Universidad Politécnica de Valencia

Departamento de Sistemas Informáticos y Computación

Oasis como Soporte Formal para la Definición de Modelos Hipermedia Dinámicos, Distribuidos y Evolutivos

Trabajo de Investigación

Programa de Doctorado: **Programación Declarativa e Ingeniería de la Programación**

Jennifer Pérez Benedí

Dirigido por: Dr. Isidro Ramos Salavert

Valencia, septiembre 2003

A Pilar, hermana, amiga
y un ejemplo a seguir
en esta vida

Agradecimientos

A mis padres, por su confianza, apoyo incondicional, paciencia …… y sobre todo por su cariño.

A Maria, por hacer luminosos y alegres los días oscuros, gracias mi vida.

A mi hermana y mi cuñado, por vuestra ayuda, consejos y cariño.

A Isidro, por estar milagrosamente siempre a mi lado, a pesar de la distancia. Gracias, de corazón, por creer en mí.

A Hilario, por sus consejos y apoyo en los momentos difíciles. Ha sido maravilloso conocerte.

A Isabel y Javi, que se han convertido en dos pilares fundamentales en mi vida. Gracias por escucharme y compartir tantos buenos y malos momentos conmigo.

A Emilio, Nour y Artur, por ser compañeros y además, amigos.

A Nour, Patricio, Javi y Pepe, por su colaboración en este trabajo, por compartir conmigo sus conocimientos en distribución, modelado y evolución.

A mis amigas, por serlo, entenderme y hacerme pasar tantos buenos ratos.

Muy especialmente, a Luis César, por entender mi dedicación a este trabajo que me apasiona y por quererme tanto. Sin ti, todo sería distinto.

Contenidos

INTRODUCCIÓN 8
1.1. Objetivos del Trabajo 11
1.2. Estructura del documento 11
TRABAJOS RELACIONADOS 14
2.1.Arquitecturas Software 16
2.1.1. Lenguajes de Definición de Arquitecturas 17
2.1.2. Lenguajes Gráficos de Definición de Arquitecturas 18
2.2. Diseño Basado en Componentes 18
2.3. Desarrollo de Software Orientado a Aspectos 21
2.4. Sistemas Hipermedia 22
2.5. Calidad Software 23
2.6. Coordinación 23
2.7. Context-Awareness 23
PRISMA 24
3.1. Modelo PRISMA 27
3.2. Visión basada en componentes 27
3.2.1. Interfaces 28
3.2.2. Aspectos 29
3.2.3. Componentes 32
3.2.4. Conectores 36
3.2.5. Sistemas 39
3.2.6. Relaciones 43
3.2.6.1. Relaciones topológicas 44
3.2.6.2. Relaciones conceptuales 45
3.2.7. Modelo Arquitectónico 47
3.3. Visión basada en aspectos 48
3.3.1. Conceptos Generales de Aspectos 48
3.3.2. Aspecto Funcional 52
3.3.3. Aspecto Coordinación 53
3.3.4. Aspecto Distribución 54
3.3.5. Aspecto Evolución 55
3.3.6. Aspecto Calidad 56
3.3.7. Aspecto Context-Awareness 56
3.3.8. Aspecto Navegación 57
3.3.9. Aspecto Presentación 57

LENGUAJE DE DEFINICIÓN DE ARQUITECTURAS PRISMA 58
4.1. Lenguaje de definición de componentes 60
4.1.1. Interfaz 60
4.1.2. Aspecto 61
4.1.2.1. Aspecto Funcional 66
4.1.2.2. Aspecto Coordinación 66
4.1.2.3. Aspecto Distribución 71
4.1.3. Componente 71
4.1.4. Conector 73
4.1.5. Sistema 74
4.2. Lenguaje de configuración 77
LENGUAJE GRÁFICO PRISMA 82
5.1. Profile UML para PRISMA 85
5.1.1. Clasificación de conceptos PRISMA en base al metamodelo UML 85
5.1.2. Profile UML para PRISMA 86
5.2. Lenguaje Gráfico PRISMA 97
5.2.1. Elemento Gráfico de Aspecto 97
5.2.2. Elemento Gráficos de Interfaz 98
5.2.3. Elementos Gráficos de Componente, Conector y Sistema 99
5.2.4. Elemento Gráfico de Protocolos y Coreografías 100
5.2.4. Elemento Gráfico de Weaving 101
5.2.4. Elemento Gráfico de Attachment 101
5.2.5. Elemento Gráfico de Binding 101
CONCLUSIONES Y TRABAJOS FUTUROS 104
BIBLIOGRAFÍA 110
ANEXO: CASO ESTUDIO SUBASTAINTERACTIVA, Un portal Web de Subastas 118
A.1. Descripción del Caso de Estudio 120
A.2.- Obtención de Requisitos 126
A.2.1. Casos de Uso para la Gestión de Administradores 126
A.2.2. Casos de Uso para la Gestión de Vendedores 127
A.2.3. Casos de Uso para la Gestión de Compradores 127
A.2.4. Casos de Uso para la Gestión de Galerías 127
A.2.5. Casos de Uso para la Gestión de Subastas 128
A.2.6. Casos de Uso para la Gestión de Ventas 129
A.2.7. Casos de Uso para la Gestión de Marketing 130
A.3. Especificación PRISMA 131

Índice de Figuras

Figura 1. Sosy Modeler® Automatic Generation 10
Figura 2. Paradigma de prototipación automática 26
Figura 3. PRISMA 27
Figura 4. Paquete interfaz del metamodelo de PRISMA 28
Figura 5. Paquete puerto del metamodelo de PRISMA 29
Figura 6. Paquete rol del metamodelo de PRISMA 29
Figura 7. Aspectos asociados a un componente funcional 30
Figura 8. Aspectos que forman un componente 31
Figura 9. Paquete aspecto del metamodelo de PRISMA 32
Figura 10. Paquete componente del metamodelo de PRISMA 33
Figura 11. Paquete weaver del metamodelo de PRISMA 35
Figura 12. Vistas de un componente PRISMA 35
Figure 13. Paquete conector del metamodelo de PRISMA 36
Figura 14. Vistas de un connector PRISMA 39
Figura 15. Paquete sistema del metamodelo de PRISMA 40
Figura 16. Vistas de un componente PRISMA 42
Figura 17. Paquete attachment del metamodelo PRISMA 44
Figura 18. Paquete binding del metamodelo PRISMA 45
Figura 19. Paquete agregación del metamodelo PRISMA 47
Figura 20. Paquete esquema del metamodelo PRISMA 48
Figura 21. Paquete identificador del metamodelo PRISMA 49
Figura 22. Paquete atributos del metamodelo PRISMA 49
Figura 23. Paquete servicios del metamodelo PRISMA 49
Figura 24. Paquete evaluaciones del metamodelo PRISMA 50
Figura 25. Paquete precondiciones del metamodelo PRISMA 50
Figura 26. Paquete restricciones de integridad del metamodelo PRISMA 51
Figura 27. Paquete triggers del metamodelo PRISMA 51
Figura 28. Paquete protocolo del metamodelo PRISMA 51
Figura 29. Paquete estado del metamodelo PRISMA 52
Figura 30. Paquete Fbf del metamodelo PRISMA 52
Figura 31. Paquete sincronizaciones del metamodelo PRISMA 53
Figura 32. Paquete estado del metamodelo PRISMA 54
Figura 33. Uso de patrones de distribución para el proceso de desarrollo PRIMA 55
Figura 34. Arquitectura 4 capas 84
Figura 35. Modelado del estereotipo Aspect 88
Figura 36. Modelado del estereotipo Component 90
Figura 37. Modelado del estereotipo Port 90
Figura 38. Modelado del estereotipo Protocol 91
Figura 39. Modelado del estereotipo Connector 92
Figura 40. Modelado del estereotipo Rol 93
Figura 41. Modelado del estereotipo Weaving 94
Figura 42. Modelado del estereotipo Aggregation 95
Figura 43. Modelado del estereotipo Choreography 95
Figura 44. Modelado del estereotipo Attachmen 96
Figura 45. Modelado del estereotipo Binding 97
Figura 46. Aspectos Gráficos de Funcionalidad, Coordinación y Distribución 98
Figura 47. Interfaz Gráfica 98
Figura 48. Enlace Gráfico entre Interfaces y Aspectos 98
Figura 49. Componente Gráfico 99
Figura 50. Definición gráfica de componentes a partir de aspectos 99
Figura 51. Visión gráfica aspecto de un componente 100
Figura 52. Visión gráfica interfaz de un component 100
Figura 53. Protocolo Gráfico 101
Figura 54. Weaving Gráfico 101
Figura 55. Attachment Gráfico 101
Figura 56. Binding Gráfico 102
Figura 57. Galerías clasificadas por categorías 120
Figura 59. Búsqueda avanzada de artículos 121

Figura 60. Exposición de artículos de una galería *122*
Figura 61. Alta subastador *122*
Figura 62. Alta comprador *123*
Figura 63. Subastas simultáneas *123*
Figura 64. Oportunidades para el usuario conectado *125*
Figura 65. Intereses para el usuario conectado *126*
Figura 66. Gestión de Administradores *126*
Figura 67. Gestión de Vendedores *127*
Figura 68. Gestión de Compradores *127*
Figura 69. Gestión de Galerías *128*
Figura 70. Gestión de Subastas *129*
Figura 71. Gestión de Ventas *130*
Figura 72. Gestión de Marketing *130*

INTRODUCCIÓN

Los sistemas de información cada vez son más difíciles de desarrollar debido a sus complejas estructuras y a la dinámica y competitividad del mercado que implica una evolución constante de los requisitos del sistema. Éstas, entre otras razones, hacen que los entornos de desarrollo deban permitir construir aplicaciones con arquitecturas complejas, distribuidas, evolutivas y reutilizables. A todas estas características se ha de sumar el de que los modelos arquitectónicos actuales han de satisfacer los requisitos necesarios.

Por otro lado, en la actualidad es fundamental que los procesos de producción de sistemas software minimicen el tiempo y el coste de desarrollo y mantenimiento. De ahí que no sólo sea importante que los entornos software permitan definir elementos reutilizables y evolutivos, sino que también han de posibilitar la importación de componentes COTS (comercial *off-the-shelf*). En cuanto a evolución se refiere, cada día esta cobrando una mayor relevancia dentro de los productos sofware, hasta el punto de que están surgiendo dos vertientes claramente definidas: la evolución debida al cambio de requisitos (*requirements-driven)* y la evolución provocada por un cambio de contexto (*context-driven*). Aquellos productos que soportan estos dos tipos de evolución se caracterizan por la mejora en la mayoría de factores relevantes de calidad.

Hoy en día existen herramientas CASE capaces de generar aplicaciones siguiendo el paradigma de la prototipación automática [Bal85]. Estas herramientas son conocidas como compiladores de modelos y son capaces de generar el código fuente y el esquema de la base de datos de un sistema de información a partir de su esquema conceptual. Dicha generación de código puede ser completa como lo hacen Oblog Case [Ser94] y Sosy Modeler® (OO-Method/CASE [Pas97]), o parcial, como Rational Rose [Rat], System Architect [Sys], Together [Tog] y otros. Concretamente, Sosy Modeler® (OO-Method/CASE) genera aplicaciones con una arquitectura de 3 capas (presentación, lógica de negocio y persistencia) y utiliza OASIS como lenguaje de especificación.

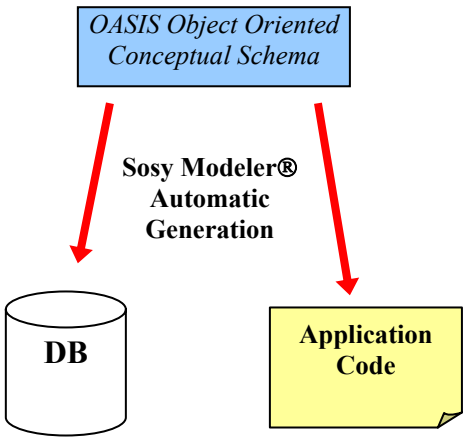

Figura 1. Sosy Modeler® Automatic Generation

El hecho de que la mayoría de estos compiladores de modelos sigan un enfoque orientado a objetos es un inconveniente, ya que el enfoque orientado a objetos por sí solo no permite abordar los requisitos de los sistemas software complejos actuales. Esto ha hecho surgir dos tendencias en el desarrollo de sistemas software, el Desarrollo de Software Basado en Componentes (DSBC) [Sou99] y el Desarrollo de Software Orientado a Aspectos (DSOA) [Aos]. Sin embargo, no existen compiladores de modelos que soporten la especificación de sistemas de información utilizando estas dos nuevas aproximaciones.

1.1. Objetivos del Trabajo

Este trabajo pretende proporcionar un modelo y unos lenguajes que permitan especificar sistemas de información complejos, distribuidos, dinámicos y reutilizables utilizando los dos nuevas tendencias de desarrollo, DSBC y DSOA, con el objetivo de obtener las ventajas de ambas aproximaciones.

Se pretende que en un futuro, dicho modelo sea parte de un marco de trabajo o *framework* que permita la generación de código a partir de los modelos arquitectónicos especificados en dicho framework. Para que dicha generación sea posible el trabajo define los lenguajes extendiendo OASIS y preserva la capacidad de evolución y generación de aplicaciones a partir de sus modelos, como ya se realizo en Sosy Modeller.

Por otro lado, se pretende que este trabajo sea la base de una tesis que se enmarcará en el ámbito de los sistemas hipermedia y por lo tanto, en la definición de modelos navegacionales y de presentación utilizando una aproximación orientada a aspectos y basada en componentes.

1.2. Estructura del documento

El trabajo está dividido en seis capítulos y un anexo. A continuación se describe brevemente el contenido de cada uno de ellos.

En el capítulo 2 se describe el estado del arte, de todos aquellos campos que abarca el trabajo: arquitecturas software, DSBD, DSOA, sistemas hipermedia, etc.

En el capítulo 3 se presenta el modelo PRISMA. En este capítulo se detallan los conceptos del modelo y el metamodelo que los representa.

En el capítulo 4 se detalla el lenguaje de especificación de arquitecturas PRISMA (Architecture Definition Language (ADL)). El capítulo presenta dos lenguajes textuales para la especificación de modelos PRISMA, el lenguaje de definición de componentes y el lenguaje de configuración.

El capítulo 5 muestra el profile UML diseñado para la definición de un lenguaje gráfico asociado a los lenguajes textuales presentados en el capítulo 4. Además del profile, el lenguaje gráfico se explica con detalle.

El capítulo 6 recoge las conclusiones y los trabajos futuros que pretenden abordarse tanto a corto como a largo plazo.

Finalmente se presenta un anexo en el que se desarrollo un caso de estudio de un portal de subastas web. Dicho caso de estudio recoge la memoria del caso, la especificación de requisitos asociada y la definición textual utilizando los lenguajes PRISMA.

TRABAJOS RELACIONADOS

En la actualidad existen una gran variedad de trabajos relacionados con modelos arquitectónicos, componentes, lenguajes de definición de arquitecturas, reconfiguración dinámica de arquitecturas y todo aquello que guarda relación con el desarrollo de sistemas software complejos. Este gran esfuerzo de investigación persigue el objetivo de encontrar un consenso en los conceptos básicos de sistemas software complejos, así como una definición precisa y una metodología a aplicar para su desarrollo.

Con el objetivo de que el capitulo no resulte muy extenso por la cantidad de áreas que abarca el trabajo, se van a tratar todas y cada una de ellas citando únicamente aquellas contribuciones más significativas.

Las principales materias que trata este trabajo son las arquitecturas software, el diseño basado en componentes y el diseño orientado a aspectos. Dentro de cada una se tratan diversos campos, no siendo cada uno de ellos de menor importancia.

Las arquitecturas software abarcan el concepto de arquitectura, los lenguajes de especificación de arquitecturas (Architecture Definition Languages (ADL's)), la capacidad de evolución de modelos y lenguajes arquitectónicos, así como los lenguajes gráficos para el modelado de arquitecturas software.

En cuanto al diseño basado en componentes, los puntos que adquieren un mayor interés son la definición de componente y los distintos criterios que se utilizan hoy en día para realizar un diseño de componentes a partir de los requisitos del sistema.

Finalmente, el desarrollo de software orientado a aspectos es el que más campos trata, ya que PRISMA tiene en cuenta varios aspectos como: distribución, hipermedia (navegación, presentación), calidad, coordinación, evolución, etc. Cada uno de ellos trata un área diferente donde existen una amplia gama de aportaciones, aunque no estén abordadas desde el punto de vista de la orientación de aspectos.

2.1.Arquitecturas Software

El concepto de arquitectura software está muy extendido, sin embargo no existe una definición única y estándar para este concepto. Existen muchas definiciones de arquitecturas, con partes en común y puntos claros que las diferencian. Estas definiciones pueden ser muy dispares, desde definiciones genéricas a definiciones demasiado especificas, todo depende del autor y de la aplicación que le desee dar al concepto.

Un autor, cuya contribución en el mundo de las arquitecturas software y las componentes ha sido muy aceptada, es David Garlan. Sus trabajos han sido un gran precedente en la elaboración de este trabajo y su definición de arquitectura es una de las más utilizadas en la literatura. Por este motivo, se ha seleccionado su última definición de arquitectura [Gar95] como referencia:

*La **arquitectura software** esta 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.*

2.1.1. Lenguajes de Definición de Arquitecturas

Los lenguajes de definición de arquitecturas son conocidos como ADLs (Architectural Defintion Languages). Uno de los ADLs más conocido es Wright [All97], Wright es un lenguaje formal basado en álgebra de procesos, cuyo inconveniente es el hecho de que es estático. Otro lenguaje conocido, que también esta basado en álgebra de procesos y resulta de interés por sus características, es δarwin [Mag95]. Las principales ventajas que aporta este lenguaje son el uso de π-cálculo, la disponibilidad de herramientas visuales interactivas y su compatibilidad con CORBA.

Otro lenguaje a resaltar es Durra [Bar01], un ADL clásico muy superior a algunos de los ADLs modernos. DURRA es un lenguaje de descripción de tareas para el desarrollo de aplicaciones distribuidas. Este lenguaje, incorpora los conceptos de componente y conector, de forma que un componente puede estar compuesto por subcomponentes que están unidos por una serie de estructuras internas. DURRA dentro de la especificación de una componente especifica los componentes que incluye y la estructura interna mediante la que se conectan dichos subcomponentes. Además, también incluye las posibles reconfiguraciones de la estructura interna que alberga, permitiendo la reconfiguración mediante el disparo de eventos que son provocados por señales de un modelo temporal.

Existen contribuciones importantes por autores españoles de las universidades de Málaga y de Valladolid como Leda [Can00] y Pilar [Cue02], respectivamente. Ambos ADLs se caracterizan por ser altamente dinámicos y usar distintos formalismos.

Un lenguaje que también resulta especialmente relevante para este trabajo es C2. Las características más importantes de C2 es su gestor de evolución, que garantiza la consistencia de la arquitectura ante los cambios, y su capacidad de generación de código mediante una biblioteca Java.

En cuanto a evolución, uno de los lenguajes que más se aproxima a los que se presenta en este trabajo es el propuesto por Loques y Leite [Loq00] en su modelo R-RIO, en el cual se incorporan aspectos de reconfiguración dinámica a través del metanivel. Además, se encuentran enfoques orientados a un nivel de abstracción inferior, es decir, orientados a implementación como [McG02]. En GUARANÁ [Oli98] también se presenta un metanivel complejo que hay que definir durante la compilación, siendo su estructura ajena al modelo.

Finalmente, cabe destacar el trabajo de [Med00], en el que se realiza un estudio y comparación exhaustiva entre la gran variedad de ADLs que existen en la actualidad.

2.1.2. Lenguajes Gráficos de Definición de Arquitecturas

En la actualidad, la mayoría de lenguajes de definición de arquitecturas reconocidos, no tienen un lenguaje gráfico asociado. De hecho, ni tan si quiera UML (versión 1.5.) proporciona un soporte para el modelado de arquitecturas software, aunque su futura versión 2.0 probablemente cubra este campo. Un enfoque que enfatiza el modelado con UML de arquitecturas es Model-Driven Architecture (MDA), propuesto por la OMG (www.omg.org). Dicho enfoque persigue la integración, interoperabilidad de los sistemas y la independencia de las plataformas de implementación. MDA está basado en el modelado de diferentes aspectos y niveles de abstracción, explotando las interrelaciones y estableciendo transformaciones hacia su implementación.

Uno de los ADLs que tiene un lenguaje gráfico asociado es RAPIDE, pero sólo en su versión 2.0. Este lenguaje no ha sido citado en el apartado anterior, ya que difiere demasiado de la idea de arquitectura de este trabajo y de la gran mayoría de lenguajes. RAPIDE define la estructura arquitectónica mediante patrones de interacciones entre interfaces, entendiendo éstas como un tipo de componentes. Por otro lado, Christian Prehofer en su trabajo [Pre02] propone un método de descripción gráfica para el modelado de componentes. Este método utiliza diagramas de transición de estados para modelar las interacciones entre componentes, componentes que en su aproximación se definen mediante la composición de *features*. Otra propuesta, enmarcada en el modelado gráfico de componentes COTS, es la de [Mon02]. Dicha propuesta presenta una extensión UML para facilitar el modelado de arquitecturas software que permiten la importación de COTS. Finalmente, hay que destacar el trabajo de Keng Ng y Jeff Kramer [Ng95] en el que se presenta un soporte automatizado para desarrollo de arquitecturas a través de una herramienta CASE con modelos gráficos. Éste trabajo se basa en el lenguaje δarwin citado en el apartado anterior.

Una linea de investigación diferente y que está despertando un gran interés, es el desarrollo de profiles UML para que las herramientas actuales de modelado soporten el modelado orientado a aspectos. De esta tendencia han surgido trabajos como [Ald01], [Suz99] y [Lio02].

2.2. Diseño Basado en Componentes

Actualmente, las tecnologías software basadas en componentes están incrementando su popularidad y accesibilidad, propiciando un cambio en los hábitos y expectativas del desarrollo de software. Las herramientas de desarrollo de aplicaciones y las tecnologías actuales han conseguido que hoy por hoy los componentes sean la clave para la reutilización de gránulos de código mayores que los objetos. De este modo, el desarrollo de software basado en componentes permite reducir el tiempo de desarrollo y mantenimiento de las aplicaciones.

El proceso de desarrollo de software tradicional no hace hincapié en la reutilización, no especifica cómo y cuándo el analista debe hacer uso de la reutilización. Por este motivo, el

potenciar la reutilización de código en el desarrollo de software implica un cambio en el proceso de desarrollo. Este nuevo proceso de desarrollo debe incluir una nueva fase de creación en la que se identifique aquellas partes del sistema que son potencialmente reutilizables y por tanto, posibles componentes. Así mismo, deberá permitir incorporar componentes ya desarrolladas al sistema, incluyendo la actividad de selección y búsqueda de dichas componentes en esta fase.

La actividad de identificación de componentes y definición de una arquitectura en las que reutilizarlas se llama *domain engineering* en la comunidad de reutilización. El objetivo de esta actividad es que el analista defina y estructure las componentes de forma que estas permitan construir aplicaciones de forma rápida, eficiente y poco costosa.

La etapa de identificación de elementos (componentes, conectores y sistemas) tiene como objetivo primordial la detección de los componentes (o sistemas) del sistema ha desarrollar, para posteriormente detectar los conectores que permiten su comunicación. Para ello, se parte de la especificación de requisitos resultante de la etapa de obtención de requisitos.

En esta etapa es primordial tener presente la definición de componente, ya que en la actualidad no existe un consenso sobre qué es un componente. Existen dos tendencias, una es más orientada a implementación y otra es más genérica. La primera recoge definiciones relacionadas con el hecho de que una componente es un paquete de código [Sou99], mientras que la segunda define un componente como un artefacto que ha sido desarrollado específicamente para ser reutilizado. Entre estas dos tendencias, este trabajo se decanta por la segunda, ya que la hace independiente a la solución del problema, manteniendo el alto nivel de abstracción. Esta segunda definición permite que un componente sea tanto un caso de uso como una clase o cualquier otro elemento que surja durante el proceso de desarrollo, permitiendo que la identificación de componentes se pueda basar en diferentes criterios. El trabajo de [Ira00] cita los criterios de casos de uso, patrones de diseño, dominios de negocio, cambios previstos y componentes existentes. Además de estos existen los criterios de entidad (clase o tipo abstracto de datos) [Jac97] y el de escena [Nor98]. A continuación se hace un pequeño análisis de estos criterios, junto con la definición de componente que presentan cada uno de ellos.

- **Entidades**

La aproximación de [Jac97] tiene como objetivo construir software reutilizable y que la reutilización esté presente en cada una de las fases del ciclo de vida de un producto software. Esta aproximación trasladada al área de las arquitecturas software, y concretamente, al diseño basado en componentes, trata de construir componentes reutilizables. De forma que dicha propiedad permita importarlos a diferentes modelos arquitectónicos optimizando las pruebas, el mantenimiento y la calidad del código y del tiempo de desarrollo.

Esta propuesta utiliza las entidades como criterio de identificación de componentes a partir del modelo de casos de un sistema de información, entiendo por entidad: tipos, clases y subsistemas. Este criterio consigue el desarrollo de software reutilizable, ya que la definición de componente y subsistema que especifican son las siguientes:

<<Un componente objeto es un tipo o clase que es reutilizable en varios sistemas>>

<<Un componente subsistema es un conjunto de componentes organizados como un subsistema>>

<<Un componente subsistema contiene tipos, clases y otros elementos que se desean reutilizar de forma conjunta >>

- **Escenas**

El concepto de escena aparece en el trabajo de [Nor98]. Este trabajo considera la escena como un criterio para la detección de componentes en la especificación de requisitos. El criterio consiste en que cada escena que se detecte en la especificación de requisitos se corresponde con una componente del sistema. Para realizar dicha detección de forma adecuada es necesario definir el concepto de escena. El concepto de escena dentro de un sistema de información esta vagamente definido. Es por este motivo, que a continuación se propone una definición.

<<Una escena es una actividad relevante dentro de un sistema de información. Una escena se caracteriza por las tareas que se desempeñan en la actividad siguiendo un determinado protocolo, los actores que las realizan y el espacio virtual o físico donde se desarrolla.>>

- **Patrones de diseño**

Cada vez está más extendido el uso de patrones en el proceso de desarrollo de software debido a que mejora dos características muy importantes: la calidad del producto software y la productividad durante su proceso de desarrollo. La optimización de la calidad es debida a que un patrón es una solución ya probada que garantiza un correcto funcionamiento y unas determinadas propiedades que aseguran la calidad de un patrón. Mientras que el incremento de la productividad se debe a la reutilización de dichos patrones, evitando su codificación y prueba.

Los primeros patrones que se diseñaron dentro del área de la informática se aplicaron para interfaces gráficas de usuario (IGU), su expansión ha llegado hasta las arquitecturas software. Los patrones arquitectónicos proporciones descripciones de la arquitectura software. Estos patrones se proponen en los trabajos de Fowler [Fow96], Gamma et al. [Gam95], y Hay [Hay95].

Cada patrón arquitectónico define una estructura mediante un conjunto de interfaces implementadas por un conjunto de clases. Un patrón arquitectónico se implementa como un solo componente que contiene y encapsula dichas clases e interfaces. De este modo, un componente interactuará con otros componentes (que implementan otros patrones) proporcionando la funcionalidad del sistema. El trabajo de [Lar99] propone que cada patrón se

corresponda con un componente. De forma que aquellas clases que participan en varios patrones quedan distribuidas en diferentes componentes.

- **Casos de Uso**

Los casos de uso constituyen una división del sistema basada en la funcionalidad. En [Jac97, Ber99, Sou99] se propone que cada caso de uso del sistema sea un componente. De forma que aquellas clases que estén relacionadas con el caso de uso, formaran parte del componente asociado al caso de uso.

- **Dominios de Negocio**

  [All98] presenta el dominio de negocio como una parte de la organización que contiene recursos, ofrece servicios y define sus políticas de negocio. En base a este concepto de dominio de negocio, propone como criterio de identificación de componentes el dominio de negocio. De forma que, que un dominio de negocio del sistema se corresponde con un componente.

- **Evolución Prevista**

Esta propuesta se basa en el análisis tanto de los requisitos actuales como de los futuros, para facilitar la evolución del sistema ante los cambios previstos. El criterio de identificación propuesto por [Hoo99] es agrupar dentro de un mismo componente todas aquellas clases que se vean afectadas por el mismo cambio.

- **Componentes Existentes**

A diferencia de los enfoques anteriores, aquí el sistema no parte desde cero, existen partes del sistema ya desarrolladas. Por este motivo, se ha de realizar una integración entre el sistema ya construido y la nueva funcionalidad que se le desea incorporar. En esta aproximación, la identificación de componentes la marcan las componentes ya existentes en el sistema original [Voa98].

2.3. Desarrollo de Software Orientado a Aspectos

El principio de *separation of concerns* propone la encapsulación de propiedades en entidades separadas que localicen los cambios y traten una misma cuestión. Se han propuestos varios modelos de separación de propiedades, sin embargo de todos ellos el que ha tenido un mayor aceptación ha sido el paradigma orientado a aspectos (AOP).

Se han propuesto una gran variedad de enfoques de programación orientada a aspectos desde mecanismos lingüísticos [Xer02] hasta técnicas basadas en filtros mediante sección transversal [Elr01], modelo de disfraces [San99] y enfoques multidimensionales [Ras01, Tar99].

También se ha realizado trabajos para incorporar aspectos en el nivel de diseño extendiendo el metamodelo de UML [Cla01].

El desarrollo de software orientado a aspectos esta teniendo un gran auge en la comunidad informática debido a que su código modular consigue que los costes de desarrollo, mantenimiento y evolución del software se reduzcan. Por este motivo, existe una tendencia hacia la incorporación del concepto de aspecto desde las primeras fases del ciclo de vida, existiendo trabajos elaborados en el área de la especificación de requisitos [Ras02].

Dentro de los distintos modelos de aspectos que se han creado hasta el momento, se pueden observar dos claras tendencias que permiten clasificarlos en dos grandes grupos: los modelos estáticos y los modelos dinámicos. La diferencia entre ambos se encuentra en el proceso de generación de código. Los modelos estáticos (AspectJ) generan un solo componente en el que se encuentra mezclado el código funcional y el código de los aspectos, mientras que en los modelos dinámicos se generan distintas entidades para los distintos aspectos y el componente (Modelo de disfraces). La ventaja de los modelos dinámicos es la facilidad que proporcionan para incluir o eliminar aspectos durante el proceso de ejecución.

2.4. Sistemas Hipermedia

Actualmente, la interacción hombre-máquina [Loz01] [Loz00] es uno de los campos de investigación que están en auge. Dentro de esta área se encuentra la tecnología hipermedia [Win97]. Estos sistemas dan soporte a la Cultura Digital, es decir, cualquier manifestación electrónica como bibliotecas digitales, museos virtuales, diccionarios electrónicos, periódicos on-line, enciclopedias interactivas, etc.

Los sistemas hipermedia [Bal93] permiten crear, enlazar y compartir información a través de distintos medios como textos, gráficos, audio, vídeo, animación y servicios web. Clásicamente, estos sistemas pueden ser vistos como un sistemas de BD que no tienen una estructura regular y que proporcionan una metodología para acceder a la información.

El modelo de diseño de sistemas hipermedia no resulta suficientemente rico para organizar, estructurar, acceder y realizar tareas de aplicaciones relacionadas con la Cultura Digital. Además, los sistemas hipermedia se caracterizan por ser de una naturaleza muy variable ya que sus requisitos continuamente están sometidos a cambios. Sin embargo, en la actualidad, los sistemas hipermedia son adaptativos y no llegan a ser evolutivos a pesar de que los sistemas hipermedia necesitan un soporte automático a la evolución tanto de las aplicaciones como de sus datos.

Finalmente, cabría destacar que debido a los fuertes cambios que padecen los sistemas hipermedia y la frecuencia con la que se dan, en muchos casos su diseño y construcción se realizan aplicando metodologías ágiles [Fow00] para soportar la evolución y realizar un mantenimiento de forma menos pesada y costosa.

2.5. Calidad Software

Actualmente, existen diferentes propuestas para medir la calidad, cualitativamente o cuantitativamente. Existe una tendencia hacia la automatización de estas mediciones y algunas aproximaciones como Win-Win[In02], NOFUN[Bot01], QRL[Rui02] han conseguido automatizar algunas de sus actividades. Dentro de este subconjunto de propuestas, es importante destacar QRL ya que dispone de una semántica operacional precisa que permite la negociación automática.

2.6. Coordinación

La coordinación esta íntimamente ligada al concepto de conector, su definición y especificación. Según el artículo de [Sha94], el concepto de conector emerge y esta justificado cuando permite una separación de intereses, de forma que un componente o sistema especifica la funcionalidad, mientras que un conector tan solo especifica la interacción entre elementos. Una de las ventajas de esta separación de intereses, según [Sha94] es que se consigue mayor nivel de abstracción y modularidad. Esta ventaja, junto con el hecho de que dota al lenguaje una orientación arquitectónica en lugar de orientada a objetos, hacen que la mayoría de lenguajes lo incorporen como ciudadano de primer orden. Sin embargo, algunos autores, como Jeff Kramer, prefieren la ausencia de conectores (LEDA), ya que según ellos distorsionan la naturaleza compositiva de la arquitectura software.

La definición semántica del concepto de conector dentro de un modelo de especificación de arquitecturas es fundamental , un trabajo que sienta precedente en dicha definición es el de Robert Allen y David Garlan [All97].

2.7. Context-Awareness

Context-awareness no es una nueva idea pero es un factor muy importante cuando se desarrollan y mantienen aplicaciones que se desea que sean auto-configurables y auto-adaptativas. El trabajo de [Dey00] incluye un estudio detallado de aplicaciones context aware , clasificando las aplicaciones en tres categorías. La primera categoría de aplicaciones incluye aquellas que ofrecen información y servicios al usuario dependiendo del contexto. Las segundas ejecutan servicios automáticamente en el momento en que se conocen determinadas condiciones contextuales. La tercera categoría de aplicaciones incluyen aquellas que etiquetan información con descripciones del contexto (por ejemplo: el contexto en el que la información se ha obtenido).

PRISMA

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

El modelo PRISMA se basa en componentes y aspectos para la construcción de modelos arquitectónicos. Dicho modelo se caracteriza por la integración que realiza deL Desarrollo de Software Basado en Componentes (DSBC) y del Desarrollo de Software Orientado a Aspectos (DSOA) y por sus propiedades reflexivas. Esta capacidad de reflexión se consigue a través de un meta-nivel que permite evolucionar al modelo arquitectónico. PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables.

PRISMA propone una metodología guiada y (semi-)automatizada para obtener el producto software final basándose en el paradigma de la prototipación automática de Balzer [Bal85] (ver figura 2). En dicha metodología, el usuario utilizará una herramienta (compilador de modelos) que le permita la validación y animación de los modelos arquitectónicos que en ella se especifiquen. Es necesario utilizar una serie de modelos y lenguajes para definir un modelo arquitectónico, que posteriormente será compilado de forma automática al código final de la aplicación. Estos lenguajes deben ser formales e independientes de la tecnología con el objetivo de poder ser compilados sin ambigüedades a cualquier tecnología y/o lenguaje de desarrollo.

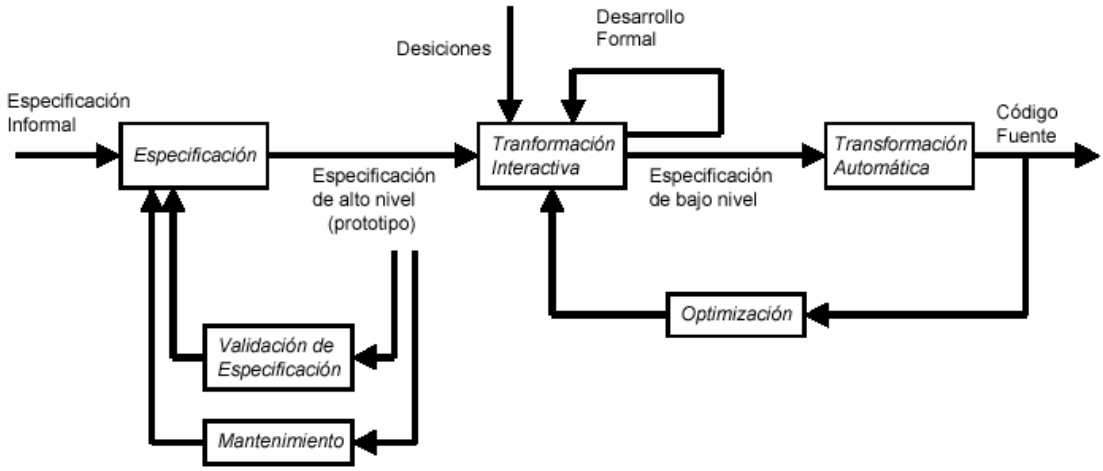

Figura 2. Paradigma de prototipación automática

PRISMA esta basado en un lenguaje de especificación formal OO llamado OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos, que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA preserva las ventajas de OASIS, garantizando la compilación de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos arquitectónicos, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

3.1. Modelo PRISMA

El modelo arquitectónico PRISMA que se presenta a continuación integra dos aproximaciones de desarrollo de software, el DSBC y el DSOA. Esta integración se consigue definiendo los tipos mediante la composición de aspectos.

La mayoría de modelos arquitectónicos analizan cuáles son los tipos básicos para la especificación de sus arquitecturas y exponen su sintaxis y semántica. El modelo PRISMA, además de definir sus tipos, también especifica los aspectos que cubren las necesidades de cada uno de ellos.

Un tipo del modelo de arquitectura puede ser visto como un prisma con tantas caras como aspectos considere. Dichos aspectos están definidos desde la perspectiva del problema y no de su solución, aumentando el nivel de abstracción y evitando el solapamiento de código que puede sufrir la programación orientada a aspectos [Kic01].

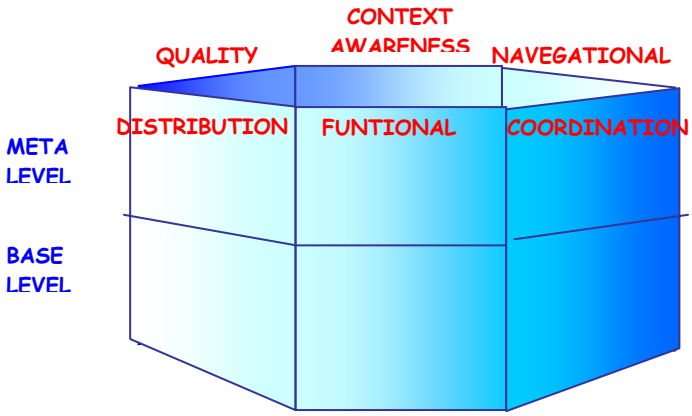

Figura 3. PRISMA

Por este motivo, el modelo PRISMA puede analizarse desde dos perspectivas diferentes, la orientada a aspectos y la orientada a componentes. A continuación se presenta de forma detallada el modelo desde ambas perspectivas.

3.2. Visión basada en componentes

El modelo arquitectónico consta de tres tipos: componentes, conectores y sistemas. Cada uno de estos tipos esta compuesto por tantos **aspectos** como se consideren relevantes para definir el sistema de información con precisión. Una especificación arquitectónica PRISMA es la instanciación de un conjunto de tipos definidos interconectados entre si.

Los componentes pueden ser simples o complejos, correspondiéndose un componente complejo con un **sistema**. Un **componente** es un tipo del sistema de información que capta la funcionalidad del sistema y que no actúa como coordinador entre otros tipos. Los componentes interaccionan con el resto de tipos de la arquitectura a través de sus puertos, cuyo tipo es una determinada **interfaz**.

Un **sistema** es un componente que encapsula un conjunto de conectores, componentes y otros sistemas correctamente conectados, pudiendo surgir de dicha encapsulación propiedades emergentes. Los sistemas se construyen mediante una **relación de composición** de componentes.

Un **conector** es un tipo del sistema de información que actúa como coordinador entre varios tipos. Los conectores conectan y sincronizan componentes, conectores o sistemas, a través de sus roles, cuyo tipo es una determinada **interfaz**. La comunicación entre instancias de tipos se consigue mediante una **relación de conexión**.

La necesidad de modelar determinadas características del sistema relacionadas con su estado y comportamiento hace necesario que el modelo permita definir relaciones semánticas entre componentes, como la agregación y la asociación. La **agregación** ayuda a expresar la pertenencia de un tipo a otro (*part_of)*, mientras que la **asociación** permite definir la vinculación existente entre dos tipos del sistema debido a su funcionalidad y no al diseño arquitectónico.

3.2.1. Interfaces

Las interfaces son el mecanismo usado en PRISMA para establecer canales de comunicación entre los tipos. Una interfaz describe la visibilidad parcial (ver figura 4), que tienen el resto de tipos del sistema, del estado y comportamiento del tipo que define la semántica de dicha interfaz. Por lo tanto, una instancia sólo puede solicitar y consultar a otras, aquellas propiedades y servicios publicados en sus interfaces. De este modo, una interfaz actúa como mecanismo de seguridad y permite que los componentes, conectores y sistemas sean vistos como cajas negras, preservando un alto nivel de abstracción.

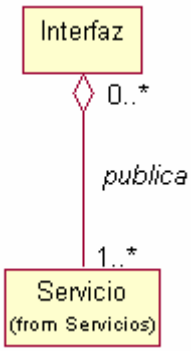

Figura 4. Paquete interfaz del metamodelo de PRISMA

Una interfaz es una signatura representada por la tupla (A, X), siendo A un conjunto de atributos y X un conjunto de servicios. El conjunto de atributos (A) son servicios de consulta que no modifican el estado, de forma que consultan el valor del atributo y lo devuelven como resultado. El conjunto de servicios (X) son servicios que modifican el estado de los atributos.

Dependiendo de si una interfaz está asociada a un puerto (ver figura 5) o a un rol (ver figura 6) de entrada o de salida de un componente o conector, la semántica de los atributos y servicios que contiene es diferente. Aquellas interfaces que definen el tipo de un puerto o rol de entrada, especifican los atributos y servicios que oferta un tipo a otros (comportamiento servidor). Mientras que aquellas interfaces que definen el tipo de un puerto o rol de salida especifican los atributos y servicios que utiliza de otros (comportamiento cliente). Por ejemplo, el servicio *incrementarsaldo(cantidad: moneda)* tiene comportamientos diferentes dependiendo de si actúa como cliente o como servidor. Cuando el servicio *incrementarsaldo(cantidad: moneda)* esta asociado a un puerto de entrada de un componente y otro componente solicita un *incrementarsaldo*, el componente que recibe la petición por su puerto de entrada (servidor) debe modificar su atributo *saldo* con la cantidad especificada. Sin embargo, cuando el servicio está asociado al puerto de salida del componente, es el componente el que solicita un *incrementarsaldo* a otro componente (cliente). En este último caso, se ha de lanzar el servicio a otro puerto en lugar de modificar el estado del atributo *saldo.*

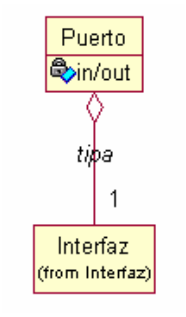

Figura 5. Paquete puerto del metamodelo de PRISMA

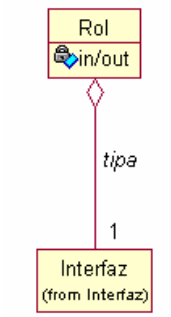

Figura 6. Paquete rol del metamodelo de PRISMA

3.2.2. Aspectos

Los aspectos se definen en [Kiz97] como propiedades de un sistema que tienden a estar presentes en varias componentes funcionales, dando lugar al término inglés *crosscutting concerns*. La distribución y sincronización son ejemplos claros de *crosscutting concerns* a nivel de implementación, ya que no están presentes en un solo componente, sino que están dispersas en varios.

En este trabajo se presenta una nueva definición de aspecto, entendiendo un aspecto como la definición completa de la estructura y el comportamiento de un tipo desde un determinado punto de vista (*concern*). Hay que tener en cuenta que dicho punto de vista puede ser cualquiera, incluso el funcional. Por lo tanto el *crosscutting* no sólo se realiza sobre propiedades funcionales sino sobre cualquier *concern.*

El hecho de que el paradigma de la orientación a aspectos (AOP) y el DSOA hayan surgido de los lenguajes de programación orientados a aspectos ya consolidados como AspectJ

[Kic01], ha ocasionado que el resto de aproximaciones abstraigan el concepto de aspecto a partir del nivel de implementación, estando claramente influenciados por dichos lenguajes. Por este motivo, éstas aproximaciones asocian siempre el concepto de aspecto con propiedades no funcionales.

Sin embargo, en el modelo se define el concepto de aspecto sin tener en cuenta estos lenguajes. Su definición se basa en las características comunes de un sistema susceptibles de ser reutilizadas. Atendiendo a esta definición, menos dependiente del lenguaje de desarrollo, la funcionalidad es susceptible de reutilización en distintos tipos e incluso dicha funcionalidad se puede subdividir en distintos aspectos. Los últimos trabajos de Ana Moreira [Brio03] empiezan a apuntar hacia esta última idea, a pesar de no haberse llevado todavía a la práctica.

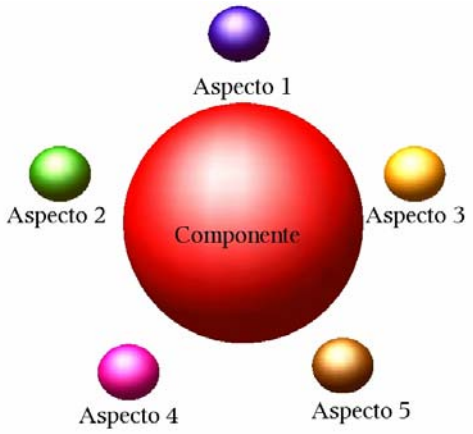

Figura 7. Aspectos asociados a un componente funcional

El cambio de la definición de aspecto proporciona un conjunto de ventajas al modelo, permitiendo que un tipo se construya mediante un operador composicional de aspectos (ver figura 8) y no mediante un tipo funcional al que se le añaden aspectos (ver figura 7). De esta manera, se consigue un modelo más uniforme, ya que los parámetros origen a partir de los cuales se construye el tipo son homogéneos. Por otro lado, se consigue una mayor reutilización y modularidad al reutilizar el aspecto funcional y considerar que puede ser común a un conjunto de tipos. Este aporte es importante, ya que la definición de [Kic97] limita la posibilidad de que existan un conjunto de tipos que tengan en común su funcionalidad y que lo que les diferencie unos de otros sean otros aspectos. Finalmente, se ha de tener en cuenta que esta definición de aspecto hace que el modelo sea completamente independiente del lenguaje de implementación, teniendo la posibilidad de compilarse tanto a lenguajes orientados a componentes como a lenguajes orientados a aspectos.

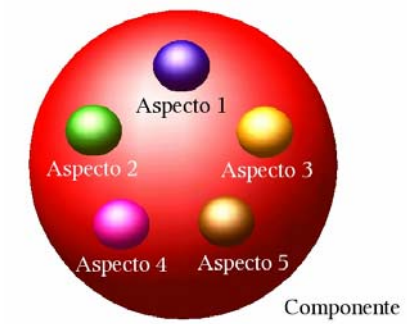

Figura 8. Aspectos que forman un componente

Un aspecto puede verse como la unión de un conjunto de interfaces que incluyen servicios y atributos del tipo del aspecto, más la especificación semántica de su estructura y comportamiento. Por lo tanto, un aspecto es una plantilla que recoge las propiedades de estructura y comportamiento compartidas por todas las instancias del componente que tengan dicho aspecto.

Un aspecto ($\Sigma_{aspecto}$) se define mediante una función de identificación (I) y una plantilla (P) (ver figura 6). De forma que dicha plantilla esta representado por la tupla (A, X ,Φ,$\prod$), siendo:

- A: El conjunto de atributos
- X: El conjunto de servicios
- Φ: El conjunto de formulas en lógica dinámica
- $\prod$: Términos en álgebra de procesos

A su vez, A y X forman la signatura o conjunto de interfaces que describen las propiedades visibles del aspecto.

Existen diferentes tipos de aspecto, definiéndose cada uno de ellos de forma independiente. El número de tipos de aspecto no esta limitado, ya que gracias al metanivel, se pueden definir nuevos. La necesidad de nuevos tipos de aspecto emerge de los requisitos de los sistemas arquitectónicos a los que se aplique PRISMA. Por lo tanto, a través de los requisitos del sistema se identificarán tanto los aspectos del sistema como las necesidades que han de cubrir cada uno de ellos [Nav03].

En este trabajo, se va a tener en cuenta aquellos aspectos relevantes para los sistemas de información hipermedia dinámicos y distribuidos: funcional, coordinación, distribución, evolución, navegación y presentación. Estos aspectos, junto con los conceptos generales de aspectos que se presentan en la figura 9, se explican con detalle en la visión de aspectos del modelo (apartado 3.3.).

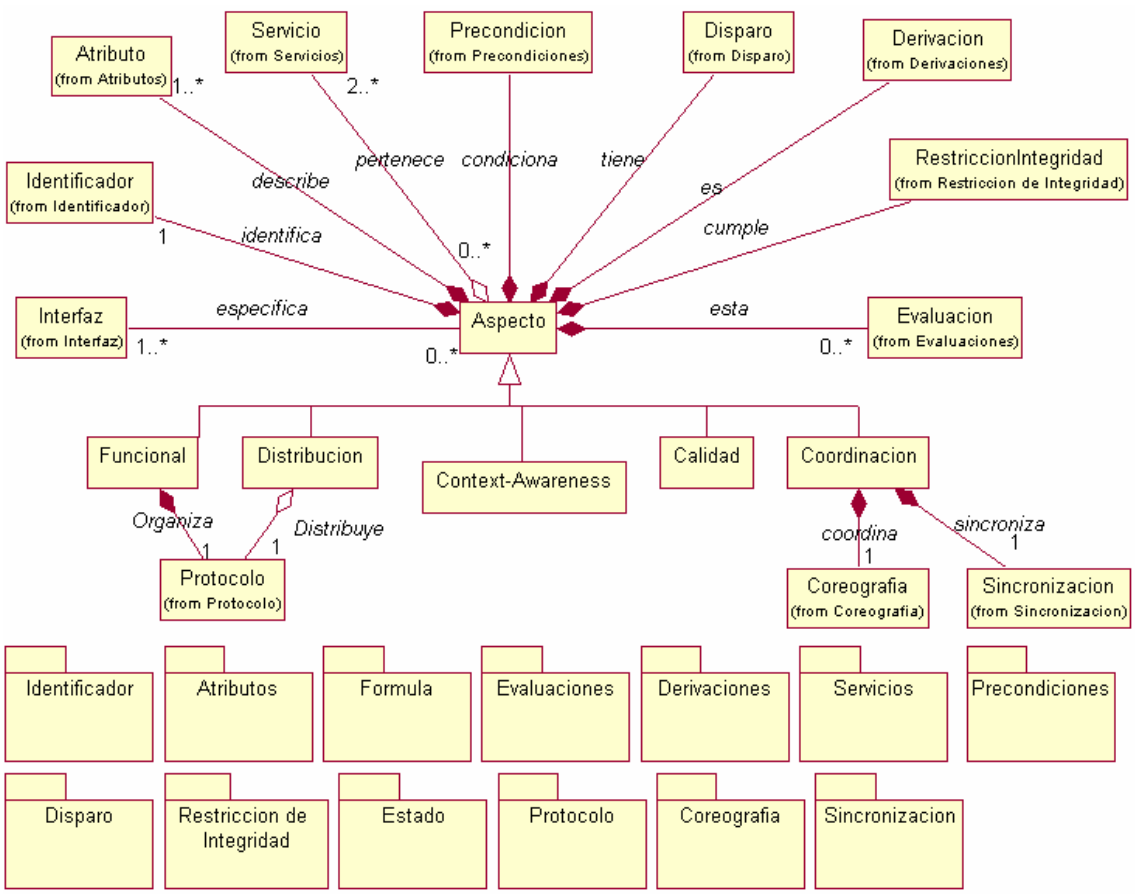

Figura 9. Paquete aspecto del metamodelo de PRISMA

3.2.3. Componentes

En la actualidad existen multitud de definiciones de componente [Sou99], [Jac97], sin embargo no se ha conseguido hasta el momento llegar al consenso de qué es un componente y cuáles deben ser sus características. Si bien es cierto, todas las definiciones de componente preservan la noción de que un componente es un artefacto que se desarrolla para su posterior reutilización. Algunas de estas definiciones están orientadas hacia la implementación y otras hacia una visión más general. La definición de componente del modelo PRISMA se enmarca dentro de este segundo grupo, ya que es una definición a un nivel alto de abstracción.

Un **componente** es un tipo del sistema de información que captura la funcionalidad del sistema y no actúa como coordinador entre otros tipos. Está formado por una función de identificación (I) , un conjunto de aspectos (funcional, distribución, etc) y por uno o más **puertos de entrada y de salida**, cuyo tipo es una determinada **interfaz** (ver figura 10).

Los puertos son aquellos artefactos que permiten la interacción de los componentes con el resto de tipos de la arquitectura. Los puertos de entrada especifican los atributos y servicios que oferta un componente a otros tipos (comportamiento servidor). Los puertos de salida especifican los atributos y servicios externos que utiliza de otros tipos (comportamiento cliente).

Existen una serie de restricciones sobre la definición de componente que se han de tener en cuenta para dotar de una semántica correcta a su definición:

- Siempre que se defina un nuevo componente PRISMA debe especificarse su aspecto funcional. Si bien es cierto, esta restricción no afecta a la importación de componentes externos (COTS).
- Un componente PRISMA nunca puede contener aspecto de coordinación.
- Un componente PRISMA nunca puede contener dos aspectos que pertenecen al mismo tipo (funcional, distribución, etc).
- El resto de aspectos únicamente se incorporarán en la definición de un nuevo componente, cuando se le quiera dotar de las propiedades que ofrece dicho aspecto.
- Los tipos de los puertos de un componente únicamente podrán ser aquellas interfaces que implementen alguno de los aspectos que formen a dicho componente.

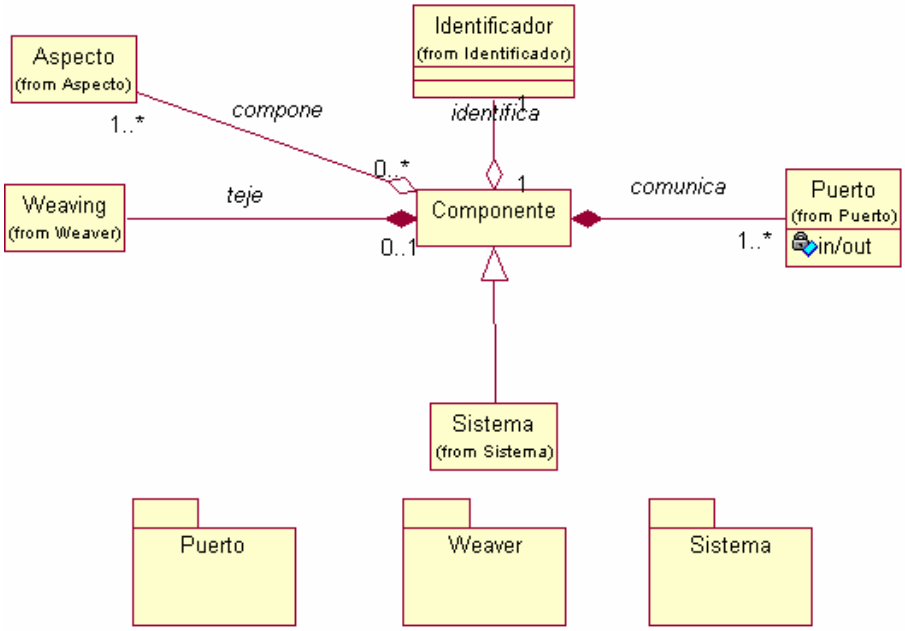

Figura 10. Paquete componente del metamodelo de PRISMA

La creación de tipos en PRISMA se realiza a través de la composición de aspectos. Para llevar a cabo dicha composición, PRISMA, mediante su operador ***weaving***, es capaz de entretejer aspectos de diferente tipo y obtener un nuevo tipo de componente. El constructor de componentes PRISMA tiene como entrada un conjunto de aspectos $\{\Sigma_{aspectoi}\}$ de distinto tipo, un conjunto de tripletas <interfaz, puerto, tipo> y la aplicación del operador ***weaving*** (ψ) a los distintos aspectos. De forma que:

- $\{\Sigma_{aspecto}\}$: Es una plantilla de aspecto.

- {<interfaz, puerto, tipo>}:
    - Siendo Interfaz la tupla <A, X>
        - A: El conjunto de atributos
        - X: El conjunto de servicios

$$\forall\ x_i \in X \rightarrow \exists\ \text{service} \in \sum_{aspectoi} \mid \text{service} = x_i$$

    - Siendo puerto el nombre del puerto
    - Siendo tipo un elemento $t \in \{in, out\}$

- Weaving (ψ):

El weaving es necesario para sincronizar los aspectos que componen un componente (ver figura 11). Esta sincronización es necesaria ya que la ejecución de un servicio en un aspecto puede desencadenar acciones en otro aspecto de la componente y a que las decisiones que se toman en un aspecto pueden estar influenciadas por el valor de los atributos de otros aspectos. Por ejemplo, el seguir un enlace del aspecto navegacional puede implicar un cambio en el aspecto de presentación de la componente.

La especificación de la relación inter-aspectos se realiza en el componente mediante el weaving. De este modo se preserva la independencia y reutilización de los aspectos y se aumenta la flexibilidad del modelo. La independencia y reutilización se consiguen al no definir dentro de cada aspecto su sincronización con el resto, ya que no se establecen referencias que generen dependencias con otros aspectos. Mientras que la flexibilidad se aumenta debido a que es posible establecer un comportamiento diferente en componentes que estén formados por los mismos aspectos, definiendo distintas sincronizaciones inter-aspectos. Por ejemplo, la invocación de un link del aspecto navegacional de un componente se puede sincronizar con un servicio que realice un cambio en el aspecto de presentación del componente; sin embargo, en otro componente, que el servicio que se desencadene en el aspecto de presentación sea diferente, obteniendo en cada caso una presentación de la información en un formato diferente.

La sincronización entre servicios PRISMA de diferentes aspectos se puede realizar de tres formas diferentes: ***after***, ***before*** o ***around***. Estas tres formas de sincronizar los servicios inter-aspectos son las que propone la programación orientada a aspectos. PRISMA mantiene los tres modos de sincronización, no sólo por seguir el enfoque orientado a aspectos, también porque cubren todas las necesidades de sincronización inter-aspecto que el modelo PRISMA necesita.

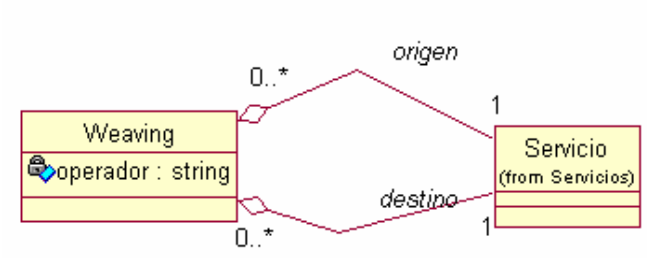

Figura 11. Paquete weaver del metamodelo de PRISMA

La operación weaving se define tal y como se muestra a continuación:

ψ ($\Sigma_{aspectoi}$.service$_x$, $\Sigma_{aspectoj}$.service$_y$) | ψ = {after, before, around}

De forma que Σaspectoi, Σaspectoj son dos aspectos de diferente tipo (funcional, distribución, etc) y la semántica del weaving es la que se muestra a continuación

after: $\Sigma_{aspectoi}$.service$_x$ se ejecuta después del $\Sigma_{aspectoj}$.service$_y$

before: $\Sigma_{aspectoi}$.service$_x$ se ejecuta antes del $\Sigma_{aspectoj}$.service$_y$

around: $\Sigma_{aspectoi}$.service$_x$ se ejecuta en lugar del $\Sigma_{aspectoj}$.service$_y$

Un componente puede tener dos vistas diferentes dentro del modelo PRISMA (ver figura 12): una visión interna, en la que un componente se ve como un prisma con tantas caras como aspectos considere, y una visión externa que es una caja negra fuertemente encapsulada dentro de un modelo de arquitectura y que interactúa con el resto del sistema a través de sus puertos.

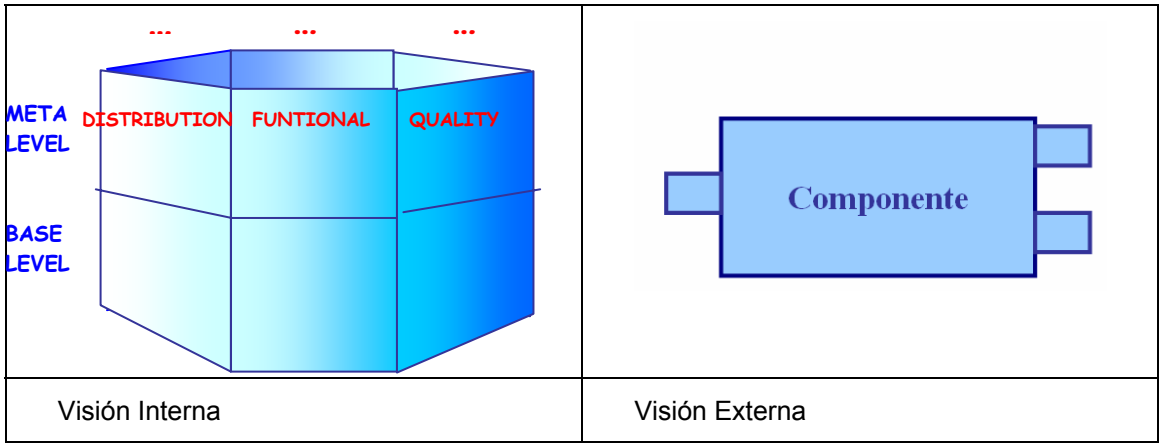

Figura 12. Vistas de un componente PRISMA

Un componente elemental en un caso de estudio puede verse como una parte del sistema que no se puede disgregar en partes más simples. Un ejemplo, en el caso de la subasta electrónica, es el de los productos, siendo *producto* una componente elemental del sistema.

3.2.4. Conectores

Según un artículo de [Sha94], el concepto de conector emerge y está justificado cuando permite una separación de intereses, de forma que un componente o sistema especifica la funcionalidad y un conector tan solo especifica la interacción entre tipos. Esta separación entre funcionalidad e interacción aumenta el nivel de abstracción, la modularidad y la visión arquitectónica del sistema de información. El modelo PRISMA incorpora el conector como tipo de su modelo por las ventajas que aporta esta separación de intereses.

Un **conector** PRISMA es un tipo del sistema de información que actúa como coordinador entre varios tipos (ver figura 13). Se compone por una función de identificación (I), un conjunto de aspectos (coordinación, distribución, etc) y por uno o más **roles de entrada y salida** que tienen como tipo una determinada **interfaz**.

Los conectores se encargan de conectar y sincronizar componentes, conectores o sistemas a través de sus roles. Los roles son aquellos artefactos que permiten la interacción de los conectores con el resto de tipos de la arquitectura. Los roles de entrada especifican los atributos y servicios que oferta el conector a otros tipos (comportamiento servidor). Los roles de salida especifican los atributos y servicios externos que utiliza de otros tipos (comportamiento cliente).

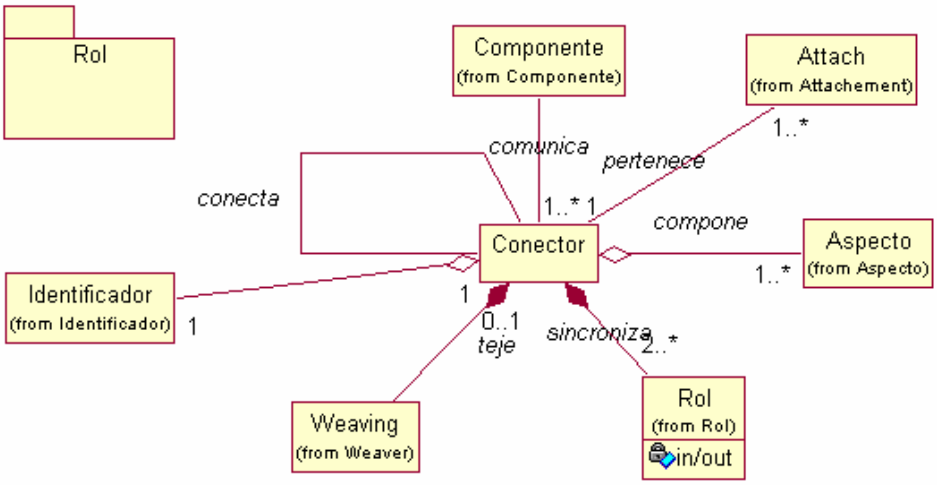

Figure 13. Paquete conector del metamodelo de PRISMA

Existen una serie de restricciones sobre la definición de conector que se han de tener en cuenta para dotar de una semántica correcta a su definición:

- Siempre que se defina un nuevo conector PRISMA debe especificarse su aspecto de coordinación.
- Un conector PRISMA nunca puede contener aspecto funcional.
- Un conector PRISMA nunca puede tener dos aspectos del mismo tipo (coordinación, distribución, etc).

- El resto de aspectos únicamente se incorporarán en la definición de un nuevo conector, cuando se le quiera dotar de las propiedades que ofrece dicho aspecto.
- Los tipos de los roles de un conector únicamente podrán ser aquellas interfaces que implementen alguno de los aspectos que formen a dicho conector.
- Una instancia de conector al menos ha de conectar dos instancias de tipos PRISMA.

Robert Allen y David Garlan en su trabajo [All97] explicitan que un conector debe tener las siguientes tres propiedades básicas:

- Un conector debe poder especificar las interacciones arquitectónicas más comunes como llamadas a evento, tuberías, eventos de broadcast y variables compartidas. PRISMA permite definir cada una de estas interacciones del siguiente modo:
    - Las llamadas a evento se especifican mediante la petición de un servicio por un puerto o rol de salida.
    - Las tuberías en PRISMA se especifican mediante las sincronizaciones del aspecto de coordinación del conector, haciendo que el servicio de entrada desencadene una petición de un servicio por uno de los roles de salida del conector.
    - Las llamadas a evento de broadcast, al igual que las tuberías, se especifican mediante las sincronizaciones del aspecto de coordinación del conector, haciendo que el servicio de entrada desencadene una petición de un servicio por cada uno de los roles de salida del conector
    - Finalmente, las variables compartidas PRISMA son los atributos del aspecto de coordinación del conector.
- Un conector debe poder describir interacciones complejas entre componentes. Por ejemplo: especificación de un protocolo preciso dentro de una arquitectura cliente servidor. PRISMA permite definir una interacción compleja como la cliente-servidor mediante las sincronizaciones y la coreografía del aspecto de coordinación del conector. De forma que la coreografía sólo permitirá solicitudes por parte del cliente y contestaciones por parte del servidor, mediante restricciones en la sincronización de los roles a los que estén conectados cada uno de ellos.
- Un conector debe permitir pequeñas variaciones en su comportamiento. Por ejemplo, asociar diferentes condiciones a una variable compartida. PRISMA soporta las variaciones del comportamiento del conector de dos maneras diferentes:
    - El metamodelo permitirá la evolución y por tanto, el cambio de comportamiento, tanto del conector como de los tipos que lo forman.
    - Las precondiciones y postcondiciones asociadas a los servicios y definidas en base a los atributos de los aspectos, harán posible la variación del

comportamiento dependiendo del valor de los atributos que participen en dichas precondiciones y postcondiciones.

La creación de tipos en PRISMA se realiza a través de la composición de aspectos. Para llevar a cabo dicha composición, PRISMA, mediante su operador **weaving,** es capaz de entretejer aspectos de diferente tipo y obtener un nuevo tipo de conector. El constructor de conectores PRISMA tiene como entrada un conjunto de plantillas de aspecto ($\{\Sigma_{aspectoi}\}$) de distinto tipo, un conjunto de tripletas { <interfaz, rol, tipo>} y la aplicación del operador **weaving** (ψ) a los distintos aspectos. De forma que:

- $\{\Sigma_{aspecto}\}$: Es una plantilla de aspecto
  - {<interfaz, rol, tipo>}:
  - Siendo Interfaz la tupla <A, X>
    - A: El conjunto de atributos
    - X: El conjunto de eventos | $\forall\ x_i \in X \rightarrow \exists\ \Sigma_{aspectoi}.service = x_i$
  - Siendo rol el nombre del rol
  - Siendo tipo un elemento $t \in \{in, out\}$

- Weaving (ψ):

La operación weaving se define tal y como se muestra a continuación:

$\psi\ (\Sigma_{aspectoi}.service_x, \Sigma_{aspectoj}.service_y) \mid \psi = \{after, before, around\}$

De forma que $\Sigma_{aspectoi}$, $\Sigma_{aspectoj}$ son dos aspectos de diferente tipo (coordinación, distribución, etc) y la semántica del weaving es la que se muestra a continuación:

after: $\Sigma_{aspectoi}.service_x$ se ejecuta después del $\Sigma_{aspectoj}.service_y$

before: $\Sigma_{aspectoi}.service_x$ se ejecuta antes del $\Sigma_{aspectoj}.service_y$

around: $\Sigma_{aspectoi}.service_x$ se ejecuta en lugar del $\Sigma_{aspectoj}.service_y$

Un conector puede tener dos vistas diferentes dentro del modelo PRISMA (ver figura 14): una visión interna, en la que un conector se ve como un prisma con tantas caras como aspectos considere, y una visión externa que es una caja negra fuertemente encapsulada dentro de un modelo de arquitectura y que interactúa con el resto del sistema a través de sus roles.

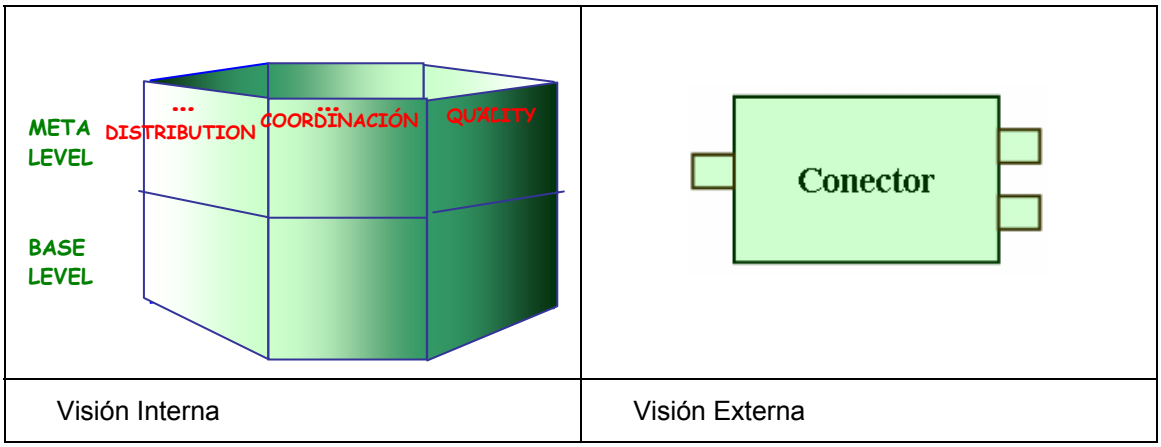

Figura 14. Vistas de un connector PRISMA

Un ejemplo de conector dentro del caso de estudio de la subasta electrónica podría ser el gestor de la subasta que comunica el pujador y el producto.

3.2.5. Sistemas

En la mayoría de modelos arquitectónicos, surge la necesidad de tener mecanismos de abstracción que permitan tener elementos de mayor granularidad, aumentando la modularidad, composición y reutilización de sus modelos arquitectónicos. Los sistemas son los tipos que dotan de estas ventajas a PRISMA.

El concepto de sistema se incorpora en otros modelos como el de Ivar Jacobson [Jac97], en el que se define el concepto de componente subsistema como un conjunto de componentes organizados. También aparece en lenguajes como DURRA [Bar91], un lenguaje de descripción de tareas para el desarrollo de aplicaciones distribuidas, en el que el concepto de sistema aparece permitiendo que se componga por subcomponentes unidos por una serie de estructuras internas. Darwin [Mag95] también incluye el concepto de sistema, estableciendo la diferencia entre componentes primitivos y componentes compuestos, siendo los componentes compuestos aquellos que se construyen jerárquicamente a partir de otros.

Un **sistema PRISMA** es un tipo que se va a utilizar como un patrón a la hora de definir una configuración especifica de un modelo arquitectónico. La diferencia entre un sistema y otros tipos, es que un sistema es un tipo compuesto y el resto son simples. Por lo tanto, los sistemas PRISMA permiten que existan tipos con diferentes niveles de granularidad.

Un sistema es un componente formado por una función de identificación (I) y un conjunto de tipos jugando un papel determinado dentro del sistema (**elementos**), que a su vez pueden estar o no conectados entre sí (ver figura 15). De dicha composición pueden surgir propiedades emergentes propias del sistema. La especificación de la conexión entre los elementos que incluye el sistema y el propio sistema se realiza mediante los **bindings**, que son los enlaces entre los puertos del sistema y los puertos o roles de los elementos que encapsula. Estos bindings permiten mantener un enlace entre los distintos niveles de granularidad de los tipos.

Existen una serie de restricciones sobre la definición de sistema que se han de tener en cuenta para dotar de una semántica correcta a su definición:

- Siempre que se defina un nuevo sistema PRISMA ha de contener su aspecto funcional. Si bien es cierto, esta restricción no afecta a la importación de componentes externos (COTS).
- Un sistema PRISMA nunca puede contener aspecto de coordinación.
- Un sistema PRISMA nunca puede contener dos aspectos que pertenecen al mismo tipo (funcional, distribución, etc).
- El resto de aspectos únicamente se incorporarán en la definición de un nuevo sistema, cuando se le quiera dotar de las propiedades que ofrece dicho aspecto.
- Los tipos de los puertos de un sistema únicamente podrán ser aquellas interfaces que implementen alguno de los aspectos que formen a dicho sistema.

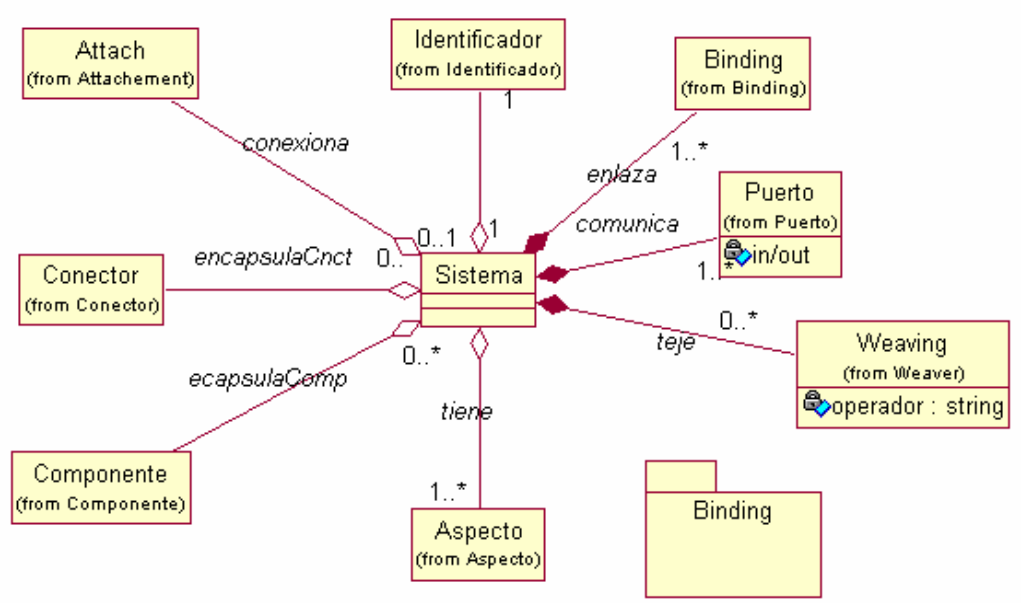

Figura 15. Paquete sistema del metamodelo de PRISMA

Hay que hacer hincapié en el concepto de elemento, ya que es el que permite el diseño de patrones reutilizables en diferentes modelos arquitectónicos. Este concepto se ubica en un nivel de abstracción intermedio entre los tipos y las instancias. Los elementos permiten que dentro de un patrón de diseño se pueda poner más de un miembro del mismo tipo sin tener que instanciarlo. Simplemente es necesario decir el número y el lugar que ocupan cada uno dentro del patrón, es decir, con quién está conectado.

Un ejemplo de elementos son las *ruedas* que componen el sistema *coche*, no siendo ni tipos ni instancias. No son tipos, ya que necesitamos decir que vamos a utilizar cuatro tipos ruedas y con quién se han de comunicar cada uno de ellos para realizar una correcta gestión del coche. Por otro lado, tampoco son instancias, porque no se van a dar valor a sus atributos hasta que no se importe el sistema como instancia dentro de un modelo arquitectónico específico.

El constructor de sistemas PRISMA es el **operador de composición** tiene como entrada la tupla <{<interfaz, puerto, tipo>}, {elementos}, {bindings}, [{conexiones_elementos}], [tipo] >. De forma que:

- {<interfaz, puerto, tipo>}:
    - Siendo Interfaz la tupla <A, X>
        - A: El conjunto de atributos
        - X: El conjunto de eventos | $\forall\ x_i \in X \rightarrow \exists\ \sum_{aspectoi}.service= x_i \vee Elemento.\sum_{aspectoi}.service= x_i$
- Siendo puerto el nombre del puerto
- Siendo tipo un elemento $t \in$ {in, out}
- {Elementos}: Siendo Elementos el conjunto cuyos miembros se definen por la tupla <tipo, cardinalidad> De forma que:
    - Tipo: Es el tipo del elemento
    - Cardinalidad: Número de elementos del tipo que participan ó pueden participar en el patrón.
- {Bindings}: Siendo el conjunto de relaciones **bind** (ver siguiente apartado) entre el sistema y los elementos que compone.
- [{conexiones_elementos}]: Este parámetro es opcional, ya que solamente puede darse cuando hay elementos de tipo conector dentro del sistema. Las conexiones entre elementos son el conjunto de relaciones **attach** (ver siguiente apartado) entre elementos componente y elementos conector que forman parte del sistema.
- [Tipo]:

  La creación de tipos en PRISMA se realiza a través de la composición de aspectos, mediante su operador **weaving**. En el caso de los sistemas, sus aspectos surgen de la unión de los aspectos de los tipos que contiene, excepto el aspecto de coordinación que no forma parte del tipo sistema, ya que este tipo no actúa como coordinador de tipos.

  Además de los aspectos que emergen de la composición de elementos, se pueden definir los aspectos propios del sistema, esto es lo que se define en el argumento [tipo]. Su carácter opcional se debe a que, tal y como se ha mostrado anteriormente, la composición del sistema genera ya sus aspectos y sólo será necesario especificar este parámetro cuando el sistema tenga propiedades y servicios propios y no de los tipos que compone. Aquellos aspectos propios del sistema que no hayan sido considerados en ninguno de los tipos que encapsule, se sumaran al resto de tipos de aspectos del sistema. Sin embargo, aquellos que ya tenga el sistema se aplicara la unión de su

plantilla con la plantilla generada por la función de composición. El argumento [tipo] se define mediante la tupla < [{$\Sigma_{aspectoi}$}], [weaving (ψ)] >. De forma que:

- {$\Sigma_{aspectoi}$}: Es la plantilla de aspecto
- Weaving (ψ):

La operación weaving se define tal y como se muestra a continuación:

ψ ($\Sigma_{aspectoi}$.service$_x$, $\Sigma_{aspectoj}$.service$_y$) | ψ = {after, before, around}

De forma que $\Sigma_{aspectoi}$, $\Sigma_{aspectoj}$ son dos aspectos de diferente tipo (coordinación, distribución, etc) y la semántica del weaving es la que se muestra a continuación:

after: $\Sigma_{aspectoi}$.service$_x$ se ejecuta después del $\Sigma_{aspectoj}$.service$_y$

before: $\Sigma_{aspectoi}$.service$_x$ se ejecuta antes del $\Sigma_{aspectoj}$.service$_y$

around: $\Sigma_{aspectoi}$.service$_x$ se ejecuta en lugar del $\Sigma_{aspectoj}$.service$_y$

De esta forma, se ve que un sistema tiene todas las propiedades y servicios de sus componentes más las suyas propias.

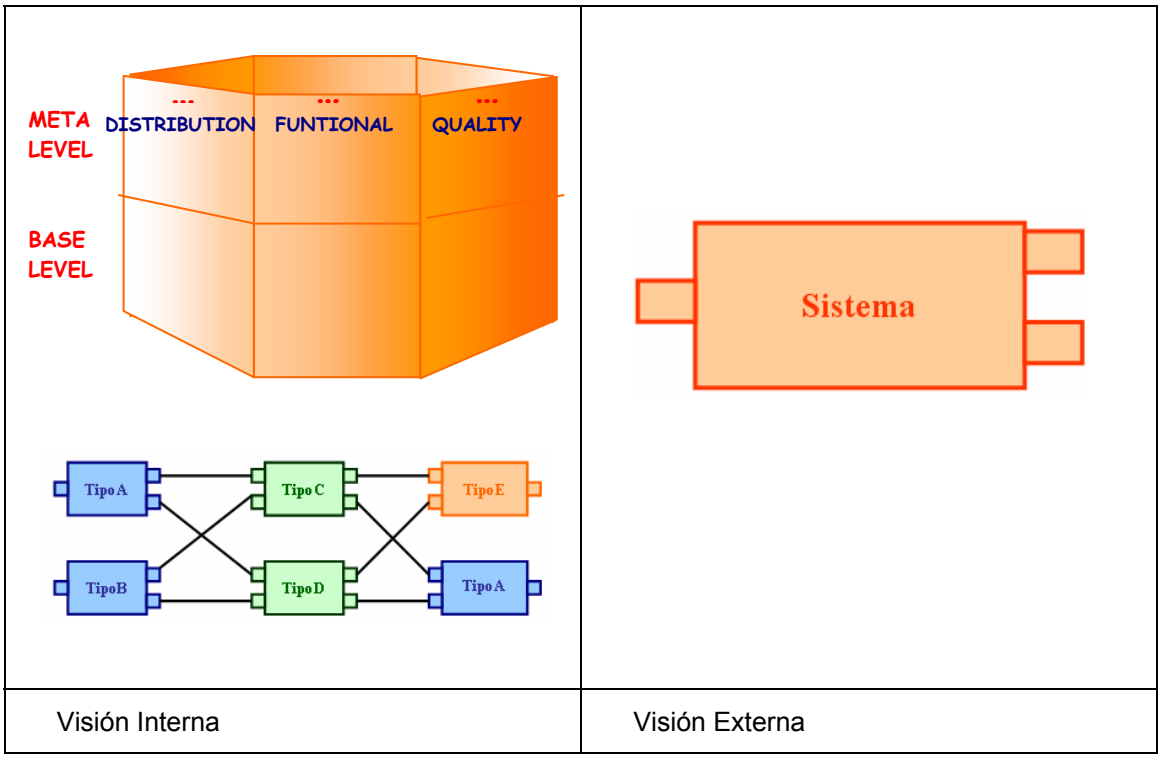

Figura 16. Vistas de un componente PRISMA

Un sistema puede tener dos vistas diferentes dentro del modelo PRISMA (ver figura 16): una visión interna, y una visión externa. A su vez la visión interna del sistema se divide en dos vistas diferentes, un prisma con tantas caras como aspectos considere el sistema y un conjunto de elementos interconectados entre si. La visión externa de un sistema es una caja negra fuertemente encapsulada dentro de un modelo de arquitectura y que interactúa con el resto del sistema a través de sus puertos.

Un ejemplo de sistema dentro del caso de estudio de la subasta electrónica podría ser el de la subasta, siendo sus elementos el subastador, los pujadores, el producto y el conector que gestiona la puja.

3.2.6. Relaciones

En PRISMA existen dos tipos de relaciones diferentes, aquellas que nos permiten definir la topología del sistema y las que nos permiten definir la semántica del sistema de información. Las primeras, a las que se les ha dado el nombre en PRISMA de **relaciones topológicas**, aparecen en la mayoría de los modelos de arquitectura y por lo tanto, en sus lenguajes de definición de arquitecturas. Sin embargo las segundas, **relaciones conceptuales**, aparecen en los modelos conceptuales y no en los arquitectónicos. El motivo de que la mayoría de modelos arquitectónicos no contemplen la posibilidad de especificar relaciones conceptuales es que los ámbitos de aplicación de dichos lenguajes arquitectónicos se limitan a sistemas de comunicación que ni tan si quiera tienen estado, tan solo son procesos que tienen una entrada y producen una salida. Algunos ejemplos extendidos que muestran la aplicación de los lenguajes de definición de arquitecturas son: sistemas para pasar letras de minúscula a mayúscula [All97], sistemas de filtrado y tuberías de sistemas arquitectónicos clásicos a bajo nivel [Loq00], etc).

Sin embargo, PRISMA pretende abarcar un ámbito de aplicación mayor, cubriendo también sistemas abiertos y activos como sistemas hipermedia de subastas electrónicas y museos híbridos [Jae03b], sistemas de tele-operación, etc. Estos sistemas tienen la necesidad de dar semántica a los distintos tipos de su sistema definiendo su estado, comportamiento y relaciones con otros tipos. Por este motivo, PRISMA tiene la necesidad de tener un mecanismo para especificar ese tipo de relaciones conceptuales entre los tipos de sus modelos arquitectónicos. Un ejemplo de esta necesidad en el caso de estudio de la subasta electrónica, es la relación de asociación entre el vendedor y el producto. Es necesario registrar en el sistema a qué vendedor pertenece un determinado producto, a pesar de que entre ellos no exista ninguna necesidad de comunicación.

Una diferencia básica entre las relaciones topológicas y las relaciones conceptuales PRISMA es el nivel de abstracción al que se definen, ya que las primeras se definen a nivel de diseño, a no ser que se utilicen para construir sistemas, y las segundas a nivel de análisis. A continuación se definen las diferentes relaciones entre elementos (sistemas) e instancias PRISMA.

3.2.6.1. Relaciones topológicas

- **Relación de conexión**

Las relaciones de conexión se definen a nivel de diseño, excepto en el caso en el que se utilicen para construir sistemas, que se establecen a nivel de análisis. Una relación de conexión establece un canal de conexión entre un conjunto de instancias (diseño) o elementos (análisis) a través de una instancia o elemento conector respectivamente. La relación de conexión (**attach**) se establece a partir de los puertos y los roles de las instancias o elementos que participan en la comunicación.

La relación **attach (>> | <<)** de PRISMA tiene como entrada un par <puerto, identificador_componente | tipo_componente> y un par <rol, identificador_conector | tipo_conector> (ver figura 17). Se ha de tener en cuenta que el identificador_componente o tipo_componente puede ser una instancia o elemento de un componente o sistema. De forma que:

- **Attach** (<puerto, identificador_componente | tipo_componente > **>>** <rol, identificador_conector | tipo_conector >) establece un canal de comunicación entre el puerto del componente y el rol del conector. El operador >> establece la dirección en la que fluye la información entre el puerto y el rol, de forma que el puerto es de salida y el del rol es de entrada.

- **Attach** (<puerto, identificador_componente | tipo_componente > **<<** <rol, identificador_conector | tipo_conector >) establece un canal de comunicación entre el puerto del componente y el rol del conector. El operador << establece la dirección en la que fluye la información entre el puerto y el rol, de forma que el puerto es de entrada y el del rol es de salida.

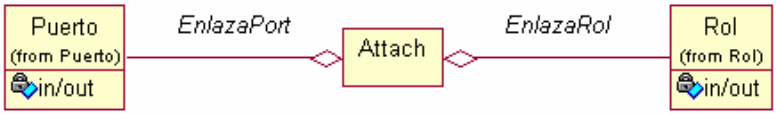

Figura 17. Paquete attachment del metamodelo PRISMA

- **Relación de encapsulación**

Las relaciones de encapsulación se definen a nivel de diseño, excepto en el caso en el que se utilicen para construir sistemas, que se establecen a nivel de análisis. La relación de encapsulación (**bind**) establece la conexión entre una instancia (diseño) o elemento (análisis) encapsulado y la instancia o elemento del sistema que lo encapsula (ver figura 18).

La relación **Bind (→ | ←)** de PRISMA tiene como entrada un par <puerto, identificador_componente | tipo_componente > y un par <puerto, identificador_sistema | tipo_sistema >. Se ha de tener en cuenta que el identificador_sistema o tipo_sistema puede ser una instancia o elemento de componente o sistema. De forma que:

- **Bind** (<puerto, identificador_componente | tipo_componente > → <puerto, identificador_sistema | tipo_sistema >) establece un canal de comunicación entre el puerto del componente o sistema encapsulado y el puerto del sistema que lo encapsula. El operador → establece la dirección en la que fluye la información entre el encapsulado y encapsulador, de forma que los puertos de ambos son de salida.

- **Bind** (<puerto, identificador_componente | tipo_componente > ← <puerto, identificador_sistema | tipo_sistema >) establece un canal de comunicación entre el puerto del componente o sistema encapsulado y el puerto del sistema que lo encapsula. El operador ← establece la dirección en la que fluye la información entre el encapsulado y encapsulador, de forma que los puertos de ambos son de entrada.

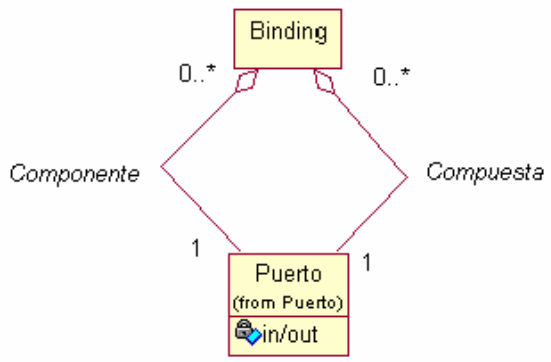

Figura 18. Paquete binding del metamodelo PRISMA

3.2.6.2. Relaciones conceptuales

- **Relación de agregación**

Las relaciones de agregación se definen a nivel de análisis, nos ayudan a decir que un componente pertenece a otro de forma estructural, representando la noción parte_de (**part_of)** (ver figura 19).

La relación **Part_of** de PRISMA se define con la tupla <componente_agregado, componente_agregador, estatica | dinamica, inclusiva | relacional>. Teniendo en cuenta que a su vez los elementos componente_agregado y componente agregador se definen por la tripleta <nombre, card_min, card_max>. Siendo:

- Componente_agregado: Componente que forma parte_de del componente agregador
    - Nombre
    - Card_min: Número mínimo de instancias del componente que deben formar parte de una instancia del componente agregador.
    - Card_max: Número máximo de instancias del componente que pueden formar parte de una instancia del componente agregador.

- Componente_agregador: Componente que agrega al componente agregado.
    - Nombre
    - Card_min: Número mínimo de instancias del componente que deben ser parte de una instancia del componente agregado.
    - Card_max: Número máximo de instancias del componente que pueden ser parte de una instancia del componente agregado.

- Estática | Dinámica:

Una relación de agregación **estática** es aquella en la que las instancias que forman parte del componente no pueden ser ni eliminadas ni insertadas durante su ciclo de vida. Por lo tanto, las instancias que forman parte del componente quedan predeterminados en el momento de su creación.

Una relación de agregación **dinámica** es aquella en la que las instancias que forman parte del componente pueden ser eliminadas e insertadas durante su ciclo de vida.

- Inclusiva | Relacional:

Una relación de agregación **inclusiva** es aquella en la que las comunicaciones entre las instancias del componente y otras instancias sólo se pueden establecer a través del componente.

Existe una relación muy estrecha entre una relación de agregación inclusiva y una relación de encapsulación, ya que ambas permiten construir un tipo de mayor granularidad, que encapsula a otros tipos de granularidad inferior formando un todo. Sin embargo, entre estas relaciones existen claras diferencias. En primer lugar, son relaciones de diversa índole, ya que son conceptuales y otras de conexión. En segundo y último lugar, la relación de encapsulación permite que sus elementos estén conectados entre si vía conectores, mientras que una relación de agregación inclusiva no.

Una relación de agregación **relacional** es aquella en la que las instancias que forman parte del componente pueden comunicarse directamente con otras, sin necesidad de que sea a través del componente agregador.

- **Relación de asociación**

Las relaciones de asociación se definen a nivel de análisis, nos permiten definir la vinculación existente entre dos componentes del sistema debido a la funcionalidad del sistema y no a la conexión del sistema arquitectónico (ver figura 19). Una relación de asociación siempre vincula dos componentes de forma dinámica y relacional.

La relación **asociación** de PRISMA se define a través de dos componentes, teniendo en cuenta que cada uno de ellos se definen mediante la tripleta <nombre, card_min, card_max>. Siendo:

- Nombre
- Card_min: Número mínimo de instancias de un componente que se relacionan con el otro
- Card_max: Número máximo de instancias de un componente que se relacionan con el otro

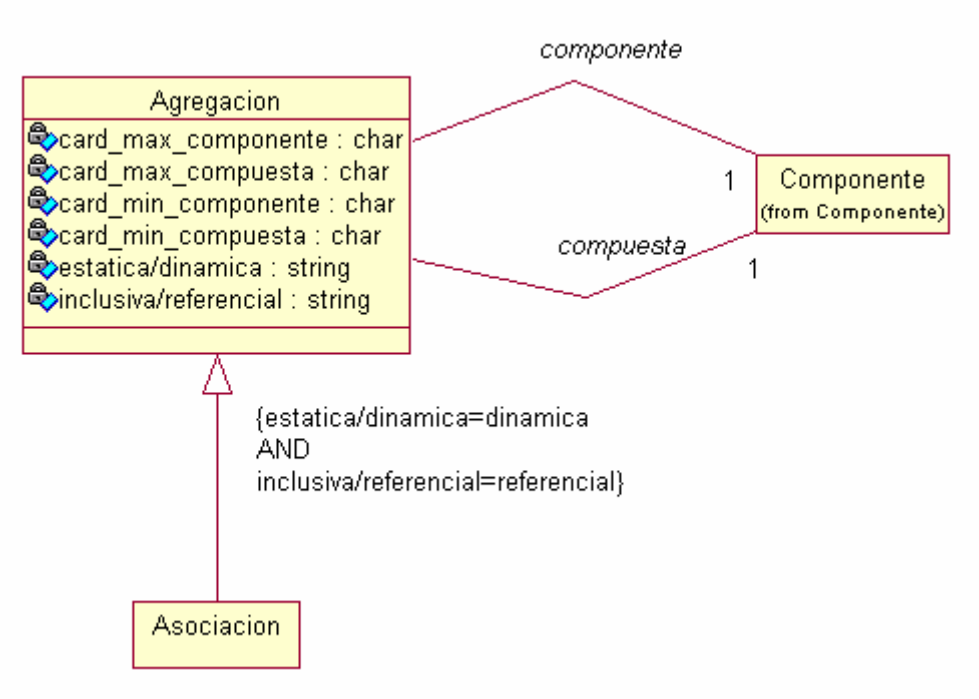

Figura 19. Paquete agregación del metamodelo PRISMA

3.2.7. Modelo Arquitectónico

Un modelo arquitectónico se construye mediante la instanciación de un conjunto de tipos. En el caso de la construcción de un modelo arquitectónico haciendo uso de patrones (sistemas) no sólo hay que instanciar los elementos que lo forman, también las interconexiones entre sus elementos.

Un modelo arquitectónico es un conjunto de instancias de componentes y conectores interconectadas entre si, definido por la tupla < {instancias}, {attachments}> (ver figura 20). Por este motivo, una vez instanciados los tipos se han de establecer las conexiones entre las instancias, las relaciones de attach entre instancias.

- {instancias}: Conjunto de instancias

$\forall$ x $\in$ {instancias} $\rightarrow$ (x **is_a** componente PRISMA) v (x **is_a** conector PRISMA) v (x **is_a** sistema PRISMA)

- {attachments}: conjunto de relaciones attach entre las instancias. De forma que la aplicación de la función se restringe únicamente a instancias del siguiente modo:

- **Attach** (<puerto, identificador_componente> **>>** <rol, identificador_conector >)

- **Attach** (<puerto, identificador_componente > **<<** <rol, identificador_conector>)

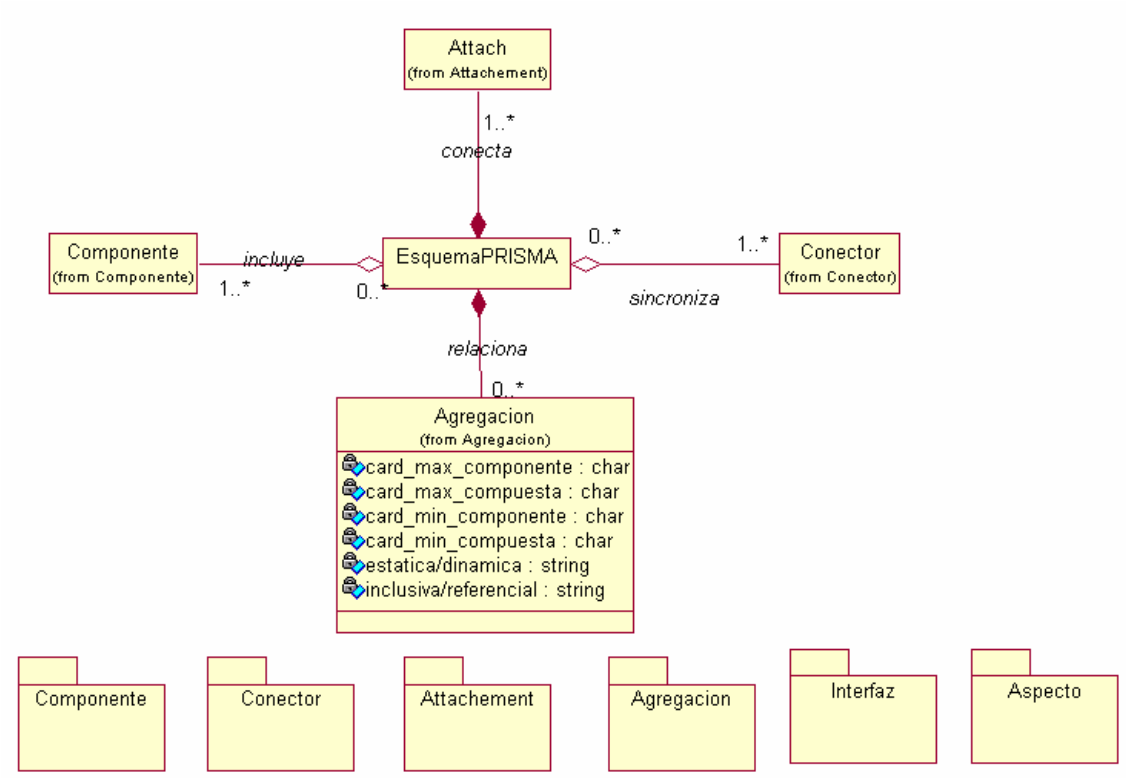

Figura 20. Paquete esquema del metamodelo PRISMA

3.3. Visión basada en aspectos

Los tipos del modelo arquitectónico PRISMA están compuestos por aspectos. En este apartado se introducen los distintos aspectos que pueden formar un tipo PRISMA y se explica qué incluye cada uno ellos. Además de los aspectos relevantes en el área de los sistemas hipermedia, concretamente del ámbito de las subastas electrónicas, se presentan otros aspectos PRISMA que se han aplicado en otras áreas de interés como los sistemas de teleoperación [Per03b] y los museos híbridos [Jae03b].

Se ha de tener en cuenta que en PRISMA, la importación de COTS se expresa gracias al carácter opcional de los aspectos. Esta opcionalidad permite ver a los tipos como cajas negras y con ello la posibilidad de incorporar componentes COTS.

3.3.1. Conceptos Generales de Aspectos

Existe un conjunto de conceptos que forman un aspecto de forma general, además de aquellos propios de cada aspecto. En este apartado se presentan los conceptos que constituyen un aspecto independientemente de cual sea su tipo.

El **identificador** del aspecto identifica inequívocamente a un aspecto, no pudiendo existir dos aspectos con el mismo identificador (ver figura 21). El identificador del aspecto puede ser tanto un atributo como una lista de atributos.

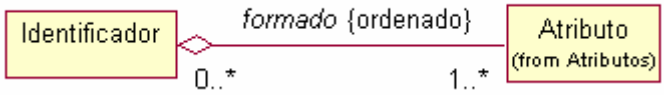

Figura 21. Paquete identificador del metamodelo PRISMA

Las propiedades del tipo vienen representadas en el aspecto mediante **atributos** (ver figura 22), cuyo valor es el estado del tipo.

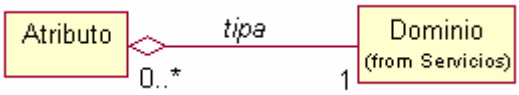

Figura 22. Paquete atributos del metamodelo PRISMA

Existen cambios de estado fuerte o débil, dependiendo si consiste en la creación o destrucción del aspecto funcional o en la modificación del valor de sus atributos. El cambio de estado funcional de un tipo se conoce por el término **servicio** (ver figura 23). Un servicio puede necesitar una serie de argumentos para ejecutarse correctamente, en cuyo caso cada uno de ellos deben pertenecer a un determinado dominio.

Un servicio puede ser elemental o no, cuando no es elemental se trata de una **operación**. A diferencia de un servicio elemental, una operación esta compuesta por un conjunto de servicios. Los servicios elementales pueden ser de creación, destrucción o modificación del valor de los atributos. En el caso en el que el conjunto de servicios de una operación se comportan como una unidad de ejecución de mayor granularidad, de forma que no es posible observar sus estados intermedios, ni ejecutar parcialmente, dicha operación es una **transacción**.

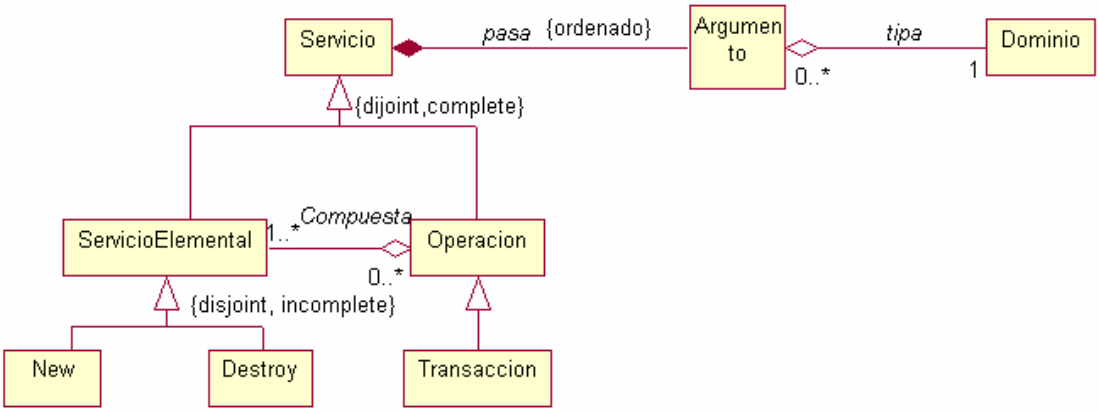

Figura 23. Paquete servicios del metamodelo PRISMA

El modo en que afecta la ejecución de un servicio a un atributo se establece mediante **evaluaciones** (ver figura 24). Una evaluación puede especificar una condición a cerca del estado de un determinado atributo, de forma que si se cumple la condición, el resultado especifica el valor de ese mismo atributo después de la ejecución del servicio bajo dicha condición. En caso de que no se especifique condición, simplemente se establece el resultado que produce la ejecución de un servicio sobre un determinado atributo, independientemente del estado anterior a la ejecución.

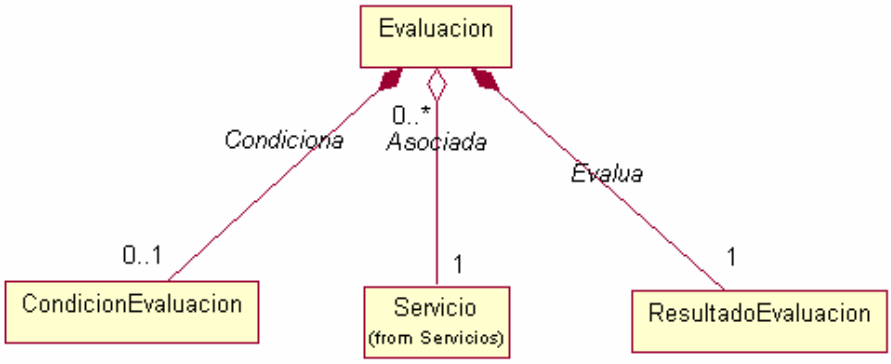

Figura 24. Paquete evaluaciones del metamodelo PRISMA

La ocurrencia de un servicio sólo tiene éxito si se cumple la **precondición** asociada a dicho servicio (ver figura 25). La precondición es una condición asociada a determinados atributos.

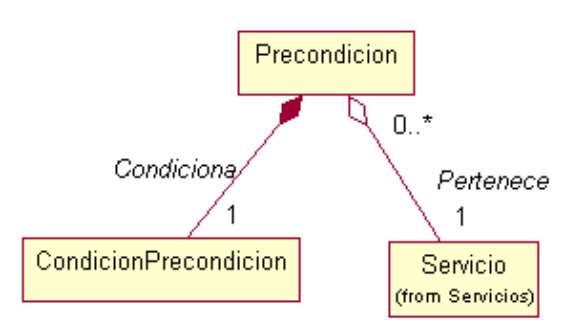

Figura 25. Paquete precondiciones del metamodelo PRISMA

Además, no se permiten todos los estados posibles ya que es posible especificar **restricciones de integridad**, entendidas como valores permitidos para los atributos del aspecto funcional (ver figura 26). Las restricciones se pueden clasificar en estáticas y dinámicas, dependiendo de si se refieren sólo a un estado o relacionan a diferentes estados, respectivamente.

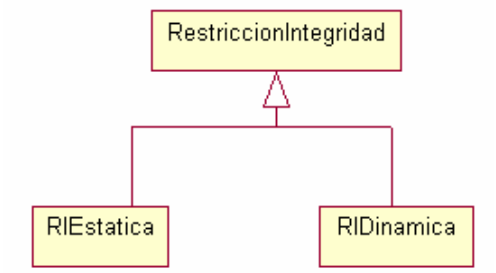

Figura 26. Paquete restricciones de integridad del metamodelo PRISMA

En el aspecto funcional se da la posibilidad de especificar **triggers**, entendidos como el disparo de servicios por parte del tipo que contiene el aspecto (ver figura 27). Dicho tipo actúa como agente del servicio de otro tipo.

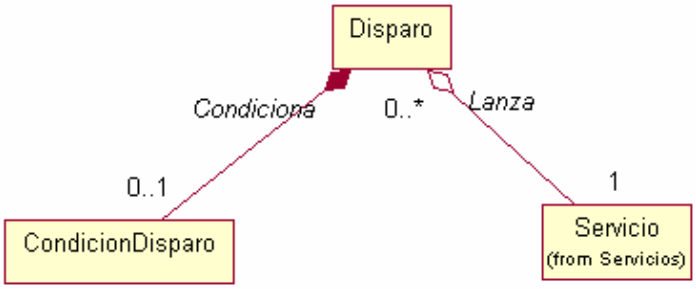

Figura 27. Paquete triggers del metamodelo PRISMA

Finalmente, se incluye la definición de **protocolos**, entendiendo protocolo como el conjunto de posibles vidas funcionales de un tipo (ver figura 28). El protocolo se define haciendo uso de los servicios y transacciones definidos en el aspecto funcional del tipo.

Dentro de un protocolo se definen un conjunto de estados de la vida de un tipo y las posibles transiciones válidas entre ellos.

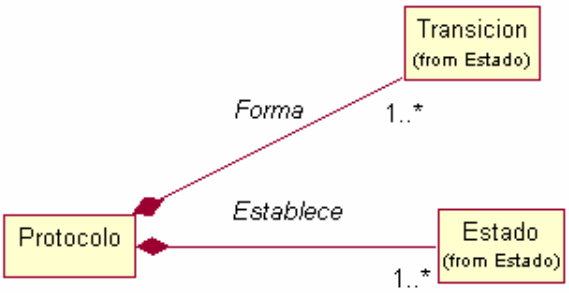

Figura 28. Paquete protocolo del metamodelo PRISMA

Un estado define el conjunto de servicios que se pueden ejecutar cuando la instancia de un tipo se encuentra en dicho estado (ver figura 29). El estado se compone por una serie de transiciones en las que se especifica cual es estado origen y cual es el estado destino después de la ejecución de un servicio. Además, existe la posibilidad de establecer guardas que

condicionen la ejecución del servicio dependiendo del valor de determinados atributos de un instante específico.

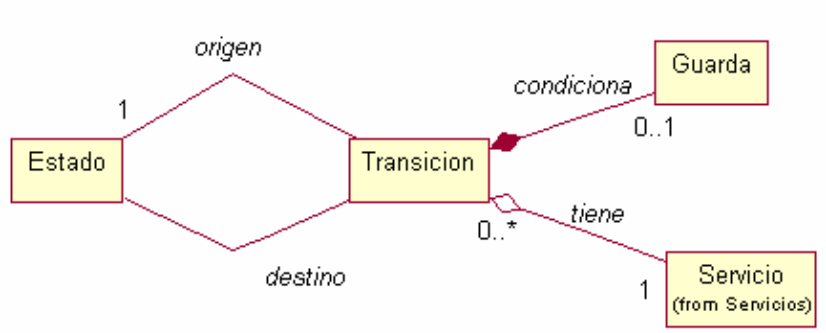

Figura 29. Paquete estado del metamodelo PRISMA

La especificación de evaluaciones, restricciones de integridad, precondiciones y condiciones se realiza mediante fórmulas bien formadas (Fbf) (ver figura 30). En nuestro modelo, toda Fbf compara o asigna un operador izquierdo y un operador derecho. Las Fbf pueden clasificarse en dos grupos diferentes temporales y no temporales.

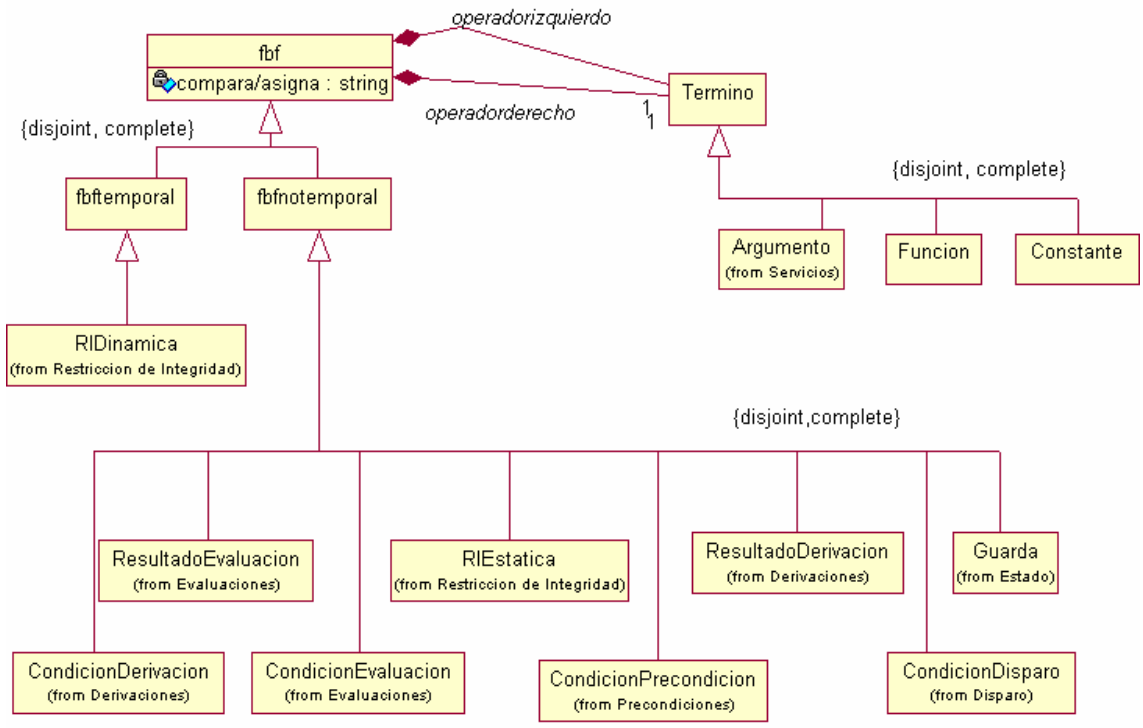

Figura 30. Paquete Fbf del metamodelo PRISMA

3.3.2. Aspecto Funcional

El aspecto funcional captura la semántica de la organización, mediante la definición de su estructura y su comportamiento y la encapsulación de su estado. Dicha información se expresa mediante los conceptos presentados en el apartado anterior.

El aspecto funcional de un componente elemental coincide con una plantilla de clase elemental OASIS recogiendo toda la información necesaria en cuanto a estado y

comportamiento funcional del tipo. Mientras que si se está especificando el aspecto funcional de un sistema, incluye la plantilla de más de un componente elemental.

3.3.3. Aspecto Coordinación

El aspecto de coordinación nos define las reglas de negocio y la sincronización entre elementos durante su comunicación. Así como la coreografía, extendiendo el concepto de "contrato" de Andrade y Fiadeiro ([And99]).

Un aspecto de coordinación especifica todas aquellas propiedades relacionadas con la coordinación entre elementos. Las características que afectan a las decisiones de coordinación, su cambio de estado, las consecuencias y restricciones asociadas a dicho cambio de estado, se especifican mediante los conceptos generales.

Un valor añadido de este aspecto son las **sincronizaciones** que se establecen en su especificación (ver figura 31). Las sincronizaciones definen las conexiones permitidas entre los servicios de entrada y los servicios de salida de las interfaces del aspecto de coordinación. Esta especificación se realiza mediante interfaces para conseguir un mayor nivel de abstracción y que un mismo aspecto de coordinación pueda ser utilizado por distintos tipos de conector. En caso de establecer las sincronizaciones mediante roles en lugar de utilizar sus interfaces, un aspecto sólo podría definirse asociado a un único conector, concretamente, aquél que incorpore los roles en base a los que se han definido las sincronizaciones. De este modo, el aspecto de coordinación dejaría de ser un ciudadano de primer orden del lenguaje y se perdería todas las ventajas que esta característica aporta al modelo.

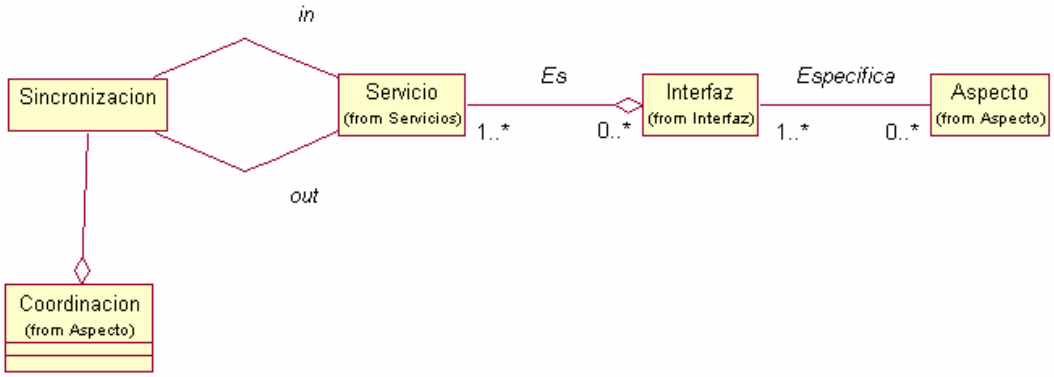

Figura 31. Paquete sincronizaciones del metamodelo PRISMA

Otra de las propiedades que especifica el aspecto de coordinación es la **coreografía** (ver figura 32). La coreografía especifica el proceso de coordinación que aplica el conector para sincronizar a los tipos que conecta. Al igual que las sincronizaciones, se especifica mediante interfaces.

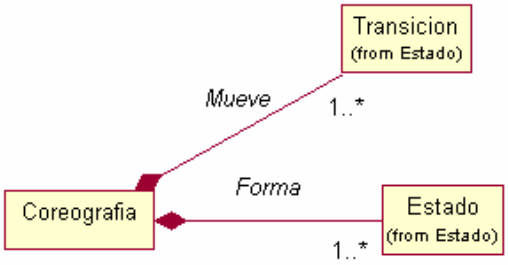

Figura 32. Paquete estado del metamodelo PRISMA

3.3.4. Aspecto Distribución

El aspecto de distribución captura aquellas propiedades que caracterizan el mecanismo de ubicación dinámico de las instancias de un elemento dentro del modelo de arquitectura software definido. Especifica las políticas y estrategias de distribución a seguir en función de los valores obtenidos en tiempo real de las métricas establecidas.

El aspecto de distribución especifica todas aquellas propiedades relacionadas con la distribución. Las características que afectan a las decisiones de distribución, su cambio de estado, las consecuencias y restricciones asociadas a dicho cambio de estado, se especifican mediante los conceptos generales de aspectos.

Además de los servicios de cambio, el aspecto de distribución ofrece servicios para la obtención de valores de métricas con el objetivo de implementar estrategias de distribución complejas tales como las que se describen en [Zin97, Che02].

El aspecto de distribución incorpora una serie de patrones de distribución que se aplican a la hora de tomar determinadas decisiones relacionadas con la distribución. Estos patrones son de utilidad no solo para sistemas PRISMA, también para cualquier sistema arquitectónico que tenga separadas las propiedades de distribución de las funcionales y tenga instancias capaces de replicarse y cambiar de ubicación en tiempo de ejecución.

El aspecto de distribución de una arquitectura SW orientada a aspectos se puede definir mediante la composición de un conjunto de patrones de distribución, tal y como se muestra en la figura 33. El aspecto de distribución varía dependiendo del subconjunto de patrones que se seleccionen del catálogo existente, cada patrón se identifica dentro de este catálogo mediante un número de patrón (**P-number)**. La selección de patrones para la creación de un aspecto se realiza en base a las necesidades de distribución del sistema que se va a definir.

En la figura 33, se puede observar que a nivel de aspecto se encuentran tres círculos: rojo, azul y amarillo. El círculo rojo y el círculo amarillo representan dos aspectos de distribución diferentes que han sido creados mediante la composición de diferentes patrones de distribución seleccionados del catálogo de patrones; mientras que el círculo azul representa un aspecto que no es de distribución. Por otro lado el nivel de tipos se compone de dos círculos más grandes, uno lila y otro verde. Estos tipos se crean aplicando la función *weaving* sobre aspectos de

diferente tipo. Finalmente varios tipos diferentes de nivel base se pueden componer para forma un tipo de una mayor nivel de granularidad, lo que en PRISMA se conoce como sistema.

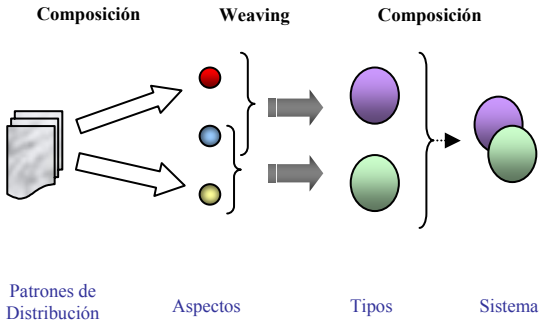

Figura 33. Uso de patrones de distribución para el proceso de desarrollo PRIMA

Entre los múltiples patrones que recoge el catálogo [Nou03], a continuación se citan algunos: Invocación de servicio, exceso de invocación de un servicio, exceso del volumen de datos intercambiado con un origen, exceso del tiempo de latencia de un mensaje, etc

3.3.5. Aspecto Evolución

El aspecto de evolución refleja la evolución del tipo debido al cambio de los requisitos de la organización. Este aspecto proporciona la dinámica del tipo, es decir, la capacidad de cambio de su estructura y comportamiento basándose en la evolución del software que plantea la reflexión de OASIS.

El metanivel de PRISMA y las propiedades reflexivas de los lenguajes diseñados dan soporte tanto para la evolución de los elementos, como para la reconfiguración dinámica de la topología. Esta característica del modelo permite definir un metanivel por aspecto que reifica las propiedades que deseen ser evolucionadas basándose en la reflexión de OASIS ([Car99]). De este modo, PRISMA da soporte a la evolución de sus modelos reduciendo el esfuerzo de mantenimiento de sus productos.

La reificación se realiza a partir del nivel base, lo que implica que no puede existir un elemento que tenga un aspecto especificado a nivel meta y no a nivel base. Por lo tanto, la relación entre el nivel meta y el nivel base de un aspecto es uno a muchos. Para expresar dicha reificación se hace uso del operador ***reify***, teniendo en cuenta que dicha reificación se realiza de forma automática.

El modelo arquitectónico consta de distintos tipos de elementos. Todos los elementos de el modelo arquitectónico forman parte del metamodelo para así poder modificarlos de forma dinámica a través de los mecanismos de reflexión del lenguaje.

Este aspecto se trata con detalle en el trabajo “Compilador para la Generación Automática del Metanivel de una Especificación mediante la Reificación de Propiedades del Nivel Base” que se presenta junto a este trabajo.

3.3.6. Aspecto Calidad

El aspecto de calidad [Per03a] especifica los requisitos no funcionales. Para su especificación se utiliza el lenguaje QRL (Quality Requirements Language) [Rui02], ya que tiene una semántica operacional precisa que no sólo permite la especificación de requisitos no funcionales, también facilita el proceso de búsqueda y negociación de elementos que cumplan unas determinadas medidas de calidad.

El aspecto de calidad especifica los atributos de calidad que el sistema quiere evaluar y el nivel de calidad que el tipo ha de satisfacer. Por otro lado especifica los servicios que implementan las métricas para medir la calidad del tipo que incorpora el aspecto.

La calidad puede medirse desde diferentes niveles de granularidad dentro de un producto software: la aplicación completa, componentes, interfaces, casos de uso, web services, etc. En PRISMA, el aspecto de calidad mide la calidad de los componentes y conectores. Sin embargo, en el caso de los sistemas, la calidad es la combinación de las evaluaciones de los elementos que incluye.

3.3.7. Aspecto Context-Awareness

El aspecto *context-awareness* [Jae03a] proporciona información del contexto y soporta análisis para recuperar propiedades estructurales de la información proporcionada.

Según el trabajo de [Jae03a] el *context-awareness* debe proporcionar la especificación de cinco interfaces: ISource, IGatherer, IAggregator, IAnalyzer and INotifier. La semántica de dichas interfaces esta definida dentro del aspecto *context-awareness* de PRISMA. A continuación se explican brevemente cada una de ellas:

- ISource: Incluye servicios para la gestión y mantenimiento del esquema.

- IGatherer: Incluye servicios que recopilan información de contexto de sus fuentes con una cierta frecuencia y la almacenan.

- IAggregator: Sus servicios elaboran información compleja asociando información de otras fuentes.

- IAnalyzer: Sus services establecen categorías y asignan etiquetas a cada categoría.

- INotifier: Incluye servicios que proporcionan la notificación de información de forma asíncrona.

Este aspecto también aporta beneficios al mantenimiento y evolución de un sistema PRISMA, ya que permite no sólo tener en cuenta aquellas modificaciones del sistema que son debidas a un cambio de requisitos, sino también aquellas debidas a un cambio de contexto. El hecho de soportar este tipo de evolución, conocida como *contex-driven evolution*, y la evolución debida a los cambios de requisitos, permite que el comportamiento de las aplicaciones PRISMA sea de calidad.

3.3.8. Aspecto Navegación

Actualmente, el modelo de diseño de sistemas hipermedia no resulta suficientemente rico para organizar, estructurar, acceder y realizar tareas de aplicaciones relacionadas con la Cultura Digital. Esta necesidad, lleva al propósito de darle solución mejorando el diseño y la construcción de sistemas hipermedia.

Por otro lado, los sistemas hipermedia se caracterizan por ser de una naturaleza muy variable ya que sus requisitos continuamente están sometidos a cambios. Sin embargo, en la actualidad, los sistemas hipermedia son adaptativos y no llegan a ser evolutivos. Hoy en día, en el mundo de la ingeniería del software, la evolución se concibe como el soporte a los cambios en la funcionalidad de un sistema, pero no se tiene en cuenta que los cambios de los requisitos pueden afectar a los datos almacenados por el sistema. Es por esto, que los sistemas hipermedia necesitan un soporte automático a la evolución, no sólo de aplicaciones, sino también de datos.

PRISMA soporta esta evolución automática a través de su metamodelo y sus mecanismos de reflexión y reificación. Por otro lado, el soporte a la evolución de los datos y su recuperación se abordará siguiendo el enfoque de los trabajos [Per02a], [Per02b] y [Per02c].

El aspecto de navegación permite modelar, junto con el aspecto de presentación y de evolución, sistema hipermedia dinámicos y evolutivos. Concretamente, el aspecto de navegación especifica los hipervínculos asociados al componente y la información multimedia asociada a estos.

3.3.9. Aspecto Presentación

El aspecto de presentación establece el modo en que se va a mostrar al usuario un determinado tipo. Este aspecto cobra una gran relevancia en aquellos sistemas de información en los que la sencillez y la interfaz gráfica de usuario son de gran importancia a la hora de manejar la aplicación, un claro ejemplo son los sistemas hipermedia

LENGUAJE DE DEFINICIÓN DE ARQUITECTURAS PRISMA

El lenguaje en el que está basado PRISMA es OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos, que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA se basa en OASIS, garantizando la compilación de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos de arquitectura, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

A diferencia de otros modelos, que sólo tienen un ADL (Architecture Definition Languages), PRISMA tiene dos lenguajes de especificación: el lenguaje de definición de componentes y el lenguaje de configuración.

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

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

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

4.1. Lenguaje de definición de componentes

4.1.1. Interfaz

Una interfaz publica dos tipos de servicios, los de consulta (atributos) y los de modificación (eventos). Las interfaces no especifican sintácticamente el tipo de servicios que incluyen y por lo tanto, no están clasificadas por tipos: funcionales, distribución, coordinación, etc, y tampoco especifican si van a ir asociadas a puertos o roles de entrada o de salida. Esto último permite que una misma interfaz pueda asociarse a puertos o roles de diferente índole.

Si bien es cierto, algo que se ha de tener en cuenta es que todos los servicios que publica una interfaz han de pertenecer al mismo tipo: funcional, distribución, coordinación, etc. Esta restricción se ha de cumplir siempre que se define una nueva interfaz. La sintaxis para especificar una interfaz con el lenguaje de definición de componentes es la siguiente:

```
Interface name
   Attributes
      service1;
        ...
      servicen;
   Services
      service1;
        ...
      servicen;
End_Interface name;
```

Un ejemplo de especificación de interfaz es la interfaz *Gestión* que se incluye en el caso de estudio del portal de subastas.

```
Interface Gestion
      Services
            Alta;
            Baja;
            Modificacion (descripcion: string);
End_Interface Gestion;
```

Tal y como se ejemplifica anteriormente, no es obligatorio que una interfaz tenga atributos y eventos. Sin embargo toda interfaz al menos ha de tener algún atributo o servicio.

4.1.2. Aspecto

Existe una plantilla genérica para especificar un aspecto PRISMA. Aquellos aspectos que necesitan especificar características propias, la extienden para especificar dichas propiedades. La plantilla genérica es la que se presenta a continuación:

```
tipo_aspecto Aspect name specifies interface1, ... interfacen;

      identification
            <nombre_identificador> <lista_id_atributos>
      attributes
            <nombre_atributo> (<tipo_atributo)
                  valuations <declaración_de_variables>
                     <formulas_observabilidad_atributo>
                  end_valuations
                  ......
      constraints <declaracion_de_variables>
```

```
            static <restricciones_estaticas>
            .....
        services <declaracion_de_variables>
                    new <nombre_service> (<arg_service>)
                    [as <nombre_service> (<arg_service)];
                    destroy <nombre_service> (<arg_service>)
                    [as <nombre_service> (<arg_service)];
                    <nombre_service> (<arg_service>)
                    [as <nombre_service> (<arg_service)];
                    .......
        preconditions <declaracion_de_variables>
            <formula_precondicion_evento>
            .......
        triggers <declaracion_de_variables>
            <formula_disparo_evento>
               ......
        operations [transactions] <declaracion_de_variables>
            <formula_transaccion>
            ......
        protocols <declaracion_de_variables>
    <formula_protocolo>
End_ tipo_aspecto Aspect name;
```

En la **cabecera** del aspecto se especifica el tipo de propiedades que define: funcionales, distribución, coordinación etc. Además, se le da nombre y se indica cuáles son las interfaces a las que da semántica. Un ejemplo de cabecera podría ser la del aspecto funcional *mantenimiento* que da semántica a las interfaces *gestión* y *consulta*:

```
Functional Aspect Mantenimiento specifies Gestion, Consulta;
```

Lo primero que se indica en la plantilla es el **identificador** del aspecto, aquel atributo o lista de atributos que identifican inequívocamente a un aspecto; de tal forma que, no pueden existir dos aspectos con el mismo identificador. Una posible especificación del identificador del aspecto de mantenimiento, que se está mostrando como ejemplo, es la siguiente:

```
identification
        my_identification: (number);
```

Dentro del aspecto se especifican aquellos **atributos** que son necesarios para guardar las propiedades del aspecto. El atributo guarda el estado de dichas propiedades a lo largo de la vida de la instancia del tipo. A la hora de definir un atributo se ha de especificar que tipo de valores va ha contener. A continuación se muestra un ejemplo de especificación de atributos:

```
Attributes
        Codigo: string;
        Number: integer;
        Nombre: string;
```

Las **evaluaciones** especifican el cambio de estado de los atributos del aspecto ante la ejecución de un servicio determinado. Por este motivo, las evaluaciones se especifican asociadas a los atributos del aspecto. Las evaluaciones son fórmulas en lógica dinámica del tipo $\varphi \rightarrow [a]\phi$ y se interpretan como: "si en un determinado estado del aspecto se satisface φ y ocurre la acción *a*, en el estado inmediatamente posterior se satisface ϕ". Se ha de tener en cuenta que se puede no declarar la condición de evaluación φ asumiéndose true.

A continuación se presenta un ejemplo del resultado la ejecución del servicio *modificar* del aspecto mantenimiento, de forma que el atributo *nombre* toma el valor del argumento *descripción* del servicio :

```
Valuations
        [modificar(Descripcion)] Nombre = Descripcion;
end_valuations
```

Las **restricciones de integridad** son fórmulas basadas en el estado del aspecto y que deben satisfacerse a lo largo de toda la vida de las instancias del tipo que incorpore dicho aspecto. Las restricciones se pueden clasificar en estáticas y dinámicas, dependiendo de si se refieren sólo aun estado o relacionan a diferentes estados, respectivamente. Las restricciones dinámicas se caracterizan por utilizar operadores temporales como *sometimes* o *always*. Un ejemplo de restricción de integridad es la que impone el sistema de subastas, de forma que ningún cliente puede realizar una puja superior a 60 € por un producto, todas las pujas que se hagan, para un mismo o diferente producto, han de ser de un precio inferior al máximo establecido.

```
Constraints
        static {CantidadPuja <= 60 € }
```

Por otro lado, se definen los **servicios** del aspecto. De forma que, todo servicio que aparezca en las interfaces que especifica el aspecto, han de definirse dentro del aspecto; además de otros servicios propios y no públicos del aspecto. Existen diferentes tipos de servicios: creación, destrucción, modificación y consulta. Los servicios de creación y destrucción se diferencian del resto por etiquetarlos con **new** y **destroy**.

El servicio *new* ayuda a crear una instancia del tipo (componente, conector, sistema) del que forma parte, inicializando su parte correspondiente de atributos necesarios para instanciar el tipo y guardándolos en memoria.

El servicio *destroy* es el que permite liberar la memoria asociada al aspecto que forma parte de la instancia del tipo (componente, conector, sistema) que se desea destruir.

La definición semántica del conjunto de servicios del aspecto ha de diferenciar el comportamiento cliente y servidor de cada servicio. Un ejemplo es el servicio *incrementarsaldo(cantidad: moneda)* de la subasta. Este mismo servicio realiza acciones diferentes cuando tiene un comportamiento cliente o servidor. Dado un componente que da semántica en su aspecto funcional a dicho servicio, cuando otro componente solicita un *incrementarsaldo(cantidad: moneda)* éste debe modificar el atributo *saldo* del aspecto con la cantidad que especifique el servicio (comportamiento servidor). Sin embargo, cuando es él, el que solicita un *incrementarsaldo(cantidad: moneda)* a otro componente, se ha de lanzar una petición de ejecución del servicio (comportamiento cliente). Además, los estados del protocolo en los que se puedan realizar estas dos acciones (servidora y cliente) puede ser distintos, es decir, puede ser que en un estado de un protocolo solamente pueda ejecutarse el servicio con su comportamiento cliente o viceversa. Por este motivo es necesario diferenciar de algún modo dentro del aspecto el comportamiento cliente y servidor de un servicio. La forma de diferenciarlo va a ser sintáctica, poniendo delante del servicio la etiqueta in o out, **in** en el caso en el que haga referencia al comportamiento servidor del servicio y **out** al comportamiento cliente. Solamente es necesario etiquetar al servicio con **in/out** cuando es necesario explicitar el comportamiento cliente o servidor, ante la ausencia de etiquetas el servicio se considera **in**, es decir, que su comportamiento es servidor.

Por otro lado, se ha de tener en cuenta que en PRISMA, un mismo servicio puede tener nombres diferentes en distintos sitios. Para conseguir esta flexibilidad en las especificaciones del lenguaje y no generar inconsistencias, el lenguaje de definición de componentes tiene un mecanismo de renombrado de servicios a través del operador **as**. De este modo, se consigue que el nombre de un servicio en la interfaz no tenga por qué coincidir con el nombre que tenga en el aspecto. Este operador se especifica del siguiente modo:

```
nombre_servicio_aspecto as nombre_servicio_interfaz
```

Los servicios del aspecto *Mantenimiento* del ejemplo de la subasta electrónica es el siguiente:

```
Services

    New Crear as Alta;

    Destroy Destruir as Baja;

    In Modificar(Descripcion: string)

    as Modificacion (Descripcion:string);

    Out Modificar(Descripcion: string);

    Consultar as Consulta;
```

Las **precondiciones** establecen condiciones que se han de satisfacer para que un servicio se pueda ejecutar. No es suficiente que entre una petición de ejecución de un determinado servicio por un puerto de entrada para que se lleve a cabo dicha ejecución; también se ha de satisfacer la condición de la precondición asociada al servicio.

Dentro del contexto de la lógica dinámica, las precondiciones tienen la forma $\neg\phi[a]$ false, donde ϕ es una fórmula bien formada (Fbf) interpretada como una condición de la ocurrencia de la acción indicada. Su semántica es " si no se cumple, la ocurrencia de la acción no lleva a la instancia del tipo a un estado siguiente". Un ejemplo de precondición es el que se impone al servicio *ComenzarPuja* del aspecto *Pujas* del ejemplo de las subasta electrónica. Dicha precondición establece que no se puede ejecutar el servicio *ComenzarPuja* si el atributo *NumPujadores* del aspecto no es superior a 2.

```
Preconditions

        ComenzarPuja if {NumPujadores >= 3};
```

Un **disparo** permite que el tipo que contiene el aspecto en el que esta definido el disparo pueda lanzar el servicio al que esta asociado dicho disparo; de forma que el tipo actúa como agente del servicio. La semántica del disparo es que si el tipo que contiene el aspecto que define el disparo esta en un estado que satisface su condición, entonces se genera una petición de ejecución del servicio.

Dentro del contexto de la lógica dinámica, los disparos tienen la forma $\phi[\neg a]$ false, donde ϕ es una fórmula bien formada (Fbf) interpretada como una condición de la ocurrencia de la acción indicada. El significado es "si ϕ se satisface y no se ejecuta la acción indicada, entonces la instancia del tipo no alcanza un estado válido". Por lo tanto, es necesario que la acción ocurra para que la instancia alcance el estado siguiente. Un ejemplo es que la subasta solicite obligatoriamente el pago del producto cuando este es adjudicado.

```
triggers
        out pagar when { adjudicado = true}
```

Un **operación** es una servicio no elemental y no atómico. Para que una operación sea atómica se ha de especificar que se trata de una **transacción**. Una operación es un proceso en el que se establece una secuencia de acciones que han de ocurrir. Un ejemplo de conjunto de servicios que forman una transacción son aquellos que cumplen con la función de enviar noticias del portal de subastas a los clientes.

```
operations transactions ENVIONOTICIAS:
        CLIENTES = seleccionarListaCorreo.NOTICIAS;
        NOTICIAS =  seleccionarnoticia.ENVIO;
        ENVIO = envioSimultaneo.CONFIRMACIÓN;
        CONFIRMACION = confirmacionLlegada};
```

Un **protocolo** define un proceso describiendo un conjunto de acciones cuya ocurrencia es posible. Tal y como se explico en el capítulo anterior, un protocolo esta compuesto por un conjunto de estados que establecen los servicios y transiciones a otros estados posibles. En el ejemplo de la subasta electrónica existe un protocolo en cada aspecto, un ejemplo de protocolo es el del aspecto de *mantenimiento.*

```
Protocol
        CREACION ≡ crear.MANTENIMIENTO;
        MANTENIMIENTO ≡ modificar + consultar + destruir;
```

4.1.2.1. Aspecto Funcional

La especificación de un aspecto funcional en el lenguaje de definición de componentes coincide con la plantilla de una clase elemental OASIS. En ella se especifican todas las propiedades que se definieron en el aspecto funcional.

La especificación de este aspecto coincide exactamente con la sintaxis genérica presentada anteriormente. Por otro lado, los ejemplos presentados junto a la sintaxis anterior pertenecen al aspecto funcional, por lo tanto el aspecto funcional no añade fragmentos nuevos lenguaje.

4.1.2.2. Aspecto Coordinación

El aspecto de coordinación, además de las especificaciones que realiza cualquier otro tipo de aspecto, define dos conceptos propios, las sincronizaciones y la coreografía.

- Sincronizaciones

Las sincronizaciones del aspecto definen las conexiones permitidas entre los servicios de entrada y los servicios de salida de las interfaces del aspecto de coordinación. Se ha de tener en cuenta, que dichas interfaces pueden ser de cualquier tipo: funcional, distribución, etc, ya que el aspecto de coordinación lo que establece es el modo en que se coordinan diferentes instancias de tipos a través de los distintos servicios que invocan y reciben.

La sincronización se realiza a través de los servicios de las interfaces que implementa el aspecto. Una sincronización se establece mediante un par de servicios de forma que la semántica de la sincronización es la siguiente: "Cuando se solicite el servicio X de la interfaz I_1 y se ejecute con éxito, se ha de invocar el servicio Y de la interfaz I_2". Teniendo en cuenta las siguientes restricciones:

- Los servicios que especifican las interfaces I_1 y I_2 han de que pertenecen al mismo tipo (funcional distribución, etc). Esto quiere decir que no se puede definir en una sincronización, que la ejecución de un servicio de carácter funcional implique la ejecución de un servicio de carácter distribuido.
- Cuando el aspecto de coordinación se importe para definir un conector, las interfaces I_1 y I_2 han de tipar a roles de diferente tipo (entrada, salida). Esto es, que si I1 esta asignado a un rol de entrada y I2 esta asignado a un rol de salida, y viceversa.

Las sincronizaciones se realizan mediante interfaces para conseguir un mayor nivel de abstracción y que un mismo aspecto de coordinación pueda ser utilizado por distintos tipos de conector. En caso de establecer las sincronizaciones mediante roles en lugar de utilizar sus interfaces, un aspecto sólo podría definirse asociado a un único conector, concretamente, aquél que incorpore los roles en base a los que se han definido las sincronizaciones. De este modo, el aspecto de coordinación dejaría de ser un ciudadano de primer orden del lenguaje y se perdería todas las ventajas que esta característica aporta al modelo. Las distintas sincronizaciones que realiza un aspecto se especifican entre llaves encabezadas por las palabras reservadas *synchronizing strategies*, tal y como se muestra a continuación:

```
Synchronizing Strategies
  {
      ......
  }
```

Actualmente, el lenguaje utilizado para especificar las sincronizaciones entre servicios es π-cálculo poliádico [Mil91]. Este lenguaje permite describir modelos de arquitectura software en los que los servicios de las componentes se ejecutan de forma concurrente. π-cálculo poliádico es una versión generalizada de π-cálculo monádico, la cual permite enviar una tupla de parámetros a través de un enlace (poliádico), en lugar de un solo parámetro (monádico). Las entidades y operadores originales de este lenguaje son los que se presentan a continuación:

- Entidades :
    - Nombre: Ligadura o canal. Puede ser de entrada o de salida:
        - Entrada: x?(λx1…xn)
        - Salida: x!(x1…xn)
    - Proceso (P): Se definen a partir de nombres
    - Prefijo (π.P): Primera acción de un proceso
- Operadores:
    - P | Q: Paralelismo (or)
    - ! P : Disparo o Instanciación
    - (ν x) P: Restricción del nombre en un determinado proceso

Para especificar las sincronizaciones PRISMA mediante π-cálculo poliádico se han establecido las correspondencias entre los conceptos de ambos modelos. Dichas correspondencias se citan a continuación:

PRISMA	π-cálculo poliádico
Interfaz	Proceso
Servicio	Acción o Prefijo
Nombre del servicio	Canal
Parámetros del servicio	Información del canal

La sintaxis para especificar las sincronizaciones entre servicios es la siguiente:

```
Synchronizing Strategies
{
        Service1 ?(λx1...xn).interface1 → Service2!(λx1...xn).interface2;
       ......
}
```

De forma que, *service₁* es un servicio que pertenece a la interfaz *interface₁,* cuyos parámetros son $x_1 \ldots x_n$. La interfaz *interface₁* deberá estar asociada a un rol de entrada (?) cuando el aspecto de coordinación en el que se define dicha sincronización se incorpore a un conector. Por otro lado, *service₂* es un servicio que pertenece a la *interface₂*, cuyos parámetros son $x_1 \ldots x_n$. La interfaz *interface₂* deberá estar asociada a un rol de salida (¡) cuando el aspecto de coordinación en el que se define dicha sincronización se incorpore a un conector.

Sin embargo, estas entidades y conceptos presentados no proporcionan la expresividad necesaria para especificar las sincronizaciones, ya que es necesario poder definir condiciones sobre la sincronización de servicios y π-cálculo poliádico no ofrece ningún mecanismo para poderlo especificar. La expresión de condiciones en las sincronizaciones es fundamental para que un mismo servicio de entrada pueda sincronizarse con distintos servicios de salida dependiendo del estado en el que se encuentre la instancia del tipo.

En el ejemplo que se muestra a continuación los servicios *service2* y *service3* de las interfaces *I2* e *I3* respectivamente, se ejecutan siempre que se ejecuta el servicio *service1* de la interfaz *I1*.

```
Synchronizing Strategies
{
        Service1 ?(λx1…xn).I1→ Service2!(λx1…xn).I2;
        Service1 ?(λx1…xn).I1→ Service3!(λx1…xn).I3;
}
```

Sin embargo no podríamos condicionar su ejecución al valor de un atributo perteneciente al aspecto. Por este motivo, se ha extendido π-cálculo poliádico con el objetivo de poder expresar todo lo necesario para el aspecto de coordinación. En el caso concreto de las sincronizaciones, se ha introducido el operador condicional:

```
|x=y|E
```

La forma de utilizar el operador de comparación se muestra a continuación con un ejemplo que condiciona mediante el valor del atributo *x* el hecho de que la ejecución del servicio *service1* de la interfaz *interface1* desencadene la ejecución de un servicio u otro.

```
Synchronizing Strategies
{
   Service1 ?(λx1...xn).interface1 → |x=y| Service1!(λx1...xn).interface2;
   Service1 ?(λx1...xn).intefaces1 → |x=z| Service1!(λx1...xn).interface3;
   ......
}
```

Cada una de las sincronizaciones establecidas es necesario nombrarlas para poder referenciarlas. Este nombramiento puede realizarse una a una o por grupos de sincronizaciones. De forma que la sintaxis final del lenguaje para especificar las sincronizaciones es la siguiente:

```
Synchronizing Strategies
        <nombre_sincronizacion>
                <formula_sincronización>
                .......;
        .......;
```

Un ejemplo del caso del portal de subastas electrónicas es el conector de la subasta que cuando se aumenta el precio del producto que se está subastando se lo comunica a todos los pujadores que están participando en dicha subasta.

```
Synchronizing Strategies
    AUMENTOAPUESTA
        Puja ?(Cantidad).SesionPuja >>
        ModificaPrecio!(precio).SesionPuja;
```

- Coreografía

La coreografía del aspecto de coordinación establece el proceso de sincronización entre las instancias de los tipos conectados a un conector La coreografía se establece mediante las sincronizaciones establecidas en el aspecto.

Al igual que las sincronizaciones, la coreografía utiliza una extensión de π-cálculo poliádico para poder expresar los procesos de coordinación de forma adecuada. A continuación se detallan los operadores de los que se dispone para especificar de coreografías:

- Inactividad:

E ::= 0

- Acciones internas:

a::=tau

- Restricciones:

(x)E

- Condición:

|x=y|E

- Alternativa:

E+E

- Composición paralela:

E||E

La sintaxis del lenguaje para especificar la coreografía de un aspecto de coordinación es la que se muestra a continuación:

```
choreography <declaracion_de_variables>
        <formula_coreografía>
```

4.1.2.3. Aspecto Distribución

La especificación del aspecto de distribución en el lenguaje de definición de componentes coincide con la plantilla genérica de aspecto. La diferencia de este aspecto no es sintáctica, sino semántica, ya que tiene atributos y servicios predefinidos para dar soporte a determinadas necesidades de distribución. Atributos como la dirección IP o URL y servicios como *clone, move, replicate,* etc.

4.1.3. Componente

La plantilla de un componente se divide en cuatro partes fundamentales: cabecera, puertos de comunicación, aspectos y *weaving,* que entreteje los aspectos que importa.

```
Component_type  name
      Inport
            InP_i : interface_i;
            … …
      End_Inport;
      Outport
            OutP_i: interface_i;
            … …
      End Outport;
      [Functional Aspect
            F_i: Functional Aspect;
            [Weaving
                  Aspect_i
                        <formula_weaving>;
                        ……
                  End_Aspect_i;
            End_Weaving;]
      End_Functional_Aspect;]
```

```
        [Distribution Aspect
              Fi: Distribution Aspect;
              [Weaving
                    Aspectᵢ
                          <formula_weaving>;
                          ……
                    End_Aspectᵢ;
              End_Weaving;]
        End_Distribution_Aspect;]
        … …
End_Component_type name;
```

La **cabecera** de la especificación de un componente se compone de la palabra reservada *component_type* y el nombre que reciba la misma, nombre que identifica inequívocamente al componente dentro de las librerías de tipos PRISMA.

Component Galería

La definición de los **puertos de comunicación** del componente se realiza diferenciando los de entrada de los de salida. De forma que aquellos puertos que son de entrada van precedidos por la palabra reservada *inport,* mientras que aquellos que son de salida les precede *outport.* La especificación de puertos se realiza dividiendo los puertos en dos grandes grupos entrada o salida, incluyendo todos los puertos de entrada del componente entre las palabras *inport* y *end_inport* y los de salida entre *outport* y *end_outport*.

```
Inport
      InP1: Gestion;
       InP2: Consulta;
End Inport;
Outport
      OutP1: Consulta;
End Outport;
```

Después de la especificación de los puertos, se realiza la importación de cada uno de los distintos **tipos de aspecto** que necesite el componente, indicando el identificador del aspecto. La especificación de los aspectos es opcional, para permitir la importación de componentes COTS. Sin embargo, si se esta definiendo un nuevo tipo de componente PRISMA, el aspecto funcional es obligatorio.

```
Functional Aspect
      Fi: Functional Aspect;
```

```
End_Functional_Aspect;
```

Solamente se puede importar un aspecto por cada tipo (funcional, distribución, etc). Dicho aspecto puede entretejerse con el resto de tipos de aspectos mediante el *weaving*. Agrupándose las coordinaciones inter-aspectos (*weaving*) por tipos de aspecto.

El *weaving* se establece mediante los servicios de aspecto, indicando si la sincronización se realiza *after* (después), *before* (antes) o *around* (en lugar de). El primer servicio es del tipo (funcional, distribución, etc) al que pertenece el aspecto en el que se define el *weaving* y el segundo pertenece al aspecto con el que se realiza el *weaving*. Por ejemplo, el servicio *CambioDireccion* es de carácter funcional, mientras *CambioNodo* es del aspecto de distribución, siendo lo semántica de la fórmula *CambioDireccion* ***after*** *CambioNodo* que "Después de ejecutarse un cambio de dirección del aspecto funcional, se realice un cambio de nodo de la componente".

```
Functional Aspect
      Mantenimiento: Functional Aspect;
      Weaving
            Distribution_Aspect
                  CambioDireccion after CambioNodo;
            End_Distribution_Aspect;
      End_Weaving;
End_Functional_Aspect;
```

4.1.4. Conector

La plantilla de un conector, al igual que una componente, se divide principalmente en cuatro partes: cabecera, roles de comunicación, aspectos y *weaving,* que entreteje los aspectos que importa.

```
Connector_type  name
      Inrol
            InR_i : interface_i;
            … …
      End_InRol;
      Outrol
            OutR_i: interface_i;
            … …
      End OutRol;
```

```
        [Coordination Aspect
              Fi: Coordination Aspect;
              [Weaving
                    Aspecti
                          <formula_weaving>;
                          ……
                    End_Aspecti;
              End_Weaving;]
        End_Coordination_Aspect;]
        [Distribution Aspect
              Fi: Distribution Aspect;
              [Weaving
                    Aspecti
                          <formula_weaving>;
                          ……
                    End_Aspecti;
              End_Weaving;]
        End_Distribution_Aspect;]
        … …
  End_Connector_type name;
```

La **cabecera** de la especificación de un connector se compone de la palabra reservada *connector_type* y el nombre que reciba el connector, nombre que identifica inequívocamente al conector dentro de las librerías de tipos PRISMA.

Connector ConctSubasta

La definición de los roles de comunicación del connector se realiza del mismo modo que la de los puertos en las componentes, cambiando las palabras reservadas por *inrol* y *outrol.* Por otro lado, también se especifica del mismo modo la importación de aspectos y el *weaving*, teniendo en cuenta que cuando no se importan COTS el aspecto de coordinación es obligatorio para la definición del conector.

4.1.5. Sistema

La plantilla de un sistema especifica los tipos que importa para construirlo, los attachments para conectar a los elementos que incorpora, en caso de que sea necesario, los puertos de comunicación, los bindings que conectan los puertos del sistema con los puertos de los elementos que contiene, los aspectos y el *weaving* que entreteje dichos aspectos.

```
System_type name
        Import Types
                Components
                        name(cardinalidad),
                        … … ;
                Connectors
                        name(cardinalidad),
                        … … ;
                Systems
                        name(cardinalidad),
                        … … ;
        End_Import Types;
        Attachments
                Associated_connector name;
                        <formula_conexión>;
                        ……
        End_Attachments;
        Bindings
                <formula_enlace>;
                        ……
        End_Bindings;
        [Functional Aspect
                Fi: Functional Aspect;
                [Weaving
                        Aspeci
                                formula_weaving>;
                                        ……
                        End_Aspecti;
                End_Weaving;]
        End_Functional_Aspect;]
        … …
End_System_type Name;
```

La **cabecera** de la especificación de un sistema se compone de la palabra reservada *system_type* y el nombre que reciba el nombre, nombre que identifica inequívocamente al sistema dentro de las librerías de tipos PRISMA.

```
System  Subasta
```

El sistema importa todos los **tipos** que incluye mediante su identificador y especifica el número de elementos que necesita de cada uno de ellos para establecer el patrón (cardinalidad).

```
Import Types
        Components
                Comprador(n);
End_Import Types;
```

La definición de ***attachments*** en el lenguaje de definición de componentes se establece a nivel de análisis, enlazando elementos que pertenecen a un sistema, con el objetivo de definir un patrón arquitectónico. La relación de *attach* se asocia a un conector para agrupar los distintos *attachments* relacionados con los distintos conectores. Sin embargo, la relación de *attach* se especifica dentro del sistema (patrón topológico) y no dentro del conector, para que el conector sea un tipo reutilizable y no tenga dependencias con los tipos a los que esta conectado.

La relación de *attach* realiza la conexión mediante los puertos y los roles de los componentes y conectores que desea conectar, respectivamente. De forma que si el puerto de salida y el rol es de entrada, la conexión se establece mediante el símbolo >>. Mientras que si el puerto es de entrada y el rol es de salida el símbolo que la establece es el <<.

```
Attachments
    Associated_connector CnctSubasta;
        Comprador(n).Outport1 >> CntSubasta.Inport1;
        … …
    End_associated_connector CnctSubasta;
    ……
End_Attachments;
```

A continuación de los attachments, se definen los bindings del sistema. Indicando cada puerto del sistema con qué puerto, de uno de los elementos que encapsula, se conecta. De forma que si ambos puertos son de salida, el enlace se realiza mediante el símbolo →. Mientras que si ambos puertos son de entrada el símbolo que lo establece es el ←. Por otro lado, el puerto del elemento que encapsula el sistema, siempre va precedido de dicho elemento mediante la notación punto.

```
Bindings
    Comprador(n).Outport2 → Ourport1;
    ……
End_Bindings;
```

La definición de la importación de aspectos y el *weaving* del sistema se realiza del mismo modo que en las componentes elementales.

4.2. Lenguaje de configuración

El lenguaje de configuración PRISMA especifica un modelo arquitectónico específico. Por este motivo, identifica el modelo por un nombre y enmarca su definición mediante la siguiente **cabecera** y fin de especificación:

```
Architectural Model  <nombre modelo>
    ......
End_Arquitectural Model <nombre modelo>;
```

Lo primero que se ha de definir en la especificación de todo modelo arquitectónico es la importación de los **tipos** para definir las instancias necesarias. Dicha definición utiliza la cláusula *import* para especificar la importación de tipos de las librerías PRISMA, tipos ya definidos mediante el lenguaje de definición de tipos y almacenados para su posterior utilización en cualquier modelo arquitectónico.

```
Import Types
    Components
            <nombre tipo>, … … ;
    Connectors
            <nombre tipo>, … … ;
    Systems
            <nombre tipo>, … … ;
End_Import Types;
```

En el caso del portal de subastas electrónico, para definir un modelo arquitectónico de la puja. Al menos se deberían importar los tipos comprador, subasta y producto.

```
Import Types
    Components
        Comprador, Producto;
    Connectors
        Subasta;
End_Import Types;
```

Una vez importados los tipos necesarios, es posible definir las **instancias** de los tipos. La sección de declaración de instancias está la delimita la palabra reservada *instances*.

```
Instances
    <nombre_instancia>: <nombre_tipo>;
        ......
End_Instances;
```

Un ejemplo de subasta mínimo sería definir una instancia de cada uno de los tipos importados.

```
Instances
    Pujador1: Comprador;
    CasaMunyecas: Producto;
    SesionSubasta1: Subasta;
End_Instances;
```

Después de definir las instancias del modelo, se definen las conexiones entre ellas a nivel de diseño, estableciendo de este modo la topología del modelo arquitectónico. Dichas conexiones, se establecen mediante ***attachments*** entre las instancias.

```
Attachments
    Associated_connector <nombre conector>;
                         <formula_conexión>;
                         ……
    End_associated_connector <nombre conector>;
End_Attachments;
```

A continuación se presentan los attachments asociados al conector *SesionSubasta1* del ejemplo de la subasta electrónica.

```
Attachments
    Associated_connector SesionSubasta1;
        Pujador1.Outport1 >> SesionSubasta1.Inrol1;
        CasaMunyecas.Outport1 >> SesionSubasta1.Inrol2;
        Pujador1.Inport1 << SesionSubasta1.Outrol1;
        CasaMunyecas.Inport1 << SesionSubasta1.Outrol2;
    End_associated_connector CnctSubasta;
End_Attachments;
```

Los conceptos hasta ahora presentados del lenguaje de configuración permiten definir un modelo arquitectónico cuyos elementos se encuentra al mismo nivel de granularidad. Sin embargo, aquellos modelos en los que existen sistemas, elementos con distintos niveles de granularidad, es necesario instanciarlos. Para ello el lenguaje de configuración presenta una definición de sistema que permite tanto instanciar los elementos que encapsula como las conexiones existentes entre ellos.

```
System_type name
    instances
        <nombre_instancia>: <nombre_tipo>;
        … …
    End_instances;
    Attachments
        Associated_connector <nombre conector>;
            <formula_conexión>;
                ……
        End_associated_connector <nombre conector>;
    End_Attachments;
    Bindings
        <formula_enlace>;
            ……
    End_Bindings;
End_System_type Name;
```

Suponiendo que el pequeño modelo que se ha presentado como ejemplo en el lenguaje de configuración, en lugar de un modelo arquitectónico fuese un sistema, su especificación sería la siguiente:

```
System_type Subasta
    instances
        Pujador1: Comprador;
        CasaMunyecas: Producto;
        SesionSubasta1: Subasta;
    End_instances;
    Attachments
        Associated_connector SesionSubasta1;
            Pujador1.Outport1 >> SesionSubasta1.Inrol1;
            CasaMunyecas.Outport1 >> SesionSubasta1.Inrol2;
            Pujador1.Inport1 << SesionSubasta1.Outrol1;
            CasaMunyecas.Inport1 << SesionSubasta1.Outrol2;
        End_associated_connector SesionSubasta1;
    End_Attachments;
    Bindings
        Pujador1.Outport1 → Ourport1;
        CasaMunyecas.Outport1 → Ourport2;
        Pujador1.Inport1 ← Inport1;
        CasaMunyecas.Inport1 ← Inport2;
    End_Bindings;
End_System_type Subasta;
```

A continuación se presenta a modo de resumen la plantilla completa del lenguaje de configuración. Dicha plantilla permite especificar un modelo arquitectónico cualquiera, independientemente de su complejidad.

```
Architectural Model  <nombre modelo>
    Import Types
        Components
            <nombre tipo>,
            … … ;
    End_Import Types;
```

```
        Instances
              <nombre_instancia>: <nombre_tipo>;
                    ……
        End_Instances;
        Attachments
              Associated_connector <nombre conector>;
                          <formula_conexión>;
                          ……
              End_associated_connector <nombre conector>;
        End_Attachments;
        System_type name
              instances
                    <nombre_instancia>: <nombre_tipo>;
                    … …
              End_instances;
              Attachments
                    Associated_connector <nombre conector>;
                          <formula_conexión>;
                                ……
                    End_associated_connector <nombre conector>;
              End_Attachments;
              Bindings
                    <formula_enlace>;
                          ……
              End_Bindings;
    End_System_type Name;
End Arquitectural Model <nombre modelo>;
```

LENGUAJE GRÁFICO PRISMA

UML es un lenguaje de modelado de ámbito general y de gran difusión en el área de la ingeniería del software, siendo el lenguaje de modelado orientado a objetos más extendido. Sin embargo, existen dominios, como las arquitecturas software, cuyos conceptos no son contemplados por UML (versión 1.5) (www.omg.org).

Hoy en día, utilizar UML como lenguaje de modelado es una gran ventaja, ya que la mayoría de usuarios están familiarizados con él. Por este motivo, se ha considerado oportuno utilizar UML como lenguaje de modelado para PRISMA, si bien es cierto, su extensión es necesaria para soportar los nuevos conceptos que incorpora el modelo. Dicha extensión es posible debido a que UML proporciona un conjunto de mecanismos para extender su lenguaje. Estos mecanismos se basan en una arquitectura de 4-capas que establece 4 niveles de abstracción diferentes: nivel meta-meta modelo (N0), nivel meta-modelo (N1), nivel modelo (N2) y nivel objeto (N3). De forma que el nivel Ni describe los constructores que se pueden emplear para definir los modelos del nivel Ni+1. El metamodelo de UML esta definido en el nivel N1, mientras que el modelo MOF se corresponde con el meta-meta modelo de UML, que se encuentra definido en el nivel N0.

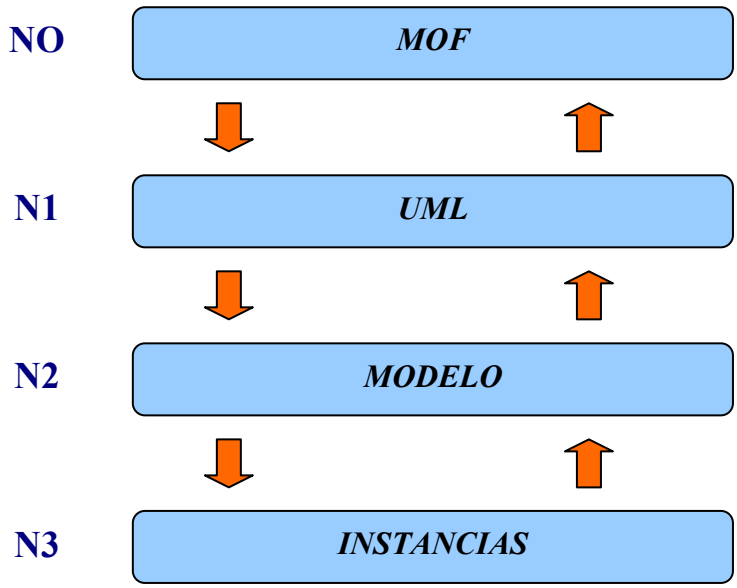

Figura 34. Arquitectura 4 capas

Existen varias propuestas para definir nuevos metamodelos en base a esta arquitectura: definir metamodelos como instancias directas del modelo MOF o hacer extensiones UML pesadas o ligeras. Anteriormente, ya se ha presentado la intención de realizar una extensión de UML para PRISMA y no la definición de un modelo nuevo no estándar, con el objetivo de preservar las ventajas que UML aporta. Los mecanismos de extensión UML se clasifican en pesados y ligeros. Las extensiones pesadas consisten en añadir nuevos conceptos al metamodelo UML, preservando la semántica de los elementos ya existentes. Mientras que las

extensiones ligeras usan los mecanismos de extensión de UML para especializar sus conceptos en un dominio específico. Este último tipo de extensión se conoce comúnmente como *profile*, el cual se construye a partir de estereotipos, valores etiquetados y restricciones.

Ambos mecanismos de extensión tienen ventajas e inconvenientes, pero en el caso de PRISMA se a optado por una extensión ligera, es decir, por la construcción de un profile PRISMA con el objetivo de un futuro adaptar una de las herramientas CASE existentes en el mercado y poder modelar arquitecturas software PRISMA para a partir de los modelos generar la aplicación de forma automática.

En este capítulo del trabajo se muestran las pautas que se han seguido para construir dicho profile UML, así como el profile resultante. Por otro lado se presenta el lenguaje gráfico asociado a cada uno de los elementos del profile.

5.1. Profile UML para PRISMA

La construcción del profile UML para PRISMA se ha realizado siguiendo los pasos propuestos en los trabajos [Ana03] y [Rib02]. Estos se resumen brevemente a continuación:

1.- Definición del metamodelo del dominio especifico para el que se desea extender UML.

2.- Determinar si cada concepto del metamodelo especificado en el punto anterior se corresponde semánticamente con una metaclase de UML..

3.- Obtención de un profile UML inicial para el dominio específico, estableciendo los estereotipos para los conceptos del metamodelo y sus valores etiquetados correspondientes.

4. – Completar el profile obtenido en el punto 3, mediante la despcripción textual de la semántica de cada concepto que lo compone y la incorporación de restricciones OCL.

En el caso de PRISMA, el primer paso de construcción del profile ya ha sido abordado y presentado en el capítulo 3 de este trabajo. Por lo tanto, la construcción del profile UML para PRISMA se va a abordar a partir del punto 2.

5.1.1. Clasificación de conceptos PRISMA en base al metamodelo UML

En este apartado se aborda el paso 2 para la construcción del profile UML. A continuación se muestra una tabla en la que se establece para cada uno de los conceptos del modelo PRISMA, si se corresponden exáctamente con una clase UML o no, especificando en caso afirmativo de qué clase se trata.

Metamodelo PRISMA	Correspondencia Exacta con Metamodelo UML
Aspecto	NO
ComponentePRISMA	NO
Sistema	NO
Conector	NO
Attachment	NO
Binding	NO
Sincronización	NO
AsociaciónPRISMA	Metaclase UML: Asociación
AgregaciónPRISMA	NO
Puerto	NO
Rol	NO
InterfazPRISMA	Metaclase UML: Interfaz
Weaving	NO
Protocolo	NO
Coreografía	NO

5.1.2. Profile UML para PRISMA

En este apartado se van a abordar simultáneamente los pasos 3 y 4 del proceso de construcción de un profile UML. Para ello se van a definir los estereotipos de cada uno de los conceptos PRISMA que no se corresponden con exactitud una clase UML, así como sus valores etiquetados, clases base, descripción textual en la que se les da semántica. Las restricciones necesarias para cada estereotipo no se van a incluir en este profile, se definiran en el momento en que el profile se incluya como add-in en una herramienta de modelado como Rational Rose o ArgoUML.

A la hora de encontrar la clase base UML del estereotipo asociado a un concepto del metamodelo, que se quiere incorporar como extensión de UML, se ha de utilizar como criterio de selección la similitud entre el modelado de la clase UML y lo que se pretende hacer gráficamente en el modelado del nuevo concepto. Este criterio, es el que se ha seguido para definir los estereotipos y clases bases de los conceptos del metamodelo PRISMA.

- **Aspecto**

La clase base del estereotipo <<aspect>> asociado al concepto de aspecto del modelo PRISMA es la metaclase UML *class* (ver figura 35). Esto es debido a que al igual que un aspecto, la metaclase *class* describe mediante una plantilla con atributos, operaciones,

métodos y semántica, tanto estructura como comportamiento. Además tienen en común el hecho de que *class* puede utilizar un conjunto de interfaces para especificar colecciones de operaciones que se proporcionan al exterior. La diferencia más relevante entre estos dos conceptos son el hecho de que un aspecto no describa un conjunto de objetos completamente, sino que lo define desde un determinado punto de vista (*concern*) y por lo tanto, incompleta.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Aspect <<Aspect>>	Class	N/A	Ninguno	Un **aspecto** especifica la estructura y el comportamiento de un tipo desde un determinado punto de vista (*concern*)

Por otro lado, cada uno de los distintos aspectos del modelo PRISMA, son estereotipos a su vez de <<Aspect>>.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Functional <<Functional>>	Aspect	Aspecto	Ninguno	Especifica la estructura y el comportamiento de un tipo desde el punto de vista **funcional**.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Distribution <<Distribution>>	Aspect	Aspecto	Ninguno	Especifica la estructura y el comportamiento de un tipo desde el punto de vista de **distribución**.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Coordination <<Coordination>>	Aspect	Aspecto	Ninguno	Especifica la estructura y el comportamiento de un tipo desde el punto de vista de **coordinación**.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Navegation <<Navegation>>	Aspect	Aspecto	Ninguno	Especifica la estructura y el comportamiento de un tipo desde el punto de vista de **navegación**.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Presentation <<Presentation>>	Aspect	Aspecto	Ninguno	Especifica la estructura y el comportamiento de un tipo desde el punto de vista de **presentación**.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Context-Awareness <<Context-Awareness>>	Aspect	Aspecto	Ninguno	Especifica la estructura y el comportamiento de un tipo desde el punto de vista de **context-awareness**.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Quality <<Quality>>	Aspect	Aspecto	Ninguno	Especifica la estructura y el comportamiento de un tipo desde el punto de vista de **calidad**.

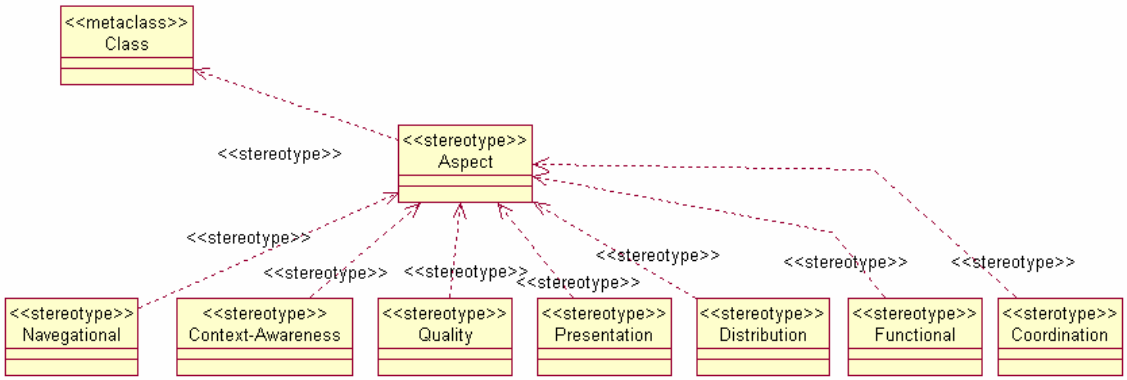

Figura 35. Modelado del estereotipo *Aspect*

Además de los valores etiquetados propios de la metaclase UML *class,* el estereotipo aspecto incluye otros nuevos para especificar con mayor detalle la semántica de los atributos y servicios del aspecto. A continuación se describen cada uno de los valores etiquetados dentro de una tabla:

Valor etiquetado	Estereotipo	Tipo	Multiplicidad	Descripción
Sychronization	<<Coordination>>	String XML	n	Una **sincronización** especifica la coordinación de servicios dentro de un mismo aspecto coordinación.

- **Componente**

La clase base del estereotipo <<component>> asociado al concepto de componente del modelo PRISMA es la metaclase UML *component* (ver figura 36). Se ha elegido *component* como clase base ya que es posible modelar mediante un diagrama de componente, las clases que la forman. Por lo que, el profile permitirá definir los aspectos que componen a una componente, debido a que *class* es la clase base de <<aspect>>. Además de esto, el concepto de componente de PRISMA y el de UML coinciden en que son una parte del sistema modular y reemplazable que expone un conjunto de interfaces. En dicha exposición de interfaces existe una diferencia, el hecho de que UML no las expone a través de puertos.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Component <<Component>>	Component	N/A	Ninguno	Un **componente** es un tipo del sistema de información que captura la funcionalidad del sistema, no actúa como coordinador entre otros tipos y se desarrolla para su posterior reutilización.

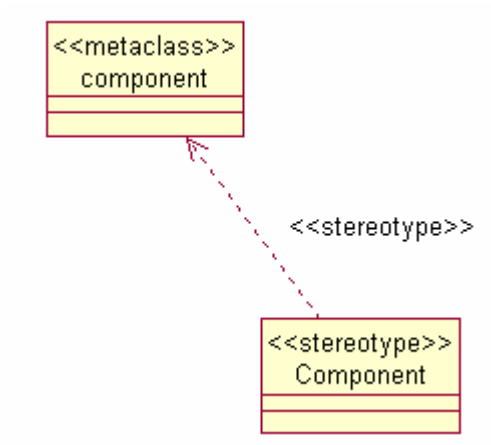

Figura 36. Modelado del estereotipo *Component*

- **Puerto**

La clase base del estereotipo <<Port>> asociado al concepto de puerto del modelo PRISMA es la metaclase UML *interface,* ya que el contenido de un puerto PRISMA es una interfaz y una interfaz PRISMA coincide con la interfaz de UML (ver figura 37).

Stereotype	Base Class	Parent	Valores etiquetados	Description
Port <<Port>>	Interface	N/A	Ninguno	Un **puerto** es un artefacto que permite la interacción de los componentes con el resto de tipos de la arquitectura.

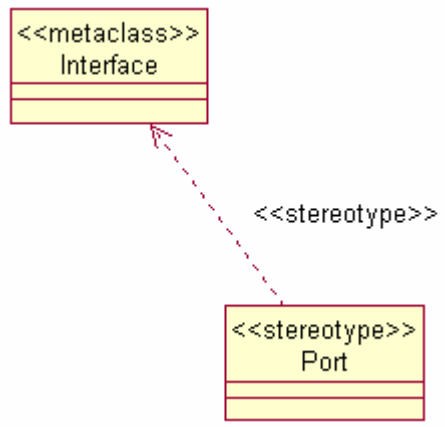

Figura 37. Modelado del estereotipo *Port*

- **Protocolo**

La clase base del estereotipo <<Protocol>> asociado al concepto de protocolo del modelo PRISMA es la metaclase UML *statemachine* (ver figura 38). Este estereotipo especifica comportamiento, de forma que ha de permitir modelar un proceso compuesto por un conjunto de estados y las transiciones entre ellos. Esto mismo, lo permite un diagrama de transición de estados, conocido en UML por *statemachine*. Existe otra razón para confirmar utilizar *statemachine* como clase base, el protocolo forma parte del aspecto dentro de una especificación PRISMA, mientras que el *statemachine* forma parte de una clase. Por lo tanto, debido a que la clase base del<<aspect>> es *class*, el diagrama de transición de estados UML cumple todas las necesidades de modelado de un protocolo PRISMA.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Protocol <<Protocol>>	StateMachine	N/A	Ninguno	Un **protocolo** define el conjunto de posibles vidas funcionales de un tipo de componente o de sistema.

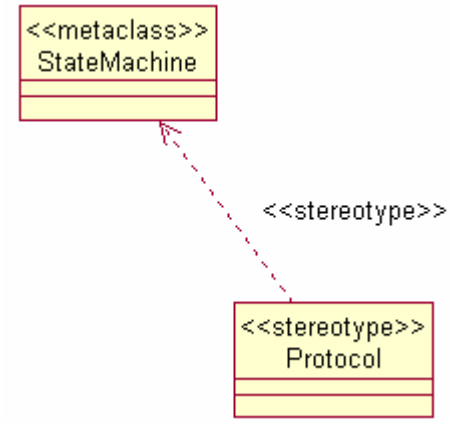

Figura 38. Modelado del estereotipo *Protocol*

- **Sistema**

La clase base del esteotipo <<system>> asociado al concepto de sistema del modelo PRISMA es la metaclase UML *component.* La clase base coincide con la de componente, ya que se trata del mismo concepto sólo que a distinto nivel de abstracción.

Stereotype	Base Class	Parent	Valores etiquetados	Description
System <<System>>	Component	Component PRISMA	Ninguno	Un **sistema** es un conjunto de tipos jugando un papel determinado dentro del sistema (elementos) que a su vez pueden estar o no conectados entre sí

- **Connector**

La clase base del estereotipo <<connector>> asociado al concepto de conector del modelo PRISMA es la metaclase UML *component* (ver figura 39). Se ha elegido *component* como clase base ya que es posible modelar mediante un diagrama de componentes las clases que la forman. Por lo que, el profile permitirá definir los aspectos que componen al conector, debido a que *class* es la clase base de <<aspect>>. Además de esto, el conector y la componente UML son una parte del sistema modular y reemplazable que expone un conjunto de interfaces, a pesar de que el conector lo haga a través de sus roles. Una característica que refina el conector frente a la componente UML, es el hecho de describir estructura y comportamiento, ya que el conector no lo hace desde el punto de vista funcional sino del de coordinación.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Connector <<Connector>>	Component	N/A	Ninguno	Un **conector** PRISMA es un tipo del sistema de información que actúa como coordinador entre varios tipos

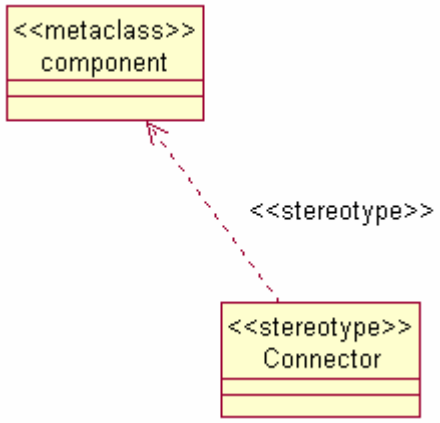

Figura 39. Modelado del estereotipo *Connector*

- **Rol**

La clase base del estereotipo <<Rol>> asociado al concepto de rol del modelo PRISMA es la metaclase UML *interface* (ver figura 40), ya que el contenido de un rol PRISMA es una interfaz y una interfaz PRISMA coincide con la interfaz de UML.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Rol <<Rol>>	Interface	N/A	Ninguno	Un **rol** es un artefacto que permiten la interacción de los conectores con el resto de tipos de la arquitectura. Dicha interacción consiste en la conexión y sincronización de componentes mediante conectores.

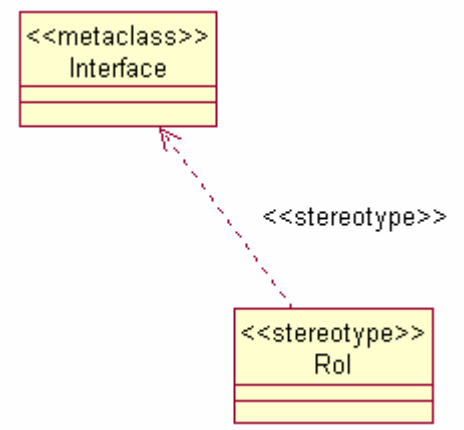

Figura 40. Modelado del estereotipo *Rol*

- **Weaving**

La clase base del estereotipo <<weaving>> asociado al concepto de weaving del modelo PRISMA es la metaclase UML *association* (ver figura 41). Se ha elegido *association,* porque el *weaving* realiza la sincronización inter-aspectos de un tipo y la clase base del concepto de *aspecto* es *class.* Por lo tanto, el concepto UML *association* es el que permite establecer una vinculación entre clases y en el profile para PRISMA, la sincronización inter-aspectos.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Weaving <<Weaving>>	Association	N/A	Ninguno	El **weaving** sincroniza los aspectos que componen un tipo.

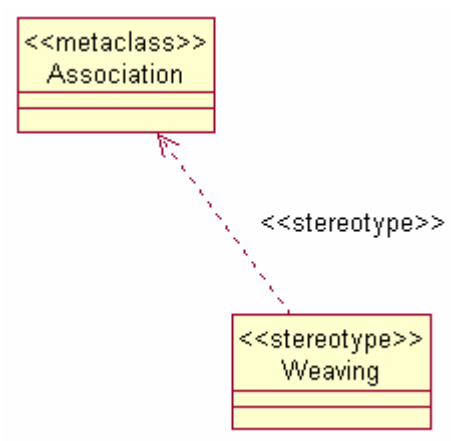

Figura 41. Modelado del estereotipo *Weaving*

- **Agregación**

La clase base del estereotipo <<aggregation>> asociado al concepto de agregación del modelo PRISMA es la metaclase UML *association* (ver figura 42). Se ha elegido *association,* ya que coincide exactamente con el concepto de asociación UML cuando cada uno de sus *associantionEnd* tienen en su atributo *aggregation* el valor *aggregate* y *composite,* respectivamente.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Aggregation <<Aggregation>>	Association	N/A	Ninguno	Un **agregación** PRISMA define que un componente pertenece a otro de forma estructural, representando la noción parte_de (**part_of)**

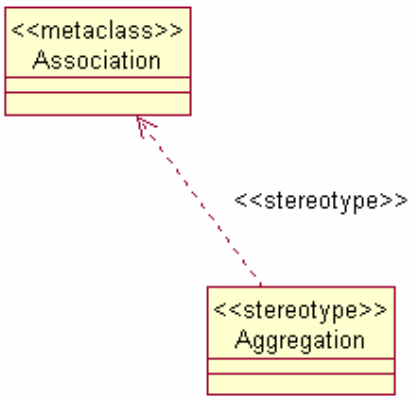

Figura 42. Modelado del estereotipo *Aggregation*

- **Coreografía**

La clase base del estereotipo <<Choreography>> asociado al concepto de coreografía del modelo PRISMA es la metaclase UML *statemachine* (ver figura 43). Este estereotipo especifica comportamiento, de forma que ha de permitir modelar un proceso compuesto por un conjunto de estado y sus transiciones entre ellos. Esto mismo, lo permite un diagrama de transición de estados, conocido en UML por *statemachine*. Existe otra razón para confirmar utilizar *statemachine* como clase base, la coreografía formar parte del aspecto dentro de una especificación PRISMA, mientras que el *statemachine* forma parte de una clase. Por lo tanto, debido a que la clase base de <<aspect>> es *class*, el diagrama de transición de estados UML cumple todas las necesidades de modelado de una coreografía PRISMA.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Choreography <<Choreography>>	StateMachine	N/A	Ninguno	La **coreografía** especifica el proceso de coordinación que aplica el conector para sincronizar a los tipos que conecta

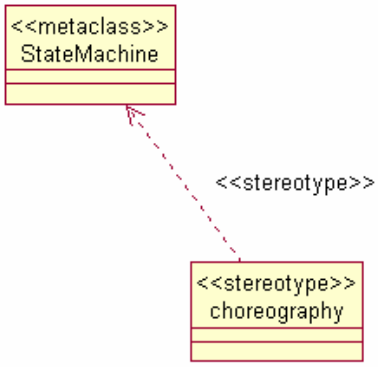

Figura 43. Modelado del estereotipo *Choreography*

- **Attachment**

La clase base del estereotipo <<attachment>> asociado al concepto de attachment del modelo PRISMA es la metaclase UML *dependency* (ver figura 44). Se ha elegido *dependency,* porque el *attachment* realiza la conexión entre componentes y conectores a traves de sus puertos y roles, respectivamente. Teniendo en cuenta que la clase base de los estereotipos <<component>> y <<connector>> es *component* y que *dependecy* es el concepto que permite modelar la dependencia entre componentes UML, este concepto permite relacionar ambos estereotipos.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Attachment <<Attachment>>	Dependency	N/A	Ninguno	Un **attachment** establece un canal de conexión entre un conjunto de instancias o elementos de componentes y conectores a través de sus puertos y roles, respectivamente.

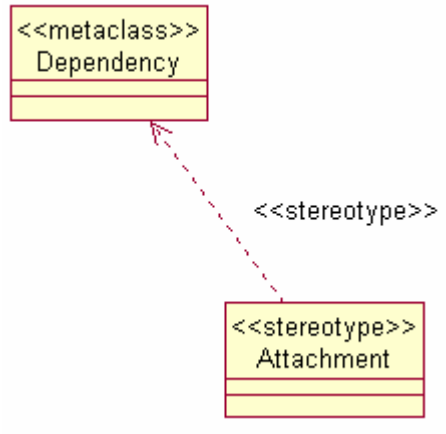

Figura 44. Modelado del estereotipo Attachment

- **Binding**

La clase base del estereotipo <<binding>> asociado al concepto de *binding* del modelo PRISMA es la metaclase UML *dependency* (ver figura 45). Se ha elegido *dependency,* porque el *binding* realiza la conexión entre componentes y sistemas a traves de sus puertos. Teniendo en cuenta que la clase base de los estereotipos <<component>> y <<system>> es *component* y que *dependecy* es el concepto que permite modelar la dependencia entre componentes UML, este concepto permite relacionar ambos estereotipos.

Stereotype	Base Class	Parent	Valores etiquetados	Description
Binding <<Binding>>	Dependency	N/A	Ninguno	El **binding** establece la conexión entre una instancia o elemento encapsulado y la instancia o elemento del sistema que lo encapsula.

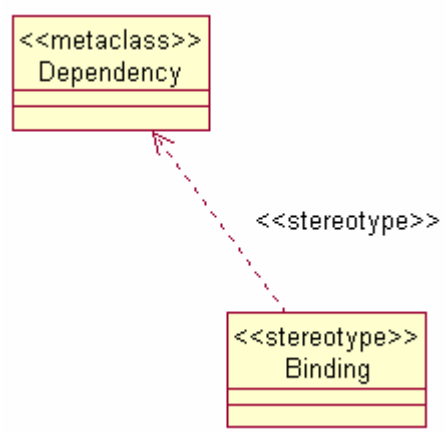

Figura 45. Modelado del estereotipo *Binding*

5.2. Lenguaje Gráfico PRISMA

En este apartado se va a presentar el elemento gráfico asociado a cada uno de los estereotipos considerados en el apartado anterior. Los ejemplos gráficos van a ser los mismos que los que se han presentado en el capitulo de los lenguajes textuales.

5.2.1. Elemento Gráfico de Aspecto

El aspecto se representa como un cuadrado en la que se especifica la plantilla asociada. Los distintos tipos de aspecto se caracterizan por tener un color asociado. De forma que, el aspecto de funcional es amarillo, el aspecto distribución es lila, el aspecto de coordinación es verde, el aspecto navegacional es azul, el aspecto de presentación es naranja , el aspecto de calidad es rojo y el aspecto context-awareness es de color gris.

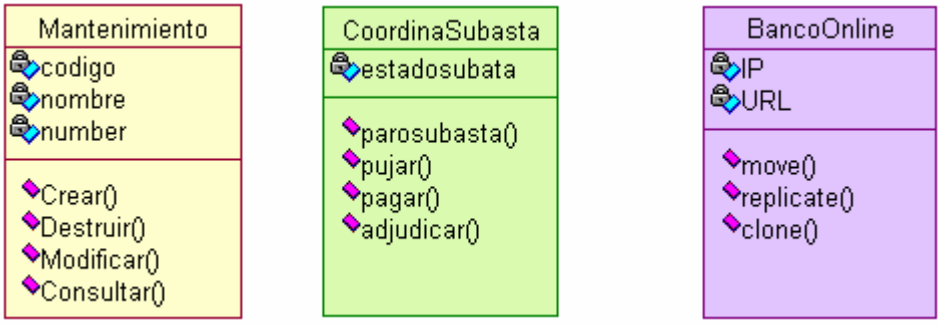

Figura 46. Aspectos Gráficos de Funcionalidad, Coordinación y Distribución

5.2.2. Elemento Gráficos de Interfaz

La interfaz PRISMA se representa gráficamente igual que una interfaz UML. A continuación se presenta un ejemplo:

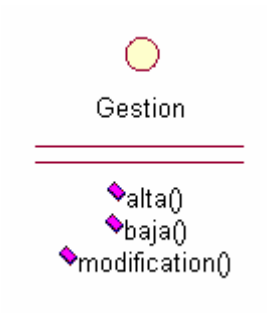

Figura 47. Interfaz Gráfica

De manera que es posible representar gráficamente que un aspecto especifica una determinada interfaz como se muestra a continuación:

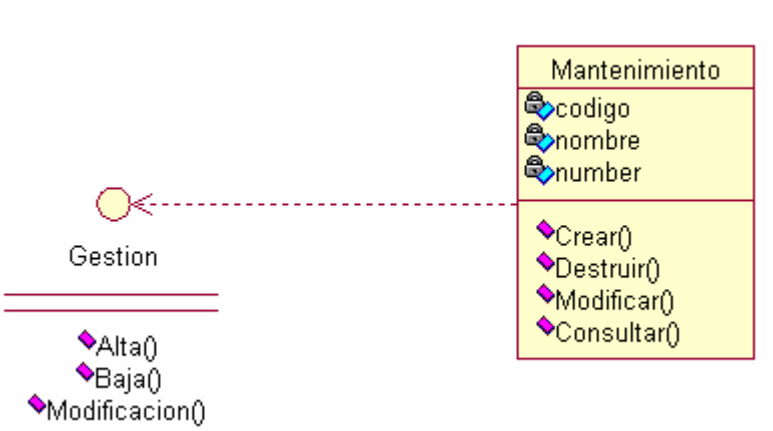

Figura 48. Enlace Gráfico entre Interfaces y Aspectos

5.2.3. Elementos Gráficos de Componente, Conector y Sistema

Tanto el componente como el conector y el sistema se representa como una caja con tantas patillas como puertos o roles tengan. Las patillas de la izquierda son puertos o roles de entrada, mientras que las patillas de la derecha son puertos o roles de salida. La diferencia entre componente, conector y sistema es el color, los componentes son azules, los conectores verdes y los sistemas naranjas.

Figura 49. Componente Gráfico

El componente, al igual que el conector, se construyen seleccionando aquellos aspectos que interesen, tal y como se muestra a continuación:

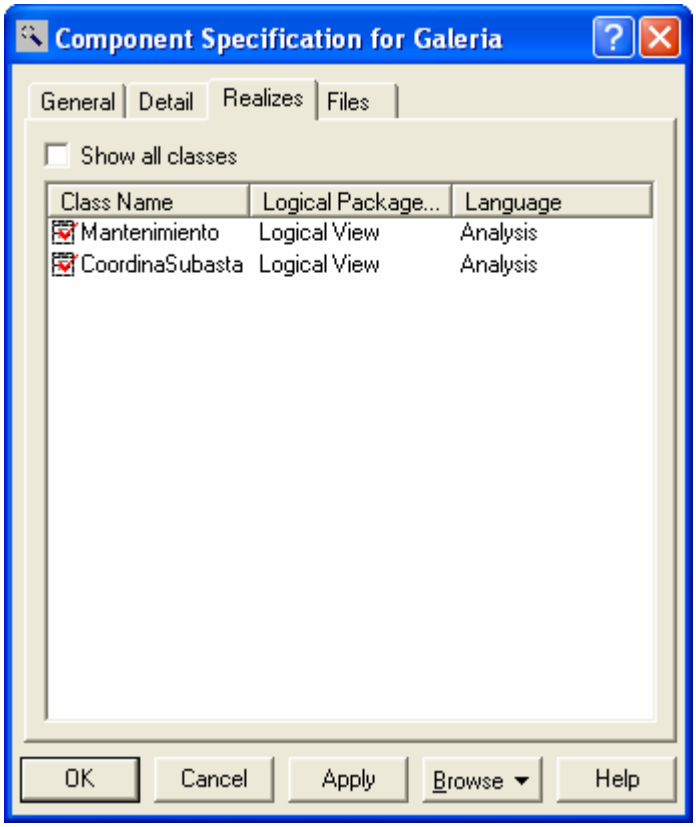

Figura 50. Definición gráfica de componentes a partir de aspectos

Una vez especificados los aspectos que forman el componente o el conector, éstos tienen dos visiones internas posibles, la visión aspectos y la visión interfaces. Ésta última visión permite asignar a cada puerto su interfaz.

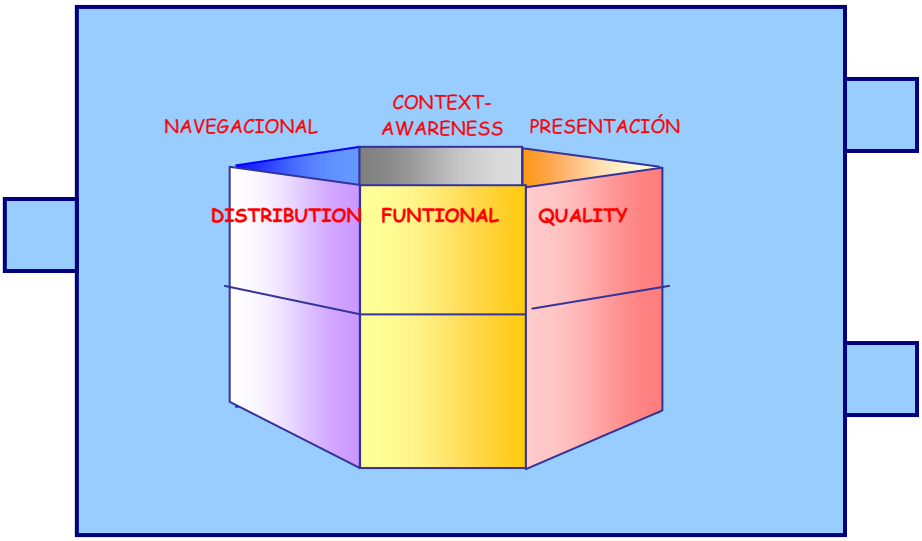

Figura 51. Visión gráfica aspecto de un componente

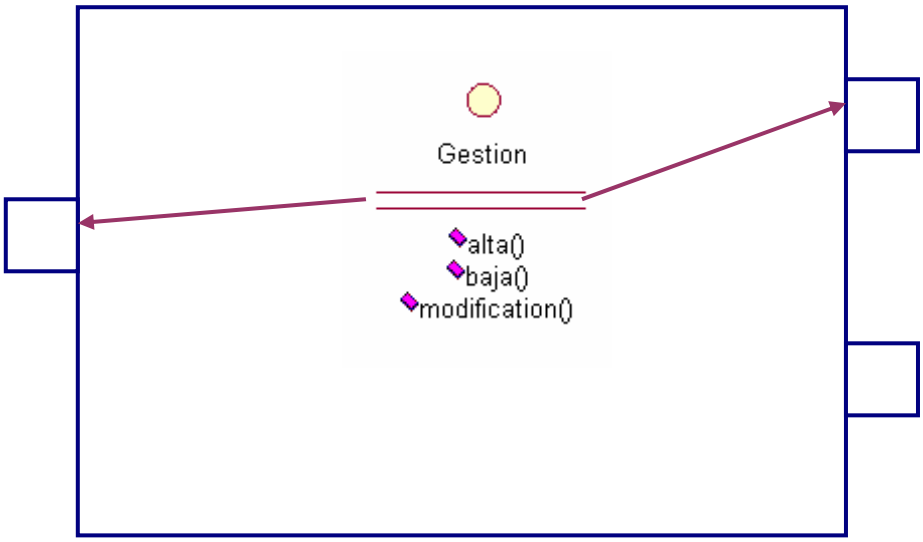

Figura 52. Visión gráfica interfaz de un component

5.2.4. Elemento Gráfico de Protocolos y Coreografías

Los protocolos y coreografías se especifica mediante un diagrama de transición de estados, de forma que cada uno de los estados del protocolo o coreografía se representan mediante un rectángulo y las transiciones entre ellos haciendo uso de flechas. Se ha de tener en cuenta que los estados de creación y destrucción tienen una representación diferente, el de creación un círculo negro y el de destrucción un círculo negro con otro círculo blanco que lo engloba.

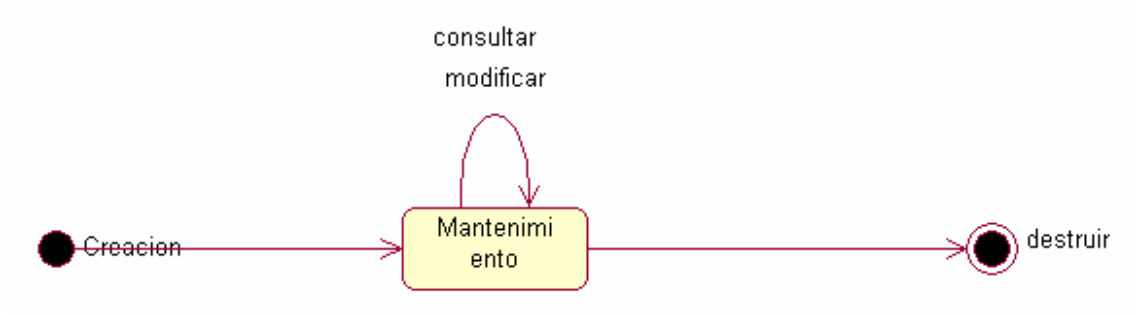

Figura 53. Protocolo Gráfico

5.2.4. Elemento Gráfico de Weaving

El weaving se especifica mediante una relación de agregación entre aspectos de diferente tipo. A continuación se muestra un ejemplo:

Figura 54. Weaving Gráfico

5.2.4. Elemento Gráfico de Attachment

El attachment se especifica mediante una relación de dependencia entre los puertos y los roles de los componentes y conectores. Del mismo modo se representa una relación de asociación, solo que la semántica es diferente. A continuación se muestra un ejemplo:

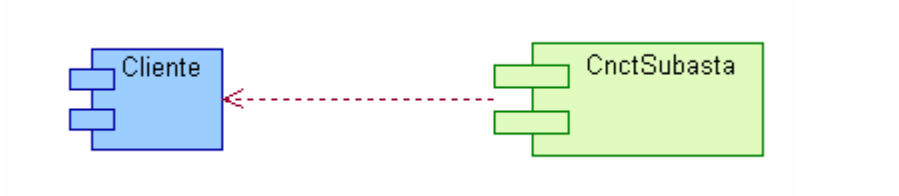

Figura 55. Attachment Gráfico

5.2.5. Elemento Gráfico de Binding

El binding se especifica mediante una relación de dependencia entre los puertos del los componente y el sistema. Del mismo modo se representa una relación de agregación, solo que la semántica es diferente. A continuación se muestra un ejemplo:

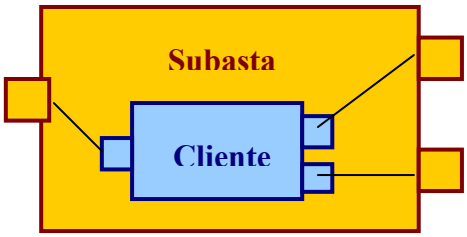

Figura 56. Binding Gráfico

CONCLUSIONES Y TRABAJOS FUTUROS

Se ha propuesto un modelo para la especificación de arquitecturas software basado en aspectos y componentes, llamado PRISMA. Dicho modelo se caracteriza por la integración que realiza de las aproximaciones de desarrollo DSBC y DSOA y las propiedades reflexivas a través de un metanivel que le da capacidad de evolución. PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables. El modelo ha sido representado detalladamente mediante un metamodelo.

Los modelos arquitectónicos de PRISMA se especifican mediante dos lenguajes, un lenguaje de definición de componentes y un lenguaje de configuración, ofreciendo la posibilidad de importar componentes COTS y reutilizar aquellos componentes que se especifiquen.

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

Se han tratado diversos aspectos del lenguaje de definición de componentes: funcional, distribución, coordinación, calidad, *context-awareness*, navegación, presentación, etc. Aunque han sido tratados con diferente nivel de detalle, debido a que algunos de ellos son elementales para el modelo básico y otros se utilizan para a ámbitos específicos de aplicación.

Además de esto, se ha presentado el profile diseñado para el modelado de arquitecturas PRISMA haciendo uso de un lenguaje estándar como UML. Su construcción permite utilizar las herramientas existentes de modelado como Rational Rose o ArgoUML para especificar gráficamente modelos PRISMA. Finalmente, se ha hecho una propuesta de lenguaje gráfico para cada uno de los conceptos del modelo PRISMA.

Como resultado de este trabajo se ha obtenido las publicaciones que se citan a continuación:

- **Jennifer Pérez**, Isidro Ramos, Javier Jaén and Patricio Letelier, Elena Navarro, *PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures,* 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 (Pendiente de Publicación)
- **Jennifer Pérez** , Isidro Ramos , Javier Jaén, Patricio Letelier, *PRISMA: Development of Software Architectures with an Aspect Oriented, Reflexive and Dynamic Approach,* Dagstuhl Seminar Nº 03081, Report Nº 36 "Objects, Agents and Features",Copyright (c) IBFI gem. GmbH, Schloss Dagstuhl, D-66687 Wadern, Germany . Eds.H.-D. Ehrich (Univ. Braunschweig, D), J.-J. Meyer (Utrecht, NL), M. Ryan (Univ. of Birmingham, GB)

- Elena Navarro, Isidro Ramos, **Jennifer Pérez**, *Software Requirements for Architectured Systems*, 11t h IEEE International Requirements Engineering Conference (RE'03), September 8-12, 2003 in Monterey, California. (Pendiente de Publicación)
- **Nour Hussein**, Josep Silva, Javier Jaen, Isidro Ramos, Jose Angel Carsí , Jennifer Peréz, *Mobility and Replicability Patterns in Aspect-Oriented Component- Based Software Architectures*, 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS), Marina del Rey, California, USA, 3-5 noviembre 2003.(Pendiente de Publicación)
- **Jennifer Pérez**, Isidro Ramos , Ángeles Lorenzo, Patricio Letelier, Javier Jaén, *PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos*, Actas VII Jornadas de Ingeniería del Software y Bases de Datos, JISBD, Escorial (Madrid), noviembre 2002

Por otro lado, se han enviado dos trabajos más que todavía no han sido evaluados, las referencias son las siguientes:

- **Jennifer Pérez**, Isidro Ramos, Javier Jaén and Patricio Letelier, *PRISMA: Aspect Oriented and Dynamic Modelling of Component and Agent Societies*, "Objects, Agents and Features Book", Lecture Notes in Computer Science (LNCS) by Springer Verlag. (Enviado)
- **Jennifer Pérez**, Nour Hussein, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, *Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA,* VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003 (Enviado)

Las líneas del trabajo futuras son muchas debido a la amplitud del modelo. Los trabajos a corto plazo se centran en la reconfiguración dinámica de los modelos arquitectónicos que se especifiquen, en la elaboración del add-in para Rational Rose o ArgoUML, en los aspectos del prisma, tanto para incorporar nuevos aspectos como para especificar de una forma más detallada los aspectos de navegación, presentación, coordinación y evolución. Dentro de cada aspecto se han de realizar las siguientes tareas:

- Aspecto de navegación: Realizar un estudio del estado del arte más exhaustivo y diseñar un modelo de navegación .
- Aspecto de presentación: Al igual que el de navegación se ha de realizar un estudio del estado del arte y proponer un modelo de presentación.
- Aspecto de coordinación: Analizar y hacer una comparativa detallada entre π-calculo y el lenguaje MSC con el objetivo de ver cual de ellos proporciona más expresividad y menos ambigüedad a la hora de especificar sincronizaciones y coreografías.
- Aspecto de evolución: Esta tarea pertenece al trabajo de investigación que se presenta junto a éste. Si bien es cierto, en éste trabajo tan sólo se hará una primera aproximación con los conceptos más básicos y posteriormente se deberá establecer un

patronaje detallado para cada uno de los casos de evolución que se presenten en el modelo.

Los trabajos a largo plazo consistirán en la aplicación de este modelo al campo de los sistemas hipermedia y la construcción de un compilador de modelos para PRISMA, de forma que a partir del lenguaje gráfico se genere el lenguaje textual y que a partir del lenguaje textual se genere código fuente.

BIBLIOGRAFÍA

[Ald01] O. Aldawud, T. Elrad, A. Bader, *A UML Profile for Aspect Oriented Modeling,* Workshop on Aspect Oriented Programming, OOPSLA 2001.

[All97] R. Allen & David Garlan, *A Formal Basis for Architectural Connection*, School of Computer Science, Carnegie Mellon University,Pittsburgh, January, 1997.

[All98] P.Allen y S.Frost, Component-Based Development for Enterprise Systems. Applying the SELECT Perspective TM , Cambridge University Press, 1998.

[Ana03] V. Anaya, P. Letelier, *Hacia un Método para la Elaboración de Profiles UML: un Caso de Estudio,* VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003 (Enviado)

[Aos] "Aspect Oriented Software Development", http://aosd.net

[Bal85] R. Balzer, *"A 15 Year Perspective on Automatic Programming"*. IEEE Transactions on Software Engineering, vol.11, num.11, pages 1257-1268, November 1985.

[Bal93] Balasubramanian V. Hypermedia Issues and Applications: A State-of-the Art Review, www, Diciembre 1993.

[Bar01] M.R. Barbacci, D.Doubleday, C. B. Weinstock y R. W. Lichota, *DURRA: An Integrated Approach to Software Specification, Modeling and Rapid Prototyping.* Technical Report CMU/SEI-91-TR-21, Software Engineering Institute (SEI), septiembre 1991.

[Ber99] B. Berkem, Traceability management from business processes to use cases with UML. A proposal for extensions to the UML's activity diagram through goal-oriented objects, Journal of Object-Oriented Programming, páginas 29-34, 1.999.

[Bri03] I Brito, A. Moreira. *Twoards a Composition Process for Aspect-Oriented Requirements*. Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design, workshop of the 2nd International Conference on Aspect-Oriented Software Development,Boston, USA, 17 March 2003.

[Can00] C. Canal Velasco, Un lenguaje para la Especificación y Validación de Arquitectecturas Software, Tesis Doctoral, Universidad de Malaga, 2000.

[Che02] Cheng S., Garlan D., Schmerl B., Steenkiste P., Hu N., *Software Architecture-based Adaptation for Grid Computing* , IEEE Conference on High Performance Distributed Computing. Scotland, July, 2002.

[Cla01] Clark R. and Walker R. J., *Composition Patterns: An Approach to Designning Reusable Aspects,* International Conference on Software Engineering (ICSE) 2001.

[Cue02] C.E. Cuesta Quintero, *Arquitectura de Software Dinámica Basada en Reflexión,* Tesis Doctoral, Departamento de Informática, Universidad de Valladolid, 2002.

[Dey00] Dey A. K. & Abowd, G. D. (2000a). Towards a better understanding of context and context-awareness. Proceedings of the Workshop on the What, Who, Where, When and

How of Context-Awareness, affiliated with the CHI 2000 Conference on Human Factors in Computer Systems, New York, NY: ACM Press.

[Elr01] Elrad T., Filman R. and Bader A., *Theme Section on Aspect-Oriented Programming*, Communication of ACM, Vol. 44, No. 10,2001

[Fow96] Fowler, M., *Analisys Patterns: Reusable Object Models,* Addison Wesley, Reading, MA, 1996.

[Fow00] Martin Fowler, *The New Methodology*, noviembre 2001, http://www.martinfowler.com

[Gam95] Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software Addison Wesley, Reading, MA, 1995.

[Gar95] D. Garland y D.E. Perry, *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering, 21(4), abril 1995.

[Loq00] Loques O., Sztajnberg A., Leite J., Lobosco, M., *On the Integration of Meta-Level Programming and Configuration Programming*, In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[Hay95] Hay, D., Data Model Patterns: Elements of Reusable Object-Oriented Software, Dorset House, NY, 1995.

[Hoo99] C. L. Hoover, P. K. Khosla, *Analytical partition of software components for evolvable and reliable mems design tools,* Int. Journal of software Engineering and Knowledge Engineering, 9(2):153-171, 1999.

[Ira00] D. Irastorza, A. Jaime and O. Díaz, *Diseño basado en componentes: alternativas en el paso de partición*, Jornadas de ingeniería del software y bases de datos (JISBD) 2000.

[Jac97] I. Jacobson, M Griss, P. Jonsson, Software Reuse. *Architecture, Process and Organization for Business Success*, Addison-Wesley 1.997.

[Jae03a] J. Jaén and I. Ramos, *A Conceptual Model for Context-Aware Dynamic Architectures*, 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with ICDCS2003, May 19-22, 2003 Providence, Rhode Island USA.

[Jae03b]J. Jaen, J.H. Canos, *A Grid Architecture for Building Hybrid Museums*, 2nd International Human.Society@Internet Conference. To Appear at Springer-Verlag LNCS. Seul 2003

[Kic01] G. Kiczales, E. Hilsdale, J. Huguin, M. Kersten, J. Palm, W.G. Griswold, *An Overview of AspectJ*, proceeding of the European Conference on Object-Oriented Programming, Springer-Verlag, 2001.

[Kiz97] G. Kiczales et al., *Aspect-Oriented Programming*, Proc. European Conference on Object-Oriented Programming (ECOOP' 97). LNCS 1241, Springer Verlang, 1997.

[Lar99] G. Larsen, *Designing component-based frameworks using patterns in the UML*, Communications of the ACM, 42(10):38-45, 1.999.

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "*Un enfoque formal para el modelado conceptual orientado a objeto*". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721- 663-0, 1998.

[Lio02] J. M. Lions, D. Simoneau, G. Pitette, Imed Moussa, *Extending OpenTool/UML Using Metamodeling: An Aspect Oriented Programming Case Study,* Second International Workshop on Aspect-Oriented Modeling with UML, Fifth International Conference on the Unified Modeling Language - the Language and its Applications, Dresden (Germany), octubre 2002.

[Loq00] Loques O., Sztajnberg A., Leite J., Lobosco, M., *On the Integration of Meta-Level Programming and Configuration Programming*, In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

[Loz01] M. Lozano, P. González, I. Ramos*, IDEAS: An Object-Oriented Environment for developing User Interfaces,* The 14th Annual ACM Symposium on User Interface Software and Technology. UIST'01. Orlando, Florida. Noviembre, 2001.

[Loz00] M. Lozano, I. Ramos, P. González, *"User Interface Specification and Modeling in an Object Oriented Environment for Automatic Software Development"*. IEEE 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2000) ISBN: 0-7695-0774-3. Págs .373-381. Santa Barbara, CA USA. 31 de Julio-3 de Agosto,2000.

[Mag95] J.N. Magee, N. Dulay, S.Eisenbach, J. Kramer *Specifying Distributed Software Architectures,* Fifth European Software Engineering Conference (ESEC), septiembre, 1995.

[McG02] McGurren F., Conroy D., "*X-ADAPT: An Arquitecture for Dynamic Systems"*, Workshop on Component-Oriented Programming, ECOOP, Málaga, Spain, 2002.

[Med00] Medvidovic N., Taylor R.N., "*A classification and Comparison Framework for Software Architecture Description Languages",* IEEE Transactions of SW Engineering, Vol. 26, nº 1, January 2000

[Mil91]Robin Milner, π- C*álculo Poliadico: Un tutorial,*1991

[Nav03] E. Navarro, I. Ramos, J. Pérez, *Software Requirements for Architectured Systems*, 11t h IEEE International Requirements Engineering Conference (RE'03), September 8-12, 2003 in Monterey, California. (Pendiente de Publicación)

[Nor98] P. Noriega, C. Sierra, *Subastas y Sistemas Multiagente,. Revista Iberoamericana de* Inteligencia Artificial, Number 6, pp. 68-84, 1998.

[Nou03] N. Hussein, J. Silva, J. Jaen, I. Ramos, J. A. Carsí , J. Peréz, *Mobility and Replicability Patterns in Aspect-Oriented Component- Based Software Architectures*, 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS), Marina del Rey, California, USA, 3-5 noviembre 2003.(Pendiente de Publicación)

[Oli98] Oliva A., Garcia I.C., Buzato L.E., *The Reflective Architecture of Guaraná*, Technical Report IC-98-14. Instituto de computación, Universidad de Campiñas, abril, 1998.

[Pas97] Pastor O. Et al, *OO-METHOD: A Software Production Environment Combining Conventional and Formal Methods*, Procc. of 9th International Conference, CaiSE97, Barcelona, 1997.

[Per02a] J. Pérez, V. Anaya, J M. Cubel, F. Domínguez, A. Boronat, I. Ramos, J A. Carsí, *Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas,* Software Evolution Through Transformations: Towards Uniform Support throughout the Software Life-Cycle Workshop (SET'02), First International Conference on Graph Transformation(ICGT2002) , Barcelona (Spain), octubre 2002 © ENTCS vol n. 72.4

[Per02b] J. Pérez, J A. Carsí, I. Ramos, *ADML: A Language for Automatic Generation of Migration Plans,* The First Eurasian Conference on Advances in Information and Communication Technology, Tehran, Iran, octubre 2002 http://www.eurasia-ict.org/ © Springer LNCS vol n.2510

[Per02c] J. Pérez, J A. Carsí, I. Ramos, *On the implication of application's requirements changes in the persistence layer: an automatic approach,* Database Maintenance and Reengineering Workshop (DBMR'2002), IEEE International Conference of Software Maintenance, Montreal (Canada), October 1st, 2002, pag. 3-16, ISBN: 84-699-8920-0.

[Per03a] J. Pérez, I. Ramos, J. Jaén and P. Letelier, E. Navarro: "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*",* 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003 (Pendiente de Publicación)

[Per03b] Jennifer Pérez, Nour Hussein, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, *Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA,* VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante 2003 (Enviado)

[Pre02] Christian Prehofer, *Feature Interactions in Statechart Diagrams or Graphical Composition of Components,* DoCoMo EuroLabs Germany, Second International Workshop on Aspect-Oriented Modeling with UML, Fifth International Conference on the Unified Modeling Language - the Language and its Applications, Dresden (Germany), octubre 2002.

[Ras01] Rashid A., *A Hybrid Approach to Separation of Concerns: The Story of SADESK,* Proc. Reflection conf. 2001, Springer-Verlang, LNCS, 2192, pp. 231-249

[Ras02] A. Rashid, P. Sawer, A. Moreira, J. Araújo. “Early Aspects: a Model for Aspect-Oriented Requirements Engineering”, International Conference on Requirements Engineering (RE 2002), University of Essen, Germany, September 9-13, 2002.

[Rat] Rational Software, Rational Rose, http://www.rational.com/products/rose/

[Rib02] J.M.Ribo, X. Franch, *Una Metodología a dos Niveles para Extender el Metamodelo UML,* VII Jornadas de Ingeniería del Software y Bases de Datos, El Escorial (Madrid) 2002

[Rui02] Ruiz A., Una Aproximación Semicualitativa al Tratamiento Automático de Requisitos de Calidad: Aplicación a la Obtención Automática de Acuerdos de Nivel de Servicio en MOWS, Ph.D. Thesis 2002. Departamento de Lenguajes y Sistemas Informáticos (Univesidad de Sevilla)

[San99] F. Sánchez, Modelo de disfraces: hacia la adaptabilidad de restricciones de sincronización en los LCOO, Tesis doctoral, Universidad de Extremadura, 1999.

[Ser94] Sernadas A., Costa J.F., Sernadas C., "Object Specifications Through Diagrams: OBLOG Approach" INESC Lisbon 1994.

[Sha94] M. Shaw, Procedure Calls Are the Assembly Language of Software Interconection: Connectors Deserve First-Class Status, Proceedings of Workshop on Studies of Software Design, Enero 1994

[Sou99] D.D’Souza, A.Wills, Objects, Components and Frameworks with UML. The Catalysis approach, Addison-Wesley 1.999.

[Suz99] J. Suzuki, Y. Yamamoto, *Extending UML with Aspects: Aspect Support in the Desing Phase,* Workshop on Aspect-Oriented Programming (AOP) at ECOOP 1999.

[Sys] System Architec, http://www.popkin.com/products/sa2001/systemarchitect.htm.

[Tar99] Rarr P.L., Ossher H., Harrison W.H. and Sutton S. M., *N Degrees pf Separation: Multi-Dimensional Separation of Concerns*, Internacional Conference on Software Engineering (ICSE), 1999, ACM, pp. 1907-119.

[Tog] TogetherSoft Corporation http://www.togethersoft.com/

[Voa98] J. Voas, *Maintaining Component-Based Systems,* IEEE Software, July/August, páginas 22-27, 1998.

[Win97] Winograd, T., The Design of Interaction. Beyond Calculation. The Next Fifty Years of Computing. Spring-Verlag, New York, 149-265, 1997.

[Xer02] Xerox PARC, USA, *AspectJ Home Page*, http://aspectj.org/, 2002

[Zin97] Zinky J., Bakken D., Schantz R., *“Architectural Support for Quality of Service for CORBA Objects,”* Theory and Practice of Object Systems 3(1), 1997.

ANEXO: CASO ESTUDIO
SUBASTAINTERACTIVA,
Un portal Web de Subastas

A.1. Descripción del Caso de Estudio

El portal Web SUBASTAINTERACTIVA es mantenido por un conjunto de **administradores**. Todo administrador del portal puede dar de alta a otros administradores y una vez dado de alta un administrador puede modificar su perfil en todo momento, así como darse de baja del sistema.

La institución electrónica SUBASTAINTERACTIVA tiene galerías en las que se exponen los artículos que en breve saldrán a subasta, detallando las características más importantes de cada uno de ellos. Aquellos usuarios que visitan su portal Web pueden entrar en dichas galerías y pujar posteriormente por aquellos artículos que les hayan resultado de interés.

Las galerías de artículos de SUBASTAINTERACTIVA están clasificadas por categorías de artículos (ver figura 57), cada galería se compone de varias salas que se corresponden con las subcategorías de artículos de la galería (ver figura 58). Los administradores se encargan de crear nuevas galerías, incluir nuevos artículos que salen a subasta y eliminar los que han sido subastados.

Figura 57. Galerías clasificadas por categorías

Figura 58. Salas de una galería clasificadas por categorías

Además, para aquellos usuarios que no desean visitar ninguna galería porque buscan un artículo en particular, pueden realizar búsquedas avanzadas sobre estos (ver figura 59).

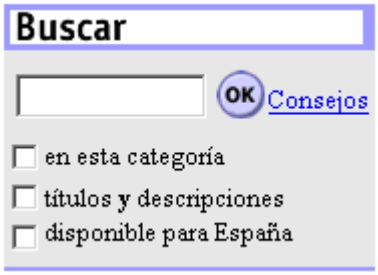

Figura 59. Búsqueda avanzada de artículos

Cada artículo que se expone en una de las galerías de SUBASTAINTERACTIVA (ver figura 60), contiene una descripción de sus características más relevantes, de su historia y por supuesto la subasta en la que saldrá a la venta, indicando el día y la hora de ésta.

Figura 60. Exposición de artículos de una galería

Aquellos usuarios que se conectan al portal Web con la intención de poner a subasta algún artículo (**vendedores)**, han de proporcionar al **registro de artículos** sus datos personales (ver figura 61), así como los datos descriptivos del artículo y una foto.

Figura 61. Alta subastador

Aquellos usuarios que desean participar en las subastas que se realizan en SUBASTAINTERACTIVA se han de dar de alta como compradores del sistema (**compradores)**. Para ello, se han de rellenar los datos personales necesarios, los datos bancarios y la forma de pago en el **registro de compradores** (ver figura 62)**.** Por otro lado, dichos compradores cuando quieran participar en una sesión de puja, se han de registrar en ésta con anticipación a su hora de comienzo. Posteriormente, dicho registro será comprobado por el **subastador**, antes de comenzar la subasta, con el fin de averiguar si el comprador esta clasificado como moroso, para que en ese caso se deniegue su participación.

Figura 62. Alta comprador

En SUBASTAINTERACTIVA acontecen varias sesiones de subasta simultáneamente (ver figura 63), cada una de ellas se diferencia por un tema y es dirigida por un gestor de subasta. Se ha de tener en cuenta, que esto hace que un comprador pueda participar en dos sesiones de subasta simultáneamente.

Cada una de las subastas que se programan en el portal son introducidas y mantenidas por el administrador de SUBASTAINTERACTIVA. Cada subasta tiene asignado un subastador, un día, una hora y un tema, correspondiéndose dicho tema a una categoría o subcategoría de alguna de las galerías o salas del portal. Por otro lado, el administrador de forma progresiva va incluyendo los artículos que estima oportunos en cada una de las subastas que ha programado.

Imagen	Artículos actuales - En venta actualmente	Precio	Pujas	Tiempo restante
	Vagón de Payá de los años 50	EUR 48,00	1	6h 38m
	MUÑECA LENCY UNA PIEZA DE MUSEO	EUR 750,00	-	8h 50m
	APISONADORA VAPOR WILLESCO	EUR 328,00	-	9h 31m
	LA VACA QUE RIE	EUR 0,50	-	12h 55m
	CASITA DE RESINA	EUR 0,50	-	12h 56m
	COCHE DODGE DAL BARREIROS DE LA CASA JYE	EUR 90,00	-	23h 49m
	FIGURITAS de PLOMO, carrera caballos	EUR 20,00	-	1d 00h 56m

Figura 63. Subastas simultáneas

Antes de comenzar la subasta, el subastador ha de comprobar que se hayan registrado al menos 3 compradores, que se subastan al menos 2 artículos y establecer el precio de salida de cada uno de ellos, así como el orden de aparición de cada uno de ellos en la sesión de subasta.

Una vez comienza una subasta el subastador selecciona el primer artículo a subastar y lo expone a los miembros participantes junto a su precio de salida. A partir de ese momento, los compradores pujan aumentando el importe del artículo respecto al del precio de salida, teniendo en cuenta que el incremento no tiene por qué ser una cantidad constante. Si el subastador, durante un periodo de tiempo de 3 minutos, comprueba que no se ha pujado por el artículo, dará una última oportunidad en la puja y si no se produce, el artículo será adjudicado al comprador que realizó la última y ser retirará de la subasta. Una vez se retire el artículo, se procederá a subastar el siguiente. Finalmente, el vendedor envía un correo electrónico al comprador con la finalidad de ejecutar la transacción de venta, adjuntando el albaran rellenado para el envío. Se han de tener en cuenta dos cosas, los compradores puede retirarse de una sesión de puja siempre que deseen, durante el transcurso de la subasta o en su finalización, y no está permitida la remisión de información de contacto entre compradores y vendedores registrados en el portal SUBASTAINTERACTIVA hasta este momento.

El comprador efectúa el pago del artículo y envía un correo electrónico al vendedor notificándoselo. Si es vendedor tiene problemas para concluir la transacción con el comprador, SUBASTAINTERACTIVA le ofrece la posibilidad de tramitar un aviso de comprador moroso y hacer los trámites para recuperar el precio final de la puja. Si por el contrario, el problema para concluir la transacción de venta la tiene el comprador con el vendedor, SUBASTAINTERACTIVA pone a su disposición un programa de protección contra el fraude.

Con el objetivo de que los usuarios tengan una mayor seguridad a la hora de realizar sus ventas, SUBASTAINTERACTIVA tienen un perfil de usuario compuesto por los comentarios de otros usuarios, es decir, como una especie de "reputación" oficial. Los comentarios son introducidos por el comprador o vendedor una vez finalizada la transacción, esos comentarios consiste en unos votos y observaciones sobre el otro miembro que ha participado en la operación. Los votos pueden ser positivos, negativos o neutrales. De este modo, si eres un posible comprador, puedes comprobar el perfil de tu vendedor y si eres vendedor, hacer lo mismo con los compradores. Los votos pueden resolver muchas dudas acerca de la forma de hacer negocios de un usuario. El sistema de votación es muy sencillo, se recibe:

- +1 punto por cada comentario positivo
- 0 puntos por cada comentario neutral
- -1 punto por cada comentario negativo

Finalmente, se pueden consultar las estadísticas de los usuarios, estando clasificados por estrellas y compradores y vendedores. Las estrellas se conceden al conseguir un determinado perfil de votos.

- Una "estrella amarilla" ( ) representa un perfil de entre 10 y 49 votos.
- Una "estrella azul" ( ) representa un perfil de entre 50 y 99 votos.

- Una "estrella turquesa" ( ) representa un perfil de entre 100 y 499 votos.
- Una "estrella púrpura" ( ) representa un perfil de entre 500 y 999 votos.
- Una "estrella roja" ( ) representa un perfil de entre 1.000 y 4.999 votos.
- Una "estrella verde" ( ) representa un perfil de entre 5.000 y 9.999 votos.
- Una "estrella fugaz amarilla" ( ) representa un perfil de entre 10.000 y 24.999 votos.
- Una "estrella fugaz turquesa" ( ) representa un perfil de entre 25.000 y 49.999 votos.
- Una "estrella fugaz púrpura" ( ) representa un perfil de entre 50.000 y 99.999 votos.
- Una "estrella fugaz roja" ( ) representa un perfil de 100.000 votos o más.

Con el fin de tener informados a los compradores más habituales de SUBASTAINTERACTIVA, ésta tiene una sección de últimas noticias en la que se pueden consultar las nuevas sesiones de subastas programadas por el portal (ver figuras 64 y 65). La visualización de esta sección se adapta de forma dinámica al comprador cuando se conecta al portal, mostrándole a cada comprador aquellas que más le interesan, teniendo en cuenta las anteriores sesiones de subasta en las que ha participado. En caso de que el usuario sea anónimo, el contenido será la página por defecto de la institución.

Figura 64. Oportunidades para el usuario conectado

Figura 65. Intereses para el usuario conectado

Otros servicios que ofrece SUBASTAINTERACTIVA es el envío de correos electrónicos informativos y una sección de programa en el que se pueden consultar el calendario de subastas, pudiendo incorporar aquellas de interés para el usuario en su agenda electrónica.

A.2.- Obtención de Requisitos

El objetivo de la etapa de obtención de los requisitos es la identificación y análisis de las necesidades del sistema de información que se va a abordar. Durante esta etapa se realiza una análisis exhaustivo del dominio del problema para conocer cuáles son los requisitos que el producto final ha de satisfacer. Estos requisitos serán especificados mediante casos de uso.

A.2.1. Casos de Uso para la Gestión de Administradores

- Gestión del Perfil Privado de un Administrador: El administrador del portal de subastas identificandose con su login y su password puede dar de alta, modificar, consultar o dar de baja a otros administradores.

Figura 66. Gestión de Administradores

A.2.2. Casos de Uso para la Gestión de Vendedores

- **Alta de Vendedor**: El usuario que navega por el portal puede darse de alta para poner a subasta artículos. El usuario deberá de proporcionar todos sus datos personales y quedará registrado en el sistema como uno de sus vendedores.
- **Gestión del Perfil Privado de un Vendedor:** El vendedor puede consultar y modificar su perfil, así como darse de baja como vendedor del sistema.

Figura 67. Gestión de Vendedores

A.2.3. Casos de Uso para la Gestión de Compradores

- **Alta de Comprador**: El usuario que navega por el portal puede darse de alta para participar en las subastas del portal y pujar por los artículos que esté interesado. El usuario deberá de proporcionar todos sus datos personales y quedará registrado en el sistema como uno de sus compradores.
- **Gestión del Perfil Privado de un Comprador:** El comprador puede consultar y modificar su perfil, así como darse de baja como comprador del sistema.

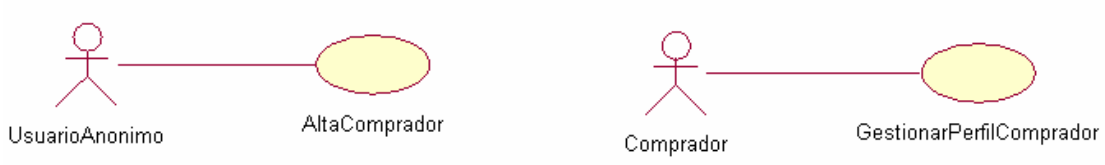

Figura 68. Gestión de Compradores

A.2.4. Casos de Uso para la Gestión de Galerías

- **Consultar**: Cualquier usuario puede visitar cualquier galería, sala y artículo sin necesidad de autentificarse para acceder a la información. Además de poder realizar búsquedas avanzadas de galerías, salas o artículos.
- **Gestión de Galerías:** El administrador del sistema puede crear, modificar y dar de baja tanto a las galerías como a las salas que estas incluyen. Para ello, el administrador deberá autentificarse previamente.
- **Gestión de Artículos:** Todo vendedor, proporcionando los datos que lo identifican, puede dar de alta artículos para subastarlos en el portal. Así mismo, aquel vendedor que haya puesto a subasta un artículo, mediante su clave de acceso, puede modificarlo y darle de baja.
- **Gestión de Categorías:** El administrador del sistema puede crear, modificar y dar de baja categorías y subcategorías. Para ello, el administrador deberá autentificarse previamente.

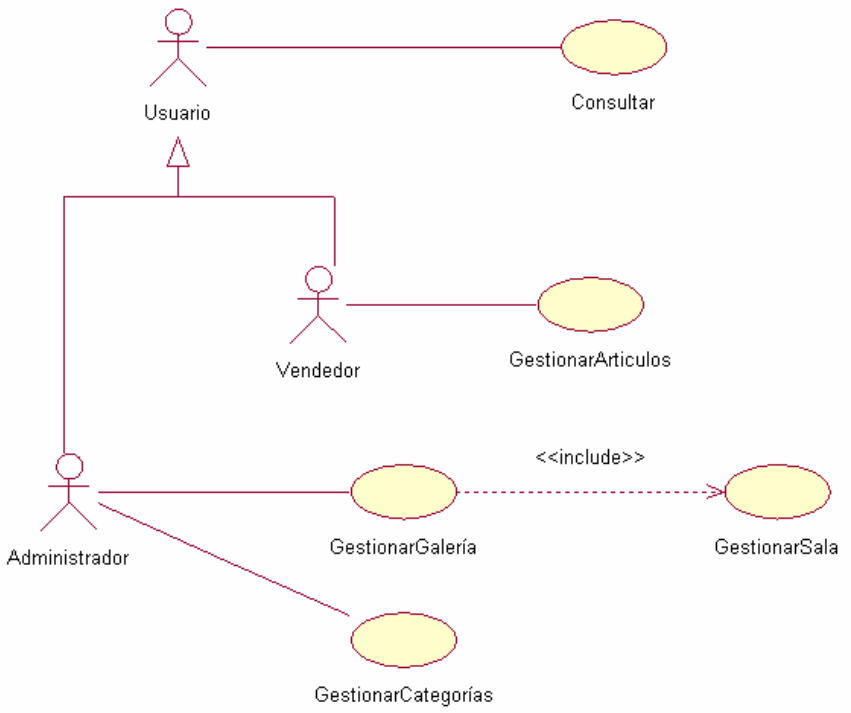

Figura 69. Gestión de Galerías

A.2.5. Casos de Uso para la Gestión de Subastas

- **Consulta de estadísticas**: Con el objetivo de que los usuarios tengan una mayor seguridad a la hora de realizar sus ventas, SUBASTAINTERACTIVA tienen unas estadísticas de sus compradores y vendedores en base a los comentarios de otros usuarios, es decir, como una especie de "reputación" oficial. Estas estadísticas se encuentran clasificadas por estrellas (niveles), compradores y vendedores, y pueden ser consultadas por cualquier usuario.

- Gestión de subastas: El administrador del sistema, tras su identificación, puede crear, modificar y dar de baja las subastas.

- **Registro de artículos en subasta**: El administrador se identifica para incluir dentro de una subasta programada aquellos artículos que se van a someter a subasta.

- **Inicio de subasta**: El subastador, asignado a una determinada subasta por administrador, ha de comprobar que se hayan registrado al menos 3 compradores, que se subastan al menos 2 artículos y establecer el precio de salida de cada uno de ellos, así como el orden de aparición de cada uno de ellos en la sesión de subasta. Seguidamente, el subastador deberá comprobar los registros de cada uno de los compradores con el fin con el de averiguar si el comprador esta clasificado como moroso, para que en ese caso se deniegue su participación.

- **Subasta de artículo**: El subastador establece cuál de los artículos de la subasta es el que se dispone a subastar en ese momento y lo expone a los miembros participantes junto a su precio de salida.

- **Adjudicar artículo**: Si el subastador, durante un periodo de tiempo de 3 minutos, comprueba que no se ha pujado por el artículo, dará una última oportunidad en la puja y si no se produce, el artículo será adjudicado al comprador que realizó la última.
- **Retirar artículo de subasta**: Una vez un artículo ha sido adjudicado a algunos de los compradores de la subasta el artículo es retirado de la subasta por el subastador.
- **Registro de comprador en subasta**: El comprador ha de identificarse y registrarse como comprador de la subasta en la que desee participar. Dicho registro ha de realizarse con anticipación a su hora de comienzo.
- **Pujar**: Los compradores pujan aumentando el importe del artículo respecto al del precio de salida, teniendo en cuenta que el incremento no tiene por qué ser una cantidad constante.
- **Retirar comprador de subasta**: Todo comprador puede retirarse de una sesión de puja siempre que desee, durante el transcurso de la subasta o en su finalización.

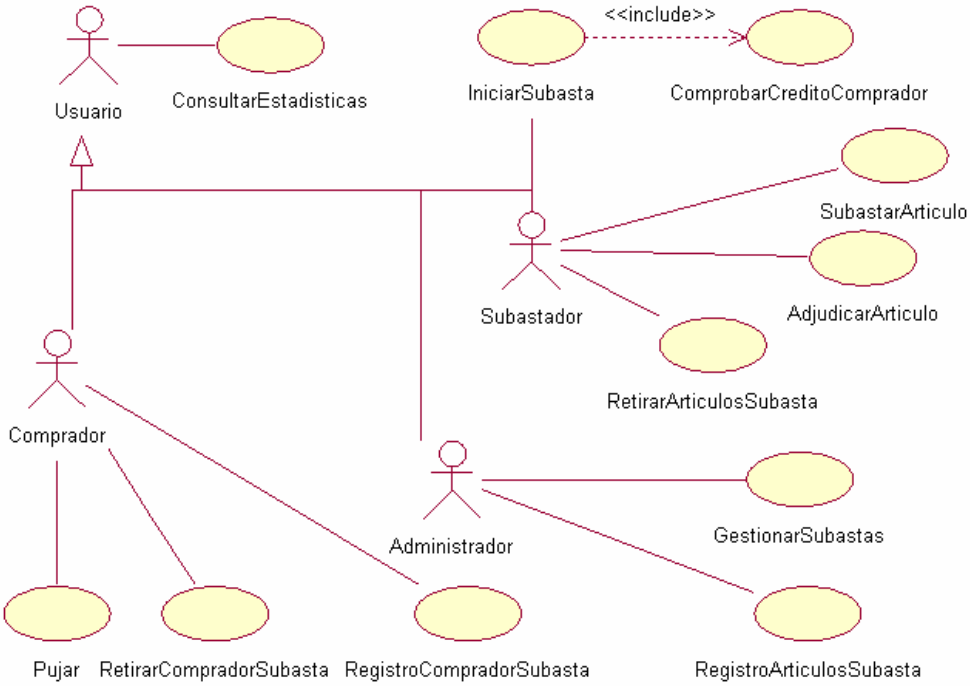

Figura 70. Gestión de Subastas

A.2.6. Casos de Uso para la Gestión de Ventas

- **Pagar**: El comprador efectúa el pago del artículo y envía un correo electrónico al vendedor notificándoselo. Si es vendedor tiene problemas para concluir la transacción con el comprador, SUBASTAINTERACTIVA le ofrece la posibilidad de tramitar un aviso de comprador moroso y hacer los trámites para recuperar el precio final de la puja.

- **Envío Artículo:** El vendedor envía un correo electrónico al comprador con la finalidad de ejecutar la transacción de venta, adjuntando el albaran rellenado para el envío. Si el comprador tiene algún problema para concluir la transacción de venta con el vendedor, SUBASTAINTERACTIVA pone a su disposición un programa de protección contra el fraude.

- **Evaluación Comprador**: El vendedor, tras su identificación, introduce los comentarios sobre el comprador con el que ha realizado la transacción, esos comentarios consiste en unos votos (estrellas) y observaciones a cerca de éste. Los votos pueden ser positivos, negativos o neutrales.

- **Evaluación Vendedor**: El comprador, tras su identificación, introduce los comentarios sobre el vendedor con el que ha realizado la transacción, esos comentarios consiste en unos votos (estrellas) y observaciones a cerca de éste. Los votos pueden ser positivos, negativos o neutrales.

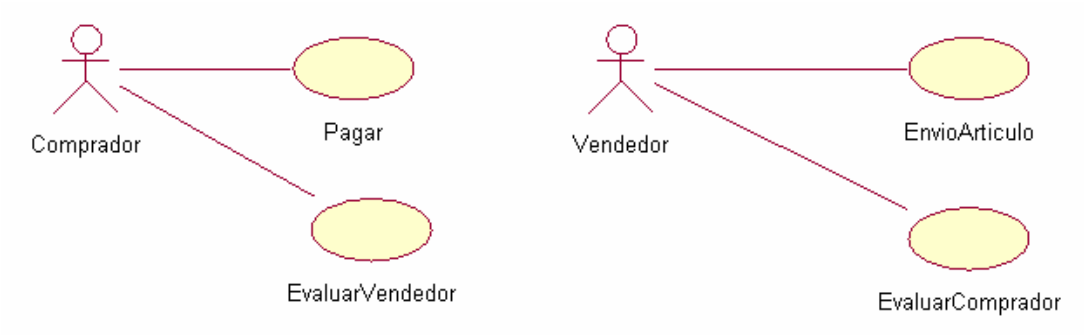

Figura 71. Gestión de Ventas

A.2.7. Casos de Uso para la Gestión de Marketing

- **Gestión del Calendario Subastas**: Cualquier usuario del sistema (administrador, vendedor, comprador) puede consultar el calendario de subastas, tras previa identificación, y puede incorporar aquellas subastas de interés en su agenda electrónica personal.

- **Envío de noticias**: El administrador del sistema se encarga del envío y mantenimiento de correos electrónicos informativos

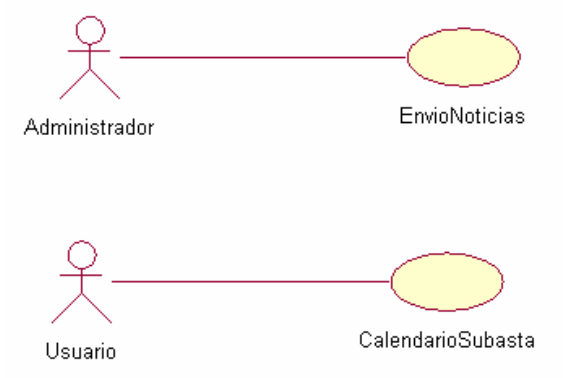

Figura 72. Gestión de Marketing

A.3. Especificación PRISMA

```
Interface Gestion
      Alta;
      Baja;
      Modificacion (descripcion: string);
End_Interface Gestion;


Interface Consulta
      Consulta;
End_Interface Consulta;



Functional Aspect Mantenimiento specifies Gestion, Consulta;

  Attributes

      Codigo: string;

      Nombre: string;

  Services

      Crear as Alta;

      Destruir as Baja;

      Modificar(Descripcion: string) as Modificacion
      (Descripcion:string);

      Consultar as Consulta;

  Valuations

      [modificar(Descripcion)] Nombre = Descripcion;

  Protocol

            CREACION ≡ crear.MANTENIMIENTO;

            MANTENIMIENTO ≡ modificar + consultar + destruir;

End_Functional Aspect Mantenimiento;


Component_type  Galería
      Inport InP1: Gestion;
             InP2: Consulta;
      End Inport;
      Outport OutP1: Consulta;
      End Outport;
      Functional Aspect
            FuncionalidadGaleria_i: Mantenimiento;
```

```
        End_Functional Aspect;
End Component_Type Galería;


Component_type  Categoría
        Inport InP1: Gestion;
                InP2: Consulta;
        End Inport;
        Outport OutP1: Consulta;
        End Outport;
        Functional Aspect
                FuncionalidadCategoría_i: Mantenimiento;
        End_Functional Aspect;
End Component_Type Categoría;


Component_type  Sala
        Inport InP1: Gestion;
                InP2: Consulta;
        End Inport;
        Outport OutP1_i: Consulta;
        End Outport;
        Functional Aspect
                FuncionalidadSala: Mantenimiento;
        End_Functional Aspect;
End Component_Type Sala;


Interface GestionArticulo
        Crear;
        Destruir;
        ModificarDesc(descripcion: string);
        ModificarObs(observaciones: string);
End_Interface GestionArticulo;


Functional Aspect Productos specifies GestionArticulo, Consulta;
          Attributes
                Codigo: string;
                Nombre: string;
                Observaciones: string;
          Services
                Crear;
                Destruir;
```

```
        ModificarDesc(Descripcion: string);

        ModificarObs(Observaciones: string);

        Consultar as Consulta;

    Valuations

        [ModificarDesc(Descripcion)] Nombre = Descripcion;

        [ModificarObs(Observaciones)] Observaciones =
        Observaciones;

    Protocol

        CREACION ≡ crear.MANTENIMIENTO;

        MANTENIMIENTO ≡ modificar + consultar + destruir;

End_Functional Aspect Productos;


Component_type  Articulo
        Inport InP1 : GestionArticulo;
                InP2 : Consulta;
        End Inport;
        Outport OutP1: Consulta;
        End Outport;
        Functional Aspect
                FuncionalidadArticulo: Productos;
        End_Functional Aspect;
End Component_Type Articulo;


Interface GestionPuja
        Crear;
        Destruir;
        ModificarFecha(FechaPuja: date);

        ModificarHora(HoraPuja: hora);

        MasPujadores;

        MasArticulos;
End_Interface GestionPuja;


Functional Aspect Pujas specifies GestionPuja, Consulta;

    Attributes

        Codigo: string;

        Fecha: date;

        Hora: hora;

        NumPujadores: natural;

        NumArticulos: natural;
```

```
  Services
      Crear;
      Destruir;
      ModificarFecha(FechaPuja: date);
      ModificarHora(HoraPuja: hora);
      Consultar as Consulta;
      ComenzarPuja;
      MasPujadores;
      MasArticulos;
  Valuations
      [ModificarFecha(FechaPuja)] Fecha = FechaPuja;
      [ModificarHora(HoraPuja)] Hora = HoraPuja;
      [MasPujadores()] NumCompradores = NumCompradores + 1;
      [MasArticulos()] NumArticulos = NumArticulos + 1;
  Preconditions
      [NumPujadores >= 3] ComenzarPuja;
  Protocol
      CREACION ≡ crear.MANTENIMIENTO;
      MANTENIMIENTO ≡ ModificarFecha + Modificar Hora +
      MasPujadores  + MasArticulos + Consultar + Destruir;
End_Functional Aspect Pujas;


Component_type  Subasta
      Inport InP1 : GestionPuja;
             InP2 : Consulta;
      End Inport;

      Outport OutP1: Consulta;
      End Outport;
      Functional Aspect
            FuncionalidadSubasta: Pujas;
      End_Functional Aspect;
End Component_Type Subasta;


Interface GestionPersonal
      Crear;
      Destruir;
      ModificarLogin(NuevoLogin: string);
```

```
        ModificarPassword (NuevoPassword: string);
        ModificarNombre(NuevoNombre: string);
        ModificarApel1(NuevoApel1: string);
        ModificarApel2(NuevoApel2: string);
        ModificarDireccion (NuevaDir: string);
        ModificarPoblacion(NuevaPobla: string);
        ModificarPais(NuevoPais: string);
        ModificarTelf(NuevoTelf: string);
        ModificarMail(NuevoMail: string);
End_Interface GestionPersonal;


Interface ConsultaUsuarios
        ConsultarEstadisticas;
End_Interface ConsultaUsuarios;


Interface GestionAdminitrador
        ENVIONOTICIAS;
End_Interface GestionAdministrador;


Functional Aspect Personal specifies GestionPersonal, Consulta,
ConsultaUsuarios, GestionAdministrador;
  Attributes
        DNI: string;
        Login: string;
        Password: string;
        Nombre: string;
        Apellido1: string;
        Apellido2: string;
        Direccion: string;
        Poblacion: string;
        Pais: string;
        Telefono: string;
        Email: string;
  Services
        Crear;
        Destruir;
```

```
ModificarLogin(NuevoLogin: string);
ModificarPassword (NuevoPassword: string);
ModificarNombre(NuevoNombre: string);
ModificarApel1(NuevoApel1: string);
ModificarApel2(NuevoApel2: string);
ModificarDireccion (NuevaDir: string);
ModificarPoblacion(NuevaPobla: string);
ModificarPais(NuevoPais: string);
ModificarTelf(NuevoTelf: string);
ModificarMail(NuevoMail: string);
Consultar as Consulta;
ConsultarEstadisticas;
```

Valuations

```
[ModificarLogin(NuevoLogin)] Login = NuevoLogin;
[ModificarPassword (NuevoPassword)] Password =
NuevoPassword;
[ModificarNombre(NuevoNombre)] Nombre = NuevoNombre;
[ModificarApel1(NuevoApel1)] Apellido1 = NuevoApel1;
[ModificarApel2(NuevoApel2)] Apellido2 = NuevoApel2;
[ModificarDireccion (NuevaDir)] Direccion = NuevaDir;
[ModificarPoblacion(NuevaPobla)] Población = NuevaPobla;
[ModificarPais(NuevoPais)] Pais = NuevoPais;
[ModificarTelf(NuevoTelf)] Telefono = NuevoTelf;
[ModificarMail(NuevoMail)] EMail = NuevoMail;
```

Transactions

```
ENVIONOTICIAS;
```

Protocol

```
CREACION ≡ crear.MANTENIMIENTO;
MANTENIMIENTO ≡ ModificarLogin + ModificarPassword +
                ModificarNombre + ModificarApel1 +
                ModificarApel2 + ModificarDireccion +
                ModificarPoblacion + ModificarPais +
                ModificarTelf + ModificarMail + Consultar
                + ENVIONOTICIAS + Out Destruir+ Out alta +
                Out baja + Out modificación + Out consulta
                + Out crear + Out destruir +  Out
```

```
                        modificarDesc + Out ModificarObs + Out
                        ModificarFecha + Out Hora + Out
                        MasPujadores + Out MasArticulos;
End_Functional Aspect Personal;

Component_type  Administrador
       Inport InP1 : GestionPersonal;
              InP2 : Consulta;
       End Inport;
       Outport OutP1: Consulta;
               OutP2:  ConsultaUsuarios;
               OutP3: Gestion;
               OutP4: GestionArticulo;
               OutP5: GestionPuja;
               OutP6: GestionAdministrador;
               OutP7: GestionPersonal;
       End Outport;
       Functional Aspect
              FuncionalidadAdministrador:Personal;
       End_Functional Aspect;
End Component_Type Administrador;

Component_type  Vendedor
       Inport InP1 : GestionPersonal;
              InP2 : Consulta;
       End Inport;
       Outport OutP1: Consulta;
               OutP2:  ConsultaUsuarios;
               OutP3: GestionArticulo;
       End Outport;
       Functional Aspect
              FuncionalidadAdministrador:Personal;
       End_Functional Aspect;
End Component_Type Vendedor;


Interface SesionPuja
       comprar;
       pujar;
       pagar;
End_Interface SesionPuja;


Functional Aspect Pujador specifies GestionPersonal, Consulta,
ConsultaUsuarios, SesionPuja;
```

```
Constant Attributes
    DNI: string;
Variable Attributes
        Login: string;
    Password: string;
    Nombre: string;
    Apellido1: string;
    Apellido2: string;
    Direccion: string;
    Poblacion: string;
    Pais: string;
    Telefono: string;
    Email: string;
    NumArticulos: natural;
Private_Services
    Crear;
    Destruir;
    ModificarLogin(NuevoLogin: string);
    ModificarPassword (NuevoPassword: string);
    ModificarNombre(NuevoNombre: string);
    ModificarApel1(NuevoApel1: string);
    ModificarApel2(NuevoApel2: string);
    ModificarDireccion (NuevaDir: string);
    ModificarPoblacion(NuevaPobla: string);
    ModificarPais(NuevoPais: string);
    ModificarTelf(NuevoTelf: string);
    ModificarMail(NuevoMail: string);
    Consultar as Consulta;
    ConsultarEstadisticas;
    comprar;
    pujar;
    pagar;
Valuations
    [ModificarLogin(NuevoLogin)] Login = NuevoLogin;
    [ModificarPassword (NuevoPassword)] Password =
    NuevoPassword;
```

```
[ModificarNombre(NuevoNombre)] Nombre = NuevoNombre;
[ModificarApel1(NuevoApel1)] Apellido1 = NuevoApel1;
[ModificarApel2(NuevoApel2)] Apellido2 = NuevoApel2;
[ModificarDireccion (NuevaDir)] Direccion = NuevaDir;
[ModificarPoblacion(NuevaPobla)] Población = NuevaPobla;
[ModificarPais(NuevoPais)] Pais = NuevoPais;
[ModificarTelf(NuevoTelf)] Telefono = NuevoTelf;
[ModificarMail(NuevoMail)] EMail = NuevoMail;
[comprar()] NumArticulos = NumArticulos+1;
```

```
Protocol
        CREACION ≡ crear.MANTENIMIENTO;
        COMPRADOR ≡ ModificarLogin + ModificarPassword +
                    ModificarNombre + ModificarApel1 +
                    ModificarApel2 + ModificarDireccion +
                    ModificarPoblacion + ModificarPais +
                    ModificarTelf + ModificarMail + Consultar +
                    Comprador: ::comprar + Comprador:  ::pujar +
                    Comprador: ::pagar + Destruir;
End_Functional Aspect Personal;



Component_type  Comprador
        Inport InP1 : GestionPersonal;
               InP2 : Consulta;
        End Inport;
        Outport OutP1: Consulta;
                OutP2:  ConsultaUsuarios;
                OutP3: GestionArticulo;
                OutP4: SesionPuja;
        End Outport;
        Functional Aspect
               FuncionalidadComprador:Pujador;
        End_Functional Aspect;
End Component_Type Comprador;


Interface GestionPuja
        SubastarArticulo;
        AdjudicarArticulo;
        IniciarSubasta;
        RetirarArticulo;
End_Interface GestionPuja;
```

```
Functional Aspect GestionSubasta specifies GestionPersonal,
Consulta, ConsultaUsuarios, GestionPuja;
  Constant Attributes
      DNI: string;
  Variable Attributes
      Login: string;
      Password: string;
      Nombre: string;
      Apellido1: string;
      Apellido2: string;
      Direccion: string;
      Poblacion: string;
      Pais: string;
      Telefono: string;
      Email: string;
 Private_Services
      Crear;
      Destruir;
      ModificarLogin(NuevoLogin: string);
      ModificarPassword (NuevoPassword: string);
      ModificarNombre(NuevoNombre: string);
      ModificarApel1(NuevoApel1: string);
      ModificarApel2(NuevoApel2: string);
      ModificarDireccion (NuevaDir: string);
      ModificarPoblacion(NuevaPobla: string);
      ModificarPais(NuevoPais: string);
      ModificarTelf(NuevoTelf: string);
      ModificarMail(NuevoMail: string);
      Consultar as Consulta;
        Valuations
      [ModificarLogin(NuevoLogin)] Login = NuevoLogin;
      [ModificarPassword (NuevoPassword)] Password =
      NuevoPassword;
      [ModificarNombre(NuevoNombre)] Nombre = NuevoNombre;
      [ModificarApel1(NuevoApel1)] Apellido1 = NuevoApel1;
      [ModificarApel2(NuevoApel2)] Apellido2 = NuevoApel2;
```

```
        [ModificarDireccion (NuevaDir)] Direccion = NuevaDir;
        [ModificarPoblacion(NuevaPobla)] Población = NuevaPobla;
        [ModificarPais(NuevoPais)] Pais = NuevoPais;
        [ModificarTelf(NuevoTelf)] Telefono = NuevoTelf;
        [ModificarMail(NuevoMail)] EMail = NuevoMail;
        [comprar()] NumArticulos = NumArticulos+1;
    Protocol
        CREACION ≡ crear.MANTENIMIENTO;
        COMPRADOR ≡ ModificarLogin + ModificarPassword +
                    ModificarNombre + ModificarApel1 +
                    ModificarApel2 + ModificarDireccion +
                    ModificarPoblacion + ModificarPais +
                    ModificarTelf + ModificarMail + Consultar +
                    Subastador: :: SubastarArticulo + Comprador:
                    :: AdjudicarArticulo+ Comprador: ::
                    IniciarSubasta + +Destruir;
End_Functional Aspect GestionSubasta;


    Component_type  Subastador
        Inport InP1 : GestionPersonal;
               InP2 : Consulta;
        End Inport;
        Outport OutP1: Consulta;
                OutP2:  ConsultaUsuarios;
                OutP3: GestionArticulo;
                OutP4: GestionPuja;
        End Outport;
        Functional Aspect
               FuncionalidadSubastador:GestionSubasta;
        End_Functional Aspect;
    End Component_Type Subastador;
```

```
Connector_type  ArticuloVendedor
      Inrol InR1 : GestionArticulo;
            InR2 : Consulta;
      End Inrol;
      Outrol OutR1: Consulta;
            OutR2: GestionArticulo;
      End Outrol;
      Coordination Aspect
            CoordinaArtVen:CoordinaArticuloVendedor;
      End_Coordination Aspect;
End Connector_Type ArticuloVendedor;


Attachments
      Associated_connector ArtículoVendedor
            Artículo.OutP1 → ArticuloVendedor.InR2
            Vendedor.OutP1 → ArticuloVendedor.InR2
            Vendedor.OutP3 → ArticuloVendedor.InR1
            Artículo.InP2← ArticuloVendedor.OutR1
            Artículo.InP1← ArticuloVendedor.OutR2
            Vendedor.InP2 ← ArticuloVendedor.OutR2
      End_associated_connector ArtículoVendedor;
End_Attachments;


Connector_type  GestionSubasta
      Inrol InR1 : GestionArticulo;
            InR2 : Consulta;
            InR3: GestionPuja;

      End Inrol;
      Outrol OutP1: Consulta;
             OutP2: GestionPuja;
      End Outrol;
      Coordination Aspect
            CoordinaSub:CoordinaSubasta;
      End_Coordination Aspect;
End Connector_Type GestionSubasta;



Attachments
```

```
        Associated_connector GestiónSubasta
                Artículo.OutP1 → GestionSubasta.InR2
                Vendedor.OutP1 → GestionSubasta.InR2
                Comprador.OutP1 → GestionSubasta.InR2
                Subasta.OutP1 → GestionSubasta.InR2
                Subastador.OutP1 → GestionSubasta.InR2
                Vendedor.OutP3 → GestionSubasta.InR1
                Comprador.OutP3 → GestionSubasta.InR1
                Subastador.OutP3 → GestionSubasta.InR1
                Subastador.OutP4 → GestionSubasta.InR3
                Subasta.InP2 ← GestionSubasta.OutR1
                Artículo.InP2 ← GestionSubasta.OutR1
                Vendedor.InP2 ← GestionSubasta.OutR1
                Comprador.InP2 ← GestionSubasta.OutR1
                Pujador.InP2 ← GestionSubasta.OutR1
                Subasta.InP1 ← GestionSubasta.OutR2
        End_associated_connector GestionSubasta;
End_Attachments;
```