la definición del nuevo LDA-OA.

Otra característica interesante de la metodología es que, al obtener un prototipo del sistema, se puede saber si el comportamiento del sistema es el esperado o se ha producido algún error; por ejemplo al asignar prioridades de ejecución a los aspectos, si éstos se aplican sobre el mismo punto, como ocurría en el epígrafe G) del apartado 5.4.2. Recuérdese que se habían insertado los aspectos *Contador* y *Buscar Habitación* en ese orden, cuando la prioridad asignada debería ser la contraria (para evitar contar solicitudes no satisfechas antes de buscar en otros hoteles). Además, dada la base formal de *LEDA*, se pueden aplicar herramientas para chequear la corrección de la arquitectura.

Finalmente, se puede concluir que esta metodología proporciona un conjunto sistemático de pasos para desarrollar sistemas OA, siguiendo un modelo asimétrico de orientación a aspectos. Esto permite insertar los aspectos como nuevos elementos en un sistema existente o en fase de diseño; de esta manera se facilita el mantenimiento al trabajar el ingeniero de software a nivel de arquitectura, sin tener en cuenta consideraciones de implementación. Los pasos de la metodología facilitan la obtención de la información necesaria para llevar a cabo el proceso de tejido y composición, y las herramientas desarrolladas facilitan igualmente el tratamiento automatizado de esta información.

CAPÍTULO 6
AspectLEDA: un LDA-OA

Del estudio del Capitulo 2 de esta tesis se puede deducir que desde que nació la arquitectura del software se tuvo la necesidad de crear una notación específica para describirla. Así surgen los Lenguajes de Descripción de Arquitecturas (LDA) como notaciones que describen y analizan, de manera formal, las propiedades observables de una arquitectura software. Los LDA permiten, por tanto, modelar una arquitectura, analizar si es correcta e, incluso, simular su comportamiento. En el Capítulo 3 se ha definido un aspecto como una unidad modular que está dispersa por la estructura de otras unidades funcionales; también se dice: ... los aspectos existen a lo largo de todo el ciclo de vida*... Un aspecto de diseño, en particular, es una unidad modular del diseño que se entremezcla en la estructura de otros elementos del diseño.*

En este capítulo se describe AspectLEDA, el lenguaje de descripción arquitectónica orientado a aspectos que se propone. Está basado en LEDA, y se convierte en una extensión de éste, de modo que permite incluir la especificación de aspectos en un sistema inicial definido en LEDA. Por otra parte, se describe una herramienta, AOSA Tool/LEDA, que facilita realizar los pasos necesarios para la obtención del prototipo de un sistema descrito en AspectLEDA, como extensión de un sistema inicial expresado en LEDA. También se realiza una comparación con otros LDA-OA.

6.1. Justificación

Como se comentó en el Capítulo 4, en [Nav+02] presentamos algunas consideraciones metodológicas sobre cómo afrontar la separación de aspectos en el diseño arquitectónico abordando el problema desde un punto de vista estructural. El estudio concluye que

> *la separación de aspectos, a nivel arquitectónico, tiene algunas similitudes con los problemas de coordinación, que se resuelven aplicando modelos de coordinación.*

Esta relación se hace patente si se considera que, por una parte la funcionalidad de los sistemas se representa, a nivel de arquitectura, como un conjunto de componentes y la interacción entre ellos mediante conectores arquitectónicos; y por otra, que la especificación de los aspectos identificados en el sistema es transversal a la especificación de los componentes, por lo que la interacción entre ambos elementos tiene que tratarse de un modo diferente.

Establecer la nueva interacción supone identificar en los componentes los puntos en los que se han extraído, o se van a insertar, los aspectos: especificación de los puntos de enlace; y especificar las conexiones entre éstos y los aspectos. Estos conectores han de describir, además, las condiciones relativas a la aplicación de cada aspecto pues son los que gestionan la ejecución del sistema con los aspectos insertados. En [Nav+02] se sugiere también el uso de modelos de coordinación para gestionar la interacción entre componentes funcionales y de aspecto; como tales modelos proponen y se materializa en *AOSA Model* unos *componentes coordinadores* gestionan la ejecución global del sistema, siendo éstos los conectores entre ellos.

Por otra parte, en la sección 3.4 ya se comentaron los requisitos que deberían cumplir los LDA para permitir la gestión de los *crosscutting concerns* usando abstracciones arquitectónicas, según se decía en [Nav+02]. Estos requisitos son:

- *Especificar los componentes funcionales con sus interfaces y la interconexión entre ellas. Esto lo proporcionan los LDA convencionales, pero además es necesario que los LDA (para representar las propiedades transversales) puedan especificar los puntos de enlace en esos componentes funcionales. Así, debería definirse un conjunto de nuevas primitivas para gestionar la especificación de los puntos de enlace.*
- *Especificar los aspectos. La morfología de éstos es diferente de la de los componentes funcionales pues no proporcionan servicios. Sin embargo, para realizar una modularización de los aspectos, éstos deberían especificarse como un tipo especial de componente, descritos como nuevas primitivas del lenguaje.*
- *Especificar las conexiones entre los puntos de enlace y los aspectos.*

Estas premisas se han tenido en cuenta en la definición del LDA-OA que se propone en este capítulo:

- Se pueden especificar los componentes funcionales y la interacción entre ellos.
- Se especifican los aspectos como componentes arquitectónicos.
- Se pueden especificar los puntos de enlace, y la relación entre éstos y los aspectos como una primitiva del lenguaje.

Por otra parte, para la definición de *AspectLEDA*, se ha creado una gramática que satisface el requisito de:

> *generar sistemas orientados a aspectos, a partir de un cierto sistema inicial al que extiende, de un modo transparente a los componentes que lo constituyen*.

Para ello, se aplicarán los conceptos propuestos en *AOSA Model*. La gramática que se define tiene una estrecha relación con la de *LEDA,* al ser el lenguaje que extiende. Una vez formalizada la gramática de *AspectLEDA* con las especificaciones necesarias, se creó un analizador para esta gramática a fin de reconocer y validar los programas escritos en él. Además, fue necesario generar un código adicional para almacenar los elementos definidos en el programa analizado (tabla de símbolos), comprobar que el sistema está bien formado o si tiene errores de consistencia.

Se ha utilizado un reconocedor del lenguaje *LEDA*, capaz de reconocer y operar con los elementos que componen un *sistema inicial* definido en *LEDA*. Esto es así pues para generar el sistema final se realiza una descomposición del código (descripción arquitectónica) del *sistema inicial,* para ampliarlo con los aspectos que se incorporan, así como con los elementos de coordinación.

Una herramienta, *AOSA Tool/LEDA*, proporciona una interfaz que permite crear, editar, interpretar, depurar, y componer sistemas basados en la especificación de *AspectLEDA*; de esta manera, se pueden definir ampliaciones para sistemas ya diseñados en *LEDA*. La herramienta asiste al diseñador en las tareas de extender con aspectos el sistema descrito*,* siguiendo el modelo arquitectónico propuesto en esta tesis doctoral.

Por tanto, este capítulo trata sobre el estudio, diseño e implementación de la gramática *AspectLEDA*, así como del reconocedor que interpreta los programas en este lenguaje, y del desarrollo de una herramienta que facilite la introducción de los datos necesarios y ayude al arquitecto de software a generar los sistemas finales.

6.2. Introducción a LEDA

LEDA [CaPiTr01] es un lenguaje de especificación para la descripción y validación de propiedades estructurales y de comportamiento de sistemas software. Es uno de los LDA más avanzados y tal vez el que alcanza mayores cotas de dinamismo. Esto se debe a que la interacción entre los componentes *LEDA* se indica directamente de manera formal mediante cálculo π.

En *LEDA*, la arquitectura se expresa como un conjunto de componentes, cuyo interfaz se segmenta en roles. La estructura de cada rol se especifica como una descripción en cálculo π, en la que los nombres hacen referencia a los elementos del

componente (sus roles) o son creados como variables auxiliares (*names*). Los componentes pueden ser atómicos o compuestos, pueden estar parametrizados y se conectan mediante conectores. *LEDA* define, además, tres tipos de conexiones: estáticas, reconfigurables y múltiples. Admite la extensión de componentes y roles, así como la definición de múltiples niveles de abstracción.

También admite la construcción de un componente como envolvente de otro ya existente (encapsulación de componentes), de modo que captura las interacciones de sus roles, sin necesidad de redefinirlos como propios. Este tipo de construcciones se denominan *adaptadores*. Éstos se definen como elementos similares a roles y descritos utilizando el cálculo π, capaces de interconectar componentes cuyo comportamiento no es compatible. El disponer de un mecanismo de adaptación de un componente a una interfaz que no sea compatible con la suya propia fomenta la reutilización de componentes de software.

La definición del lenguaje se completa con mecanismos de herencia y parametrización de componentes y roles, que aseguran la preservación de la compatibilidad. Mediante estos mecanismos, se puede reemplazar un componente en una arquitectura por otro que herede del primero, con la certeza de que esta sustitución no afectará a la composición del sistema. En este sentido, las especificaciones en *LEDA* pueden considerarse como patrones o marcos de trabajo arquitectónicos genéricos que pueden ser extendidos y reutilizados, adaptándose a nuevos requisitos según el entorno en el que se inserten y, por tanto, facilitando la reutilización de componentes.

El lenguaje *LEDA* está articulado en dos niveles, uno para la definición de *componentes* y el otro para la definición de *roles*. Los *componentes* representan los módulos del sistema y proporcionan una determinada funcionalidad. Los *roles* describen el comportamiento de dichos componentes; este nivel será utilizado para el prototipado, validación y ejecución de la arquitectura. Cada *rol* proporciona una visión parcial de la interfaz de comportamiento de un determinado *componente*. La arquitectura de un *componente* se indica mediante las relaciones que se establecen entre sus subcomponentes, lo que se expresa mediante un conjunto de conexiones entre los *roles* de dichos subcomponentes. *LEDA* no hace distinción, a nivel del lenguaje, entre componentes y conectores; es decir, los conectores se especifican como un tipo más de componentes, permitiendo que el lenguaje sea más simple y regular; y que a la vez tenga una mayor flexibilidad a la hora describir las arquitecturas.

Como la semántica de *LEDA* está escrita en términos de cálculo π, las especificaciones pueden ser analizadas y ejecutadas, permitiendo el prototipado de la arquitectura.

Por la vinculación de este lenguaje con *AspectLEDA*, en el Anexo 1 de esta memoria se presenta un resumen más extenso de su especificación.

6.3. *AspectLEDA*: un LDA-OA

Como se ha mostrado en el Capítulo 2 de esta memoria, hay varios LDA que se apoyan en una semántica formal que permite analizar y, algunas veces, verificar la estructura arquitectónica que definen; otros lenguajes permiten ejecutar un prototipo del sistema desde su diseño arquitectónico. Pero, desafortunadamente, estos LDA no soportan los conceptos de orientación a aspectos. En el Capítulo 3 se trató este paradigma y se mostraron algunos LDA–OA. Sin embargo, del estudio se dedujo que, en su mayoría, presentan carencias de diversa índole. Por ello se desarrolló *AspectLEDA*:

> AspectLEDA *es un LDA que permite integrar aspectos en un sistema descrito con un LDA convencional como es* LEDA. *La integración se hace a partir de las consideraciones de* AOSA Model *y de las características de los aspectos a integrar, teniendo en cuenta las premisas expuestas en el apartado 6.1*.

Con la definición de lenguajes de descripción arquitectónica OA se pretende dotar de soporte lingüístico a la definición arquitectónica de sistemas orientados a aspectos. En este caso, las especificaciones de un sistema orientado a aspectos expresadas en *AspectLEDA* se pueden traducir a *LEDA* y obtener un prototipo ejecutable en Java, puesto que *LEDA* lo permite; esto facilita al arquitecto de software comprobar si el comportamiento del sistema es el esperado. La especificación de *AspectLEDA* se ha realizado utilizando una gramática formal definida mediante los correspondientes analizadores léxico, sintáctico y semántico, utilizando las herramientas Flex y Bison. Cuando se introduce una especificación arquitectónica en *AspectLEDA,* el analizador devuelve la estructura y la composición del sistema, así como todos los datos correspondientes a los resultados de la interpretación. En los apartados siguientes se describen someramente las características de los analizadores diseñados.

6.3.1. Analizador Léxico

El analizador léxico de *AspectLEDA* se define mediante expresiones regulares y una tabla de palabras reservadas. De esta manera, al proporcionar una entrada al analizador, éste va reconociendo los tokens y devuelve una estructura de datos a la salida, que, a su vez, es la entrada al analizador sintáctico. Para implementar el analizador léxico de *AspectLEDA* se han definido las *expresiones regulares* que se muestran en Figura 6.1 y las palabras reservadas en Figura 6.2.

6.3.2. Analizador Sintáctico

Para modelar *AspectLEDA* se definió una gramática independiente del contexto mediante la notación *BNF* atendiendo a los requisitos presentados por el modelo *AOSA*. La entrada del analizador sintáctico es la salida del analizador léxico.

Una gramática está compuesta por un conjunto de símbolos terminales o tokens, un conjunto de símbolos no terminales y un conjunto de producciones. Los símbolos

terminales definidos en el analizador léxico para *AspectLEDA* son los que se muestran en Tabla 6.1; los símbolos no terminales de la gramática se muestran en Tabla 6.2.

- *barra = ["\"]* : define la barra invertida.
- *digito = [0-9]* : rango de dígitos entre el 0 y el 9, es decir un dígito del 0 al 9.
- *letra = [a-zA-Z]* : rango de letras del abecedario, minúsculas y mayúsculas.
- *espacios= ([]|[\t])+*:repetición de espacios en blanco y tabulaciones, minimo una ocurrencia
- *retorno = [\n]* : fin de línea o nueva línea.
- *carácter = {letra}|[_]* : define un carácter, el cual puede ser una letra o un guión bajo.
- *entero ={digito}+* : define un entero, que es más de un dígito (serie de dígitos).
- *cadena = '[^'\n]*'* : define una cadena, como una secuencia de cualquier símbolo y terminado con fin de línea.
- *identificador = {letra}({caracter}|{digito})** : define identificador como una serie de caracteres y digitos, pero siempre empezando por una letra, y como mínimo de longitud 1.
- *abrir_comentario1 = "/*"* : abrir sección de comentarios, forma /**/.
- *cerrar_comentario1 = "*/"* : cerrar sección de comentarios, forma /**/.
- *abrir_comentario2 = "(*"* : abrir sección de comentario, forma (**).
- *cerrar_comentario2 = "*)"* : cerrar sección de comentarios, forma (**).

Figura 6.1. Expresiones regulares para *AspectLEDA*.

- *AFTER* : condición temporal para la ejecución de aspectos.
- *AROUND* : condición temporal para la ejecución de aspectos.
- *ASPECT* : clase Aspecto, para instantación de componentes aspecto.
- *ATTACHMENTS* : sección de declaración de enlaces entre componentes.
- *BEFORE* : condición temporal para la ejecución de aspectos.
- *COMPONENT* : clase Componente, para instantación de cualquier componente.
- *COMPOSITION* : sección de declaración de composición de la arquitectura.
- *FALSE* : valor booleano falso para condición de ejecución de aspectos.
- *INSTANCE* : clase instancia, para instanciar el sistema orientado a aspectos.
- *PRIORITY* : sección de declaración de prioridad de ejecución entre aspectos.
- *RMA* : condición de sincronismo para la ejecución de aspectos.
- *RMS* : condición de sincronismo para la ejecución de aspectos.
- *SYSTEM* : clase sistema, para declarar clases de sistema orientado a aspectos.
- *TRUE* : valor booleano verdadero para condición de ejecución de aspectos.

Figura 6.2. Tabla de palabras reservadas para *AspectLEDA*.

entero	*letra*	*component*	*rma*	*around*
cadena	*barra*	*priority*	*rms*	*true*
carácter	*aspect*	*composition*	*after*	*false*
identificador	*attachments*	*instance*	*before*	*system*

Tabla 6.1. Símbolos terminales de *AspectLEDA*.

arquitectura	*identificador_operacion_destino*
declaracion_componente	*identificador_operacion_origen*
cuerpo_declaracion_clase_componente	*declaracion_extraelements*
declaracion_composicion	*declaracion_aspecto*
declaracion_sistema	*declaracion_prioridades*
lista_declaracion_aspecto	*lista_declaracion_prioridad*
identificador_variable	*declaracion_prioridad*
lista_parametros	*letra_unidad,*
lista_parametros_aspectos	*lista_directorios*
declaracion_parametros	*directorio*
declaracion_parametros_aspectos	*instantacion_sistema*
declaracion_ruta	*parametro_tipo_evento*
declaracion_enlaces	*parametro_condicion*
lista_declaracion_enlaces	*parametro_condicion_when*
declaracion_enlace	

Tabla 6.2. Símbolos no terminales de *AspectLEDA*.

A continuación se muestra la descomposición del lenguaje en secciones de código y, para cada una de ellas, el tipo de instrucciones que forman las descripciones *AspectLEDA*. En el Apéndice A de este capítulo se muestra la gramática completa.

Formato del Código AspectLEDA

Las descripciones *AspectLEDA* definen una arquitectura basada en *LEDA*; como en este lenguaje, un sistema está compuesto por un componente *sistema* y una serie de componentes y roles que lo forman. Para *AspectLEDA* se propone la estructura similar: una descripción arquitectónica en *AspectLEDA* ha de tener dos secciones principales:

- Sección de *definición de clase componente* del sistema *AspectLEDA*: se declara la clase que representará al sistema que se pretende crear a partir de un sistema *LEDA*.
- Sección de *instanciación de clase componente* del sistema *AspectLEDA*: se crea una instancia para la clase del sistema previamente definida, de tal manera que esta instancia representará al nuevo sistema.

Sección de *definición de clase componente*

Está compuesta por la implementación de un componente principal que contiene todos los demás elementos que forman el sistema. Su estructura es la siguiente:

```
component identificador_clase_sistema {
    ...
}
```

En *AspectLEDA* la declaración de la clase *componente* contiene los elementos que lo forman. Se han definido tres secciones dentro del código del componente principal: sección de *composición*, sección de *enlaces de componentes* y sección de *prioridad* de aspectos.

1. *Sección de composición* (Figura 6.3)

Empieza con el símbolo terminal *composition* y termina al encontrar la palabra que define la *sección de enlaces*. Esta sección contiene los componentes que componen el sistema en *AspectLEDA*; han de definirse:

- El *sistema inicial* (descrito en *LEDA*) mediante la clase *system*.
- El o los aspectos que vayan a incorporarse al sistema que se identifican mediante la clase *aspect*.

```
component identificador_clase_sistema {
composition
identificador_sistema_inicial : system:'ruta_fichero_sist_inicial;
identificador_aspecto1 : aspect : 'ruta_fichero_aspecto';
identificador_aspecto2 : aspect : 'ruta_fichero_aspecto';
...
}
```

Figura 6.3. Sección de composición.

La definición de la *sección de composición* tiene las siguientes restricciones:

- Sólo se puede declarar una variable de clase *system* para definir el *sistema inicial*.
- El sistema debe contener, al menos, una declaración de variable de la clase *aspect*.
- A continuación de las palabras clave *system* y *aspect* hay que indicar la ruta (*path*) en la que se encuentra el fichero correspondiente.

2. *Sección de enlaces de componentes* (Figura 6.4)

Va después de la *sección de composición*. Empieza con el símbolo terminal *attachments* y termina con el inicio de la *sección de prioridad*, en el caso de que ésta exista. Si no existe, termina con el cierre de llave '}' (fin del componente principal en el que se encuentran las secciones que se están declarando). En esta sección se describen los enlaces entre las operaciones de los componentes de aspecto declarados en la *sección de composición* y las operaciones de los componentes del *sistema inicial* que van a ser interceptadas por esos aspectos. En esta *sección de enlaces* hay que especificar ciertos parámetros (información contenida en la estructura *Common Items*).

```
attachments
id_sistema_inicial.id_componente1.id_operacion1(id_comp_cliente1)
 <<parametros_common_items1>>
           id_aspecto1.id_operacion_asp1(parametros_operacion1);
id_sistema_inicial.id_componente2.id_operacion2(id_comp_cliente2)
 <<parametros_common_items2>>
           id_aspecto2.id_operacion_asp2(parametros_operacion2) ;
    }
```

Figura 6.4. Sección de enlaces.

La *sección de enlaces* está compuesta por varias instrucciones de *declaración de enlace*, una por cada operación que resulte interceptada por los aspectos. A su vez, una instrucción de enlace se compone de tres partes:

i) La operación origen que se corresponde con

id_sistema_inicial.id_componente.id_operacion (id_comp_cliente)

Donde:

- *id_sistema_inicial*: es el identificador declarado para la clase del *sistema inicial* (clase *system*).
- *id_componente*: indica el componente del *sistema inicial* que contiene la operación que resulta interceptada (componente servidor).
- *id_operacion*: define la operación del componente del *sistema inicial* que va a ser interceptada por el aspecto.
- *id_comp_cliente*: contiene el nombre del componente que solicita la operación interceptada.

ii) La parte central corresponde a la expresión:

<<parametros_common_items>>

que especifica los parámetros o condiciones de ejecución del aspecto enlazado al componente *sistema_inicial*; van separados por comas. Estos son:

- *tipo_evento*: puede tomar los valores *rma* y *rms*. Indica si el aspecto se ejecuta de manera síncrona *(rms)* o de manera asíncrona (*rma*).
- *condición*: puede tomar los valores *true, false* y representa el valor que tiene que tomar un cierto atributo para que se ejecute un cierto aspecto. En los *Common Items* viene especificada la condición que se evalúa, y el valor que tiene que tomar, en tiempo real, para que el aspecto se ejecute.
- condición *when*: puede tomar los valores *before, after* y *around*, e indica cuándo se ejecuta la operación asociada al aspecto: antes, después o en lugar de la ejecución de la operación.

iii) La tercera parte de esta instrucción contiene información sobre la operación del componente de aspecto que intercepta la operación ejecutada por el componente *sistema_inicial:*

id_comp_asp.id_operacion_asp (parametros_operacion) ;

Donde:

- *id_comp_asp:* es el identificador del componente de aspecto implicado en la intercepción de la operación del componente *sistema_inicial*.
- *id_operacion_asp*: el identificador de la operación del aspecto.

- *parametros_operacion*: son los parámetros que necesita el aspecto. Su presencia es opcional

3. *Sección de prioridad de los aspectos*

Se inicia con el símbolo terminal *priority* y termina con el símbolo '}' que concluye la declaración del componente. Si sólo se ha declarado un aspecto para el sistema, esta sección no tiene sentido, por lo que no es necesario declararla (el analizador la ignora). Su formato es:

```
priority
id_componente_aspecto1>id_componente_aspecto2;  para dos aspectos
 }
```

Para dos aspectos, esta instrucción está compuesta por dos operandos y por un operador que asigna la prioridad; los operandos son los identificadores de los aspectos a los que se pretende asignar el valor de prioridad y el operador indica cual tiene la mayor prioridad. En el caso asp1>asp2, asp1 tendrá mayor prioridad que asp2.

Sección de *instanciación de clase de componente*

Esta sección comienza con el símbolo terminal *instance* y consta de una instrucción de instanciación de la clase que representa al *sistema extendido*. Se declara una instancia de la clase *sistema extendido*. El analizador sólo contempla la declaración de una instancia de clase de *sistema extendido*, ya que sólo se permite la definición de una clase de *sistema extendido* y por tanto no parece lógico tener varias instancias de una misma arquitectura. La sintaxis de la *sección de instanciación* es:

```
instance  identificador_sistema : identificador_clase_sistema;
```

La estructura completa de una descripción *AspectLEDA* se muestra en Figura 6.5.

```
component identificador_clase_sistema {
composition
  identificador_sistema_inicial:system : 'ruta_fichero_sist_inicial';
  identificador_aspecto1 : aspect : 'ruta_fichero_aspecto';
  identificador_aspecto2 : aspect : 'ruta_fichero_aspecto';

attachments
  id_sistema_inicial.id_componente1.id_operacion1(id_comp_cliente1)
   <<parametros_common_items>>
              id_aspecto1.id_operacion_asp1(parametros_operacion1);
  id_sistema_inicial.id_componente2.id_operacion(id_comp_cliente2)
   <<parametros_common_items>>
              id_aspecto2.id_operacion_asp2(parametros_operacion2);

priority
  id_componente_aspecto1 > id_componente_aspecto2;
  }
```

Figura 6.5. Estructura de una descripción en *AspectLEDA*.

6.3.3. Analizador Semántico

El analizador semántico se encarga de dar sentido a las expresiones del lenguaje definido mediante el analizador léxico y el sintáctico, según el contexto en el que se encuentren dichas expresiones, y según los valores de las variables de entorno del sistema. Por ejemplo, cuando se reconoce una declaración de aspecto, el analizador semántico es capaz de determinar el número de aspectos declarados para determinar cómo tratar dicha declaración. De esta forma, el analizador semántico es capaz de realizar las operaciones necesarias con los datos de entrada para satisfacer los requisitos del programa y por tanto del analizador.

En este apartado se enumeran las operaciones que se han considerado de mayor interés para describir la funcionalidad del intérprete *AspectLEDA*:

- *Buscar identificadores*: cada vez que se declara un identificador de una clase, se comprueba que no ha sido declarado ninguna otra con el mismo identificador. También se utiliza cuando se hace referencia a una instancia de clase, ya que se ha de referenciar una instancia de clase válida. Una vez obtenidos los valores de ejecución de la función, se opera según convenga en cada momento.
- *Manejo de la tabla de símbolos*: para almacenar todos los datos relevantes de una descripción en *AspectLEDA* se hace uso de la tabla de símbolos. Cada vez que se declara o instancia una clase, o se utiliza una sentencia de enlace o de prioridad, se ejecuta una serie de procedimientos para inicializar, comprobar, borrar o insertar datos en la tabla de símbolos. Las funciones para buscar identificadores lo hacen dentro de esta estructura.
- *Comprobación y gestión de errores:* se han definido rutinas para detectar un amplio número de errores en los sistemas definidos en *AspectLEDA*. También existen rutinas para tratar esos errores, ya sea ignorándolos o realizando las acciones necesarias; por otro lado, hay rutinas para emitir los mensajes de error que se haya encontrado.
- *Generación de resultados*: el analizador es capaz de generar resultados, para que el usuario los conozca tras la ejecución del sistema.
- *Manejo de ficheros*: la comunicación de los resultados generados por el analizador se realiza por medio de ficheros. Por ello, el analizador dispone de rutinas para la creación y almacenamiento de datos en ficheros de texto. De esta manera, el analizador comunica sus resultados al "exterior".

6.3.4. Descripción de una arquitectura *AspectLEDA*

Cuando el arquitecto de software va a realizar la descripción arquitectónica de un *sistema extendido* con aspectos siguiendo *AOSA Model* ha de obtener la información que se obtiene al seguir el modelo y necesita para llevar a cabo la extensión; esta información está almacenada en la estructura *Common Items*. Por otra parte, el arquitecto conoce el nombre y la descripción arquitectónica del sistema a extender (*sistema inicial,* expresado en *LEDA*) y el nombre de los aspectos que extienden el sistema (igualmente expresados

en *LEDA*). Con esta información, el arquitecto de software puede definir la estructura del *sistema extendido* usando *AspectLEDA*.

Figura 6.6 muestra, a modo de ejemplo, la descripción arquitectónica en este lenguaje del caso de estudio completo que se ha desarrollado a lo largo del Capítulo 5.

```
 1 Component Extnddsyst {
 2 composition
 3   basesystem: System;'ruta_fich_S_base';
 4     counterroom: Aspect; 'ruta_fich_Aspect1';
 5     countertable: Aspect; 'ruta_fich_Aspect1';
 6     findroom:Aspect; 'ruta_fich_Aspect2';
 7     waitinglistroom:Aspect; 'ruta_fich_Aspect3';
 8     waitinglisttable:Aspect; 'ruta_fich_Aspect3';
 9 attachements
10 basesystem.Roomhandler.Retrieveop(resroomhandler)
11     <<RMS,NoAvail=True,after>>counterroom.Countertop();
10 basesystem.Restauranthandler.Retrieveop(customerapp)
11     <<RMS,NoAvail=True,after>>countertable.Countertop();
12 basesystem.Roomhandler.Retrieveop(resroomhandler)
13     <<RMS,NoAvail=True,after>>findromm.Findroomop();
14 basesystem.Roomhandler.Request(resroomhandler)
15     <<RMS,Full=True,after>>waitinglistroom.Waitinlistop();
16 basesystem.Restauranthandler.Query(customerapp)
17     <<RMS,Full=True,after>>waitinglisttable.Waitinglistop();
18 }
19 priority counterroom>findroom;

20 instance extsys: Extnddsyst;
```

Figura 6.6. Ejemplo de un programa en *AspectLEDA*.

Nótese que el *sistema extendido* está compuesto por:

- Una declaración de la clase *System* (que representa el *sistema inicial*) en la línea 3.
- Cinco declaraciones de variables de la clase *Aspect* (líneas 4 a 8).
- Cinco declaraciones de asociaciones –*attachments*- (líneas 10 a 17).
- La declaración de prioridad afecta a los aspectos *contador* (de habitaciones) y *encontrar habitación.*

Finalmente es posible demostrar las propiedades de la arquitectura del *sistema extendido* una vez que se traduce a *LEDA* dada la base formal de éste (calculo π). Así, se puede obtener una especificación formal del sistema que se puede usar, a su vez, como entrada en herramientas que permiten validar la arquitectura generada (Apartado 6.7.7).

Figura 6.7 muestra *grosso modo* el proceso de traducción de una arquitectura en *AspectLEDA* a un programa Java, que proporciona un prototipo de la descripción inicial. La primera parte del proceso (Traducción de *AspectLEDA* a LEDA) la realiza la herramienta *AOSA Tool/LEDA*, descrita en este capítulo; la traducción de LEDA a Java la realiza *EVADeS*, que se describe en el Anexo 2 de esta memoria.

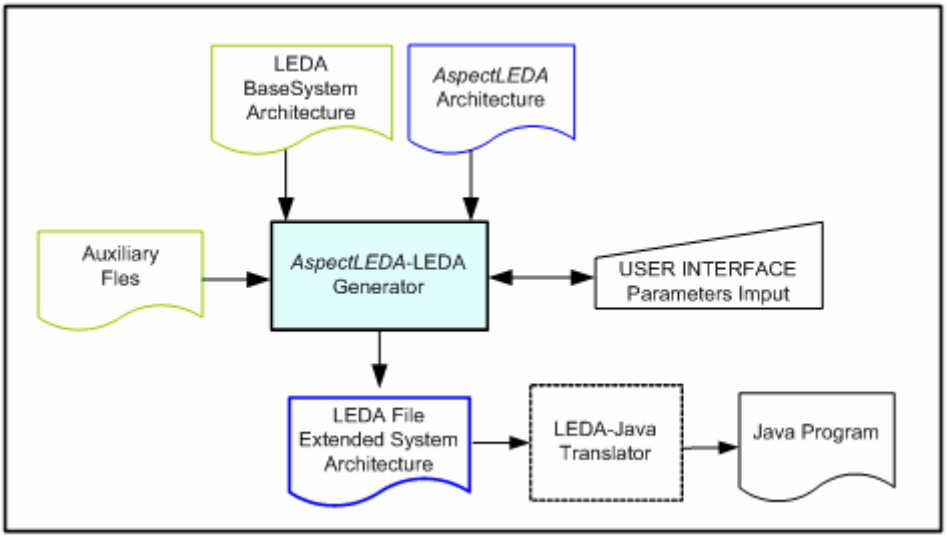


Figura 6.7. Esquema de proceso de traducción de *AspectLEDA* a Java.

6.4. Utilización de *LEDA*

Dado que la arquitectura del *sistema inicial* que se pretende extender está descrita en *LEDA*, se han utilizado analizadores de *LEDA* en el proceso de generación del *sistema extendido*: su interés viene determinado porque la arquitectura final del sistema se va a expresar en *LEDA* y el código de esta descripción se construye (se genera) a partir del código del *sistema inicial* (en *LEDA*), incorporando, en los lugares adecuados, el código *LEDA* asociado a la extensión (nuevos componentes y conexiones). Por ello, para poder insertar nuevo código *LEDA* en el ya existente, manteniendo la corrección de la nueva descripción, se han usado los analizadores de LEDA que permiten reconocer y manipular sistemas descritos en este lenguaje.

6.5. Comunicación entre intérpretes

Para llevar a cabo el tratamiento automático de la información generada por los intérpretes de ambos lenguajes (*AspectLEDA* y *LEDA*) se estableció un medio de comunicación entre ellos y la interfaz de usuario basado en ficheros. Primeramente, se almacenan las descripciones arquitectónicas del *sistema inicial* y del *sistema extendido* en sendos ficheros que se pasan a los intérpretes; una vez analizados, cada intérprete genera varios ficheros con los datos de salida, necesarios para obtener el *sistema extendido* en *LEDA*.

En esta sección se describen los ficheros de salida que se han generado y el concepto de *coherencia* entre los datos. Figura 6.8 representa cada uno de estos ficheros.

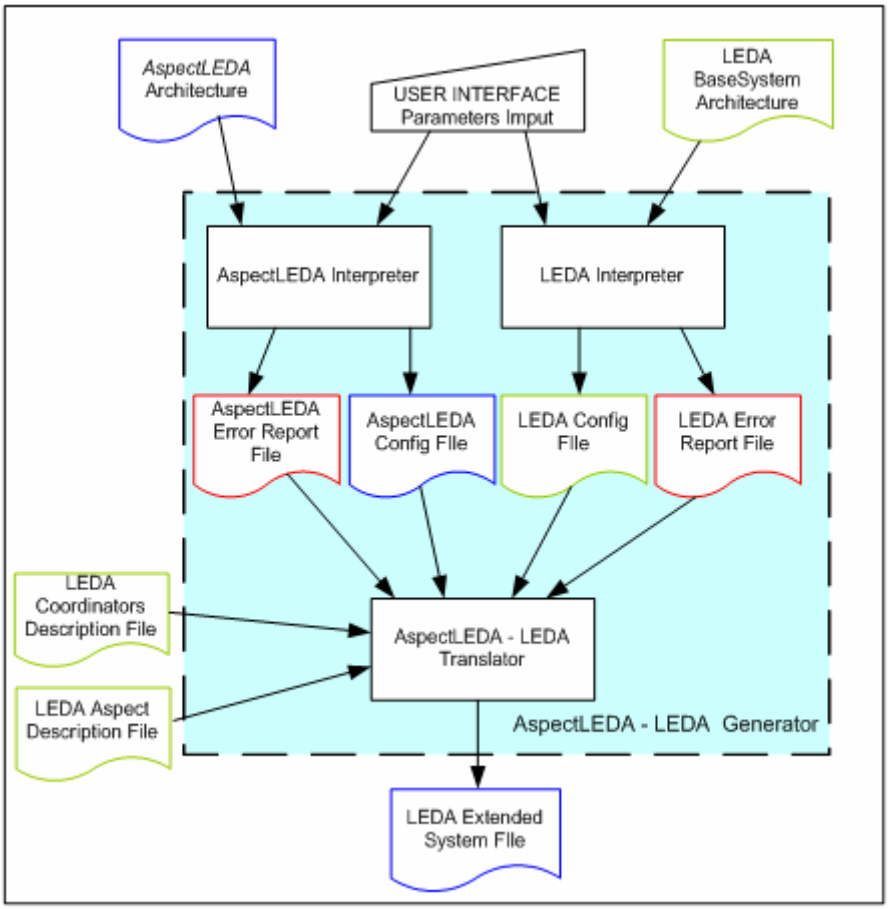


Figura 6.8. Ficheros de entrada y salida para la comunicación de los analizadores.

6.5.1. Ficheros de salida de los intérpretes

El tratamiento de las descripciones arquitectónicas en *AspectLEDA* y en *LEDA* genera los siguientes ficheros, que contienen información extraída del proceso de interpretación. Cada uno contiene un tipo de información de salida relevante para el arquitecto de software:

- *Ficheros de código de salida (.cfg)*: cada analizador sintáctico reconoce el código introducido a través del fichero de entrada y cada analizador semántico genera un fichero de salida con el código analizado.
  - La arquitectura en *AspectLEDA* (*AspectLEDA Architecture*) es analizada por el intérprete *AspectLEDA* generándose un fichero de configuración (*AspectLEDA Config File)* que es entrada al intérprete *AspectLEDA-LEDA*.
  - Por otra parte, el intérprete de *LEDA* tiene como entrada el *sistema inicial* en *LEDA* (*LEDA BaseSystem Architecture*) que genera un fichero de configuración (*LEDA Config File*) que también es entrada al intérprete *AspectLEDA-LEDA*.

- *Ficheros de errores y avisos (error.cfg)*: en ellos se almacenan los avisos y errores encontrados por el analizador semántico; esto permite realizar una primera validación del código de entrada:
  - Mensajes de error detectados por el intérprete *AspectLEDA* (*AspectLEDA Error Report*): el intérprete de *AspectLEDA* reconoce 21 errores (leves y graves), más uno genérico para otro tipo de errores no identificados.
  - Mensajes de error detectados para *LEDA* (*LEDA Error Report*): se identifican 11 errores, más uno genérico.

  Durante el proceso de generación es necesario determinar si se han producido errores, puesto que si los ha habido no se puede generar el sistema o bien la generación es incorrecta.

  La relación de errores que se detectan para ambos intérpretes se adjunta en el Apéndice B de este capítulo.

Además de los datos mencionados, es necesario considerar otra información adicional para poder generar la arquitectura final (*User Interface Data Input*, *LEDA Coordinators Description, LEDA Aspects Description* en la figura). Estos datos proceden de la información de la estructura *Common Items,* de los componentes *coordinadores* que propone el modelo y de los componentes de aspecto.

6.5.2. Comprobación de la coherencia

Además de la detección de errores en el código de los programas *AspectLEDA* y *LEDA*, se ha considerado interesante comprobar la corrección de los datos (de entrada o generados por los analizadores) que van a servir para componer el *sistema final*.

Hemos denominado *coherencia* del sistema a la correcta relación entre los datos obtenidos de los análisis realizados, de tal manera que los datos relativos al *sistema inicial* en *LEDA* y los relativos al sistema en *AspectLEDA* tengan una semántica coherente para la composición del *sistema extendido*; y que, por tanto, el *sistema extendido* que se genere en *LEDA* a partir de ambos sea correcto y esté bien definido a partir de esos datos. Un ejemplo de la *no coherencia* del sistema es declarar en el código *AspectLEDA* una instancia de una clase de *sistema inicial* (*system*) que no esté declarada en el código *LEDA*; el *sistema extendido* referenciaría a una clase inexistente. Esta situación se ha considerado como una *incoherencia*.

6.5.3. Descripción del proceso de traducción de *AspectLEDA*

El proceso interno de traducción de la descripción arquitectónica de un sistema expresado en *AspectLEDA* a *LEDA* se realiza mediante un intérprete (Figura 6.8): el *Generador de LEDA* que está constituido por dos elementos principales: Por una parte los *intérpretes de AspectLEDA y de LEDA*, que recorren los ficheros con extensión *aled* y

led, y reconocen el código *AspectLEDA y LEDA respectivamente*; y, por otra parte el traductor *AspectLEDA-LEDA*, que genera el *sistema extendido* en *LEDA,* a partir de la salida de los *intérpretes.* Cada uno de estos elementos lleva a cabo una fase del proceso de traducción.

1) Descripción de los *interpretes:*

* *Intérprete de AspectLEDA*. Este elemento tiene dos entradas:

   a) Un conjunto de parámetros que definen las características concretas del *sistema extendido*. El interfaz de usuario permite introducir esta información, que se ha deducido al seguir *AOSA Model* y, como se ha mencionado, constituyen la estructura *Common Items* propuesta por el modelo. El *Intérprete de AspectLEDA* completa una plantilla de la arquitectura en *AspectLEDA* (*AspectLEDA architecture*) usando para ello los parámetros que se han introducido. Como consecuencia se crea el fichero *AspectLEDA config*. (y el de errores si los hubiera).

   b) El arquitecto de software puede completar esta fase sin usar el interprete de *AspectLEDA* editando él mismo la plantilla de *AspectLEDA (AspectLEDA architecture*) para obtener el fichero *AspectLEDA config*, creando de este modo la arquitectura *AspectLEDA*.

* *Intérprete de LEDA.* Este elemento tiene como entradas:

   a) El fichero con el *sistema inicial*. El *Intérprete de LEDA* verifica la corrección del código introducido. Como consecuencia se crea el fichero *LEDA config*. (y el de errores si los hubiera).

2) La segunda fase del proceso de traducción de *AspectLEDA* la realiza el traductor *AspectLEDA-LEDA*. Éste tiene cuatro entradas:

   a) El fichero *AspectLEDA Config* obtenido tras la ejecución de la primera fase del proceso de traducción. Contiene la información necesaria para generar el *sistema extendido* en *LEDA*

   b) Un fichero *LEDA* (*LEDA Config*) con la descripción arquitectónica del sistema que se extiende (*sistema inicial*). Este fichero ha sido examinado por el *intérprete de LEDA* para asegurar la corrección de la especificación que representa.

   c) Un fichero *LEDA* que contiene los elementos que definen las nuevas interacciones (*Coordinador* y *MetaCoordinador*).

   d) Un fichero *LEDA* que contiene la descripción arquitectónica de los componentes de aspecto que extienden el sistema.

Durante la primera fase, el *generador de AspectLEDA* determina los nombres de los ficheros y el valor de los parámetros necesarios para, durante la segunda, generar la descripción arquitectónica del *sistema extendido*. Para ello, el *traductor AspectLEDA-LEDA* combina adecuadamente el código de los distintos ficheros *LEDA* para obtener el código de la arquitectura del nuevo sistema en código *LEDA* correcto. Si se han

producido errores, se notifica al arquitecto: errores fatales interrumpen el proceso, errores leves permiten que el proceso continúe.

Finalmente especificación del *sistema extendido* se puede expresar formalmente en cálculo π, que, a su vez, se puede usar como entrada en herramientas que permitan comprobar si el comportamiento del sistema es el esperado. Esta característica permite realizar una comprobación exhaustiva de la corrección del sistema, superior a la obtenida con la ejecución del prototipo. La especificación en cálculo π de la arquitectura *AspectLEDA* del caso de estudio se muestra más adelante (apartado 6.7.7).

6.6. Composición del *sistema extendido*

Con la información necesaria y comprobada la coherencia de los datos del sistema (expresados en ambos lenguajes) se tiene la certeza de que los valores de los datos interrelacionados son correctos y se pueden llevar a cabo las tareas de composición para generar la especificación del *sistema extendido*.

Para generar el nuevo sistema orientado a aspectos en *LEDA* a partir de las informaciones que se han introducido:

- Hay que incluir las descripciones de componentes y roles, procedentes del *sistema inicial* (en *LEDA*).
- Es necesario reestructurar los enlaces del sistema según se propone en *AOSA Model*.
- Se han de crear las clases e instancias para el nuevo sistema a partir de los datos disponibles.
- Hay que incluir las descripciones de los componentes de coordinación del sistema (propuestos por *AOSA Model*) que atienden a los eventos generados por los componentes afectados y controlan la ejecución de los aspectos, dependiendo de los eventos disparados.

Con esta información, la composición del *sistema final* se lleva a cabo redefiniendo y ampliando la estructura del *sistema inicial,* e incorporando los nuevos componentes y roles necesarios para completar la especificación del *sistema extendido.*

A continuación, se describe de una manera general cómo se lleva a cabo la generación de sistemas orientados a aspectos. El objetivo es dar al lector una visión global de cómo se realiza el proceso, desde el punto de vista del uso de la herramienta, que se ha considerado dividido en varias etapas:

1- Creación e inserción del código AspectLEDA

El arquitecto de software ha de describir un *sistema extendido* en *AspectLEDA* con las especificaciones requeridas y basándose en un *sistema inicial* escrito en *LEDA*. Tanto el sistema descrito en *AspectLEDA* como el descrito en *LEDA* son ficheros de texto con las extensiones *.aled* y *.leda*, que se pueden crear utilizando cualquier editor de textos o mediante el proporcionado por *AOSA Tool/LEDA*.

2- Llamada al intérprete/analizador AspectLEDA

El código del *sistema extendido* (en *AspectLEDA*) tiene que ser analizado por el intérprete para comprobar su corrección. Si no se detectan errores tras el análisis realizado por el intérprete de *AspectLEDA* puede continuar el proceso de generación del *sistema final.* Si por el contrario el intérprete ha detectado alguno, se comunica al usuario para que los corrija, antes de continuar el proceso.

3- Llamada al intérprete/analizador LEDA

Por otra parte, una vez introducido el programa en *LEDA* que describe el *sistema inicial*, se ha de realizar la llamada al intérprete *LEDA* para analizar su código y extraer los datos necesarios. Cuando el proceso de interpretación haya finalizado, la herramienta muestra, en una ventana, los errores detectados. Si se ha detectado alguno en la especificación de la arquitectura en *LEDA*, el ingeniero de software debe solventarlos antes de continuar con el proceso.

4- Extracción de datos

Por su parte, la herramienta extrae tanto del *sistema inicial* en *LEDA* como de la especificación en *AspectLEDA* la información necesaria para poder generar el *sistema extendido* también en *LEDA*. Esta información se presenta al usuario en diversos campos de la interfaz de la herramienta, de modo que pueda revisar los datos para comprobar si la especificación de la arquitectura corresponde con la del sistema que desea generar. Esta posibilidad de edición de los datos puede ser útil para depurar y realizar pruebas sobre el sistema en desarrollo.

El proceso de composición del *sistema extendido* (mediante la herramienta) se puede llevar a cabo de dos formas: de un modo automático y de un modo manual, según se describe en los apartados siguientes. Por último, el *sistema extendido* generado se puede guardar como un fichero de texto.

6.6.1. Método manual de composición, inserción de datos en la interfaz

Este método consiste en introducir los datos necesarios para la especificación del *sistema extendido* a través de la interfaz de usuario de la herramienta *AOSA Tool/LEDA*, según se explica detalladamente en la sección 6.7. La información necesaria se ha agrupado en tres bloques:

1. *Datos relativos al sistema inicial en LEDA*

Hay que introducir la descripción arquitectónica del *sistema inicial* objeto extensión. Además, hay que indicar los siguientes valores:

- El identificador de los componentes cliente y de los componentes servidor, entre los que se va a insertar uno o más aspectos.

- Ciertos parámetros (información contenida en la estructura de *Common Ítems* propuesta por *AOSA Model*).

2. *Datos asociados a los aspectos*

Hay que insertar la descripción arquitectónica de los componentes de aspecto. Asimismo, hay que insertar los valores de los parámetros para esos aspectos:

- El identificador del aspecto.
- Operación que intercepta.
- Sus condiciones de ejecución, de temporalidad y de sincronismo.

Esta información también está contenida en la estructura de los *Common Items*. La descripción arquitectónica, en *LEDA*, de un aspecto (del componente y su rol) que se insertará en el código del *sistema final* se muestra en Figura 6.9.

```
component Aspect {
interface
   aspectrole:AspectRole;
}
role AspectRole(aspectop, param){
spec is
aspectop?(respuesta).t.(res)respuesta!(res). AspectRole(aspectop, param);
}
```

Figura 6.9. Descripción arquitectónica de aspecto en *LEDA*.

3. *Datos de los componentes coordinadores*

Éstos se refieren a los elementos de coordinación que se encargan de coordinar la ejecución de los componentes de aspecto que van a incorporarse al sistema en extensión; los datos a introducir son:

- El código de los componentes coordinadores (en *LEDA*).

La descripción arquitectónica de estos elementos coordinadores que se insertará en el código del *sistema final* se muestra en los cuadros siguientes. El texto sombreado se refiere a código complementario que también hay que incluir en la extensión, pero que, sin embargo, no se puede considerar que forme parte de los elementos coordinadores. Hay que distinguir entre el código en *LEDA* de los *coordinadores regulares*, que atienden a un aspecto (Figura 6.10) y el código en *LEDA* de los coordinadores cuando un coordinador atiende a más de un aspecto sobre el mismo punto (*coordinador complejo*). En Figura 6.11 se muestra el código para un *coordinador complejo* que atiende a dos aspectos.

4. *Generación del sistema extendido*

Una vez introducidos todos los datos, se puede generar el *sistema extendido*, insertando adecuadamente en el código del *sistema inicial* los nuevos componentes y los roles correspondientes. También hay que definir los enlaces con el fin de establecer la interacción de los elementos insertados con los del *sistema inicial* que resultan interceptados. Finalmente se añade la instrucción que define la instancia del *sistema extendido*.

```
component Extendedsystem {
interface none;
composition
     coor : Coordinator;
       sc : Sc;
attachments
  coor.calltosc(scoperation,match)<>
sc.replytoco(scoperation,match);

component Coordinator {
interface
 act: Act;
 resend: Resend;
 intercept: Intercept;
 calltosc: Calltosc;
}
component Sc{
interface
  replytoco:Replytoco;
}

role
Resend(operation,param){
spec is
t.(sirviente)operation!(sirviente).sirv
iente?(val).Resend(operation,param);
}
```

```
role Intercept(operation,param){
spec is
operation?(answer).t.(val)answer!(val).
Intercept(operation,param);
}

role Act(op,respuesta){
spec is
t.(respuesta)op!(respuesta).respuesta?(
res).Act(op,respuesta);
}

role Calltosc(scoperation, match){
spec is
t.(r)scoperation!(r).r?(match).Calltosc
(scoperation, match);
}

role Replytoco(scoperation, match){
spec is
scoperation?(r).t.(match)r!(match).Repl
ytoco(scoperation, match);
}
```

Figura 6.10. Descripción arquitectónica de *coordinador regular* en *LEDA*.

```
component Extendedsystem {
composition
       coort : Coordinatortwice;
       sc : Sc;
attachments
   coor.calltosc(scoperation,match)<>
sc.replytoco(scoperation,match);

component Coordinatortwice {
interface
actuno:Actuno;
 actdos:Actdos
 resend:Resend;
 intercept:Intercept;
 calltosc:Calltosc;
}
component Sc{
interface
  replytoco:Replytoco;
}

role Resend(operation,param){
spec is
t.(sirviente)operation!(sirviente).sir
viente?(val).Resend(operation,param);
}
```

```
role Intercept(operation,param){
spec is
operation?(sirviente).t.(val)sirviente!(v
al).Intercept(operation,param);
}
role Calltosc(scoperation, match){
spec is
t.(r)scoperation!(r).r?(match).Calltosc(s
coperation, match);
}
role Actuno(opuno,resp){
spec is
t.(respuesta)opuno!(respuesta).resp?(valo
r).Actuno(opuno,resp);
}
role Actdos(opdos,resp){
spec is
t.(respuesta)opdos!(respuesta).resp?(valo
r).Actdos(opdos,resp);
}

role Replytoco(scoperation, match){
spec is
scoperation?(r).t.(match)r!(match).Replyt
oco(scoperation, match);
}
```

Figura 6.11. Descripción arquitectónica de *coordinador complejo* (para dos aspectos) en *LEDA*.

6.6.2. Método automático de composición, código *AspectLEDA*

En la generación del *sistema extendido* siguiendo el método automático el proceso se realiza en un sólo paso. Sin embargo, la herramienta permite que el arquitecto decida si quiere hacer una generación supervisada (paso a paso) o de modo totalmente automático.

Para crear un sistema de este modo, hay que editar o abrir un fichero con la especificación arquitectónica en *AspectLEDA* del sistema que se quiera generar en LEDA. Este código es analizado por el intérprete/analizador de *AspectLEDA* que extrae los datos necesarios hacer la generación. Con esta información, la aplicación correspondiente de la herramienta abre:

i) El fichero con la descripción arquitectónica del *sistema inicial* (expresado en *LEDA*, y que debe haber sido almacenado previamente). El nombre del *sistema inicial* forma parte de la descripción se sistema en *AspectLEDA*.
ii) El fichero con la descripción arquitectónica de aspecto, en *LEDA*.
iii) El fichero de los elementos coordinadores.

El código de todos ellos es analizado por el intérprete/analizador de *LEDA* para obtener la información necesaria que permita realizar la composición del *sistema extendido*. A continuación se genera el *sistema*.

La diferencia entre los modos de composición es que en el primero, el arquitecto introduce los datos a través de la interfaz y el *sistema extendido* se va componiendo paso a paso, a la vista del usuario. Si se sigue el método automático, se introduce la especificación del *sistema extendido* en *AspectLEDA* y el *sistema final* en *LEDA* se genera automáticamente, a partir de los datos proporcionados por dicha especificación.

Finalmente, el método automático de composición también se puede realizar paso a paso, para que sea supervisado por el ingeniero de software (siguiendo las instrucciones de la herramienta). La generación supervisada del *sistema extendido* permite al usuario modificar cualquier parte del código o cualquier valor de los datos. De este modo se puede cambiar la especificación del sistema en "tiempo de composición", por lo que se puede decir que, de algún modo, sirve para la depuración del propio sistema durante su construcción.

6.7. AOSA Tool/LEDA

En las secciones anteriores se ha descrito el lenguaje *AspectLEDA*, su estructura e instrucciones; igualmente, se ha descrito el proceso de generación del *sistema extendido*. En esta sección se describe la herramienta que se ha desarrollado para facilitar el proceso de diseño de sistemas orientados a aspectos, sus funciones y características.

La herramienta se puede ejecutar en cualquier máquina con una arquitectura x86, con el S.O. Windows XP, del orden de 10 Mb de especio en disco y un mínimo de 256 MB de memoria RAM.

Cuando se lanza la ejecución de la herramienta se presenta en pantalla la ventana principal de su interfaz de usuario que está compuesta por tres partes, diferenciadas en cuanto a funcionalidad. Esta ventana se muestra en Figura 6.12:

- En la sección marcada con 1 y el recuadro rojo de la figura se muestran cuatro campos para la búsqueda y apertura de ficheros:
  - Un campo de filtro para fijar el tipo de ficheros que se desea seleccionar.
  - El campo relativo a la unidad de almacenamiento que se está inspeccionando.
  - El tercer campo muestra el árbol de carpetas de la unidad que está seleccionada.
  - Una ventana muestra los ficheros contenidos en la carpeta seleccionada. Se listan los ficheros cuya extensión coincida con el filtro seleccionado.
  - En la parte inferior de esta sección hay un botón para *Abrir Fichero Seleccionado*. Su contenido se presenta en la ventana de texto de la pantalla.

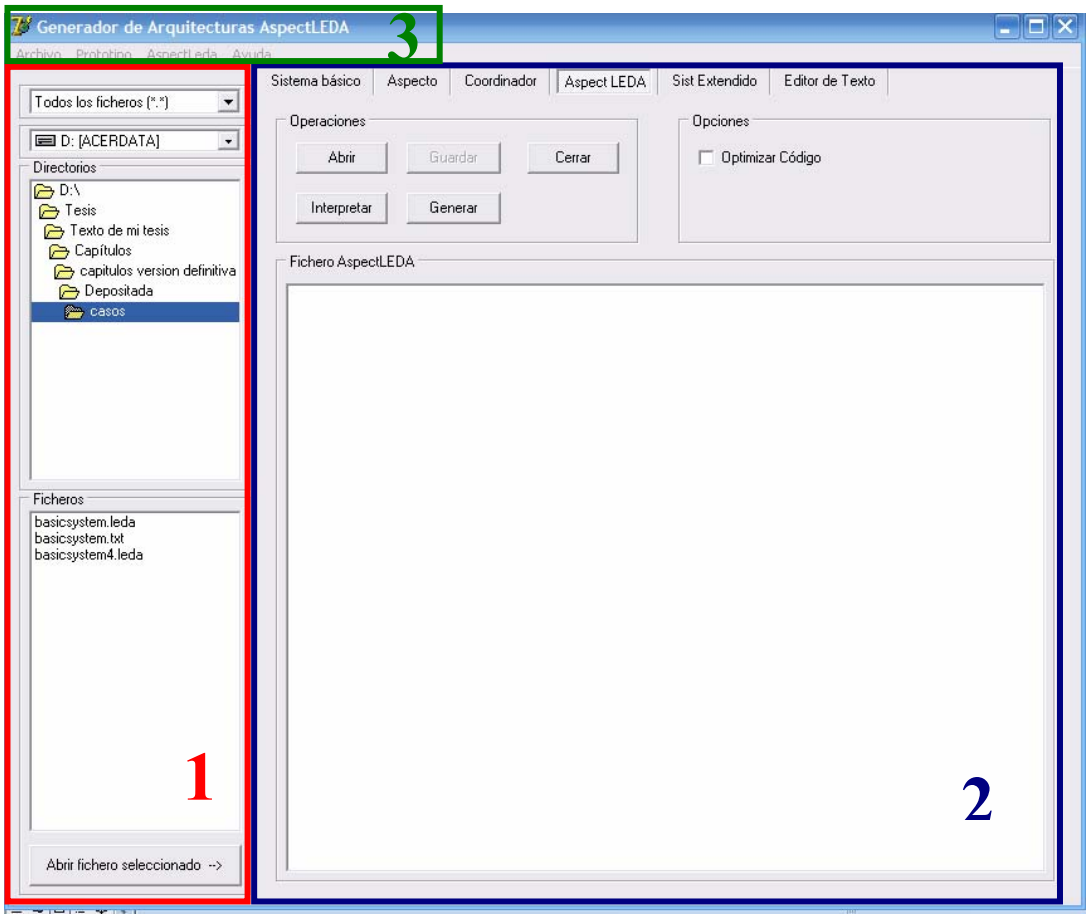


Figura 6.12. Secciones de la ventana principal de *AOSA Tool/LEDA*.

- La sección marcada con 2 y un recuadro de color azul de la figura, contiene los campos relativos a la composición del sistema, así como los valores de los parámetros asociados, ya sean de entrada o salida. Esta sección se ha estructurado en seis pestañas que contienen los datos correspondientes a cada parte del código a componer:
  - *Sistema inicial* (*Sistema básico* en la Figura).
  - Aspectos.
  - Elementos de coordinación (*Coordinador* en la figura).
  - Sistema en *AspectLEDA*.
  - El *sistema extendido* generado.
- La sección marcada con 3 contiene el menú principal de la aplicación y permite:
  - Abrir, guardar, imprimir archivos y salir de la herramienta.
  - La opción *Prototipo* (apartado 6.7.7) enlaza esta herramienta con el entorno *EVADeS* (Anexo 2) que permite ejecutar prototipos de sistemas en *LEDA*.
  - La opción *AspectLEDA* permite realizar las tareas necesarias para la generación del *sistema extendido,* visualizar los errores y los valores de los registros de datos.

En los próximos apartados se explica cada una de las pestañas en las que se divide la opción principal de este menú.

6.7.1. Pestaña *Sistema Básico*

La apariencia de la ventana asociada a esta pestaña es similar a la anterior, pero en este caso la información a introducir es la siguiente:

- Los datos relativos a los nombres y los parámetros de los componentes *cliente* y *servidor* entre los que se va a insertar un aspecto. Figura 6.13a representa la parte superior de la ventana y permite introducirlos.
- *Sistema inicial* en *LEDA* sobre el que se va a trabajar. Figura 6.13b muestra la ventana de edición de texto de esta ventana que es editable por lo que el código se puede crear o modificar. El contenido de esta ventana es lo que analiza el intérprete *LEDA*. Si el fichero con el *sistema inicial* ya existía, debe tener la extensión *.leda*, y se puede seleccionar desde la ventana de la izquierda.

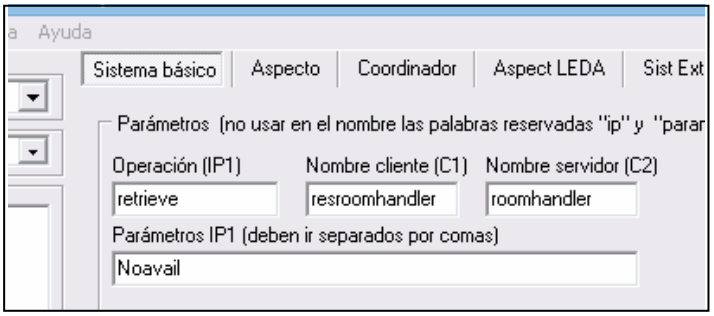


Figura 6.13a. Elementos de datos de la pestaña *Sistema Básico.*

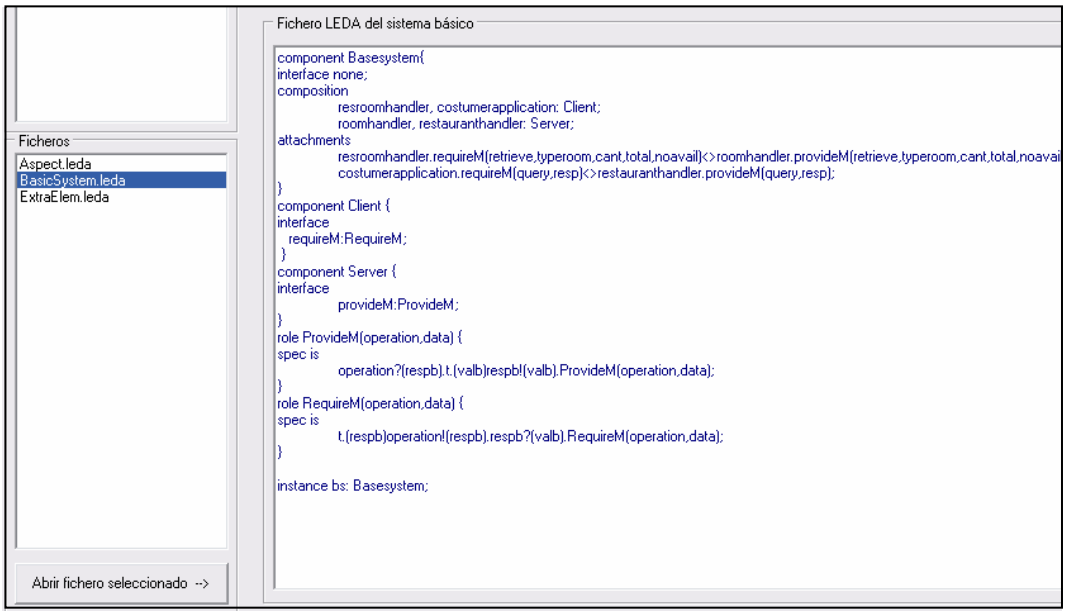


Figura 6.13b. Ventana de edición de la pestaña *Sistema Básico o Sistema Inicial.*

Un botón *Interpretar* llama al intérprete *LEDA* para analizar el código del *sistema inicial* y extraer sus datos para componer luego el *sistema final*.

6.7.2. Pestaña *Aspecto*

Esta pestaña permite introducir los datos relativos a los aspectos (Figura 6.14):

- El recuadro rojo (1) de la figura representa el conjunto de datos relativos a cada componente de aspecto: nombre, operación que ejecuta, nombre del componente cuya operación va a ser interceptada por él, parámetros del aspecto, condición de ejecución, tipo de evento asociado al aspecto, condición de temporalidad y, en su caso, prioridad de ejecución del aspecto.
- El recuadro azul (2) muestra una ventana de texto que contiene el código *LEDA* para la clase componente aspecto que se instancia con cada aspecto declarado en el sistema. La definición de esta clase es predeterminada.
- Un botón *Insertar* permite insertar la información de los aspectos en el código del sistema que se genera como salida. Esta acción forma parte de la funcionalidad de *AspectLEDA Translator* en Figura 6.8.

La información que se introduce a través de los campos representados en Figuras 6.13a y 6.14 se corresponde con la de los *Common Items.*

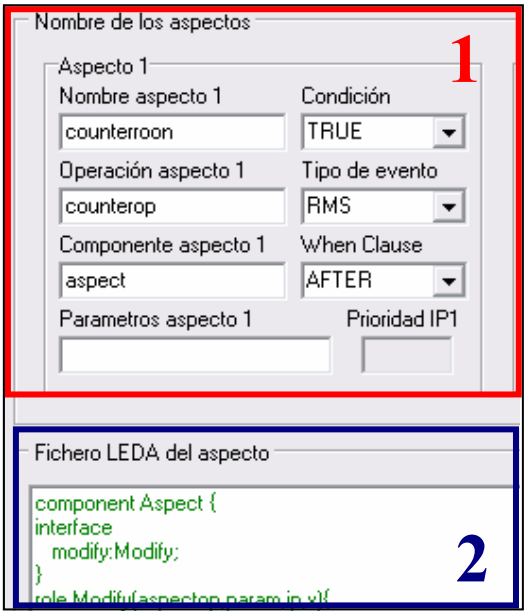


Figura 6.14. Elementos de interés de la pestaña *Aspecto*.

6.7.3. Pestaña *Coordinador*

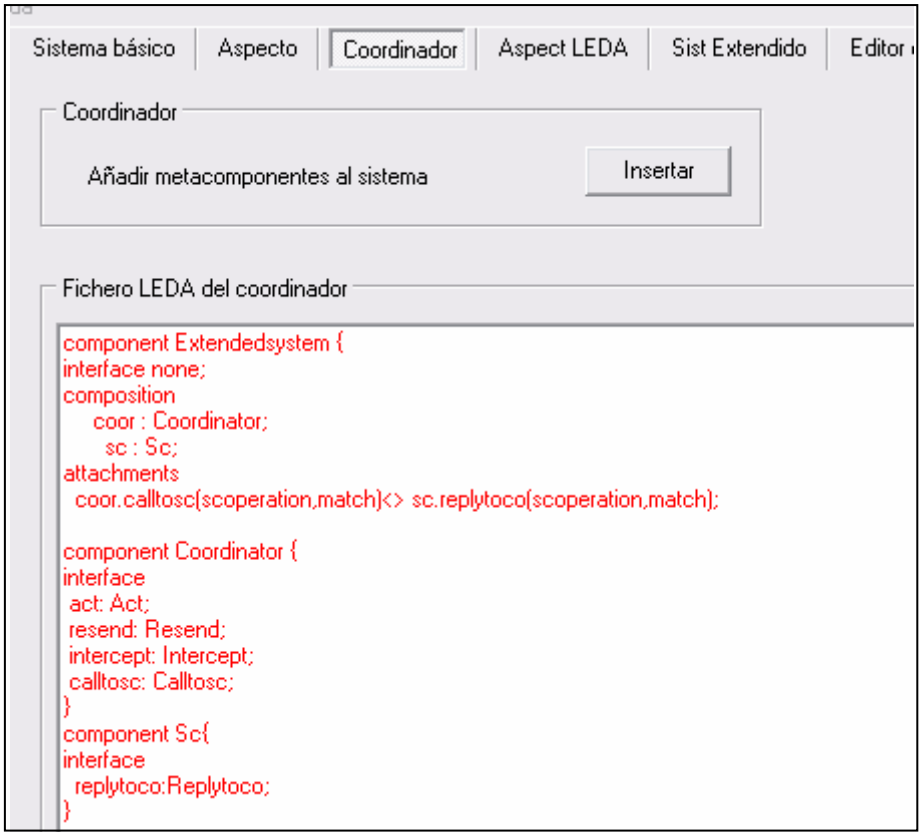
En esta pestaña se muestra una ventana de texto editable que contiene el código de los componentes *coordinadores* (Figura 6.15). Se muestra el código predeterminado para estos componentes y roles. Su definición se deduce de *AOSA Model*. El botón *Insertar* ejecuta la operación de insertar este código del *sistema final* que también forma parte de la funcionalidad de *AspectLEDA Translator* en Figura 6.8. El código genérico de los coordinadores se particulariza para cada aspecto asociado al realizarse la inserción.

6.7.4. Pestaña *AspectLEDA*

En la pestaña correspondiente a *AspectLEDA* (Figura 6.16) se muestra una ventana de texto con la especificación en *AspectLEDA* (1) y una sección con botones (2):

- En el recuadro azul (1) se muestra el código del sistema expresado en *AspectLEDA*. Éste es analizado por el intérprete de *AspectLEDA* que extrae los datos necesarios para llevar a cabo la composición del *sistema final*. Por tanto, si se introduce el *sistema extendido* en *AspectLEDA* no hay que introducir los parámetros que se solicitan en las ventanas anteriores. Por el contrario, si se introducen los parámetros anteriores, no es necesario introducir el código en *AspectLEDA* para obtener el *sistema extendido*, pues la herramienta lo genera al presionar *Insertar* en la ventana asociada. Esta ventana es editable.
- En el recuadro rojo (2) están los botones que permiten realizar las operaciones asociadas el uso de programas en *AspectLEDA*: *Abrir, Guardar, Cerrar,*

Interpretar y Generar. *Interpretar* se utiliza para realizar la composición del sistema por pasos; *Generar*, para realizar el proceso de una vez.



Fgura 6.15. Elementos de la pestaña *Coordinador*.

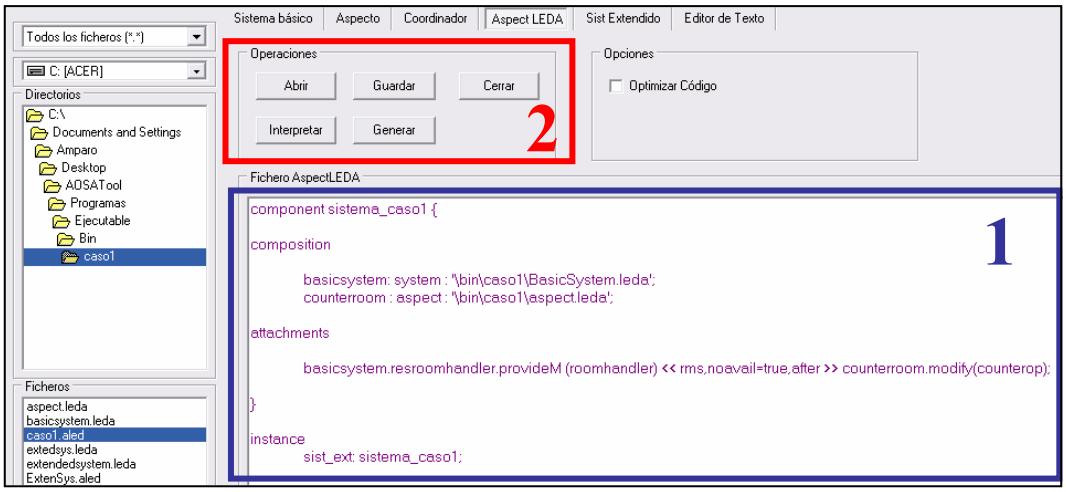


Figura 6.16. Pestaña *AspectLEDA*. Inserción de un aspecto.

6.7.5. Pestaña *Sistema Extendido*

Esta pestaña tiene tres partes (Figura 6.17):

- En el recuadro rojo (1) figuran los campos correspondientes al identificador a asignar a la clase del *sistema extendido* (*Identificador Clase*) y el identificador de la instancia de dicha clase (*Identificador Instancia*). Estos datos son necesarios pues siempre que se define un sistema en *LEDA* se ha de instanciar.
- En el recuadro azul (2) se muestran los botones correspondientes a las acciones asociadas a esta pestaña: *Insertar*, *Deshacer* y *Guardar*. La acción *Insertar* forma parte de la funcionalidad de *AspectLEDA Translator* en Figura 6.8.
- El recuadro verde (3) es una ventana de texto que muestra el código *LEDA* del *sistema extendido* generado por la herramienta. Esta ventana no es editable pues presenta el resultado de la composición del sistema.

El texto completo del *sistema extendido* para la inserción de un aspecto se muestra en Figura 6.18.

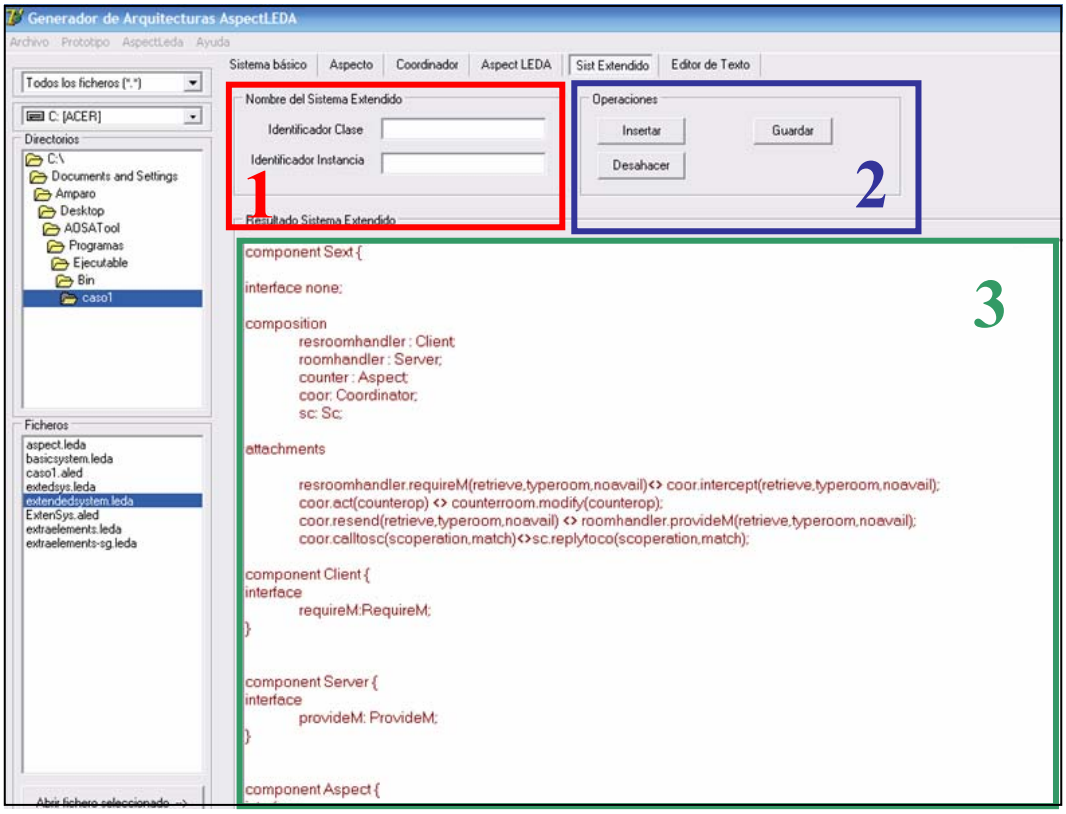


Figura 6.17. Pestaña *Sistema extendido.* Inserción de un aspecto.

```
component Sext {
interface none;
composition
resroomhandler:Client;
roomhandler:Server;
counterroom :Aspect;
coor : Coordinator;
sc:Sc;
attachments
resroomhandler.requireM(retrieve,typeroom
,noavail)<>
coor.intercept(retrieve,typeroom,noavail)
;
coor.act(counterop)<>counter.modify(count
erop);
coor.resend(retrieve,typeroom,noavail)<>r
oomhandler.provideM(retrieve,typeroom,noa
vail);
coor.calltosc(scoperation,match)<>sc.repl
ytoco(scoperation,match);
}
component Client{
interface
requireM:RequireM;
}
component Server{
interface
provideM:ProvideM;
}
component Aspect {
interface
modify:Modify;
}
component Sc{
interface
replytoco:Replytoco;
}
component Coordinator{
interface
act:Act;
resend:Resend;
intercept:Itercept;
calltosc: Calltosc;
}

role
ProvideM(retrieve,typeroom,noavail)
{
spec is
retrieve?(sirviente).t.(valor)sirviente!(
valor).ProvideM(retrieve,typeroom,noavail
);
}
role
Resend(retrieve,typeroom,noavail){
spec is
t.(sirviente)retrieve!(sirviente).sirvien
te?(val).Resend(retrieve,typeroom,noavail
);
}
role Modify(counterop){
spec is
counterop?(respuesta).t.(res)respuesta!(r
es).Modify(counterop);
}
role
RequireM(retrieve,typeroom,noavail)
{
spec is
t.(answer)retrieve!(answer).answer?(value
).RequireM(retrieve,typeroom,noavail);
}
role
Intercept(retrieve,typeroom,noavail){
spec is
retrieve?(answer).t.(val)answer!(val).Int
ercept(retrieve,typeroom,noavail);
}
role Act(counterop){
spec is
t.(respuesta)counterop!(respuesta).respue
sta?(res).Act(counterop);
}
role Calltosc(scoperation, match){
spec is
t.(r)scoperation!(r).r?(match).Calltosc(s
coperation, match);
}
role Replytoco(scoperation, match){
spec is
scoperation?(r).t.(match)r!(match).Replyt
oco(scoperation, match);
}

instance se: Sext;
```

Figura 6.18. *Sistema extendido* generado en *LEDA*. Inclusión de un aspecto.

6.7.6. Pestaña *Editor de texto*

Esta pestaña contiene un editor de texto para editar textos; en particular, permite crear y modificar especificaciones en *LEDA*. El editor permite abrir, modificar y guardar ficheros de texto, por lo que no hay que abandonar la herramienta para realizar estas tareas.

6.7.7. Obtención de un prototipo

La opción *Prototipo* del menú principal facilita al arquitecto de software las tareas necesarias para realizar una simulación de la ejecución del sistema generado. En este apartado se describe cómo se puede obtener.

Según se ha explicado a los largo de esta sección, *AOSA Tool/LEDA* facilita las tareas de traducción de una arquitectura expresada en *AspectLEDA* a otra equivalente en *LEDA*. Por otra parte, *LEDA* se puede traducir a Java (Anexo 1) si se ejecuta el intérprete de *LEDA* y obtener así la ejecución de un prototipo de un sistema. Después de la traducción *LEDA*-Java, los componentes y roles *LEDA* se convierten en clases Java.

Cuando se realiza la extensión de un sistema expresado en *AspectLEDA* a una arquitectura equivalente en *LEDA* los componentes del *sistema inicial* se habrán traducido a clases Java y los aspectos añadidos (que fueron definidos como componentes *LEDA*) ahora también son clases Java. Además, como ya se ha mencionado, los aspectos son independientes del contexto. La información que depende del contexto figura en los elementos cuya creación propone el modelo (*coordinadores*) y en las nuevas interacciones. Estos elementos que forman parte de la descripción del *sistema extendido* en *LEDA*, también han sido traducidos a Java, siendo ahora un conjunto de clases Java.

La especificación en LEDA del *sistema extendido* se puede expresar en calculo π. Figura 6.19 muestra la expresión en cálculo π del sistema representado en Figura 6.18.

Rol={ *ProvideM, Resend, Modify, RequireM, Intercept, Replytoco, Calltosc, Act}*
Comp={ *Client, Server, Aspect, Sc, Coordinator }*
Att={ *resroomhandler.requireM(retrieve,typeroom,noavail) <> coor.intercept(retrieve,typeroom, noavail); coort.act(counterop,resp) <> counterroom.modify(couterop,resp); coor.resend(retrieve, typeroom,noavail) <> roomhandler.provideM(retrieve,typeroom,noavail); coor.calltosc(scoperation,match)<> sc.replytoco (scoperation,match)*;
}
then
Architecture= Rol, Comp **instance** *se: Sext;*
and
[[AO*Architecture*]] = (Rol, Comp)
([[*ProvideM*]](*provideM*) | [[*RequireM*]](*requireM*) | [[*Resend*]](*resend*) | [[*Intercept*]] (*intercept*) | [[*Calltosc*]](*calltosc*) | [[*Replytoco*]](*replytoco*) | [[Act]](*act*) | [[*Modify*]](*modify*) | [[*Client*]](*resroomhandler*) | [[*Server*]](*roomhandler*) | [[*Aspect*]](*counterroom*) | [[*Sc*]](*sc*) | [[*Coordinator*]](*coor*) | (*se*) *sext* ! (*se*).**0**)

Figura 6.19. Arquitectura del ejemplo expresada en calculo π. Inclusión de un aspecto.

6.7.8. Caso de estudio. CASO GENERAL

En el apartado anterior se ha mostrado cómo usar la herramienta siguiendo el caso de estudio, en el caso más sencillo. A continuación se describe el código para el caso de estudio incluyendo varios aspectos.

a) Sistema inicial en LEDA

Considérese el *sistema inicial* descrito en el capítulo 5:

```
Component Basesystem {
interface none;
composition
  resroomhandler,customerapp: Client;
  roomhandler,restauranthandler: Server;
attachments
resroomhHandler.requireM(retrieve,typeroom,noavail)<>ro
omhandler.provideM(retrieve,typeroom, noavail);
customerapp.requireM(query,resp)<>restauranthandler.pro
videM(query,resp);
}
component Client {
interface
  requireM:RequireM;
}
component Server {
interface
  provideM:ProvideM;
}

Role ProvideM(operation,param){
spec is
operation?(answer).(value)answer!(value).ProvideM(opera
tion,param);
}
Role RequireM(operation,param){
spec is
(answer)operation!(answer).answer?(value).RequireM(oper
ation,param);
}

instance basesys:Basesystem;
```

b) Sistema extendido en AspectLEDA

La arquitectura en *AspectLEDA* se muestra a continuación. En ella se puede observar la inclusión de tres aspectos que se aplican sobre los componentes *Roomhandler* y *Restauranthandler*, dando lugar a la definición de 5 instancias de la clase aspect Además se establece prioridad entre los aspectos *counterroom* y *findroom:*

```
Component Extnddsyst {
composition
 basesystem: System;'ruta_fich_S_base';
 counterroom: Aspect; 'ruta_fich_Aspect1';

 countertable: Aspect; 'ruta_fich_Aspect1';
 findroom:Aspect; 'ruta_fich_Aspect2';
 waitinglistroom:Aspect; 'ruta_fich_Aspect3';
 waitinglisttable:Aspect; 'ruta_fich_Aspect3';
  attachements
basesystem.Roomhandler.Retrieveop(resroomhandler)
   <<RMS,NoAvail=True,after>>counterroom.Countertop();
basesystem.Restauranthandler.Retrieveop(customerapp)
   <<RMS,NoAvail=True,after>>countertable.Countertop();
basesystem.Roomhandler.Retrieveop(resroomhandler)
   <<RMS,NoAvail=True,after>>findromm.Findroomop();
basesystem.Roomhandler.Request(resroomhandler)
   <<RMS,Full=True,after>>waitinglistroom.Waitinlistop();
basesystem.Restauranthandler.Query(customerapp)
  <<RMS,Full=True,after>>waitinglisttable.Waitinglistop();
}
priority counterroom>findroom;

instance se: Sext;
```

c) Sistema extendido en LEDA

El *sistema extendido* en *LEDA* que se genera está formado por el *sistema inicial* y los aspectos incorporados, junto con los elementos del meta nivel. Figura 6.20 lo muestra. Por otra parte, la arquitectura del *sistema extendido* en LEDA se puede expresar en calculo π según se muestra en el cuadro siguiente de Figura 6.21.

Tras la ejecución de la arquitectura (del prototipo generado) se puede evaluar si el comportamiento del sistema es el esperado o no. De esta manera, se pueden detectar errores que de otro modo se trasladarían a fases posteriores. Especialmente interesante es detectar errores en la descripción de la arquitectura cuando varios aspectos se aplican sobre el mismo punto de inserción, para determinar si la prioridad establecida entre ellos es la correcta. En el caso del ejemplo que se describe en los párrafos precedentes, los aspectos *Counterroom* y *Findroom* se han aplicado en un orden de prioridad incorrecta, pues se asignó mayor prioridad al primero, lo que conlleva que se cuenten solicitudes no satisfechas antes de buscar en otros hoteles de la cadena (*Findroom*). Sin embargo, el arquitecto puede no darse cuenta de este detalle al especificar los requisitos y establecer las prioridades de un modo inadecuado.

```
component Sext {

interface none;

composition
resroomhandler,customerapp: Client;
roomhandler,restauranthandler:Serve
r;
       counterroom : Aspect;
       countertable: Aspect
       findroom :Aspect;
       waitinglistroom: Aspect;
       waitinglisttable: Aspect;
       coort: Coordinatortwice;
       coor2: Coordinator;
       coor3: Coordinator;
       coor4: Coordinator;
       sc: Sc;


attachments
//---Para el aspecto counterroom y
findroom sobre el resroomhandler y
el coordinador coor1-----

resroomhandler.requireM(retrieve,ty
peroom,noavail)<>coort.intercept(re
trieve,typeroom,noavail);
coort.actuno(counterop,resp)<>count
erroom.modify(counterop,resp);
coort.actdos(locate,resp)<>findroom
.modify(locate,resp);
coort.resend(retrieve,typeroom,noav
ail)<>roomhandler.provideM(retrieve
,typeroom,noavail);
coort.calltosc(scoperation,match)<>
sc.replytoco(scoperation,match);
//---Para el aspecto waitinglisttab
le sobre el componente restaurantha
ndler y el coordinador coor2-----

customerapp.requireM(query,resp)<>c
oor2.intercept(query,resp);
coor2.act(query,resp)<>waitinglistt
able.modify(query,resp);


coor2.resend(query,resp)<>restauran
thandler.provideM(query,resp);
coor2.calltosc(scoperation,match)<>
sc.replytoco(scoperation,match);

//---Para el aspecto waitinglistr
oom sobre el componente roomhandler
y coor3-----

customerapp.requireM(query,resp)<>c
oor3.intercept(query,resp);
coor3.act(query,resp)<>waitinglistr
oom.modify(query,resp);
coor3.resend(query,resp)<>roomhandl
er.provideM(query,resp);
coor3.calltosc(scoperation,match)<>
sc.replytoco(scoperation,match);


//--Para el aspecto countertable
sobre el componente restaurant
handler y coor4—

customerapp.requireM(retrieve,resp)
<>coor4.intercept(retrieve,resp);
coor4.act(retrieve,resp)<>counterta
ble.modify(retrieve,resp);
coor4.resend(retrieve,resp)<>restau
ranthandler.provideM(retrieve,resp)
;
coor4.calltosc(scoperation,match)<>
sc.replytoco(scoperation,match);
//---------------

}


component Client{
interface
   requireM:RequireM;
 }


component Server {
interface
   provideM: ProvideM;
 }


component Aspect {
interface
   modify:Modify;
}
```

Figura 6.20. *Sistema extendido* generado en *LEDA*. Inclusión de varios aspectos.

```
//---------------
component Coordinatortwice {
interface
 actuno:Actuno;
 actdos:Actdos;
 resend:Resend;
 intercept: Intercept;
 calltosc: Calltosc;
}

//-----------
component Coordinator {
interface
 act:Act;
 resend:Resend;
 intercept: Intercept;
 calltosc: Calltosc;
}
//------------------

component Sc{
interface
  replytoco:Replytoco;
}



role ProvideM(operation,param){
spec is
operation?(sirvient).t.(val)sirvie
nt!(val).Provide(operation,param);
}

role RequireM(operation,param){
spec is
t.(resp)operation!(resp).resp?(val
).Require(operation,param);
}

role Resend(operation,param){
spec is
t.(sirvient)operation!(sirvient).s
irvient?(val).Resend(operation,par
am);
}
```

```
role Intercept(operation,param) {
spec is
operation?(resp).t.(val)resp!(val)
.Intercept(operation,param);
}

role Calltosc(scoperation, match){
spec is
t.(r)scoperation!(r).r?(match).Cal
ltosc (scoperation, match);
}

role    Replytoco    (scoperation,
match){
spec is
scoperation?(r).t.(match)r!(match)
.Replyt oco(scoperation, match);
}

//-------
role Actuno(opuno,resp){
spec is
t.(respuesta)opuno!(respuesta).res
p? (valor).Actuno(opuno,resp);
}

role Actdos(opdos,resp){
spec is
t.(respuesta)opdos!(respuesta).res
p?(valor).Actdos(opdos,resp);
}
//-------

role Act(op,resp){
spec is
t.(respuesta)op!(respuesta).resp?(
valor).Act(op,resp);
}
//--------

role Modify(modifyop,resp){
spec is
modifyop?(sirviente).t.(re)resp!(r
e).Modify (modifyop,resp);
}

instance se: Sext;
```

Figura 6.20 Cont. *Sistema extendido* generado en *LEDA*. Inclusión de varios aspectos.

Rol={*ProvideM, RequireM, Resend, Intercept, Calltosc, Replytoco, Actuno, Actdos, Act, Modify, Modify}*

Comp={ *Client, Server, Aspect, Sc, Coordinatortwice, Coordinator* }

Att={*resroomhandler.requireM(retrieve,typeroom,noavail) <> coort.intercept(retrieve,typeroom, noavail); coort.actuno(counterop,resp) <> counterroom.modify(counterop,resp); coort.actdos (locate,resp) <> findroom.modify (modifyop,resp); coort.resend (retreve,typeroom,noavail) <> roomhandler.provideM(retrieve,typeroom,noavail); coort.calltosc(scoperation,match) <> sc.re plytoco(scoperation,match); customerapp.requireM(query,resp) <> coor2.intercept(query, resp); coor2.act(query,resp) <> waitinglisttable.modify(query,resp); coor2.resend(query,resp) <> restauranthandler.provideM(query,resp); coor2.calltosc(scoperation,match) <> sc.re plytoco(scoperation,match); customerapp.requireM(query,resp) <> coor3.intercept (query,resp); coor3.act(query,resp) <> waitinglistroom.modify (query,resp); coor3.resend (query,resp) <> roomhandler.provideM(query,resp); coor3.calltosc (scoperation,match) <> sc.replytoco(scoperation,match); customerapp.requireM(retrieve,resp) <> coor4.intercept(re trieve,resp); coor4.act(retrieve,resp) <> countertable.modify(retrieve,resp); coor4.resend(re trieve,resp) <> restauranthandler.provideM(retrieve,resp); coor4.calltosc(sc operation,match) <> sc.replytoco(scoperation,match)*

}

then

Architecture= Rol, Comp **instance** *se: Sext;*

and

[[AO*Architecture*]] = (Rol, Comp)

([[*ProvideM*]](*provideM*) | [[*RequireM*]](*requireM*) | [[*Resend*]](*resend*) | [[*Intercept*]](*intercept*) | [[*Calltosc*]](*calltosc*) | [[*Replytoco*]](*replytoco*) | [[*Actuno*]](*actuno*) | [[*Actdos*]](*actdos*) | [[Act]](*act*) |[[*Modify*]](*modify*) | [[*Client*]](*resroomhandler, customerapp*) | [[*Server*]](*roomhandler, restauranthandler*) | [[*Aspect*]](*counterroom, countertable, findroom, waitinglistroom, waitinglisttable*) | [[*Sc*]](*sc*) | [[*Coordinatortwice*]](*coort*) | [[*Coordinator*]](*coor1,coor2, coor3, coor4*)

| (*se*) *Sext* ! (*se*).**0**)

Figura 6.21. Arquitectura del ejemplo expresada en cálculo π. Inclusión de varios aspectos.

6.8. Similitudes y diferencias con otros LDA-OA

En esta sección se establece una comparación entre *AspectLEDA* y los LDA-OA que se estudiaron en el Capítulo 3 de esta memoria.

DAOP-ADL

- En *DAOP-ADL* los aspectos se describen como componentes, igual que en *AspectLEDA*. Sin embargo, la descripción de sistemas orientados a aspectos en *DAOP-ADL* requiere la descripción previa de la arquitectura en el modelo *CAM*. Por el contrario, en *AspectLEDA* no se requiere que los componentes estén descritos en ningún modelo previo, sólo es necesario que su arquitectura se exprese en *LEDA*.
- Una arquitectura descrita en *DAOP-ADL* debe ser interpretada por la plataforma

DAOP siendo independiente de cualquier lenguaje de programación. La propuesta de *AspectLEDA* es independiente de la plataforma.

- Está basado en XML, mientras que *AspectLEDA* se apoya en un LDA basado en el cálculo π, lo que le proporciona una fuerte base formal. Además, permite generar código Java lo que facilita la ejecución de un prototipo del sistema.
- En *DAOP* la coordinación se realiza definiendo un aspecto dedicado a esta labor: a nivel de lenguaje, se define un componente específico: *aspecto coordinador*. En el modelo que se propone en esta memoria se define un meta nivel que facilita la coordinación entre el *sistema base* (componentes del sistema) y el o los aspectos añadidos mediante la figura de un componente *coordinador* y se aplica un modelo de coordinación; así, en *AOSA Model*, el problema de diseñar sistemas orientados a aspectos se trata como un problema de coordinación.
- No se establece ningún modo de especificar la *quantification* mientras que en *AspectLEDA* sí.
- Ambos permiten la composición dinámica de aspectos,
- *DAOP-ADL* permite la utilización de herramientas de validación que aseguren la corrección, desde el punto de vista arquitectónico, de la composición de componentes y la evaluación de aspectos. Al igual que *AspectLEDA* a partir de las capacidades del LDA que lo sustenta.

PrismaADL

- Los componentes en *Prisma ADL* tienen que estar definidos en el modelo de componentes que facilita el modelo *Prisma*, mientras que en *AspectLEDA* no se requiere que los componentes estén descritos en ningún modelo determinado.
- *Prisma ADL* se basa en un modelo simétrico y la estructura del lenguaje es acorde a él. Por el contrario, *AspectLEDA* se ha definido asociado *AOSA Model* que sigue una aproximación asimétrica, por lo que las características de componentes y aspectos son diferentes en ambos lenguajes.
- La especificación formal de *Prisma ADL* facilita la generación de código (C#) que se puede ejecutar sobre el *middleware* de *Prisma* (.NET) que proporciona servicio de distribución y facilita la evolución. Sin embargo, *AspectLEDA* es independiente de la plataforma de implementación.
- No se establece ningún modo de especificar la *quantification*, mientras que en *AspectLEDA* sí.
- Ambos lenguajes permiten la reconfiguración dinámica del sistema en tiempo de ejecución.
- *Prisma ADL*, como *AspectLEDA,* dispone de herramientas que facilitan el diseño de los sistemas orientados a aspectos. Igualmente ambos lenguajes permiten el análisis y validación de la arquitectura que se diseña bajo ellos. Sin embargo, sólo Aspect*LEDA* permite generar prototipos ejecutables en Java.

AO-Rapide

- *AO-Rapide* se basa, como *AspectLEDA,* en un LDA existente que permite el análisis de la arquitectura y la simulación de la ejecución de un sistema en fase

de diseño. Además, ambos lenguajes (*Rapide* y *LEDA*) se han extendido para adaptarlos al desarrollo de sistemas orientados a aspectos. El carácter de las ampliaciones es permitir la ejecución coordinada de los aspectos (considerados como componentes arquitectónicos) y los componentes funcionales sobre los que se aplican.

- En *AO-Rapide* los aspectos se describen como componentes del lenguaje y son reutilizables, al igual que el lenguaje que se propone.
- La generación de código en *AO-Rapide* es una característica heredada de *Rapide*, que permite la representación de modelos ejecutables en tiempo de diseño. Igualmente permite comprobar el comportamiento del sistema en fase de diseño. Similar a *AspectLEDA*.
- En ambos casos se utilizan modelos de coordinación para gestionar la composición de los aspectos, si bien los modelos de coordinación son distintos.
- La *quantification*, como en *AspectLEDA*, depende del modelo de coordinación (en este caso *Reo*).
- En *AO-Rapide*, cada componente de aspecto tiene asociado un componente coordinador, definido externamente al aspecto, lo que facilita la reutilización de éste. En AO-Rapide se definen unos componentes (*wrappers*) que facilitan la composición final.
- Finalmente, hasta el momento, *AO-Rapide* no dispone de herramientas propias, que facilite el desarrollo de sistemas orientados a aspectos. Sólo dispone de las utilidades que proporciona *Rapide* (algunas de las cuales permiten el análisis y validación de la arquitectura). Por el contrario, *AspectLEDA* sí dispone de tales herramientas.

AspectualAcme

- Como el lenguaje anterior, *AspectualAcme* es una extensión de un LDA existente para realizar la descripción arquitectónica de sistemas orientados a aspectos.
- Como en *AspectLEDA*, la funcionalidad de los componentes del *sistema inicial* se mantiene separada de la de los aspectos, que también son considerados componentes del lenguaje.
- En *AspectualAcme*, como en *AspectLEDA*, se mantiene el principio de inconsciencia de modo que los componentes base no resultan modificados al incorporar aspectos al sistema. Ambos soportan la *quantification*.
- Sin embargo, ambos lenguajes difieren en la forma de integrar los aspectos en el sistema: en *AspectualAcme* se definen *conectores aspectuales* para llevar a cabo la composición y coordinación de componentes y aspectos. Por el contrario, en *AspectLEDA* se definen unos elementos *coordinadores* que son componentes del *sistema extendido* y que realizan las labores de coordinación y composición desde un meta nivel.
- En ambos lenguajes la arquitectura se define independiente de la plataforma y del modelo de componentes.

FAC ADL

- Es una aproximación similar a *AspectLEDA*, en la que también se utiliza el concepto de componente de aspecto. Sin embargo, para realizar la composición de éstos con los componentes del sistema no es necesario modificar los enlaces existentes en la arquitectura inicial como se hace en *AspectLEDA*. En *FAC* se añaden una *interfaz de weaving* a los componentes a los que se asocian a los aspectos.
- Ambos lenguajes permiten la composición dinámica de aspectos y la especificación de la *quantification*.
- Al contrario que *AspectLEDA* que se define independiente del modelo de componentes, el modelo de componentes de *FAC* extiende el modelo de *Fractal* y las arquitecturas están orientadas al desarrollo de sistemas basados en componentes.
- El formalismo que proporciona *FAC ADL* es el de XML, mientras que *AspectLEDA* hereda las características formales de *LEDA* (cálculo π).

6.9. Conclusiones

En este capítulo se ha presentado *AspectLEDA,* un LDA-OA que permite generar sistemas AO en un lenguaje regular, *LEDA*. *AspectLEDA* se define a partir de este LDA, no siendo necesario incluir nuevas abstracciones para representar aspectos. Se ha descrito el lenguaje, sus características y una herramienta que facilita al arquitecto de software las tareas relativas a la generación del *sistema extendido*: la especificación arquitectónica de un sistema en *AspectLEDA* se traduce a *LEDA* mediante *AOSA Tool/LEDA*, que también comprueba la corrección del código. A continuación, la descripción arquitectónica del *sistema extendido* en *LEDA* se puede traducir a Java, pudiéndose obtener y ejecutar asimismo un prototipo. También se puede chequear la corrección de la arquitectura.

Por otra parte, el lenguaje cumple las premisas propuestas en [Nav+02] pues es posible especificar componentes funcionales y de aspectos, así como realizar la especificación de los puntos de enlace y su interacción con los aspectos. Esta interacción internamente es gestionada por unos componentes *coordinadores* (que siguen un modelo de coordinación, según se propone en *AOSA Model*) que gestionan la ejecución del sistema orientado a aspectos cuando se genera una arquitectura equivalente en el LDA convencional. De este modo, se materializa que el *problema de la separación de aspectos se puede resolver tratándolo como uno de coordinación*.

Como complemento a lo expuesto es este capítulo sobre *AOSA Tool/LEDA*, en [MeNa08] se puede encontrar una descripción detallada de los manuales de usuario y del programador.

Por otra parte, se ha desarrollado una versión en inglés de la herramienta.

6.10. Apéndice A: gramática *AspectLEDA*

<arquitectura>$_1$ ::= <declaracion_componente>$_3$ <instantacion_sistema>$_2$
<instantacion_sistema>$_2$ ::= ***instance*** *<identificador>$_{30}$ ':' <identificador>$_{30}$ ';'*
<declaracion_componente>$_3$ ::= ***component*** *<identificador>$_{30}$ '{'*
<cuerpo_declaracion_clase_componente>$_4$'}'
<cuerpo_declaracion_clase_componente>$_4$ ::= ***composition*** *<declaracion_composicion>$_5$*
<declaracion_enlaces>$_{10}$
<declaracion_composicion>$_5$ ::= <declaracion_sistema>$_6$ <lista_declaracion_aspecto>$_7$
<declaracion_extraelements>$_9$
<declaracion_sistema>$_6$ ::= <identificador>$_{30}$ ':' ***system*** *';'*
| <identificador>$_{30}$ ':' ***system*** *<declaracion_ruta>$_{26}$ ';'*
<lista_declaracion_aspecto>$_7$ ::= <declaracion_aspecto>$_8$
| <lista_declaracion_aspecto>$_7$ <declaracion_aspecto>$_8$
<declaracion_aspecto>$_8$ ::= <identificador>$_{30}$ ':' ***aspect*** *';'*
| <identificador>$_{30}$ ':' ***aspect*** *<declaracion_ruta>$_{26}$ ';'*
<declaracion_extraelements>$_9$ ::= **λ**
| <identificador>$_{30}$ ':' ***component*** *';'*
| <identificador>$_{30}$ ':' ***component*** *<declaracion_ruta>$_{26}$ ';'*
<declaracion_enlaces>$_{10}$ ::= ***attachments***
*|**attachments**<lista_declaracion_enlaces>$_{14}$<declaracion_prioridades>$_{11}$*
<declaracion_prioridades>$_{11}$ ::= **λ**
| ***priority*** *<lista_declaracion_prioridad>$_{12}$*
<lista_declaracion_prioridad>$_{12}$ ::= <declaracion_prioridad>$_{13}$
| <lista_declaracion_prioridad>$_{12}$ <declaracion_prioridad>$_{13}$
<declaracion_prioridad>$_{13}$ ::= <identificador>$_{30}$ '>' <identificador>$_{30}$ ';'
| <identificador>$_{30}$ '<' <identificador>$_{30}$ ';'
<lista_declaracion_enlaces>$_{14}$ ::= <declaracion_enlace>$_{15}$ ';'
| <lista_declaracion_enlaces>$_{14}$ <declaracion_enlace>$_{15}$ ';'
<declaracion_enlace>$_{15}$ ::= <identificador_operacion_origen>$_{16}$ '<' '<'
<declaracion_parametros_aspectos>$_{19}$ '>' '>' <identificador_operacion_destino>$_{17}$
<identificador_operacion_origen>$_{16}$::= <identificador_variable>$_{25}$'.' <identificador_variable>$_{25}$
'.' <identificador_variable>$_{25}$ '(' <identificador>$_{30}$ ')'
<identificador_operacion_destino>$_{17}$ ::= <identificador_variable>$_{25}$ '.'
<identificador_variable>$_{25}$ '(' <identificador>$_{30}$ ')'
<declaracion_parametros>$_{18}$ ::= **λ**
| <lista_parametros>$_{20}$
<declaracion_parametros_aspectos>$_{19}$ ::= **λ**
| <lista_parametros_aspectos>$_{21}$
<lista_parametros>$_{20}$ ::= <identificador>$_{30}$
| <lista_parametros>$_{20}$ ',' <identificador>$_{30}$
<lista_parametros_aspectos>$_{21}$ ::= <parametro_tipo_evento>$_{22}$ ',' <parametro_condicion>$_{23}$
',' <parametro_condicion_when>$_{24}$
<parametro_tipo_evento>$_{22}$ ::= ***rma***
| ***rms***
<parametro_condicion>$_{23}$ ::= <identificador>$_{30}$ '=' ***true***
| <identificador>$_{30}$ '=' ***false***
<parametro_condicion_when>$_{24}$ ::= ***before***
| ***after***
| ***around***

<identificador_variable>$_{25}$ *::= <identificador>*$_{30}$

<declaracion_ruta>$_{26}$ *::= ':' <cadena>*$_{33}$
| ':' <letra_unidad>$_{27}$ *<lista_directorios>*$_{28}$
<letra_unidad>$_{27}$ *::= <letra>*$_{36}$ *':'*
<lista_directorios>$_{28}$ *::=* **λ**
| <lista_directorios>$_{28}$ *<directorio>*$_{29}$
<directorio>$_{29}$ *::= <barra>*$_{37}$ *<identificador>*$_{30}$
<identificador>$_{30}$ *::= <letra>*$_{36}$ *<lista_caracteres_alfanumericos>*$_{31}$
<lista_caracteres_alfanumericos>$_{31}$ *::=* **λ**
| <lista_caracteres_alfanumericos>$_{31}$ *<caracter_alfanumerico>*$_{32}$
<caracter_alfanumerico>$_{32}$ *::= <caracter>*$_{35}$ *| <digito>*$_{39}$
<cadena>$_{33}$ *::= <lista_simbolos>*$_{34}$
<lista_simbolos>$_{34}$ *::=* ***** *| <lista_simbolos>* $_{34}$ *****
<caracter>$_{35}$ *::= <letra>*$_{36}$ *| '_'*
<letra>$_{36}$ *::=* ***a | .. | z | A | .. | Z***
<barra>$_{37}$ *::= '\'*
<entero>$_{38}$ *::= <digito>*$_{39}$ *| <entero>*$_{38}$ *<digito>*$_{39}$
<digito>$_{39}$ *::=* ***0 | .. | 9***

6.11. Apéndice B: mensajes de error

El objetivo de este apéndice es presentar la relación de errores detectados por el intérprete de *AspectLEDA* y los detectados por el intérprete *LEDA*.

B1.- Mensajes de error del intérprete de *AspectLEDA*

El formato de salida de los mensajes de error del intérprete *AspectLEDA* y su correspondiente código de error es el siguiente:

*###**Código_Error**###Línea* ***Número_Línea*** *:* ***mensaje_error token valor_token***

(en negrita los valores variables de las sentencias)

###Error0###Linea ***numlinea****: error de sintaxis en el codigo, el analizador no termino de analizar el código del programa AspectLEDA.*

###Warning1###Linea ***numlinea****: Identificador->****parametro_error*** *para sistema base no declarado.*

###Warning2###Linea ***numlinea****: no se pudo insertar el enlace->****parametro_error****, todos los enlaces insertados o el aspecto usado en el enlace no ha sido declarado en el sistema.*

###Warning3###Linea ***numlinea****: Enlace no corresponde con ningún aspecto declarado.*

###Warning4###Linea ***numlinea****: Ningun aspecto declarado.*

###Warning5###Linea ***numlinea****: Parametro IP (COMPONENT_NAME) no declarado.*

###Warning6###Linea ***numlinea****: Parametro IP (INSERTION_POINT) no declarado.*

###Warning7###Linea ***numlinea****: Parametro IP (CONDITION) no declarado.*

###Warning8###Linea ***numlinea****: Parametro IP (ASPECT_NAME) no declarado.*

###Warning9###Linea ***numlinea****: Parametro IP (EVENT_TYPE) no declarado.*

###Warning10###Linea ***numlinea****: Parametro IP (WHEN_CLAUSE) no declarado.*

###Warning11###Linea ***numlinea****: Identificador->****parametro_error*** *de aspecto no declarado.*

###Warning12###Linea ***numlinea****: Todos los aspectos insertados, imposible insertar aspecto ->* ***parametro_error.***

###Error1###Linea ***numlinea****: Identificador duplicado: identificador =* ***parametro_error.***

###Error2###Linea ***numlinea****: Identificador de sistema extendido no declarado.*

###Error3###Linea ***numlinea****: Identificador de sistema base no declarado.*

###Error13###Linea ***numlinea****: Identificador->****parametro_error*** *de aspecto en operación de declaracion de prioridad no existe en el sistema.*

###Error14###Linea ***numlinea****: Identificadores->****parametro_error*** *idénticos en sentencia de declaracion de prioridad.*

###Error15###Linea ***numlinea****: identificador de clase usado en instanciación de clase sistema, no ha sido declarado, clase inexistente -->* ***parametro_error.***

###Error 16###Linea ***numlinea****: identificador de instancia de sistema ya declarado o duplicado ->* ***parametro_error.***

###Error17###Linea ***numlinea****: identificador de clase sistema usado en declaración de enlace, no ha sido declarado -->* ***parametro_error.***

###Error Desconocido (Default)###Linea ***numlinea****: Error desconocido: origen =* ***parametro_error.***

B2.- Mensajes de error del intérprete LEDA

El formato de salida de los mensajes de error del intérprete *LEDA* y su correspondiente código de error es el siguiente:

###Código_Error*###Línea* ***Número_Línea*** *:* ***mensaje_error token valor_token***

(en negrita los valores variables de las sentencias)

###Error0###Linea ***numlinea****: error de sintaxis en el código, el analizador no terminó de analizar el código del programa LEDA.*

###Error1###Linea ***numlinea****: identificador de componente id=****parametro_error*** *duplicado.*

###Error2###Linea ***numlinea****: componente utilizado en la declaración de enlace, id=****parametro_error*** *no declarado.*

###Error3###Linea ***numlinea****: id de clase componente en implementación de componente, id=****parametro_error*** *duplicado.*

###Error4###Linea ***numlinea****: identificador de rol id=****parametro_error*** *no declarado.*

###Error5###Linea ***numlinea****: clase de instancia de sistema id=****parametro_error*** *no declarada.*

###Error6###Linea ***numlinea****: identificador de instancia de sistema id=****parametro_error*** *duplicado.*

###Error7###Linea ***numlinea****: identificador de operación id=****parametro_error*** *duplicado en otra operación.*

###Error8###Linea ***numlinea****: identificador de rol id=****parametro_error*** *duplicado en otro rol.*

###Error9###Linea ***numlinea****: declaración de enlace id=****parametro_error*** *no válida, enlace duplicado.*

###Warning1###Linea ***numlinea****: declaración de operación id=****parametro_error*** *ignorada, componente no válido (duplicado).*

###Default###error desconocido ***numlinea parametro_error.***

CAPÍTULO 7
Conclusiones y trabajos futuros

En este capítulo se resumen y se analizan las principales contribuciones del trabajo presentado en esta tesis doctoral. Igualmente se proponen los trabajos futuros que se pueden realizar para continuar extendiendo los resultados obtenidos y presentados en esta memoria.

7.1. Conclusiones

La complejidad de los sistemas software actuales y la demanda de cambios en los sistemas por parte de los usuarios, para adaptarlos a nuevas circunstancias o para incluir nuevos servicios, son razones por las que la ingeniería del software debe disponer de técnicas, herramientas y modelos que le permitan afrontar estos retos. La arquitectura del software como disciplina y el DSOA como paradigma de desarrollo se han mostrado útiles para resolver estas cuestiones.

Por una parte, la arquitectura del software es una actividad del diseño que ayuda a los ingenieros de software a reducir la complejidad de los sistemas y su desarrollo, así como a definir su estructura de modo que el mantenimiento y la evolución de los mismos sea sencilla. Esto es así porque durante la AS se define la estructura de un sistema como un conjunto de componentes que realizan las computaciones y un conjunto de conectores que definen la interacción entre ellos.

Por otra parte, el DSOA es un paradigma que propone técnicas de modularización de los *crosscutting concerns* (propiedades transversales) a lo largo del ciclo de vida. Dado que el DSOA permite modelar como elementos software las propiedades que atraviesan los sistemas, éstos se pueden adaptar más fácilmente a los cambios y la evolución, frente

a la utilización de paradigmas tradicionales, en los que el código transversal queda disperso. Así, el DSOA mejora la calidad de los sistemas haciendo los diseños más claros y potenciado la reutilización del software.

Dicho esto, consideramos que es interesante tratar conjuntamente los conceptos de arquitectura del software y de DSOA, pues ambas aproximaciones facilitan la gestión de la complejidad de los sistemas y su uso conjunto potencia las características de cada uno de ellas. La conveniencia de definir un diseño arquitectónico orientado a aspectos surge cuando se observa que los *crosscutting concerns* también atraviesan los componentes arquitectónicos. Por ello, son necesarios mecanismos para identificar aspectos durante la arquitectura del software.

En este entorno es en el que se ha desarrollado el trabajo que se ha presentado. Se han analizado algunas propuestas realizadas en torno al DSOA concluyéndose que no todas proporcionan un soporte durante todo el ciclo de vida, desde la especificación a la implementación y mantenimiento.

Con el objetivo de aprovechar los beneficios del uso conjunto de las dos aproximaciones mencionadas, en esta memoria se ha presentado un marco de trabajo, *AOSA Space,* que combina los conceptos del desarrollo orientado a aspectos y la definición estructural de los sistemas, de modo que se reduce la complejidad de los sistemas desarrollados y se facilita su mantenimiento y adaptación. *AOSA Space* está constituido por un modelo, una metodología que lo desarrolla, un LDA-OA, que permite expresar formalmente la arquitectura del sistema desarrollado, y un juego de herramientas.

La propuesta es una contribución en el ámbito del desarrollo de sistemas orientados a aspectos. A lo largo del trabajo, y como una aportación del mismo, se ha mostrado la conveniencia de definir una arquitectura OA: el modelo propuesto hace algunas consideraciones metodológicas sobre cómo llevar a cabo la integración de los conceptos relativos a la separación de aspectos durante la especificación arquitectónica. Esto permite manejar la evolución de los sistemas considerando sus cambios (nuevos requisitos) como aspectos. Para lograrlo, los aspectos se extraen, se definen o se identifican durante la etapa de diseño arquitectónico (o durante la evolución) y se consideran artefactos de diseño que describen su comportamiento a alto nivel. Los aspectos, además, se incorporan de un modo transparente y se pueden manipular a lo largo del proceso de desarrollo. De este modo, se dispone de reconfiguración arquitectónica *ad hoc* del sistema cuando un arquitecto de software necesita incluir un nuevo requisito o un cambio no previsto, obteniéndose así un nuevo diseño a partir del existente, sin cambiar los componentes de diseño que formaban el *sistema inicial*. Como consecuencia, el diseño obtenido es claro, y la reutilización y evolución son sencillas.

En consecuencia, de la aplicación conjunta de los conceptos de AS y de DSOA y de la utilización de un modelo de coordinación exógeno orientado a control, como es *Coordinated Roles*, se obtiene la principal contribución de esta tesis:

> *Reducir el problema de la separación de aspectos, en tiempo de diseño, a uno de coordinación.*

Otra contribución de esta tesis es el modelo arquitectónico descrito, *AOSA Model*, que se caracteriza por:

- Ser *reflexivo* y con *dinamismo estructural,* ya que el modelo propone incluir nuevos componentes a partir de un sistema ya diseñado y el lenguaje en el que se apoya la propuesta lo soporta, pues permite la creación y destrucción de componentes, y de los enlaces correspondientes.
- La *reconfiguración* del sistema se lleva a cabo *en tiempo de diseño* en cuanto que *AOSA Model* propone redefinir la arquitectura del sistema incluyendo los elementos que representan la ampliación (como aspectos).
- Se dispone de *reconfiguración dinámica* una vez extendido el sistema, ya que los componentes de aspecto, si bien están ya especificados, no siempre tienen instancias activas o, incluso, puede haber varias activas a la vez.
- Permite, además, *reconfiguración programada*, según la clasificación propuesta por Endler [EnWe92].
- El modelo arquitectónico resultante es *altamente dinámico* por las facilidades que proporciona el LDA que se utiliza, que tiene gran capacidad de dinamismo.
- Permite diseñar sistemas orientados a aspectos independientes de la plataforma y de la tecnología subyacente.
- Los componentes no tienen que haber sido definidos bajo ningún modelo de componentes específico, sino que nos basamos en la especificación genérica del modelo de componentes de UML 2.0.

Una aportación añadida de *AOSA Space* es su utilización durante la evolución de los sistemas, en cuanto que, desde un punto de vista arquitectónico, se dan las normas para incluir nuevas características o requisitos, que sean ortogonales a los existentes en el *sistema inicial*.

Por otra parte, si se quiere relacionar *AOSA Space* con MDA, habría que decir que se ha especificado y desarrollado en el nivel de PIM (*Platform Independent Model*). En él, a partir de la definición gráfica se especifica un sistema equivalente en el LDA-OA *AspectLEDA* que se transforma automáticamente (mediante la herramienta *AOSA Tool/LEDA*) en una especificación arquitectónica en un LDA convencional.

Las características de los elementos del marco de trabajo *AOSA Space* se resumen en los apartados que siguen.

7.1.1. *AOSA Model*

AOSA Model es un modelo que describe arquitecturas software de sistemas complejos. En él, los *concerns* (materias de interés) que atraviesan los módulos arquitectónicos se han considerado como aspectos arquitectónicos que, a su vez, son requisitos identificados en etapas tempranas del ciclo de vida, durante la evolución o el mantenimiento. Los aspectos, se han considerado como componentes arquitectónicos

En *AOSA Model*, los componentes del sistema, conteniendo su funcionalidad, se han tratado como cajas negras. La interacción entre los aspectos y los componentes se lleva a cabo de un modo transparente a éstos, siendo los aspectos independientes del

contexto, por lo que se potencia su reutilización. Los aspectos se insertan en el sistema mediante la identificación de los puntos de corte localizados en las interfaces públicas de los componentes. La interacción entre los componentes del sistema y los componentes de aspecto se ha especificado en *AOSA Model* mediante la inclusión de unos componentes arquitectónicos que llevan a cabo las tareas de coordinación y composición de los aspectos y el sistema en el que se insertan; estos componentes responden a las características de los elementos coordinadores definidos en el modelo de coordinación en el que se basa su definición. Las acciones de composición se ejecutan dinámicamente en tiempo de ejecución, considerando la prioridad de ejecución entre aspectos si fuera necesario.

En el Capítulo 4 se muestran brevemente los diversos modelos de coordinación. Tras su estudio, según se explica, se concluyó que lo más conveniente, en nuestro caso, era aplicar un modelo de coordinación que permitiera que los nuevos elementos fueran insertados de un modo transparente a los componentes del *sistema inicial*. Se eligió por ello un modelo de coordinación exógeno dirigido por control, en particular *Coordinated Roles* [Mur01] que permite la inclusión de nuevos elementos –*coordinadores*- utilizando los protocolos de notificación de eventos.

El modelo proporciona las siguientes aportaciones:

- *AOSA Model* se ha definido para describir la arquitectura de sistemas complejos y facilitar la inclusión de aspectos durante el diseño arquitectónico de un sistema. Para especificar los diseños arquitectónicos, el modelo se basa en componentes, aspectos y un modelo de coordinación, por lo que integra conceptos de Desarrollo de Software Basado en Componentes (DSBC), de Desarrollo de Software Orientado a Aspectos (DSOA) y de coordinación.
- El modelo facilita el desarrollo de sistemas OA desde un punto estructural, a partir de las especificaciones que definen el sistema y un juego de requisitos que se incluyen en él en tiempo de diseño. Se sigue un modelo OA asimétrico durante la fase de diseño arquitectónico, lo que facilita la reutilización tanto de componentes como de aspectos.
- Se define un meta nivel que incluye los elementos que introduce el modelo (coordinadores y aspectos) para gestionar la extensión del *sistema inicial*. De este modo, el *sistema extendido* tiene capacidades reflexivas al poder modificar el comportamiento del *sistema inicial*. Los aspectos se mantienen independientes del contexto en el que se insertan, siendo los elementos coordinadores los que contemplan la información contextual.
- El metamodelo del sistema está basado en el estándar UML.

7.1.2. Metodología para *AOSA Model*

Las principales características de la metodología que se ha propuesto se basan en las siguientes etapas:

- A partir de un cierto sistema a desarrollar y considerando las especificaciones que lo definen, se realiza su diseño centrando el estudio únicamente en las especificaciones básicas, para determinar la estructura del *sistema inicial*. Aquellas especificaciones que se puedan considerar como aspectos no se incluyen en este momento.
- Después, la metodología propone representar el *sistema inicial* mediante diagramas UML (en concreto usando diagramas de casos de uso y de secuencia) y describir su arquitectura en un LDA convencional.
- A continuación, se incorporan al *sistema inicial* las especificaciones no consideradas previamente para obtener el s*istema extendido* con aspectos.
- Finalmente, se genera la descripción arquitectónica para el nuevo sistema en un LDA. Para ello, es necesario disponer de lenguajes que puedan expresar a nivel arquitectónico los conceptos de orientación a aspecto. En esta tesis se ha propuesto un LDA-OA que permite expresar la arquitectura del sistema OA diseñada siguiendo *AOSA Model*.

Como contribución de esta metodología, dentro del marco de trabajo *AOSA Space*, se puede destacar que cubre las actividades de diseño dentro del desarrollo de un sistema orientado a aspectos, actividades que se pueden aplicar iterativamente.

7.1.3. Un LDA-OA para *AOSA Model*

Como se ha mostrado en el Capítulo 3 de esta memoria, hay varias propuestas para desarrollar sistemas OA, pero sólo algunas proporcionan soporte lingüístico. En este sentido, pensamos que es interesante dotar a los sistemas OA de soporte formal, igual que en las propuestas de desarrollo de sistemas convencionales. Así, los sistemas OA deben poder expresarse también mediante lenguajes de descripción arquitectónica. Sin embargo, los LDA convencionales no permiten describir adecuadamente conceptos OA. De ahí la necesidad y el interés de definir LDA-OA.

En este trabajo de investigación hemos mostrado cómo es posible dotar al proceso de desarrollo de un sistema OA de soporte lingüístico durante la AS. *AspectLEDA* es un LDA-OA que extiende un LDA convencional, LEDA, y que cumple las premisas descritas en [Nav+02]. Así pues, una de las aportaciones de esta tesis es la definición de un soporte formal para los sistemas OA. En particular, se ha desarrollado un Lenguaje de Descripción Arquitectónica Orientado a Aspectos cuyas características son:

i) Se ha definido con un pequeño conjunto de instrucciones que extiende el lenguaje en el que se basa (LEDA).

ii) Las especificaciones arquitectónicas que describen se traducen a LEDA.

iii) La base formal de LEDA permite razonar sobre las propiedades de la arquitectura software del sistema.

iv) Los aspectos se consideran como componentes del lenguaje que se insertan en el sistema de modo transparente a los componentes que lo forman.

v) Dota de soporte formal a la descripción arquitectónica de los sistemas orientados a aspectos.

vi) La nueva arquitectura se puede verificar utilizando herramientas de validación adecuadas.

La definición del LDA-OA desarrollado para definir la arquitectura de sistemas orientados a aspectos ha puesto de manifiesto lo siguiente:

- Se pueden describir sistemas OA desde un punto de vista arquitectónico, sin necesidad de incluir nuevas abstracciones.
- El LDA se ha definido como extensión a un lenguaje de descripción arquitectónica regular, mostrándose que no es necesario la definición de nuevos lenguajes específicos para expresar arquitecturas software OA, sólo adaptar y extender alguno existente.
- Es especialmente interesante la característica de generación de código ejecutable en tiempo de diseño.
- Además, para comprobar la corrección de la arquitectura se ha elegido un LDA con soporte formal fuerte (cálculo π) que permite aplicar herramientas de validación y chequeo, a fin de asegurar que la arquitectura expresa adecuadamente los requisitos propuestos, sin que sea necesario esperar a realizar esta comprobación en tiempo de implementación.

7.1.4. Juego de herramientas

Otra aportación de *AOSA Space* es el juego de herramientas que se proporciona:

- Se ha diseñado una herramienta, *AOSA Tool/LEDA,* que facilita la traducción de la arquitectura del *sistema extendido,* expresada en *AspectLEDA*, al LDA en el que se apoya (LEDA), mediante un proceso de interpretación en dos pasos. Durante el proceso de generación del *sistema extendido* se realizan labores de validación sintáctica y semántica, y de coherencia de datos.
- Asimismo, realizada la traducción a LEDA, otra herramienta, *EVADeS*, facilita la generación y ejecución de un prototipo de la nueva arquitectura en Java.

7.1.5. Marco de trabajo

Figura 7.1 muestra, a modo de resumen, el marco de trabajo que se presenta como aportación final de esta tesis doctoral. En ella, las elipses con líneas finas

representan especificaciones en UML; las elipses con líneas gruesas simbolizan la representación formal de la arquitectura (en un LDA). En negrita y fondo coloreado se representa el sistema OA, expresado en UML y su descripción arquitectónica formal. Las flechas representan las acciones ejecutadas por el marco de trabajo. Los pasos que se siguen son los siguientes:

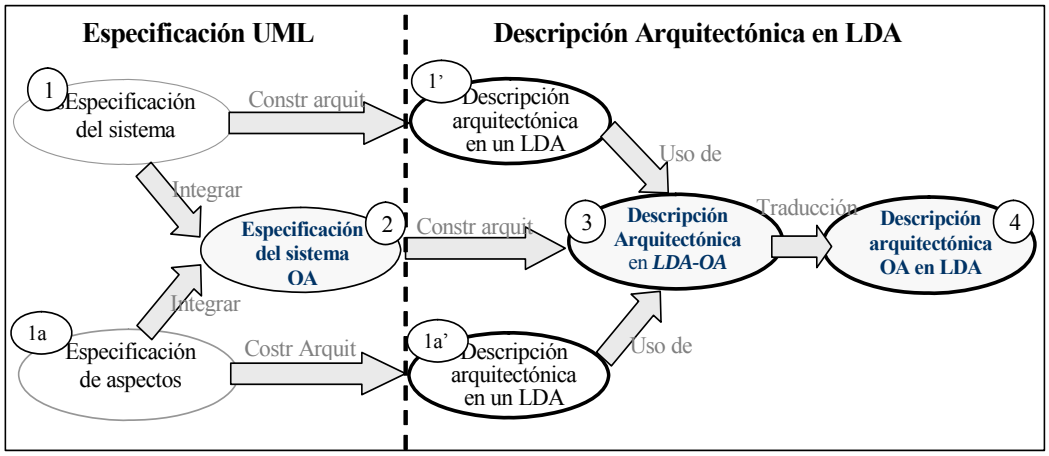


Figura 7.1. Marco de trabajo propuesto.

(1) El sistema se representa mediante un conjunto de especificaciones expresadas en su diagrama de casos de uso y los correspondientes diagramas de secuencia. Cada elemento debe describirse completamente a fin de elaborar la oportuna documentación. De este modo se obtiene el punto de vista externo del sistema. Este es el *sistema inicial*, que también debe describirse desde un punto de vista arquitectónico en un ADL (1').

A continuación, se debe expresar la modificación del comportamiento del *sistema inicial* incorporando la nueva funcionalidad: requisitos que se consideran como aspectos. Estos aspectos a insertar se tienen que describir adecuadamente; se representan como *use case extension* siguiendo la propuesta de Jacobson en [JaNg05]. Los nuevos requisitos se consideran componentes arquitectónicos (1a) y, por tanto, deben expresarse en el LDA (1a').

(2) La especificación del sistema OA, *sistema extendido*, se obtiene incorporando a las especificaciones del *sistema inicial* las de los nuevos requisitos. El diagrama de casos de uso inicial se extiende con los casos de uso que contienen las especificaciones de aspecto (como *use case extension*). Como consecuencia, los diagramas de secuencia también resultarán modificados. Para realizar la inserción de cada aspecto hay que considerar cierta información extra: sus condiciones de aplicación, dónde, cuándo y cómo deben aplicarse.

(3) Realizada la descripción arquitectónica del sistema OA, se debe expresar en un Lenguaje de Descripción de Arquitecturas Orientado a Aspectos. De este modo, se puede chequear la consistencia y corrección del *sistema extendido*.

(4) La representación del sistema OA en el LDA-OA *AspectLEDA* se traduce a un LDA convencional (LEDA), que permite a su vez generar un prototipo del sistema cuando se traduce a Java (pues el lenguaje elegido lo permite).

7.2. Trabajos futuros

En esta sección se enumeran algunos trabajos en curso o en fase de planteamiento.

A) Trabajos en curso

- Actualmente está en una fase muy avanzada de desarrollo un nuevo LDA-OA, *AAJ*, definido a partir de *ArchJava*. El nuevo lenguaje tiene unas características similares a *AspectLEDA*. El desarrollo de *AAJ* ha dado lugar a una publicación interna [BoNa07] y ha sido aceptado un artículo sobre el mismo en un congreso de índole nacional [BoNa08]. Una herramienta, como en el caso de *AspectLEDA,* facilita la labor del arquitecto de software en su tarea de describir arquitecturas de sistemas orientadas a aspectos.
- Actualmente está en fase avanzada un estudio que permitirá comparar el código Java generado por los dos LDA-OA desarrollados. El siguiente paso será comparar los resultados obtenidos tras la aplicación del *framework* desarrollado con los obtenidos al aplicar otras propuestas.
- En fase de estudio se encuentra el desarrollo de un nuevo lenguaje *Acme-OA* que permitirá obtener el diseño arquitectónico de un sistema OA expresado en *Acme*, tras aplicar *AOSA Model*. A su vez, los resultados obtenidos podrán compararse con los resultados tras la aplicación de *AspectLEDA* o *AAJ*.
- En fase de desarrollo está la definición de una versión mejorada de *AOSA Tool/LEDA*.

B) Trabajos futuros

- Una ampliación de *AOSA Tool/LEDA* que pensamos abordar es, a partir de la especificación LEDA generada, realizar automáticamente la conversión a especificaciones en cálculo π.
- Igualmente pensamos que es necesario completar el trabajo aquí presentado desarrollando algún entorno de trabajo o herramienta que facilite el testeo, validación y prueba de la arquitectura generada.
- Es nuestra intención ampliar o modificar *EVADeS* para que se pueda usar, no sólo con arquitecturas descritas en *LEDA*, sino también en *ArchJava* y/o *Acme*.
- En un estado de estudio preliminar se encuentra la definición de un entorno que integre las tres herramientas desarrolladas en el marco de esta tesis doctoral, así como la mejora de cada una de ellas.

- Dado el auge que está tomando hoy en día XML, se ha pensado que los datos de entrada y salida de los intérpretes (pasos intermedios de la metodología) se pueden expresar mediante esquemas XML. Además, estos esquemas se presentan como una manera sencilla y clara de comprender la estructuración de dichos ficheros. Por otro lado, con la integración de XML cabría la posibilidad de que el sistema generase su salida del *sistema extendido* en un nuevo lenguaje *XLEDA* y/o *XArchJava*.
- Por otra parte, y dado el interés que ha suscitado el nuevo paradigma de Desarrollo de Sistemas Dirigido por Modelos (DSDM), estamos estudiando la posibilidad de integrar *AOSA Space* en él. Sin embargo, hay que dejar constancia de que el marco de trabajo presentado es independiente de la plataforma, por lo que ha sido especificado y desarrollado en el nivel de PIM (*Platform Independent Model*): a partir de la definición gráfica (en este nivel), se especifica en un LDA-OA un sistema equivalente que se transforma automáticamente en una especificación arquitectónica en un LDA convencional. En este sentido, como trabajo futuro se piensan desarrollar transformaciones automáticas de modelo a texto: desde el modelo basado en UML a especificaciones textuales en un LDA-OA.

7.3. Publicaciones relacionadas

A) Publicaciones

Esta tesis doctoral se apoya en las siguientes publicaciones:

[NaPeMu00] **Navasa**, **Amparo,** Pérez, Miguel Ángel, Murillo, Juan M. Un modelo arquitectónico para desarrollos software basados en componentes funcionales y no funcionales. I Taller de trabajo en ingeniería del software basada en componentes distribuidos (IScDIS'00) en IV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'00). Valladolid, España. Informe técnico TR-12/2000. Universidad de Extremadura, pp. 109-118. Noviembre, 2000.

[Nav+02] **Navasa**, **A.** Pérez, M. A., Murillo, J. M., Hernández, J. Aspect-Oriented Architecture: a Structural Perspective. Proceeding of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. 1st International Conference on Aspect-Oriented Software Development (AOSD). Enschede, Holanda. Abril, 2002. http://trese.cs.utwente.nl/AOSDEarlyAspectsWS/workshop_papers.htm,

[NaPeMu02] **Navasa, A.,** Pérez, M. A., Murillo, J. M. Definición de un estilo arquitectónico para desarrollos software de sistemas complejos, basado en separación de aspectos. TR-13/2002. U. de Extremadura. 2002.

[NaPeMu04] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. Una arquitectura software para DSOA IX Jornadas de Ingeniería del Software y Bases de Datos. JISBD'04. ISBN 84-688-8983-0, pp. 267-278. Málaga, España. Noviembre, 2004.

[Nav+04] **Navasa**, **A.**, Palma, K., Murillo, J. M., Eterovic, Y. Dos modelos arquitectónicos para DSOA. II Taller de Desarrollo Software Orientado a Aspectos. DSOA'04 en JISBD 2004. TR-23/04 Universidad de Extremadura, pp. 19-26. Málaga, España. Noviembre, 2004.

[NaPeMu05a] **Navasa, A.,** Pérez, M. A., Murillo, J.M.. Aspectual Modelling at Architecture Design. Eds. R. Morrison y F. Oquendo LNCS 3527. pp. 41-58. Springer Verlag; Berlin Heidelberg. ISBN 3-540-26275-X. Junio, 2005.

[NaPeMu05b] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. Una aproximación metodológica para soportar la evolución de requisitos a partir de un modelo arquitectónico OA. X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2005). Granada, España. ISBN 84-9732-434-X, pp. 241-246. Septiembre, 2005.

[NaPeMu05c] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. *AspectLEDA*: Un lenguaje de descripción arquitectónica Orientado a Aspectos. Actas del III Taller de Desarrollo de Sistemas Orientados a Aspectos, en JISBD'05. TR 24/05 Universidad de Extremadura, pp. 24-31. Granada. Septiembre, 2005.

[NaPeMu07] **Navasa**, **A.,** Pérez, M. A., Murillo J.M. *AspectLEDA*: Extending an ADL with Aspectual Concepts. Ed F. Oquendo. LNCS 4758, pp. 330-334. Springer Verlag; Berlin Heidelberg. ISBN 978-3-540-75131-1. 2007.

[BoNa07] Botón, M., **Navasa, A.** *AAJ*: Un lenguaje de descripción arquitectónica orientado a aspectos. TR-27/2007.Universidad de Extremadura. 2007.

[GaNa08] García, C., **Navasa, A.** *EVADeS*: Un Entorno Visual de Asistencia al Desarrollo de Sistemas en LEDA. TR-30/2008. Universidad de Extremadura. 2008.

[NaPeMu08] **Navasa**, **A.** Pérez-Toledano, M. A., Murillo, J. M. An ADL dealing with aspects at software architecture stage. Information and Software Technology, in press. doi:10.1016/j.infsof. 2008.03.009. on line in: http://dx.doi.org/10.1016/j.infsof.2008.03.009. 2008.

[MeNa08] Merín, I, **Navasa, A.** *AspectLEDA*: Un lenguaje de descripción arquitectónica orientado a aspectos. TR 31/2008. Universidad de Extremadura. 2008.

[BoNa08] Botón, M., **Navasa, A.** *AAJ*: Un lenguaje de descripción arquitectónica orientado a aspectos. Actas de JISBD'08, pp. 361-366. ISBN 978-84-612-5820-8. Octubre, 2008.

B) Contribuciones a congresos

Las siguientes referencias son contribuciones a congresos:

[NaPeMu00] **Navasa**, **Amparo,** Ángel Pérez, Miguel, Murillo, J. M. Un modelo arquitectónico para desarrollos software basados en componentes funcionales y no funcionales. I Taller de trabajo en ingeniería del software basada en componentes distribuidos (IScDIS'00) en IV Jornadas de Ingeniería del Software y Bases de Datos (JISBD'00). Valladolid, España. Noviembre, 2000.

[NaPeMu01a] **Navasa**, **Amparo,** Ángel Pérez, Miguel, Murillo, J. M. Developing Component Based Systems using AOP Concepts. ECOOP Workshop on Advanced Separation of Concerns, ASoC'01. Budapest, Hungría. Junio, 2001.

[NaPeMu01b] **Navasa**, **Amparo,** Ángel Pérez, Miguel, Murillo, J. M. An Architectural Style to Integrate Components and Aspects. ECOOP Workshop on Feature Interaction in Composed Systems, FICS'01. Budapest, Hungría. Junio, 2001.

[NaPeMu01c] **Navasa**, **Amparo,** Ángel Pérez, Miguel, Murillo, J. M. Modelos de desarrollo software basados en componentes funcionales y no funcionales. VI Jornadas sobre Innovación y Calidad del Software. Universidad Europea CEES. Madrid, pp. 113-124. Julio, 2001.

[PeNaMu01] Ángel Pérez, Miguel, **Navasa**, **Amparo,** Murillo, J. M. Searching and Retrieving non-functional Components. 11 ECOOP Ws on PhD Students in Object Oriented Systems, PhDOOS'2001. Budapest, Hungría. Junio, 2001.

[Nav+02] **Navasa**, **A.** Pérez, M. A., Murillo, J. M., Hernández, J. Aspect-Oriented Architecture: a Structural Perspective. Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. 1st International Conference on Aspect-Oriented Software Development (AOSD). Enschede, Holanda. Abril, 2002.

[NaPeMu03] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. Using an ADL to Design Aspect Oriented Systems. 13 ECOOP Workshop on PhD Students in Object Oriented Systems, PhDOOS'2003. Tecnical Report. Darmstadt, Alemania. Julio, 2003.

[Nav+04] **Navasa**, **A.**, Palma, K., Murillo, J. M., Eterovic, Y. Dos modelos arquitectónicos para DSOA. II Taller de Desarrollo Software Orientado a Aspectos. DSOA'04 en JISBD 2004. Málaga, España. Noviembre, 2004.

[NaPeMu04] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. Una arquitectura software para DSOA. IX Jornadas de Ingeniería del Software y Bases de Datos. JISBD'04. Málaga, España. Noviembre, 2004.

[NaPeMu05a] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. Aspectual Modelling at Architecture Design. 2nd European Workshop on Software Architecture, EWSA 2005. Pisa, Italia. LNCS vol. 3527. Junio, 2005.

[NaPeMu05b] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. Una aproximación metodológica para soportar la evolución de requisitos a partir de un modelo arquitectónico OA. X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2005). Granada, España. Septiembre, 2005.

[NaPeMu05c] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. *AspectLEDA*: Un lenguaje de descripción arquitectónica Orientado a Aspectos. III Taller de Desarrollo de Sistemas Orientados a Aspectos, en JISBD'05. Granada, España. Septiembre, 2005.

[NaPeMu07] **Navasa**, **A.,** Pérez, M. A., Murillo, J. M. *AspectLEDA*: Extending an ADL with Aspectual Concepts First European Conference on Software Architecture (ECSA 2007). Aranjuez, España. Septiembre, 2007.

[BoNa08] Boton, M., **Navasa, A.** *AAJ*: Un Lenguaje de Descripción Arquitectónica Orientado a Aspectos. XIII Jornadas de Ingeniería del Software y Bases de Datos (JIDBD'08). Gijón, España. Octubre, 2008.

C) Colaboraciones

Además de los artículos mencionados, y fruto de la colaboración con las tesis [Per08] y [Cle07] del grupo de investigación, he participado también en la elaboración de los siguientes trabajos:

[Per+2] Pérez Toledano, M. A., Clemente Martín, P. J., **Navasa Martínez**, **A.,** Murillo Rodríguez, J. M. 12th workshop for Ph. Doctoral Students in Object Oriented Systems. TR-17/2002 Universidad de Extremadura. 2002.

[PeNaMu04a] Pérez-Toledano M.A., **Navasa Martínez A.**, Murillo Rodríguez J.M. Síntesis de los Diagramas de Secuencias de UML en Grafos de Comportamientos. Technical Report TR-22/2004. Universidad de Extremadura. 2004.

[PeNaMu04b] Pérez-Toledano M. A., **Navasa Martínez A.**, Murillo Rodríguez J. M. Documentación de componentes: Una aproximación basada en Diagramas de secuencia. Ampliación del artículo presentado en el taller de Nuevas Tecnologías de la Información. Colombian Journal of Computation, vol 5. nº 1, pp. 67-76. ISSN: 1657-2831. 2004.

[Per+05a] Pérez-Toledano M. A**., Navasa Martínez A**., Murillo Rodríguez J. M., Canal C. Síntesis de patrones de interacción a partir de diagramas de secuencias de UML. Actas de X Jornadas de Ingeniería del Software y Bases de Datos. ISBN: 84-9732-434-x, pp. 83-91. 2006.

[Per+05b] Pérez-Toledano M. A., **Navasa Martínez A**., Murillo Rodríguez J. M., Canal C. Using Interaction Patterns for Making Adaptors among Software Components. En los proceedings de 2º International Workshop on Coordination and Adaptation Techniques for Software Entities. ITI-05-07, TR 23/05, TR 119-2005, TR 006/2005, pp. 63-70. 2005.

[Per+05c] Pérez-Toledano M. A., **Navasa Martínez A**., Murillo Rodríguez J. M., Canal C. Desarrollo de Sistemas Basados en Componentes Utilizando Diagramas de Secuencia. En los proceedings de 8º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software (IDEAS'2005), pp. 229-240. 2005.

[Per+06a] Pérez-Toledano M. A., **Navasa Martínez A.**, Murillo Rodríguez J. M., Canal C. Evolución de Sistemas Orientados a Aspectos Utilizando Patrones de Interacción. En los proceedings de XI Jornadas de Ingeniería del Software y Bases de Datos. ISBN: 84-95999-99-4, pp. 514-520. 2006.

[Per06b] Pérez-Toledano M. A., **Navasa Martínez A.,** Murillo Rodríguez J. M., Canal C. Making Aspect Oriented System Evolution Safer. En los proceedings de 3º International Workshop on Reflection, AOP and Meta-Data for Software Evolution (ECOOP 2006). TR-13/2006 Universidad de Extremadura, pp. 23-35. 2006.

[Per06c] Pérez-Toledano M. A., **Navasa Martínez A**., Murillo Rodríguez J. M., Canal C. Definición de Máquinas de Estado Extendidas usadas en descripción de protocolos de interacción. Technical Report TR-23/2006. Univdersidad de Extremadura. 2006.

[Per+07a] Pérez-Toledano M. A., **Navasa Martínez A.**, Murillo Rodríguez J. M., Canal C. TiTan: a Framework for Aspect-Oriented System Evolution. En los proceedings de Second International Conference on Software Engineering Advances, IEEE Computer Society Press. 2007.

[Per07b] Pérez-Toledano M. A., **Navasa Martínez A.**, Murillo Rodríguez J. M., Canal C. Dynamic Adaptation Using Aspect-Oriented Programming. En los proceedings de 4º International Workshop on Coordination and Adaptation Techniques for Software Entities. ISBN:978-84-690- 993-6, pp. 53-63. 2007.

[Per+08] Pérez-Toledano M. A., **Navasa Martínez A.**, Murillo Rodríguez J. M., Canal C. A Safe Dynamic Adaptation Framework for Aspect-Oriented Software Development. Journal of Universal Computer Science. Special Issue on Practical Approaches to Software Adaptation. Septiembre, 2008.

D) Asistencia a congresos

A lo largo de los años que ha durado el desarrollo de esta tesis doctoral he asistido a los siguientes congresos y jornadas de trabajo:

- I Jornadas de Ingeniería del Software. Sevilla, 1996.
- CURSO: Ingeniería del Software y Reutilización: Aspectos Dinámicos y Generación Automática. Cursos Complementarios. Universidad de Vigo. Julio 1998.
- III Jornadas de Ingeniería del Software. Murcia, 1998.
- IV Jornadas de Ingeniería del Software y Bases de Datos. Cáceres, 1999.
- V Jornadas de Ingeniería del software y Bases de Datos. Valladolid, 2000.
- VI Jornadas de Ingeniería del software y Bases de Datos. Almagro, 2001.
- XI ECOOP. Budapest, 2001.
- VI Jornadas sobre Innovación y Calidad del Software. Madrid, 2001.
- I International Conference on Aspect-Oriented Software Development (AOSD). Enschede, 2002.
- XIII ECOOP. Darmstadt, 2003.
- IX Jornadas de Ingeniería del Software y Bases de Datos JISBD. Málaga, 2004.
- II European Workshop on Software Architecture, EWSA. Pisa, 2005.
- X Jornadas de Ingeniería del Software y Bases de Datos, (JISBD) en I CEDI. Granada, 2005
- XI Jornadas de Ingeniería del Software y Bases de Datos, (JISBD). Sitges, 2006.
- I European Conference on Software Architecture (ECSA). Aranjuez, 2007.
- XIII Jornadas de Ingeniería del Software y Bases de Datos, (JISBD). Gijón, 2008.

Bibliografía

[Acme] http://www-2.cs.cmu.edu/~acme/

[AcNi00] Acherman, F., Nierstrasz, O. Applications = Components + Scripts – A tour of Piccola. In Software Architecture and Component Technology. Enschede, Holanda, Kluwer Academic Press Pub. 2000.

[adml] http://www.opengroup.org/tech/architecture/adml/adml_home.htm. Dic, 2002.

[AlChNo02a] Aldrich, J. Chambers, C. Notkin D. Architectural reasoning in ArchJava. ECOOP. LNCS vol. 2374, pp. 334 – 367. ISBN: 3-540-43759-2. 2002.

[AlChNo02b] Aldrich, J. Chambers, C., Notkin D. ArchJava: Connecting Software Architecture to Implementation. In ICSE, ISBN: 1-58113-472-X, pp 187-197. 2002.

[Ald] Aldawud site: http://www.csam.iit.edu/¬oaldawud

[AlDoGa98] Allen, R., Douence, R., Garlan, D. Specifying and Analyzing Dynamic Software Architectures. Proceeding ofongerence on Fundamental Approaches to Software Engineering. LNCS vol 1382, pp 21-37. Lisboa, Portugal. 1998.

[All97] Allen R., A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMUCS-97-144, Pittsburgh, Pennsylvania, USA. May, 1997.

[Alv00] Álvarez F., AGRA: Sistema de Distribución de Objetos para un Sistema Distribuido Orientado a Objetos soportado por una Máquina Abstracta. Tesis doctoral, Universidad de Oviedo. Septiembre, 2000.

[And+01] Andrade, L. et al. A two –layer approach to Architecture evolution. Proceeding of the 4th workshop on OO Architectural Evolution in ECOOP'01. Technical Report Programming Technology Lab. Vrije Universiteit Brusel 2001. http://prog.vub.ac.be/OOAE/ECOOP2001/ooaesubmissions.pdf

[AOSD] Aspect-Oriented Software Development homepage: http://aosd.net.

[AOSDEur] AOSD Europe site: http://www.aosdeurope.net/.

[Ara+02] Araujo, J., Moreira, A., Brito, I., Rashid, A. Aspect-Oriented Requirements with UML. Proceeding of the AO Modelling with UML Workshop. 5th UML Conference. Dresden, Alemania. 2002.

[Arb04] Arbab, F., Reo: A Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science, vol. 14, Issue 3, pp. 299-366. Junio, 2004.

[Arb96] Arbab, F. The IWIM Model for Coordination of Concurrent Activities. 1st int. Conference on Coordination Models, Languages and Applications (Coordination'96) LNCS, vol 1061, pp 24-56. Springer Verlag. Cesena. Abril, 1996.

[ArchJava] http:// www.Archjava.org.

[ArHeSp93] Arbab, F., Hernan, I., Spilling, P. An Overview of Manifold and its implementation. Concurency: Practice and Exprerience, vol 5, nº 1, pp. 23-70, Also Report CS-R9142. Febrero, 1993.

[ArMo03] Araujo, J., Moreira, A. An Aspecutal Use Case Driven Approach. Proceeding of VIII Jornadas de Ingeniería del Software y Bases de Datos. Alicante, España. 2003.

[AsLuPf99] Asmann, U., Ludwig, A. Pfeifer, D. Programming Connectors in an Open Language en electronics Proceeding of 1st Working IFIP Conference on Software Architecture (WICSA). Febrero, 1999.

[Ast99] Astley, M. Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures. Tesis doctoral. Universidad de Illinois. 1999.

[BaCl04a] Baniassad, E., Clarke, S. Theme: An approach for Aspect-Oriented Analysis and Design. In 26th Int. Conf. on Software Engineering (ICSE 2004), pp 158-167, Edinburg, Escocia, IEEE Computer Society Press. Mayo, 2004.

[BaCl04b] Baniassad, E., Clarke, S. Finding Aspects in Requirements with Theme/Doc. Workshop on Early Aspects, Int Conference on AOSD, Lancaster, UK. 2004.

[BaClKa03] Bass, L., Clements, P., Kazman, R. Software Architecture in Practice. Addison Wesley, 2ª edicion. 2003.

[BaDu05] Barais, O., Duchien, L. SafArchie studio: An ArgoUML Extension to build safe architectures. In Architecture Description Languages, pp. 85-100. Springer-Verlag. 2005.

[BaDuMe05] Barais, O., Duchien, L., Le Meur, A. F. A Framework to Specify Incremental Software Architectures Transformations. Proceeding of the 31st Euromicro Conference on Software Engineering and Advanced Applications SEAA, pp 62-69. Porto, Portugal. IEEE Computer Society. Septiembre, 2005.

[Bal97] Balzer, B. Instrumenting, Monitoring and Debugging Software Architectures. Technical Report USC/ISI, 1997.

[BaMeDu06] Barais, O., Le Meur, A. F., Duchien, L. INRIA/LIFL, J. Lawall, Integration of New Concerns in a Software Architecture. Proceeding of the 13th annual International Symposium and Workshop on Engineering of Computer Based Systems, ECBS'06, pp 52-64. 2006.

[Bar+04] Barais, O. et al. TranSat: a Framework for the Specification of Software Architecture Evolution. Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04) in ECOOP. Oslo, Noruega. 2004.

[Bar+06] Barais, O., Lawall, J., Le Meur, A.-F. Duchien, L. Safe integration of new concerns in a software architecture. In Proceedings of the 13th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS'06), pages 52-64, Potsdam, Alemania. IEEE Computer Society. Marzo, 2006.

[Bat+06a] Batista, T., Chavez, C., García, A., Sant'Anna, C., Kulesza, U., Rashid, A. Castor, F. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. Workshop on Early Aspects, ICSE'06, pp 3-9, Shangai, China. Mayo, 2006.

[Bat+06b] Batista, T., Chavez, C., García, A., Kulesza, U., Sant'Anna, C., Lucena, C. Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs. Proceedings of the ACM Sigsoft. XX Brazilian Symposium on Software Engineering (SBES'06) Florianópolis, Brasil. Octubre, 2006.

[BeAk01] Bergman, L. M., Aksit, M. Composing Crosscutting Concerns using Composition Filters. Communications of ACM 44, (10), pp. 51-57. Octubre, 2001.

[BeCoHe06] Berg, K. van den, Conejero, J.M., Hernández, J. Identification of Crosscutting in Software Design in Aspect Oriented Modeling Workshop in AOSD Conference. 2006.

[BeCoHe07] Berg, K. van den, Conejero, J.M., Hernández, J. Analysis of Crosscutting in Early Software Development Phases Based on Traceability. Transactions on AOSD III, LNCS vol. 4620 Ed. Springer Berlin / Heidelberg. pp. 73 – 104. ISBN: 978-3-540-75161-8. http://www.springerlink.com/content/g6p15816415k/. 2007.

[BiAlHa06] Bierhoff, K., Aldrich, J., Han S. A Language-based Approach to Specification and Enforcement of Architectural Protocols. Technical Report CMU-ISRI-07-121. Carnegie Mellon University. http://www.cs.cmu.edu/~kbiernof/papers/protocol-tr.pdf. Abril, 2006.

[Bla+00] Blait, G. et al. The role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. Proceedings of Middleware 2000. LNCS IFIO/ACM. Abril, 2000.

[BoNa07] Boton, M., Navasa, A. AAJ: Un Lenguaje de descripción arquitectónica Orientado a Aspectos. Technical Report TR-27/2007. Universidad de Extremadura. 2007.

[BoNa08] Boton, M., Navasa, A., *AAJ*: Un Lenguaje de Descripción Arquitectónica Orientado a Aspectos. Actas de JISBD'08, pp. 361-366. ISBN: 978-84-612-5820-8. XIII Jornadas de Ingeniería del Software y Bases de Datos. Octubre, 2008.

[BrIv69] Brooks, F. P., Iverson, K. E. Automatic Data Processing, New York. Wiley. ISBN: 0471106054. 1969.

[BrMo03] Brito, I., Moreira, A. Towards a Composition Process for Aspect-Oriented Requirements. Workshop on Early Aspects. AOSD conference. Boston, USA. 2003.

[Bro95] F.P. Brooks Jr. The Mythical Man-Month: Essays on Software Engineering. Addison Wesley, 2nd. Ed. ISBN: 0-201-83595-9. 1995.

[Bus+96] Buschmann, F., Meunier, R. Rohnert, H., Sommerlad, P., Stal, P. Pattern-Oriented Software Architecture. John Wiley & Sons. 1996.

[Can00] Canal, C. Un lenguaje para la especificación y validación de arquitecturas software. Tesis doctoral. Universidad de Malaga. 2000.

[CaPiAn05] Cazzola, W. Pini, S., Ancona, M. AOP for software evolution: a design oriented approach in Proceedings of the 2005 ACM symposium on Applied computing (SAC 2005), pp. 1346-1350. ISBN: 1-58113-964-0. 2005.

[CaPiTr01] Canal, C., Pimentel, E., Troya, J.M. Compatibility and Inheritance in Software Architecture. Science of Computing Programming, vol 41, nº 2, pp. 105-130. 2001.

[CaPiTr99] Canal, C., Pimentel, E., Troya, J.M. Specification and Refinement of Dynamic Software Architectures. 1st IFIP Conference on Software Architecture, WICSA'99. San Antonio, USA. Software Architecture, ISBN: 0-7923-8453-9. Kluwer Academic Publishers, pp. 107–126. Febrero, 1999.

[Chi+05] Chitchyan R., Rashid A., Sawyer P., Garcia A., Pinto M., Bakker J., Tekinerdogan B., Clarke S., Jackson A. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design**.** Lancaster University, Lancaster, AOSD-Europe Deliverable D11, http://www.aosdeurope.net/. AOSD-Europe-ULANC-9. Mayo, 2005.

[CiGoZa96] Ciacarini, P., Gorrieri, R., Zavattaro, G. Towards a Calculus for Generative Communication. Proceeding of IFIP Conference on Formal Methods for Open Object-based Distributed Systems, pp. 289-306. Paris, 1996.

[Cla02] Clarke, S. Extending Standard UML with Model COmposition Semantics. Science of Computer Programming, vol 44, nº 1, pp. 71-100. Julio, 2002.

[ClBa05] Clarke, S., Baniassad, E. Theme: AO Analysis and Design. http://www.dsg.cs.tcd.ie/index.php?category_id=353. 2005.

[Cle02] Clements P. et al. Documenting Software Architecture: Views and Beyond. Addison Wesley. 2002.

[Cle07] Clemente, P. J. Desarrollo dirigido por modelos de sistemas basados en componentes y aspectos. Tesis doctoral. Universidad de Extremadura. 2007.

[Cle96] Clements, P. Coming attractions in Software Architecture. Technical Report, CMU/SEI-96-TR-008, ESC-TR-96-008. Enero, 1996.

[ClKaKl02] Clements, P., Kazman, R., Klein, M. Evaluating Software Architecture, Addison Wesley. 2002.

[ClNo96] Clements, P., Northrop, L. Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, ESC-TR-96-003. Febrero,1996.

[CoHe08] Conejero, J.M., Hernández, J. Analysis of crosscutting features in software product lines. International Conference on Software Engineering. Proceedings of the 13th Int. workshop on Software architectures and mobility. Leipzig, Germany pp. 3-10, http://portal.acm.org/citation.cfm?id=1370828.1370831&coll=ACM&dl=ACM&CFID=32128684&CFTOKEN=11363420 ACM. ISBN: 978-1-60558-032-6. 2008.

[Col98] Coleman, D. A. Use Case Template: Draft for discussion. Fusion Newsleter, http://www.npl.hp.com/fusion/md-newsletters.html. Abril, 1998.

[Con+07] Conejero, J.M., Hernández, J., Jurado, E., Berg, K. van den. Crosscutting, what is and what is not? A Formal Definition based on a Crosscutting Patern. Technical Report TR28/07. http://quercusseg.unex/chemaam/research/TR3_07.pdf.

[Cue+01] Cuesta, C., Fuente, P., Barrio, M., Beato, E. Dynamic Coordination Architecture through the use of Reflection. Proceeding of 16th symposium on Applied Computing (SAC'01), pp. 134-140. Las Vegas. ACM Press. Marzo, 2001.

[Cue+02] Cuesta, C., Fuente, P., Barrio, M., Beato, E. Introducting Reflection in Architecture Description Lenguages. En J. Bosch, M. Gentleman, C. Hofmeister, J. Kuusela, Ed., Software Architecture: System Design, Development and Maintenance, pp. 143-156. 2002.

[Cue+04a] Cuesta, C., Romay, P, Fuente, P., Barrio, M. Reflection-Based Aspect-Oriented Software Architecture. In Proceedings of EWSA'2004. LNCS vol. 3047, pp. 43-56. Springer, ISBN 3-540-22000-3. 2004.

[Cue+04b] Cuesta, C., Romay, P., Fuente, P., Barrio, M. Aspectos como Conectores en Arquitectura del Software. En II Jornadas de Trabajo Dynamica (Dynamic and Aspect-Oriented Modelling for Integrated Component-Based Architectures), pp. 63-72. Noviembre, 2004.

[Cue+04c] Cuesta, C., Romay, P., Fuente, P., Barrio, M. Arquitectura del Software Orientada a Aspectos: una perspectiva para la Arquitectura Dinámica. En JISBD`04, pp. 37-48. 2004.

[Cue+05] Cuesta, C., Romay, P., Fuente, P., Barrio, M. Architectural Aspects of Architectural Aspects, 2nd European Workshop, EWSA 2005. LNCS 3572, pp. 247- 262, Morrison and Oquendo Eds. ISBN: 3-540-26275-X. 2005.

[Cue02] Cuesta, C. E. Dynamic Software Architecture based on Reflection. Tesis doctoral. Universidad de Valladolid. Julio, 2002.

[DaHo01] Dashofy, E., van der Hoek, A. Representing Product Family Architectures in an Extensible Architecture Description Language. 4th International Workshop on Software Product-Family Engineering PFE, pp. 330-341. Octubre, 2001.

[DaHoTa01] Dashofy, E., Hoeck, A., Taylor, R. A Highly-Extensible, XML-Based Architecture Description Language. Proceeding of the working IEEE/IFIP Conference on Software Architectures (WICSA´01). Amsterdam. 2001.

[Dij68] Dijkstra, E. The Structure of the Multiprogramming System. Communications of the ACM, 26(1), pp. 49-52. 1968. Enero, 1968.

[Dij76] Dijkstra, E. A Discipline of Programming. Prentice-Hall, 1976.

[Dul90] Dulay, N. A configuration Language for Distributed Programming. Tesis Doctoral. Universidad de Londres. 1990.

[DuRi97] Duchase, S., Richner, T. Executable Connectors: Towards Reusable Design Elements. En Software Engineering. Proceedings of ESEC/FSE'97, LNCS vol. 1301, pp. 483-500. 1997.

[Early] Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design web site, http://www.early-aspects.net/.

[Early02] Workshop on Early Aspect: Aspect-Oriented Requirements Engineering and Architecture Design. 1st International Conference on Aspect-Oriented Software

Development (AOSD). April 2002. Enschede. Holanda Web site download. http://trese.cs.utwente.nl/AOSD-EarlyAspectsWS/workshop_papers.htm

[EnWe92] Endler M., J. Wei. Programming Generic Dynamic Reconfigurations for Distributed Applications. Proceedings of the 1st Int. Workshop on Configurable Distributed Systems, pp. 68-79, IEE. 1992.

[Fel03] Felici, M. Taxonomy of Evolution and Dependability in Proceeding of the 2nd Int. workshop on Unanticipated Software Evolution (USE'03) in ETAPS'03, pp. 95-104. Warsovia, Polonia. 2003.

[Fie00] Fielding, R. Architectural Syles and the Design of Network-based Software Architectures. Tesis doctoral. University of California, Irvine. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000.

[FiFr00] Filman, R. E., Friedman, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. Proceeding of the Advanced Separation of Concerns Workshop. OOPSLA'00, Minneapolis, USA. Octubre, 2000.

[Fil+05] Filman, R. et al. Aspect Oriented Software Development. Addison Wesley. 2005.

[FuGaPi06] Fuentes, L. Gámez, N., Pinto, M. DAOPxADL: Una extensión del Lenguaje de Definición de Arquitecturas xADL con Aspectos. IV Taller de Desarrollo Software Orientado a Aspectos, en JISBD'06, pp. 41-50. TR-24/06 Universidad de Extremadura. 2006.

[GaMoWi00] Garlan D., Monroe R. T., Wile D., Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (ed), Cambridge University Press, pp. 47-68. 2000.

[GaMoWi97] Garlan D. Moore, R, Wile, D. ACME: An Architecture Description Interchange Language. Proceeding CASCON'97. Noviembre, 1997.

[GaNa08] Garcia, C., Navasa, A. EVADeS: Un Entorno Visual de Asistencia al Desarrollo de Sistemas en LEDA. TR-30/2008 Universidad de Extremadura. 2008.

[GaNo91] Garlan, D. Notkin, D. Formalizing Design Spaces: Implicit Invocation Mechanisms. Proceeding of VDM: Formal Software Development Methods. LNCS, vol. 551. Springer Verlag,. 1991.

[GaPe94] Garlan, D., Perry, D. Software Architecture: Practice, Pitfalls, and Potential' Panel Introduction, 16th International Conference on Software Engineering, Sorrento IT. Mayo, 1994.

[GaPe95] Garlan, D., Perry, D. Introduction to the Special Issue on Software Architecture, IEEE Transactions on Software Engineering, 21:4. Abril, 1995.

[Gar+00] Garlan, D., Monroe, R.T., Wile, D. Acme: Architectural Description of Component Based Systems. Foundations of Component-Based Systems, Cambridge University Press, pp. 47-68. 2000.

[Gar+06] García, A., Chavez, C., Batista, T., Sant'Anna, C., Kulesza, U., Rashid, A., Lucena, C. On the Modular Representation of Architectural Aspects. Proceedings of the 3th European Workshop on Software Architecture, Nantes. Francia. Septiembre, 2006.

[Gar95] Garlan, D. What is style?. Proceedings of IWASS. Abril, 1995.

[GaSh93] Garlan, D., Shaw, M. An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering, vol. 1. Ed. V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jersey. 1993.

[Gel85] Gelerter, D. Generative communication in Linda. ACM Transactions on Programming Languages and Systems, vol 7, nº 1, pp. 80-112. Enero, 1985.

[Gia99] Giannakopoulou, D. Model Checking for Concurrent Software Architectures. Tesis Doctoral. Universidad de Londres. 1999.

[GrMu98] Grundy J. C., Mugridge W. B., Hosking J. G. Static and dynamic visualisation of component-based software architectures. The 10th International Conference on Software Engineering and Knowledge Engineering, KSI Press, San Francisco, California, USA. 18-20 June, 1998.

[Gru00] Grundy, J. Multiperspective Specification, Design, and Implementation of Software Components Using Aspects. Int. Journal of Software Engineering and Knowledge Engineering, vol 10, (6), pp. 713-734. Diciembre, 2000.

[Gru99] Grundy, J. Aspect-Oriented Requirements Engineering for Component-Based Software Systems, in 4th IEEE Int symposium on RE, IEEE Computer Society. Limerik, Ireland, pp. 84-91. 1999.

[HaOs93] Harrison, W., Ossher, H., Subject-Oriented Programming – A Critique of Pure Objects, in Proceeding of 1993 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93). ACM Press. Septiembre, 1993.

[Har+02] Harrison W., Harold L., Ossher H., Tarr, P. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. IBM Research Report RC22685 (W0212-174), T.J. Watson Research Center, IBM. Diciembre, 2002.

[HeMeWe02] Heckel, R., Mens, T., Wermelinger, M. Towards uniform Support throughout the Software Life-Cycle. Proceedings on Software Evolution'02. LNCS vol. 2505/2002, pp. 450-454. ISBN: 3-540-44310-X. 2002.

[Her03] Herrero, J. L. Propuesta de una plataforma, lenguaje y diseño, para el desarrollo de sistemas orientados a aspectos. Tesis doctoral. Universidad de Extremadura. 2003.

[IEEE] Recommended Practice for Software Architecture Descriptions of Software Intensive Systems. IEEE Presss. 2000.

[IES06] IEEE Software, Vol 23,nº2. Marzo/Abril 2006.

[Jac03] Jacobson I. Use Cases and Aspects-Working Seamlessly Together. Journal of Technology, vol 2, pp. 7-28. 2003.

[JaNg05] Jacobson, I., Ng Pan-Wei. Aspect-Oriented Software Development with Use Cases. ISBN: 0-321-26888-1. Addison Wesley. 2005.

[JBoss] JBOSS AOP, http://www.jboss.org/developers/projects/jbossaop. 2005.

[KaKa03] Katara M., Katz S., Architectural Views of Aspects. The 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), Boston, Massachusetts, USA. 2003.

[Kan03] Kande M. A concern-oriented approach to software architecture. Tesis doctoral, Lausanne, Switzerland: Swiss Federal Institute of Technology (EPFL). 2003.

[KaSt01] Kande, M. Strohmeier, A. Modeling Crosscutting Concerns using Software Connectors. ASoC3. Tampa Bay, Florida. 2001.

[Kat93] Katz, S. A Superimposition Control Construct for Distributed Systems. ACM Transactions on Programming Languages and Systems, vol 15, nº 2, pp. 337-356. Abril, 1993.

[Kaz+94] Kazman, B. et al, SAAM: A Method for Analyzing the Properties of the Software Architecture. Proc 16th Int. Conf. Software Engineering (ICSE'94) IEEE press, pp. 81-90. 1994.

[Kic+01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G. Getting Started with AspectJ. Communications of the ACM, vol 44, nº 10, pp. 59-65. Octubre, 2001.

[Kic+96] Kiczales, G. et al. Aspect-Oriented Programming. Special Issue in Object Oriented Programming. ECOOP'96. Linz, Australia. 1996.

[KlKa99] Klein, M., Kazman, R. Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University. 1999.

[KoCl94] Kogut, P, Clements, P. Features of Architecture Description Languages. Draft of a CMU/SEI Technical Report. Diciembre, 1994.

[KrMa90] Kramer, J., Magee, J. The evolving Philosophers Problem: Dynamic Change Management. IEEE Transaction on Software Engineering. Noviembre, 1990.

[KrObSt06] Kruchten, P., Obbink, Stafford, P. The Past, Present and Future of Software Architecture. IEEE Software, vol 23, nº 2, pp. 22-30. Marzo/Abril, 2006.

[Kru91] Kruchten, P. Un Processus de Developpment de Logiciel Iteratif et centré sur l'Architecture. Proc 4eme Congres de Genie Logiciel, EC2, pp. 369-378. 1991.

[Kru95] Kruchten, P. The 4+1 View Model of Architecture. IEEE Software, vol 12, nº 6, pp. 45-50. 1995.

[Led99] Ledoux, T. OpenCorba: a Reflective Open Broker. Proceedings of the second International Conference on Reflection. LNCS 1616, pp. 197-214, Saint Malo. Julio, 1999.

[Leh96] Lehman, M. M. Laws of software evolution revised. In Proceeding of the European Workshop on Software Process Technology, pp. 108-124. 1996.

[Let+98] Letelier, P., Sánchez, P., Ramos, I., Pastor, O. OASIS 3.0: A formal language for the object oriented conceptual modelling. Universidad Politécnica de Valencia, SPUPV-98.4011, ISBN: 84-7721-663-0. 1998.

[Lie+99] Lieberherr K., Lorenz D., Mezini M., Programming with Aspectual Components. Technical Report NU-CCS-99-01, pp. 1-27, Northeastern University, Boston, Massachusetts. Marzo, 1999.

[LiOrOv01] Lieberherr, K., Orleans, D., Ovlinger, J. Aspect-Oriented Programming with Adaptive Methods. Communications of ACM, vol. 44, nº 10, pp. 39-41. Octubre, 2001.

[Lop02] Lopez, J. A. Generación de prototipos a partir de especificaciones arquitectónicas. PFC. Universidad de Málaga. Junio, 2002.

[Luc+95] Luckham, D. C., Kenney, J. J., Augustine L. M., Vera J., Bryan D., Mann W. Specification and Analysis of Software Architecture using Rapide. IEEE Transactions on Software Engineering, vol 21, nº 4, pp. 336-355. Abril, 1995.

[Mae87] Maes, P. Concepts and Experiments in Computational Reflection. In Proceedings of OOPSLA'87. SIGPLAN Notices, New York, vol 22, nº 12, pp. 147-169. 1987.

[Mag+95] Magee, J., Dulay, N., Eisenbach, S., Kramer, J. Specifying Distributed Software Architectures. Proceeding of European Software Engineering Conference. Septiembre, 1995.

[MaJaDe96] Malenfant, J., Jacques, M., Demers, F. A Tutorial on Behavioural Reflection and its Implementation. Proceedings of Reflection 96, pp. 1-20, 1996.

[MaKr96] Magee J., Kramer J., Dynamic Structure in Software Architectures. ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4), pp. 3-14, San Francisco, California. Octubre, 1996.

[Mar01] Marcos, C., Patrones de Diseño como entidades de primera clase. Tesis Doctoral. U.N.C. de la Provincia de Buenos Aires. Abril, 2001.

[McA96] McAffer, J. Engineering the Meta-Level. Proceedings on Reflection'96, pp. 39-61. Abril, 1996.

[MeBu05] Mencl, V., Bure, T. Microcomponent-Based Component Controllers: A Foundation for Component Aspect. In Proceedings of the 12th Asian-Pacific Software Engineering Conference. APSEC'05, pp. 729-738. 2005

[Med96] Medvidovic, N. ADLs and Dynamic Architecture Changes. Proceedings of the ISAW/SIGSOFT 1996, pp. 24-27. 1996.

[Med99] Medvidovic, N. Architecture-based Specification –time Software Evolution. Tesis doctoral, University of California. 1999.

[MeDaTa07] Medvidovic, N., Dashof, E. M., Taylor, R. N. Moving Architectural Descriptions from under the Thechnology Lamppost. En Information and Software Technology, vol 49, nº 1, pp. 12-31. 2007.

[MeMeTo04] Mens, T., Mens K., Tourne, T. Aspect-Oriented Software Evolution. In ERCIM news. Special Issue on Automate Software Engineering, nº 58. Julio, 2004.

[Men+98] Mens, K., Lopes, C., Tekinerdogan, B., Kiczales, G. Aspect-Oriented Programming. In Bosch, J. and Mitchell, S. (Eds), ECOOP'97 Workshop Reader, LNCS 1357, pp. 481-494. 1998.

[Men02] Mens, K. Architectural Aspects, Workshop on Early Aspects. AOSD 2002. http://trese.cs.utwente.nl/AOSD-EarlyAspectsWS/Papers/AllEarlyAspectsPapers.pdf

[MeNa08] Merín, I, Navasa, A. AspectLEDA: Un Lenguaje de Descripción Arquitectónica Orientado a Aspectos. TR 31/2008. Universidad de Extremadura. 2008.

[MeOs03] Mezini, M., Ostermann, K. Conquering Aspects with Caesar. International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 90- 100, Boston, Massachusetts, USA. Marzo, 2003.

[MeRoTa99] Medvidovic N., Rosenblum D. S., Taylor R. N. A Language and Environment for Architecture-Based Software Development and Evolution. The 21st International

Conference on Software Engineering (ICSE'99), Los Angeles, California. Mayo, 1999.

[MeTa00] Medvidovic N., Taylor R. N. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, vol. 26, nº 1. Enero, 2000.

[MeTa97] Medvidovic N., Taylor R. N., Architecture Description Languages. Software Engineering – ESEC/FSE, Springer Verlang, LNCS 1301, Zurich, Switzerland. Septiembre, 1997.

[Mon98] Monroe, R. Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163. Carnegie Mellon University School of Computer Science. Octubre, 1998.

[MoRaAr05] Moreira, A., Rashisd, A., Araujo, J. Multidimensional Separation of Concerns, in Int. Conf. on Requirements Engineering, IEEE Computer Society. 2005.

[Mur+99] Murillo, J. M., Hernandez, J., Sánchez, F., Alverez, L. Coordinated Roles: Promoting Re-Usability of Coordinated Active Objects Using Event Notification Protocol. Coordination Proceeding, pp. 53-68. 1999.

[Mur01] Murillo, J.M. Coordinated Roles: un modelo de coordinación de objetos activos. Tesis doctoral. Universidad de Extremadura. 2001.

[NaPeMu04] Navasa, A., Pérez, M. A., Murillo, J. M. Una arquitectura software para DSOA IX Jornadas de Ingeniería del Software y Bases de Datos. JISBD'04. ISBN: 84-688-8983-0, pp. 267-278. Málaga, España. Noviembre, 2004.

[NaPeMu05a] Navasa, A., Perez, M. A., Murillo, J. M. Aspect Modelling at Architecture Design. EWSA 2005. LNCS 3527, pp. 41-58, Ed. R. Morrison and F. Oquendo. Springer Verlag. Berlin Heidelberg. ISBN: 3-540-26275-X. 2005.

[NaPeMu05b] Navasa, A., Perez, M. A., Murillo, J. M. *AspectLEDA*: Un Lenguaje de Descripción Arquitectónica Orientado a Aspectos. En actas del III Taller de Desarrollo de Sistemas Orientados a Aspectos, en JISBD'05, pp. 24-31. Granada. 2005.

[NaPeMu07] Navasa, A., Pérez, M. A., Murillo J.M. AspectLEDA: Extending an ADL with Aspectual Concepts. Ed F. Oquendo. LNCS 4758. pp. 330-334. Springer Verlag; Berlin Heidelberg. ISBN: 978-3-540-75131-1. 2007.

[NaPeMu08] Navasa, A. Pérez-Toledano, M. A., Murillo, J. M. An ADL dealing with aspects at software architecture stage. Information and Software Technology, en prensa. doi:10.1016/j.infsof. 2008.03.009. on line in: http://dx.doi.org/10.1016/j.infsof.2008.03.009. 2008.

[NATO69] Naur, P., Randell, B. (Eds.). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Alemania. Octubre, 1968. Brussels, Scientific Affairs Division, NATO. 1969.

[NATO70] Randell, B., Buxton, J.N. (Eds.). Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Roma, Italia. Octubre, 1969, Brussels, Scientific Affairs Division, NATO. 1970.

[Nav+02] Navasa, A., Perez, M. A., Murillo, J. M., Hernandez, J. Aspect-Oriented Software Architecture: A structural Perspective. Workshop on Early Aspect. 1st AOSD

Conf. http://trese.cs.utwente.nl/AOSDEarlyAspectsWS/workshop_papers.htm, Enschede, Holanda. Abril, 2002.

[Nav+04] Navasa, A., Palma, K., Murillo, J. M., Eterovic, Y. Dos modelos arquitectónicos para DSOA. II Taller de Desarrollo Software Orientado a Aspectos. DSOA'04 en JISBD 2004. TR-23/04. Universidad de Extremadura, pp. 19-26. Málaga, España. Noviembre, 2004.

[NuKrFi94] Nuseibeh, B., Kramer, J., Finkekstein, A. Framework for Expressing the Relationships between Multiple Views in Requirements Specification. IEEE Transactions on Software Engineering, vol 20, nº 10, pp. 760-773. Octubre, 1994.

[OGR] OpenGroup site www.opengroup.org

[OMG] OMG home page: http://www.omg.org/

[OMG-MDA] Model Driven Architecture (MDA) http://www.omg.org/docs/omg/03-06-01.pdf y *www.omg.org/mda/*

[Ort02] Ortín, F. Sistema computacional de programación flexible diseñado sobre una máquina abstracta reflectiva no restrictiva. ISBN: 84-8317-304-2. Tesis Doctoral. 2002.

[OsTa00] Ossher, H., Tarr, P. Multi-Dimensional Separation of Concerns and the Hyper-Space Approach. In Proceeding of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer. 2000.

[OsTa01] Ossher, H., Tarr, P. Using Multidemensional Separation of Concern to (re)shape. Evolving Software Communications of the ACM, vol 44, nº 10, pp. 43-50. 2001.

[PaAr98] Papadopoulus, G. Arbab, F. Coordination Models and Languages, en The Engineering of Large Systems, vol. 46 de Advances in Computers. Acedamic Press. Agosto, 1998.

[PaEtMu05] Palma, K., Eterovic, Y., Murillo, J.M. Extending the Rapide ADL to Specify Aspect-Oriented Software Architectures. Technical Report, University of Extremadura. España. 2005.

[Pal04] Palma, K. Using a Coordination Model to Specify Aspect-Oriented Software Architectures. Master Thesis, Universidad Católica de Chile, 2004.

[Par72] D. Parnas. On the Criteria for Decomposing Systems into Modules. Communications of the ACM, vol. 15, nº 12, pp. 1053-1058. Diciembre, 1972.

[Pau+04] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L. JAC: an Aspect-Based Distributed Dynamic Framework. Software – Practice and Experience, vol 34, pp. 1119-1148. 2004.

[Per+05] Perez, J., Ali, N., Carsí, J. A., Ramos, I. Dynamic Evolution in Aspect-Oriented Achitectural Models. In 2nd European Workshop, EWSA'05. LNCS 3572, pp. 59-76, Morrison and Oquendo Eds. ISBN: 3-540-26275-X. 2005.

[Per+06] Perez, J., Navarro, E., Letelier, P., Ramos, I. A Modelling Proposal for Aspect-Oriented Software Architectures. 13th IEEE Conference on the ECBS. IEEE Computer Society Press. Postdam, Alemania. Marzo, 2006.

[Per06] Perez, J. PRISMA: Aspect-Oriented Architectures. Tesis Doctoral. Universidad Politécnica de Valencia, 2006.

[Per08] Perez-Toledano, M.A. Titan: un framework de modelado para el estudio de la integración de aspectos en un sistema software. Tesis doctoral. Universidad de Extremadura. 2008.

[Per94] Perry, D. E. Issues in Process Architecture, 9th International Software Process Workshop, Airlie VA. Octubre,1994.

[Pes+06] Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L. A model for Developing Component-Based and Aspect-Oriented Systems. Proceeding of the 5th International Symposium on Software Composition, SC'06, Vienna, Austria. LNCS 4089, pp. 259-274. Marzo, 2006.

[PeSeDu04] Pessemier, N., Seinturner, L., Duchien, L. Components, ADL and AOP: Towards a Common Approach, Workshop on Reflection, AOP and Meta-Data for Software Evolution. ECOOP Oslo, Norway. 2004.

[PeSu98] Peterson, J., Sulzmann, M., Análisis of Architectures using Constraint-Based types. Technical Report YALEU/DCS/RR-1157, Department of Computer Science, University of Yale. 1998.

[PeWo89] Perry, D. E., Wolf, A. L. Software Architecture. August 1989.

[PeWo92] Perry, D. E., Wolf, A. L. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17:4. Octubre, 1002.

[PiFuTr03] Pinto, M., Fuente, L., Troya, J. M. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. Proceeding of the 2nd International Conference in Generative Programming and Component Engineering, GPCE'03. Springer-Verlag, LNCS 2830, ISSN: 0302-9743, Erfurt, Germany, pp. 118-137. Septiembre, 2003.

[PiFuTr05] Pinto, M., Fuente, L., Troya, J. M. A Dynamic Component and Aspect-Oriented Platform. The computer journal, vol 48, nº 4, pp. 401-420. Mayo, 2005.

[Pin+02] Pinto, M., et al. Separation of Coordination in a Dynamic Aspect Oriented Framewrok. AOSD Conf. Enschede. Netherlands, pp. 134-140. Abril, 2002.

[Pin04] Pinto, M. CAM/DAOP: Modelo y Plataforma Bsados en Componentes y Aspectos. Tesis Doctoral. Universidad de Málaga. 2004.

[PrBaCa00] Prior, J., Bastán, N., Campo, M. A Reflective Approach to Support Aspect Oriented Programming in Java. Proceedings of First Argentine Symposium on Software Engineering (ASSE 2000) - Jornadas Argentinas de Informática e Investigación Operativa. Buenos Aires, Argentina. Septiembre, 2000.

[Pru+98] Pruitt, S., Stuart, D., Sull, W., Cook, T. W. The Merit of XML as an Architecture Description Language Meta-Languge. Technical Report of Microelectrocins and Computer Technology Corporation. 1998. Disponible también en http://www.mcc.com/projects/ssepp/papers/meritofxml.html

[Ras+03] Rashid, A., Moreira, A., Araujo, J. Modularization and Composition of Aspect Requirements. Proceeding of the 2nd AOSD Conference. Boston, USA, pp. 11-20. 2003.

[Rec91] Rechtin, E. Systems Architecting: Creating and Building Complex Systems. Prentice Hall, 1991.

[RoRo91] Royce, W. E., Royce, W. Software Architecture. Integrating Process and Technology, TRW Queso, vol 14, nº 1. 1991.

[SaChCh06] Sandé, M., Choren, R., Chavez. C. Mapping AspectualAcme into UML 2.0. 9th International Aspect Oriented Modeling. Genoa, Italia. Octubre, 2006.

[San04] Sanchez, M. S. COFRE: Un entorno formal para la especificación y desarrollo de sistemas de coordinación. Tesis doctoral. Universidad de Extremadura. 2004.

[San99] Sánchez, F. Modelo de disfraces. Hacia la adaptabilidad de restricciones de sincronización en los lenguajes concurrentes orientados a objeto. Tesis doctoral. Universidad de Extremadura. 1999.

[Sch01] Schmerl, B. xAcme:CMU Acme Extensions to xArch. 2001. http://www.cs.cmu.edu/-acme/pub/xAcme/guide.pdf

[Sch99] Schneider, J. Components, Scripts and Glue: A Conceptual Framework for Software Composition. Tesis Doctoral. Universidad de Berna. 1999.

[Sha+95] Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, pp. 314-335. Abril, 1995.

[Sha01] Shaw, M. The coming-of-age of Software Architecture research. Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001). 2001.

[Sha84] Shaw, M. Abstraction Techniques in Modern Programming Languages. IEEE Software, pp. 10-26. Octubre, 1984.

[Sha89] Shaw, M. Large Scale Systems Require Higher- Level Abstraction. Proceedings of Fifth International Workshop on Software Specification and Design, IEEE Computer Society, pp. 143-146. 1989.

[ShGa94] Shaw, M., Garlan, D. Characteristics of Higher-Level Languages for Software Architecture. Technical Report CMU-CS-94-210, Carnegie Mellon University. Diciembre, 1994.

[ShGa96] Shaw, M., Garlan, D., Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, Upper Saddle River, 1996.

[ShKa03] Shiman M., Katz S., Superimpositions and Aspect-Oriented Programming. The Computer Journal, Vol 46, nº 5, pp. 529-541. Septiembre, 2003.

[Smi82] Smith, B., Reflection and Semantics in Procedural Languages. Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology. 1982.

[Spe00] Spencer, J. http://www.opengroup.org/tech/architecture/adml/background.htm Technical Report. Septiembre, 2000.

[StHaUn02] Stein, S., Hanenberg, S., Unland, R. A UML-Based Aspect-Oriented Design Notation for AspectJ. Proceeding of the 1st AOSD Conference, Enschede, Holanda, pp. 106-112. 2002.

[StVo06] Stahl, T. Voelter, M. Model-Driven Software Development. Wiley. 2006.

[SuFrVa06] Suvee, D., De Fraine, B., Vanderperren, W. A Symmetric and Unified Approach towards Combining Aspect-Oriented and Component-Based Software Development. Proceedings of the 9th Int. SIGSOFT Symposium on Component-Based Software Engineering. CBSE'06, Estocolmo, Suecia. Junio, 2006.

[SuRo02] Sutton, S., Rouvellou, I., Modelling of Software Concerns in Cosmos. The 1st International Conference on AOSD, G. Kiczales, Ed., pp. 127-133. 2002.

[SuVaJo03] Survee, D., Vanderperren, W., Jonkers, V. JasCo: an Aspect-Oriented Approach Tailored for CBSD. Proceeding of the 2nd AOSD Conference, pp. 21-29, Boston, USA. 2003.

[Szy98] Szyperski C., Component software: beyond object-oriented programming. ACM Press and Addison Wesley, New York, USA. 1998.

[Tar+05] Tarr P., Ossher, H., Sutton, S. M., Harrison, W. N Degrees of Separation: Multi-dimensional Separation of Concerns. In Filman, R. et al. Eds. Aspect-Oriented Software Development, pp. 37-61. Addison Wesley. 2005.

[Tay+96] Taylor, R. N. et al. A Component and Message-Based Architecture Style for GUI Software. IEEE Transaction on Software Engineering. Junio, 1996.

[Tek04] Tekinerdogan, B. ASAAM: Aspectual Software Architecture Analysis Method. WICSA, pp. 4-14. Oslo, Noruega. Junio, 2004.

[Theme] Theme site: http://www.dsg.cs.tcd.ie/index.php?category_id=353.

[Tis+01] Tisato, F., Savigni, A., Cazzola, W., Sosio, A. Architectural Reflection- Realising Software Architectures via Reflective Activities. LNCS vol. 1999, pp. 102-115 Springer. ISBN: 3-540-41792-3. 2001.

[Val+01] Valentino, F., Ramos, A., Marcos, C., Prior, J. A Framework for the Development of Multi-Level Reflective Applications. Proceedings of Second Argentine Symposium on Software Engineering (ASSE 2001), Buenos Aires, Argentina. Septiembre, 2001.

[Val98] Valmari, A. The State Eplosion Problem. LNCS vol.1491, pp.429-528. Springer Verlag, 1998.

[VePeLu99] Vera, J., Perrochon, L., Luckhan, D. Event-based Execution Architectures for Dynamic Software Systems, en proceeding of WICSA. Kluwer Ac. Publ. pp. 303-318. 1999.

[WaFr88] Wand, M., Friedman, D. The Mystery of the Tower Revealed: A Non Reflective Description of the Reflective Tower. Lisp and Symbolic Computation. 1988.

[WaYo88] Watanabe, Yonezawa. Reflection in an Object-Oriented concurrent Lanaguage. En OOPSLA'88 Proceeding, pp. 305-315. Septiembre, 1988.

[WCAT04] 1st WS on Coordination and Adaptation Techniques for software entities (WCAT'04) in ECOOP'04. 2004.

[WeBe02] Wagelaar, D., Bergmans, L. M. Using a Concept-Based Approach to Aspect-Oriented Software Design. AOD workshop. 1st AOSD Conference. Enschede, Holanda. Abril, 2002.

[Wer99] Wermelinger, M. Specification of Software Architecture Reconfiguration. Tesis doctoral. Universidad Nova de Lisboa. Septiembre, 1999.

[Wil99] Wile D., AML: an Architecture Meta-Language. Automated Software Engineering ASE, IEEE Computer Society, pp. 183-190. 1999.

[Wol97] Wolf. A. Succeedings of the Second International Software Architecture Workshop (ISAW-2). ACM SIGSOFT Software Engineering Notes, pp. 42-56. Enero, 1997.

[xacme] Acme Extension to xArch: http://www-2.cs.cmu.edu/-acme/pub/xAcme/

[xarch] http://www.isr.uci.edu/projects/xarch/

[YAH] http://anna.fi.muni.cz/yahoda

[Zha02] Zhao, J. Change Impact Analysis for Aspect-Oriented Software Evolution Proceedings. 5th International Workshop on Principles of Software Evolution, pp. 108-112, ACM Press, Orlando. Mayo, 2002.

ANEXO 1
LEDA

LEDA es un Lenguaje de Descripción Arquitectónica desarrollado en la Universidad de Málaga por Carlos Canal, dotándole tanto de aspectos sintácticos como semánticos, e incluyendo las nociones formales necesarias para que el lenguaje facilite la descripción de arquitecturas software. LEDA es un lenguaje de especificación para la descripción y validación de propiedades estructurales y de comportamiento de sistemas software. En este anexo se describe someramente la especificación del lenguaje.

A1.1. Introducción

LEDA [CaPiTr01] es uno de los LDA más avanzados, y tal vez el que alcanza mayores cotas de dinamismo. Esto se debe a que la interacción entre los componentes se indica de manera formal mediante cálculo π. El hecho de seleccionar un álgebra de procesos tan potente permite obtener un lenguaje con enormes capacidades dinámicas y de flexibilidad. Al estar escrita la semántica de *LEDA* en términos de *cálculo* π, las especificaciones pueden ser tanto analizadas como ejecutadas, permitiendo esto último el prototipado de la arquitectura.

LEDA está articulado en dos niveles, uno para la definición de *componentes* y el otro para la definición de *roles*. Una arquitectura se expresa como un conjunto de componentes, cuyo interfaz se segmenta en roles. Los *componentes* representan partes o módulos del sistema y cada uno proporciona una determinada funcionalidad al mismo; pueden ser elementales o compuestos y pueden estar parametrizados. Los componentes se

conectan entre sí mediante conexiones. *LEDA* define tres tipos de conexiones: estáticas, reconfigurables y múltiples. Admite la extensión de componentes y roles, así como la definición de múltiples niveles de abstracción, vistos como un mecanismo de subtipado. Por su parte, los *roles* describen el comportamiento externo de los componentes; este nivel se utiliza para el prototipado, validación y ejecución de la arquitectura. Cada *rol* proporciona una visión parcial del comportamiento de un cierto componente. La estructura de cada rol se especifica como una descripción en cálculo π. La arquitectura de un componente se indica mediante las relaciones que se establecen entre sus subcomponentes, lo que se expresa mediante un conjunto de conexiones entre los roles de dichos subcomponentes.

LEDA, a nivel del leguaje, no hace distinción entre componentes y conectores. Es decir, los conectores se especifican como un tipo más de componentes, permitiendo que el lenguaje sea más simple y regular, dotándole de mayor flexibilidad en la descripción las arquitecturas. El lenguaje dispone, además, de mecanismos de herencia y parametrización de roles y componentes que aseguran la preservación de la compatibilidad. Mediante estos mecanismos, se puede reemplazar un componente en una arquitectura por otro que herede del primero, con los enlaces que se envían como argumentos en la comunicación entre procesos. Esto lo hace especialmente adecuado para describir la estructura de sistemas en los que los componentes se puedan crear y eliminar de forma dinámica, y en los que las conexiones entre componentes se puedan establecer y modificar también dinámicamente, consiguiendo que la comunicación del sistema evolucione en el tiempo según los requisitos que imponga la evolución de éste. Este es uno de los aspectos más notables del lenguaje, en particular por su tratamiento formal.

El disponer de un mecanismo de adaptación de un componente a una interfaz que no sea compatible con la suya propia fomenta la reutilización de los componentes. *LEDA* admite también la construcción de un componente como envolvente de otro ya existente, de modo que capture las interacciones de sus roles sin necesidad de redefinirlos como propios. Para esta función específica en *LEDA* se han diseñado los *adaptadores*. Estos son elementos similares a roles y descritos en *cálculo* π, capaces de interconectar con éxito componentes cuyo comportamiento no es compatible por tener especificaciones diferentes.

Los sistemas de topología dinámica se representan en cálculo π mediante colecciones de procesos o agentes que interactúan por medio de *enlaces* o nombres. La restricción del ámbito de un *enlace* permite la definición de *enlaces* privados a un grupo de agentes determinado. En cálculo π, los *enlaces* y datos son considerados genéricamente como nombres, lo que permite construir un cálculo muy simple, pero a la vez muy potente, debido a que lo que se envía a través de los nombres de enlaces son, a su vez, nombres de enlaces. Cuando un agente recibe un nombre, como objeto o argumento en una comunicación a través de un enlace, puede usar este nombre como sujeto para una transmisión futura, lo que redunda en sencillez y efectividad en la configuración del sistema. Estos nombres pueden considerarse como canales bidireccionales compartidos por dos o más agentes, que actúan como sujetos de la comunicación, en el sentido de que la comunicación se realiza a través de ellos.

A1.2. Sintaxis de LEDA

La especificación de componentes y roles se hace de forma modular lo que facilita su reutilización.

A1.2.1. Especificación de componentes

LEDA distingue entre clases e instancias de componentes y proporciona mecanismos para la extensión y la derivación de clases de componentes. Su sintaxis es la que se muestra en Figura A1.1.

```
component ClaseDeComponentes {
constant
listaDeConstantes;
var
variable: Tipo:=valorInicial;
...
interface
instanciaDeRol: ClaseDeRoles;
...
composition
instanciaDeComponente: ClaseDeComponentes;
...
attachments
listaDeConexiones;
}
```

Figura A1.1. Especificación de componente en *LEDA*.

Tras la palabra reservada *component* va el identificador de la clase componente que se está describiendo y, a continuación, entre llaves, el cuerpo de la especificación. Ésta se divide en cinco secciones: declaración de constantes, declaración de variables, descripción de la interfaz, composición y lista de conexiones.

- *Las declaraciones de constantes y variables* figuran, respectivamente, a continuación de las palabras reservadas *constant* y *var*. El resto de elementos de la especificación pueden hacer referencia a estas constantes y variables, que servirán para coordinar y comunicar unos elementos con otros, en aquellos casos en que sea necesario. En particular, los valores de las variables determinan el estado del componente. Para cada una de ellas se puede indicar un valor inicial, aquel que toma cuando se crea el componente.
- *La descripción de la interfaz del componente* viene indicada por la palabra reservada *interface*, seguida de una *lista de instancias de roles*, para cada uno de los cuales se indica su clase. Por tanto, la interfaz de un componente se describe por medio de un conjunto de *roles*. Se tendrá un *rol* por cada una de las interacciones que dicho componente establece con otros componentes del sistema. Cada *rol* es una abstracción que representa tanto el comportamiento que el componente ofrece a su entorno, como el que exige a aquellos componentes que están conectados a él.

- *La sección de composición* describe la estructura interna del componente como una lista de instancias de clases de componentes, precedida de la palabra reservada *composistion*. Esto permite la construcción de *componentes compuestos* a partir de otros más simples, en tantos niveles como sea necesario.
- En *la sección de conexiones* se utiliza la palabra reservada *attachments* que contiene una *lista de conexiones* entre los roles de componentes de la sección de composición. Estas conexiones sirven para indicar cómo se construye un compuesto a partir de sus componentes constituyentes. Al crear una instancia de un componente, se crean e inicializan sus constantes y variables locales. Así mismo, se crean las instancias de *roles* que forman su interfaz, los subcomponentes que los constituyen y, por último, las conexiones entre dichos subcomponentes. No todas las clases de componentes precisan que se detallen las cuatro secciones.

A1.2.2. Especificación de *rol*

La sintaxis de los *roles* puede especificarse de dos formas: inmersa, dentro del propio componente o ser una especificación independiente del *rol*, como una clase de roles.

A1.2.2.1. Inmersa

La declaración inmersa de una clase de *roles*, va dentro de la sección de interfaz del componente, a continuación del nombre de la instancia de *rol*, indicando cuáles son sus nombres libres o parámetros que servirán de canales de comunicación para su conexión con otros *roles;* seguido de la especificación de su comportamiento, por medio del cálculo π. Se puede realizar de tres formas:

- ***De forma anónima*** (Figura A1.2): no permite la reutilización, la extensión de la clase descrita en otros componentes y arquitecturas ni el uso de recursión en la especificación de un rol:

```
component ClaseDeComponentes {
interface
     instanciaDeRol(parámetros){
     especificación del rol;
}
```

Figura A1.2. Especificación de rol en *LEDA*. Forma anónima.

- ***Usando el identificador de clase*** (Figura A1.3): para hacer referencia a un *rol* definido en un componente de manera inmersa se indica el nombre de la clase de componentes seguido del nombre de la clase de *roles,* separados por un punto -*ClaseDeComponente.ClaseDeRoles*- e instanciando la clase de *rol*.

```
component ClaseDeComponentes {
interface
    instanciaDeRol: ClaseDeRoles(parámetros){
    especificación del rol;
}
```

Figura A1.3. Especificación de rol en *LEDA*. Con identificador de clase.

- ***De manera implícita:*** se usa igual que el anterior, pero haciendo la especificación implícita, usando las palabras reservadas *spec is implicit* (Figura A1.4).

```
component ClaseDeComponente {
interface
     instanciaDeRol(parámetros){
     spec is implicit;
}
composition
instanciaDeComponente: ClaseDeComponente;
attachments
instanciaDeComponente.instanciaDeRol >> instanciaDeRol
}
```

Figura A1.4. Especificación de rol en *LEDA*. Definición de manera implícita.

A1.2.2.2. Especificación independiente de un rol

La especificación de su comportamiento consiste en la definición de uno o varios *agentes* o procesos (Figura A1.5). Si se trata de un solo *agente* se define en *spec is***.** El *agente* que describe su comportamiento inicial figura en primer lugar tras las palabras reservadas *spec is*, mientras que la descripción del resto va precedida por la palabra reservada *agent* y seguida por *is*. La especificación del comportamiento del *agente* se realiza en cálculo π.

```
role ClaseDeRoles(parámetros){
constant
listaDeConstantes;
var
listaDeVariables;
spec is
especificación de agente;
agent NombreDeAgente is
especificación de agente;
...
}
```

Figura A1.5. Definición independiente de rol en *LEDA*.

A1.2.3. Arquitectura, composición y conexiones

Una clase del tipo componente en *LEDA* especifica la arquitectura de un grupo de componentes, ya sean simples, subsistemas o sistemas completos. En el caso de componentes simples y subsistemas se definen como instancias de dichas clases en la sección *composition* de otros componentes, para formar compuestos cada vez más complejos. Para utilizar o reutilizar una clase determinada sólo hay que instanciarla:

```
instance elSistema: Sistema
```

La arquitectura de un compuesto viene determinada por las relaciones que mantienen sus componentes entre sí. Las conexiones se establecen cuando se crean las correspondientes instancias de componentes y *roles*. Hay varios tipos de conexiones.

A1.2.3.1. Conexiones estáticas

Son aquellas que no se modifican una vez que se establecen. Sirven para describir sistemas que tienen una arquitectura estática:

```
comp.rol(nombres) <> comp'.rol'(nombres');
```

Es posible conectar más de dos roles en la misma conexión:

```
comp.rol(p1, p2) <> comp'.rol'(p1, p3) <> comp''.rol''(p2,p3);
```

A1.2.3.2. Conexiones reconfigurables

Se utilizan en arquitecturas que presentan varias configuraciones. Es decir, en aquellas en las que los patrones de interconexión entre componentes varían en el tiempo y los *roles* conectados dependen de ciertas condiciones. En una conexión reconfigurable algunos de sus términos son una expresión condicional. Como se puede apreciar en Figura A1.6a, en la definición de esta conexión se indica que el rol *rol* del componente *comp* se conecta a *rol'* o a *rol''* (de *comp'* y *comp''* respectivamente) dependiendo del valor que tenga *condición.* La condición se comprueba mediante las sentencias condicionales *if* o *case* (Figura A1.6b), según el número de valores posibles de la condición a comprobar. En este caso, si las condiciones de las alternativas no son excluyentes, se selecciona una de las que tome el valor lógico *True* de forma no determinista:

```
comp.rol(nombres) <> if condición
then comp'.rol'(nombres')
else comp''.rol''(nombres'');
```

Figura A1.6a). Definición de conexiones reconfigurables en *LEDA*. Con if.

```
comp.rol(nombres) <> case condición' then comp'.rol'(nombres')
case condición'' then comp''.rol''(nombres'')
…….
default compn.roln(nombresn)
```

Figura A1.6b). Definición de conexiones reconfigurables en *LEDA*. Con Case.

A1.2.3.3. Conexiones múltiples

Las conexiones múltiples describen patrones de comunicación entre conjuntos de componentes. Cada par de componentes interconectados puede usar enlaces privados en su comunicación o bien estos enlaces pueden estar compartidos entre todos los componentes implicados. Por tanto, las conexiones múltiples pueden ser compartidas o privadas.

- **Privadas**: la sintaxis de una conexión múltiple que utilice nombres privados es la misma que la de una conexión estática, exceptuando que en cada término, en vez de una única instancia de componente y *rol*, figura al menos una colección.

```
comp[ ].rol(nombres) <> comp'.rol'[ ](nombres');
```

Esta expresión indica que el rol de cada componente de la colección de componentes (el vector `comp[]`) se conecta a cada uno de los *roles* (`rol'[]`). Las conexiones se establecen de forma dinámica, según se crean los correspondientes componentes y *roles* que figuran en cada término.

- **Compartidas**: el uso de un asterisco en alguna de las colecciones que figuren en una conexión múltiple indica que todos sus *roles* utilizan nombres compartidos a la hora de conectarse con el *rol* o *roles* que figuren en el otro término.

```
comp[ * ].rol(nombres) <> comp'.rol'[ ](nombres');
```

Esta expresión indica que los roles *rol* de todos los componentes de la colección `comp` se conectan al mismo *rol* `rol'` del componente `comp'`.

Por tanto, el uso de una conexión múltiple compartida implica el uso de una serie de canales de comunicación 1:M, uno por cada nombre libre de los *roles*, y la realización de una difusión cada vez que se realice una acción de salida, desde el extremo simple del canal hacia el extremo múltiple. Por el contrario, una conexión privada establece múltiples canales de comunicación 1:1, uno por cada par de los conectados y por cada nombre libre de dichos *roles*.

A1.2.3.4. Exportación de roles

Una forma adicional de conexión es la exportación de *roles*. Al especificar un compuesto, no todos los *roles* de sus componentes se utilizan para interconectarlos unos con otros, sino que alguno de ellos pasa a formar parte de la interfaz de dicho componente compuesto. En ese caso, se dice que estos *roles* son exportados por el componente compuesto, lo que se indica en *LEDA* por medio del operador >> en lugar de <>. En el

primer término de la conexión de exportación pueden aparecer varias instancias de *roles*, separadas por comas, en el segundo término sólo puede aparecer un *rol*:

```
comp'.rol'(nombre'), comp''.rol''(nombre)>> rol(nombres);
```

A1.3. Extensión de roles y componentes. Herencia

Para facilitar la reutilización efectiva de componentes, *roles* y arquitecturas es necesario un mecanismo de redefinición y extensión de *roles* y componentes. En *LEDA* este mecanismo se basa en las relaciones de herencia y extensión de *roles*.

A1.3.1. Extensión de roles

La extensión de *roles* puede utilizarse para redefinir parcial o totalmente el *rol* antecesor, proporcionando una nueva especificación para alguno de sus *agentes*; o simplemente para extender un *rol*, dotándolo de funcionalidad adicional. En ambos casos, es necesario comprobar que el *rol* extendido es, efectivamente, descendiente del *rol* progenitor. Esto se hace durante el análisis de la especificación, y es lo que permite reemplazar componentes y el refinamiento de arquitecturas. Su formato se representa en Figura A1.7.

```
role ClaseDerivada(parámetros) extends ClaseProgenitora {
| spec is NuevaEspecificación;
| redefining AgenteProgenitor as NuevaEspecificacion;
| redefining AgenteProgenitor adding NuevaEspecificación;
…
}
```

Figura A1.7. Extensión de roles en *LEDA*.

La herencia y extensión de *roles* se expresa en *LEDA* mediante la palabra reservada *extends*. Dependiendo del carácter de esta derivación, la especificación del nuevo *rol* consiste en la descripción completa del mismo (utilizando para ello una cláusula *spec is*) o bien en la redefinición de algunos de los *agentes* del progenitor. A su vez, la redefinición de un *agente* puede ser completa (lo que se indica mediante la cláusula *redefining as*) o puede consistir únicamente en la adición de un comportamiento alternativo al ya descrito por el agente (indicado mediante la cláusula *redefining adding*).

A1.3.2. Extensión de componentes

Los componentes derivados heredan la especificación de sus progenitores, incluyendo *roles*, subcomponentes y conexiones. Un componente derivado extiende a su progenitor, al añadir nuevos *roles* componentes o conexiones, o al redefinir algunos de

los especificados por su progenitor. En caso de redefinición de un componente, la instancia redefinida debe ser, a su vez, heredera de la original. También se expresa mediante la cláusula *extend* (Figura A1.8):

```
Component ClaseDerivada extends ClaseProgenitora {
…
}
```

Figura A1.8. Definición de componente derivado en *LEDA*.

Al especificar el cuerpo del componente se indica cuales son los elementos que se añaden o redefinen respecto de la clase padre, de forma que cualquier elemento de la clase padre al que no se haga referencia en la derivada, se hereda en esta última.

A1.4. Refinamiento de Arquitecturas

Las descripciones arquitectónicas pueden utilizarse con diferentes niveles de abstracción durante el proceso de desarrollo. Esto es lo que se denomina *refinamiento*. En *LEDA* el refinamiento se realiza mediante la instanciación de arquitecturas, en la que es posible indicar la sustitución de una instancia de componente en una arquitectura por otra instancia cuya clase extienda la de la primera. La sintaxis se muestra en Figura A1.9 en la que *componenteDerivado* es una instancia de *ClaseComponente* en la que se ha reemplazado su *subcomponente.*

Al refinar una arquitectura, instanciando alguno de sus componentes, se modifican algunas de las conexiones que figuran en dicha arquitectura, debido a que los componentes originales son reemplazados por versiones derivadas de los mismos:

```
componenteDerivado: ClaseComponente [
subcomponente:ClaseDerivadaSubcomponente;
];
```

Figura A1.9. Refinamiento de arquitecturas en *LEDA*.

A1.5. Adaptadores

A veces el comportamiento de dos componentes no es compatible, pero pueden adaptarse de forma que sean capaces de colaborar entre sí. En *LEDA* esto se lleva a cabo haciendo uso de un *adaptador*, que actúa como mediador entre ambos, permitiendo la construcción de compuestos a partir de componentes que no son estrictamente compatibles. Los *adaptadores* pueden utilizarse también para modificar la interfaz que un determinado componente exporta a su entorno. Los *adaptadores* se representan utilizando la misma sintaxis que los roles, aunque éstos describen la interfaz de un componente,

mientras que los *adaptadores* se utilizan como nexo de unión entre los componentes de un compuesto y se declaran en la sección *composition*, junto al resto de integrantes de un compuesto.

A1.6. Implementación *LEDA*

Canal [CaPiTr01] desarrolló un compilador para *LEDA* para generar código Java a partir de las especificaciones arquitectónicas en este lenguaje. Dicho compilador acepta archivos *LEDA* (con extensión *.leda*), compila cada uno bajo Java e invoca a *javac* sobre los ficheros *.java* resultantes. La técnica del compilador es modular, esto es, cuando un archivo fuente se modifica o actualizada, solamente ese fichero y los ficheros que dependan de él se compilarán.

La generación de código en *LEDA* se basa en las llamadas propuestas generativas, de modo que las partes de código no descritas durante la especificación y la arquitectura del sistema se completan durante la fase de implementación.

La arquitectura definida en *LEDA* debe ser validada por medio del análisis de compatibilidad de cada una de sus conexiones. Se verifican también las relaciones de herencia y de extensión de los *roles* y componentes. Así la arquitectura resulta validada. La traducción a Java como lenguaje de implementación mantiene las propiedades de la arquitectura.

En *LEDA* el código que se genera produce un prototipo ejecutable del sistema, en el que la implementación de los métodos de los componentes inicialmente no tiene por qué estar definida todavía. La ejecución del prototipo permite al diseñador conocer si el comportamiento del sistema es el esperado, detectando errores anticipadamente.

El compilador de *LEDA* traduce cada clase componente a una clase Java que hereda de *Component*, que es una de las clases básicas definidas para *LEDA*, dejando los cuerpos de los métodos vacíos. Por su parte, una clase de *rol* de una arquitectura en *LEDA* da lugar a una clase Java que hereda de la clase básica *Role* (clase definida para implementar los enlaces necesarios en la arquitectura especificada).

En *LEDA* se pueden especificar sistemas formados por un número indeterminado de componentes iguales (instrucción *new*). De este modo, se tiene una creación dinámica de componentes.

Sin embargo, la versión inicial de este compilador obligaba al arquitecto software a trabajar en modo comando. Dentro del marco de trabajo que se ha creado durante el desarrollo de esta tesis doctoral y dado que se pretendía definir un LDA-OA que extendiera *LEDA*, se consideró oportuno definir una interfaz gráfica para la definición de arquitecturas *LEDA* y llevar a cabo todo el proceso de compilación y ejecución dentro del mismo entorno. Así surgió *EVADeS*, herramienta que se explica en Anexo 2.

A1.7. Conclusiones

Como conclusión se puede decir que *LEDA* es un LDA que permite especificar arquitecturas software a través de un conjunto de componentes y *roles*. Los componentes se comunican a través de los *roles* que definen su comportamiento externo y cuya semántica se expresa en cálculo π.

Finalmente, *LEDA* es un LDA que permite analizar y validar las arquitecturas, así como la ejecución de un prototipo tras la generación de código Java.

ANEXO 2
EVADeS

El Entorno Visual de Asistencia al Desarrollo de Sistemas *para LEDA –EVADeS- es una herramienta que permite obtener, de una manera sencilla, prototipos ejecutables de sistemas especificados en LEDA. La aplicación se apoya en un generador de código Java a partir del procesamiento de ficheros que contienen código LEDA. La herramienta sirve para crear ficheros LEDA, interpretarlos, modificar el código generado y para la obtención, modificación, compilación y ejecución de prototipos obtenidos a partir de la descripción arquitectónica en el LDA. En este anexo se describe la herramienta.*

A2.1. Justificación

El objeto de este anexo es, a partir de un generador de código Java creado por [Can00] y López [Lop02], que trabaja en línea de comando, y considerando las especificaciones arquitectónicas en código *LEDA* [CaPiTr01], crear una interfaz gráfica que facilite esta tarea al usuario. Para ello, se ha creado una aplicación interactiva que ayude al arquitecto de software a diseñar sistemas expresando, de una manera cómoda, la especificación de su arquitectura en *LEDA* y que posteriormente se pueda realizar la generación de código Java. Por otra parte, como *LEDA* sigue una definición evolutiva, a medida que se va refinando la especificación arquitectónica, se pueden obtener prototipos del sistema mejorados sucesivamente.

El generador de código inicial (y el propuesto en *EVADeS*) intenta proporcionar una guía a lo largo del proceso de desarrollo que permita garantizar la generación de una codificación fiel a la especificación original, que sirva de base para el prototipado. Esto

dota de gran potencia a *LEDA* ya que, los lenguajes de descripción arquitectónica, en general, no suelen llegar a una implementación que conserve las propiedades de la especificación. Normalmente el paso entre la especificación y la implementación sigue siendo responsabilidad del diseñador, lo que conlleva un esfuerzo considerable sin garantía de éxito, al no estar asegurado que el sistema resultante se comporte como la especificación original; aunque algunos LDA permiten crear una arquitectura libre de errores.

EVADeS integra en un entorno amigable todos aquellos programas que es necesario utilizar para crear especificaciones arquitectónicas de sistemas en *LEDA* y llegar a obtener su prototipo ejecutable en Java. Por tanto, permite acometer el proceso de manera sencilla, sin tener que realizar los pasos uno a uno, ni cambiando de herramienta y de entorno cada vez, como ocurría en el proceso original, haciéndolo lento, tedioso y aburrido.

Como parte del proceso de desarrollo de la herramienta, la gramática de *LEDA* fue modificada en algunos puntos para mejorar la versión inicial –se usaron los programas Pclex y Pcyacc-, y poder ejecutar *EVADeS* convenientemente. Así mismo, se han usado compiladores de C y C++ para la obtención del programa ejecutable que genera el código en Java (necesario para la obtención del prototipo del sistema). La interfaz de usuario se realizó en Borland C++ Builder 5.

En este apéndice se describen las características de la herramienta que permite obtener un prototipo ejecutable desde la fase de diseño arquitectónico. Una descripción más detallada se puede encontrar en [GaNa08].

A2.2. Generador de Código *LEDA*

El desarrollo de esta herramienta está basado en lo que se denomina generación de código. Este concepto es un mecanismo por el cual se puede traducir un lenguaje o notación de más alto nivel a un lenguaje de implementación, y se asume que los desarrolladores trabajan tanto a nivel de especificación como de implementación. Para la especificación se usa un lenguaje de alto nivel, a partir del cual se generará código, y aquellos aspectos no definidos en la especificación se completan durante la fase de implementación. Este enfoque facilita la búsqueda de la arquitectura más adecuada para un cierto sistema, para posteriormente completarla con detalles de más bajo nivel.

A2.2.1. Antecedentes

El desarrollo de *EVADeS* se basa en la propuesta generativa de *LEDA* como lenguaje de especificación de arquitecturas y su paso al lenguaje de implementación Java. En la propuesta de Canal, a partir del código obtenido, se obtiene un prototipo ejecutable; la propuesta empieza con la descripción de los componentes, los roles que representan la interfaz de dichos componentes y la arquitectura de un sistema, descrita mediante las conexiones que relacionan los roles de unos componentes y otros. Una vez realizada la

especificación del sistema se procede a la validación de la arquitectura, por medio de un análisis de la compatibilidad de cada una de sus conexiones. También se realiza en este momento la verificación de las relaciones de herencia y extensión de roles y componentes. De esta forma se valida la arquitectura del sistema, lo que permite comprobar que la especificación satisface determinadas propiedades. A continuación, la especificación se traduce al lenguaje de implementación, conservando sus propiedades, en particular la compatibilidad entre componentes demostrada durante el análisis del sistema. La implementación resultante debe contener código correspondiente a los protocolos de interacción seguidos en las conexiones entre los componentes. Como una especificación en *LEDA* consta de tres elementos fundamentales: componentes, roles y conexiones entre roles; la traducción se articula en torno a cinco clases Java (*clases básicas*): componente (*component*), rol (*role*), enlaces (*link*), acciones (*action*) y estados (*state*).

El esquema de generación de código que se llevó a cabo inicialmente [Lop02] tenía en cuenta los siguientes objetivos:

- El código generado deberá ser un prototipo ejecutable de la aplicación final.
- Debía fomentarse el desarrollo evolutivo e incremental. Inicialmente sólo es necesaria la especificación de los roles de los componentes, dejándose para más tarde la descripción e implementación de las computaciones.
- La implementación se basa en los patrones de interacción descritos en la especificación, debiendo asimismo conservar todas las propiedades demostradas al respecto durante la fase de verificación.
- Se debía permitir la manipulación directa por parte del diseñador del código generado.
- El mecanismo de comunicación utilizado para interconectar componentes debía permanecer oculto al diseñador.

El resultado obtenido tras la generación de código de la herramienta inicial era un prototipo que servía como base para la implementación final del sistema, siguiendo un proceso evolutivo e incremental que permite probar el sistema según va siendo construido. Se consigue así hacer evolucionar el prototipo hasta convertirlo en el sistema final, y que los encargados del desarrollo únicamente tengan que agregar aquello referente a la computación sobre los datos; es decir, a la codificación de los métodos, de modo que los componentes desempeñen la funcionalidad deseada. La implementación de las computaciones, según Canal, se puede hacer de dos formas:

> *manipulando el código generado, para añadir los métodos de cada componente asociados a la lectura y escritura de enlaces o bien refinando la especificación original en LEDA, de forma que se complete con aquellos aspectos computacionales relevantes para el sistema y para los cuales se podrá generar código de forma automática.*

Finalmente, para llegar al prototipo ejecutable de un sistema utilizando el generador de código inicial hay que realizar una serie de pasos. Esto supone llevar a cabo un proceso en el que no se utiliza una única aplicación, sino que hay que ejecutar varios programas independientes y en distintos entornos. Estos inconvenientes han llevado al diseño e implementación de *EVADeS* como una herramienta capaz de generar un

prototipo ejecutable de un sistema integrando todos los pasos en una única aplicación, en un entorno gráfico sin tener que recurrir al trabajo en línea de comando.

A2.2.2. Requisitos y objetivos de *EVADeS*

EVADeS se ha desarrollado a partir del generador de código inicial [Lop02], procurando la definición de un entorno más amigable que el original, que trabajaba en modo consola y manipulado cada una de las herramientas necesarias desde diferentes entornos. El objetivo final que se debe cumplir es

> *el diseño de una herramienta que permita cargar un fichero LEDA, procesarlo y crear un prototipo ejecutable*.

Para lograr este objetivo principal se han identificado otros objetivos que debería cumplir (y cumple):

- Realizar un entorno de trabajo sencillo que interactúe con el usuario.
- Crear una aplicación integrada que permita la generación de prototipos ejecutables a partir de la especificación de sistemas en *LEDA*.
- Esta aplicación debe ser capaz de editar los distintos ficheros que contienen el código fuente que se utilizará en las distintas utilidades que tendrá la herramienta.
- La herramienta debe permitir tanto la creación de un programa generador de código Java propio como manejar el prototipo que se cree.
- Además, debe detectar los errores que se puedan producir, tanto en el código del fichero fuente como en el código generado por la aplicación.
- La herramienta debe ser capaz de compilar prototipos de sistemas cuyo código fuente sea Java y detectar los errores que pueda haber.
- Finalmente, se debe poder ejecutar el prototipo que se genere.

Para alcanzar estos objetivos se han identificado una serie de requisitos que *EVADeS* tiene que cumplir:

- Editar, cargar y procesar ficheros con la extensión *LEDA*. Controlar los errores de procesamiento así como detectar errores léxicos, sintácticos y semánticos, y mostrarlos por pantalla.
- La herramienta debe ser capaz de crear la aplicación que genera el prototipo: editar y modificar los ficheros que contienen el analizador léxico y sintáctico, detectar y controlar los posibles errores, y mostrarlos al usuario.
- Detectar y controlar errores de compilación de C y mostrarlos al usuario.
- Editar, modificar y ampliar el prototipo Java generado, para, de este modo, hacerlo evolucionar incluyendo el código de los métodos correspondientes a las computaciones. Se requiere también que la herramienta facilite al usuario el acceso a las clases básicas que se usan como base para generar el prototipo, para modificarlas, compilarlas y en su caso, mostrar los errores que se puedan producir.
- Compilación y procesamiento del prototipo. Control de los errores de compilación Java, mostrándolos por pantalla.
- Todo el proceso debe realizarse desde el mismo entorno de trabajo.

Finalmente, se supone que el usuario de esta herramienta es un ingeniero de software con conocimientos de programación.

A2.3. Diseño de la herramienta

A partir de los requisitos y los objetivos propuestos para la herramienta, la interfaz gráfica se ha configurado en tres aplicaciones integradas en un mismo entorno de trabajo y organizadas del siguiente modo:

- Una aplicación que constituye el módulo principal y gestiona todo lo relativo a los archivos *LEDA*.

- Una aplicación que gestiona el generador de código.

- Una aplicación que gestiona los archivos relativos al prototipo generado.

A) Aplicación para la gestión de archivos LEDA

Se encarga de realizar la traducción del fichero fuente *LEDA* al prototipo ejecutable Java del sistema especificado. Para realizar su funcionalidad se han considerado dos procesos:

Cargar Archivo Leda: esta función permite al usuario cargar/abrir/editar un archivo *LEDA* y presentarlo en pantalla. A continuación, se puede realizar el segundo proceso.

Generar Prototipo Java de la arquitectura anterior. Para ello, la herramienta procesa sus cadenas de caracteres, controlando los errores para la generación del prototipo.

B) Aplicación para la gestión del generador de código

La aplicación denominada *Generador de código* permite construir/modificar el generador de código que se va a usar para generar código Java (entrada del segundo proceso de la primera aplicación). Este generador es un programa ejecutable que la herramienta proporciona por defecto, pero que el usuario puede modificar. Para realizar su función esta aplicación utiliza tres funciones:

Cargar Ficheros Lex y Yac que definen la gramática LEDA para que el usuario pueda modificarlos. Si se modifican, se debe realizar la tarea siguiente.

Procesar Archivos Lex y Yac. Procesa estos archivos con pclex y pcyacc y genera los analizadores léxico y sintáctico sobre archivos con código C.

Crear Herramienta Generadora. Este proceso se encarga de realizar la compilación de los archivos C, por el compilador de C Borland.

C) Aplicación para la gestión del prototipo

Esta aplicación realiza la compilación y ejecución del prototipo y maneja las clases Java que se generan o aquellas que son la base del prototipo (*clases básicas*). Esta aplicación permite cargar los archivos Java, compilarlos y ejecutar el prototipo, utilizando JDK. Si se produjeran errores, los detecta y presenta en pantalla.

Como conclusión de este apartado se puede decir que la herramienta se ha diseñado para interactuar con el usuario, informándole de los pasos que puede realizar, todo dentro del mimo entorno de trabajo. Además, dispone de un conjunto de opciones que le ayudan durante el proceso. En TR 30/2008 se describen los problemas que surgieron al desarrollar la herramienta, así como se describen también cada una de las funciones y archivos que se requieren para realizar cada tarea. Se omiten aquí en orden a dar mayor claridad a la explicación.

A2.4. Construcción de la herramienta

En este apartado se describen las características generales del diseño interno y del diseño externo.

A2.4.1. Diseño Interno

El diseño interno se refiere a cómo se ha estructurado y cómo se ha aplicado la especificación de la herramienta a su implementación. Se han identificado 3 tareas principales, cada una con sus propias características que han dado lugar a las 3 aplicaciones de EVADeS que permiten, finalmente, obtener un prototipo ejecutable:

1) *Tareas para gestionar los archivos LEDA*: el usuario introduce en el sistema un programa (especificación arquitectónica) en *LEDA* que representa un sistema cuyo prototipo se desea obtener. Este código *LEDA* puede estar almacenado como un fichero con extensión *.leda*. Un proceso se encarga de realizar la generación de código java y del prototipo (y de almacenarlo), así como de detectar los posibles errores de procesamiento de la arquitectura.
2) *Tareas para gestionar los archivos lex y yac*: se realizan cuando el usuario quiere cambiar el programa generador de código. Utiliza los archivos analizadores sintáctico y semántico.
3) *Tareas para gestionar el prototipo*: las lanza el usuario cuando quiere modificar el prototipo que se ha generado en el primer proceso (gestionar archivos LEDA).

Estas tareas dan lugar a tres procesos principales y dan forma al diseño externo.

A2.4.2. Diseño Externo

A partir de las características descritas en el apartado anterior, el diseño externo de *EVADeS* se realizó con las siguientes características relativas a la interfaz gráfica que está formada por las siguientes ventanas:

1) *Pantalla Principal* desde la que se pueden cargar ficheros *LEDA*. Desde ella se tiene acceso a las otras dos pantallas principales asociadas a las aplicaciones de *Generación de código* y *Modificación de Prototipo*. Dispone de una ventana para mostrar los errores de *LEDA* que pueda haber y los mensajes que el sistema ofrece.
2) *Pantalla Gestionar Generador de código*. Tiene dos vistas, una para el analizador léxico y otra para el analizador sintáctico. A través de esta pantalla se genera como salida un ejecutable que servirá para generar el prototipo del sistema en *LEDA*.
3) *Pantalla Gestionar Prototipo*. A través de esta pantalla se pueden editar los ficheros Java generados para crear el prototipo y las clases básicas en las que se apoya. La herramienta guía e informa al usuario de los pasos que se dan, de los posibles errores Java y la salida de la ejecución del prototipo.

Figura A2.1 muestra la jerarquía de menús asociada a la herramienta.

A2.4.3. Requisitos de instalación

Para poder ejecutar la herramienta tienen que estar convenientemente instalados los siguientes programas:

- jdksdk 21.4.2_05 (u otra versión de jdk de Java).
- Borland C++ 5.0 (u otra versión). Se ha utilizado Borland C++ Builder, con la opción de compilar en línea de comando mediante *bcc32.exe.*
- Pclex y Pcyacc de Abraxas Software.
- La aplicación inicial del traductor LEDA-JAVA.

Esta aplicación se ha realizado bajo el sistema operativo Windows XP y son necesarios un mínimo de 25 MB libres para la instalación; la resolución de la pantalla debe ser 1280x800. Para instalar la herramienta *EVADeS* hay que copiar los archivos que la constituyen (agrupados en una carpeta), respetar la estructura de directorios y realizar unos cambios en el *path* del sistema operativo. La carpeta debe ir en la unidad C:\. Para modificar el *path* del sistema, hay que añadir las líneas:

```
PATH=%PATH%C:\LEDA\bin;C:\LEDA\clases;
SET CLASSPATH=%CLASSPATH%;.;C:\LEDA\bin;C:\LEDA\clases;
```

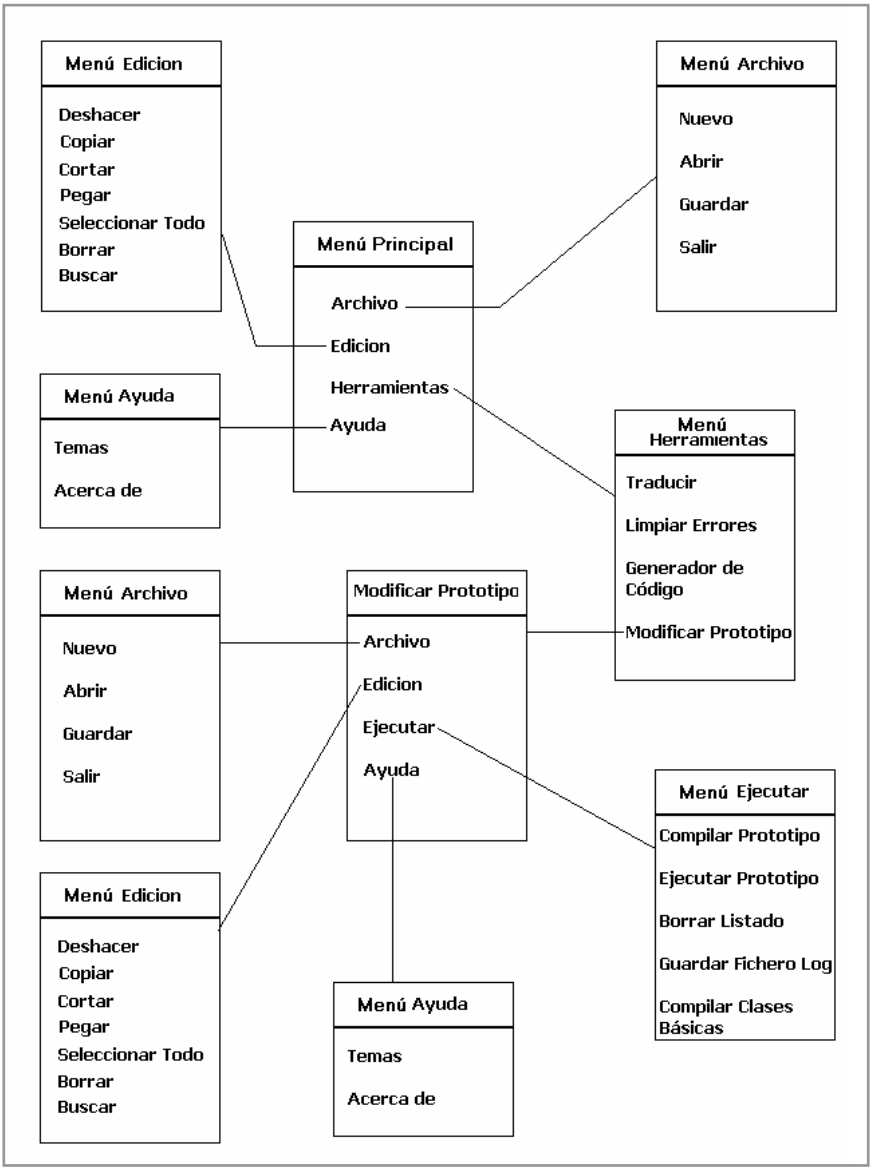


Figura A2.1. Jerarquía de menús.

A2.5. Utilización de la herramienta

En esta sección se describe el funcionamiento de *EVADeS,* que permite crear ficheros *LEDA* y que sean procesados utilizando las aplicaciones que forman la herramienta. Es decir, permite realizar el análisis léxico y sintáctico y, si no hay errores, generar los archivos en Java necesarios para obtener un primer prototipo ejecutable del sistema especificado en *LEDA*. La herramienta también permite crear un generador de

código propio capaz de procesar ficheros *LEDA*. La modificación del prototipo que se genera es otra de las aportaciones de *EVADeS*.

Esta herramienta se ha realizado pensando en la interacción con el usuario, informándole en cada momento de los pasos a realizar y de cómo se usan las distintas aplicaciones que la forman. Con la herramienta se pretenden abordar todas las tareas que antes se realizaban a través de la consola, en un entorno más amigable como es Windows, pero sin que la aplicación inicial deje se ejecute por debajo.

A2.5.1. Ventana principal: *Traductor*

Esta ventana es la primera que aparece cuando se entra en la herramienta, siendo la aplicación *Traductor* la primera que se ejecuta. La ventana tiene cuatro partes: barra de menús, barra de herramientas, ventana principal de edición y ventana o lista de mensajes (Figura A2.2).

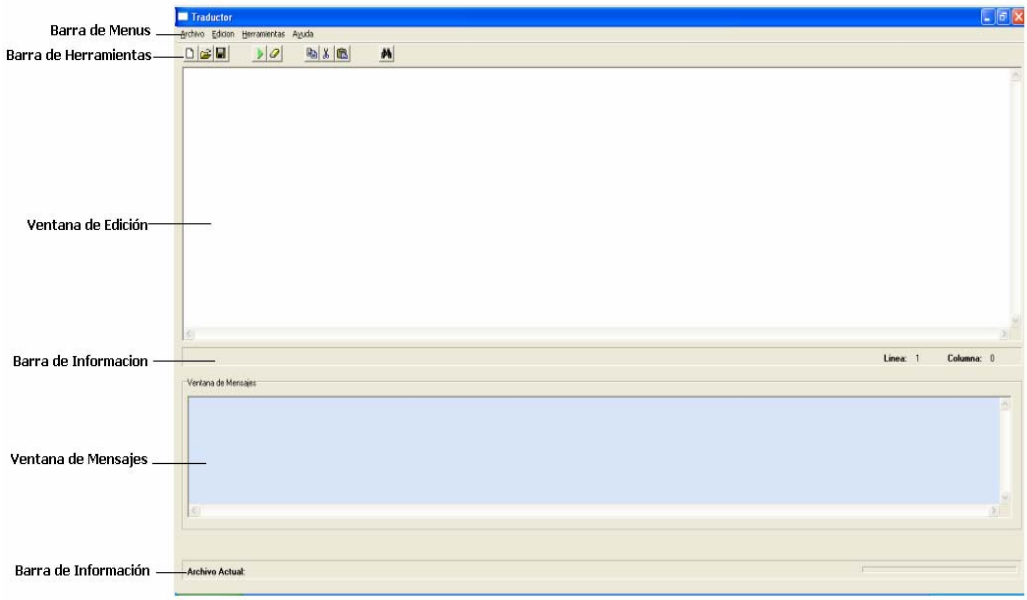


Figura A2.2. Pantalla principal de la aplicación.

A2.5.1.1. Barra de menús y barra de herramientas

La barra de menús contiene los menús *Archivo, Edición, Herramientas* y *Ayuda*.

Menú Archivo: tiene las opciones usuales: *Nuevo, Abrir, Guardar* y *Salir*.
Menú Edición: Tiene opciones usuales de edición de texto: *Deshacer, Cortar, Pegar, Copiar, Seleccionar todo, Borrar* y *Buscar*.
Menú Herramientas: contiene las opciones relativas a la funcionalidad de la herramienta:

Opción *Traducir*

Esta opción toma el fichero *LEDA* que está en uso (en la ventana de edición) y genera el prototipo Java de la arquitectura en *LEDA*. Tras la ejecución de esta opción, se crean una serie de archivos Java, que pueden ser compilados para obtener un prototipo del sistema. Además, se crea un archivo de texto que contiene el nombre de todos los archivos creados para su prototipo; su existencia significa que el archivo *LEDA* actual ya ha sido traducido a Java y por tanto se puede modificar el prototipo (editar estos ficheros Java) e introducir los métodos y variables necesarias para la ejecución del prototipo.

Cuando se genera el prototipo o se realiza la traducción a Java la aplicación muestra un mensaje (Figura A2.3) indicándolo, y los archivos que se han generado. Este mensaje se presenta en una ventana que invita al usuario a elegir entre las opciones de *ir a modificar el prototipo* en ese momento o *dejarlo para más tarde*. Si se elige la opción *ir a modificar el prototipo*, se abre la ventana del prototipo (apartado A2.5.3) para editar los ficheros Java, modificarlos, compilarlos y ejecutar el prototipo. Si se decide *dejarlo para más tarde* se puede acceder a esta opción en este menú herramientas en la opción Modificar Prototipo.

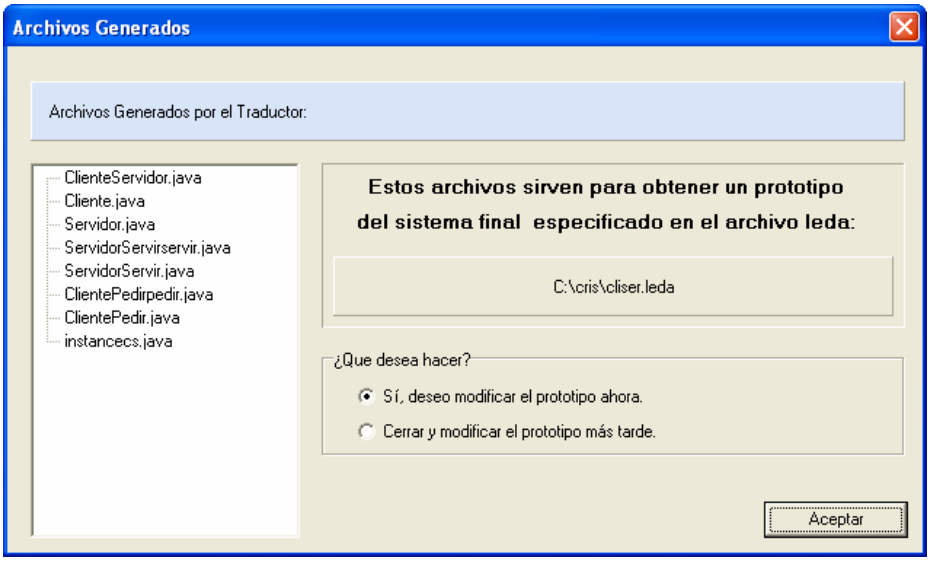


Figura A2.3. Ventana de información de prototipo.

Opción *Limpiar errores*

Esta opción borra el contenido de la ventana de mensajes. Por ejemplo, elimina los errores de un archivo que se hubiera cargado anteriormente.

Opción *Generador de código*

Esta opción llama a la segunda aplicación; se explica en el apartado A2.5.2.

Opción *Modificar prototipo*

Esta opción abre la ventana correspondiente a la tercera aplicación para generar el prototipo cuyo archivo *LEDA* esté actualmente cargado. Si el prototipo no se ha creado, hay que generarlo antes. En el caso de que sí exista, se va a la aplicación *Generar prototipo* (apartado A2.5.3)

La *barra de herramientas* contiene varios botones de acceso rápido a las opciones principales de estos menús. Si se pasa por encima de estos botones con el ratón aparece una etiqueta que indica su utilidad. Todas las opciones del menú tienen métodos abreviados de acceso, ya sean por teclas de función o por la utilización de las teclas ctrl+letra o alt+letra subrayada de la opción (parte alta de Figura A2.2).

A2.5.1.2. Ventanas de edición y de mensajes

Ventana Edición: es el espacio de trabajo en el que se escribe el código *LEDA*. Tiene las características generales de un editor de texto (Figura A2.2), de modo que la edición y compilación de los ficheros se realiza de manera más sencilla, sin tener que trabajar desde la consola.

Ventana de mensajes: es una ventana en la que se muestran los mensajes que genera la aplicación *Traductor* (parte inferior de Figura A2.2). Por ejemplo, si hay un error sintáctico en el archivo *LEDA*, éste se muestra por pantalla y se detiene la creación del prototipo. Por el contrario, si no ha habido problemas, se muestra en pantalla una lista de mensajes que indican cómo transcurre la creación del prototipo. Estos mensajes pueden ser modificados por el usuario mediante la aplicación *Generador de código*

A2.5.2. Ventana *Generador de código*

A esta ventana se llega seleccionando la opción *Generador de código* del menú *herramientas* de la ventana del *Traductor*. Desde ella se puede modificar el programa ejecutable que trabaja por debajo de la aplicación en la opción *Traducir* del menú *herramientas*. Esto quiere decir que *EVADeS* puede crear/modificar el programa ejecutable que se usa en la compilación de archivos *LEDA*. Con la herramienta se facilita la versión original de este ejecutable. Figura A2.4 muestra el aspecto que tiene la ventana del *Generador de código*.

Esta aplicación ejecuta dos funciones, que en la ventana se representan como dos pestañas, una se refiere al analizador léxico y la otra al analizador sintáctico.

A2.5.2.1. Analizador Léxico

Esta aplicación permite cargar y editar el fichero asociado al programa que genera código Java a partir de código *LEDA*. En Figura A2.4 se muestra la ventana correspondiente a esta función, que está dividida en cinco partes:

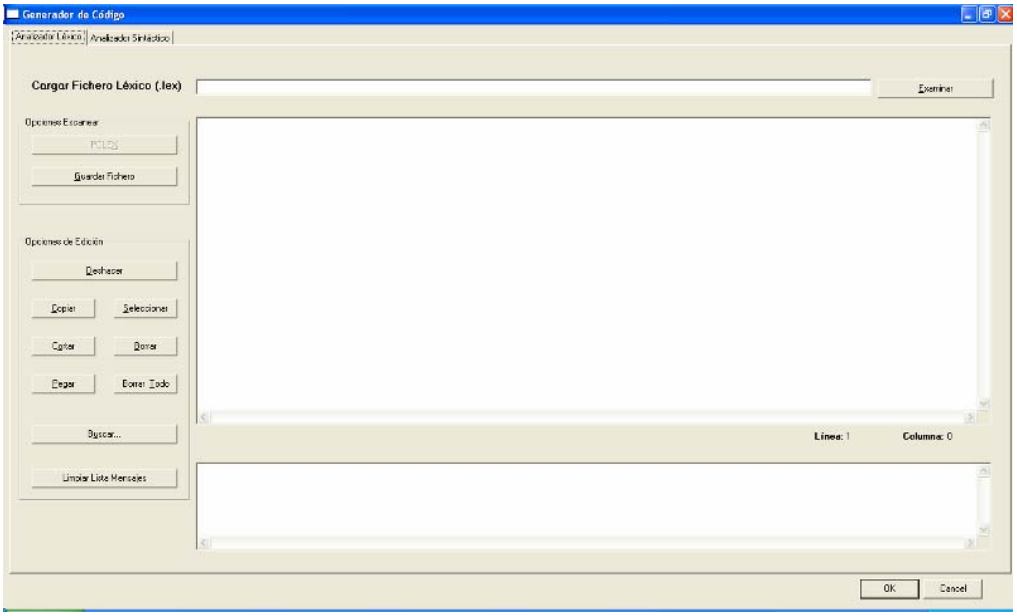


Figura A2.4. Ventana del Generador de código. Vista Analizador léxico.

Cargar Fichero.lex

Muestra el archivo *.lex* actualmente en uso. Para cargarlo se usa el botón *Examinar*.

Opciones Escanear

Permite realizar las acciones necesarias para crear el fichero generador de código (cargando pclex y generando un archivo en C) si el arquitecto de software hubiera cambiado la gramática de *LEDA*.

Opciones de Edición

Contiene botones correspondientes a las opciones de edición de esta ventana: *Deshacer, Copiar, Cortar, Pegar, Seleccionar Todo, Borrar, Buscar y Limpiar mensajes*.

Ventana de Edición

Es la ventana de trabajo que permite editar/modificar el fichero *.lex* actual.

Ventana de mensajes

Muestra los pasos que se han ido realizando.

A2.5.2.2. Analizador Sintáctico

Esta es la ventana asociada al analizador sintáctico de la aplicación *Generador de código*. La ventana (Figura A2.5) tiene las mismas características generales que la del analizador léxico. En este caso, esta función permite cargar o editar el fichero que contiene el analizador sintáctico (especificación de la gramática y tokens de *LEDA*), la tabla de símbolos y una serie de funciones necesarias para la generación de código Java.

En la ventana de mensajes, en este caso, se muestran los mensajes que fueran surgiendo, como por ejemplo errores de compilación.

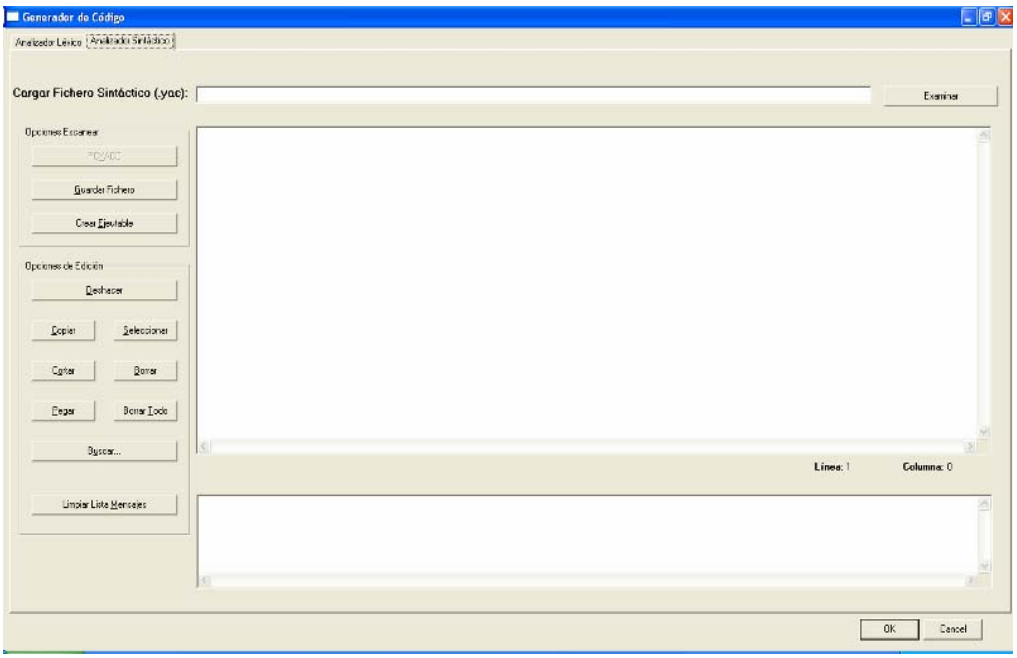


Figura A2.5. Ventana Analizador sintáctico del Generador de código.

A2.5.2.3. Modificar mensajes de la pantalla principal

Esta opción permite modificar los mensajes que muestran los pasos que sigue la generación del prototipo en Java.

A2.5.3. Ventana *Modificar prototipo*

A esta función se llega desde el menú *herramientas* del menú principal. Permite acceder a los ficheros que se han generado en la traducción de un fichero *LEDA* a Java. Estos archivos Java son las clases que usará el prototipo para implementar el sistema. Estas clases Java generadas se apoyan en las *clases básicas* que también se muestran en esta ventana. Desde aquí, además de editar estos ficheros, se pueden compilar (crear los archivos class o bytecodes) haciendo uso del programa *javac.exe* para, más tarde, ejecutarlos usando el interprete de Java.

Siempre que se genera un prototipo, a la clase Java que contiene el programa principal se le asigna el nombre *instance + nombre de la instancia del sistema especificado en leda.exe*. Al compilarlo con *javac.exe* se obtiene el archivo *instanceNombre.class* que es el que se usará para la ejecución del prototipo. Todas estas acciones se hacen automáticamente pulsando el botón de la ventana *Compilar* código generado y la opción *Ejecutar* el prototipo (Figura A2.6).

La ventana *Prototipo* contiene un menú principal, una barra de herramientas, una ventana de edición, una ventana de mensajes y dos ventanas más pequeñas donde se muestran los archivos generados para el prototipo y las *clases básicas* sobre las que se apoya. Se describen a continuación.

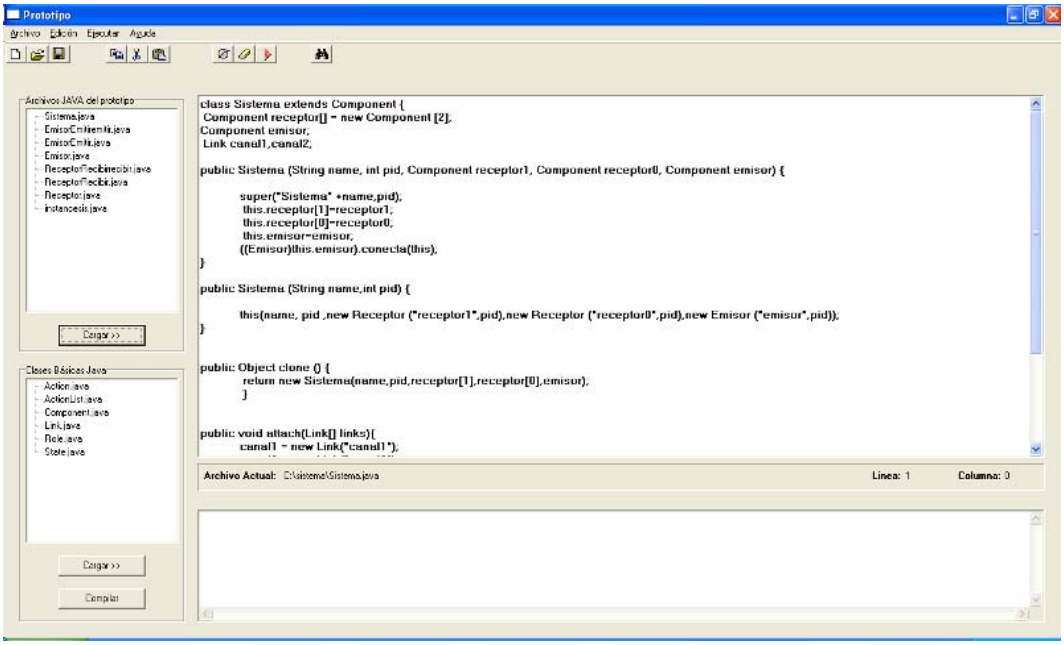


Figura A2.6. Ventana del Prototipo.

A2.5.3.1. Menú principal de la Ventana Prototipo

Este menú contiene todas las opciones que se pueden realizar desde esta ventana:

Menú Archivo: Contiene las opciones: *Nuevo, Abrir, Guardar* y *Salir* que permiten agregar nuevos archivos Java al prototipo generado. Por ejemplo, las clases necesarias para refinar el prototipo obtenido.

Menú Edición: Contiene las opciones usuales de edición.

Menú Ejecución: En este menú está formado por aquellas opciones que se relacionan con la compilación y ejecución del prototipo generado:

Ejecutar Prototipo

Esta opción permite ejecutar el prototipo, para ello, llama al intérprete de Java.

Compilar Prototipo

Esta opción permite realizar la compilación de los archivos Java del prototipo. En la ventana de mensajes se muestran los resultados de la compilación. Si hay errores también se muestran; en otro caso se puede ejecutar el prototipo.

Borrar Listado

Esta opción borra la ventana de mensajes.

Guardar Fichero Log

Se utiliza esta opción si se desea volcar a un fichero los mensajes (en la ventana de mensajes) que se generan cuando se ejecuta el prototipo.

Compilar Clases Básicas

Esta opción permite compilar las *clases básicas* Java en las que se apoya el prototipo.

La barra de tareas de la aplicación *Prototipo* es similar a la de la *Pantalla principal*. Proporciona acceso directo a las siguientes opciones: *Nuevo, Abrir, Guardar, Copiar, Cortar, Pegar, Compilar prototipo, Limpiar mensajes, Ejecutar Prototipo y Buscar*.

A2.5.3.2. Ventanas Archivos Java del prototipo y Clases Básicas

La ventana *Archivos Java del prototipo* contiene los ficheros Java que se han generado al traducir *LEDA* a Java, para crear el prototipo asociado (Figura A2.7). La ventana *Clases Básicas Java* muestra los ficheros Java correspondientes a las *clases básicas*. Cualquiera de estos ficheros se puede editar en la ventana de edición. *Las clases básicas* se pueden/deben recompilar si se modifican.

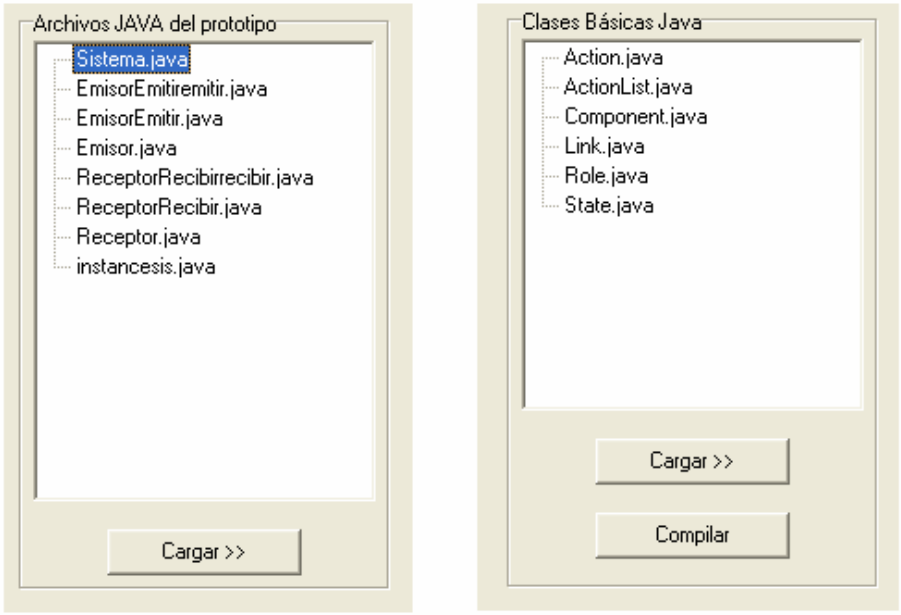


Figura A2.7. Listado de Archivos del prototipo y Clases Básicas.

A2.6. Control de errores

La aplicación controla una serie de errores que se pueden clasificar en cuatro categorías:

1. Los que se refieren a errores léxicos y sintácticos que puedan tener los ficheros *LEDA* que se procesen. Si se da un error de éstos, se muestra en la ventana de mensajes de la *pantalla principal*.
2. Errores que puedan derivarse de usar los programas *pclex* y *pcyacc* en la aplicación *Generador de código*. En este caso, al crearse el ejecutable daría un error, indicando que fue imposible crearlo. Habría que revisar la gramática o el analizador léxico.
3. Errores de compilación de C: aún no habiendo errores en el analizador léxico ni en el sintáctico, si se comete un error en C, éste también se muestra.
4. Errores que indican que el fichero de mensajes o el fichero de las clases generadas no existen, porque el generador de código que se está utilizando ha sido generado a partir un fichero *.yac* que no produce estos archivos. Por ello se aconseja utilizar el fichero original y realizar los cambios que se necesiten para modificar el generador de código, pero sin modificar las instrucciones que hacen que se creen estos ficheros.

A2.7. Actuaciones sobre la gramática *LEDA*

La gramática *LEDA*, como tal, no ha sido modificada puesto que sigue teniendo el mismo símbolo inicial, los mismos símbolos terminales y los mismos tokens. Aunque sí se puede decir que el generador de código ha cambiado, aunque no en lo esencial, pero ha tenido que sufrir algunos cambios. Por una parte, se han realizado algunos cambios triviales en la generación de código, y por otra, cambios que modifican la generación de código, añadiendo instrucciones en el código Java generado.

A2.7.1. Cambios triviales

Estos se refieren a algunos errores que se detectaron en la generación de código (como la falta de puntos y comas en la compilación), probablemente porque la versión de la gramática sobre la que se trabajó no fuera la definitiva.

A2.7.2. Otras modificaciones

Estas modificaciones no triviales son necesarias para lograr un funcionamiento adecuado de la herramienta. El generador de código inicial creaba los ficheros Java que forman el prototipo del sistema. Ahora se crean tres archivos para almacenar la información que antes se mostraba en pantalla. Estos ficheros son:

- El primero (n*ombres.txt*) indica que el prototipo se ha generado con anterioridad. El cambio que se ha realizado en la gramática es que, cuando el generador decide que una nueva clase Java debe ser creada, se guarda su nombre en el fichero *nombres.txt*, para mostrarlos en una ventana.
- *Mensajes.txt*: Este fichero almacena los mensajes que antes se mostraban por pantalla. Ahora se redirige la salida a un fichero de texto (*mensajes.txt)*.
- *Errores.txt*: Este fichero lo genera el programa ejecutable del generador de código si se ha producido algún error léxico o sintáctico. El mensaje que antes salía por pantalla se ha redirigido a este fichero.

El resto de modificaciones realizadas, aunque necesarias, son de menor interés por lo que remitimos al lector a [GaNa08].

A2.8. Conclusiones

Como conclusión de este apéndice se puede decir que *EVADeS* es una herramienta que asiste al ingeniero de software en las tareas relativas a la generación de un prototipo a partir de una especificación arquitectónica en LEDA. Se han descrito las aplicaciones que constituyen la herramienta y sus principales características. *EVADeS* proporciona un entorno grafico para realizar las tareas necesarias conducentes a obtener un prototipo ejecutable, de un modo más cómodo para el usuario que el que ofrecía la

herramienta original que trabajaba en modo consola. En [GaNa08] se describe la *EVADeS* de un modo detallado y completo, incluyendo el manual del programador y diversos ejemplos de utilización que muestran su potencia.

Finalmente, por su interés, se ha realizado una versión en inglés.