Departamento de Lenguajes y Computación
Universidad de Almería

Un Modelo de Mediación para el Desarrollo de Software basado en Componentes COTS

TESIS DOCTORAL

Presentado por

Luis F. Iribarne Martínez

para optar al grado de
Doctor en Informática

Dirigida por

Dr. D. Antonio Vallecillo Moreno
Profesor Titular de Universidad
Área de Lenguajes y Sistemas Informáticos
Universidad de Málaga

Dr. D. José María Troya Linero
Catedrático de Universidad
Área de Lenguajes y Sistemas Informáticos
Universidad de Málaga

Almería, 14 de Julio de 2003

Tesis Doctoral

Un Modelo de Mediación para el Desarrollo de Software Basado en Componentes COTS

Luis F. Iribarne Martínez

Este documento ha sido generado
con la versión 3.14 de LaTeX

COTSTRADER.COM es un sitio web registrado
en el gestor de nombres NOMINALIA.COM
y ha sido creado con fines totalmente
de investigación y sin ánimo de lucro.

Todas las figuras y tablas contenidas en
el presente documento son originales.

Un Modelo de Mediación para el Desarrollo de Software basado en Componentes COTS

Luis F. Iribarne Martínez
Departamento de Lenguajes y Computación
Universidad de Almería
Almería, 14 de Julio de 2003
http://www.ual.es/~liribarn

A mis hijos, Marta y Sergio

Agradecimientos

Estos tres últimos años han sido para mí unos de los más importantes, intensos y fascinantes de mi trayectoria profesional. En este tiempo he tenido la enorme suerte y satisfacción de conocer y de trabajar con personas que me han ayudado —de una forma u otra— en la consecución de un esfuerzo de investigación, que se recoge en el presente documento de tesis doctoral, y a las que les estoy profundamente agradecido. Aunque el hecho de exponer una lista de personas siempre supone un riesgo de olvidar a alguna de ellas, sí quisiera hacer una especial mención de agradecimiento para las siguientes.

En primer lugar quisiera mencionar a José María Troya, que ha sido para mi un auténtico privilegio y honor tenerlo como co-director, y al que me gustaría agradecerle la gran oportunidad que me ha dado y la confianza que ha depositado en mí en el decurso de esta tesis doctoral. Pero también quisiera darle las gracias por haberme presentado a una de las personas más auténticas que jamás he conocido, a la que realmente admiro como persona y como profesional, Antonio Vallecillo, otro de mis directores. A él quisiera francamente agradecerle todo lo que me ha enseñado en estos últimos años, y sobre todo, el tiempo y el esfuerzo que me ha dedicado, y sus continuos consejos en el transcurso de la tesis doctoral: muchas gracias de todo corazón Antonio.

También quisiera darle las gracias a Samuel Túnez por haberme facilitado el contacto con José María. A Manuel Cantón por tenerlo como tutor y por su interés en la consecución de esta tesis. Y especialmente a José Ramón Díaz, por su constante aliento por la investigación desde mis comienzos en la universidad.

En estos momentos me gustaría mencionar a algunos colegas de la Universidad de Málaga que me han acogido magníficamente en mis frecuentes viajes a la Facultad de Informática, gracias Mami, Lidia, Eva y Paco. Mi agradecimiento especial a Carlos Canal por sus útiles comentarios acerca de las arquitecturas de software.

A Antonio Becerra, compañero y amigo, le quisiera agradecer su inestimable apoyo en todos estos años. Y a Daniel Landa le agradezco profundamente sus eficientes gestiones y su infinita paciencia.

Pero especialmente quisiera darle las gracias a una persona que siempre ha confiado en mi, que realmente sabía lo importante que era para mi este esfuerzo de investigación, y que me ha apoyado en los momentos duros, previos a esta tesis y durante el transcurso de la misma, Rosa Ayala, mi mujer, la verdadera artífice de que este trabajo de investigación se haya visto culminado: te quiero Rosa.

También quisiera mencionar a mis dos hijos, Marta y Sergio, que tanta fuerza me han dado en los momentos difíciles. A mis padres y a mis suegros, que los adoro, y al resto de mi familia, les agradezco todo su apoyo y confianza, y su valiosa ayuda prestada en estos últimos meses.

Gracias también a todos aquellos compañeros del departamento, Escuela Politécnica y otros centros de la Universidad, y colegas de otras universidades que me han ayudado, aconsejado y apoyado en algún momento del transcurrir de esta tesis doctoral.

Y finalmente agradecer a los proyectos CICYT (TIC99-1083-C02-01) "Diseño de software basado en componentes. Metodologías y herramientas para el desarrollo de aplicaciones distribuidas" (1999-2002) y CYTED (Subproyecto VII.18) "WEST: Web-Oriented Software Technology" (2000-2003) que han subvencionado parcialmente este trabajo.

A todas estas personas que aquí he mencionado, muchas gracias y que Dios les bendiga.

Luis Iribarne Martínez
Departamento de Lenguajes y Computación
Universidad de Almería
Almería, 2003

Contenido

Índice de Figuras

Índice de Tablas

Prólogo

Uno de los objetivos tradicionales de la ingeniería del software ha sido la necesidad de desarrollar sistemas mediante el ensamblaje de módulos independientes. La ingeniería del software basada en componentes distribuidos (ISBCD) es una de las disciplinas de la ingeniería del software que más impulso está teniendo en los últimos tiempos, y que se caracteriza por describir, desarrollar y utilizar técnicas basadas en componentes software para la construcción de sistemas abiertos y distribuidos, como por ejemplo los sistemas de información distribuidos bajo Web.

Ingeniería del software basada en componentes COTS (ISBC-COTS)

Los componentes software, en cierta medida, surgen de la necesidad de hacer un uso correcto del software reutilizable dentro de las aplicaciones. Sin embargo, la mayoría de las metodologías de reutilización actuales no contemplan el uso de "componentes comerciales" para el desarrollo de las aplicaciones de software. A este tipo de software se le conoce con el nombre de componentes *commercial-off-the-shelf* o COTS. Con el término COTS queremos hacer referencia a aquel software comercial desarrollado previamente por terceras partes, fuera de las estrategias de desarrollo y aplicadas durante todo el ciclo de vida del producto a desarrollar en base a productos comerciales, entre los que se incluyen: los sistemas heredados (*legacy systems*), el software de dominio público, y otros elementos desarrollados fuera de la organización, también llamados software NDI (*Non-Developmental Item*) [Carney y Long, 2000]. Esta clase de componente software generalmente es adquirido en formato binario, sin posibilidad de tener acceso al código fuente y sin información adicional que ayude a una selección correcta de los mismos.

Por supuesto, esto hace que la ingeniería del software se enfrente a nuevos retos y problemas, pues esta clase de componentes necesita de un desarrollo ascendente (*bottom-up*) de los sistemas, frente al desarrollo tradicional descendente (*top-down*). El proceso de desarrollo de aplicaciones de software basadas en componentes comienza con la definición de la arquitectura de software, donde se recogen las especificaciones a nivel "abstracto" de los componentes que "dan forma" a la aplicación que se desea construir. En el desarrollo tradicional descendente, los requisitos de la aplicación se van desglosando (refinando) sucesivamente en otros más simples hasta llegar a diseñar e implementar los componentes finales. Sin embargo, en el desarrollo ascendente, gran parte de estos requisitos iniciales pueden ser cubiertos por otros componentes que ya han sido desarrollados y que se encuentran almacenados en repositorios de componentes. Estos repositorios contienen por tanto, unas especificaciones "concretas" de componentes comerciales que son requeridos por unos procesos automatizados de "búsqueda y selección" que intentan localizar aquellas especificaciones concretas de componentes que cumplen con las restricciones impuestas a nivel abstracto en la arquitectura de software.

Además, no solo es suficiente con disponer de procesos automáticos que busquen y seleccionen los componentes especificados en la arquitectura de software, sino que también son necesarios procesos (automatizados) que contemplen las dependencias entre los componentes de la arquitectura, y que generen posibles combinaciones entre los componentes

extraídos por estos procesos de búsqueda y selección. Cada una de estas combinaciones luego podrían ofrecer una solución parcial o total a la "arquitectura de software" de la aplicación que se pretende construir. Incluso, en algunos casos, podría no ser necesario adquirir software COTS para ciertas partes del sistema, sino que los componentes podrían ser implementados directamente por el desarrollador del sistema. En otros casos, puede que los componentes COTS deban ser adaptados haciendo uso de envolventes (*wrappers*) y conectados según dicta la arquitectura de software, haciendo uso de código especial (*glue*). En cualquier caso, es necesario aplicar procesos y técnicas de evaluación y adaptación de componentes para obtener la solución final. También se puede seguir un desarrollo basado en espiral, donde los requisitos iniciales de los componentes son refinados en sucesivas ocasiones hasta encontrar la solución que mejor se ajusta a las necesidades de la aplicación.

El problema de la "conexión diseño-implementación"

Este planteamiento de desarrollo ascendente basado en componentes COTS, ayuda a reducir los costes, tiempos y esfuerzos de desarrollo, mientras mejora la flexibilidad, fiabilidad y la reutilización de la aplicación final. Sin embargo, este planteamiento hoy día sigue siendo un problema en el área de la ingeniería del software basada en componentes, conocido como "el problema de la conexión diseño-implementación", también conocido como *Gap Analysis* [Cheesman y Daniels, 2001]. Como muestra la figura 1, actualmente existe un hueco importante abierto entre las especificaciones de unos componentes abstractos definidos en fase de diseño —cuando se describe la arquitectura de software— y las especificaciones de unos componentes COTS con implementaciones concretas residentes en repositorios. Como hemos adelantado antes, esta "desconexión" se debe en parte a la falta de unos procesos de búsqueda, selección y adaptación de componentes comerciales; aunque en cierta medida también se debe a otros factores no menos importantes, como la falta de un modelo para documentar componentes COTS, o la falta de un "mediador" o servicio de mediación (*trading*) que funcione para componentes COTS, o incluso a la falta de una notación adecuada para escribir arquitecturas de software utilizando esta clase de componentes.

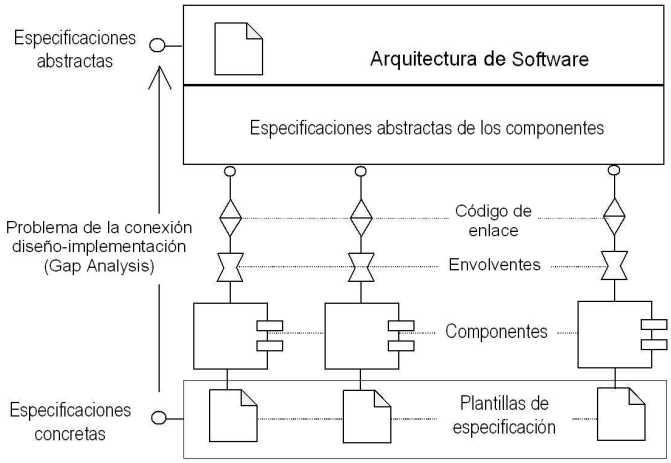

Figura 1: Problema de la conexión diseño-implementación

Ante este marco, nos podríamos plantear las siguientes cuestiones para averiguar con mayor detalle los motivos de este problema. Por ejemplo ¿cómo se lleva a cabo actualmente la localización, selección e integración de componentes COTS?, ¿existen repositorios de componentes COTS y servicios de mediación adecuados?, ¿qué tipo de información ofrece un vendedor de componentes COTS a sus clientes?, ¿qué tipo de información es la que

debe utilizar un cliente de componentes COTS para que el vendedor sepa lo que quiere?, ¿cómo aunar la información de los clientes y de los vendedores para poder seleccionar componentes, asegurando que la aplicación construida con ellos satisface los requisitos de las especificaciones?

Para dar respuesta a estas cuestiones, primero es necesario determinar el flujo de información que existe, por un lado, desde el vendedor hacia el cliente, que intenta decir "esto es lo que tengo, dime lo que quieres", y por otro lado desde el cliente hacia el vendedor, que intenta decir "esto es lo que quiero, dime que es lo que tienes". El verdadero problema es que tal flujo de información no está bien estructurado, ni soportado por las actuales metodologías y herramientas software. El vendedor de componentes COTS no ofrece a sus clientes información suficiente acerca de sus productos para que estos puedan determinar si los componentes que ellos ofrecen son válidos para sus arquitecturas diseñadas, y para determinar cómo integrar esos componentes en sus arquitecturas a partir de la información disponible. Por otro lado, el cliente no sabe cómo hacer llegar a su/s vendedor/es la información que necesita. En la mayoría de los casos el vendedor desconoce detalles técnicos que su cliente omite por considerarlos obvios, y en mayor grado sucede a la inversa.

Uno de los inconvenientes COTS es su falta de información para ser descubiertos y evaluados

Parte de estos problemas se podrían resolver si existiera un mecanismo o una técnica unificada para la documentación de componentes COTS, común tanto para los vendedores como para los clientes que buscan un determinado producto para: (a) *seleccionar* los componentes COTS que se ajustan a las especificaciones de uno buscado; (b) *probar* los seleccionados para escoger el más adecuado; (c) *validar* sus resultados de prueba y su conveniencia de ser incorporado en el subsistema correspondiente (incompatibilidad con el tipo de arquitectura); (d) *comprar* el producto según los patrones de especificación, como contrato cliente/vendedor; y por último (e) *integrar* los componentes COTS.

Necesidad de un modelo para documentar componentes COTS

Como resultado final en la elaboración de su producto, los vendedores o encargados de desarrollar componentes COTS (desarrollo "centrado") podrían especificar y documentar sus componentes a partir de unas plantillas de documento que los clientes también entenderían, ya que estos usarían unas plantillas similares para solicitar productos COTS cuando desarrollen sus sistemas software (desarrollo "basado"). Incluso estos documentos de componente podrían servir como fuente de información para posibles repositorios de componentes COTS que serían usados por servicios de mediación —encargados de su localización— para automatizar los procesos descritos anteriormente.

Por tanto, como se ha justificado hasta el momento, el marco en el cual nos vamos a centrar contempla la necesidad de establecer un método basado en un modelo de mediación para el desarrollo de aplicaciones con componentes COTS. Desde una perspectiva global como la que hemos visto en la figura 1, este método puede quedar identificado por tres actividades: (a) la definición de la arquitectura de software con componentes COTS, (b) los procesos de búsqueda, selección y adaptación de componentes COTS, y (c) la estructura de los repositorios de componentes COTS.

En primer lugar, las arquitecturas de software definen las características de los componentes que dan forma a la aplicación a construir, como sus interfaces, sus protocolos de interacción y/o sus nexos de unión. El gran inconveniente, como adelantamos, es la ausencia de una notación para escribir arquitecturas de software con COTS.

Para el caso de los procesos de búsqueda y selección de componentes se suelen utilizar los servicios de mediación (conocidos como *traders*) y que son los encargados de buscar, comprobar y seleccionar aquellos componentes del repositorio que cumplen con los requisitos de los componentes impuestos en la arquitectura de software. Actualmente se utiliza el modelo de mediación de ODP [ISO/IEC-ITU/T, 1997], pero sólo es válido para objetos CORBA y además presenta una serie de limitaciones para el caso de los componentes COTS

(p.e., no soporta funciones de búsqueda para componentes con múltiples interfaces, ni funciones de búsqueda con restricciones impuestas sobre el comportamiento y propiedades extra-funcionales de un componente, entre otras limitaciones).

En último lugar, para el caso de los repositorios de componentes, en la actualidad existen tres clases: (a) los repositorios IDL, que contienen la definición abstracta de las interfaces de un componente; (b) los repositorios con los tipos de servicios —como el repositorio de tipos de ODP— que contienen información acerca de los tipos de componentes registrados, y que suelen estar ligados a los repositorios IDL; y (c) los repositorios de implementación que, aunque no suelen estar muy extendidos, contienen el código ejecutable de los componentes (en modo binario). El problema, una vez más, es que actualmente no existen repositorios que soporten especificaciones de componentes COTS, debido a la falta de una técnica común para documentarlos.

Objetivos y contribución

Para llegar a conseguir un método de desarrollo de aplicaciones basado en componentes COTS como el que hemos planteado, hace falta lograr primero varios objetivos:

1. En primer lugar, y como requisito prioritario, se ha de disponer de un modelo para la documentación de componentes COTS con el propósito de lograr una conexión entre los componentes especificados en la arquitectura y los componentes implementados residentes en el repositorio. Además, como hemos adelantado antes, los repositorios sobre los cuales trabajan los actuales procesos de mediación sólo almacenan información de las interfaces en forma de IDLs, lo que permite sólo llevar a cabo actividades de compatibilidad sintáctica entre componentes. Siguiendo la línea de los trabajos existentes [Leavens y Sitaraman, 2000] [Vallecillo et al., 1999], es necesario extender estos repositorios para que contemplen también información semántica y de protocolos de los componentes. Esta extensión debería quedar también reflejada en el modelo de documentación de componentes COTS.

2. Por otro lado, es necesaria una notación para definir arquitecturas de software a partir de componentes COTS, donde se establezcan las restricciones de la aplicación de software que se desea construir.

3. También, es necesario contar con un modelo de mediación para componentes COTS que funcione en sistemas abiertos y distribuidos, que busque y seleccione instancias de componentes que cumplan con las restricciones de los componentes impuestas en la arquitectura de software.

4. Por último, también es necesario un proceso automático que genere las posibles combinaciones entre los componentes extraídos por el proceso de mediación, para dar soluciones a la arquitectura de software de la aplicación que se construye.

Contribución

Nuestra contribución ofrece una propuesta de solución para el "problema de la conexión diseño-implementación" que surge en el campo del desarrollo de software basado en componentes comerciales (DSBC-COTS), comentado anteriormente. La propuesta está basada en un modelo de mediación que intenta "conectar" aquellas especificaciones de los componentes abstractos que se definen en fase de diseño cuando se desarrolla la arquitectura de software de una aplicación, y las de los componentes software —residentes en repositorios conocidos— relacionadas con unas implementaciones concretas desarrolladas por terceras partes (componentes comerciales).

Las contribuciones principales se desglosan en las siguientes aportaciones concretas:

Un modelo de documentación de componentes COTS

- En primer lugar se ha desarrollado un modelo de documentación de componentes COTS basado en un lenguaje en XML-Schemas (W3C) para elaborar plantillas (*template*) de documento en XML (llamadas COTScomponent). Este tipo de documento COTS permite recoger cuatro clases de información para un componente comercial. Por un lado recoge información de tipo funcional, para especificar las interfaces que el componente proporciona y requiere para su funcionamiento. Como información funcional también se describe el comportamiento semántico de dichas interfaces y sus protocolos de interacción, que indican el orden en el que deben ser llamados los métodos de las interfaces del componente. En segundo lugar, el modelo de documentación también recoge información de tipo extra-funcional, que está relacionada por ejemplo con aspectos de calidad de servicio (*QoS, Quality of Service*) o requisitos y atributos no funcionales (conocidos como NFRs). En tercer lugar, la información de empaquetamiento del componente, que ofrece detalles de implantación, dependencias con otros paquetes software o requisitos sobre la plataforma donde puede funcionar, como el tipo del sistema operativo, procesador o lenguaje de programación usado, entre otros factores. En cuarto y último lugar, la información de marketing, como opciones de licencia, certificados, precio, datos del vendedor, entre otros valores. Además, el modelo de documentación COTS propuesto admite usar distintas notaciones formales de especificación para ciertas partes de un documento COTS, como por ejemplo IDL, OCL, Larch, π-calculus, entre otros.

Un modelo de mediación para componentes COTS

- El trabajo también define un modelo de mediación de componentes COTS que está soportado por una herramienta llamada COTStrader, un servicio de mediación para componentes comerciales que funciona en sistemas abiertos y distribuidos. Para desarrollar el modelo, se ha analizado y extendido el modelo de mediación de ODP [ISO/IEC-ITU/T, 1997] y también se ha definido una serie de características que debe tener un servicio de mediación para componentes COTS. El servicio de mediación desarrollado tiene asociado un repositorio de documentos en XML (COTScomponent), que soporta los modelos "bajo demanda" (*push*) y "por extracción" (*pull*) para el almacenamiento de documentos COTS. Para consultar en el repositorio se ha diseñado una plantilla de consulta en XML (COTSquery) para establecer los criterios de búsqueda de componentes. Las operaciones de emparejamiento (*matching*) permitidas han sido dos, emparejamiento fuerte (o *exact*) y emparejamiento débil (o *soft*). El tipo de emparejamiento que el servicio de mediación debe usar en el proceso de búsqueda y selección, se puede indicar bien directamente en una plantilla de consulta COTSquery, estableciendo un enlace al lugar donde se encuentra el programa de emparejamiento, o bien directamente en la especificación del componente (la plantilla COTScomponent) en el momento de registrarla en el repositorio.

Un generador de configuraciones

- También se ha desarrollado un mecanismo de combinación de componentes COTSconfig que genera, desde una colección de componentes dada, una lista de configuraciones alternativas que satisfacen total o parcialmente los requisitos de los componentes de la aplicación de software que se desea construir. La colección de componentes es un repositorio calculado previamente por el proceso de mediación COTStrader desde los repositorios originales a los que tiene acceso. Para desarrollar COTSconfig se ha estudiado una estrategia de composición y los problemas que esta acarrea, como son los problemas de las lagunas y los solapamientos que se generan al hacer combinaciones de componentes con múltiples interfaces (como es el caso de los componentes COTS). Para ello, se han definido una serie de operaciones y conceptos, y se han extendido

los tradicionales operadores de reemplazabilidad y compatibilidad de componentes con múltiples interfaces, y que sustentan un algoritmo de vuelta atrás (*backtracking*) para el cálculo de la lista de configuraciones.

Entorno de pruebas

- Por último, para evaluar y validar la propuesta se ha desarrollado un entorno de pruebas a partir de dos aplicaciones de software basadas en componentes COTS: una aplicación de escritorio —que utiliza cuatro componentes: una agenda, una calculadora, un calendario y un *meeting scheduler*— y un conversor distribuido de imágenes de satélite GTS, muy utilizado por ejemplo en sistemas de información geográficos (SIG). Para las pruebas se ha elaborado un repositorio de componentes COTS a partir de algunos componentes ofrecidos por vendedores como IBM, Sun, ComponentSource, Flashline, y OpenSource RedHat Community, y utilizando las plantillas COTScomponent. También se ha desarrollado una utilidad GUI llamada COTSbrowser que facilita la edición de plantillas COTScomponent.

Estructura de la memoria

La presente memoria está organizada de la siguiente manera: en total, la memoria se ha compuesto (y en ese orden) de seis capítulos, un apéndice, un glosario de términos, un listado de acrónimos, un listado con las referencias bibliográficas usadas.

La memoria consta de
6 Capítulos
1 Apéndice
1 Glosario de términos
1 Listado de acrónimos y Bibliografía

El *Capítulo 1* pretende dar una visión general de la evolución y el estado del arte en el desarrollo de aplicaciones basadas en componentes comerciales, poniendo especial énfasis en las áreas sobre las que se sustenta nuestra propuesta, esto es, en las arquitecturas de software, la documentación de componentes y los servicios de mediación. El *Capítulo 2* describe un modelo para la documentación de componentes COTS. En él se presentan sus conceptos y mecanismos, y se describen las cuatro partes de un documento COTS: funcional, extra-funcional, empaquetamiento y marketing. La definición de un modelo para la documentación de componentes COTS es esencial para el resto de los capítulos de esta memoria. El *Capítulo 3* y el *Capítulo 4* definen, respectivamente, un modelo de mediación para componentes COTS y un mecanismo para la generación de configuraciones para sistemas abiertos y distribuidos.

La integración de nuestra propuesta dentro de una metodología en espiral para el desarrollo de aplicaciones de software basadas en componentes COTS se presenta en el *Capítulo 5*. En ese mismo capítulo también se discute un caso ejemplo de aplicación basada en COTS, siguiendo las pautas de la metodología anterior. Para concluir, el *Capítulo 6* recoge las conclusiones de nuestro trabajo. En él se describen tanto las aportaciones realizadas en esta tesis como sus posibles extensiones futuras.

Como complemento a ciertas secciones de esta memoria, se ha incluido el *Apéndice A*, que contiene la descripción en notación BNF de un documento COTS, y un listado del esquema (gramática) utilizando notación XMLSchema del W3C.

Por otro lado, algunos de los términos utilizados en esta tesis, como *software* o *servlet* entre otros, suelen ser más fácilmente reconocidos por el lector por su representación en inglés que en castellano, por lo que no han sido traducidos para evitar confusiones de interpretación. No obstante, y como complemento al texto, al final de esta memoria se ha incluido un *Glosario de términos* que contiene, para el caso de los términos en inglés, una traducción al castellano, además de una descripción. También se ha incluido un listado de *Acrónimos* usados para la redacción de esta memoria, y un apartado de *Bibliografía* para las referencias bibliográficas al texto.

Además, cada capítulo presenta una estructura similar, comenzando con una *Introducción y conceptos relacionados* del capitulo que se está tratando, luego las secciones propias de dicho capítulo, y se finaliza con dos apartados también comunes a todos ellos: uno de *Trabajos relacionados*, y otro como *Resumen y conclusiones del capítulo*.

Capítulo 1

Desarrollo de Software Basado en Componentes

Capítulo 1

Desarrollo de Software Basado en Componentes

Contenidos

El desarrollo de sistemas de software basado en componentes, o simplemente "desarrollo de software basado en componentes" (DSBC)[1], es una aproximación del desarrollo de software que describe, construye y utiliza técnicas software para la elaboración de sistemas abiertos y distribuidos mediante el ensamblaje de partes software reutilizables. La aproximación DSBC es utilizada para reducir los costes, tiempos y esfuerzos de desarrollo del software, a la vez que ayuda a mejorar la fiabilidad, flexibilidad y la reutilización de la aplicación final. Durante algunos años, DSBC fue referida como una filosofía conocida como "compre, y no construya" promulgada por Fred Brooks en 1987 [Brooks, 1987] y que abogaba por la utilización de componentes prefabricados sin tener que desarrollarlos de nuevo. Hoy día, muchos autores que trabajan en el campo de DSBC, como [Heineman y Councill, 2001] o [Wallnau et al., 2002], entre otros, empiezan a reconocer y a aceptar el uso de estándares, guías, procesos y prácticas de ingeniería, sin los cuales el desarrollo de software basado en componentes sería una mera mezcla entre competitivos y confusos lenguajes, metodologías y procesos. Estos estándares, guías, procesos y prácticas han propiciado que se empiece a hablar del término de "Ingeniería del Software Basada en Componentes" (ISBC)[2], como una subdisciplina de la "Ingeniería del Software".

En este capítulo ofrecemos una visión global de aquellas áreas de DSBC y ISBC que sustentan las bases de un método para el desarrollo de sistemas de software basado en un modelo de mediación para componentes COTS para sistemas abiertos y distribuidos. El presente capítulo está organizado en nueve secciones, comenzando con una introducción histórica a los conceptos previos en entornos abiertos y distribuidos. Describiremos la situación actual en el área de la "Ingeniería de componentes software" y veremos cómo se lleva a cabo la especificación de un componente software. También introduciremos algunas nociones de los componentes COTS y del área de la ingeniería de requisitos. Analizaremos conceptos relacionados con las arquitecturas de software y servicios de mediación. Finalizaremos con un breve resumen y algunas conclusiones de este capítulo.

1.1. Conceptos previos

La disciplina de los "sistemas distribuidos y abiertos" empezó a ser reconocida ampliamente desde hace relativamente poco tiempo. En la década de los 90, los ingenieros encargados de desarrollar y de mantener los grandes sistemas de información de la empresa, ven la necesidad de escalar y ampliar sus sistemas para dar cobertura ya no solo al personal interno de una sección, de un departamento o de una organización, si no también para dar servicio a otros miembros de la organización ubicados en diferentes localizaciones geográficas, y como no, a otros miembros externos a la organización.

Sistemas centralizados frente a sistemas descentralizados

El auge de los "sistemas distribuidos" coincide justo con la época del "boom de Internet", y con ello, un cambio radical en las metodologías de desarrollo de los sistemas. En pocos años se pasa de una mentalidad centralizada, donde prevalecía la confidencialidad y sistemas basados en *Intranet*, a una mentalidad totalmente opuesta, descentralizada y basada en *Internet*. Evidentemente, todo esto se ve influenciado por la caída progresiva de los precios de los equipos hardware y materiales de comunicación, lo cual, como hemos dicho, permitiría que en pocos años surgiera una multitud de nueva tecnología alrededor de una única idea: mantener sistemas descentralizados, distribuidos y abiertos, y generalmente —si no en su totalidad, sí en una parte— funcionando sobre Web.

[1] En la literatura se puede encontrar el término como CBD (*Component-Based Development*).

[2] En la literatura se puede encontrar el término como CBSE (*Component-Based Software Engineering*).

El paradigma de los "sistemas distribuidos" históricamente ha estado relacionado con el paradigma de la "programación distribuida", como algoritmos distribuidos, modelos para la implementación abstracta de la memoria compartida distribuida, sistemas de archivos y sistemas de gestión de bases de datos distribuidos, comunicación y paso de mensajes entre procesos concurrentes, sincronización, seguridad, y tolerancia a fallos, entre otros factores [Attiya y Welch, 1998] [Lynch, 1996].

En los años 80, el creciente aumento de los PCs, el uso de grandes computadoras y sistemas operativos Unix, y el fomento de la investigación paralela y concurrente, parecen ser la principal causa de descentralizacion de los sistemas

Como hemos dicho, el explosivo crecimiento de Internet y la enorme variedad de información disponible por la red ha dado lugar a una nueva realidad, la convergencia de estas dos disciplinas. La rápida evolución de los sistemas de computadoras y de las tecnologías de última generación tiene que estar en constante sintonía con las demandas reales de los profesionales de desarrollo de software, organizaciones y empresas.

El proceso de desarrollo de sistemas informáticos de empresa ha cambiado gradualmente en pocos años para pasar de un modelo centralizado y rígido, hacia un modelo descentralizado, abierto y distribuido. El sistema informático de una empresa —a nivel de recursos software, hardware y humanos— solía estar localizado en un mismo espacio geográfico, en un departamento o una sección de la empresa. Desde aquí, el equipo de profesionales, que tradicionalmente estaba compuesto por las categorías de analistas y programadores de sistemas, elaboraba las aplicaciones del sistema de información haciendo uso de conocimientos y prácticas tradicionales del proceso de ingeniería del software.

A mediados de los años 80 empiezan a converger diversos factores en el mundo de la informática, y que serían el detonante de un cambio en el proceso de ingeniería de los sistemas informáticos. Por un lado comienza la explosión de los PCs, que irrumpe con fuerza dentro de la empresa, básicamente en los centros de cálculo. Aunque la mayor parte de la "lógica de negocio"[3] aún residía en grandes estaciones de trabajo o en *mainframes*, la masiva presencia de equipos de bajo coste (PCs, comparados con los grandes sistemas) permitiría a los ingenieros desarrollar grandes aplicaciones desglosadas en módulos software que podían estar ubicados en distintos ordenadores, propiciando un nuevo enfoque en el desarrollo de los sistemas.

Inicialmente, estos bloques software funcionaban como elementos de cómputo independientes dentro del sistema, pero pronto, los ingenieros vieron la necesidad de disponer de nuevas técnicas para la comunicación y transferencia de los datos entre estos elementos de cómputo. Precisamente por esta fecha, y ajeno a estas necesidades, empezaban a consolidarse fuertes líneas de investigación en computación paralela y programación concurrente [Agha et al., 1993] [Milner, 1989] motivadas, en un principio, por la masiva presencia de sistemas operativos tipo Unix en sistemas multiprocesador.

Estas líneas de investigación en programación paralela y concurrente, junto con las necesidades de comunicación de procesos en ambientes de cómputo independientes, dieron lugar a los primeros esfuerzos en la elaboración de una nueva tecnología para la programación distribuida de aplicaciones [Corbin, 1991] [Raynal, 1988]. Precisamente, uno de los primeros resultados fue el desarrollo de la técnica RPC (*Remote Procedure Call*), origen de gran parte de la tecnología de interconexión actual (conocida como tecnología *middleware*). Esta técnica permite que los desarrolladores de software puedan diseñar sus aplicaciones mediante módulos software comunicantes, como si fuesen un conjunto de procesos cooperativos independientes.

Esta nueva técnica empezó a utilizarse de forma masiva en la empresa para el desarrollo de grandes sistemas de información. Pero esto provocó principalmente dos problemas. Por un lado, se empezó a echar en falta un "modelo distribuido" estándar que sirviera de guía para los ingenieros en la elaboración de sus aplicaciones distribuidas. Debido a la rápida

[3]Software crítico, de cálculo o muy ligado a los gestores de bases de datos.

utilización de la técnica RPC, se empezó a dar forma a todo un entorno de computación distribuida sin la elaboración de un marco teórico que lo sustentase.

La descentralizacion de los sistemas, y por consiguiente la distribución de las aplicaciones, hacen que aparezcan dos problemas principales. Por un lado la falta de un modelo distribuido, y por otro, la herencia de sistemas y aplicaciones existentes. Las soluciones a ello fueron el modelo distribuido DCE y la reingeniería de componentes software

Esto propició la aparición del primer modelo de distribución en 1994, conocido con el nombre de DCE (*Distributed Computation Environment*, [OSF, 1994]). Este modelo fue desarrollado por OSF (*Open Systems Foundation*), una organización formada por IBM, DEC y Hewlett-Packard. El modelo establecía las pautas y normas que los ingenieros debían seguir para desarrollar sus sistemas. Entre otras características [OSF, 1994], el modelo DCE destacó por ser un "modelo cliente/servidor" basado en el lenguaje C y que inicialmente funcionaba para plataformas Unix[4]. Posteriormente el modelo se extendió para soportar diversos sistemas operativos, como VMS, Windows y OS/2, entre otros.

Por otro lado, esta nueva mentalidad de construir aplicaciones divididas en partes comunicantes y residentes en distintos ambientes de cómputo fue un gran paso en el campo programación distribuida. Evidentemente, las antiguas aplicaciones del sistema no dejaron de funcionar, pero los ingenieros sí vieron pronto la necesidad de integrar las partes existentes del sistema con las nuevas diseñadas.

Esto dio paso a la aparición de nuevos conceptos, como los "sistemas heredados" (*legacy systems*), que hace referencia a la integración de partes software existentes con las del sistema actual. Otro concepto es el de "envolvente" (*wrapper*), que son porciones de código especialmente diseñados para encapsular y dar funcionalidad a otras partes del sistema ya existentes. O el concepto de "código de enlace" (*glue*), que son porciones de código cuyo efecto es similar al de un "pegamento" y que sirve para unir distintas partes funcionando con "envolventes".

Pero el concepto más importante que ha cambiado y sigue cambiando los procesos de ingeniería y reingeniería, es el concepto de "componente". Inicialmente este concepto surge ante la necesidad de reutilizar partes o módulos software existentes que podían ser utilizadas para la generación de nuevas extensiones de las aplicaciones, o para la generación de aplicaciones completas. Pero esto suponía un gran esfuerzo, pues había que localizar estas partes reutilizables y almacenarlas en repositorios especiales que más tarde pudieran ser consultadas en fase de diseño.

Desarrollo ascendente vs. descendente

Con el término componente se empieza a diferenciar dos estilos de desarrollo de software. Por un lado está el estilo de desarrollo de software basado en reutilización, donde las aplicaciones se construyen a partir de otras partes software ya existentes y accesibles en repositorios conocidos. Por otro lado está el desarrollo de software de reutilización, donde se ponen en práctica procesos de ingeniería para la elaboración de partes eficientes de software que luego pueden ser utilizadas para la construcción de aplicaciones (en el otro estilo de desarrollo de software). A estas partes software se las conoce como componentes software, y han dado lugar a los paradigmas de programación de componentes *top-down* o descendente (para reutilizar) y *bottom-up* o ascendente (basado en reutilización).

Modelos de objetos distribuidos

Pero el uso generalizado de los componentes en procesos de ingeniería de software realmente empieza a tomar presencia y sentido con la aparición de nuevos modelos de distribución, como CORBA, DCOM o EJB, modelos que actualmente se están utilizando para el desarrollo de aplicaciones distribuidas. Su predecesor, el modelo DCE, empieza a ser visto por los ingenieros de sistemas como un modelo difícil y costoso de llevar a la práctica.

Por este motivo, *Object Management Organization* (OMG) empezó a desarrollar un modelo para la distribución y localización dinámica de objetos en tiempo de ejecución, el

[4]La terna *cliente/servidor + Unix + C* se puso muy de moda en esas fechas. La existencia de un modelo distribuido ya reconocido hizo que la demanda de profesionales con conocimientos en estas áreas se incrementase enormemente.

modelo CORBA (*Common Object Request Broker Architecture*). Por otro lado, Sun Microsystems (tecnología Unix) y Microsoft (tecnología Windows) elaboran sendos modelos, conocidos como EJB (*Enterprise Java Beans*) y DCOM (*Distributed Component Object Model*), respectivamente. Finalmente, OMG propone el nuevo modelo de componentes de CORBA llamado CCM (*CORBA Component Model*).

La revolución industrial

Sin embargo, la presencia de distintos modelos de objetos distribuidos dentro de la empresa, cada vez más influenciada por intereses de la industria —intereses de soluciones *Sun* frente a intereses de soluciones *Microsoft*— y la fuerte evolución de nuevas tecnologías (XML, SOAP, Servlets, UDDI, entre otros), está haciendo que los ingenieros de sistemas tengan que hacer grandes esfuerzos de ingeniería para seleccionar aquellas tecnologías adecuadas para el desarrollo de sus sistemas. Incluso, en la mayoría de los casos, los ingenieros se ven obligados a utilizar e incorporar múltiples métodos y técnicas para dar soporte a distintos clientes —software y humanos— del sistema.

Por tanto, los grandes sistemas informáticos y aplicaciones software de hoy día están basados en modelos cliente/servidor con arquitecturas multicapa y que hacen uso de una gran variedad de tecnologías. La tendencia actual es que estos sistemas estén distribuidos y localizados en distintos lugares geográficos, comunicándose con modelos distribuidos CORBA, EJB y/o DCOM, haciendo uso de normas y técnicas de seguridad importantes, utilizando nuevas técnicas como XML para la representación intermedia de la información entre componentes software, o SOAP para la localización y activación automática de servicios web, entre otras muchas nuevas tecnologías.

La utilización de estándares hace que los sistemas abiertos sean más adaptables a los avances tecnológicos y a los continuos cambios de mercado

Esto último ha sido motivo para la consolidación del concepto "abierto", y que se refiere a una colección de componentes software y hardware y de usuarios que interactúan entre sí, diseñados para satisfacer las necesidades establecidas, con especificaciones de componente completamente definidas, fácilmente accesibles y mantenidas por consenso, y donde las implementaciones de componente respetan estas especificaciones [Meyers y Oberndorf, 2001].

La tendencia en los procesos de ingeniería del software para el desarrollo de sistemas abiertos y distribuidos, es elaborar sistemas colaborativos compuestos de subsistemas, componentes y objetos especializados y coordinados para ofrecer servicios. En este sentido, están empezando a distinguirse distintas subdisciplinas de la ingeniería del software conocidas como "ingenierías basadas" o "ingenierías orientadas", como por ejemplo la ingeniería del software basada en aspectos (*Aspect-Based Software Engineering*, ABSE), la ingeniería del software orientada a objetos (*Object-Oriented Software Engineering*, OOSE), la ingeniería del software basada en conocimiento (*Knowledge-Based Software Engineering*, KBSE), o la ingeniería del software basada en componentes (*Component-Based Software Engineering*, CBSE), entre otros.

El área de interés en la que se enmarca nuestro trabajo es precisamente en ésta última disciplina, la ingeniería del software basada en componentes (ISBC). En la siguiente sección trataremos algunos conceptos relacionados con ISBC.

1.2. Ingeniería de componentes software

Como adelantamos en el *Prólogo* de esta memoria, los componentes software surgen en cierta medida de la necesidad de hacer un uso correcto de software reutilizable para la construcción de aplicaciones software mediante el ensamblaje de partes ya existentes. De hecho, etimológicamente hablando, el término "componente" procede de la palabra **cumponere**, que en Latín significa **cum** "junto" y **ponere** "poner".

Desde el punto de vista de la ingeniería del software, el término "componente" procede de las "técnicas orientadas a objetos", de los problemas de descomposición usados en

"técnicas de descomposición de problemas", y de su necesidad para desarrollar sistemas abiertos. Con la aparición de los modelos de componentes COM, EJB y CCM, en pocos años ha ido emergiendo una práctica de desarrollo basada en componentes. Sin embargo, su expansión se ha visto ralentizada por la falta de acuerdo entre los especialistas, a la hora de dar una definición concreta sobre lo que es y no es un componente software [Bachman et al., 2000].

1.2.1. Definición de componente

En la literatura existe una gran diversidad de opiniones sobre lo que debe ser un componente software. Esto se pone de manifiesto, por ejemplo, en el artículo "*What characterizes a (software) component?*" [Broy et al., 1998], donde se hace una variada recopilación de definiciones de componente y ofrece una discusión sobre lo que caracteriza a un componente software. Uno de los factores que impide una definición concreta del término se debe a la falta de un acuerdo común sobre cuales son las características y propiedades que lo diferencian de un objeto [Henderson-Sellers et al., 1999]. En algunos casos, incluso, se llega a utilizar de forma indistinta los términos componente y objeto. Algo similar sucede también con el término "agente software", donde los autores tampoco se ponen de acuerdo en una definición concreta que los diferencie.

Una de las definiciones de componente software más extendidas es la de Clemens Szyperski, y que dice lo siguiente:

Componente de Szyperski

Definición 1.1 (Componente Szyperski, [Szyperski, 1998]) *Un componente es una unidad binaria de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio.*

Según Clemens Szyperski, las nociones de "instanciación", "identidad" y "encapsulación" son más propias de los objetos que de los componentes, y define un objeto como: "Una unidad de instanciación que tiene una única identidad, un estado que puede ser persistente, y que encapsula su estado y comportamiento". Sin embargo, un componente puede contener múltiples objetos, clases y otros componentes.

Granularidad de componente

La noción de componente puede variar dependiendo del nivel de detalle desde donde se mire, conocido como "granularidad de un componente". Un componente software puede ser desde una subrutina de una librería matemática, hasta una clase en Java, un paquete en Ada, un objeto COM, un JavaBeans, o incluso una aplicación que pueda ser usada por otra aplicación por medio de una interfaz especificada. Un componente con granularidad gruesa se refiere a que puede estar compuesto por un conjunto de componentes, o ser una aplicación para construir otras aplicaciones o sistemas a gran escala, generalmente abiertos y distribuidos. A medida que descendemos en el nivel de detalle, se dice que un componente es de grano fino. Un ejemplo de componentes de grano grueso son los componentes *Advanced Components* de la arquitectura WSBC (*WebSphere Business Components*) de IBM. En WSBC se define un componente de la siguiente manera:

Componente WSBC de IBM

Definición 1.2 (Componente IBM, [IBM-WebSphere, 2001]) *Una implementación que (a) realiza un conjunto de funciones relacionadas, (b) puede ser independientemente desarrollado, entregado e instalado, (c) tiene un conjunto de interfaces para los servicios proporcionados y otro para los servicios requeridos, (d) permite tener acceso a los datos y al comportamiento sólo a través de sus interfaces, (e) opcionalmente admite una configuración controlada.*

Otra definición de componente es la que adopta el SEI (*Software Engineering Institute*), y que dice lo siguiente:

COMPONENTE SEI

Definición 1.3 (Componente SEI, [Brown, 1999]) *Un componente software es un fragmento de un sistema software que puede ser ensamblado con otros fragmentos para formar piezas más grandes o aplicaciones completas.*

Esta definición se basa en tres perspectivas de un componente: (a) la perspectiva de empaquetamiento (*packaging perspective*) que considera un componente como una unidad de empaquetamiento, distribución o de entrega. Algunos ejemplos de componente de esta perspectiva son los archivos, documentos, directorios, librerías de clases, ejecutables, o archivos DLL, entre otros; (b) la perspectiva de servicio (*service perspective*) que considera un componente como un proveedor de servicios. Ejemplos son los servicios de bases de datos, las librerías de funciones, o clases COM, entre otros; (c) la perspectiva de integridad (*integrity perspective*) que considera un componente como un elemento encapsulado, como por ejemplo una base de datos, un sistema operativo, un control ActiveX, una *applet* de Java, o cualquier aplicación en general.

COMPONENTE UML

En [Cheesman y Daniels, 2001] se identifican diferentes "visiones" en las que puede aparecer el término componente en las etapas de desarrollo de un sistema software. Concretamente se identifican hasta cinco formas de componente, mostrados en la figura 1.1.

En primer lugar está la "especificación de componente", que representa la especificación de una unidad software que describe el comportamiento de un conjunto de "objetos componente" y define una unidad de implementación. El comportamiento se define por un conjunto de interfaces. Una especificación de componente es "realizada" como una "implementación de componente".

En segundo lugar está la "interfaz de componente", que es una definición de un conjunto de comportamientos (normalmente operaciones) que pueden ser ofrecidos por un objeto componente.

En tercer lugar está la "implementación de componente", que es una realización de una especificación de componente que puede ser implantada, instalada y reemplazada de forma independiente en uno o más archivos y puede depender de otros componentes.

En cuarto lugar está el "componente instalado", que es una copia instalada de una implementación de componente.

Y en quinto y último lugar está el "objeto componente", que es una instancia de un "componente instalado". Es un objeto con su propio estado e identidad única y que lleva a cabo el comportamiento implementado. Un "componente instalado" puede tener múltiples "objetos componente" o uno solo.

COMPONENTE EDOC DE OMG

Para finalizar está la visión de componente EDOC (*Enterprise Distributed Object Computing*) [OMG, 2001] del OMG (*Object Management Group*). EDOC es una especificación para la computación de objetos distribuidos que se ajusta al modelo de la arquitectura de objetos de OMG denominado OMA (*Object Management Architecture*, véase en `http://www.omg.org`). La definición de componente que se ofrece en este ambiente dice simplemente lo siguiente:

Definición 1.4 (Componente EDOC, [OMG, 2001]) *Un componente es algo que se puede componer junto con otras partes para formar una composición o ensamblaje.*

Componente Software

Como conclusión de estas definiciones podemos hacer las siguientes valoraciones. Los componentes son partes software que se pueden combinar con otros componentes para generar un conjunto aún mayor (p.e. otro componente, subsistema o sistema). Un componente juega el papel de una unidad software reutilizable que puede interoperar con otros

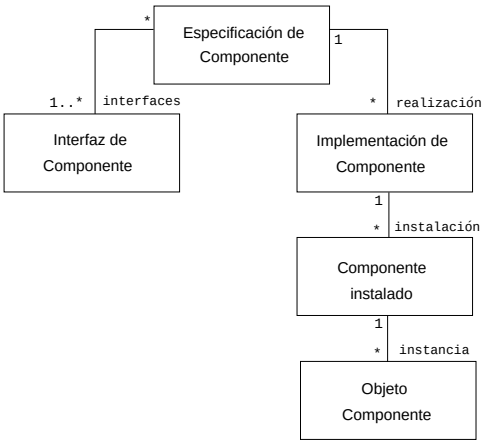

Figura 1.1: Formas en las que puede presentarse el término componente

módulos software mediante sus interfaces. Un componente define una o mas interfaces desde donde se puede tener acceso a los servicios que éste ofrece a los demás componentes.

Un componente puede presentarse en forma de código fuente o código objeto; puede estar escrito en un lenguaje funcional, procedural o en un lenguaje orientado a objetos; y puede ser tan simple como un botón GUI o tan complejo como un subsistema.

1.2.2. Ingeniería del software basada en componentes (ISBC)

Una práctica generalizada en un proyecto software es utilizar partes software ya desarrolladas en proyectos previos o adquiridos por terceras partes. Esta cultura de reutilización es esencial en casi todas las organizaciones que desarrollan software. Sin embargo, la mayoría de los desarrolladores de software suelen utilizar métodos de desarrollo internos a la organización que conducen, en la mayoría de los casos, a aplicaciones mal construidas, retrasos en los plazos de finalización del proyecto, y un aumento en los costes finales del desarrollo. Esto se debe a la falta de procesos y técnicas bien definidas que guíen a los desarrolladores de software durante la construcción de la aplicación basada en reutilización.

El sueño de los especialistas en ingeniería del software ha sido disponer de "factorías de software" donde partes software ya estandarizadas de una aplicación puedan ser automáticamente seleccionadas desde un catálogo y ensambladas e implantadas fácilmente como solución a una necesidad de desarrollo de la organización.

En cierta medida este desarrollo de software ideal se corresponde con la "visión optimista" de [Wallnau et al., 2002], como muestra la figura 1.2. El desarrollo de un sistema software es visto como fases en la resolución de un problema planteado, y que coinciden con las tradicionales del ciclo de vida (análisis, diseño e implementación). Una "visión optimista" del desarrollo de un producto software supone la utilización de "partes software" (componentes) bien especificadas, con interfaces descritas como contratos, con arquitecturas de software estándares para hacer el diseño abstracto de la aplicación que hay que construir, y con herramientas y entornos adecuados, aceptados y orientados a la composición de las partes (componentes) implementadas y/o reutilizadas y extraídas previamente desde repositorios reconocidos por procesos de selección y búsqueda.

Por otro lado, una "visión pesimista" del desarrollo —quizá la mas realista— supone que las partes software (componentes) ocultan muchas de sus propiedades, siendo difícil su acceso a la información y al comportamiento interno (*black box*) en tareas de evaluación; y en lugar de arquitecturas de software estándares se tienen particulares notaciones que

permiten definir "arquitecturas de partes software" inestables y no probadas; finalmente, en lugar de tener entornos altamente cualificados que soporten la metáfora composicional para el desarrollo de sistemas y que utilicen múltiples lenguajes (C, C++, Java, Perl, entre otros), se tienen herramientas y entornos de desarrollo especializados y particulares, por ejemplo un entorno de desarrollo Microsoft (.NET, DCOM, herramientas Visual, ASP, entre otros) y no Microsoft (CORBA, EJB, Servlets, entre otros).

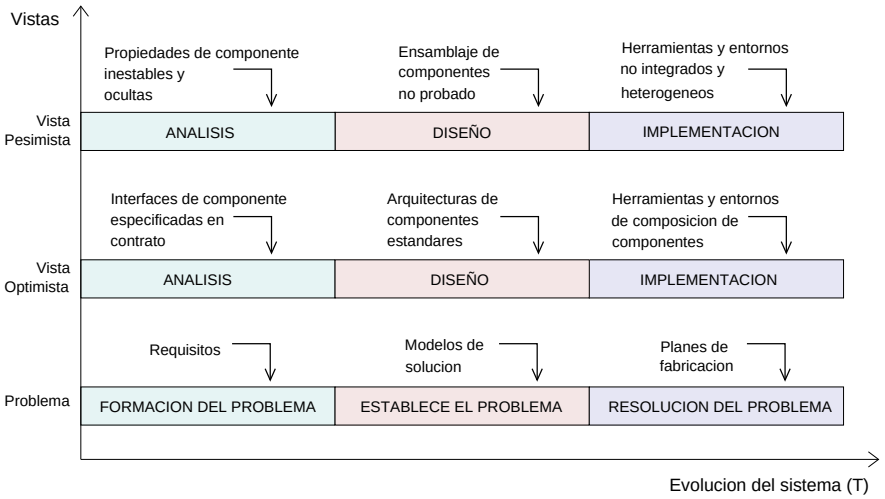

Figura 1.2: Vistas en la evolución del desarrollo de software

En los años 80 se intentó llevar a la práctica una idea similar a la que presenta la "visión optimista", recopilando y catalogando "módulos software" de aplicaciones para su reutilización, conocidos generalmente como "artefactos" o "módulos software" (*assets*). Sin embargo, la mayoría de las iniciativas de reutilización no prosperaron por varias razones obvias: (a) la infraestructura técnica que daba soporte a la reutilización estaba aún inmadura; (b) la catalogación de módulos software era muy difícil; (c) los módulos software fueron muy diversos y de calidad también muy variada; (d) las interfaces y el comportamiento de los módulos software fueron "pobremente" definidos; (e) y finalmente, la cultura de la reutilización fue infravalorada e insuficientemente apoyada por el mercado.

La confluencia de varios factores, como la madurez en los conceptos y técnicas orientadas a objetos, la consolidación del "software de interconexión" de componentes (*middleware*) como CORBA, o el progresivo aumento de Internet, entre otros factores, hace que a finales de los años 90 se empiece a hablar con fuerza del término componente software y del desarrollo basado en componentes software (*Componet-Based Development*). Algunos de los autores que han colaborado al impulso de este estilo de desarrollo han sido por ejemplo [Broy et al., 1998], [Cottman, 1998], [Garlan et al., 2000], [Kiely, 1998], [Krieger, 1998], [Meyer, 1999], o [Szyperski, 1998], entre otros.

Emerge ISBC

Incluso en estos últimos años hemos podido comprobar cómo se está haciendo un uso continuo del término "ingeniería del software basada en componentes" (*Component-Based Software Engineering*, CBSE) como una subdisciplina de la ingeniería del software para construir sistemas distribuidos y abiertos con componentes software. Algunos autores que promueven el uso del término ISBC son por ejemplo [Brown y Wallnau, 1998], [Heineman y Councill, 2001] o [Wallnau et al., 2002], entre otros autores. Estos creen que hoy día ya existen metodologías, técnicas y herramientas basadas en componentes que empiezan a estar lo suficientemente maduras como para poder empezar a hablar de una ingeniería de componentes ISBC.

Algunas de estas metodologías utilizan componentes software para la construcción de sistemas. Ejemplos de ellas son: CMM[5] (*Capability Maturity Model*) del SEI (*Software Engineering Institute*) o RUP[6] (*Rational Unified Process*) de Rational Software. En la tabla 1.1 resumimos algunas categorías de interés en la ISBC[7].

Líneas de interés en ISBC	
Procesos de desarrollo de componentes	Reutilización de componentes
Ingeniería de requisitos basada en componentes	Diseño, prueba e implementación de componentes
Construcción de sistemas con componentes	Especificación de componentes
Evaluación de componentes	Arquitecturas basadas en componentes
Composición de componentes	Tecnologías y modelos de componentes
Entornos y herramientas de desarrollo	COTS (*Commercial Off-The-Shelf*)
Métricas software basadas en componentes	Casos de estudios en ISBC

Tabla 1.1: Categorías de interés dentro de ISBC

1.2.3. Etapas DSBC y tecnología de componentes

Tradicionalmente los ingenieros del software han seguido un enfoque de desarrollo descendente (*top-down*) basado en el ciclo de vida en cascada (análisis-diseño-implementación) para la construcción de sus sistemas, donde se establecen los requisitos y la arquitectura de la aplicación y luego se va desarrollando, diseñando e implementando cada parte software hasta obtener la aplicación completa implementada. En ISBC, la idea de construir un sistema escribiendo código ha sido sustituida por una idea basada en el ensamblaje e integración de componentes software ya existentes (desarrollo ascendente o *bottom-up*).

En el tiempo han sido numerosas las propuestas para el Desarrollo Basado en Componentes. El método RUP es uno de las más conocidos, y actualmente el método está siendo usando como práctica en Rational y en los Componentes UML

El proceso desarrollo de sistemas basados en componentes (DSBC) está compuesto por diferentes etapas, aunque en la literatura podemos encontrar distintas categorías y formas a la hora de referirse a estas etapas. Por ejemplo en [Dean y Vigder, 1997] se identifican cuatro procesos de desarrollo de sistemas basados en componentes, que son: (a) la búsqueda de componentes software (residentes en repositorios) que pueden satisfacer las necesidades del usuario o de la arquitectura de software de la aplicación; (b) la evaluación de componentes según algunos criterios; (c) la adaptación o extensión de los componentes seleccionados para que se ajusten a las necesidades de la arquitectura del sistema; y (d) el ensamblaje o la integración de estos componentes.

Otra clasificación de las etapas del desarrollo de los sistemas basados en componentes, es la de RUP (*Rational Unified Process*) [Jacobson et al., 1999], un proceso de desarrollo adoptado por Rational (`http://www.rational.com`) y por los "Componentes UML" [Cheesman y Daniels, 2001] (véase sección 1.6.4). En este caso el proceso se divide en seis etapas, que son: (a) fase de requisitos, donde se identifican los requisitos de los componentes, de la arquitectura de software y del sistema en general; (b) fase de especificación, donde se realiza la especificación de los componentes y de la arquitectura de software[8]; (c) fase de aprovisionamiento de componentes, donde se buscan y seleccionan componentes; (d) fase de ensamblaje de los componentes encontrados; (e) fase de pruebas; y (f) fase de implantación de la aplicación del sistema.

[5] `CMM: http://www.cmu.sei.edu`

[6] `RUP: http://www.rational.com`

[7] Estas categorías han sido obtenidas de Conferencias en ISBC, como el *Component-Based Software Engineering Track* de las Conferencias del EUROMICRO.

[8] En RUP el concepto de especificación coincide con aspectos de diseño, y se utilizan diagramas de casos de uso, diagramas de clases y diagramas de colaboración de UML. En la sección 1.6.4 se tratará todo esto.

Por último, también destacamos la clasificación de [Brown, 1999]. En este caso el proceso de desarrollo está compuesto de cuatro etapas: (a) la selección de componentes; (b) la adaptación de componentes; (c) el ensamblaje de los componentes al sistema; y (d) la evolución del sistema.

En la figura 1.3 hemos preparado un cuadro comparativo entre diferentes visiones del proceso de desarrollo de los sistemas basados en componentes. Dada su similitud con otras propuestas, hemos tomado como referencia la visión de clasificación de [Brown, 1999] como la más extendida en el proceso de desarrollo de los sistemas basados en componentes. En la parte superior de la figura hemos incluido la visión del ciclo de vida clásico. También hemos incluido la visión RUP (por su relación con los Componentes UML que veremos más adelante) y una visión del proceso de desarrollo de sistemas basados en componentes comerciales; para este caso hemos seleccionado la que se ofrece en [Meyers y Oberndorf, 2001] por ser una de las más extendidas en el campo de los componentes comerciales. Estas etapas serán descritas en la sección 1.4.4 (página 27), cuando tratemos los sistemas basados en componentes COTS.

Figura 1.3: Comparación entre diferentes procesos de desarrollo basados en componentes

Como vemos, aunque no existe un consenso generalizado en las etapas del proceso de desarrollo de un sistema basado en componentes software, sí que podemos observar que algunas de ellas coinciden (aunque con distinto nombre) o están englobadas en dos o más etapas. Esto último sucede por ejemplo con la etapa de "Selección de componentes" de CBD, que cubre las etapas de "Requisitos", "Análisis" y parte de la etapa de "Aprovisionamiento" del proceso RUP. En otros casos el término conlleva las mismas tareas pero con distinto nombre. Por ejemplo, la etapa de "Evaluación" en el proceso COTS es similar al término "Adaptación" en CBD, y parecido al "Aprovisionamiento" en RUP, aunque como vemos, éste último requiere una parte de selección en los otros procesos.

1.2.3.1. Selección de componentes

La "selección de componentes" es un proceso que determina qué componentes ya desarrollados pueden ser utilizados. Existen dos fases en la selección de componentes: la fase de búsqueda y la fase de evaluación. En la fase de búsqueda se identifican las propiedades de un componente, como por ejemplo, la funcionalidad del componente (qué servicios proporciona) y otros aspectos relativos a la interfaz de un componente (como el uso de estándares), aspectos de calidad que son difíciles de aislar[9] y aspectos no técnicos, como la cuota de mercado de un vendedor o el grado de madurez del componente dentro de la organización. Sin embargo, la fase de búsqueda es un proceso tedioso, donde hay mucha información difícil de cuantificar, y en algunos casos, difícil de obtener.

Por otro lado, para la fase de evaluación, existen técnicas relativamente maduras para efectuar el proceso de selección. Por ejemplo ISO (*International Standards Organization*) describe criterios generales para la evaluación de productos [ISO/IEC-9126, 1991]. En

[9]Como la fiabilidad, la predicibilidad, la utilidad, o la certificación [Voas, 1998], entre otros.

[IEEE, 1993] y en [Poston y Sexton, 1992] se definen técnicas que tienen en cuenta las necesidades de los dominios de aplicación. Estas evaluaciones se basan en el estudio de los componentes a partir de informes, discusión con otros usuarios que han utilizado estos componentes, y el prototipado.

1.2.3.2. Adaptación de componentes

Siguiendo con las cuatro actividades del ISBC, en segundo lugar está la actividad de "adaptación de componentes". Para este caso, debido a que los componentes son creados para recoger diferentes necesidades basadas en el contexto donde se crearon, estos tienen que ser adaptados cuando se usan en un nuevo sistema. En función del grado de accesibilidad a la estructura interna de un componente, podemos encontrar diferentes aproximaciones de adaptación [Valetto y Kaiser, 1995]:

(a) de caja blanca (*white box*), donde se permite el acceso al código fuente de un componente para que sea reescrito y pueda operar con otros componentes;

(b) de caja gris (*grey box*), donde el código fuente del componente no se puede modificar, pero proporciona su propio lenguaje de extensión o API;

(c) de caja negra (*black box*), donde el componente sólo está disponible en modo ejecutable (binario) y no se proporciona ninguna extensión de lenguaje o API desde donde se pueda extender la funcionalidad.

1.2.3.3. Ensamblaje de componentes

En tercer lugar, para "ensamblar"[10] los componentes en el sistema existe una infraestructura bien definida (conocida como *middleware*), la cual puede estar basada en diferentes estilos. Los más conocidos son el bus de mensajes MOM (*Message-Oriented Middleware*) y la tecnología ORB (*Object Request Broker*).

Message-Oriented Middleware (MOM)

La tecnología MOM es una infraestructura cliente/servidor que mejora la interoperabilidad, portabilidad y flexibilidad de los componentes de una aplicación permitiendo que esta sea distribuida en múltiples plataformas heterogéneas. Es también conocida como tecnología *Queuing* de encolado de mensajes, incorporado por ejemplo en el modelo de objetos COM+ de Windows 2000 y Windows NT, y en el último modelo .NET para Windows NT y XP.

La tecnología MOM es una tecnología asíncrona que reduce la complejidad de desarrollo al ocultar al desarrollador detalles del sistema operativo y de las interfaces de red. MOM está basada en el uso de colas de mensajes que ofrecen almacenamiento temporal cuando el componente destino está ocupado o no está conectado.

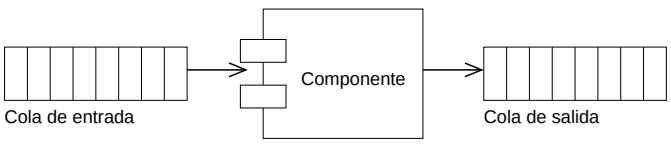

Figura 1.4: Cola de mensajes de un objeto basado en MOM

[10] En la literatura, en ocasiones podemos encontrar el término "Ensamblaje" referido como fase de "Integración" de componentes software.

Como ilustra la figura 1.4, un objeto basado en MOM puede hacer uso de dos colas: una de entrada, donde se almacenan los mensajes que recibe de otros; y otra de salida, donde se almacenan los mensajes hacia otros.

La tecnología MOM es apropiada para aplicaciones basadas en eventos (como las que propugna ahora Microsoft). Cuando un evento tiene lugar, la aplicación cliente delega a la aplicación intermedia (la aplicación con servicio MOM) la responsabilidad de notificar al servidor que se tiene que llevar a cabo alguna acción dentro del sistema.

Object Request Broker (ORB)

Un ORB (*Object Request Broker*) es una tecnología de interconexión (conocido como *middleware*) que controla la comunicación y el intercambio de datos entre objetos. Los ORBs mejoran la interoperabilidad de los objetos distribuidos ya que permiten a los usuarios construir sistemas a partir de componentes de diferentes vendedores.

Los detalles de implementación del ORB generalmente no son de importancia para los desarrolladores. Estos sólo deben conocer los detalles de las interfaces de los objetos, ocultando de esta forma ciertos detalles de la comunicación entre componentes. Las operaciones que debe permitir por tanto un ORB son básicamente tres: (a) la definición de interfaces, (b) la localización y activación de servicios remotos, y (c) la comunicación entre clientes y servicios.

La definición de interfaces se lleva a cabo mediante el uso de un "lenguaje de definición de interfaces" (IDL). Debe proporcionar al desarrollador un programa desde donde poder definir las interfaces de los componentes. También es necesario un programa que compile la interfaz y genere código ORB (desde C, Java, Tcl, y otros, dependiendo de la versión ORB) para posteriormente implementar la funcionalidad del servicio que se esté construyendo.

En la figura 1.5 se puede ver la utilidad de un programa ORB. Antes de funcionar la aplicación cliente, un programa ORB debe estar en ejecución, accesible desde el programa cliente. El cliente delega al programa ORB la tarea de localizar el servicio remoto (1). La ubicación es desconocida por el cliente. Una vez localizado, el ORB se encarga de activar el servicio remoto en su ubicación correspondiente (2). El proceso de activación del servicio genera una especie de código único interno de identificación de dicho servicio. Finalmente el ORB pasa este código al programa cliente (3) que lo utiliza para conectarse punto-a-punto con el servicio remoto requerido (4).

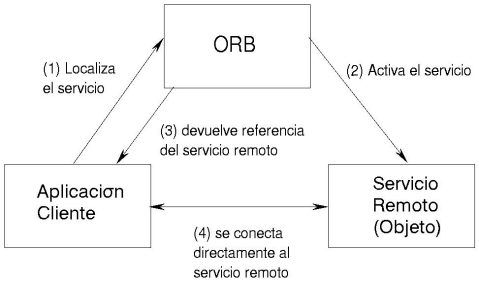

Figura 1.5: Funcionamiento básico de un ORB

Como muestra la tabla 1.2, existen básicamente tres tecnologías ORB ampliamente utilizadas, como son CORBA, COM y EJB. El hecho de utilizar un tipo de modelo u otro para ensamblar componentes, puede depender de múltiples factores; por ejemplo, por las necesidades actuales de los requisitos del sistema, o por las preferencias profesionales de los ingenieros, o por restricciones económicas, entre otros factores.

ORB	Descripción
CORBA	*Common Object Request Broker Architecture* del OMG (*Object Managament Group*), que la engloba en el modelo distribuido CCM (*CORBA Component Model*). Está basado en el protocolo para objetos distribuidos IIOP (*Internet Inter-ORB Protocol*).
COM/DCOM/COM+	Es el modelo de componentes de Microsoft (*Component Object Model*), que la engloba dentro de su modelo distribuido .NET, basado en lenguaje `C#`. Está basado en el protocolo para objetos distribuidos ORPC (*Object Remote Procedure Call*).
Java/RMI	RMI (*Remote Method Invocation*) es una parte de JVM de Sun, que la engloba dentro de su modelo distribuido EJB (*Enterprise JavaBeans*). Está basado en el protocolo para objetos distribuidos JRMP (*Java Remote Method Protocol*).

Tabla 1.2: Tecnología ORB para el ensamblaje de componentes

ORB	COM/DCOM	CORBA	RMI
Plataforma	Plataforma PC básicamente; aunque existen variaciones para otras plataformas	Plataformas independientes y plataformas que permiten interoperabilidad	Toda aquella que ejecute Java Virtual Machine (JVM)
Aplicable	Arquitecturas de sistemas distribuidos basados en PC	Arquitecturas de sistemas distribuidos en general	Arquitecturas de sistemas distribuidos en general y en Internet basado en Web
Mecanismo	APIs y DLLs	Especificación de tecnología de objetos distribuidos	Implementación de tecnología de objetos distribuidos
Implementaciones	Una. Microsoft mantiene la única implementación para PC, aunque trabaja con otros vendedores para otras plataformas	Muchas: Orbix (IONA) Neo (Sun) VisiBroker (Visigenic) DSOM (IBM), etc.	Varias

Tabla 1.3: Comparación de los ORBs DCOM-CORBA-RMI

1.2.3.4. Evolución del sistema

Los sistemas basados en componentes deberían ser fácilmente evolucionables y actualizables. Cuando un componente falla (por cualquier motivo) éste debe poder cambiarse por otro equivalente y con las mismas prestaciones. De igual forma, si un componente del sistema debe ser modificado, para que incluya nuevas funcionalidades o elimine algunas de ellas, esto se puede hacer sobre un nuevo componente que luego será cambiado por el que hay que modificar. Sin embargo, este punto de vista es poco realista. La sustitución de un componente por otro suele ser una tarea tediosa y que consume mucho tiempo, ya que el nuevo componente nunca será idéntico al componente sustituido, y antes de ser incorporado en el sistema, éste debe ser perfectamente analizado de forma aislada y de forma conjunta con el resto de los componentes con los que debe ensamblar dentro del sistema.

1.3. Especificación de componentes software

Como adelantamos, DSBC es un paradigma para desarrollar software, donde todos los artefactos (desde el código ejecutable hasta los modelos de especificación de interfaces, arquitecturas y modelos de negocio) pueden ser construidos mediante el ensamblaje, la adaptación y la instalación de todos ellos juntos y en una variedad de configuraciones. Sin embargo, esto no sería posible sin un comportamiento claro y completamente especificado de los componentes.

Un componente software requiere de información de especificación para los usuarios y los implementadores del módulo. En el contexto de reutilización del software, la especificación ayuda a determinar si un módulo puede satisfacer las necesidades de un nuevo sistema. En el contexto de interoperación del módulo, la especificación se utiliza para determinar si dos módulos pueden interoperar.

Por lo tanto, podemos decir que un componente software C puede quedar caracterizado de la siguiente forma [Han, 1998]:

$$C = (Atr + Oper + Even) + Comp + (Prot * Esc) + Prop$$

Según esto, los atributos (*Atr*), las operaciones o métodos (*Oper*) y los eventos (*Even*) son parte de la interfaz de un componente, y representa su nivel sintáctico. El comportamiento (*Comp*) de estos operadores representa la parte semántica del componente. Otros autores se refieren a estos niveles como "nivel estático" y "nivel dinámico" [Konstantas, 1995], o como componentes "plug" y "play" [Bastide y Sy, 2000].

Los protocolos (*Prot*) determinan la interoperabilidad del componente con otros, es decir, la compatibilidad de las secuencias de los mensajes con otros componentes y el tipo de comportamiento que va a tener el componente en distintos "escenarios" (*Esc*) donde puede ejecutarse. Finalmente, el término *Prop* se refiere a las "propiedades" extra-funcionales que puede tener el componente (como propiedades de seguridad, fiabilidad o rendimiento, entre otras). Veamos a continuación algunos detalles de estas partes o aspectos que caracterizan a un componente software.

1.3.1. Interfaces

Un componente software puede quedar identificado por medio de una o más interfaces. Tradicionalmente a las interfaces se las conocían con el nombre de API (*Application Programming Interface*). Una interfaz es: "una abstracción de un servicio, que define las operaciones proporcionadas por dicho servicio, pero no sus implementaciones". En términos de objetos, una interfaz puede contener también un conjunto de "atributos", además de las operaciones proporcionadas. Por ejemplo, en la figura 1.6 se muestra un estereotipo de interfaz en notación UML para un buffer con un atributo y dos operaciones.

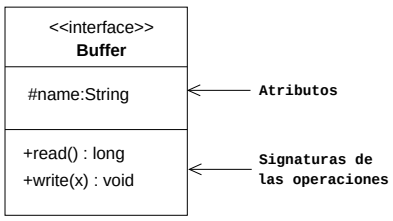

Figura 1.6: Elementos de una interfaz en notación UML

Los atributos son uno de los aspectos relevantes de una interfaz ya que son los elementos que forman parte de la "vista externa" del componente (los representados como públicos). Estos elementos observables son particularmente importantes desde el punto de vista del control y del uso del componente, y son la base para el resto de los aspectos que caracterizan a un componente.

En general, una interfaz puede presentarse de distintas maneras, dependiendo del nivel de detalle que se quiera describir o de la perspectiva desde donde se mire. Por ejemplo, en CORBA las interfaces de los objetos siguen un enfoque orientado a objetos, formadas por

el conjunto de las variables y métodos del objeto accesibles de forma pública. En COM los objetos pueden tener más de una interfaz, cada una de ellas con las signaturas de las operaciones que admite el componente. Esto sucede también en el nuevo modelo de componentes de CORBA (CCM [OMG, 1999]), en donde además, se contemplan las operaciones que los objetos necesitan de otros, y los eventos que emite y recibe el componente.

De esta forma, en los actuales modelos de componentes las interfaces están formadas por el conjunto de las signaturas de las operaciones entrantes y salientes de un componente. Las primeras determinan las operaciones que un componente implementa, mientras que las salientes son las que precisa utilizar de otros componentes durante su ejecución.

Una interfaz contiene sólo información sintáctica de las signaturas de las "operaciones" entrantes y salientes de un componente, con las cuales otros componentes interactúan con él. A este tipo de interacción se le conoce con el nombre de "control proactivo" —las operaciones de un componente son activadas mediante las llamadas de otro.

Como ejemplo, en la tabla 1.4 mostramos las interfaces de un componente buffer de una sola celda definidas utilizando el IDL de CORBA. El componente se corresponde con el comportamiento de un buffer con dos operaciones usuales: `write()` y `read()`. Estas dos operaciones se muestran en la columna de la izquierda como una interfaz ofrecida por el componente. El buffer también hace uso de otro componente que imprime los valores de su celda utilizando la operación `print()` cada vez que un nuevo valor es escrito (columna de la derecha).

Interfaz ofertada	Interfaz requerida

`interface OnePlaceBuffer {`
`  void write(in long x);`
`  long read();`
`};`
	`interface out {`
	`  oneway void print(in long x);`
	`};`

Tabla 1.4: Interfaces ofrecidas y requeridas por un componente buffer de una sola celda

Además del control proactivo —en general la forma en la que se llama a una operación— existe otra forma que se denomina "control reactivo" y que se refiere a los "eventos" de un componente, como el que permite por ejemplo el modelo de componentes EJB (*Enterprise JavaBeans*). En el control reactivo, un componente puede generar eventos que se corresponden con una petición de llamada a operaciones; más tarde otros componentes del sistema recogen estas peticiones y se activa la llamada a la operación. Un símil de este funcionamiento sería por ejemplo cuando un icono de escritorio cambia de forma al pasar por encima el cursor del ratón. En este caso, el componente icono está emitiendo un evento asociado a una operación que "cambia la forma del icono".

1.3.2. Comportamiento

Sin embargo, no todas las secuencias de invocación de operaciones están permitidas. Existen restricciones operacionales que especifican los patrones de operación permisibles. Este aspecto es similar por ejemplo al que captura el diagrama de transiciones de un objeto.

Por tanto, a la hora de construir aplicaciones no basta con tener especificaciones de componente que contengan sólo el nombre las signaturas y de los atributos del componente [Yellin y Strom, 1997] [Zaremski y Wing, 1997]. Cuando desarrollamos software, también es necesario incluir especificación semántica de las interfaces, para el significado de las operaciones y la especificación de su comportamiento [Vallecillo et al., 1999].

La información a nivel "semántico" se puede describir con formalismos como las pre/post condiciones, como por ejemplo Larch [Dhara y Leavens, 1996] [Zaremski y Wing, 1997], JML (*Java Modeling Language* o JavaLarch, `ftp://ftp.cs.iastate.edu/pub/leavens/JML`) [Leavens et al., 1999] y OCL (*Object Constraints Language*) [Warmer y Kleppe, 1998]. Lenguajes de programación como Eiffel [Meyer, 1992] y SPARK [Barnes, 1997] permiten escribir especificaciones de comportamiento utilizando pre/post condiciones dentro del código. Otros formalismos para la especificación semántica son las ecuaciones algebraicas [Goguen et al., 1996], el cálculo de refinamiento [Mikhajlova, 1999], y otras extensiones de métodos formales orientados a objetos tradicionales que están siendo usados para la especificación de componentes software, como OOZE [Alencar y Goguen, 1992], VDM++ [Dürr y Plat, 1994] y Object-Z [Duke et al., 1995].

A modo de ejemplo, en la figura 1.7 mostramos una descripción pre/post, en JML (JavaLarch), del comportamiento de las operaciones del componente buffer (visto anteriormente), ofrecidas dentro de la interfaz `OnePlaceBuffer`. Como vemos, esta definición combina código Java, para la descripción de la interfaz (líneas 3, 12, 20 y 21), y notación JML, expresada entre comentarios (`//` o `/* */`) y comenzando cada línea JML por el símbolo `@`. Como vemos en la tabla, la descripción JML se coloca por encima de la descripción abstracta de cada operación.

```
1:  //@ model import edu.iastate.cs.jml.models.*;
2:
3:  public interface OnePlaceBuffer {
4:    /*@ public model JMLValueSequence unBuffer; (* un OnePlaceBuffer *)
5:      @        initially unBuffer != null && unBuffer.isEmpty();
6:      @*/
7:
8:    /*@ public normal_behavior
9:      @    requires unBuffer.isEmpty();
10:     @    ensures unBuffer@post == unBuffer.insertBack((JMLInteger) x);
11:     @*/
12:   public void write(JMLInteger x);
13:
14:   /*@ public normal_behavior
15:     @    requires !unBuffer.isEmpty();
16:     @    ensures  result == ((JMLInteger) unBuffer.first()) &&
17:     @             unBuffer@post == unBuffer.removePrefix(1) &&
18:     @             unBuffer@post.isEmpty();
19:     @*/
20:   public JMLInteger read();
21: };
```

Figura 1.7: Comportamiento en JavaLarch de un buffer de una sola celda

1.3.3. Protocolos (coreografía)

Además de la información sintáctica para las interfaces del componente y de la información semántica para el comportamiento de las operaciones de las interfaces, se ha visto que también es necesaria otro tipo de información semántica que concierne al orden en el que deben ser llamadas las operaciones de las interfaces de un componente. A esta información semántica se la conoce con el nombre de "protocolos" de interacción (también llamada "coreografía"). Los protocolos de interacción pueden variar dependiendo del tipo

de componente con el que interacciona y también del "escenario" donde se lleva a cabo la interacción. En este caso, a las interfaces se las denominan "roles". Por tanto, un componente puede ser usado en diferentes escenarios, donde sus interfaces pueden jugar un papel diferente en cada uno de ellos. Así pues, desde el punto de vista del componente, se podría hablar de diferentes protocolos de interacción en función del escenario donde pueda ser utilizado [Han, 1998].

Para especificar información de protocolos existen diferentes propuestas —dependiendo del formalismo— como redes de Petri [Bastide et al., 1999], máquinas de estados finitas [Yellin y Strom, 1997], lógica temporal [Han, 1999] [Lea y Marlowe, 1995] o el π-cálculo [Canal et al., 2001] [Canal et al., 2003] [Henderson, 1997] [Lumpe et al., 1997]. Existen otros lenguajes para la sincronización de componentes (otra forma de referirse a los protocolos), como Object Calculus [Lano et al., 1997], Piccola [Acherman et al., 1999] o ASDL (*Architectural Style Description Language*) [Riemenschneider y Stavridou, 1999].

A modo de ejemplo, en la figura 1.8 mostramos una descripción de un protocolo para el caso del componente buffer de una sola celda, visto en la sección anterior. Para la descripción se ha utilizado la notación de π-cálculo.

```
OnePlaceBuffer(ref,out) =
   ref?write(x,rep).out!print(x).rep!().Full(ref,out,x);
Full(ref,out,x) =
   ref?read(rep).rep!(x).OnePlaceBuffer(ref,out);
```

Figura 1.8: Protocolo en pi-calculus de un buffer de una sola celda

1.3.4. Información de calidad y extra-funcional (NFR)

Otro aspecto importante es la información de calidad de un componente, recogido en la literatura como atributos de calidad. Un atributo de calidad se refiere tanto a información de calidad de servicio[11] —p.e., el tiempo máximo de respuesta, la respuesta media y la precisión— como a atributos relacionados con la funcionalidad y la extra-funcionalidad de un componente, como por ejemplo la interoperabilidad y la seguridad para el caso de los atributos funcionales, o la portabilidad y la eficiencia para el caso de los atributos extra-funcionales, entre otros muchos [ISO/IEC-9126, 1991]; aunque la mayor parte de los atributos de calidad están relacionados con la información extra-funcional.

La información extra-funcional se puede clasificar de muy diversas formas. Por ejemplo, en [Franch, 1998] se utiliza una notación NoFun que clasifica la información extra-funcional en tres categorías: (a) atributos extra-funcionales o propiedades (*NF-attributes*); (b) comportamiento extra-funcional de una implementación de componente (*NF-behavior*); y (c) requisitos extra-funcionales de un componente software (*NF-requirements*).

La información extra-funcional también viene recogida con otros términos en la literatura tradicional, como información extra-funcional, restricciones extra-funcionales NFR (*Non-Functional Restrictions*), o *ilities* [Bass et al., 1998, Thompson, 1998]. Para este último caso, el término *ilities* hace referencia a todas aquellas propiedades extra-funcionales, originarios del inglés, y que generalmente terminan por "*ility*", como por ejemplo: "*reliability*", "*portability*", "*usability*", o "*predictability*"; sin embargo hay otros que no terminan así, y que también son de la misma categoría extra-funcional, por ejemplo: "*safety*" o "*maturity*", entre otros. En la literatura existen dos referencias clásicas en el concepto de

[11] QoS, *Quality of Service*

Planificación	Diseño	Ejecución
Robustez	Portabilidad	Prestaciones
- Integridad conceptual	Reutilidad	- Ancho de banda
- Completitud	Comprobable	- Tiempo de respuesta
- Coherencia	Extensibilidad	- Recursos que consume
Permisibilidad	Configurable	Fiabilidad
- Precio de compra	Escalabilidad	- Disponibilidad
- Coste de mantenimiento	Interoperabilidad	- Vulnerabilidad
- Coste de integración		- Seguridad
		- Tolerancia a fallos
		Utilidad

Tabla 1.5: Atributos extra-funcionales en las etapas de desarrollo [Bass et al., 1998]

"*ilities*": [Azuma, 2001] y [Chung et al., 1999]. El primero es un estándar ISO 9126, y el segundo recoge y clasifica más de 100 *ilities*.

La tabla 1.5 muestra algunas propiedades extra-funcionales que pueden ser tratadas en distintas fases del ciclo de vida en la construcción de un sistema [Bass et al., 1998]. Por sus características, la información extra-funcional puede hacer que su especificación sea difícil de llevar a cabo [Chung et al., 1999], por ejemplo: (1) La información extra-funcional puede ser "subjetiva", ya que ésta puede ser interpretada y evaluada por diferentes personas (analistas, diseñadores, programadores); (2) La información extra-funcional puede ser "relativa", ya que su importancia y su descripción puede variar dependiendo del dominio de aplicación (planificación, diseño, ejecución); (3) La información extra-funcional puede ser "derivada", en el sentido de que pueden aparecer nuevas propiedades funcionales a partir de una determinada, como por ejemplo para la propiedad "Prestaciones" o la propiedad "Fiabilidad" (véase tabla 1.5).

Pero además de sus características, algunas tareas como la elicitación, análisis y evaluacion de la información extra-funcional se ven dificultadas por la conveniencia de una notación adecuada para recoger esta clase de información en los componentes software, como se pone de manifiesto en [Franch, 1997]. En la sección 1.5.3 veremos algunos trabajos relacionados con información extra-funcional de los componentes COTS.

1.3.5. Contratos y credenciales

Los contratos y las credenciales son otra forma de referirse a la información funcional (concretamente la semántica) y a la información extra-funcional de la especificación de un componente software, como vimos anteriormente en la sección 1.3.1.

Los contratos son una forma de garantizar que las partes software de un sistema, desarrolladas por diferentes personas —y posiblemente de diferentes organizaciones—, puedan funcionar correctamente de forma conjunta.

Si consideramos un componente como una "pieza" software que proporciona y requiere unos servicios para su funcionamiento, también podríamos considerar una relación similar entre las compañías que desarrollan software y los componentes. En este sentido, las compañías se verían como entidades que proporcionan servicios (componentes) a sus clientes (que pueden ser otras compañías, organizaciones o clientes independientes) y que también pueden depender de los servicios de otras compañías. De igual forma que en la vida real los contratos sirven para "cerrar" acuerdos alcanzados entre dos o más partes (compañías, organizaciones, personas), éstos (los contratos) también pueden ser utilizados como acuerdos formales de especificación entre partes software.

Los contratos son una metáfora cuyo significado es muy útil en el área del desarrollo de software basado en componentes (DSBC) [Bachman et al., 2000]. Por ejemplo: (a) En la vida real los contratos tienen lugar entre dos o más partes; (b) estas partes normalmente negocian los detalles del acuerdo antes de firmar un contrato; (c) además los contratos están sujetos a normativas y convenios de comportamiento sobre todos los firmantes; (d) y que una vez firmadas, las condiciones del contrato no pueden ser modificadas, a no ser que los cambios puedan ser renegociados por todas partes firmantes del actual contrato.

Esta visión realista del término "contrato" aparece de diferentes formas en DSBC. Por ejemplo, la perspectiva (a) inspira un caso de contrato que se da en coordinación de componentes, como en la composición de componentes. En este caso, múltiples componentes se ponen de acuerdo (contractualmente) para implementar un fin común, como por ejemplo, un componente más simple, o un subsistema. La perspectiva (b) inspira un caso de contrato que se da en colaboración entre componentes. En este caso los contratos son utilizados como acuerdos para negociar las condiciones de interacción entre múltiples componentes que colaboran para conseguir un bien particular, en lugar de uno común (como en el caso anterior). Ejemplo de estos últimos son los agentes bursátiles. La perspectiva (c) tiene implicaciones en el área de la certificación de componentes. Por último, la perspectiva (d) tiene implicaciones en el área del desarrollo de estándares para el mercado de componentes. En nuestro caso, sólo nos centraremos en los contratos de la primera perspectiva.

La concepción realista del término "contrato" ha inspirado su aplicación en áreas de los componentes software como en la composición, la colaboración o la certificación, entre otros

Como hemos adelantado antes, las interfaces de un componente contienen la definición abstracta de las operaciones y atributos. Originariamente, las interfaces podían ser descritas imponiendo también condiciones de uso en un solo sentido, y que especificaban cómo un cliente podía acceder o utilizar los servicios (operaciones) implementados por un componente. Sin embargo, en el desarrollo de sistemas basados en componentes, estas condiciones de uso en un solo sentido (del componente al cliente) se convierten en condiciones recíprocas de doble sentido entre el cliente y el componente. Por regla general, el papel del cliente es asumido por otro componente software, y que también impone sus condiciones de uso. Por lo tanto, de igual forma que un componente servidor (el componente que ofrece servicios) impone sus restricciones de uso, un componente cliente (aquel que consume o utiliza los servicios ofertados por el componente servidor) también podría "exigir" al componente servidor que describa lo que sucede tras consumir el servicio. Estas co-dependencias se suelen definir mediante "pre" y "post" condiciones para especificar el comportamiento de las operaciones (similar al ejemplo de la figura 1.7). Su uso garantiza que un cliente pueda interpretar el comportamiento de las implementaciones de una interfaz de componente.

A la forma de describir las co-depedencias entre dos o más partes se le conoce con el nombre de "contrato de interfaz", introducidos originariamente como "contratos de interacción" [Holland, 1993]. Estos especifican las obligaciones recíprocas (co-dependencias) a través de los tipos de interfaces que pueden ser implementados por un componente. Los contratos de interacción juegan el papel de especificaciones de diseño que deberán ser cumplidas por los componentes. Además, como un componente puede implementar múltiples tipos de interfaces, éste también puede tomar diferentes roles. Por lo tanto, cada contrato de interacción describe un "patrón de interacción" a través de los diferentes roles.

Algunos modelos de componentes como EJB definen modelos de interacción (considerados como contratos) entre por ejemplo los contenedores (Containers) *y los componentes* (SessionBeans)

En 1997 Bertrand Meyer [Meyer, 1997] introduce el concepto de "diseño por contrato", una forma de construcción de software que asegura que el software es fiable desde sus comienzos, construyéndolo como colecciones de elementos que cooperan entre sí sobre la base de precisas obligaciones de cada uno de ellos (los contratos). Esto guía en los procesos de análisis, diseño, implementación, documentación y prueba, entre otros. El "diseño por contrato" está siendo utilizado como base para desarrollar "componentes de confianza" (*trusted components*) [Meyer et al., 1998]. El concepto de componente de confianza

está siendo investigado en el proyecto *Trusted Components for the Software Industry* de Bertrand Meyer, Christine Mingins y Heinz Schmidt (en `http://www.trusted-components.org` existe abundante información de la iniciativa). Como objetivo, el proyecto persigue la creación de una fundación sólida para la industria del software con amplias librerías de componentes software reutilizables y de confianza ("*componentware*"). El calificativo "confianza" se consigue combinando diferentes aproximaciones, como el uso de "Diseño por contrato", pruebas de corrección matemática, o métricas de evaluación, entre otros.

Por lo que hemos visto hasta el momento, podríamos decir que hay dos tipos de contratos: (a) los contratos entre las interfaces y los clientes (usuarios o componentes software) que utilizan dichas interfaces; y (b) los contratos entre las interfaces y sus implementaciones. Los primeros se corresponden con los "contratos de interacción" de Holland [Holland, 1993], y son contratos en tiempo de ejecución. Los segundos se corresponden con el "diseño por contrato" de Meyer [Meyer, 1997], y son contratos (como su nombre indica) en tiempo de diseño. De hecho, en [Bachman et al., 2000] podemos encontrar estos dos tipos de contratos como "contratos de interacción" y "contratos de componente". Y en [Cheesman y Daniels, 2001] como "contratos de uso" y "contratos de realización". La figura 1.9 resume las dos vistas de contrato referidas.

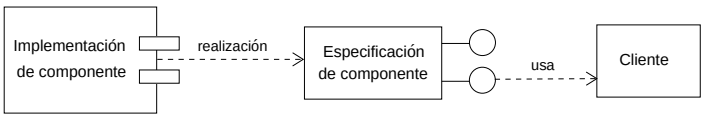

Figura 1.9: Diferentes tipos de contratos

La interfaz de un componente define el comportamiento de las operaciones. La especificación de un componente define las características de implementación e implantación del componente, y cómo interactúa con otros componentes (p.e., mediante diagramas de colaboración)

Un contrato de uso describe la relación entre una interfaz del objeto componente y un cliente, y se describe (el contrato) en forma de interfaz. La especificación contiene descripción sintáctica de las operaciones de la interfaz y la descripción del comportamiento de las operaciones en la forma de pre/post condiciones. Por otro lado, un contrato de realización describe la relación entre una especificación de componente y una implementación de componente, que debe ser asumida por la persona que creó la implementación (el realizador).

Por último, las credenciales son una forma de recoger las propiedades extra-funcionales de un componente [Shaw, 1996]. Una credencial (propiedad extra-funcional) se define como una terna `<propiedad, valor, confianza>`, donde `property` es el nombre de la propiedad de componente, `valor` es el valor de esta propiedad para un componente en particular, y `confianza` es el grado de certeza del valor dado para esta propiedad.

1.4. Ingeniería del software basada en componentes COTS

Desde hace tiempo, la reutilización de software ha venido siendo una práctica común para la construcción de productos software. La reducción de los costes, tiempos y esfuerzos en los procesos de elaboración han sido algunos de los motivos que han llevado a los ingenieros de software a considerar técnicas para la reutilización de partes software en prácticamente cualquier fase del ciclo de vida del producto (análisis, diseño e implementación). Estas partes software, generalmente, se corresponden con fragmentos de código, procedimientos, librerías y programas desarrollados en otros proyectos dentro de la organización, y que pueden ser utilizados de nuevo para ser incorporados en ciertas partes del nuevo producto que hay que desarrollar.

Además, en estos últimos años hemos podido comprobar un aumento en el uso de componentes comerciales en prácticas de reutilización de software. Concretamente, estos

componentes comerciales, que comúnmente se conocen con el nombre de componentes COTS (*Commercial Off-The-Shelf*), están siendo considerados con mayor asiduidad para la construcción de sistemas complejos, distribuidos y abiertos. Para la elaboración de estos sistemas, los ingenieros utilizan metodologías, procesos y técnicas de desarrollo basados en componentes (DSBC). El sistema a desarrollar estará compuesto por una o más aplicaciones software, que pueden ser consideradas o no como componentes. Incluso puede que algunas de estas aplicaciones software hayan sido construidas mediante la composición de otras partes software (componentes) durante el desarrollo del sistema. Estas partes software ensambladas han podido ser obtenidas de muy diversas formas:

a) *Desarrolladas*. En este caso los componentes son construidos directamente por la organización dentro del proyecto de desarrollo del sistema, y todavía sigue siendo (aunque cada vez menos) la práctica habitual y tradicional en la elaboración de un producto software.

b) *Reutilizadas*. Otra posibilidad es que los componentes hayan sido reutilizados a partir de repositorios, propios a la organización, que contienen código (fuente y ejecutable) bien definido, desarrollado en otros proyectos de la organización, y que pueden ser utilizados en nuevos proyectos. Sin embargo, esta no es una práctica muy habitual dentro de las organizaciones, ya que por regla general no se disponen de estos repositorios internos con partes software (componentes) con especificaciones bien definidas, y en ocasiones hay que aplicar prácticas de *reingeniería inversa*, donde a partir del código y de la documentación comentada dentro del código (si lo hay) se intenta extraer sus características de funcionamiento para comprender su comportamiento.

c) *Adquiridas*. Una última posibilidad consiste en adquirir el componente software a través de terceras partes, en lugar de construirlo, reduciendo con ello el tiempo, coste y esfuerzo que conlleva el desarrollo del componente. No obstante, esto supone también disponer de repositorios y catálogos de componentes comerciales conocidos, con especificaciones bien definidas y estandarizadas, con adecuadas técnicas y procesos de búsqueda, selección y evaluación de componentes: en definitiva, disponer de un mercado de componentes comerciales consolidado. Desafortunadamente, todo esto no sucede en la realidad, aunque es cierto que en el área ISBC existen grandes esfuerzos de investigación en estos últimos años para llegar a conseguir estos objetivos.

Como adelantamos en el prólogo de esta memoria, el trabajo que hemos desarrollado se enmarca precisamente dentro de esta área de la ingeniería del software basada en componentes (ISBC). En el trabajo proponemos un método para elaborar sistemas de software a partir de componentes COTS, basada en un modelo de mediación (*trading*) que efectúa labores de búsqueda y selección de componentes dentro de un repositorio, con documentos de especificación de componentes COTS que también hemos desarrollado.

Como una justificación a los siguientes apartados de este capítulo, debemos decir que el proceso se caracteriza básicamente por tres aspectos: (a) Los "requisitos" de los componentes, que se establecen a partir un modelo de documentación de componentes COTS; (b) La definición de la "arquitectura de software" del sistema que hay que construir en base a los componentes COTS; y (c) Un "servicio de mediación" (*trading service*) que localiza documentos de componentes COTS ya existentes que coinciden con aquellos documentos necesitados y definidos en la arquitectura de software.

En esta sección vamos estudiar algunas de las propiedades que caracterizan a un componente COTS, su relación con los sistemas abiertos y estándares, y de forma general, el estado actual en los procesos de desarrollo de software con este tipo de componente.

1.4.1. Componentes comerciales (COTS)

El término COTS, como sucede con muchos otros términos en el campo de la informática (como por ejemplo Internet), surge desde el Ministerio de Defensa de los Estados Unidos [Oberndorf, 1997]. Históricamente hablando, el término COTS se remonta al primer lustro de los años 90, cuando en Junio de 1994 el Secretario de Defensa americano, William Perry, ordenó hacer el máximo uso posible de especificaciones y estándares comerciales en la adquisición de productos (hardware y software) para el Ministerio de Defensa. En Noviembre de 1994, el Vicesecretario de Defensa para la Adquisición y Tecnología, Paul Kaminski, ordenó utilizar estándares y especificaciones de sistemas abiertos como una norma extendida para la adquisición de sistemas electrónicos de defensa. A partir de entonces, los términos "comercial", "sistemas abiertos", "estándar" y "especificación" han estado muy ligados entre sí (aunque un término no implica los otros), estando muy presentes en estos últimos años en ISBC.

El principio básico de los sistemas abiertos es el uso de interfaces estándares junto con el uso de implementaciones que se adecuan dichas interfaces

El término "componente comercial" puede ser referido de muy diversas formas, como por ejemplo, software *Commercial Off-The-Shelf* (COTS), o *Non-Developmental Item* (NDI), o incluso *Modifiable Off-The-Shelf* (MOTS) [Carney y Long, 2000]. En realidad existen unas pequeñas diferencias entre ellos, diferencias que reflejamos en la tabla 1.6. En cualquier caso, nos referiremos a las tres categorías como componentes COTS.

Categoría	Descripción
COTS	Software que (a) existe a priori, posiblemente en repositorios; (b) está disponible al público en general; y (c) puede ser comprado o alquilado.
NDI	Software desarrollado —inicialmente sin un interés comercial— por unas organizaciones para cubrir ciertas necesidades internas, y que puede ser requerido por otras organizaciones. Por tanto es un software que (a) existe también a priori, aunque no necesariamente en repositorios conocidos; (b) está disponible, aunque no necesariamente al público en general; y (c) puede ser adquirido, aunque más bien por contrato.
MOTS	Un tipo de software *Off-The-Shelf* donde se permite tener acceso a una parte del código del componente, a diferencia del componente COTS, cuya naturaleza es de caja negra, adquirido en formato binario, y sin tener posibilidad de acceder al código fuente.

Tabla 1.6: Tipos de software comercial

Como sucede para el caso de los componentes software, en la literatura tampoco existe una definición concreta y comúnmente usada para el término COTS. Una definición híbrida del término "componente COTS" la podemos encontrar utilizando, por un lado, la definición de "componente" de [Szyperski, 1998], visto anteriormente (véase **definición Szyperski**, página 7), y por otro lado la definición de elemento "COTS" del SEI, que dice lo siguiente:

Definición 1.5 (Elemento COTS del SEI, [Brown y Wallnau, 1998]) *Un elemento COTS se refiere a un tipo particular de componente software, probado y validado, caracterizado por ser una entidad comercial, normalmente de grano grueso y que reside en repositorios software, y que es adquirido mediante compra o alquiler con licencia, para ser probado, validado e integrado por usuarios de sistemas.*

Existen otros autores, como [Addy y Sitaraman, 1999] o [Meyers y Oberndorf, 2001], que también consideran que un componente comercial no tiene necesariamente que ser

adquirido mediante compra o licencia, sino que también puede ser adquirido como software de dominio público o desarrollado fuera de la organización. Según estas perspectivas, para nuestros propósitos, entendemos por componente COTS lo siguiente:

Definición 1.6 (Componente COTS) *Un componente COTS es una unidad de elemento software en formato binario, utilizada para la composición de sistemas de software basados en componentes, que generalmente es de grano grueso, que puede ser adquirido mediante compra, licencia, o ser un software de dominio público, y con una especificación bien definida que reside en repositorios conocidos.*

1.4.2. Limitaciones del desarrollo de software con componentes COTS

Sin embargo, el uso que se hace de la definición de componente COTS conlleva una serie de limitaciones. En primer lugar, aunque es cierto que desde 1994 se están utilizando prácticas para la utilización de componentes comerciales en procesos de desarrollo, la realidad es que muchas organizaciones encuentran que el uso de componentes COTS conlleva un alto riesgo y esfuerzo de desarrollo, y para controlar su evolución y mantenimiento dentro del sistema [Dean y Vigder, 1997]. Estos problemas se deben en cierta medida, a que las organizaciones utilizan procesos y técnicas tradicionales para el desarrollo basado en componentes, pero no para componentes comerciales.

Otro inconveniente, es que los fabricantes de componentes COTS tampoco documentan de forma adecuada sus productos para que puedan ser consultados por usuarios desarrolladores que necesitan conocer detalles de especificación del componente, como información acerca de sus interfaces, protocolos de comunicación, características de implantación (tipos de sistemas operativos y procesadores donde funciona, lenguaje utilizado, dependencias con otros programas, etc.) y propiedades extra-funcionales, como las tratadas en la sección 1.3.4.

Por último, dado que no existen formas globalmente conocidas para documentar los componentes COTS, también son inexistentes los repositorios que almacenen especificaciones de componentes COTS, y por tanto, en mayor medida, no se puede pensar en procesos de mediación que permitan por un lado a los "fabricantes de componentes COTS" poder anunciar sus productos, y por otro a los "usuarios de componentes COTS" poder buscar los componentes COTS que necesitan.

Nuestra trabajo —presentado en los próximos capítulos— pretende ofrecer una propuesta de solución para estas deficiencias planteadas, y ser consecuentes así con la definición de componente COTS ofrecida anteriormente (ver **definición 4.1**). Veamos a continuación algunas características más acerca de esta clase de componente.

1.4.3. Características de un componente comercial

Por regla general, existe una gran diversidad de parámetros que caracterizan a un componente COTS, pero sin embargo, dos son los más comunes en la literatura de componentes COTS. En primer lugar, un componente COTS suele ser de grano grueso y de naturaleza de "caja negra" sin posibilidad de ser modificado o tener acceso al código fuente. Una de las ventajas de un software comercial es precisamente que se desarrolla con la idea de que va a ser aceptado como es, sin permitir modificaciones. Hay algunos desarrolladores de componentes que permiten la posibilidad de soportar técnicas de personalización que no requieren una modificación del código fuente, por ejemplo mediante el uso de *plug-ins* y *scripts*. Y en segundo lugar, un componente COTS puede ser instalado en distintos lugares y por distintas organizaciones, sin que ninguna de ellas tenga el completo control sobre

la evolución del componente software. Es sólo el vendedor de componentes COTS quien decide su evolución y venta.

Son muy numerosas las ventajas —aunque también lo son los inconvenientes— de utilizar componentes COTS en lugar de componentes de "fabricación propia". Comentemos algunas de ellas, empezando por las ventajas.

Una de las ventajas más claras es el factor económico, relacionado con el coste de desarrollo. Puede ser mucho más barato comprar un producto comercial, donde el coste de desarrollo ha sido amortizado por muchos clientes, que intentar desarrollar una nueva "pieza" software. Por otro lado, el hecho de que un componente COTS haya sido probado y validado por el vendedor y por otros usuarios del componente en el mercado, suele hacer que sea aceptado como un producto mejor diseñado y fiable que los componentes construidos por uno mismo. Otra ventaja es que el uso de un producto comercial permite integrar nuevas tecnologías y nuevos estándares más fácilmente y rápidamente que si se construye por la propia organización.

En cuanto a las desventajas, destacamos principalmente dos, aunque estas derivan en otras más. En primer lugar, y como adelantamos al inicio de esta sección, los desarrolladores que han adquirido un componente comercial no tienen posibilidad de acceso al código fuente para modificar la funcionalidad del componente. Esto significa que en las fases de análisis, diseño, implementación y pruebas, el componente es tratado como un componente de caja negra, y esto puede acarrear ciertos inconvenientes para el desarrollador, como por ejemplo no saber cómo detectar y proceder en caso de fallos; o que el sistema requiera un nivel de seguridad no disponible en el componente, entre otros problemas. Además, los productos comerciales están en continua evolución, incorporando el fabricante nuevas mejoras al producto y ofreciéndoselo a sus clientes (por contrato, licencia o libre distribución). Sin embargo, de cara al cliente desarrollador, reemplazar un componente por uno actualizado puede ser una tarea laboriosa e intensiva: el componente y el sistema deben pasar de nuevo unas pruebas (en el lado cliente).

Se pretende utilizar los componentes COTS como partes ensamblables en la construcción de sistemas, en lugar de considerarlos únicamente como programas finales de usuario. Por este motivo los desarrolladores de componentes COTS se están interesando en disponer de especificaciones estandarizadas para sus productos

En segundo lugar, otra gran desventaja es que, por regla general, los componentes COTS no suelen tener asociados ninguna especificación de sus interfaces, ni de comportamiento, de los protocolos de interacción con otros componentes, de los atributos de calidad de servicio, y otras características que lo identifiquen. En algunos casos, las especificaciones que ofrece el fabricante de componentes COTS puede que no sean siempre correctas, o que sean incompletas, o que no sigan una forma estándar para escribirlas (las especificaciones). Otras veces, aunque el vendedor de componentes COTS proporcione una descripción funcional del componente, puede que ésta no satisfaga las necesidades del integrador, y que necesite conocer más detalles de la especificación del comportamiento y de los requisitos del componente. Además, la falta de una información de especificación puede acarrear ciertos problemas al desarrollador que utiliza el componente COTS, como por ejemplo la imposibilidad de estudiar la compatibilidad, la interoperabilidad o la trazabilidad de los componentes durante el desarrollo del sistema basado en componentes.

En la actualidad son muchos los autores que proclaman la necesidad de un modelo para la especificación de componentes COTS [Wallnau et al., 2002] [Dong et al., 1999] utilizando diferentes notaciones y estrategias. Estas propuestas estudian el tipo de información básica que debe ser capturada en una plantilla de especificación de componente COTS; pero son muy pocas las propuestas existentes que están soportadas por herramientas, y probablemente ninguna de ellas es ampliamente aceptada por la industria para documentar componentes software comerciales.

En el *Capítulo 2* presentaremos un modelo para la documentación de componentes COTS [Iribarne et al., 2001b] [Iribarne et al., 2001c] que puede ayudar a solventar algunos

de los inconvenientes arriba indicados. El modelo de documentación recoge información funcional (interfaces, protocolos y comportamiento), información extra-funcional, información de implementación e implantación, e información de marketing. El modelo de documentación de componentes COTS está soportado por plantillas en XML que están basadas en el esquema COTS-XMLSchema (`http://www.cotstrader.com/COTS-XMLSchema.xsd`) desarrollado a partir del lenguaje *XMLSchema Language* del W3C (`http://www.w3.org`).

1.4.4. Sistemas basados en componentes COTS

Los sistemas de componentes COTS se construyen mediante la integración a gran escala de componentes adquiridos a terceras partes. La naturaleza de la caja negra de estos componentes, la posibilidad de que exista una incompatibilidad con la arquitectura, y su corto ciclo de vida, requiere una aproximación de desarrollo diferente [Basili y Boehm, 2001]. En la tabla 1.7 se muestran algunas de las actividades en el desarrollo de software basado en componentes COTS [Meyers y Oberndorf, 2001]. Estas actividades de desarrollo afectan tanto a la organización encargada de hacer la adquisición como a la organización responsable de llevar a cabo la integración de los componentes COTS.

Actividad	Descripción
Evaluación de Componentes	Cuando se diseña un sistema se debe localizar un conjunto de componentes COTS candidatos y evaluarlos con el propósito de seleccionar de aquellos componentes más adecuados para el sistema. Para un máximo beneficio en el uso de los componentes COTS, la evaluación se debe hacer a la vez que se definen los requisitos y se diseña la arquitectura. La evaluación debe requerir un conjunto de pruebas en fases muy tempranas del ciclo de desarrollo.
Diseño y Codificación	Se basa principalmente en la implementación de "envolventes" (código *wrapper*) y componentes de enlace (código *glue*).
Prueba	Las pruebas individuales y de integración se deben hacer como una caja negra. Como se ha mencionado, las pruebas individuales se deben hacer en las primeras fases del ciclo de desarrollo para llevar a cabo la evaluación de componentes COTS.
Detección de Fallos	Cuando la operatividad del sistema falla, los gestores deben ser capaces de determinar cual ha sido el componente/s que ha/n provocado el fallo.
Actualización de Componentes	Nuevas versiones de componentes COTS suelen aparecer en un corto periodo de tiempo. Los integradores deben poder: (a) evaluar toda nueva versión del componente; (b) determinar si la nueva versión debe ser integrada y cuando debe hacerlo; (c) realizar la instalación, integración y pruebas necesarias.
Gestión de Configuraciones	Los integradores deben llevar a cabo gestiones de configuración sobre varios componentes COTS. El modelo de componentes de un sistema basado en componentes COTS incluye: (a) determinar conjuntos de versiones de componentes compatibles con otros; (b) actualizar las versiones del componente como se requiere; (c) registrar toda versión de componente COTS que ha sido instalada; (d) generar un histórico de los componentes actualizados.

Tabla 1.7: Actividades de los sistemas de componentes COTS [Meyers y Oberndorf, 2001]

La "Evaluación de componentes" conlleva conjuntamente las tareas de búsqueda, selección y evaluación de componentes comerciales. También es conocida como fase de adquisición de componentes comerciales, o simplemente fase de selección [Maiden y Ncube, 1998] [Meyers y Oberndorf, 2001]. En la figura 1.3 (página 12) vimos un cuadro donde se comparaban las distintas etapas de los procesos de desarrollo de los sistemas basados en componentes software (DSBC) y los basados en componentes comerciales (COTS). En dicha figura se puede ver cómo la adquisición de componentes COTS coincide con las etapas de

"Selección de componentes" y "Adaptación" en DSBC; o con las etapas de "Requisitos", "Especificación" y "Aprovisionamiento" del proceso RUP. Además, la etapa de "Diseño y codificación", "Prueba" y "Detección de fallos", coincide con la etapa de "Ensamblaje" en DSBC, y con las de "Ensamblaje" y "Pruebas" en RUP.

La selección de componentes COTS (evaluación, o adquisición en la vista COTS) suele ser una tarea no trivial, donde hay que considerar diferentes aspectos de los componentes comerciales y de la arquitectura de software [Ncube y Maiden, 2000]. En la actualidad representa el principal centro de interés en el área de los sistemas basados en componentes comerciales, y en la literatura podemos encontrar diversos enfoques. Por ejemplo, podemos encontrar enfoques de la selección considerando aspectos de requisitos funcionales y/o extra-funcionales (p.e., [Rosa et al., 2001] [Maiden et al., 1997] [Rolland, 1999]), o aspectos de atributos de calidad (p.e., [Burgués et al., 2002] [Bertoa y Vallecillo, 2002]), o considerando métricas (p.e., [Sedigh-Ali et al., 2001] [Azuma, 2001] [Ochs et al., 2000b]). Estos enfoques basados en requisitos serán tratados de nuevo en la sección 1.5.3.

Otros enfoques son por ejemplo los que consideran aspectos sobre cómo recoger los requisitos de un componente COTS en una arquitectura de software (p.e., [Nuseibeh, 2001] [Monge et al., 2002] [Vigder, 1998]), o enfoques sobre aspectos de evaluación (adaptación o validación) de componentes comerciales (p.e., [Kontio et al., 1996] [Lawlis et al., 2001] [Poston y Sexton, 1992]).

En cuanto a los métodos de selección que consideran varios enfoques, en la literatura podemos encontrar una gran variedad. A continuación destacamos algunos de ellos. Por un lado está el método CAP (*COTS Acquisition Process*) [Ochs et al., 2000a], que se compone de tres partes: inicialización, ejecución y reutilización. La primera parte tiene que ver con procesos de adquisición y estimación de costes. La segunda parte guía en la evaluación de componentes COTS y en la toma de decisiones para la compra de los mejores componentes COTS (aquellos que se adecuan a las necesidades del usuario). Y la tercera parte recopila y almacena toda la información recogida en procesos anteriores para reducir el coste de futuros procesos en adquisición de componentes COTS.

Otro método es el que se ofrece en [Rolland, 1999], que propone una técnica que permite razonar semánticamente durante los procesos de selección y ensamblaje sobre aquellos componentes que cumplen los requisitos del sistema. Estos requisitos son recogidos mediante unos diagramas de transiciones llamados "mapas" que se basan en cuatro modelos: el modelo "As-Is", el modelo "To-Be", el modelo COTS y el modelo de emparejamiento.

Relacionado con los trabajos que soportan el proceso completo para la construcción de sistemas basados en componentes comerciales, destacamos los siguientes. Por un lado está el método K-BACEE [Seacord et al., 2001] que propone unos procesos para la identificación de componentes basada en conocimiento de las reglas de integración del sistema. Este trabajo se desarrolla dentro del *COTS-Based Systems (CBS) Initiative* del *Software Engineering Institute* (SEI).

En segundo lugar encontramos los proyectos CAFE y ESAPS[12], disponibles en el *European Software Institute* (ESI). Aquí destacamos la plataforma Thales [Cherki et al., 2001] para la construcción de sistemas de software basados en partes COTS. Esta propuesta utiliza herramientas Rational para definir la arquitectura de software y diagrama de clases.

Ninguno de los métodos anteriormente comentados sigue un enfoque donde intervenga un servicio de mediación para componentes COTS. Como veremos en la sección 1.7, un servicio de mediación permite albergar especificaciones de componentes y localizar aquellas especificaciones que cumplen con las restricciones de una deseada.

Son varios también los trabajos aplicados a casos reales en el desarrollo de sistemas

[12]Disponibles en `http://www.esi.es/Cafe` y `http://www.esi.es/esaps`

con componentes COTS. Este es el caso por ejemplo de [Dean y Vigder, 1997], que realiza un estudio de los experimentos llevados a cabo por el NRC (*National Research Council*, Canada) para desarrollar un prototipo de un gestor de bases de datos distribuido. El estudio descubre la necesidad de disponer de una metodología para integrar un significativo número de componentes COTS en las fases de desarrollo de sistemas de software.

Otro caso de aplicación real se da en [Addy y Sitaraman, 1999] que demuestra cómo el uso de los métodos formales puede ayudar en el desarrollo de sistemas de software de alto riesgo basados en componentes COTS. El caso real consiste en un controlador para el aterrizaje de aviones en un aeropuerto. Se utiliza un modelo matemático para la formalización del sistema, y RESOLVE/C++ para su implementación.

Por otro lado, a simple vista todo parece indicar que el desarrollo de sistemas basados en componentes COTS está empezando a ser factible debido numerosas razones: (a) en primer lugar debido a la mejora en la calidad y variedad de productos de componentes COTS; (b) también debido a presiones económicas para reducir costes de desarrollo y de mantenimiento del sistema; (c) debido a la consolidación de una tecnología para la integración (ensamblaje) de componentes (la tecnología ORB); (d) debido a una orientación en los últimos años hacia sistemas abiertos y estándares; y por último (e) debido a un paulatino aumento en las prácticas de reutilización de software dentro de las organizaciones.

Sin embargo, hay otros autores que condicionan el desarrollo factible de sistemas basados en componentes comerciales a la existencia previa un mercado de componentes COTS consolidado [Basili y Boehm, 2001] [Voas, 2001] [Wallnau et al., 2002]. El problema actual es que no existen unas estrategias de mercado estandarizadas, y son más bien estrategias particulares a cada fabricante de componentes COTS, o a lo sumo, estrategias de coalición entre dos o más fabricantes para lograr mayor cota de mercado. Incluso hay algunos autores (como [Wallnau et al., 2002]) que predicen que las estrategias de mercado nunca llegarán a ser estandarizadas, y por contra predicen una mayor consolidación de un mercado de componentes COTS multifacetas y de coaliciones.

Propiedades de un Sistema Basado en Componentes COTS

Basadas en las actividades de desarrollo mostradas en la tabla 1.7, un sistema software basado en componentes COTS puede tener las siguientes propiedades deseables:

— *Adaptable*. Las actualizaciones de la configuración de un componente es una tarea frecuente. Puesto que algunos fabricantes de componentes COTS generan nuevas versiones a lo largo de un año, el proceso de reemplazar componentes puede tener lugar varias veces al año para cada componente COTS del sistema. Para reducir este esfuerzo, un sistema debería tener una configuración de componentes adaptable que permita a los componentes una fácil incorporación, eliminación o sean reemplazados.

— *Auditable*. Un sistema es auditable si los integradores y gestores son capaces de ver y monitorizar su comportamiento interno. La auditoría es crítica en tareas de prueba, monitorización y detección de errores del sistema. El software COTS puede o no ofrecer visibilidad de su comportamiento interno. Ya que el código fuente no está disponible, esta visibilidad puede ofrecerse de diferentes formas, por ejemplo a través de interfaces especiales, archivos "log" o estructuras de datos visibles. Además de proporcionar visibilidad a nivel de componente, el sistema y la arquitectura pueden ofrecer visibilidad de aspectos de comportamiento. Por ejemplo, protocolos de comunicación pueden ser monitorizados con rastreadores (*sniffers*), o el sistema operativo puede proporcionar información acerca de los recursos de uso, procesos en curso, etc.

— *Abierto.* Un sistema abierto es aquel que ha sido construido acorde a unos estándares y tecnologías abiertas y disponibles. La apertura de un sistema permite que éste sea extendido e integrado.

— *Seguro.* La seguridad es una propiedad que debe ser considerada en un sistema a nivel arquitectónico. Los componentes individuales pueden o no tener seguridad, pero es la arquitectura del sistema de software la que debe implementar las políticas de seguridad apropiadas.

1.5. Requisitos y componentes

Las primeras fases del ciclo de vida en la construcción de un sistema software comienzan con actividades de ingeniería para la identificación y el análisis de requisitos. Está comprobado que estas fases iniciales del desarrollo son las más críticas. De hecho, se sabe que aproximadamente el 60 % de los errores detectados en las fases de diseño están directamente provocados por errores en las actividades de requisitos, aumentando con esto los plazos de entrega y los costes preliminares planificados [Robertson y Robertson, 1999].

1.5.1. Ingeniería de requisitos tradicional

La "ingeniería de requisitos" se refiere al conjunto de tareas que tratan los requisitos de un sistema software. Desde una perspectiva tradicional, un requisito define una condición de necesidad, y consecuentemente, el diseño se hace como respuesta al requisito y para cumplir esta condición.

Como ha sucedido para otros términos vistos anteriormente en esta memoria, en la literatura podemos encontrar también numerosas definiciones para el término "ingeniería de requisitos". Veamos algunas de ellas. En primer lugar está la perspectiva IEEE ofrecida en el glosario de términos de estándares software [IEEE, 1990], y que dice los siguiente:

Definición 1.7 (Ingeniería de requisitos, [IEEE, 1990]) *La ingeniería de requisitos es el proceso de estudiar las necesidades de usuario para llegar a una definición de sistema, hardware o requisitos software... donde un requisito es definido como (1) una condición o capacidad que necesita un usuario para resolver un problema o para cumplir un objetivo; (2) una condición o capacidad que debe tener un sistema o componente para cumplir un contrato, estándar, especificación o cualquier otro documento impuesto formalmente; (3) una representación documentada de una condición o capacidad como en (1) o (2).*

Sin embargo, esta definición es demasiado general y puede llevar a interpretaciones particulares o erróneas a cada organización. Una definición algo más específica del término "ingeniería de requisitos" es la que se ofrecen Jordon y Davis, que dice lo siguiente:

Definición 1.8 (Ingeniería de requisitos, [Jordon y Davis, 1991]) *La ingeniería de requisitos es el uso sistemático de principios, técnicas, lenguajes y herramientas que hacen efectivos el análisis, la documentación, la evolución de las necesidades de usuario y las especificaciones del comportamiento externo para cumplir aquellas necesidades de usuario.*

Como vemos, esta definición centra su atención en tres áreas: el análisis, que se refiere a la identificación y recogida de requisitos; la documentación, que se refiere a la especificación y almacenamiento de los requisitos; y la evolución, que se refiere a que los requisitos sufren continuos cambios en las fases del ciclo de vida.

Otros autores dividen la ingeniería de requisitos en dos dominios [Leffingwell, 2001]: el dominio del problema y el dominio de la solución. Por un lado está el "dominio del problema", donde se analiza el sistema como un problema que hay que resolver y sus necesidades; y por otro lado está el "dominio de la solución", donde se estudia cómo se van a abordar cada una de las partes del problema identificadas en el dominio del problema.

En cualquier caso, lo que sí parece estar relativamente claro en la literatura de requisitos es en las categorías en las que se divide un requisito software, y que son: (a) los requisitos funcionales, (b) los requisitos extra-funcionales, y (c) las restricciones de diseño. Las dos primeras categorías fueron tratadas en la sección 1.3, cuando describimos la especificación de un componente. Allí tratamos estos dos requisitos como información funcional e información extra-funcional. Las restricciones de diseño imponen limitaciones sobre el diseño del sistema o el proceso utilizado para construirlo.

Trazabilidad

Además de identificar y definir los requisitos de un sistema, también es importante considerar una característica que relacione entre sí estos requisitos, la "trazabilidad" (*traceability*).

Un factor importante en calidad de software es la habilidad de poder ver y entender cómo se "plasma" la relación de un mismo requisito del sistema en diferentes etapas del ciclo de vida (especificación, diseño, implementación y prueba). Por esta razón, una de las claves importantes en los procesos de control de calidad de software, es la posibilidad de detectar relaciones y de saber tratar y entender esas relaciones cuando ocurre un cambio. Para ello, cada requisito debe ser bien definido, y con un código de requisito fácilmente identificable. En la figura 1.10 mostramos un ejemplo de plantilla usada para capturar las características de un requisito [Robertson y Robertson, 1999].

Requisito nº: **125** Tipo de requisito: **12** Evento/caso de uso nº: **7**

Descripción: .....

.....

Fundamento: .....

.....

Fuente: **Sección de Sistemas**

Criterios establecidos: .....

.....

Grado satisfacción del cliente: **4** Dependencias: **ninguna**

Grado de disconformidad del cliente: ***3*** Conflictos: **ninguno**

Materiales que soporta: .....

Historia: .....

.....

Figura 1.10: Una plantilla de especificación de requisitos [Robertson y Robertson, 1999]

Por lo tanto, la trazabilidad es un término que influye en todo el proceso de desarrollo, y que permite ver en todo momento cómo se está considerando (o ha sido considerado) un requisito en todas las etapas del ciclo de vida, esto es, la trazabilidad permite hacer un seguimiento del requisito desde el análisis hasta la implementación (y viceversa). Por ejemplo, dado el requisito número 125, visto en la figura 1.10, la trazabilidad nos permitiría afirmaciones como las siguientes (a modo de ejemplo): "...en fase de diseño el requisito 125 está recogido en la clase A" o "...la implementación del componente C3 cubre los requisitos 125, 134 y 203", entre otras.

La trazabilidad es una característica de relación espacio-temporal, pues afecta tanto a la relación existente entre categorías o tipos de requisitos (espacio), como a la relación entre los requisitos y los elementos (de diseño o de implementación) generados durante el desarrollo (tiempo)

En otros muchos casos la trazabilidad no solo afecta en el tiempo (durante el desarrollo del producto), sino que también se refiere a la habilidad de relacionar requisitos entre categorías distintas, como funcional, extra-funcional y restricciones (esto se podría corresponder por ejemplo con el campo "tipo de requisito" de la plantilla mostrada en la figura 1.10). Por ejemplo, podría ser necesario establecer relaciones de correspondencia entre un requisito extra-funcional con uno o más requisitos funcionales, y viceversa.

Finalmente, podemos considerar (o llevar control de) diferentes tipos de trazabilidad, dependiendo de su utilidad. Así por ejemplo, en [Gao et al., 2000] se consideran hasta cinco tipos de trazos (*trace*) distintos: (a) "trazado operacional", donde se lleva registro de todas las interacciones de las operaciones de los componentes, como por ejemplo todas las invocaciones que se han realizando; (b) "trazado de ejecución", donde se lleva registro por ejemplo de cómo van variando los datos, los mensajes, o las referencias a las funciones entre los componentes; (c) "trazado de estado", donde se sigue el rastro del estado de un componente[13]; (d) "trazado de eventos", donde se van almacenando todas las secuencias de eventos que han tenido lugar en un componente; y por último (e) "trazado de errores", donde se lleva registro de todos los mensajes de error producidos por el componente.

1.5.2. Prácticas de ingeniería de requisitos

Métodos

Como prácticas de ingeniería de requisitos, tradicionalmente se utilizan diversos métodos y herramientas. En el caso de los métodos podemos encontrar diferentes categorías, como los métodos de elicitación de requisitos, métodos de análisis de requisitos, y métodos de validación de requisitos. En esta línea destacamos el método REM (*Requirement Engineering Method*) [Durán, 2000] que describe un modelo iterativo de procesos de elicitación, análisis y validación de requisitos, basado en UML y unas plantillas y patrones de requisitos.

Como métodos de análisis de requisitos, en [Thayer y Dorfman, 1999] se definen hasta cuatro categorías diferentes: (a) métodos orientados a proceso, (b) métodos orientados a datos, (c) métodos orientados a control, (d) métodos orientados a objeto.

Los métodos orientados a proceso tienen en cuenta la forma en la que el sistema transforma las entradas y salidas, haciendo menor énfasis en los datos y en el control. El análisis estructurado clásico [Svodoba, 1990] es un caso de esta categoría, como lo son por ejemplo las técnicas de análisis y de diseño estructurado [Ross, 1977], y los métodos formales como VDM [Bjoerner, 1987] y Z [Spivey, 1992].

Los métodos orientados a datos destacan el estado del sistema como una estructura de datos. A diferencia del análisis estructurado y las técnicas de análisis y diseño estructurado, que consideran los datos a nivel secundario, la modelización entidad-relación [Reilly, 1990] y JSD [Cameron, 1986] están principalmente orientados a datos.

Los métodos orientados a control hacen hincapié en la sincronización, bloqueos, exclusión, concurrencia y procesos de activación y desactivación. Las técnicas de análisis y diseño de datos, y extensiones de tiempo real (*real-time*) para análisis estructurado [Ward y Mellor, 1985], son también orientados a control.

Finalmente, los métodos orientados a objetos basan el análisis de requisitos en clases de objetos y sus interacciones. En [Bailin, 1994] se examinan los fundamentos del análisis orientado a objetos y varios métodos de análisis de requisitos orientados a objetos. En esta categoria también incluimos las técnicas para describir "casos de uso" [Jacobson et al., 1992].

Herramientas

En cuanto a las herramientas utilizadas en ingeniería de requisitos, podemos encontrar una gran variedad, clasificadas básicamente en cinco categorías: (a) herramientas de edición

[13] En los componentes de caja negra, como los componentes COTS, es muy importante llevar el rastro de los datos que entran y salen del componente si queremos hacer trazabilidad (al menos externa al componente, debido a la imposibilidad de hacer trazabilidad interna, por su característica de caja negra).

gráfica, (b) herramientas de trazabilidad, (c) herramientas para modelar comportamiento, (d) herramientas de bases de datos y procesamiento de textos[14], y (e) herramientas híbridas (como combinación de las anteriores).

Dos de las primeras herramientas de requisitos fueron SREM [Alford, 1985] y PSL/PSA [Sayani, 1990]. Estas herramientas soportan funciones de análisis de requisitos, y están basadas en representación gráfica y modelización de comportamiento. Por aquellas fechas aparecen las primeras herramientas que tratan la trazabilidad [Dorfman y Flynn, 1984].

Hoy día, la tendencia es a utilizar notación UML (basadas en [Booch et al., 1999]) como herramienta de edición gráfica para modelar sistemas de software (y no solo para objetos), ya que integra múltiples técnicas que pueden ser utilizadas para el análisis de requisitos, como los diagramas de secuencias, diagramas de casos de uso, y diagramas de estados, entre otros. Un ejemplo de ello son los trabajos [Franch, 1998] o [Botella et al., 2001], que utilizan UML como herramienta de edición gráfica para la definición de comportamiento extra-funcional de componentes software, como atributos de calidad.

Por último, destacamos el trabajo [Grünbacher y Parets-Llorca, 2001] como ejemplo de una técnica híbrida. Esta técnica, que utiliza UML como herramienta de edición gráfica, es aplicable para la elicitación y la evolución de requisitos software, y estudia aspectos para tratar la trazabilidad de los requisitos desde la implementación hacia la arquitectura.

1.5.3. Ingeniería de requisitos y componentes COTS

Inicialmente, para la construcción de un sistema software basado en componentes los ingenieros seguían el enfoque tradicional descendente basado en el modelo de desarrollo de software en cascada clásico. En este enfoque existe primero una fase de análisis, donde se identifican y definen los requisitos de los componentes software; luego se diseñan los componentes y la arquitectura de software del sistema; se lleva a cabo labores de implementación y pruebas por separado de los componentes, y ensamblaje de estos (creando componentes ORB y envolventes) y pruebas finales. Sin embargo, este modelo es totalmente secuencial, desde el análisis de requisitos hasta llegar a obtener el producto final.

Con el tiempo se ha visto que el modelo de desarrollo de software en cascada no era el más idóneo para la construcción de sistemas basados en componentes. Los procesos de reutilización de software hacen que los requisitos de algunos componentes puedan ser satisfechos directamente, sin necesidad de tener que llegar a etapas de implementación (para esos componentes). En el mejor de los casos, los requisitos de los componentes involucrados en los procesos de reutilización, estaban completamente controlados, ya que las técnicas de reutilización de componentes se aplicaban sobre repositorios, catálogos o librerías de componentes que la propia organización había desarrollado en proyectos anteriores.

El problema aparece cuando, a falta de estos repositorios de componentes internos, la organización decide incluir componentes desarrollados fuera de ésta en los procesos de reutilización, como es el caso de los componentes COTS. En este caso, los procesos de adquisición de componentes COTS involucran tareas de búsqueda, selección y evaluación de esta clase de componente. Como resultado, se podría dar el caso de que, por desconocimiento, algunos requisitos que deberían haber sido impuestos para un componente podrían omitirse en su planteamiento inicial (cuando se hizo la definición de los requisitos del componente).

La adquisición de componentes COTS es un caso particular del proceso de reutilización de software

El modelo de desarrollo en espiral [Boehm, 1988] es el más utilizado en la construcción de sistemas basados en componentes COTS [Boehm, 2000] [Maiden y Ncube, 1998] [Carney, 1999] [Saiedian y Dale, 2000]. Este tipo de desarrollo asume que no todos los requisitos de un componente puedan ser identificados en las primeras fases del análisis de

[14] Aunque no fueron diseñados para ingeniería de requisitos, sí se utilizan en aplicaciones de requisitos.

requisitos, pudiendo aparecer nuevos de ellos en cualquier otra fase del ciclo de vida, que tengan que ser también contemplados; o incluso puede que los requisitos actuales tengan que ser refinados como resultado de la evaluación de los componentes adquiridos.

Técnicas de evaluación de componentes COTS

Existen varias técnicas que pueden ser utilizadas para evaluar productos de componentes COTS, todas ellas llevadas a cabo de "forma manual" por el equipo de personas encargado de realizar las tareas de evaluación. Algunos ejemplos de estas técnicas de evaluacion son [Meyers y Oberndorf, 2001]:

- Análisis de la literatura existente, referencias de otros usuarios y del vendedor.
- Análisis de la conexión diseño-implementación (conocido como *Gap analysis*), que consiste en determinar qué requisitos son, o no, satisfechos por el producto (siendo evaluado) y las características del producto que no han sido recogidas como requisitos.
- Mediante demostraciones facilitadas por el vendedor.
- Mediante problemas de modelo, que son pequeños experimentos que se centran en cuestiones específicas de diseño y de comportamiento del producto.
- Prototipos de otros subsistemas que utilizan componentes COTS.
- Utilizando técnicas de evaluación de otros usuarios, como por ejemplo la técnica RCPEP de [Lawlis et al., 2001], que consiste en un proceso de evaluación de productos de componentes COTS guiado por requisitos.

Métodos de ingeniería de requisitos de componentes COTS

Hoy día también existen diversos métodos que se están aplicando en ingeniería de requisitos para componentes COTS. Ejemplos de estos métodos son, PORE (*Procurement-Oriented Requirement Engineering*) [Maiden y Ncube, 1998] y ACRE (*Acquisition of Requirements*) [Maiden et al., 1997]. El primero utiliza un método basado en plantilla a tres niveles para la recogida de requisitos[15] y el segundo proporciona un marco de trabajo para la selección y utilización de diversas técnicas de adquisición de requisitos.

Otro método conocido es OTSO (*Off-The-Shelf Option*) [Kontio et al., 1996], desarrollado en procesos de selección de componentes reutilizables, y que tiene en cuenta requisitos funcionales específicos de la aplicación, aspectos de diseño y restricciones de la arquitectura.

Otro método es el que utiliza el lenguaje NoFun [Franch, 1998], citado anteriormente. Como adelantamos, este lenguaje utiliza UML para la definición de comportamiento extra-funcional de componentes software como atributos de calidad. Además, en una reciente revisión del lenguaje [Botella et al., 2001], NoFun permite definir diferentes tipos de componentes y relaciones entre componentes y subcomponentes, y adopta el estándar [ISO/IEC-9126, 1991] para tratar atributos de calidad.

También están los métodos relacionados con atributos de calidad para la selección de componentes COTS. En esta línea están conjuntamente los trabajos [Botella et al., 2002] y [Franch y Carvallo, 2002] donde se proponen unos modelos de calidad y de certificación de componentes COTS basados en el modelo de calidad de ISO [ISO/IEC-9126, 1991] y en el lenguaje NoFun como notación estructurada para la formalización de estos modelos. Aquí también destacamos el trabajo [Bertoa y Vallecillo, 2002] que propone un modelo de calidad para componentes COTS inspirado en el modelo de ISO [ISO/IEC-9126, 1991] y que clasifica las características de calidad de un producto software.

Otros trabajos se centran en métodos para la selección de componentes COTS a partir de requisitos extra-funcionales. En esta línea está el trabajo [Rosa et al., 2001] que realiza

[15] Estas plantillas están disponibles en `http://www.soi.city.ac.uk/pore/welcome.html`

un estudio formal en Z del proceso de desarrollo de software basado en componentes COTS considerando requisitos extra-funcionales. El estudio se centra en las fases de selección de componentes desde librerías COTS y ofrece una solución ante la dificultad de tratar los requisitos extra-funcionales en los procesos de selección.

Para finalizar, están los métodos para la construcción de componentes COTS considerando atributos de calidad (funcionales y extra-funcionales) y métricas para la evaluación y selección de componentes COTS. En esta línea destacamos el modelo COCOTS [COCOTS, 1999], basado en el modelo de construcción de software [COCOMO-II, 1999].

1.6. Arquitecturas de software

Las arquitecturas son fundamentales en cualquier sistema, especialmente para los sistemas abiertos. Como en un modelo de referencia, una arquitectura permite centrarse en las características y funciones de un sistema, pero con la diferencia de que además también permite especificar algunas características de implementación del mismo.

Como hemos hecho para otros apartados, también aquí vamos a ofrecer algunas definiciones concretas para el término "arquitectura de software". Debido a la extensa y variada gama de trabajos en el campo de las arquitecturas, sólo nos centraremos en dos de las definiciones más usadas últimamente: la ofrecida por el estándar 1471 de IEEE [Maier et al., 2001], y la ofrecida por Bass, Clements y Kazman [Bass et al., 1998].

El estándar 1471 de IEEE identifica ciertas prácticas para establecer un marco de trabajo (*framework*) y un vocabulario unificado para conceptos relacionados con las arquitecturas de software. El estándar define una arquitectura de software como:

Definición 1.9 (arquitectura de software IEEE 1471, [Maier et al., 2001]) *Parte fundamental de un sistema expresado por sus componentes, sus relaciones con otros componentes y otros entornos, y los principios que guían en su diseño y evolución.*

Otra definición interesante de arquitectura de software es la que ofrecen Bass, Clements y Kazman en [Bass et al., 1998]. Esta definición es ampliamente adoptada por otros autores en arquitecturas de software [Garlan, 2001] [Hofmeister et al., 1999] [Rumpe et al., 1999]. La definición de Len Bass dice lo siguiente:

Definición 1.10 (arquitectura de software, [Bass et al., 1998]) *Arquitectura de software de un programa o sistema de computación es la estructura o estructuras del sistema, que están compuestas de componentes software, de las propiedades visibles de esos componentes, y las relaciones entre ellos.*

Hay un par de aspectos a considerar para esta definición. En primer lugar, el término "componente" se refiere a un elemento software simple o a una colección de otros elementos software. En segundo lugar, las propiedades visibles se refieren a los requisitos de componente (funcionales y extra-funcionales) identificados en la fase de análisis de requisitos.

1.6.1. Características de las arquitecturas de software

Las arquitecturas de software generalmente juegan el papel de "pasarelas" entre los requisitos y la implementación. Mediante una descripción abstracta de un sistema, la arquitectura expone ciertas propiedades, mientras oculta otras.

Las arquitecturas de software pueden jugar un importante papel en al menos seis aspectos del desarrollo de software [Garlan, 2000]:

a) *Comprensión del sistema.* Una arquitectura de software facilita la comprensión de un sistema, al poder representarlo con un alto nivel de abstracción, y donde aspectos de diseño del sistema pueden ser fácilmente comprendidos a alto nivel.

b) *Reutilización.* Las descripciones arquitectónicas soportan reutilización de múltiples formas, generalmente de componentes y marcos de trabajo (*framework*). Ejemplos de estos son los estilos arquitectónicos y los patrones de diseño arquitectónicos.

c) *Construcción.* Una descripción arquitectónica permite tener una visión parcial del sistema que hay que construir, describiendo sus componentes y las dependencias entre ellos.

d) *Evolución.* Una arquitectura permite separar lo que concierne a la parte funcional de un componente de las formas en las que este componente puede ser conectado a otros componentes. Esta separación facilita que luego se puedan hacer cambios en la arquitectura por aspectos de interoperabilidad, prototipado y reutilización.

e) *Análisis.* Una arquitectura de software es una buena oportunidad para hacer de nuevo, prácticas de análisis y refinar los requisitos identificados en fases de análisis de requisitos. Algunas prácticas de análisis que se pueden aplicar a este nivel son, por ejemplo: comprobaciones de consistencia [Luckham et al., 1995], análisis de dependencias [Stafford et al., 1998] o comprobaciones para ver si se cumplen con las restricciones impuestas en las partes de la arquitectura [Abowd et al., 1993].

f) *Decisión.* Una arquitectura permite desvelar ciertos detalles que pueden decidir las estrategias de implementación a seguir, o modificar o incluir nuevos requisitos.

En una arquitectura de software se describen los detalles de diseño de una colección de componentes y sus interconexiones, que conforman una vista abstracta a alto nivel del sistema que se está diseñando, y donde se consideran los requisitos identificados en la fase de análisis de requisitos del sistema.

Actualmente, en la comunidad de arquitecturas de software existe una gran variedad de elementos arquitectónicos que simplifican las tareas de diseño en la construcción de una arquitectura. Estos elementos arquitectónicos se conocen con el nombre de "estilos arquitectónicos". Un estilo arquitectónico está compuesto por un conjunto de estilos de componente a nivel arquitectónico y por unas descripciones de "patrones" de interacción entre ellos [Shaw y Garlan, 1996] [Bass et al., 1998]. Estos tipos de componente son utilizados para modelar las interacciones que tienen lugar en una infraestructura de componentes. De igual forma que los patrones de diseño orientados a objetos de [Gamma et al., 1995] ayudan a los desarrolladores a diseñar sus clases, los estilos arquitectónicos sirven de ayuda en las tareas de diseño de componentes en una arquitectura de software.

Tradicionalmente una arquitectura de software se ha centrado en la descripción y análisis de estructuras "estáticas". Sin embargo, en sistemas complejos (abiertos, distribuidos, adaptativos y evolutivos), donde intervienen por ejemplo componentes de grano grueso, como los componentes comerciales (véase página 7), y en los que la estructura evoluciona a lo largo del tiempo, el sistema podría necesitar de patrones arquitectónicos "dinámicos", como en [Cuesta, 2002], donde se propone un arquitectura de software dinámica que utiliza un enfoque reflexivo[16] para permitir la reconfiguración automática de la arquitectura como respuesta a la evolución del sistema.

[16]El concepto de "reflexión" se define como la capacidad de un sistema computacional de razonar y actuar sobre sí mismo, proporcionando una forma controlada de auto-modificación [Cuesta, 2002].

1.6.2. Vistas para una arquitectura de software

Una arquitectura (software o hardware) puede tener diferentes vistas, cada una de ellas con diferentes interpretaciones del sistema y métodos para su desarrollo [Bass et al., 1998].

Son muchos los autores que coinciden en afirmar que una arquitectura de software puede tener diferentes vistas (por ejemplo [Bass et al., 1998] [D'Souza y Wills, 1999] [Garlan, 2001] [Hofmeister et al., 1999] [Medvidovic et al., 2002] [Shaw y Garlan, 1996], entre otros muchos más); aunque cada una de ellas referidas también de diferente forma. Veamos algunos ejemplos.

Vistas AS de D'Souza y Wills

D'Souza y Wills [D'Souza y Wills, 1999] utilizan nueve tipos de vistas de arquitectura, y que son: (1) vista de dominio (tipos, atributos, asociaciones), (2) vista lógica (componentes, conectores), (3) vista de proceso (procesos, hilos, componentes, sincronización), (4) vista física (unidades hardware, red, topología), (5) vista de distribución (componentes software, procesos, hardware), (6) vista de llamadas (métodos, clases, objetos, programas, procedimientos), (7) vista de usos (paquetes —importa, usa, necesita), (8) vista de flujo de datos (acciones), y (9) vista de módulos (elementos de diseño).

Aunque estos nueve tipos de vistas se definen para una arquitectura de un sistema (donde se consideran aspectos software y hardware), en general, casi todas ellas pueden ser utilizadas para en la definición de una arquitectura de software, siendo la "vista lógica" la más usual, donde se definen los componentes de la arquitectura (del sistema) y sus interconexiones por medio de conectores.

Vistas AS de Len Bass

Otra separación en vistas de una arquitectura de software es la que hace Len Bass en [Bass, 2001], donde las contempla en cuatro categorías: (1) la "vista lógica", (2) la "vista de concurrencia", (3) la "vista de implementación", y (4) la "vista de implantación".

En la vista lógica el diseñador describe las responsabilidades y las interfaces de los elementos de diseño (p.e., los componentes) que intervienen en la arquitectura. Las responsabilidades del elemento de diseño define sus roles dentro del sistema. Estas incluyen aspectos funcionales y aspectos de calidad (p.e., que la implementación de un elemento de diseño tenga un tiempo de ejecución inferior a 50 milisegundos).

La vista de concurrencia describe el sistema como un conjunto de usuarios, recursos y actividades en paralelo. En esta vista se contemplan elementos como procesos, hilos, o aspectos de sincronización. La vista de implementación recoge detalles concretos de implementación, como el lenguaje, pseudo-código librerías.

Por último, la vista de implantación representa (a nivel abstracto) cómo quedaría físicamente la estructura del sistema. En este caso la vista considera los componentes como unidades de implantación (preparados para ser instalados), y describe cómo deben ser instalados y sus conexiones a bajo nivel, considerando detalles de la plataforma, como los sistemas operativos o los procesadores[17].

Otras vistas

En la literatura existen otras distinciones de vistas de una arquitectura de software, como por ejemplo: (1) la "vista conceptual" [Shaw y Garlan, 1996], (2) la "vista modular" [Prieto-Diaz y Neighbors, 1986], (3) la "vista orientada a código" [Kruchten, 1995], (4) la "vista orientada a ejecución" [Purtilo, 1994].

La vista conceptual describe la arquitectura en términos de elementos de dominio, diseñando las características funcionales del sistema. La vista modular describe cómo se organiza el software en módulos (componentes) y qué dependencias existen entre ellos. La vista orientada a código describe cómo los módulos (componentes) definidos en la vista modular pueden ser representados como archivos y la estructura de árbol de directorios donde se organizan estos archivos (componentes). Finalmente, la vista orientada a ejecución

[17] En todo momento estamos asumiendo el uso de sistemas a gran escala, distribuidos y abiertos.

describe el estado del sistema en tiempo de ejecución y en particular explora aspectos dinámicos del sistema. Estas vistas son adecuadas para documentar y analizar propiedades de ejecución, como el rendimiento, la fiabilidad y la seguridad del sistema.

1.6.3. Lenguajes para la definición de arquitecturas de software

Como hemos adelantado, en una arquitectura de software se describen los detalles de diseño de una colección de componentes y sus interconexiones, que conforman una vista abstracta a alto nivel del sistema que se está diseñando, y donde se consideran los requisitos identificados en la fase de "análisis de requisitos" del sistema.

Las interconexiones entre los componentes permiten definir aspectos de interoperabilidad y analizar detalles de compatibilidad entre estos, y suelen ser diseñados como componentes independientes que controlan aspectos de interconexión, como los protocolos de interconexión, que establecen el orden en el que se establecen las llamadas a las operaciones entre dos componentes, y la compatibilidad sintáctica y de comportamiento en las llamadas controladas por la interconexión. A estas interconexiones, o componentes de interconexión, se les denominan "conectores".

Para describir una arquitectura de software se utiliza un lenguaje para la descripción de arquitecturas (*Architecture Description Language*, ADL). En general, un LDA debería ofrecer las siguientes características: (1) una colección de elementos que permita modelar las partes de la arquitectura, como componentes, conectores o puertos, entre otros; (2) métodos y herramientas que faciliten la construcción de la arquitectura, como compiladores, herramientas con notación gráfica para "dibujar" los elementos arquitectónicos (componentes, puertos, conectores, etc.); y (3) que soporte los aspectos de comprensión, reutilización, construcción, evolución, análisis y decisión, tratados anteriormente en la sección 1.6.1 (página 36).

No obstante, tradicionalmente un LDA ha sido identificado como un lenguaje que ofrece una colección de elementos para modelar la arquitectura de software siguiendo el modelo "Componente-Puerto-Conector" de [D'Souza y Wills, 1999]:

— *Componentes*. Representan los elementos computacionales de un sistema, y pueden tener múltiples interfaces definidas en los puertos.

— *Puertos*. Representan la forma de acceso al componente. Los puertos definen las interfaces que el componente proporciona y las interfaces que éste requiere de otros componentes para funcionar.

— *Conectores*. Son las interconexiones entre componentes, y se realizan por medio de los puertos. Un conector queda definido como un componente independiente con tareas exclusivas de interconexión entre dos componentes.

Tal y como podemos comprobar en la tabla 1.8, hoy día existe una gran variedad de LDAs en la literatura, y difieren o asemejan entre ellos por poseer o carecer algunas cualidades. Por ejemplo, en las columnas de la 3 a la 8 se recogen algunas de estas cualidades (o propiedades) deseables en un LDA [Medvidovic y Taylor, 2000]. Estas columnas (propiedades) se interpretan en la tabla de la siguiente forma: (columna 3) el tipo de notación utilizada para escribir la arquitectura (gráfica o textual); (columna 4) lenguajes que soporta; (columna 5) si soporta trazabilidad; (columna 6) si contempla la evolución del sistema; (columna 7) si permite reconfiguración de la arquitectura; (columna 8) si soporta la definición de propiedades extra-funcionales.

	Ref.	G/T	Lenguajes	Trazab.	Evol.	Dinam.	Extra Func.
ACME	[Garlan et al., 2000]	T	propio	No	Si	No	No
AESOP	[Garlan et al., 1994]	G	C++	No	No	No	No
C2	[Medvidovic et al., 1996]	G	C++, Java y Ada	No	Si	Si	No
DARWIN	[Magee y Kramer, 1996]	G	C++	Si	No	Si	No
LEDA	[Canal, 2000]	T	Java	No	Si	Si	No
METAH	[Binns et al., 1995]	G	Ada	Si	No	No	Si
RAPIDE	[Luckham et al., 1995]	G	VHDL, C++ y Ada	Si	No	Si	Si
SADL	[Moriconi et al., 1995]	T	propio	Si	No	No	No
UNICON	[Shaw et al., 1995]	G	propio	Si	No	No	No
WRIGHT	[Allen y Garlan, 1997]	T	propio	No	Si	No	No

Tabla 1.8: Cuadro comparativo para algunos LDAs conocidos

En recientes trabajos en el área de las arquitecturas de software hemos observado una tendencia a utilizar notación UML para la descripción de arquitecturas de software, como por ejemplo en [Garlan et al., 2001] [Hofmeister et al., 1999] [Medvidovic et al., 2002]. Otros ejemplos son el trabajo [Monge et al., 2002], que analiza una propuesta para la descripción en UML de arquitecturas de software basadas en componentes COTS; o el trabajo [Franch y Botella, 1998], que analiza cómo incluir, mediante notación UML, los tipos de requisitos extra-funcionales de NOFUN (véase página 19) dentro una arquitectura de software. También es importante destacar en este sentido la propuesta "Componentes UML" [Cheesman y Daniels, 2001], donde se habla de arquitecturas de componentes modeladas en UML. En la siguiente sección cubriremos algunos detalles de esta propuesta.

Tradicionalmente, UML no ha sido considerado como un LDA por la falta de una notación suficiente para describir elementos arquitectónicos, como los conectores, protocolos o propiedades. Los recientes trabajos recurren a las extensiones de UML, como las restricciones, valores etiquetados, estereotipos y notas, para suplir esta carencia arquitectónica.

Sin embargo, el uso de diagramas de clases UML para modelar la descripción de una arquitectura de software puede dar lugar a representaciones gráficas demasiado extensas, ya que cada elemento arquitectónico (componentes, conectores, puertos y protocolos) debe ser representado como una clase estereotipada (p.e., `<<componente>>`, `<<conector>>`). Además, los sistemas basados en componentes COTS suelen ser sistemas a gran escala, compuestos por un gran número de componentes (comerciales y no comerciales) y con múltiples conexiones. Por tanto, esto multiplica el número de clases posibles en una definición UML de una arquitectura de software.

En nuestro caso, para modelar la descripción de una arquitectura de software hemos adoptado la notación UML-RT de Bran Selic [Selic, 1999], y que analizaremos brevemente en la siguiente sección. Las razones que han hecho inclinarnos por UML-RT como LDA han sido principalmente tres:

(1) En primer lugar, por la tendencia a utilizar UML para describir arquitecturas, ya que UML-RT también es UML y por tanto admite todos las representaciones tradicionales (diagramas de clases, diagramas de estados o diagramas de secuencias, entre otros).

(2) En segundo lugar, UML-RT además adopta la notación original de ROOM (*Real-time Object Oriented Modelling*), también desarrollada por Bran Selic en [Selic et al., 1994]. ROOM (y por extensión UML-RT) utiliza un conjunto reducido de notaciones gráficas que cubren todas las necesidades para hacer una representación visual de una arquitectura de software en poco tiempo.

(3) En tercer y último lugar, como vimos en la sección 1.5.3, los sistemas basados en componentes COTS suelen requerir un prototipado rápido de la arquitectura de software para permitir continuos refinamientos de la misma, impuesto esto por el modelo en espiral que caracteriza el desarrollo de sistemas basados en componentes COTS. UML-RT permite hacer un prototipado rápido de una arquitectura de software.

Como hemos adelantado antes, a continuación vamos a ver algunos detalles de la propuesta "Componentes UML" por su relación con la modelización de arquitecturas de componentes en UML, y seguidamente veremos algunas características de UML-RT como LDA seleccionado en nuestra propuesta (justificado anteriormente).

1.6.4. Componentes UML

Los Componentes UML son una propuesta original de John Cheesman y John Daniels [Cheesman y Daniels, 2001] para modelar arquitecturas de componentes y especificar componentes software utilizando extensiones de UML con estereotipos, y diseño por contratos utilizando OCL (*Object Constraint Language*) [Warmer y Kleppe, 1998].

Como muestra la figura 1.11, los Componentes UML están inspirados en trabajos previos en los cuales los propios autores han participado. Los trabajos directamente relacionados son: UML [Rumbaugh et al., 1999], donde John Cheesman colabora con OIM (*Open Information Model*) [OIM, 1995]; los procesos de desarrollo de software RUP (*Rational Unified Process*) [Jacobson et al., 1999] y Catalysis [D'Souza y Wills, 1999]; y el método Advisor para el desarrollo basado en componentes (inspirado en Catalysis) [Dodd, 2000]. Por otro lado, Catalysis se inspira en el método Syntropy [Cook y Daniels, 1994], un trabajo original de John Daniels que ha servido de base para desarrollar OCL y OIM.

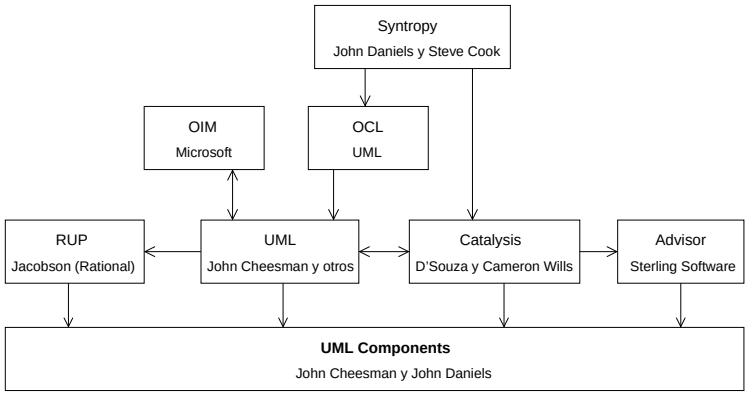

Figura 1.11: Trabajos base de los Componentes UML [Cheesman y Daniels, 2001]

En Componentes UML, una arquitectura de componentes es un conjunto de componentes software a nivel de aplicación, con sus (1) relaciones estructurales y (2) sus dependencias de comportamiento. Una arquitectura de componentes puede ser utilizada para modelar una aplicación de software (a partir de componentes) o para modelar un sistema software como un conjunto de aplicaciones (consideradas estas como componentes y subcomponentes). Las relaciones estructurales (1) significan asociaciones y relaciones de herencia entre especificaciones de componente e interfaces de componente, y relaciones de composición entre componentes. Las dependencias de comportamiento (2) son relaciones de dependencia: (a) entre componentes, (b) entre componentes e interfaces, (c) entre interfaces, y (d) entre subcomponentes y componentes, (como se muestra en la figura 1.12.A)

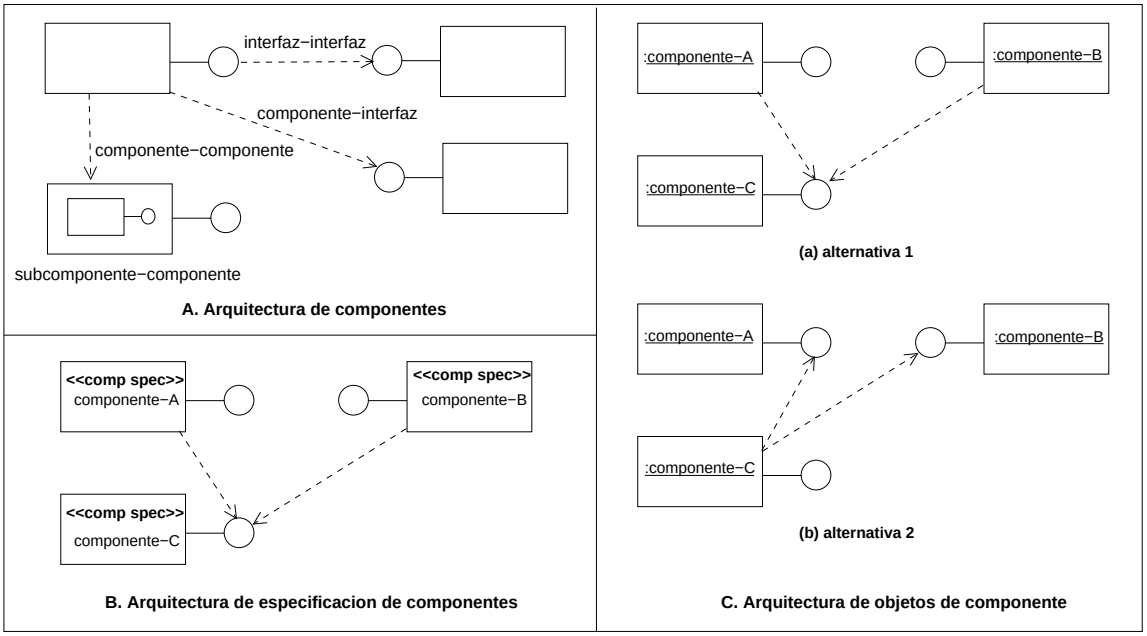

Figura 1.12: Dependencias en una arquitectura de componentes

Además, una arquitectura de componentes se puede centrar en especificaciones de componente, en implementaciones de componente o en objetos de componente (en la página 40 enumeramos las formas en las que puede aparecer un componente según John Cheesman y John Daniels).

Un diagrama de una arquitectura de especificaciones de componente contiene sólo especificaciones de componente e interfaces. Un diagrama de una arquitectura de implementaciones de componente muestra las dependencias que existe entre las implementaciones de un componente en particular. Y por último, un diagrama de una arquitectura de objetos de componente especifica la relación entre las instancias de cada componente. En la figura 1.12.C se muestra dos posibles alternativas de arquitecturas de objetos de componente que cumplen con la arquitectura de especificaciones de componente de la figura 1.12.B.

Los Componentes UML utilizan los diagramas de casos de uso para modelar requisitos, y los diagramas de clases y colaboraciones para modelar especificaciones. Por ejemplo, una especificación de un sistema de componentes está compuesta de cuatro partes: (a) los tipos de datos, (b) las especificaciones de interfaz, (c) las especificaciones de componente, y (d) la arquitectura de componentes. Para modelar estas partes se utilizan los diagramas de clases, y en Componentes UML cada diagrama resultante recibe el nombre de: (a) diagrama de tipos de datos, (b) diagrama de especificaciones de interfaz, (c) diagramas de especificaciones de componente, y (d) diagrama de arquitectura de componentes. Para esta última (los diagramas de arquitectura de componentes) las interacciones se modelan con diagramas de colaboración, y reciben el nombre de diagramas de interacciones de componente.

Como hemos adelantado antes, los Componentes UML utilizan los estereotipos para extender UML. Por ejemplo, para los diagramas de tipos de datos se utilizan los estereotipos `<<type>>` y `<<datatype>>`; para los diagramas de especificaciones de interfaz se utilizan estereotipos como `<<interface type>>` o `<<info type>>`; para los diagramas de especificaciones de componente se utilizan estereotipos como `<<comp spec>>` (como hemos visto en la figura 1.12) o `<<offers>>`. En el diagrama de arquitectura de componentes se pueden utilizar todos los estereotipos.

1.6.5. UML-RT

Aunque es algo más antigua que las demás, la notación de Bran Selic [Selic et al., 1994] denominada ROOM (*Real-time Object Oriented Modelling*), y usada para modelar objetos complejos en tiempo real, ha sido una de las más empleadas para extensiones y propuestas de LDA futuras. Inicialmente esta propuesta no se basaba en UML, pero recientes trabajos de Selic [Selic y Rumbaugh, 1998] [Selic, 1999] [Selic, 2001] la han extendido para conocerse ahora como UML-RT, una mezcla entre UML estándar y ROOM. En la actualidad UML-RT ha sido adoptado por Rational en su herramienta *Rational Rose RealTime*.

UML-RT es una notación gráfica que se basa en un diagrama tradicional de "líneas-y-cajas". En la figura 1.13 se puede ver un ejemplo de este tipo de diagrama[18]. Según la notación de UML-RT, los componentes se representan mediante cajas, denominadas "cápsulas", que pueden contener a su vez otras cápsulas. Esto último se representa en el diagrama mediante dos pequeños cuadros conectados, como se puede ver en la esquina inferior derecha de la cápsula GTS. Esto nos asegura que GTS es el resultado de la composición de otras cápsulas internas a ella.

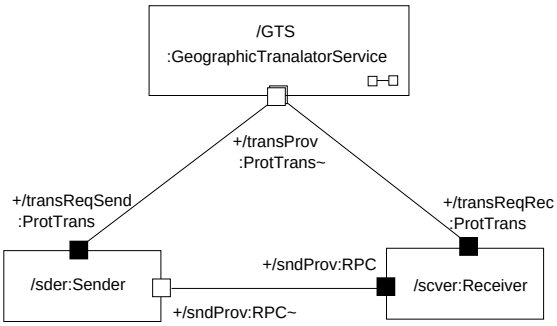

Figura 1.13: Un ejemplo que utiliza notación UML-RT

Cada cápsula (componente) puede tener una o más interfaces, denominados "puertos" y representados por unos cuadrados pequeños de color negro o blanco. Un puerto de color blanco representa los mensajes de entrada y los de color negro los de salida. En cualquier caso, para nuestros propósitos esto lo interpretaremos como las interfaces que el componente oferta (color blanco) y las que requiere para poder funcionar (color negro).

Según esto, en el diagrama de la figura 1.13 intervienen tres componentes: `GTS`, `sver` y `rcver`. Una notación de la forma `/GTS:GeographicTranlatorService` significa que `GTS` es una instancia del componente base `GeographicTranslatorService`. Cada puerto se denota con un nombre y un "protocolo" de comunicación, que establece el orden en el que se envían los mensajes del puerto que encapsula. Por ejemplo, un puerto `+transReqSend:ProtTrans` significa que es una interfaz requerida llamada `transReqSend` y que respeta un protocolo llamado `ProtTrans`. Una interfaz ofertada se representa de forma similar, pero con su "dual", `+transReqSend:ProtTrans~`.

Para finalizar, los "conectores" se representan como simples líneas que unen en sus extremos dos puertos. Si un conector une un puerto con más de uno, estos se deben duplicar en la cápsula que lo requiere. Esto se representa mediante un doble cuadro pequeño, y fuera se indica la cardinalidad de la repetición (aunque esto último no es obligatorio). Por ejemplo, el componente `GTS` replica dos veces la interfaz `transProv` porque enlaza con dos conectores, uno proveniente de la interfaz `transReqRec` del componente `rvcer`, y otro proveniente de la interfaz `transReqSend` del componente `sder`.

[18] El diagrama presentado aquí se corresponde con uno de los ejemplos utilizados en el *Capítulo 5*.

1.7. El Servicio de mediación

La función de mediación es una especificación ISO/ITU-T

El modelo de referencia RM-ODP (*Reference Model for Open Distributed Processing*) es un modelo que está siendo desarrollado conjuntamente por ISO (*International Standard Organization*) e ITU-T (*International Telecommunication Union*). Este modelo defiende la utilización transparente de servicios distribuidos en plataformas y redes heterogéneas y una localización dinámica de estos servicios.

La función de mediación (*trading function*) es una de las 24 funciones del modelo ODP. Originariamente, la función de mediación de ODP fue conocida como ODP Trader [Beitz y Bearman, 1995], y fue desarrollada para proporcionar un servicio de "paginas amarillas" dinámico al Servicio de Directorios de DCE[19] (*Distributed Computing Environment*). Un servicio de páginas amarillas permite que los clientes de un sistema distribuido puedan localizar, a partir de unos atributos de búsqueda y en tiempo de ejecución, los servicios que estos necesitan.

Desde que apareciera la propuesta de la función de mediación de ODP para DCE en 1995, se han desarrollado un gran número de implementaciones, conocidas como servicios de mediación, o como "*traders*". Algunos servicios de mediación basados en la función de mediación de ODP son AI-Trader [Puder et al., 1995], PC-Trader [Beitz y Bearman, 1995], Melody [Burger, 1995], TRADE [Müller-Jones et al., 1995] o DRYAD [Kutvonen, 1996]. Todos ellos aplicados al modelo DCE.

No obstante, antes de aparecer la función de mediación de ODP, ya existían otras implementaciones de servicios de mediación para objetos distribuidos, como Go-Between [Kerry, 1991], RHODOS [Goschinski, 1993] o Y [Popescu-Zeletin et al., 1991]. Sin embargo, estas propuestas no prosperaron debido a la falta de un modelo distribuido común, y se basaban en modelos distribuidos propios.

En 1997 la función de mediación de ODP fue establecida como norma por ISO/ITU-T [ISO/IEC-ITU/T, 1997], y más tarde adoptada como especificación por el OMG (*Object Management Group*) con el nombre de CosTrading, el actual servicio de mediación de CORBAServices del modelo CORBA. En el año 2000, se presenta la versión 1.0 de la especificación del servicio de mediación de OMG [OMG, 2000], basado en ODP/ISO/ITU-T.

En la actualidad persiste la versión 1.0 del servicio de mediación de OMG CosTrading, a partir de la cual se han desarrollado múltiples implementaciones ofrecidas por diversos fabricantes de ORBs de CORBA. Los ORBs más conocidos que implementan el servicio de mediación de CORBA son[20]: AceORB de TAO [Shmidt, 2001], OpenORB de DOG/Intalio [Intalio, 2001], ORBacus de IONA [OOC, 2001], y OpenFusion de PrismTech [PrismTech, 2001].

1.7.1. Roles de los objetos

Desde el punto de vista de la programación orientada a objetos (POO), una función de mediación (conocida como servicio de mediación o *trader*), es un objeto software que sirve de intermediario entre unos objetos que ofertan ciertas capacidades, que se denominan servicios, y otros objetos que demandan la utilización dinámica de estas capacidades.

Roles: importador o exportador

Desde el punto de vista ODP, los objetos que ofertan capacidades al resto del sistema distribuido se les denominan "exportadores", y se dice que anuncian sus servicios a los demás objetos llevando a cabo la acción de exportar sobre el objeto de mediación. Por otro lado, los objetos que demandan capacidades al sistema se les denominan "importadores", y se dice que demandan anuncios de servicio mediante la acción importar sobre el objeto de

[19] Una breve introducción a DCE se hizo en la sección 1.1, página 5.

[20] Estos ORBs implementan además parte de los servicios de CORBAServices del modelo CORBA.

mediación. Como se puede comprobar en la figura 1.14, por tanto, un cliente de un proceso de mediación puede tener uno de los siguientes roles [ISO/IEC-ITU/T, 1997]:

1. Rol **exportador**. Un objeto adopta el rol exportador cuando tiene que anunciar un servicio al sistema. Para ello, utiliza la operación `export()` del objeto de mediación, estableciendo los parámetros oportunos correctamente para llevar a cabo el registro del servicio correspondiente en el repositorio que mantiene el objeto de mediación asociado.

2. Rol **importador**. Un objeto adopta el rol importador cuando requiere o necesita conocer ciertos servicios almacenados en el repositorio del servicio de mediación. Para ello utiliza la operación `query()` del objeto de mediación para consultar en el repositorio, estableciendo los parámetros oportunos correctamente, como las restricciones de consulta que condicionan el número de los servicios que debe devolver el servicio de mediación.

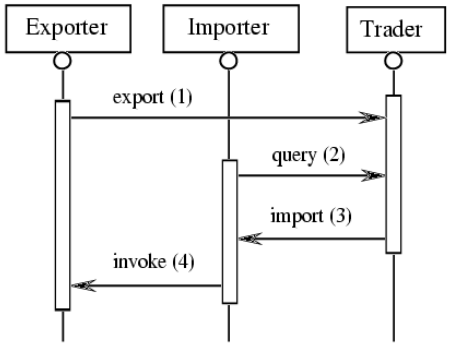

Figura 1.14: Los roles del servicio de mediación de ODP

Si consideramos la figura 1.14 como un diagrama de transiciones, una secuencia de interacción podría ser la siguiente. En primer lugar existe un cliente del servicio de mediación que adopta el rol exportador porque quiere anunciar un servicio en un servicio de mediación conocido (1). Mas tarde (2), otro cliente del servicio de mediación requiere ciertos servicios que se los puede ofrecer el mismo servicio de mediación conocido. éste adopta el rol importador e interroga al servicio de mediación imponiendo sus restricciones de búsqueda. Como resultado (3), el servicio de mediación devuelve al objeto cliente una o varias referencia de objeto asociadas con las instancias del servicio deseado, y que indican dónde se localizan dichos servicios dentro del sistema distribuido. Por último (4), ya es decisión del cliente seleccionar una de las referencia de objeto ofrecidas por el servicio de mediación, y conectarse directamente al objeto que ofrece el servicio buscado. Como vemos, el comportamiento global es parecido al el de un ORB, mostrado en la figura 1.5.

Ejemplos de aplicación de un servicio de mediación

Las aplicaciones que puede tener un servicio de mediación pueden ser múltiples, por ejemplo, para reutilización de aplicaciones existentes (*legacy systems*) [Mili et al., 1995], para recolección y clasificación de las capacidades proporcionadas por un entorno distribuido [Davidrajuh, 2000], o para ser usado como un gestor de buffer de un servicio de impresión en red [OOC, 2001], entre otras aplicaciones. Pero como veremos más adelante, la actual función de mediación ODP presenta algunas limitaciones a la hora de trabajar con componentes COTS.

1.7.2. Las interfaces del servicio de mediación

Un objeto de mediación cuenta con diferentes interfaces funcionales que permiten la interacción con los distintos objetos cliente (importador o exportador). En total, la función de mediación cuenta con cinco interfaces, mostradas en la Tabla 1.9 junto con una pequeña descripción de lo que hace cada una de ellas. En la figura 1.15 también mostramos un esquema donde se relacionan todas las interfaces del servicio CosTrading.

CosTrading: *El servicio de mediación de ODP*

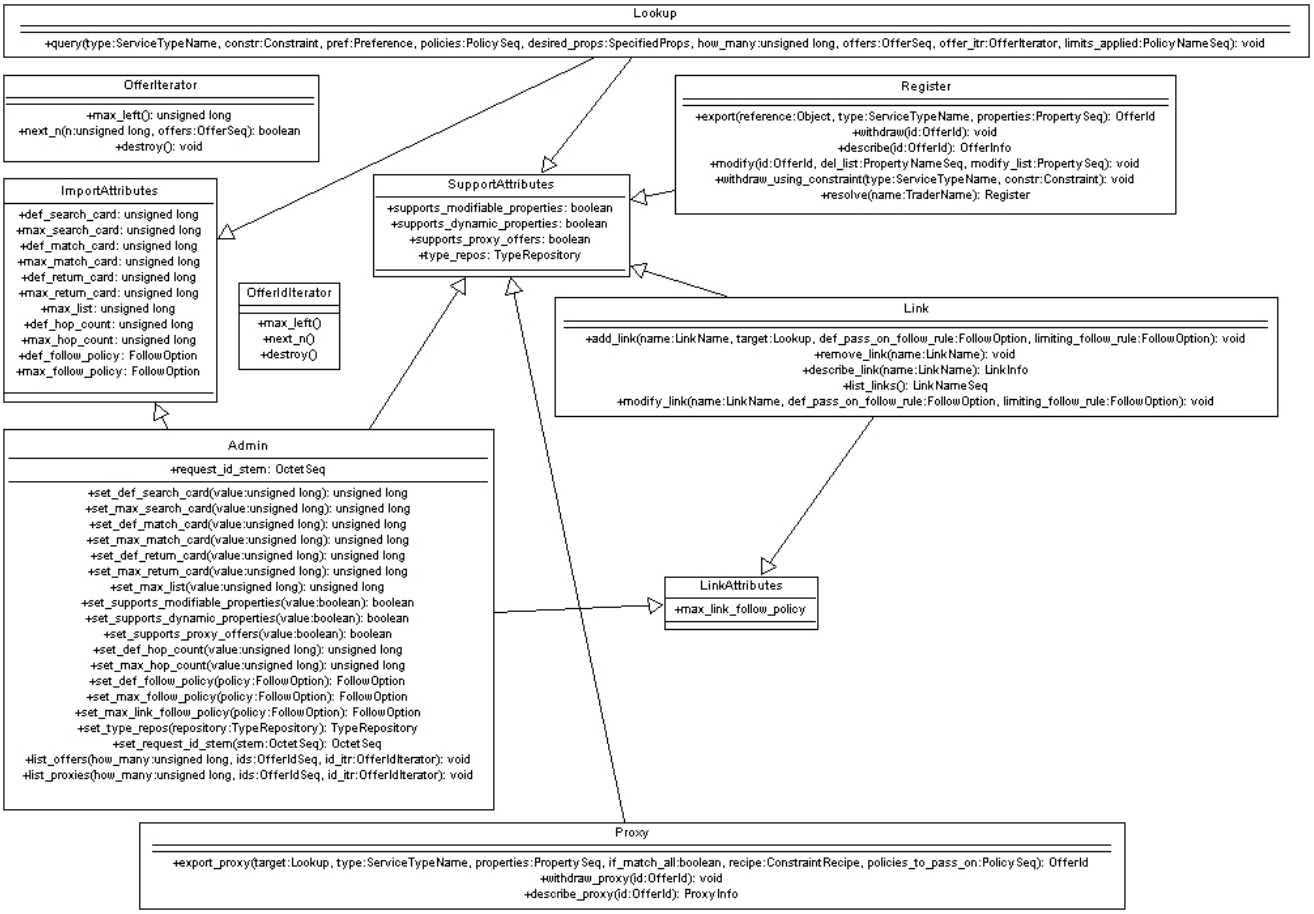

Figura 1.15: Esquema de las interfaces del servicio de mediación CosTrading

En la Figura 1.16 mostramos también una parte del IDL del servicio de mediación de ODP con sus cinco interfaces. De estas cinco interfaces sólo destacamos las interfaces `Register` y `Lookup`, por ser estas las interfaces en las que se ha inspirado el servicio de mediación para componentes COTS, presentado en el *Capítulo 3*.

La interfaz `Register` describe una operación principal denominada `export()`, que permite la acción de exportar[21] un servicio en un repositorio asociado al servicio de mediación. Esta interfaz también dispone de otras operaciones, como la operación `withdraw()` para retirar un servicio, la operación `describe()` para obtener la descripción de un servicio almacenado en el servicio de mediación, `modify()` para modificar un servicio ya registrado, y por último, la operación `withdraw_using_constraint()`, similar a la primera orden de retirar, pero en este caso con acceso al repositorio para seleccionar y eliminar ciertos servicios que cumplen unas condiciones impuestas en la llamada de la operación.

[21]En este entorno, el término exportar un servicio es equivalente a anunciar o registrar un servicio en el servicio de mediación.

Interfaz	Descripción corta
`Lookup`	Permite que los clientes puedan consultar las ofertas de servicio que hay almacenadas dentro del servicio de mediación.
`Register`	Permite que los clientes puedan exportar sus ofertas de servicio en el objeto de mediación.
`Link`	Permite conectar el servicio de mediación con otros, posibilitando la federación de servicios de mediación.
`Proxy`	Puede ser usada como pasarela entre servicios de mediación y también puede ser usada para la herencia de otros servicios de mediación o federación de servicios de mediación ya existentes (*legacy systems*).
`Admin`	Permite que el servicio de mediación pueda ser configurado.

Tabla 1.9: Interfaces del servicio de mediación de ODP

```
module CosTrading {
   interface Lookup { void query(); };
   interface Register {
      OfferId export();
      void withdraw();
      OfferId describe();
      void modify();
      void withdraw_using_constraint();
   };
   interface Link { ... };
   interface Proxy { ... };
   interface Admin { ... };
};
```

Figura 1.16: Una parte del IDL del servicio de mediación de ODP

Por otro lado, la interfaz `Lookup` sólo contiene una operación llamada `query()` que permite consultar en el repositorio del servicio de mediación aquellas referencias de servicio en base a ciertas restricciones de consulta impuestas en la llamada a la operación.

Los tres tipos de repositorios de un servicio de mediación ODP

Un objeto de mediación trabaja con tres tipos de repositorios independientes, y que son: (a) un repositorio de interfaces (RI), (b) otro repositorio donde se almacenan los tipos de servicios que soporta el servicio de mediación (RTS), y (c) por último, un repositorio con las ofertas (RO) registradas por los clientes a partir de los tipos de servicio declarados en RTS.

Cuando un cliente exporta una oferta de servicio de objeto en el servicio de mediación, éste debe primero definir el servicio a partir de un tipo de servicio ya existente en el repositorio de tipos RTS, indicando para ello una descripción de la interfaz —que encapsula la funcionalidad del objeto que se oferta— junto con una serie de características extrafuncionales que identifican dicho objeto y que no pueden ser capturadas dentro de la definición de interfaz. La especificación de la función de mediación acepta la definición de interfaces haciendo uso del IDL de OMG. Cuando el cliente registra su servicio en el servicio de mediación, la interfaz asociada a éste servicio se almacena en el "repositorio de interfaces" independiente, RI, mientras que la oferta de servicio en sí se hace en el "repositorio de ofertas" RO.

Veamos a continuación con más detalle algunos aspectos relacionados con la definición de servicios, tipos de servicio y ofertas de servicio. Luego, para ayudar a entender mejor estos conceptos, introduciremos un simple ejemplo.

1.7.3. Servicios y tipos de servicio

Un servicio de mediación puede soportar diferentes tipos de servicios, todos ellos almacenados en un "repositorio de tipos de servicio" independiente, *RTS*. Un servicio de una función de mediación puede quedar identificado por una terna de la forma:

$$<\textit{Tipo, IDL, Propiedades}>$$

donde, *Tipo* es el tipo de servicio anunciado, *IDL* es el nombre de la interfaz IDL que describe la signatura computacional del servicio anunciado, y *Propiedades* es una lista de propiedades que capturan aspectos extra-funcionales y no recogidos por la interfaz del servicio anunciado.

Utilizando la sintaxis de la especificación del servicio de mediación ODP, un servicio se puede expresar de la siguiente forma:

```
service <ServiceTypeName>[:<ServiceTypeName>[,<ServiceTypeName>]*]{
    interface <InterfaceTypeName>;
    [[mandatory] [readonly] property <Type> <PropertyName>;]*
}
```

Según la notación anterior podemos observar varias características asociadas a un servicio de mediación de ODP. Un nuevo tipo de servicio puede estar compuesto por tipos de servicio ya existentes soportados por el servicio de mediación, permitiendo herencia múltiple de supertipos de servicio. También, un tipo de servicio requiere la declaración de un único nombre de interfaz que encapsula los aspectos computacionales de la capacidad del servicio que se desea anunciar. Además, un servicio puede contener una o más propiedades o atributos que encierran aspectos extra-funcionales y que no pueden ser recogidos por la interfaz asociada. Cada propiedad puede ser construida como una terna:

$$<\textit{nombre, tipo, modo}>$$

En este caso, *nombre* es el nombre de la propiedad que se está declarando, *tipo* es el tipo de valor que acepta la propiedad, como por ejemplo un tipo **string**, **float**, o **long**, entre otros valores. Aunque esto último no viene reflejado en la especificación de ODP, la mayoría de las paquetes software industriales que implementan la función de mediación de ODP, como ORBacus [OOC, 2001], OpenORB [Intalio, 2001] o TAO [Shmidt, 2001], entre otros, utilizan los tipos `CORBA::TypeCode` definidos en CORBA. Por último, *modo* es el modo de acceso a la propiedad que se está declarando. Por ejemplo, un modo **mandatory** quiere decir que es obligatorio proporcionar un valor para la propiedad cuando se exporte una oferta de ese tipo de servicio, mientras que un modo **readonly** no obliga a especificar el valor, es opcional hacerlo.

1.7.4. Oferta y consulta de servicios

Como hemos visto, un objeto cliente puede exportar una capacidad o servicio mediante la operación `export()` de la interfaz `Register` asociada al servicio de mediación. Cuando un objeto cliente exporta un servicio se dice que está anunciando una "oferta" de un tipo de servicio que soporta el servicio de mediación, y es registrada en un "repositorio de ofertas" independiente, *RO*. Toda oferta de servicio queda identificada de la siguiente forma:

<Tipo, Localización, Propiedades>

donde, *Tipo* es el nombre del tipo de servicio que se anuncia, *Localización* es el lugar donde se encuentra el objeto que implementa el tipo de servicio que se está anunciando, y *Propiedades* es una lista de las propiedades que cumplen con la definición del tipo de servicio *Tipo*. Para este último caso, cada propiedad es un par *<Nombre, Valor>*, donde *Nombre* representa el nombre de la propiedad y *Valor* es el valor proporcionado para dicha propiedad.

1.7.5. Un ejemplo

A continuación vamos a introducir un simple ejemplo para analizar mejor las definiciones introducidas hasta el momento. El ejemplo también servirá de argumento para justificar, más tarde, algunas limitaciones del servicio de mediación de ODP que hemos detectado para el caso de los componentes COTS.

Supongamos que disponemos de un componente software que implementa distintos algoritmos para poder convertir de forma eficiente archivos de imagen con formato **GIF** a otros formatos de imagen, como por ejemplo **JPG**, **PNG**, y **TIF**. Una simple interfaz para este ejemplo podría ser la siguiente:

```
interface ConversorImagenGif {
   string gifajpg(in string nombre);
   string gifapng(in string nombre);
   string gifatif(in string nombre);
};
```

La interfaz `ConversorImagenGif` puede ser registrada en un servicio de mediación creando un tipo de servicio ODP para ella (normalmente por el cliente), antes de llevar a cabo la llamada a la función que efectúa la operación de registro (`export()`) en el servicio de mediación. Siguiendo la notación utilizada por ODP para definir un tipo (vista anteriormente), podríamos crear un servicio como el que muestra a continuación para la interfaz `ConversorImagenGif`:

```
service ConversorGif {
    interface ConversorImagenGif;
    mandatory property long altura_max;
    mandatory property long anchura_max;
    mandatory property string unidad;
    readonly  property boolean color;
}
```

Para este ejemplo, el servicio de mediación declara un tipo de servicio con el nombre `ConversorGif`, el cual encapsula la interfaz `ConversorImagenGif` y declara cuatro propiedades adicionales. La propiedad `altura_max` y `anchura_max` son del tipo `long` y se utilizan para especificar las dimensiones máximas de imagen que soporta el servicio de conversión. La propiedad `unidad` indica el tipo de la unidad de medida de la dimensión de la imagen que soporta el servicio de conversión, por ejemplo, podría ser `pt`, para pixeles, o `mm` para milímetros, o `in` para pulgadas. La propiedad `color` es un parámetro de tipo lógico que indica si la oferta de servicio acepta imágenes en color o no.

Los objetos cliente anuncian sus implementaciones particulares para el tipo de servicio `ConversorGif` mediante ofertas que se registran en el servicio de mediación. Por ejemplo, lo siguiente representa tres ofertas para el servicio anterior, anunciadas por tres clientes distintos, respetando la sintaxis de oferta *<Tipo, Localización, Propiedad>*, descrita anteriormente.

```
offer ConversorGif {
  Location objeto1;
  altura_max=860;
  anchura_max=1024;
  unidad="pt";
  color=true;
}
```

```
offer ConversorGif {
  Location objeto2;
  altura_max=860
  anchura_max=860;
  unidad="mm";
  color=false;
}
```

```
offer ConversorGif {
  Location objeto3;
  altura_max=300;
  anchura_max=600;
  unidad="in";
}
```

De las ofertas anteriores es necesario destacar que `Location` es una referencia al objeto que implementa el tipo de servicio que se está ofertando en el servicio de mediación, en este caso una referencia a un objeto que implementa el servicio `ConversorGif`.

Para las tres ofertas de servicio anteriores, vemos que las tres limitan el tamaño máximo de imagen que soportan, expresado en diferentes unidades, pixeles para el primer objeto, milímetros para el segundo, y pulgadas para el tercero. Además, también podemos comprobar que la primera oferta de servicio admite imágenes en color, la segunda no, y la tercera se desconoce. Como hemos visto antes, cuando definimos el tipo de servicio `ConversorGif` la propiedad `color` fue declarada como opcional (tipo de dato `readonly`), mientras que las otras tres eran obligatorias (tipo de dato `mandatory`).

La principal responsabilidad de un servicio de mediación es satisfacer las peticiones de un cliente importador. Estas peticiones las realiza el cliente en base a cuatro características que éste debe establecer antes de consultar o efectuar la acción de importar sobre el servicio de mediación. Estas características se definen como sigue:

$$<\textit{TipoServicio, Expresión, Preferencia, Políticas}>$$

donde,

(a) *TipoServicio*, es el tipo de servicio de las ofertas almacenadas por el servicio de mediación que se ven involucradas en la búsqueda bajo las condiciones impuestas por el cliente en los tres siguientes valores.

(b) *Expresión*, es una expresión lógica escrita en el lenguaje de restricciones de OMG que define las restricciones de búsqueda que el cliente impone sobre las ofertas del servicio de mediación. Las restricciones se imponen sobre las propiedades del tipo de servicio. Por ejemplo, la expresión "`color==true or unidad=='pt'`" podría ser una restricción para localizar las ofertas del tipo de servicio `ConversorGif` que admita imágenes en color con unidades de medida en pixeles.

(c) *Preferencia*, indica cómo deben ser ordenadas las ofertas que el servicio de mediación devuelve al cliente. El servicio de mediación de ODP admite cinco preferencias distintas de ordenación, y que son: `max` (ascendente), `min` (descendente), `with` (bajo condición), `random` (aleatorio) y `first` (en el orden natural del servicio de mediación). Por ejemplo, para el caso del tipo de servicio `ConversorGif`, una preferencia "`with(color==true)`" significa que el servicio de mediación colocará en primer lugar las ofertas que admiten imágenes en color, como resultado de la evaluación de búsqueda bajo las condiciones impuestas en *Expresión*.

(d) *Políticas*, son unas políticas que condicionan la búsqueda. Por ejemplo, hay unas políticas que limitan la cardinalidad de las ofertas que intervienen en el proceso de búsqueda del servicio de mediación.

En la figura 1.17 (especificación de mediación, [ISO/IEC-ITU/T, 1997]) se ilustra la secuencia de tareas que tiene lugar en una acción *importar*, llevada a cabo por un cliente sobre el servicio de mediación. En esta secuencia, el cliente puede establecer cardinalidades diferentes que limitan el número de ofertas consideradas en cada paso.

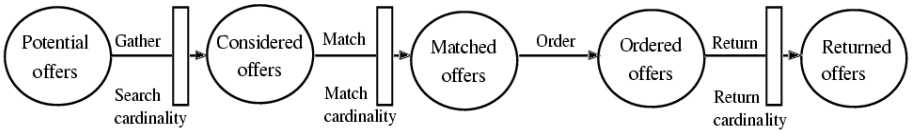

Figura 1.17: Secuencia de tareas para el rol importador del servicio de mediación ODP

1.7.6. Federación de servicios de mediación

En una federación, un objeto servicio de mediación puede jugar el papel de objeto cliente sobre otro servicio de mediación distinto

La especificación del servicio de mediación de ODP permite a los administradores de un sistema basado en servicios de mediación, enlazar su servicio de mediación con otros conocidos, permitiendo así la propagación de las peticiones de consulta en una red. Cuando un cliente con el rol de importador ejecuta una operación de consulta, el servicio de mediación siempre busca las ofertas en su propio conjunto de ofertas de servicio, pero también puede reenviar la petición de búsqueda a otros servicios de mediación enlazados para que localicen nuevas ofertas de servicio bajo las mismas condiciones de búsqueda, impuestas inicialmente sobre el servicio de mediación origen.

Cada servicio de mediación impone sus propias políticas internas que gestionan su funcionalidad y que incluso pueden contrarrestar las políticas impuestas por el cliente en su petición de consulta.

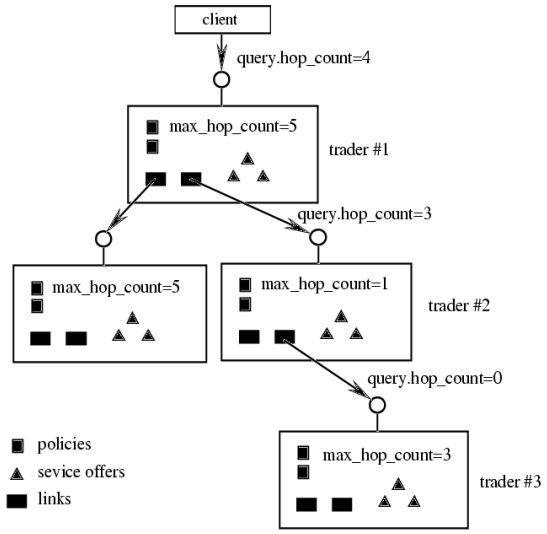

Figura 1.18: Propagación de una consulta en una federación de servicios de mediación

Por ejemplo, como se puede ver en la figura 1.18 (obtenida de la especificación del servicio de mediación [ISO/IEC-ITU/T, 1997]), una de estas políticas limita la profundidad de la búsqueda o el número de servicios de mediación por los que se puede propagar la petición de consulta (`hop_count`). El valor de política asociado a la petición de la consulta se decrementa en uno cada vez que se propaga de un servicio de mediación a otro, y siempre tiene que ser inferior o igual al valor de política que mantiene internamente el

servicio de mediación que recibe la petición. Esto último no sucede para el valor de política de la petición que recibe el servicio de mediación `#2`, que se ve contrarrestado por el valor interno del propio servicio de mediación.

1.7.7. Limitaciones de la función de mediación de ODP

Una vez analizadas algunas de las características de la especificación de la función de mediación de ODP, en esta sección vamos a destacar algunas limitaciones que presenta para el caso de componentes COTS. Estas limitaciones han sido detectadas en base a la experiencia obtenida tras analizar algunas de las implementaciones industriales del servicio de mediación [Shmidt, 2001] [Intalio, 2001] [OOC, 2001] [PrismTech, 2001] y otros trabajos relacionados [Bearman, 1997] [Beitz y Bearman, 1995] [Merz et al., 1994].

En total hemos detectado ocho limitaciones que analizamos a continuación. Las siete primeras se corresponden con limitaciones funcionales en general, y la última de ellas tiene que ver con limitaciones específicas con aspectos de computación.

A. Modelo de Objetos de OMG

La especificación de la función de mediación de ODP está desarrollada para trabajar con objetos que siguen el modelo de OMG, pues adopta su conocido IDL (*Interface Description Language*) para la identificación de la interfaz de un servicio anunciado.

Uno de los principales objetivos de nuestro trabajo de investigación ha sido desarrollar un servicio de mediación que soporte componentes COTS y que funcione haciendo uso del potencial que tiene Internet. Por tanto, era necesario una función de mediación que soportara varios modelos de componentes, además del modelo de CORBA. En este sentido, se han desarrollado unas plantillas de especificación de componentes COTS en XML donde se pueden indicar las definiciones de las interfaces de un componente.

B. No es Suficiente IDL + Propiedades

Como se ha visto en la sección 1.7, todo servicio queda identificado por una interfaz, que recoge los aspectos funcionales del servicio, y por un conjunto de propiedades, que recogen características extra-funcionales del servicio. Sin embargo, pensamos que el tándem "IDL+propiedades" no es suficiente para el caso de los componentes COTS. La tarea de importación de un servicio de mediación sólo permite hacer declaraciones de restricción y consultas sobre las propiedades de un servicio, pero no soporta esto para la información contenida en el IDL asociado a las propiedades.

Por ejemplo, el modelo de componente COTS (*Capítulo 2*) utiliza un IDL que contempla cuatro tipos de información: (a) funcional; (b) extra-funcional; (c) información de empaquetamiento/implantación del componente; y (d) información de marketing (p.e., precio, licencia o información del vendedor, entre otros valores). Cuando un cliente desee hacer una consulta en el servicio de mediación, éste debería poder imponer sus restricciones de búsqueda sobre las características internas al IDL del servicio (componente), como por ejemplo, sobre las signaturas o métodos que ofrece y requiere el servicio para funcionar, o sobre los protocolos del servicio, entre otros.

C. Emparejamiento Parcial

Siguiendo en la línea anterior, el hecho de que en una petición de búsqueda el servicio de mediación ODP limite las operaciones de restricción sólo sobre las propiedades de una oferta de servicio, y no sobre la interfaz de ésta, obliga a que la función de mediación tenga

que realizar emparejamientos exactos (*Exact Matchings*, o emparejamientos fuertes) entre petición/ofertas. Sin embargo, para el caso de los componentes COTS es necesario que la función de mediación también permita imponer restricciones de consulta sobre el IDL de una oferta de servicio, con el fin de soportar emparejamientos parciales (*Partial Matches*, o emparejamientos débiles) [Zaremski y Wing, 1995].

Por ejemplo, consideremos de nuevo el tipo de servicio `ConversorGif` tratado anteriormente. Como vimos, este tipo de servicio respondía a la interfaz `ConversorImagenGif`, la cual contenía tres operaciones, `gifajpg()`, `gifapng()` y `gifatif()`. Para simplificar, y solo para este ejemplo, vamos a definir un servicio como un conjunto de operaciones de la forma $C = \{O_1, O_2, ..., O_n\}$, siendo O_i una operación del servicio, y n el número de operaciones que incluye dicho servicio.

Por tanto, para el ejemplo que estamos tratando, el tipo de servicio `ConversorGif` queda definido como $Conversor = \{gifajpg(), gifapng(), gifatif()\}$. Un servicio de mediación con emparejamientos parciales (*partial matches*) debería devolver las ofertas de servicio de tipo `ConversorGif` para una petición de consulta que incluya por ejemplo el servicio $B = \{gifajpg()\}$, y también para la petición de consulta que incluya el servicio $C = \{gifajpg(), gifapng(), tex2html()\}$.

D. Comportamiento con Múltiples Interfaces

Uno de los aspectos que caracteriza a un componente COTS es que puede tener múltiples interfaces. Aunque la función de mediación de ODP asocia una interfaz con un servicio y también permite herencia entre servicios, ésta no permite el registro directo de componentes con múltiples servicios (múltiples interfaces).

Por ejemplo, supongamos un componente `Conversor` que traslada archivos TEX a archivos en formato `HTML`. Además, los archivos de imagen con formato `GIF` vinculados en el archivo TEX los traslada a archivos de imagen con formato `JPG`. Este componente puede venir identificado por dos interfaces, `ConversorTeX` y `ConversorGif` —como el de la sección 1.7.4. Según esto, cuando un cliente desee anunciar este componente en el servicio de mediación, primero se debe crear un servicio para la interfaz `ConversorTeX`, otro para la interfaz `ConversorGif`, y por último otro servicio para el componente `Conversor`, que herede los dos servicios anteriores, en lugar de hacerlo directamente con un único servicio. Sin embargo, el tipado y subtipado (en este caso de servicios) puede acarrear ciertos problemas de compatibilidad con nuevos servicios creados a partir de estos. Estos problemas se describen en detalle en el trabajo "Compound Types" [Weck y Büchi, 1998].

E. Emparejamiento uno a uno

El proceso de interacción entre un cliente y un servicio de mediación es un proceso que involucra emparejamientos uno a uno entre servicios. La función de mediación de ODP no permite procesos de emparejamiento a partir de criterios de selección para más de un servicio, impuestos por el cliente.

En la línea del ejemplo expuesto para el anterior caso, un cliente no puede especificar a la vez criterios de consulta sobre las propiedades del tipo de servicio `ConversorTeX` y sobre las propiedades del tipo `ConversorGif`, y considerando además estos dos servicios como servicios independientes. El cliente, no obstante, sí puede hacer la consulta sobre el tipo de servicio `Conversor`, que hereda los dos tipos anteriores, pero de nuevo involucra emparejamiento uno a uno. Por tanto, estamos ante la necesidad de disponer de procesos de mediación que soporten enfrentamientos muchos a muchos, útiles por ejemplo para el desarrollo de funciones de mediación composicionales.

F. Modelo bajo Demanda para Registrar Servicios

La función de mediación de ODP se basa en un modelo bajo demanda (modelo *push*) para el almacenamiento de las ofertas de servicio en el repositorio de ofertas (*RO*) ligado al servicio de mediación. En este sentido, las peticiones llegan al servicio de mediación de forma desinteresada bajo demanda de un cliente, sin tener conocimiento de cómo, cuándo y quién las realiza. No obstante, en un entorno de componentes COTS que funcione bajo Internet, la función de mediación debería soportar además un modelo basado en extracción de información (o modelo *pull*).

G. Federación Directa Basada en Consulta

Como hemos visto en la sección 1.7.6, la función de mediación de ODP permite enlazar un servicio de mediación con otros, y estos a su vez con otros, obteniendo así una colección de servicios de mediación interconectados (una federación). Sin embargo, tal y como se pudo ver en la figura 1.18, estos enlaces son de sentido único. Si queremos que un servicio de mediación A comparta la información con otro servicio de mediación B y viceversa, es necesario establecer un enlace en A hacia B, y otro en B hacia A.

Por otro lado, no se aprovecha el potencial de federación para el caso de las exportaciones de ofertas. Cuando un cliente anuncia un servicio —exporta o registra un servicio— siempre lo hace sobre un servicio de mediación conocido, sin llevar a cabo comprobaciones de existencia en la federación.

Sin embargo, esto no ocurre para el caso de las importaciones, donde sí se realizan recorridos por los servicios de la federación para satisfacer una petición de consulta. Por tanto, la función de mediación de ODP utiliza un modelo de federación directa orientada sólo a importación. Un servicio de mediación de componentes COTS debería soportar un comportamiento similar para la exportación.

La calidad de servicio incluye atributos tales como el tiempo de respuesta máximo, la respuesta media, o la precisión. También se podrían exigir otros campos adicionales, como campos para la seguridad o confidencialidad, certificados, cotas de mercado, plataformas donde funciona el servicio o el modelo de componentes que soporta, entre otros campos.

H. Otras Limitaciones Computacionales

En cuanto a limitaciones a nivel computacional, la función de mediación de ODP presenta algunas limitaciones para trabajar con componentes COTS. Seguidamente enumeramos algunas de ellas:

— Es posible modificar una oferta de servicio almacenada en el repositorio de ofertas de servicio (*RO*), mediante la operación `modify()` de la interfaz `Register`. No existe sin embargo ninguna función similar para modificar un tipo de servicio una vez almacenado en el repositorio de tipos de servicio (*RTS*).

— El cliente registra ofertas de servicio en el *RO* en base a unos tipos de servicio que deben existir previamente en el *RTS*. En la especificación no queda claro quién, cómo y cuando realiza el registro de un tipo de servicio en el *RTS*.

— La operación `query()`, ligada a la acción de importar (dentro de la interfaz `Lookup` del servicio de mediación), no permite realizar consultas sobre las propiedades de dos o más tipos de servicio a la vez.

— En una federación no existen políticas que afecten, de forma global, a un grupo o a

toda la federación a la vez. Por ejemplo, si se desea una cardinalidad de búsqueda de 10, hay que indicarlo para cada servicio de mediación de la federación.

— Cuando se exporta un servicio, el servicio de mediación sólo realiza comprobaciones de existencia en base al nombre del tipo de servicio, enfrentándolo con los nombres de los tipos de servicios del *RTS*, sin considerar el contenido del tipo de servicio. Por ejemplo, sea $A = \{IDL_A, a, b\}$ un tipo de servicio que se va a registrar en el servicio de mediación, con una interfaz IDL_A y dos propiedades a y b; y sea $A = \{IDL_A, c, d, e\}$ otro tipo de servicio ya existente en el *RTS*. Cuando el proceso de consulta que rastrea el repositorio *RTS* en busca de un servicio similar a A, encuentra el servicio B en el repositorio, el servicio de mediación considerará a los dos como idénticos al coincidir sus nombres de tipo, y por tanto no lleva a cabo el registro del servicio A en el *RO*, pues piensa que ya existe. Sin embargo, el servicio de mediación sí registraría un servicio de la forma $B = \{IDL_A, a, b\}$, idéntico al servicio A.

— Existe un conjunto de características no incluidas en la especificación del servicio de mediación que afectan a su normal funcionamiento y que deben establecerse en tiempo de implantación o de activación del servicio. Por ejemplo, la característica de permisividad de la persistencia del servicio de mediación, y en su caso, el intervalo de actualización (*commits*) de los datos sobre los repositorios.

1.8. UDDI: Directorio de servicios webs

UDDI es una especificación de una función de directorio para "servicios web" (*Web Services*). Las siglas UDDI significan *Universal Description, Discovery, and Integration*. Esta especificación aparece en el año 2000 con el nombre de "proyecto UDDI" versión 1.0, y en su elaboración participaron conjuntamente empresas como IBM, Microsoft y Ariba.

UDDI 1.0 surge como una necesidad de estandarizar las tareas de construcción de repositorios de servicios web

El objeto de esta sección es analizar la función de directorio de UDDI y ver cómo ésta ha influido en el desarrollo de la propuesta del servicio de mediación *COTStrader*, descrito en el *Capítulo 3*. Pero antes de abordar las características de la función de directorio de UDDI, veamos primero algunos detalles acerca de los servicios web y de la tecnología que los sustenta.

1.8.1. Servicios web y su tecnología

Los servicios web son pequeñas aplicaciones que pueden ser publicadas, localizadas e invocadas desde cualquier sitio web o red local basados en estándares abiertos de Internet. Combinan los mejores aspectos de la programación basada en componentes y la programación web, y son independientes del lenguaje en el que fueron implementados, del sistema operativo donde funcionan, y del modelo de componentes utilizado en su creación.

Una definición algo más formal y extendida es la que ofrece el grupo de trabajo de Servicios Web del W3C [W3C-WebServices, 2002] que dice lo siguiente:

Definición 1.11 (Servicio Web de W3C) *Un servicio web en un sistema software identificado por un URI (un identificador de recursos uniforme) [Berners-Lee et al., 1998], cuyas interfaces públicas y enlaces están definidos y descritos en XML, y su definición puede ser localizada por otros sistemas de software que pueden luego interactuar con el servicio web en la manera prestablecida por su definición, utilizando mensajes basados en XML transmitidos por protocolos de Internet.*

SOAP (Simple Object Acess Protocol)

Los servicios web surgieron como aplicaciones remotas accesibles por Internet (usando Servlets, JSP, ASP y otra tecnología web) utilizadas para la programación web en entornos de comercio electrónico. Con la consolidación de XML, los servicios web empezaron a utilizar mensajes en XML para la consecución de transacciones distribuidas utilizando el protocolo de transferencia HTTP.

Para estandarizar el formato de transferencia de los mensajes en XML entre servicios web, en mayo del año 2000 se crea la primera especificación de SOAP (*Simple Object Access Protocol*). Esta especificación fue elaborada y propuesta al W3C (*World Wide Web Consortium*, véase `http://www.w3c.org`) por compañías como Ariba, CommerceOne, Compaq Computer, DevelopMentor, Hewlett-Packard, IBM, IONA, Lotus, Microsoft, SAP AG y Userland Software. La versión actual de SOAP es la 1.2 (véase `http://www.w3c.org/TR/soap12`).

A modo de ejemplo, en la figura 1.20 se muestra un mensaje SOAP que se transfiere entre dos aplicaciones. Este mensaje se corresponde con una transacción típica entre dos aplicaciones de comercio electrónico para la identificación de un usuario por Internet. Como podemos comprobar, el mensaje SOAP "transporta" una llamada al procedimiento remoto `Identificar()`. El procedimiento devuelve dos valores en los parámetros `Nombre` y `ApellidoA`: los datos del usuario que se identifica con un valor de DNI en la llamada. Por tanto, la signatura de nuestro procedimiento ejemplo podría ser:

```
Identificar(in long DNI, out string Nombre, out string ApellidoA)
```

El ejemplo SOAP mostrado suponemos que es un mensaje de respuesta de una aplicación *B* servidora hacia otra aplicación *A* cliente. La idea, como vemos, es similar a una llamada convencional RPC.

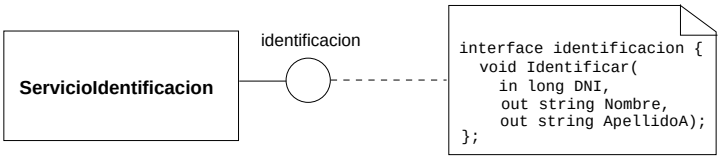

Figura 1.19: Un servicio de identificación por Internet

```
<soap:Envelope
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:tipos="http://www.w3.org/2000/XMLSchema">
    <soap:Body>
        <Nombre tipos:type="xsd:string">Marta</Nombre>
        <ApellidoA tipos:type="xsd:string">Iribarne</ApellidoA>
    </soap:Body>
</soap:Envelope>
```

Figura 1.20: Ejemplo de un mensaje SOAP

El mensaje consta de un elemento principal de "envoltorio" (`Envelope`) que encapsula el mensaje completo. Este elemento está definido en el esquema `http://schemas.xmlsoap.org/soap/envelope/` al que apunta el espacio de nombres `soap`. Para los tipos de datos del ejemplo se ha utilizado los definidos en los esquemas del W3C en `http://www.w3.org/2000/XMLSchema`, al que apunta el espacio de nombres `tipos`, aunque se podrían utilizar

otros tipos definidos por la propia organización o en otros esquemas conocidos que hayan sido definidos por otras organizaciones.

Por regla general, un mensaje SOAP es algo más complejo que el mostrado anteriormente en la figura 1.20. De hecho, como muestra la figura 1.21, la versión 2.0 de la especificación SOAP soportará hasta siete extensiones diferentes del lenguaje de esquemas SOAP. Estas extensiones son las de adjuntos (*attachments*), encaminamiento (*routing*), mensajería (*messaging*), calidad de servicio (*QoS*), seguridad (*security*), privacidad (*privacy*) y soporte para transacciones (*transactions support*). Como protocolos de transporte, los mensajes SOAP se envían/reciben en HTTP, SMTP, FTP, o HTTPS. Además, como podemos ver en su arquitectura, SOAP es similar a un ORB convencional para interconectar objetos distribuidos, pero en este caso objetos web.

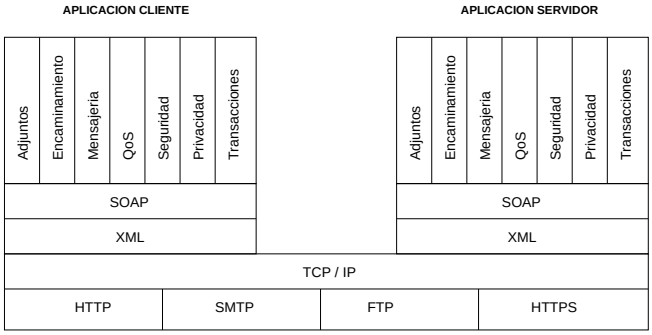

Figura 1.21: Extensiones de la especificación SOAP 1.2

La especificación SOAP está fuera del alcance del presente documento. Aquí sólo hemos ofrecido una escueta descripción para situarnos en el campo de los servicios web, y también por su utilidad en las llamadas a las funciones del servicio UDDI (como veremos más adelante). Para mayor información sobre la especificación SOAP puede consultar en la página web del W3C (`http://www.w3c.org`) o en [Cauldwell et al., 2001].

WSDL (Web Services Definition Language)

Desde su aparición, los estándares XML y SOAP empezaron a ser utilizados en la programación de aplicaciones web. Sin embargo, las dos compañías pioneras en la creación y soporte del estándar SOAP (IBM y Microsoft), también vieron la necesidad de definir los servicios web en notación XML. En los primeros meses del año 2000, IBM desarrolló NASSL (*Network Accessibility Service Specification Language*), un lenguaje basado en XML para la definición de interfaces de servicios web, y que utilizaba el XML-Schemas del W3C como lenguaje para definir los tipos de datos de los mensajes SOAP. Por otro lado, en el mes de abril de ese mismo año, Microsoft lanzó al mercado SCL (*Service Contract Language*), y que también utilizaba XML para definir interfaces de servicios web, y el XDR (*XML Data-Reduced*) como lenguaje para la definición de tipos de datos. Pocos meses después, Microsoft ofrece SDL (*Services Description Language*) incorporado a Visual Studio .NET.

Afortunadamente, y como objetivo común a ambas compañías, IBM y Microsoft —junto con Ariba— aunaron sus esfuerzos y crearon un grupo de trabajo para la elaboración de un lenguaje estandarizado. Como resultado se genera WSDL (*Web Services Definition Language*), el primer lenguaje para la definición de servicios web. WSDL fue aceptado como recomendación por el W3C en marzo de 2001, y su especificación se puede encontrar en `http://www.w3.org/TR/wsdl`. Finalmente, este lenguaje utiliza la definición de tipos de datos base del W3C, pudiendo ser extendidos. IBM ofrece soporte WSDL en `http:`

//alphaworks.ibm.com/tech/webservicestoolkit. Microsoft ofrece soporte WSDL en http://msdn.microsoft.com/xml y en http://msdn.microsoft.com/net.

Por tanto, un servicio web definido en WSDL permite que cualquier cliente web pueda llamarlo mediante programación sin tener que conocer nada acerca de los detalles de implementación del servicio web o sobre qué plataforma o sistema operativo está funcionando.

```
1:  <?xml version="1.0"?>
2:  <wsdl:definitions name="servicioIdentificacion"
3:        targetNamespace="http://sitioficticio.es/servicioIdentificacion.wsdl"
4:        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5:        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
6:     <wsdl:types>
7:        <xsd:schema targetNamespace="http://sitioficticio.es/identificacion.xsd"
8:                    xmlns:xsd="http://www.w3c.org/2000/10/XMLSchema">
9:           <xsd:element name="peticionIdentificacion">
10:             <xsd:complexType>
11:                <xsd:element name="DNI" type="xsd:float"/>
12:             </xsd:complexType>
13:          </xsd:element>
14:          <xsd:element name="devolucionIdentificacion">
15:             <xsd:complexType>
16:                <xsd:element name="Nombre" type="xsd:string"/>
17:                <xsd:element name="ApellidoA" type="xsd:string"/>
18:             </xsd:complexType>
19:          </xsd:element>
20:       </xsd:schema>
21:    </wsdl:types>
22:    <wsdl:message name="datosEntrada">
23:       <wsdl:part name="body" element="peticionIdentificacion"/>
24:    </wsdl:message>
25:    <wsdl:message name="datosSalida">
26:       <part name="body" element="devolucionIdentificacion"/>
27:    </wsdl:message>
28:    <wsdl:portType name="identificacionTipoPuerto">
29:       <wsdl:operation name="identificar">
30:          <wsdl:input message="datosEntrada"/>
31:          <wsdl:output message="datosSalida"/>
32:       </wsdl:operation>
33:    </wsdl:portType>
34:    <wsdl:binding name="identificacionEnlace" type="identificacionTipoPuerto">
35:       <wsdl:operation name="Identificar">
36:          <soap:operation soapAction="http://sitioficticio.es/identificar"/>
37:          <wsdl:input><soap:Body use="literal"/></wsdl:input>
38:          <wsdl:output><soap:Body use="literal"/></wsdl:output>
39:       </wsdl:operation>
40:    </wsdl:binding>
41:    <wsdl:service name="servicioIdentificacion">
42:       <wsdl:documentation>Un servicio de identificación</wsdl:documentation>
43:       <wsdl:port name="identificacion" binding="identificacionEnlace">
44:          <soap:address location="http://sitioficticio.es/servlet/control.Id"/>
45:       </wsdl:port>
46:    </wsdl:service>
47: </wsdl:definitions>
```

Figura 1.22: Ejemplo de un servicio web definido en WSDL

Un documento WSDL está compuesto por siete elementos XML, aunque no necesariamente todos ellos son obligatorios para definir un servicio web. Estos elementos son: (1) los tipos (<types>), que se definen utilizando un esquema del W3C (aunque también se pueden utilizar otros tipos particulares accesibles desde un espacio de nombres); (2) los mensajes de entrada y salida (<message>); (3) los tipos de puerto (<portType>), donde se definen las interfaces del servicio; (4) las operaciones de una interfaz (<operation>);

(5) los enlaces (<binding>); (6) los puertos (<port>); y el (7) servicio (<service>). Todos estos elementos (y algunos más) están definidos dentro del esquema WSDL http://schemas.xmlsoap.org/wsdl/.

En la figura 1.22 mostramos una definición de un servicio web definido utilizando el lenguaje WSDL. Este servicio web se corresponde con el ejemplo tratado anteriormente, para la identificación de un usuario de un sitio web. Como vimos, el servicio web acepta como dato de entrada un valor de DNI. Internamente el servicio utiliza el DNI para buscar los datos del usuario correspondiente en su base de datos asociada. Como respuesta, el servicio web devuelve el nombre y el primer apellido del usuario identificado.

El contenido de un documento WSDL se "escribe" de forma modular, definiendo primero cada una de sus partes en las primeras secciones del documento, y luego componiéndolas en el resto del documento. En primer lugar se definen los tipos de datos utilizados por el servicio web (tipos de entrada y de salida). Luego se define la forma de los mensajes y los tipos de puertos que acepta el servicio (las interfaces). Seguidamente se establece el punto de conexión para el servicio web. Y finalmente se establece el servicio web, con sus puertos y la forma de acceso por Internet.

Algunos productos para el desarrollo de servicios webs son e-Speak *de HP,* Dynamic e-Business *de IBM,* .NET Framework *de Microsoft, y* SunONE *de Sun*

Para mayor información sobre la especificación WSDL puede consultar en la página web del W3C (http://www.w3c.org/TR/wsdl) o también en [Cauldwell et al., 2001] o en [Jewell y Chappell, 2002] (por citar tan sólo tres fuentes).

Aunque hoy día en el mercado existen otros muchos más, aquí sólo hemos destacado aquellos productos de las compañías más importantes que han sido pioneras en el campo de los servicios web y que proporcionan los actuales productos UDDI del mercado —que discutiremos a continuación en el siguiente apartado.

1.8.2. El papel de UDDI en los servicios web

Al mismo tiempo que IBM y Microsoft se unieron para desarrollar una especificación de lenguaje para definir servicios web (el lenguaje WSDL), ambas compañías (junto con Ariba) también crearon otro grupo de trabajo con la idea de desarrollar una especificación de una función de localización y almacenamiento de servicios web. Al grupo de trabajo lo llamaron "Grupo del Proyecto UDDI" (http://UDDI.org). En septiembre del año 2000 este grupo crea la primera especificación UDDI del mercado, y nueve meses después, en junio de 2001, crea la versión 2.0. En la actualidad existe la versión 3.0, publicada a finales de junio del año 2002. El grupo de trabajo es mantenido por la organización OASIS en su enlace http://www.uddi.org y en el cual colaboran más de 200 organizaciones.

Desde su primera aparición en septiembre de 2000, en el mercado han aparecido muchas implementaciones de la especificación UDDI, básicamente para las versiones 1.0 y 2.0 (actualmente ninguna completa para la versión 3.0). En la tabla 1.10 recogemos algunos productos y compañías que ofrecen una implementación de dicha especificación, junto con una pequeña descripción del producto.

Por otro lado, aunque rápidamente aparecieron en el mercado implementaciones de la especificación UDDI, no ha sido tan rápida la aparición de uno o varios sitios web conocidos, importantes y con la infraestructura necesaria para dar soporte a repositorios de servicios web que sigan la especificación UDDI. Esta laguna la aprovecharon otras compañías, como SalCentral (http://www.salcentral.com), BindingPoint (http://www.bindingpoint.com) y XMethods (http://www.xmethods.com), para crear importantes repositorios de servicios web accesibles por Internet. El inconveniente es que estos repositorios no siguen el modelo de especificación UDDI, y simplemente se limitan a hospedaje de documentos WSDL, utilizando modelos de almacenamiento y consulta propios a cada organización. Hoy día estos repositorios de servicios web siguen siendo un punto de referen-

Producto	Compañía	Descripción
CapeConnect	Cape Software	Es una plataforma para el desarrollo de servicios web y que proporciona soporte para la generación y mantenimiento de repositorios UDDI privados y públicos. Permite generar descripciones de servicios WSDL en Java, EJB y CORBA. Implementa también la especificación SOAP 1.1 para el paso de mensajes en plataformas Microsoft .NET.
GLUE	Mind	Hace uso de estándares abiertos como XML, SOAP, WSDL y UDDI para la implementación de sitios y entornos basados en servicios web.
WSDP	Sun	Es una parte de JavaTM Web Services Developer Pack de Sun, y que implementa la especificación completa de UDDI versión 2.
Orbix E2A WSIP	IONA	Es un entorno completo de desarrollo y mantenimiento de servicios webs para la integración de aplicaciones heterogéneas construidas en .NET, Java, J2EE, J2ME y CORBA. Compatible con las especificaciones XML, SOAP, WSDL y UDDI, y utiliza herramientas como JAXR (para el UDDI), JAXM (mensajería en SOAP) y SAML (autenticación y autorización, obligatorio en la versión UDDI 3.0).
WASP	Systinet	WASP (*Web Applications and Services Platform*) es una plataforma para el desarrollo y mantenimiento de servicios webs.
Interstage	Fujitsu	Es una infraestructura para la generación y mantenimiento de servicios web basados en J2EE y UDDI, y soporta el uso de múltiples lenguajes y estándares abiertos, como SOAP, WSDL, RosettaNet y ebXML.
WebSphere UDDI Registry	IBM	Es una implementación de UDDI utilizada para el mantenimiento de servicios web en entornos intranet privados. IBM ofrece el WSG (*Web Services Gateway*) como un componente intermedio (*middleware*) que proporciona soporte para hacer públicos en Internet algunos de los servicios web registrados en una red Intranet (usando WebSphere UDDI Registry).
UDDI4J	IBM	Es una implementación completa y de libre distribución de la especificación UDDI, utilizada para la generación y mantenimiento de repositorios de servicios web privados o públicos.
UDDI Services .NET Server	Microsoft	Es una implementación basada en estándares de la especificación UDDI utilizada para la implantación de repositorios de servicios web de forma privada en Intranet, semi-privada en afiliación de organizaciones, o de forma pública en Internet.

Tabla 1.10: Algunas implementaciones de la especificación UDDI

cia muy importante para los desarrolladores de aplicaciones basadas en componentes web. El más importante quizás sea el sitio SalCentral (`http://www.salcentral.com`), donde en los últimos meses hemos notado un continuo aumento en la publicación (registro) de servicios web desarrollados por importantes compañías.

Pero a finales del año 2002 hemos presenciado la consolidación de las primeras implementaciones y mantenimiento de repositorios web basados en el modelo UDDI. A la infraestructura que soporta y mantiene un repositorio de servicios web UDDI se la conoce con el nombre de UBR (*UDDI Business Registry*). Los repositorios UBR más importantes son precisamente aquellos mantenidos por las compañías pioneras en el modelo UDDI: IBM y Microsoft. Aquellas compañías que mantienen y ofrecen un punto de entrada por Internet a repositorios UBR reciben el nombre de "operadores UDDI" (*UDDI operator node*). En la tabla 1.11 hemos recopilado algunos de los operadores UDDI más importantes.

Un UBR puede ser mantenido de forma común por uno o mas operadores, cada uno de ellos con su propio repositorio. UDDI permite hacer duplicados de un servicio publicado en los repositorios afiliados para mantener así la consistencia del UBR. Por tanto, un servicio publicado en el operador A mas tarde puede ser consultado en el operador B, ya que su

Operador	URL de publicación	URL de búsqueda
HP	`https://uddi.hp.com/publish`	`http://uddi.hp.com/inquire`
IBM	`https://uddi.ibm.com/ubr/publishapi`	`http://uddi.ibm.com/ubr/inquiryapi`
Microsoft	`https://uddi.microsoft.com/publish`	`http://uddi.microsoft.com/inquiry`
NTT	`https://www.uddi.ne.jp/ubr/publishapi`	`http://www.uddi.ne.jp/br/inquiryapi`
SAP	`https://uddi.sap.com/uddi/api/publish`	`http://uddi.sap.com/uddi/api/inquire`
Systinet	`https://www.systinet.com/wasp/uddi/publish`	`http://www.systinet.com/wasp/uddi/inquiry`
Algunos operadores de repositorios UDDI de pruebas		
IBM	`https://uddi.ibm.com/testregistry/publish`	`http://uddi.ibm.com/testregistry/inquiry`
Microsoft	`https://test.uddi.microsoft.com/publish`	`http://test.uddi.microsoft.com/inquiry`
SAP	`https://udditest.sap.com/UDDI/api/publish`	`http://udditest.sap.com/UDDI/api/inquire`

Tabla 1.11: Algunos puntos de acceso a operadores UDDI

repositorio asociado se actualiza con un duplicado automático del servicio publicado en el operador A. Los duplicados no son automáticos, siendo el tiempo mínimo de actualización de 12 horas. Como ejemplo, IBM y Microsoft mantienen de forma conjunta un UBR.

De forma parecida al servicio de mediación de ODP, que se basa en un modelo "exportar/consultar" (*export/query*) para registrar y localizar objetos, UDDI se basa en un modelo "suscribir/publicar/preguntar" (*subscribe/publish/inquiry*). Para la publicación de servicios web, UDDI requiere que un proveedor de servicios tenga que estar suscrito previamente en el operador. Para la consulta no es necesario estar suscrito en el operador UDDI. Por esta razón, en la tabla 1.11 hemos incluido dos columnas: una para el punto de entrada por Internet para hacer publicaciones, y otra para el punto de entrada por Internet para hacer consultas. Además, como podemos observar en las direcciones de publicación (columna 2), los puntos de entrada de publicación de los operadores UDDI requieren una conexión segura `https` con una identificación del proveedor. En los puntos de entrada de consulta, directamente se pueden hacer búsquedas en el repositorio.

En las últimas filas de la tabla 1.11 también hemos incluido algunos puntos de entrada a operadores UDDI que ofrecen repositorios de pruebas, denominados "repositorios UDDI de desarrollo". Esta clase de repositorio es muy útil para los desarrolladores de servicios web para labores de construcción y validación.

Los desarrolladores pueden registrar sus prototipos de servicio en los repositorios UDDI de pruebas para validar sus servicios, siempre de forma privada en su cuenta particular. Cuando estos servicios han sido suficientemente probados y validados, el propio desarrollador los publica luego en otro repositorio denominado "repositorio UDDI de producción". Los servicios publicados en esta clase de repositorio pueden ser consultados por otros desarrolladores o clientes de servicios web.

Seguidamente veremos con más detalle ciertos aspectos de implementación de una interfaz UDDI, y también su comportamiento en las tareas de "publicación/consulta" de registros UDDI. Para finalizar discutiremos algunas limitaciones de UDDI.

1.8.3. Modelo de información de UDDI

El modelo de información de un repositorio UDDI básicamente almacena información acerca de aquellas organizaciones que publican servicios web en el repositorio. Los documentos WSDL, que describen a los servicios web, no se almacenan en el repositorio UDDI; estos permanecen en un sitio web al que apunta el registro publicado. Conceptualmente, una empresa proveedora de servicios web puede publicar o consultar su registro de información de forma similar a los tres tipos de un listín telefónico:

— *Páginas blancas*. Contiene información básica de contacto, como el nombre de la empresa, direcciones postales y de correo electrónico, contactos personales, números de teléfono, y un identificador único. Los identificadores son valores alfabéticos o numéricos que distinguen de forma única a una empresa proveedora. Un identificador se asigna cuando la empresa se suscribe en el operador UDDI. En la tabla 1.12 se muestran algunos ejemplos de identificadores que utilizan algoritmos particulares para la generación de códigos.

— *Páginas amarillas*. Agrupa a las empresas registradas por categorías, en función del identificador asignado y de los tipos de servicios ofertados. Esta opción usa un catálogo de servicios web durante el proceso de almacenamiento y de consulta. Esta clase es muy útil para hacer búsquedas selectivas de servicios web dentro de una o varias categorías seleccionadas, reduciendo el tiempo de respuesta.

— *Páginas verdes*. Contienen información técnica sobre los servicios que una empresa proporciona. Esta información es muy útil para que un objeto cliente de servicio pueda conocer detalles de conexión y de programación de comunicación con el servicio web, pues esta información define cómo invocar al servicio (como vimos en la sección anterior). Las páginas verdes normalmente incluyen referencias a los documentos WSDL, que contienen información sobre cómo interactuar con el servicio web.

Taxonomías de categorización estandarizadas		
Nombre	Nombre tModel	Descripción
NAICS	`ntis-gov:naics:1997`	`http://www.census.gov/epcd/www/naics.html`
UNSPC	`unspsc-1`	`http://www.unspsc.org`
ISO 3166	`iso-ch:3166:1999`	`http://www.din.de/gremien/nas/nabd/iso3166ma`
Otros	`uddi-org:general_keywords`	`http://uddi.org`
Tipos de identificadores soportados		
D-U-N-S	`dnb-com:D-U-N-S`	`http://www.d-u-n-s.com`
Thomas Register	`thomasregister-com:supplierID`	`http://www.thomasregister.com`

Tabla 1.12: Taxonomías de categorización y tipos soportados en UDDI

Como ya hemos adelantado antes, la especificación UDDI requiere que la empresa proveedora de servicios web esté suscrita en el operador donde desea acceder. Cuando se accede por primera vez, UDDI suscribe al proveedor, solicitando su dirección de correo electrónico y una contraseña. Internamente, UDDI genera una clave única siguiendo algún modelo para la generación de claves (ver tabla 1.12), y crea un catálogo (un espacio) para el proveedor, con derechos para modificar, dar de alta y de baja a sus servicios web. Al catálogo creado (una única vez) para una empresa proveedora se le denomina documento `businessEntity`. Dentro, la empresa proveedora podrá incluir nuevos servicios web cuando ésta lo desee, estableciendo previamente una conexión segura (`https`), figura 1.11.

Concretamente, el modelo de información de un registro (catálogo) UDDI está compuesto básicamente por cinco tipos de información: (a) `businessEntity`, información acerca de la empresa; (b) `businessService`, información acerca de los servicios que la empresa ofrece; (c) `bindingTemplate`, información técnica de un servicio; (d) `tModel`, información sobre cómo interactuar con el servicio web; (e) `publisherAssertion`, información acerca de la relación de la empresa con otras empresas.

Un registro UDDI es un documento XML que incluye los cinco tipos de información mencionados. En la figura 1.23 hemos creado un diagrama de bloques con las dependencias entre estos tipos de información. Como vemos, `businessEntity` encapsula a los

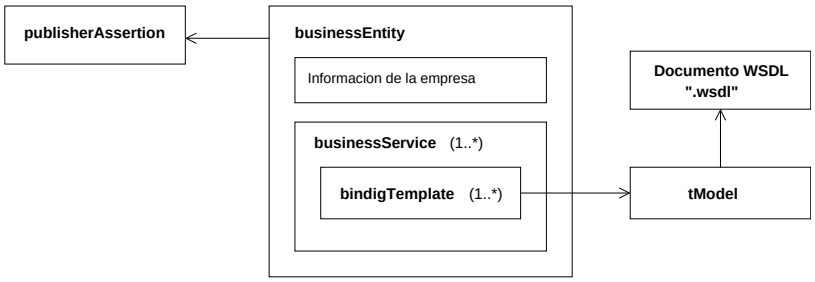

Figura 1.23: Modelo de información de un registro UDDI

demás tipos, salvo `publisherAssertion` y las definiciones WSDL de los servicios web. Cuando la empresa proveedora se suscribe en el operador UDDI, se crea un documento `businessEntity` para ella, con cierta información de empresa (como su nombre o dirección de contacto).

Cuando la empresa proveedora realiza su primer registro de servicio web en UDDI, dentro de `businessEntity` se crea una sección `businessServices`, y dentro de ésta se crea una sección `businessService` por cada servicio web publicado.

Cuando se lleva a cabo el almacenamiento del servicio web en el repositorio UDDI, la información de servicio se desglosa en las secciones técnicas `bindingTemplate` y en un documento XML externo `tModel`. Un documento `tModel` contiene un enlace hacia la definición WSDL del servicio web que describe cómo se hacen las conexiones con las operaciones del servicio. Un documento `tModel` es considerado como un tipo de servicio y que puede ser reutilizado por la empresa proveedora para publicar otros servicios web.

En la figura 1.24 mostramos un meta-modelo de información que hemos creado a partir de la especificación de la estructura de datos de UDDI 3.0 [Bellwood et al., 2002]. Además, para analizar mejor este meta-modelo y algunos conceptos mencionados anteriormente acerca del modelo de información UDDI, en la figura 1.25 mostramos un ejemplo de documento UDDI en XML que se corresponde con una instancia del meta-modelo de la figura 1.24. Para analizar el modelo de información de UDDI haremos referencia de forma conjunta a estas dos figuras (1.24 y 1.25).

En el diagrama se ven las partes de un documento XML `businessEntity`. Cada clase UML modela una etiqueta del documento XML. Hemos utilizado el apartado de atributos de clase para modelar los atributos de una etiqueta. El signo "`-`" significa que el atributo es opcional, y "`+`" significa que el atributo es obligatorio. Por ejemplo, la clase principal `businessEntity` tiene un atributo requerido llamado `businessKey`, que representa la clave única de la empresa proveedora que se le asignó cuando ésta realizó su suscripción en el operador UDDI. Esta clase modela la etiqueta `businessEntity` de un documento UDDI (línea 1 de la figura 1.25).

Como información general, el documento `businessEntity` contiene: el nombre de la empresa proveedora (línea 2), una descripción de la empresa (línea 3) y datos de contacto (líneas 4 a la 11). Esta información se corresponde con la de "páginas blancas".

Los servicios web se registran dentro de la sección `businessServices` (entre las líneas 12 y 34). Dentro de ésta, se utiliza una sección `businessService` para definir cada servicio web publicado por la empresa proveedora (líneas 13 a la 32). Para cada servicio se utilizan dos atributos: uno representa la clave única de la empresa proveedora, y el otro representa la clave única del servicio web publicado. Para este último, la asignación de claves para los servicios publicados siguen el mismo criterio que para la asignación de claves para las empresas proveedoras (la clave `businessKey`).

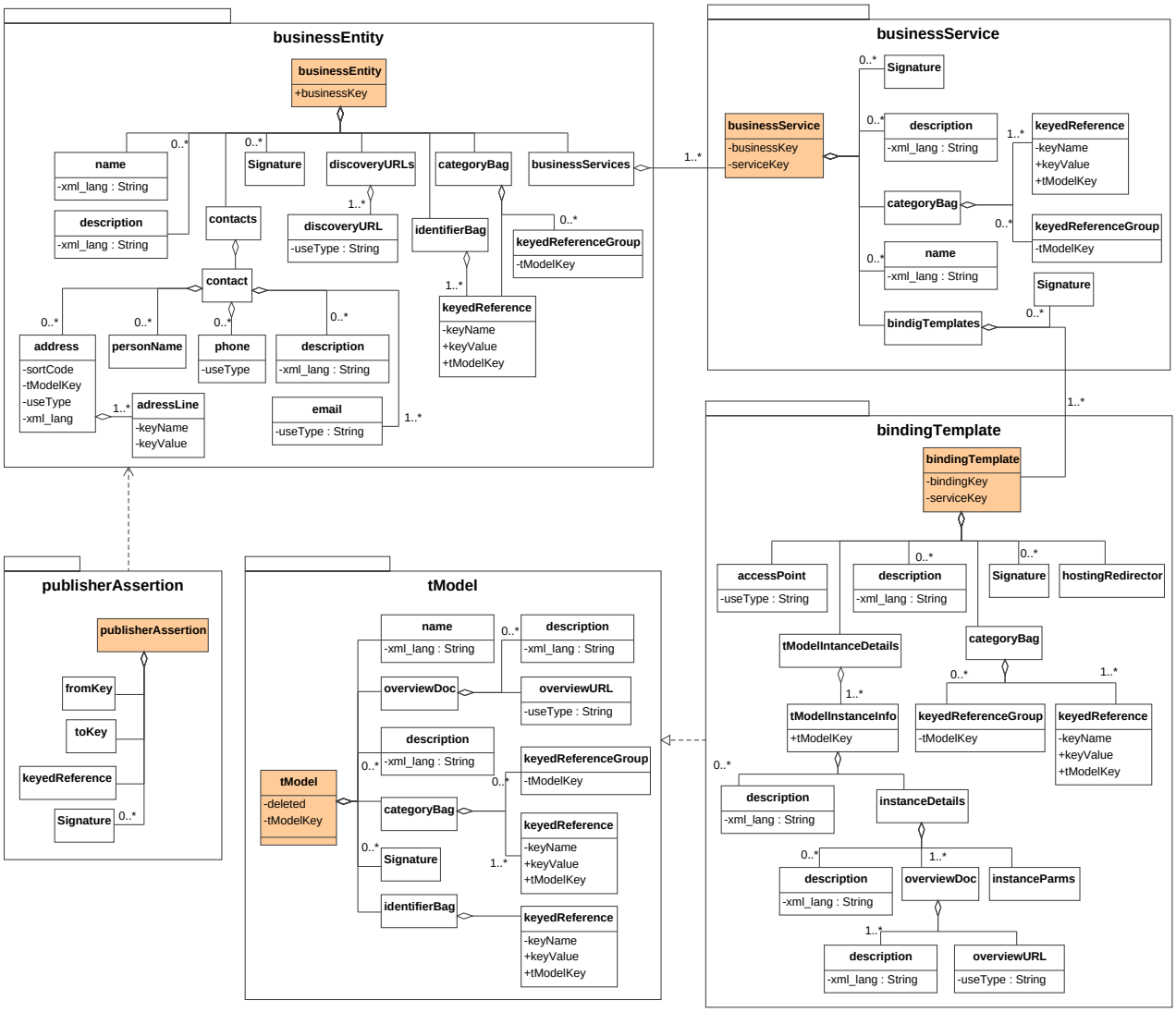

Figura 1.24: Arquitectura interna de un registro UDDI

Para cada servicio definido en la sección `businessService`, se indica: un nombre (línea 15), una descripción del servicio (línea 16) e información de catalogación (entre líneas 17 y 21). Toda esta información se corresponde con la de "páginas amarillas".

Además, una sección de servicio `businessService` puede contener información técnica, recogida en la sección `bindingTemplates` (entre las líneas 22 a la 31). Esta información se corresponde con información de "páginas verdes". La sección `bindingTemplates` contempla información de conexión (línea 25) y un enlace al tipo del servicio `tModel` que está almacenado en otro documento XML dentro del repositorio del operador UDDI (líneas 26 a la 28). La etiqueta `tModelInstanceInfo` contiene un atributo con la clave del tipo de servicio `tModel`.

Dentro de un documento `tModel` existe un puntero hacia la definición WSDL (un archivo "`.wsdl`") que contiene la descripción acerca de cómo se debe invocar las operaciones del servicio, y cómo estas devuelven los datos.

Debido a la extensión del informe de especificación de UDDI 3.0 (el apartado de las estructuras de datos ocupa más de 300 páginas) aquí sólo hemos descrito aquellas

```
<businessEntity businessKey="D2033110-3AAF-11D5-80DC-002035229C64»
   <name>SitioFicticio S.A.</name>
   <description xml:lang="es">Empresa de servicios para correo electrónico</description>
   <contacts>
      <contact>
         <personName>Director técnico</personName>
         <phone>+34-950-123456</phone>
         <email>manager@sitioficticio.es</email>
         <address><addressLine>Ctra. Sacramento s/n 04120</addressLine></address>
      </contact>
   </contacts>
   <businessServices>
      <businessService businessKey="D2033110-3AAF-11D5-80DC-002035229C64"
                       serviceKey="894B5100-3AAF-11D5-80DC-002035229C64">
         <name>Servicio de Identificacion</name>
         <description xml:lang="es">Un servicio de identificacion Internet</description>
         <categoryBag>
           <keyedReference keyName="NAICS: e-commerce" keyValue="..." tModelKey="... "/>
           <keyedReference keyName="NAICS: Database soft." keyValue="..." tModelKey="... "/>
           ...
         </categoryBag>
         <bindingTemplates>
            <bindingTemplate bindingKey="6D8F8DF0-3AAF-11D5-80DC-002035229C64"
                             serviceKey="894B5100-3AAF-11D5-80DC-002035229C64">
               <accessPoint>http://sitioficticio.es/servlet/control.Identificacion</accessPoint>
               <tModelInstanceDetails>
                  <tModelInstanceInfo tModelKey="564B5113-1BAE-21A3-906E-122035229C75"/>
               </tModelInstanceDetails>
            </bindingTemplate>
            ...
         </bindingTemplates>
      </businessService>
      ...
   </businessServices>
</businessEntity>
```

Figura 1.25: Un documento XML como registro UDDI

etiquetas de documento más significativas. Una descripción bastante detallada del modelo de información se puede encontrar en [Bellwood et al., 2002] [Boubez et al., 2002a] [Boubez et al., 2002b].

1.8.4. Las interfaces de UDDI y su comportamiento

UDDI cuenta con diferentes interfaces funcionales que permiten la interacción con los distintos objetos cliente: (a) el objeto que publica servicios web (*publisher*), y (b) el objeto que consulta servicios (*inquiry*). En total, la función UDDI contempla seis interfaces, mostradas en la tabla 1.13, junto con una pequeña descripción de cada una de ellas.

De estas seis interfaces, sólo destacamos `Publisher` e `Inquiry`. En esencia, estas dos interfaces se corresponden con las interfaces `Register` y `Lookup` del servicio de mediación de ODP (véase la sección 1.7).

Las llamadas y las respuestas a todas las operaciones UDDI se realizan utilizando documentos XML. En la tabla 1.14 mostramos los tipos de mensajes de petición y respuesta para las operaciones de las interfaces de publicación y consulta (`Publisher` y `Inquiry`).

Las operaciones de la interfaz de publicación utilizan unos documentos XML como mensajes para almacenar, modificar, borrar o eliminar información de páginas blancas, amarillas o verdes. Las operaciones de la interfaz de consulta también utilizan documentos

Interfaces	Descripción
`Publisher`	Permite hacer registros de servicios web siguiendo el modelo de información `businessEntity`, tratado en la sección anterior.
`Inquiry`	Permite consultar en el repositorio UDDI dentro de los documentos `businessEntity` y `tModel`. Las consultas permiten acceder a la información de páginas blancas, amarillas y verdes, discutidas también en la sección anterior.
`Security`	Son una colección de operaciones para establecer normas de seguridad a la hora de modificar, borrar o eliminar información en el repositorio UDDI. También establece normas de seguridad en la interconexión de afiliados en un UBR (véase página 59).
`Custody`	Contiene tres operaciones para el duplicado de registros UDDI a otros repositorios UDDI de organizaciones afiliadas y que mantienen de forma conjunta un UBR (véase página 59).
`Subscription`	Esta interfaz contiene dos operaciones ligadas a la tarea de suscripción de proveedores dentro del operador UDDI.
`ValueSet`	Esta interfaz también tiene dos operaciones para la configuración del objeto UDDI.

Tabla 1.13: Interfaces y operaciones de la especificación UDDI

Publicación		Consulta	
Petición	Respuesta	Petición	Respuesta
`<add_publisherAssertion>`	`<dispositionReport>`	`<find_binding>`	`<bindingDetail>`
`<delete_binding>`	`<dispositionReport>`	`<find_business>`	`<businessList>`
`<delete_business>`	`<dispositionReport>`	`<find_relatedBusinesses>`	`<relatedBusinessesList>`
`<delete_publisherAssertions>`	`<dispositionReport>`	`<find_service>`	`<serviceList>`
`<delete_service>`	`<dispositionReport>`	`<find_tModel>`	`<tModelList>`
`<delete_tModel>`	`<dispositionReport>`	`<get_bindingDetail>`	`<bindingDetail>`
`<discard_authToken>`	`<dispositionReport>`	`<get_businessDetail>`	`<businessDetail>`
`<get_assertionStatusReport>`	`<assertionStatusReport>`	`<get_serviceDetail>`	`<serviceDetail>`
`<get_authToken>`	`<authToken>`	`<get_tModel>`	`<tModelDetail>`
`<get_publisherAssertions>`	`<publisherAssertions>`		
`<get_registeredInfo>`	`<registeredInfo>`		
`<save_binding>`	`<bindingDetail>`		
`<save_business>`	`<businessDetail>`		
`<save_service>`	`<serviceDetail>`		
`<save_tModel>`	`<tModelDetail>`		
`<set_publisherAssertions>`	`<publisherAssertions>`		

Tabla 1.14: Ordenes de publicación y de consulta en el repositorio UDDI

XML, en este caso para interrogar en los tres tipos de páginas. Los documentos de consulta del estilo `<find_xxx>` extraen una colección de documentos que cumplen con las restricciones impuestas dentro del documento de consulta. La colección de documentos devueltos son encapsulados por una etiqueta global, mostradas en la cuarta columna de la tabla 1.14. Los documentos de consulta del estilo `<get_xxx>` extraen información detallada.

En la tabla 1.15 mostramos algunos ejemplos de uso de algunas operaciones de consulta, junto con los resultados que estas generan. Como vemos en estos ejemplos, podemos ir refinando la búsqueda deseada aplicando documentos de consulta sobre aquellos documentos devueltos en consultas anteriores, limitando cada vez más el espacio de la búsqueda. En los ejemplos de la tabla comenzamos buscando todos los servicios publicados por la empresa proveedora "SitioFicticio S.A.". Luego, sobre el resultado devuelto, queremos conocer si dicha empresa ha publicado un "Servicio de Identificacion". UDDI devuelve sólo una referencia al servicio, para indicar que lo ha encontrado. Finalmente, en la última consulta pedimos a UDDI que obtenga una descripción más detallada.

Consulta 1: busca todos los servicios de una empresa proveedora

```
1:  <find_business»
2:     <name>SitioFicticio S.A.</name>
3:  </find_business>
```

Resultado 1: la lista de servicios

```
4:  <businessList>
5:     <businessInfos>
6:        <businessInfo businessKey="D2033110-3AAF-11D5-80DC-002035229C64">
7:           <name>SitioFicticio S.A.</name>
8:           <description xml:lang="es»Empresa de servicios para correo electrónico</description>
9:           <serviceInfos>
10:             <serviceInfo businessKey="D2033110-3AAF-11D5-80DC-002035229C64"
11:                          serviceKey="894B5100-3AAF-11D5-80DC-002035229C64">
12:                <name>Servicio de Identificacion</name>
13:             </serviceInfo>
14:             ...
15:          </serviceInfos>
16:       </businessInfo>
17:    <businessInfos>
18: </businessList>
```

Consulta 2: ahora busca un servicio concretro dentro de la lista de servicios

```
19:  <find_service businessKey="D2033110-3AAF-11D5-80DC-002035229C64">
20:     <name>Servicio de Identificacion</name>
21:  </find_service>
```

Resultado 2: una referencia al servicio buscado

```
22:  <serviceList>
23:     <serviceInfos>
24:        <serviceInfo businessKey="D2033110-3AAF-11D5-80DC-002035229C64"
25:                     serviceKey="894B5100-3AAF-11D5-80DC-002035229C64">
26:           <name>Servicio de Identificacion</name>
27:        </serviceInfo>
28:     </serviceInfos>
29:  </serviceList>
```

Consulta 3: busca información más detallada del servicio

```
30:  <get_serviceDetail>
31:     <serviceKey>894B5100-3AAF-11D5-80DC-002035229C64</serviceKey>
32:  </get_serviceDetail>
```

Resultado 3: la información de servicio

```
33:  <serviceDetail>
34:     <businessService businessKey="D2033110-3AAF-11D5-80DC-002035229C64">
35:        Véase lineas 13 a la 32 de la figura 1.25
36:     </businessService>
37:  </serviceDetail>
```

Tabla 1.15: Diferentes ejemplos de consulta en UDDI

1.8.5. Limitaciones de UDDI

Hasta aquí, hemos analizado dos funciones de objetos, un referida al servicio de mediación de objetos de ODP, y la otra referida al servicio de directorios para la publicación y localización de servicios web de UDDI. Como vimos, el servicio de mediación de ODP soporta funciones para el registro, consulta y mantenimiento de especificaciones de objetos localizados de forma distribuida y siguiendo un modelo de objetos. Para el caso del servicio UDDI, éste también soporta funciones para la publicación[22] (registro), consulta y mantenimiento de especificaciones de objetos que funcionan y son accesibles de forma distribuida por Internet y siguiendo el modelo de conexión SOAP.

Al igual que hicimos para el servicio de mediación de ODP, vamos a dedicar una sección (la actual) para destacar las limitaciones que presenta el servicio UDDI para el caso de los componentes COTS. Estas limitaciones han sido detectadas en base a la expe-

[22]Utilizamos el término publicación UDDI para referirnos a registro.

riencia obtenida tras analizar la especificación UDDI 3.0 [Bellwood et al., 2002] y algunas de las implementaciones industriales existentes para el servicio UDDI, como la API UDDI4J de IBM (`http://www-124.ibm.com/developerworks/oss/uddi4j`), los *UDDI Business Registry* de Microsoft (`http://uddi.ibm.com`), de IBM (`http://uddi.ibm.com`) y de HP (`http://uddi.hp.com`). Además, también hemos analizado otros repositorios de servicios web que no siguen el modelo UDDI, como el repositorio de SalCentral (`http://www.salcentral.com`) y el repositorio de XMethods (`http://www.xmethods.com`).

(a) *Protocolos*. WSDL utiliza el término protocolo para referirse al protocolo de transporte (HTTP, MIME, FTP, SMTP). Sin embargo, el esquema XML de WSDL no contempla una notación para referirse al concepto de protocolo (o coreografía) en el sentido de conocer el orden en el que se llaman a las operaciones de entrada y de salida. Por tanto, el lenguaje debe permitir incluir directamente (en XML) o indirectamente (un puntero externo) notación para la definición de protocolos: por ejemplo, un protocolo definido en SDL o π-cálculo.

(b) *Comportamiento*. Es necesario que el lenguaje permita definir también el comportamiento semántico de las operaciones (pre/post condiciones e invariantes), y no solo la definición sintáctica de las signaturas usando etiquetas XML. Por tanto, como antes, el lenguaje debe permitir incluir directamente (en XML) o indirectamente (un puntero externo) notación semántica: por ejemplo, una definición en Larch.

(c) *Información extra-funcional*. Por último, también es necesario que el lenguaje pueda utilizar una notación para definir propiedades extra-funcionales, además de la información funcional (sintaxis, semántica y protocolos de las interfaces) y la información técnica del servicio. De nuevo, debe permitir incluir directa o indirectamente esta clase de información: por ejemplo, propiedades de la forma nombre, tipo, valor.

Como hemos adelantado antes, todas estas limitaciones del lenguaje WSDL, de alguna forma, también repercuten como limitaciones en el modelo UDDI, y básicamente afectan a las operaciones de búsqueda en el repositorio. Por lo tanto, una primera limitación de UDDI es que las operaciones de búsqueda estan sujetas sólo a aspectos técnicos de una empresa proveedora de servicios web, a aspectos técnicos de los propios servicios web, y también aspectos de localización, conexión y comunicación de las operaciones de un servicio web. Sin embargo, estas operaciones no permiten búsquedas sobre: (a) los protocolos de interacción de las operaciones, (b) el comportamiento de las operaciones; y (c) información extra-funcional.

En segundo lugar, aunque la especificación UDDI permite el concepto de afiliación (`publisherAssertion`), esto no contempla realmente la posibilidad de federación de repositorios UDDI (como lo hace el servicio de mediación de ODP). Como vimos, la afiliación de operadores UDDI permite mantener la consistencia de la información de un único repositorio virtual UBR (*UDDI Business Registry*). Cada vez que se publica un servicio en el repositorio de un operador UDDI, éste (el servicio) se duplica en los repositorios afiliados al UBR. Sin embargo, un operador filial puede estar asociado a más de un UBR y esto puede acarrear ciertos problemas.

Por ejemplo, supongamos la existencia de dos UBRs (*UBR1* y *UBR2*). En *UBR1* están afiliados los operadores *A* y *B*, y en *UBR2* están afiliados los operadores *B* y *C*. Como vemos, el operador *B* está afiliado a los dos UBRs. Según esto, la publicación de un servicio en *UBR1* no implica que éste (el servicio) sea también publicado en el *UBR2* mediante un duplicado (y viceversa); aun estando el operador *B* como filial en las dos UBRs.

Además, tampoco está permitida la propagación de consultas entre los operadores de un UBR (la consulta se realiza sobre el operador y no sobre el UBR). Lo único que se propaga (como hemos visto) un duplicado del servicio publicado en un operador hacia sus operadores afiliados. Siguiendo con el ejemplo, si suponemos que las publicaciones se llevan a cabo con más regularidad sobre el *UBR1*, un cliente X de servicios web podrá hacer consultas indistintamente desde el operador A o desde el operador B, aun desconociendo éste que los dos operadores son afiliados, y que (en principio) deberían tener los mismos servicios registrados. Sin embargo, las consultas que realice otro cliente Y en el operador C, sólo se limitan al repositorio de dicho operador, y no se propagan a B, supuestamente con más información.

En este sentido, UDDI ofrece un útil y completo servicio de directorio, pero no un servicio de mediación. Un servicio de mediación puede ser considerado como un servicio de directorio avanzado que permite búsquedas basadas en atributos, como por ejemplo atributos de calidad [Kutvonen, 1995].

1.9. Resumen y conclusiones del capítulo

En este capítulo hemos visto, de forma global, el estado en el que se encuentran aquellas áreas de la ingeniería del software que sustentan un modelo de mediación para componentes COTS, presentado en esta memoria en los siguientes capítulos.

Concretamente, las áreas de la ingeniería del software estudiadas han sido (y en este orden) la especificación y documentación de componentes software, los componentes comerciales (COTS, *Commercial Off-The-Shelf*), la ingeniería de requisitos, las arquitecturas de software, la función de mediación de ODP, y la función UDDI para la publicación y localización de servicios web. En definitiva, el área de interés de la ingeniería del software en la que nos centramos es en el desarrollo de sistemas de software basados en componentes COTS, generalmente más conocido como Desarrollo Basado en Componentes (DSBC).

En DSBC los desarrolladores hacen uso múltiples prácticas de ingeniería para construir sus sistemas de software a partir de componentes. Estas prácticas de ingeniería se utilizan en distintas fases del desarrollo y que afectan a (1) la planificación, (2) los requisitos, (3) la arquitectura, (4) la selección y uso de estándares, (5) la reutilización de componentes software, (6) la selección y evaluación de componentes COTS, y (7) la evolución del sistema.

Hoy día los componentes comerciales (COTS) están presentes en diversos campos de la informática, como las bases de datos, los sistemas de mensajería, generadores de interfaces gráficas, software ofimático (procesadores de textos, hojas de cálculo, agendas, etc.); aunque también están siendo utilizados para construir sistemas complejos y de misión crítica, como en sistemas de control para el tráfico aéreo y rodado, o en sistemas médicos para la toma de decisiones.

En el presente capítulo hemos tratado de justificar la necesidad de un servicio de mediación para componentes COTS que será el objeto de la presente memoria. Además de estudiar la función de mediación de ODP, adoptada por OMG (y un estándar de ISO/ITU-T) para objetos distribuidos CORBA, y la función de directorio UDDI, que funciona para servicios web en Internet, junto con algunas de sus implementaciones más conocidas, también hemos analizado algunas de las limitaciones del actual servicio de mediación de ODP que son necesarias a la hora de desarrollar aplicaciones de software a partir de componentes COTS.

Capítulo 2

Un Modelo de Documentación de Componentes COTS

Capítulo 2

Un modelo de documentación de componentes COTS

Contenidos

Un proceso de desarrollo de software basado en componentes (DSBC) se caracteriza, entre otros factores, por seguir un enfoque de desarrollo ascendente (*bottom-up*) para la construcción de sistemas de software. En una visión ascendente, existen actividades que utilizan técnicas para la reutilización de unos componentes software ya desarrollados, y que residen en librerías o repositorios conocidos. Estas actividades del DSBC están fuertemente condicionadas por la presunción de que los componentes software reutilizables están bien definidos, y por la presunción de que lo que se almacena en los repositorios de componentes software son las especificaciones de las interfaces, similares a las APIs.

Hay otras actividades del DSBC que trabajan directamente con la noción de especificación de componente, como en las actividades de diseño de la arquitectura de software de la aplicación (a desarrollar). En este caso, la arquitectura de software se define como un conjunto de especificaciones de componentes abstractos y sus interconexiones. Estas especificaciones abstractas de los componentes (a nivel arquitectónico) son las que luego se utilizan en las actividades de "búsqueda y selección" para localizar especificaciones de componente concretas (aquellas que residen en los repositorios de componentes reutilizables) que son similares. La similitud entre las dos especificaciones, la abstracta y la concreta, puede estar condicionada a aspectos funcionales, como interfaces, comportamiento, eventos o protocolos, o a aspectos extra-funcionales, como atributos de calidad, NFRs.

Para el caso de un proceso de desarrollo de software basado en componentes COTS (DSBC-COTS), la utilización de las actividades tradicionales del DSBC se ven dificultadas por la ausencia de especificaciones para los componentes COTS, uno de los inconvenientes más importantes de los componentes comerciales. Esta ausencia de especificaciones en los componentes COTS se debe, en parte, a la falta de un modelo unificado de especificación que ayude a los fabricantes de componentes COTS en las labores de documentación de sus productos. En el presente capítulo presentamos un modelo para la documentación de componentes comerciales. A los documentos generados a partir de este modelo los denominamos "documentos COTS", la base sobre la que se sustenta el modelo de mediación de componentes COTS, presentado en el *Capítulo 3*.

El presente capítulo se desglosa en ocho secciones. En la sección 2.1 se analiza y justifica más detalladamente la necesidad de un modelo de documentación para componentes COTS. Seguidamente, en la sección 2.2 se presenta la estructura de un documento COTS, sobre la cual se centran las secciones siguientes. Estas secciones cubren aspectos sobre cómo recoger la información de un componente comercial, como la información funcional (sección 2.3), información extra-funcional (sección 2.4), información de implementación e implantación (sección 2.5) e información de contacto y no técnica del vendedor y del componente (sección 2.6). Al final del capítulo, en la sección 2.7, se discuten algunos trabajos relacionados con la propuesta del modelo de documentación de componentes COTS que aquí se presenta. Se concluye con un breve resumen y algunas conclusiones.

2.1. Introducción y conceptos relacionados

Como se puso de manifiesto en el *Capítulo 1*, el proceso de desarrollo de software basado en componentes se centra en la construcción de aplicaciones a partir de software reutilizable, conocido en la forma de "componente software". Por regla general, las prácticas de reutilización de software afectan a librerías o repositorios de componentes que han sido desarrollados dentro de la organización. En algunos casos, puede que estas librerías o repositorios de componentes hayan sido elaborados por terceras partes, ligadas o relacionadas

a la propia organización. En cualquier caso, es seguro que estos componentes disponen de especificaciones adecuadas y particulares a la organización (u organizaciones) que las ha(n) construido (esto es, no estandarizadas). La disponibilidad de estas especificaciones de componentes (en librerías o repositorios conocidos) permite la práctica de actividades basadas en algoritmos o técnicas de reutilización que localizan aquellos componentes software que pueden ser reutilizados para la construcción de la aplicación objeto, comparando la similitud de las especificaciones de estos componentes reutilizables con la de los componentes descritos en fase de diseño en una arquitectura de software.

Es sabido que algunas de las actividades que intervienen en un proceso DSBC, como las actividades de diseño de la arquitectura de software, la búsqueda y selección de componentes software o la actividad de validación de componentes, entre otras, presuponen la existencia de especificaciones de componente (ya conocidas) sobre las que opera cada una de estas actividades. Decimos que una especificación de componente es conocida cuando un actor[1] es capaz de conocer la estructura interna de la especificación (su tipo de información), independientemente de su contenido (la información).

Para el caso de los componentes comerciales (o componentes COTS), las actuales actividades del DSBC son difíciles de llevar a la práctica. Como vimos, los componentes comerciales son un caso particular de reutilización de componente software. Este tipo de componente se caracteriza por ser un software desarrollado por terceras partes, fuera de la organización y que carece de información de especificación. La información de un componente comercial se suele ofrecer en forma de ficha técnica —generalmente desde una página web— o es facilitada por el vendedor a petición de la persona que lo necesite. En el mejor de los supuestos, la información que ofrece un fabricante de componentes COTS suele ser información de contacto o, a lo sumo, información con características técnicas del componente, como el tipo de sistema operativo o procesador donde puede funcionar, o los requisitos de instalación del componente, o aspectos de licencia, entre otros datos.

Esta información de componente comercial (cuando está disponible) suele ser bastante limitada para las pretensiones reales de las actividades del DSBC. En la literatura sobre especificación de componentes software (en general) tampoco existe un criterio unificado que establezca cómo y qué clase de información se debe definir en una especificación de componente software. Con el tiempo, parece que se está consolidando caracterizaciones de componente software como la de Jun Han [Han, 1998], en la cual se considera que una especificación de componente pueda contener la siguiente información:

(a) *información sintáctica* de los atributos, las operaciones (o métodos) y los eventos de las interfaces de un componente;

(b) *información semántica* acerca del comportamiento de estos operadores;

(c) *información de protocolos*, que determina la interoperabilidad del componente con otros componentes;

(d) y un conjunto de *propiedades extra-funcionales* del componente, como propiedades de seguridad, fiabilidad o rendimiento, entre otras.

Sin embargo, los componentes COTS carecen de especificaciones con una clase de información similar a esta.

[1] Con el término actor queremos representar indistintamente si la actividad se lleva a cabo por un proceso automático —un programa— o una persona.

Por otro lado, tal y como adelantamos en el *Prólogo* de esta memoria, uno de nuestros objetivos ha sido la elaboración de un proceso de DSBC-COTS que estuviera soportado por un servicio de mediación automatizado para componentes COTS. El proceso de DSBC-COTS finalmente elaborado, ha sido un método de desarrollo basado en un modelo en espiral que combina metodologías y técnicas de ingeniería del software. Este método está compuesto por las siguientes actividades:

En el Capítulo 5 *veremos una estrategia DSBC-COTS basada en un modelo en espiral que integra las propuestas que discutiremos en éste y próximos capítulos*

(a) definición de la *arquitectura de software*, para la especificación de los requisitos de los componentes de la aplicación que hay que construir y de sus interconexiones;

(b) *servicio de mediación*, para la búsqueda y selección de aquellas especificaciones de componentes COTS (ya existentes) que puedan satisfacer, total o parcialmente, y de forma individual, las necesidades de un componente definido en la arquitectura de software; y

(c) *generación de configuraciones*, que son combinaciones de especificaciones de componentes (localizadas en la actividad anterior) que satisfacen, total o parcialmente, y en su conjunto, a las restricciones arquitectónicas de los componentes definidas en la arquitectura de software (primera actividad).

El denominador común a las tres actividades básicas —y de cualquier otro método DSBC— es el uso de especificaciones de componentes expresadas en alguna notación. Dada la falta de un mecanismo unificado y reconocido para la especificación de componentes comerciales, en nuestro caso hemos elaborado un modelo para la documentación de componentes COTS inspirado en las propuestas originales (discutidas en el *Capítulo 1*, en la sección *Especificación de Componentes Software*).

En las siguientes secciones describiremos el modelo de documentación de componentes COTS desarrollado. Durante la exposición, haremos uso de una definición semi-formal en notación XML-Schema. La nomenclatura introducida por estas definiciones serán de utilidad para exponer algunos de los algoritmos presentados en los *Capítulos 3* y *4*. Por otro lado, puesto que el modelo de documentación está soportado por un lenguaje para "escribir" plantillas en XML —las cuales recogen la información de un componente comercial— la expresividad que ofrece una definición en notación XML-Schema ayudará a entender mejor la estructura gramatical de un documento en XML. Junto a estas definiciones, ofreceremos también una instancia XML ejemplo de la definición de esquema que se esté tratando en ese momento. Para las instancias ejemplo utilizaremos de base un simple componente buffer, ya introducido en el *Capítulo 1*. No obstante, dedicaremos una pequeña sección para recordar las características de este componente ejemplo. Veamos pues la descripción del modelo de documentación.

2.2. Documentos COTS (COTScomponent)

Tradicionalmente, una especificación de componente software puede quedar definida por medio de sus interfaces, que describen las operaciones, atributos, parámetros y otra información sintáctica. No obstante, como vimos en el *Capítulo 1*, esta clase de información a nivel sintáctico no es suficiente para construir aplicaciones software. También es necesaria otra clase de información adicional, como la información de los *protocolos* (que describen la interacción del componente), los *eventos* que el componente emite y consume, o la información *semántica* (que describe el comportamiento de las operaciones del componente). En conjunto, a toda esta clase de información (sintáctica, semántica y de protocolos) se

la conoce con el nombre de información funcional, ya que hace referencia a aspectos de funcionamiento y de computación del componente.

Por otro lado, en un proceso de desarrollo basado en COTS es necesario tener en cuenta otra clase de información (además de la funcional) y que juega un papel muy importante en actividades como la selección y evaluación de componentes. Nos estamos refiriendo a aquella información que no tiene que ver con aspectos funcionales de las interfaces de un componente, como la información de atributos de calidad, atributos no funcionales, información de implementación e implantación del componente, información no técnica del componente o información de contacto del fabricante.

Sobre esta base, el modelo de documentación de componentes COTS elaborado intenta recoger esta variedad de información en una especificación de componente comercial que denominaremos "documento COTS".

2.2.1. Un ejemplo

Para exponer el modelo de documentación vamos a recurrir a un ejemplo que fue introducido en el *Capítulo 1*, en la sección *Especificación de Componentes Software*. El ejemplo se trata de un componente buffer con una sola celda llamado `OnePlaceBuffer`, con las operaciones usuales `read()` y `write()`. El componente `OnePlaceBuffer` utiliza otro componente para imprimir los valores que se escriben en su celda, utilizando el método `print()` cada vez que se escribe un nuevo valor. La figura 2.1 muestra un esquema de la estructura de componentes al cual nos estamos referimos. Este ejemplo y todos los descritos en esta memoria están disponibles en el sitio web `http://www.cotstrader.com`.

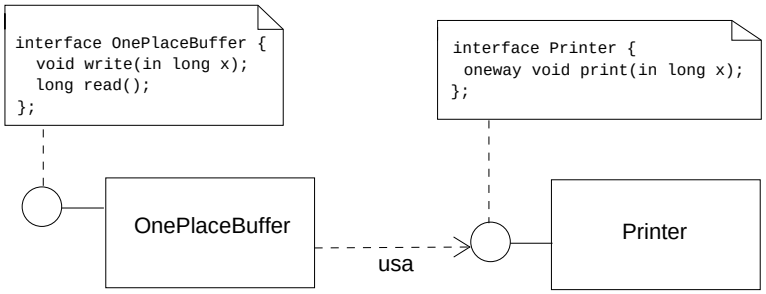

Figura 2.1: Ejemplo utilizado para ilustrar un documento COTS

2.2.2. Plantilla de documento

El modelo de documentación de componentes COTS está soportado por un lenguaje que permite definir documentos COTS. El lenguaje ha sido diseñado primero utilizando la notación BNF, y más tarde traducido a un esquema gramatical en XML, utilizando la notación XML-Schemas del W3C (`http://www.w3.org/2000/10/XMLSchema`). Al final de esta memoria, en el *Apéndice A*, ofrecemos la definición completa del lenguaje para escribir documentos COTS en ambas notaciones. El esquema gramatical determina cómo se debe escribir un documento COTS del modelo en una plantilla (*template*) XML para recoger la especificación de un componente comercial, elemento básico para todas las transacciones en las actividades del DSBC-COTS. A esta clase de plantilla la denominamos plantilla COTScomponent o documento COTS.

En la figura 2.2 se muestra la definición de esquema de un documento COTS, y en la figura 2.3 se muestra el esqueleto de una instancia de plantilla de un documento COTS,

correspondiente al componente `OnePlaceBuffer`.

Documento COTS: <COTScomponent>

```
<xsd:element name="COTScomponent" type="exportingType"/>
<xsd:complexType name="exportingType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:sequence>
    <xsd:element name="functional" type="functionalType"/>
    <xsd:element name="properties" type="propertiesType" minOccurs="0"/>
    <xsd:element name="packaging" type="packagingType" minOccurs="0"/>
    <xsd:element name="marketing" type="marketingType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 2.2: Definición de esquema de un documento COTS

```
1: <COTScomponent name="OnePlaceBuffer"
2:                xmlns="http://www.cotstrader.com/COTS-XMLSchema.xsd"
3:                xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
4:   <functional> ... </functional>
5:   <properties> ... </properties>
6:   <packaging> ... </packaging>
7:   <marketing> ... </marketing>
8: </COTScomponent>
```

Figura 2.3: Una instancia ejemplo de un documento COTS

Como se puede comprobar, una plantilla de documento COTS comienza por el elemento `COTScomponent`, y dentro de éste se definen sus cuatro partes. Además, el elemento principal `COTScomponent` también puede contener uno o más atributos, siendo `name` el único atributo obligatorio utilizado para recoger el nombre del componente que se está documentando.

Los dos atributos `xmlns` que se ven junto al elemento `COTScomponent` (en el ejemplo de la figura 2.3) no son obligatorios en un documento COTS. Un atributo `xmlns` se utiliza para establecer espacios de nombres adicionales[2] en una plantilla XML y, según la especificación de XML-Schema del W3C, puede ser usado en cualquier elemento XML sin necesidad de definirlo en su esquema gramatical (esta es la razón por la que no aparece definido el atributo `xmlns` en el fragmento de esquema gramatical de un elemento `COTScomponent`, ofrecido anteriormente en la figura 2.2).

De estos dos espacios de nombres, el primero de ellos se utiliza para establecer el lugar donde reside la gramática del lenguaje de un documento COTS, y que por omisión se encuentra disponible en el sitio web `http://www.cotstrader.com/`. `COTS-XMLSchema.xsd` es el nombre del archivo que contiene la definición de todos los elementos y atributos necesarios para escribir un documento COTS, como los elementos `functional`, `properties`, `packaging` o `marketing`, entre otros que iremos descubriendo en las siguientes secciones (el *Apéndice A* de esta memoria contiene el esquema completo de `COTS-XMLSchema`).

El segundo espacio de nombres permite utilizar los tipos de datos definidos en la gramática base `XMLSchema` del W3C. Para el caso de la plantilla ejemplo, mostrada en

[2]En XML-Schemas, un espacio de nombres permite hacer extensiones de un lenguaje mediante un enlace en la cláusula `xmlns` hacia el archivo que contiene la extensión escrita en notación XML-Schemas. Similar a una cláusula `import` en el lenguaje Java.

la figura 2.3, el nombre asignado al espacio de nombres con los tipos base ha sido `xsd`, normalmente usado en el área de los esquemas para este fin (aunque se podría haber usado cualquier otro nombre). De esta forma, para referirse a un tipo base de un elemento de la gramática se hace de la forma `xsd:int`, `xsd:float` o `xsd:bool` (entre otros), estando los tipos base `int`, `float` y `bool` definidos en notación XML-Schemas dentro del sitio donde apunta el espacio de nombres `xmlns:xsd`. Por omisión, el modelo utiliza los tipos base del W3C (ubicados en `http://www.w3.org/2000/10/XMLSchema`)[3].

En las siguientes secciones discutiremos las cuatro partes de un documento COTS: funcional, extra-funcional, empaquetamiento y marketing.

2.3. Descripción funcional <functional>

Una descripción funcional de un componente recoge información de los requisitos funcionales a nivel sintáctico y semántico de las interfaces proporcionadas y requeridas por el componente. Una descripción funcional también cubre aspectos relacionados con los eventos producidos o consumidos por el componente, y también aspectos de los protocolos de interacción de las interfaces del componente con otros componentes.

Siguiendo las pautas adoptadas para la exposición del modelo de documentación COTS, definamos en primer lugar el significado de "descripción funcional". Para la parte funcional de un documento COTS, el modelo establece con carácter obligatorio la presencia de al menos una interfaz proporcionada por el componente. Tanto las interfaces requeridas por el componente, como los eventos producidos y consumidos por éste y sus protocolos de interacción, pueden no aparecer en un documento COTS (debido al carácter opcional en su definición, `minOccurs="0"`).

En la figura 2.4 se muestra la definición (en notación XML-Schemas) de la parte funcional de un documento COTS. La parte funcional queda especificada por el elemento `functional`, que es, a su vez, una parte del elemento `COTScomponent`. En la figura 2.5 se muestra un fragmento de plantilla XML de documento COTS, correspondiente al componente `OnePlaceBuffer`, con las cinco partes de una descripción funcional (como instancia del esquema). Como se observa, las interfaces se recogen en los elementos `<providedInterfaces>` y `<requiredInterfaces>` mediante el elemento `<interface>` (que se tratará en la sección 2.3.1).

En el esquema, los elementos `<providedInterfaces>` y `<requiredInterfaces>` se definen del tipo `InterfaceList` (figura 2.6) usado para especificar una lista de interfaces:

[3]En el caso de la ausencia de estos dos espacios de nombres en un documento COTS, el modelo utiliza los valores establecidos por omisión para cada uno de ellos: (a) `http://www.cotstrader.com/COTS-XMLSchema.xsd`, para localizar los elementos de la gramática, y (b) `http://www.w3.org/2000/10/XMLSchema`, para localizar los tipos de datos usados en la plantilla XML. Aunque en un principio esto podría parecer poco importante, no es así para el modelo de mediación que presentamos en el *Capítulo 3*. El servicio de mediación —implementado a partir del modelo propuesto— realiza dos operaciones de análisis sobre un documento COTS antes de registrarlo en el repositorio asociado. Por un lado se lleva a cabo un análisis sintáctico sobre el documento COTS, que comprueba si dicho documento sintácticamente es un documento XML. Esto se hace, a nivel de programación, llamando directamente a una orden de la API que se utilice para tratar XML (en nuestro caso ha sido la API XML4J de IBM). En segundo lugar se lleva a cabo un análisis a nivel semántico o gramatical, donde se comprueba si la plantilla de documento COTS es un tipo de documento `COTScomponent` establecido por la gramática `http://www.cotstrader.com/COTS-XMLSchema.xsd`. Esta operación no está disponible en la API XML4J de IBM, ya que los analizadores actuales sólo realizan análisis sintácticos. Para ello se ha utilizado un operador de análisis semántico de esquemas, desarrollado por nosotros en el año 2002 (`http://www.cotstrader.es/resources/COTS-XMLSchemaParser.html`). Durante ese mismo año, el W3C ha recibido varias propuestas de analizadores gramaticales similares al desarrollado.

Descripción Funcional: <functional>

```
<xsd:element name="functional" type="functionalType"/>
<xsd:complexType name="functionalType">
  <xsd:sequence>
    <xsd:element name="providedInterfaces" type="InterfaceList"/>
    <xsd:element name="requiredInterfaces" type="InterfaceList" minOccurs="0"/>
    <xsd:element name="consumedEvents" type="eventType" minOccurs="0"/>
    <xsd:element name="producedEvents" type="eventType" minOccurs="0"/>
    <xsd:element name="choreography" type="behaviorType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complezType>
```

Figura 2.4: Definición de esquema de la descripción funcional de un documento COTS

1: <**functional**>
2: <**providedInterfaces**>
3: <**interface** name="P_1">...</**interface**> ... <**interface** name="P_m">...</**interface**>
4: </**providedInterfaces**>
5: <**requiredInterfaces**>
6: <**interface** name="R_1">...</**interface**> ... <**interface** name="R_n">...</**interface**>
7: </**requiredInterfaces**>
8: <**consumedEvents**> ... </**consumedEvents**>
9: <**producedEvents**> ... </**producedEvents**>
10: <**choreography**> ... </**choreography**>
11: </**functional**>

Figura 2.5: Una instancia ejemplo de la descripción funcional de un documento COTS

en este caso son las colecciones de interfaces proporcionadas y requeridas por el componente. El tipo `InterfaceList` encapsula un elemento `interface` para definir una interfaz.

```
<xsd:complexType name="InterfaceList">
  <xsd:sequence>
    <xsd:element name="interface" type="InterfaceType" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 2.6: Definición de esquema para el tipo `InterfaceList`

Para definir los eventos de un componente se utilizan los elementos `<consumedEvents>` y `<producedEvents>`, los cuales recogen las colecciones de eventos consumidos y generados por el componente, respectivamente. Ambos elementos son opcionales (`minOccurs="0"`) y del tipo `eventType`, el cual permite definir una lista de nombres de eventos. El tipo `eventType` se trata en la sección 2.3.3.

Por último, el protocolo del componente se corresponde con el elemento `choreography`. Como se ha adelantado, la presencia de este elemento dentro de un documento COTS también es opcional, y es del tipo `"behaviorType"`[4], tratado más tarde en la sección 2.3.2.

[4]Por su estructura, el tipo `behaviorType` (sección 2.3.2) ha sido utilizado para definir otros elementos del modelo de documentación, como el elemento `behavior` o `choreography`.

2.3.1. Interfaces

Véase Capítulo 1, *sección 1.3.1*

El modelo de documentación de componentes COTS contempla una interfaz de componente como una colección de información que está relacionada con aspectos sintácticos y semánticos de la interfaz. Los aspectos sintácticos se refieren a la forma en la que tradicionalmente se define una interfaz, esto es, con sus atributos, operaciones (o métodos) y eventos, como se hace en los modelos de componentes EJB, CCM o EDOC [OMG, 2001]. Los aspectos semánticos como los contratos (véase *Capítulo 1*, sección 1.3.5), recogen el comportamiento individual de las operaciones de interfaz.

En la figura 2.7 se muestra la definición de esquema de una interfaz de componente de un documento COTS. Como se puede ver, una interfaz se representa mediante el elemento `interface`, con un nombre de interfaz establecido por el atributo `name`. Dentro, el modelo establece que una interfaz pueda contener una secuencia (`<xsd:sequence>`) de cuatro bloques, los tres primeros se corresponden con una descripción sintáctica de la interfaz, y la cuarta para una descripción de comportamiento. Este último bloque, contemplado en el esquema como un elemento `behavior`, será tratado más tarde en la sección 2.3.2. En la figura 2.8 se puede ver un fragmento de plantilla donde se definen las interfaces proporcionadas y requeridas para un componente comercial.

Interfaz de componente (sintaxis): <interface>

```
<xsd:element name="interface" type="InterfaceType" maxOccurs="unbounded"/>
<xsd:complexType name="InterfaceType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:sequence>
    <xsd:element name="description" type="locationType"/>
    <xsd:element name="exactmatching" type="referenceType" minOccurs="0"/>
    <xsd:element name="softmatching" type="referenceType" minOccurs="0"/>
    <xsd:element name="behavior" type="behaviorType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 2.7: Definición de esquema de las interfaces de un documento COTS

Veamos cómo el modelo de documentación de componentes COTS define una descripción sintáctica. El esquema gramatical de una interfaz contempla la descripción sintáctica de una interfaz en un elemento `description` de tipo `locationType`. Este tipo[5] requiere la presencia de un atributo `notation` (ligado a `description`) para indicar el nombre de la notación utilizada en la descripción sintáctica de la interfaz.

Asociada a una descripción sintáctica de interfaz se puede tener (de forma opcional) dos tipos de operadores de emparejamiento sintáctico, uno fuerte o exacto $\leq_E$ (representado en el esquema como `"exactmatching"`), y otro débil o relajado $\leq_S$ (representado en el esquema como `"softmatching"`). Estos dos operadores son los que luego utiliza el modelo de mediación en los procesos de búsqueda y selección en sus labores de reemplazilidad sintáctica de interfaces (el modelo de mediación se discutirá en el *Capítulo 3*).

El operador de emparejamiento-sintáctico exacto entre dos interfaces A y B, escrito de la forma $A \leq_E B$, determina si la interfaz B puede reemplazar a la interfaz A, esto es, si B es un subtipo de A, considerando las relaciones tradicionales de subtipado para signaturas de interfaces [Zaremski y Wing, 1995].

[5]Para no complicar la exposición, hemos preferido no incluir en este punto la descripción de esquema gramatical del tipo `locationType`. No obstante, esta descripción se puede localizar en el *Apéndice A* que hay al final de esta memoria.

```
1:  <functional>
2:    <providedInterfaces>
3:      <interface name="OnePlaceBuffer">
4:        <description notation="CORBA-IDL">
5:          interface OnePlaceBuffer { void write(in long x); long read(); };
6:        </description>
7:        <exactMatching href="http://soleres.ual.es/servlet/CORBA.exact"/>
8:        <softMatching href="http://soleres.ual.es/servlet/CORBA.soft"/>
9:        <behavior>...</behavior>
10:     </interface>
11:   </providedInterfaces>
12:   <requiredInterfaces>
13:     <interface name="Printer">
14:       <description notation="CORBA-IDL">
15:         interface out { oneway void print(in long x); };
16:       </description>
17:     </interface>
18:   </requiredInterfaces>
19: </functional>
```

Figura 2.8: Una instancia ejemplo de las interfaces de un documento COTS

Por otro lado, se utiliza una operación de emparejamiento-sintáctico débil entre dos interfaces para llevar a cabo emparejamientos parciales. A nivel de signaturas, diremos que una interfaz B puede reemplazar parcialmente a otra A (escrito en la forma $A \leq_S B$) cuando la interfaz B "contiene" algunos de los servicios definidos en A, tal que $A \cap B \neq \varnothing$. Intuitivamente, esta relación significa que B ofrece al menos uno de los métodos que también ofrece A. Todo esto se describirá con más detalle en el *Capítulo 4*.

En la figura 2.8 se muestra el esqueleto de una instancia ejemplo de componente COTS con información de interfaces. Para este ejemplo tenemos dos interfaces: una que es ofrecida por el componente (definida entre las líneas 3 y 10) y la otra requerida por el componente (entre las líneas 13 y 17). Tomemos como referencia la interfaz `OnePlaceBuffer` (línea 3).

Como se ha visto, la descripción sintáctica de una interfaz queda identificada por cuatro partes. Las dos primeras se refieren a la descripción de la interfaz y a la notación usada, recogidas en el documento (como se ha dicho) por un elemento `description` y su atributo `notation` (línea 4). Se puede utilizar el elemento `description` para describir otras partes de un documento COTS, como el comportamiento de las operaciones de un componente o los protocolos. En general, una descripción (ya sea de interfaz, de comportamiento o de protocolo) se puede indicar de dos formas[6]: (a) incluyendo la descripción completa de la interfaz directamente dentro de la sección `description` (como se hace en el ejemplo) o (b) mediante un enlace hacia un archivo remoto (p.e., `OnePlaceBuffer.idl`) con el contenido de la descripción de la interfaz, por ejemplo:

```
<description notation="CORBA-IDL"
    href="http://www.cotstrader.com/examples/OPB/OnePlaceBuffer.idl"/>
```

En cualquiera de los dos casos (directa o indirectamente) es necesario indicar la notación en la que viene expresada la descripción, y que para el caso del ejemplo anterior (figura 2.8) ha sido la notación IDL de CORBA (`notation="CORBA-IDL"`).

[6]Establecido en la gramática por el tipo de dato `locationType` (véase *Apéndice A*).

Las otras dos partes de una descripción sintáctica de interfaz se refieren a los dos operadores de emparejamiento discutidos. El operador de emparejamiento sintáctico fuerte ($\leq_E$) se representa por el elemento `exactMatching` (línea 7), y el operador de emparejamiento sintáctico débil ($\leq_S$) se representa mediante el elemento `softMatching` (línea 8). Similar al elemento `description`, estos dos operadores utilizan un enlace `href` hacia el lugar remoto donde se encuentra disponible el programa que realiza las labores de emparejamiento sintáctico entre dos interfaces, uno a nivel fuerte (línea 7) y otro a nivel débil (línea 8) respectivamente. Según el ejemplo, `CORBA.exact` es el nombre del programa remoto que efectúa emparejamientos sintácticos fuertes, y de forma similar, `CORBA.soft` es el nombre del programa usado para emparejamientos a nivel sintáctico y débil.

2.3.2. Comportamiento de las interfaces

Una vez visto cómo el modelo de documentación trata la descripción sintáctica de una interfaz, veamos ahora cómo trata la descripción semántica, o descripción de comportamiento. Expongamos primero el concepto de descripción semántica según el modelo de documentación.

Véase Capítulo 1, *sección 1.3.2*

Similar a como se hace para la descripción sintáctica, el modelo de documentación recoge la descripción semántica de una interfaz mediante una descripción del comportamiento de la interfaz junto con la notación en la que ésta viene expresada. Las notaciones más usuales para recoger el comportamiento de un componente son aquellas basadas en pre-post condiciones, como por ejemplo Larch [Dhara y Leavens, 1996] [Zaremski y Wing, 1997], JML (*Java Modeling Language* o JavaLarch, `ftp://ftp.cs.iastate.edu/pub/leavens/JML`) [Leavens et al., 1999] y OCL (*Object Constraints Language*) [Warmer y Kleppe, 1998]. Asociada a la descripción de comportamiento, el modelo de documentación también admite la presencia (opcional) de dos tipos de operadores de emparejamiento semántico: uno fuerte (o exacto) $\sqsubseteq_E$, y otro débil (o relajado) $\sqsubseteq_S$. Estos dos operadores son los que luego utilizará el modelo de mediación para los procesos de búsqueda y selección en las labores de reemplazabilidad semántica entre interfaces.

El operador de emparejamiento-semántico exacto entre el comportamiento de dos interfaces A y B, escrito de la forma $B \sqsubseteq_E A$, determina si la interfaz B puede reemplazar a la interfaz A en comportamiento, esto es, si el comportamiento de A está en el de B. Esta relación de subtipado del comportamiento de una interfaz puede ser tratada como una extensión de las relaciones de subtipado de las signaturas de interfaz [Zaremski y Wing, 1995] para descripciones semánticas de componentes realizadas en la forma de pre-post condiciones [Leavens y Sitaraman, 2000] [Zaremski y Wing, 1997].

Una operación de emparejamiento-semántico débil entre dos interfaces se utiliza para hacer emparejamientos parciales, esto es, partiendo del comportamiento definido para cada signatura de una interfaz, diremos que, semánticamente hablando, una interfaz A puede reemplazar parcialmente a otra B (escrito en la forma $A \sqsubseteq_S B$) cuando la interfaz A "contiene" algunas de las definiciones de comportamiento de signatura definidas en B, tal que $A \cap B \neq \varnothing$. Por tanto, esta relación significa que el comportamiento de la interfaz B (definido para cada una de las signaturas de la interfaz) ofrece, al menos, una definición de comportamiento para una signatura idénticamente definida (en comportamiento) dentro de la interfaz A. La operación de emparejamiento parcial de comportamiento de interfaces puede ser tratada como una extensión de la operación parcial de subtipado de signaturas de interfaces [Zaremski y Wing, 1995] para descripciones semánticas [Mili et al., 2000]. Obsérvese que, a diferencia de $\sqsubseteq_E$, el operador $\sqsubseteq_S$ no puede definir un pre-orden.

Como en el caso de los operadores sintácticos, el modelo de documentación establece con carácter opcional la presencia de estos dos operadores de emparejamiento semántico

dentro de un documento COTS: pudiendo aparecer los dos, uno de ellos o incluso ninguno.

En la figura 2.9 se muestra el esquema que especifica el comportamiento de una interfaz mediante el tipo `behaviorType`, el cual es utilizado mas tarde para definir también la parte de la coreografía de un componente COTS.

Interfaz de componente (comportamiento): <behavior>

```
<xsd:element name="behavior" type="behaviorType" minOccurs="0"/>
<xsd:complexType name="behaviorType">
  <xsd:sequence>
    <xsd:element name="description" type="locationType"/>
    <xsd:element name="exactmatching" type="referenceType"/>
    <xsd:element name="softmatching" type="referenceType"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 2.9: Definición de esquema del comportamiento de un documento COTS

En la figura 2.10 se muestra un ejemplo donde se puede ver cómo se recoge la descripción semántica de una interfaz dentro de un documento COTS, en este caso expresada en notación Larch-CORBA (`http://www.cs.iastate.edu/~leavens/main.html#LarchCORBA`). Esta notación (como otras muchas) se basa en pre (`requires`) y post (`ensures`) condiciones para hacer la descripción de comportamiento de cada operación de la interfaz. En el ejemplo se hace la descripción semántica para las dos operaciones de la interfaz `OnePlaceBuffer`: una descripción para la operación `write()` (líneas de la 5 a la 8) y otra para la operación `read()` (líneas de la 9 a la 12).

```
1:  <behavior>
2:    <description notation="Larch-CORBA">
3:      interface OnePlaceBuffer {
4:        initially self' = empty;
5:        void write(in long x){
6:           requires isEmpty(self); modifies self;
7:           ensures self' = append(self^,x);
8:        }
9:        long read(){
10:          requires ~isEmpty(self); modifies self;
11:          ensures result = head(self^) /\ isEmpty(self');
12:       }
13:     }
14:   </description>
15:   <exactMatching href="http://soleres.ual.es/servlet/Larch.exact"/>
16:   <softMatching href="http://soleres.ual.es/servlet/Larch.soft"/>
17: </behavior>
```

Figura 2.10: Una instancia ejemplo del comportamiento de un documento COTS

Como vemos en el ejemplo, se utiliza el elemento `behavior` para hacer una descripción de comportamiento de una interfaz dentro de un documento COTS. Este elemento contiene la descripción de comportamiento en la forma establecida por la definición de esquema de la figura 2.9, esto es, una descripción de comportamiento de la interfaz junto con la notación en la que viene expresada dicha descripción (línea de la 2 a la 14) y dos programas remotos para el emparejamiento semántico entre interfaces, uno fuerte (línea 15) y otro débil (línea

16). Según el ejemplo, `Larch.exact` es el nombre del programa (en forma de servlet) que realiza emparejamientos semánticos fuertes en notación Larch-CORBA. De forma similar, `Larch.soft` es el nombre del programa que realiza emparejamientos semánticos débiles en notación Larch-CORBA.

2.3.3. Eventos

Véase Capítulo 1, *sección 1.3.1*

El modelo de documentación de componentes comerciales contempla la posibilidad de incluir la definición de los eventos producidos y consumidos por un componente en la parte funcional de un documento COTS, tal y como se define en los modelos de componentes CCM, EJB, o EDOC [OMG, 2001].

Los eventos se definen dentro de los elementos `<consumedEvents>` y `<producedEvents>`, como parte de la sección funcional `<functional>` de un componente COTS. En la figura 2.11 se recoge la definición de esquema del tipo `eventType`, establecido en la definición de los elementos `<consumedEvents>` y `<producedEvents>`. Los eventos de un componente se representan dentro de la plantilla como una lista de nombres de evento utilizando etiquetas `<event>`, tal y como se puede ver en la figura 2.12 en las líneas 2 y 5.

Descripción eventos: <event>

```
<xsd:complexType name="eventType">
  <xsd:element name="event" type="xsd:string" maxOccurs="unbounded"/>
</xsd:complexType>
```

Figura 2.11: Definición de esquema de los eventos de un documento COTS

```
1: <consumedEvents>
2:   <event>eventoA</event> ... <event>eventoM</event>
3: </consumedEvents>
4: <producedEvents>
5:   <event>eventoN</event> ... <event>eventoZ</event>
6: </producedEvents>
```

Figura 2.12: Una instancia ejemplo de los eventos de un documento COTS

El modelo de documentación no contempla la necesidad de indicar programas de emparejamiento exacto o débil para los eventos, como lo hace para otras partes de un documento COTS: interfaces, comportamiento o coreografía. Como sabemos, estos programas son de utilidad para el modelo de mediación (*Capitulo 3*) en procesos de búsqueda y selección. Para el caso de los eventos, el modelo de mediación utiliza las tradicionales operaciones de emparejamiento (exacta o débil) entre listas de palabras, como palabras clave. A nivel de eventos, un componente puede reemplazar a otro de forma "exacta" si emite y consume los mismos eventos, es decir, los conjuntos de nombres de evento consumidos y producidos son los mismos. Y se produce emparejamiento "suave" si consume (o produce) alguno de los nombres de evento que consume (o produce) el otro.

2.3.4. Coreografía (protocolos)

La última parte de una descripción funcional de un documento COTS hace referencia a la coreografía del componente, normalmente conocida como información de protocolo [Canal et al., 2001] [Vallecillo et al., 2000]. Un protocolo establece el orden en el cual las

operaciones de las signaturas de una interfaz deben ser llamadas. Además (como vimos en 1.3.3) el orden de estas llamadas puede variar dependiendo del tipo del componente con el que interacciona y del *escenario* donde se lleva a cabo la interacción (en este caso, a las interfaces se las denomina *roles*). Por tanto, a nivel de componente, se puede hablar de diferentes protocolos en función del escenario donde éste sea utilizado [Han, 1998].

Véase Capítulo 1, *sección 1.3.3*

La descripción funcional de un documento COTS establece con carácter opcional la presencia de la coreografía de un componente, y que, en caso de existir, ésta queda establecida con una descripción de la coreografía junto con la notación en la cual ésta viene expresada, por ejemplo, usando formalismos como redes de Petri [Bastide et al., 1999], máquinas de estados finitas [Yellin y Strom, 1997], lógica temporal [Han, 1999] [Lea y Marlowe, 1995] o el π-cálculo de Milner [Canal et al., 2001] [Henderson, 1997] [Lumpe et al., 1997], entre otros. Además, como ocurre para las otras partes funcionales de una interfaz (parte sintáctica y semántica), para una coreografía también se puede indicar, con carácter opcional, dos operadores de emparejamiento de protocolos, uno fuerte ($\preceq_E$) y otro débil ($\preceq_S$).

El operador de emparejamiento fuerte (o exacto) entre las coreografías de dos interfaces A y B, escrito de la forma $A \preceq_E B$, determina si la interfaz B puede reemplazar a la interfaz A a nivel de protocolos, esto es, si los protocolos de la interfaz A (establecidos todos ellos en una única descripción D, según el modelo) están definidos como protocolos dentro de la interfaz B. De nuevo, esta relación de subtipado de interfaces (esta vez a nivel de protocolos) puede ser tratada como una extensión de las relaciones de subtipado de signaturas de interfaces [Zaremski y Wing, 1995] para considerar descripciones de protocolos de componente [Canal et al., 2001] [Nierstrasz, 1995] [Canal et al., 2003].

Una operación de emparejamiento débil a nivel de protocolos entre dos interfaces se utiliza para hacer emparejamientos parciales. En este caso, partiendo de la descripción de los protocolos de una interfaz, diremos que, una interfaz B puede reemplazar parcialmente a otra A (escrito en la forma $A \preceq_S B$) cuando la interfaz B "contiene" algunas de las definiciones de protocolo definidas en A. Intuitivamente, esto significa que la interfaz B ofrece, al menos, una definición de protocolo definida en la interfaz A. En el *Capítulo 4* se describirán estos operadores con más detalle.

En la figura 2.13 se puede ver un ejemplo que ilustra cómo se describe la descripción de coreografía (protocolos) de una interfaz dentro de un documento COTS, expresada para este ejemplo en notación π-cálculo. La descripción de esquema del elemento `choreography` coincide con la del tipo `behaviorType`, tratado en la sección 2.3.2 (página 81) para describir el comportamiento de las interfaces de un componente.

```
1:  <choreography>
2:    <description notation="pi-protocols">
3:      OnePlaceBuffer(ref,out) =
4:            (^r).ref?write(x,rep).out!print(x,r).r?().rep!().Full(ref,out,x);
5:      Full(ref,out,x) =
6:            ref?read(rep).rep!(x).OnePlaceBuffer(ref,out);
7:    </description>
8:    <exactMatching href="http://soleres.ual.es/servlet/PI.exact" />
9:    <softMatching href="http://soleres.ual.es/servlet/PI.soft" />
10: </choreography>
```

Figura 2.13: Una instancia ejemplo de la coreografía (protocolos) de un documento COTS

Como vemos en la figura, la descripción de coreografía se establece dentro de una sección XML llamada `choreography`. Una vez más —como ocurría en los elementos `interface` o

`behavior`— el modelo recurre a un elemento `description` para indicar la notación usada, y a dos programas de emparejamiento remotos establecidos mediante un enlace `href` dentro de los elementos `exactMatching` (línea 8) y `softMatching` (línea 9). En el ejemplo, `PI.exact` (línea 8) representa el nombre del programa remoto que realiza el emparejamiento fuerte entre dos protocolos, y `PI.soft` (línea 9) el nombre del programa remoto para emparejamientos débiles entre dos protocolos. Estos programas representarían las implementaciones de los operadores definidos en [Canal et al., 2001] para el caso de los protocolos descritos en π-calculus.

2.4. Descripción extra-funcional <properties>

La segunda parte de un documento COTS tiene que ver con la información extra-funcional de un componente. La información extra-funcional cubre aspectos que no están directamente relacionados con la funcionalidad del componente, como la calidad de servicio, el contexto del funcionamiento, y otras propiedades extra-funcionales.

Esta información juega un papel muy importante en algunas de las actividades del desarrollo de software basado en componentes, como son las actividades de búsqueda y selección de componentes, o las actividades de evaluación de componentes COTS. Estos requisitos extra-funcionales podrían tener, en algunos casos, una prioridad mayor que los requisitos funcionales, o incluso ser decisivos a la hora de decidir por un componente u otro en actividades de selección, en el caso de encontrar dos o más componentes con una funcionalidad similar.

Para la definición de los requisitos extra-funcionales, el modelo de documentación de componentes COTS adopta el modelo de propiedades de ODP [ISO/IEC-ITU/T, 1997], que usa tripletas de la forma: nombre, tipo, valor (véase también 1.7.3). Las *propiedades* son la forma usual en la cual se expresa en la literatura los aspectos extra-funcionales de los objetos, servicios y componentes. El modelo de documentación también propone el uso de tipos W3C basados en XML para describir las propiedades, aunque cualquier notación sería valida para describirlas, como por ejemplo el estilo CCM de OMG [OMG, 1999] (páginas 10–365) que utiliza también un vocabulario XML.

La parte extra-funcional de un componente puede quedar especificada por una secuencia de propiedades capturando, cada una de ellas, un atributo extra-funcional de componente con tres partes: (a) el nombre del atributo, (b) el tipo de dato del valor capturado por el atributo, y (c) su valor actual. Además de representar propiedades simples, el modelo también permite definir propiedades complejas, y aspectos de trazabilidad entre los atributos extra-funcionales y la funcionalidad del componente.

2.4.1. Definición de propiedades

Como se puede ver en la definición de esquema mostrada en la figura 2.14, el modelo de documentación recoge la colección de las propiedades extra-funcionales dentro de un elemento `properties`. Para representarlas, se puede escoger (`<xsd:choice>`) entre dos formas posibles: (a) bien directamente con uno o más elementos `property` (como se muestra en la figura 2.15), (b) o bien mediante un enlace (`href`) hacia el archivo remoto que contiene la descripción con las propiedades. En cualquiera de los dos casos, es necesario indicar junto al elemento `properties` el tipo de notación que se está utilizando para describir las propiedades extra-funcionales (p.e., `notation="W3C"`).

Además, las tres partes que componen la especificación de una propiedad (nombre, tipo y valor) están definidas por el tipo `propertyType` que es asignado al elemento `property`.

Propiedades: <properties>

```
<xsd:element name="properties" type="propertiesType" minOccurs="0"/>
<xsd:complexType name="propertiesType">
  <xsd:attribute name="notation" type="xsd:string"/>
  <xsd:choice>
    <xsd:attribute name="href" type="xsd:uriReference"/>
    <xsd:element name="property" type="propertyType" maxOccurs="unbounded"/>
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="propertyType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:sequence>
    <xsd:element name="type" type="xsd:datatype"/>
    <xsd:element name="value" type="valueType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="valueType">
  <xsd:choice>
    <xsd:attribute name="href" type="xsd:uriReference"/>
    <xsd:simpleContent> <xsd:extension base="xsd:string"/> </xsd:simpleContent>
  </xsd:choice>
</xsd:complexType>
```

Figura 2.14: Definición de esquema de las propiedades de un documento COTS

El tipo `propertyType` establece, para cada propiedad, un atributo `name` —para indicar el nombre de la propiedad— y una secuencia `<xsd:sequence>` de dos elementos internos a la propiedad: (a) el elemento `type` y (b) elemento `value`. El elemento `type` establece el tipo del valor capturado por el elemento `value`. Los valores que puede tomar el elemento `type` están establecidos a su vez por el tipo `xsd:datatype` del W3C. Este tipo está definido en `http://www.w3.org/2000/10/XMLSchema` al que apunta el espacio de direcciones `xmlns:xsd` establecido al inicio de un documento `COTScomponent` (véase figura 2.3 en la página 75). El elemento `value` encapsula el valor de la propiedad.

```
1:  <properties notation="W3C">
2:    <property name="capacity">
3:      <type>xsd:int</type><value>1</value>
4:    </property>
5:    <property name="isRunningNow"> <!– Una propiedad dinámica –>
6:      <type>xsd:bool</type> <value href="http://soleres.ual.es/servlet/isRunning"/>
7:    </property>
8:  </properties>
```

Figura 2.15: Una instancia ejemplo de las propiedades de un documento COTS

Un ejemplo de propiedad simple se muestra entre las líneas 2 y 4 de la figura 2.15. En este caso, el componente tiene una propiedad extra-funcional llamada `capacity` que almacena un valor de tipo entero `xsd:int`, con valor igual a `1` en el ejemplo.

Además, tal y como se recoge en la definición de su esquema, una propiedad extra-funcional también puede contener valores que no estén directamente fijados en la plantilla de componente dentro de una etiqueta `<value>`. En estos casos decimos que una propiedad es "dinámica", ya que su valor debe ser evaluado dinámicamente mediante un programa,

que es el indicado en el atributo `href`. La propiedad `isRunningNow` (mostrada en la figura 2.15 entre las líneas 5 a la 7) es un ejemplo de propiedad dinámica. Para este caso, el valor de la propiedad `isRunningNow` es un enlace a un programa llamado `isRunning` (línea 6) que devuelve un valor lógico (`xsd:bool`) para indicar, en este caso, si el componente está ejecutándose o no en el momento de la invocación.

2.4.2. Propiedades complejas (AND y OR)

Para tratar de forma efectiva los requisitos extra-funcionales, el modelo de componentes utiliza algunos de los principios de una aproximación cualitativa denominada *NFR Framework* [Chung et al., 1999]. Esta aproximación se basa en la representación explícita y análisis de los requisitos extra-funcionales. Teniendo en cuenta la naturaleza compleja de los requisitos extra-funcionales, no siempre podemos afirmar que estos requisitos se puedan cumplir en su totalidad en los procesos de selección o evaluación de componentes. La aproximación *NFR Framework* representa los requisitos extra-funcionales como "objetivos-débiles" (*soft-goals*), los cuales no tienen necesariamente unos criterios de satisfacción bien definidos.

Los requisitos extra-funcionales pueden ser desglosados en otros más simples o específicos. Por ejemplo, un requisito de seguridad podrá ser visto como algo demasiado amplio y abstracto. Para tratar explícitamente este tipo de requisito, podríamos dividirlo en otras partes más pequeñas y fáciles de entender, de forma que conduzcan a soluciones no ambiguas en procesos de selección. Por tanto, este requisito complejo lo podríamos tratar como un "objetivo-débil" (*soft-goal*) que se llega a él por la composición de otros "subobjetivos-débiles" más simples. Por ejemplo, un objetivo de "seguridad" se puede alcanzar si los sub-objetivos en los que éste se descompone también se han alcanzado, como la integridad, la confidencialidad o la disponibilidad. En realidad, en función del tipo de composición que se establezca entre sus sub-objetivos, un objetivo se puede alcanzar de dos formas: (a) por una composición de tipo AND, (b) por una composición de tipo OR. Una composición de tipo AND obliga a que todos los sub-objetivos se tengan que cumplir para que el objetivo principal también se cumpla. Por el contrario, una composición de tipo OR alcanza el objetivo principal si al menos se ha alcanzado uno de los sub-objetivos.

Para soportar una perspectiva de "objetivos" y "sub-objetivos" basada en tipos de composición AND y OR, como la que hemos visto, el modelo de documentación permite que una propiedad `<property>` pueda estar compuesta a su vez de dos o más propiedades (similar a los sub-objetivos). A esta clase de propiedad la vamos a denominar "propiedad compleja". En la figura 2.16 se muestra la definición de esquema de una propiedad compleja, donde se hace una extensión del tipo `propertyType` (el tipo de una propiedad) para soportar propiedades complejas. Ahora, el tipo `propertyType` está compuesto por tres atributos (`name`, `composition` y `priority`) y por una elección (`xsd:choice`) entre una pareja tipo-valor (`type`, `value`) o una propiedad interna. Esto último facilita que las propiedades se puedan definir unas dentro de otras. Además, XML-Schemas de W3C permite hacer un uso especial del atributo de esquema `href` para referirse a un elemento que haya sido definido previamente en alguna parte de la plantilla. Por ejemplo, la cláusula `href="property"` (que hay en la selección `<xsd:choice>`, casi al final de la figura 2.16) se refiere a un elemento `property` que ya ha sido definido previamente, al inicio de la figura.

En cuanto a los atributos de una propiedad (citados anteriormente), `name` representa el nombre de la propiedad, y `composition` es un atributo opcional (`minOccurs="0"`) que se utiliza para establecer el tipo de composición de las propiedades definidas dentro de la propiedad objeto. El tipo de valor que puede tomar el atributo `composition` está limitado (`xsd:restriction`) a AND (para una composición de tipo AND) y a OR (para una composición de tipo OR). En la figura 2.17 se muestran dos propiedades complejas que hacen

Propiedades (complejas): <properties>

```
<xsd:element name="property" type="propertyType" maxOccurs="unbounded"/>
<xsd:complexType name="propertyType">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="composition" minOccurs="0">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="OR"/>
        <xsd:enumeration value="AND"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute name="priority" minOccurs="0">
    <xsd:simpleType>
      <xsd:restriction base="xsd:positiveInterger">
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="9"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:choice>
    <xsd:sequece>
      <xsd:element name="type" type="xsd:datatype"/>
      <xsd:element name="value" type="valueType" minOccurs="0"/>
    </xsd:sequence>
    <xsd:element href="property"/>
  </xsd:choice>
</xsd:complexType>
```

Figura 2.16: Definición de esquema de las propiedades complejas de un documento COTS

uso de estos atributos de composición. La primera de ellas (entre las líneas 2 y 9) se refiere a una propiedad compleja de "seguridad" cuyo valor queda establecido por la composición `OR` de otras dos propiedades simples. La segunda propiedad compleja, que se muestra en la figura entre las líneas 10 y 17, se refiere a un caso especial de elemento `<property>` para recoger "palabras claves" (*keywords*) dentro de la plantilla. Las palabras claves son muy usuales para procesos de búsqueda por una o más palabras (del mismo estilo a como se hace por ejemplo en los motores de búsqueda de Internet). Para este caso, una la lista de palabras claves se puede establecer mediante una propiedad compleja de tipo `AND`, cuyo contenido es un secuencia de etiquetas `property`, una por cada palabra clave.

El tercer y último atributo de propiedad es `priority` (véase la definición de esquema de la figura 2.16). Se puede utilizar el atributo `priority` para establecer niveles de prioridad (pesos) en las propiedades, muy útiles para labores de toma de decisiones en procesos de selección o evaluación de componentes. Estos niveles de prioridad se pueden fijar utilizando una escala del 0 (prioridad nula) al 9 (la más alta). Esta escala es la más utilizada en la mayoría de los procesos de toma de decisiones actuales. Por ejemplo, la propiedad compleja `security` mostrada en la figura 2.17 tiene asignada un nivel de prioridad bastante elevado y que (visto el ejemplo) prevalece sobre la propiedad compleja `keywords`, al no tener ésta un nivel de prioridad establecido (por omisión la prioridad es nula). Además, a las propiedades que hay dentro de a una propiedad compleja también se les puede asignar un nivel de prioridad. El orden de evaluación de las propiedades dinámicas se realiza desde las propiedades más internas hacia las más externas (véase *Capítulo 3*).

```
1:  <properties notation="W3C">
2:    <property name="security" composition="OR" priority="8">
3:      <property name="user-authorization">
4:        <type>xsd:string</type> <value>LOGIN</value>
5:      </property>
6:      <property name="manager-authorization">
7:        <type>xsd:string</type><value>ADMIN</value>
8:      </property>
9:    </property>
10:   <property name="keywords" composition="AND">
11:     <property name="keyword">
12:       <typexsd:string></type><value>storage</value>
13:     </property>
14:     <property name="keyword">
15:       <typexsd:string></type><value>bounded</value>
16:     </property>
17:   </property>
18: </properties>
```

Figura 2.17: Una instancia ejemplo de las propiedades complejas de un documento COTS

2.4.3. Trazabilidad de requisitos

Para finalizar con la parte de la descripción extra-funcional de un documento COTS, el modelo de documentación también contempla la necesidad de poder expresar si una propiedad dada (ya sea simple o compleja) está "implementada por" un elemento funcional del componente (por ejemplo dentro de una de las interfaces del componente) o si una propiedad dada está "presente en" un elemento funcional concreto del componente. El primer caso permite cierto nivel de trazabilidad de los requisitos extra-funcionales, esto es, conocer qué elemento funcional proporciona uno o más de los atributos extra-funcionales requeridos. El segundo caso establece qué elementos funcionales exhiben un determinado atributo extra-funcional.

Propiedades (trazabilidad): <properties>

```
<xsd:complexType name="propertyType">
  ...
  <xsd:sequence>
    <xsd:element name="type" type="xsd:datatype"/>
    <xsd:element name="value" type="valueType" minOccurs="0"/>
    <xsd:element name="implementedBy" type="xsd:string" minOccurs="0"/>
    <xsd:element name="implementedIn" type="xsd:string" minOccurs="0"/>
    <xsd:element href="property"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 2.18: Definición de esquema de la trazabilidad de un documento COTS

Para soportar estos dos casos, el modelo de documentación de componentes COTS incluye dos elementos más en la definición de esquema del tipo `propertyType`: el elemento `implementedBy` y el elemento `implementedIn` (véase figura 2.18). Como se observa en el esquema, estos dos elementos son opcionales dentro del apartado `properties` de una plantilla de documento COTS. Además, los elementos a los que se refieren las etiquetas

`implementedBy` e `implementedIn` deben ser elementos funcionales previamente definidos en la plantilla: interfaces, eventos o protocolos.

En la figura 2.19 mostramos un sencillo ejemplo que ilustra el funcionamiento de los elementos `implementedBy` e `implementedIn` introducidos anteriormente.

```
1:  <properties notation="W3C">
2:    <property name="user-authorization">
3:      <type>xsd:string</type> <value>LOGIN</value>
4:      <implementedBy>LoginInterface</implementedBy>
5:    </property>
6:    <property name="replication">
7:      <type>xsd:string</type><value>CONSENSUS</value>
8:      <implementedIn>LonginInteface</implementedIn>
9:    </property>
10: </properties>
```

Figura 2.19: Trazabilidad en un documento COTS

2.5. Descripción de empaquetamiento <packaging>

La tercera parte de un documento COTS tiene que ver con la información de empaquetamiento del componente. Esta información está relacionada con aspectos de descarga e instalación del componente y aspectos de implementación, como por ejemplo los tipos de sistemas operativos o procesadores válidos para que el componente pueda funcionar, o las dependencias del componente con aquellos programas que deben estar instalados en el entorno de implantación antes de hacer la descarga, entre otros aspectos de empaquetamiento.

Descripción de empaquetamiento: <packaging>

```
<xsd:element name="packaging" type="packagingType" minOccurs="0"/>
<xsd:complexType name="packagingType">
  <xsd:element name="description" type="locationType"/>
</xsd:complexType>
```

Figura 2.20: Definición de esquema de empaquetamiento de un documento COTS

En la figura 2.20 se muestra la definición de esquema para la descripción de empaquetamiento de un componente COTS. Como vemos, esta definición de esquema utiliza el elemento `packaging` para recoger la información de empaquetamiento. Una vez más, el modelo de documentación de componentes COTS establece el uso del elemento `description` para llevar a cabo la descripción de empaquetamiento de un componente. Como ya sabemos, el elemento `description` tiene la facilidad de poder representar una descripción de dos formas diferentes (establecido por el tipo `locationType`[7]): incluyendo la descripción directamente dentro de la propia plantilla, o mediante un enlace `href` hacia el archivo que contiene la descripción.

En la figura 2.21 se muestra cómo se recoge la descripción de empaquetamiento de un componente en una sección `<packaging>` mediante una etiqueta `<description>`. Como se observa, en el ejemplo se ha utilizado un enlace al archivo `UnaImplementacionOPB.csd` que

[7]El *Apéndice A* contiene la definición de esquema para el tipo `locationType`.

```
1:  <packaging>
2:    <description notation="CCM-softpkg" href=".../UnaImplementacionOPB.csd"/>
3:  </packaging>
```

Figura 2.21: Una instancia ejemplo de empaquetamiento en un documento COTS

contiene la descripción de empaquetamiento en notación "CCM-softpkg" [OMG, 1999]. Un paquete de componente CORBA mantiene una o más implementaciones de un componente, y está compuesto de uno o más descriptores y un conjunto de archivos. Los descriptores describen las características del paquete (tales como sus contenidos o sus dependencias) mediante un enlace hacia sus archivos. Esta información permite describir recursos, archivos de configuración, la localización de diferentes implementaciones de componente para diferentes sistemas operativos, la forma en la que están empaquetadas estas implementaciones, los recursos y programas externos que necesitan, etc. Un ejemplo de todo esto se puede ver bien en la figura 2.22, que muestra el contenido del archivo `UnaImplementacionOPB.csd` (línea 2 de la figura 2.21).

```
1:  <?xml version="1.0"?>
2:  <softpkg name="OnePlaceBufferService" version="1.0">
3:    <pkgtype>CORBA Component</pkgtype>
4:    <idl id="IDL:vendor1/OnePlaceBuffer:1.0">
5:      <link href="http://www.cotstrader.com/examples/OPB/OnePlaceBuffer.idl"/>
6:    </idl>
7:    <implementation>
8:      <os name="WinNT" version="4.0"/>
9:      <os name="AIX"/>
10:     <processor name="x86"/>
11:     <processor name="sparc"/>
12:     <runtime name="JRE" version="1.3"/>
13:     <programminglanguage name="Java"/>
14:     <code type="jar">
15:       <fileinarchive name="OPB.jar"/>
16:       <entrypoint>OPB.OnePlaceBuffer</entrypoint>
17:     </code>
18:     <dependency type="ORB"> <name>ORBacus</name> </dependency>
19:   </implementation>
20: </softpkg>
```

Figura 2.22: Un documento de empaquetamiento en notación `softpackage` de CCM

En el documento de empaquetamiento CCM de la figura, los aspectos técnicos de implementación del componente se recogen mediante una sección `<implementation>`. En ella se especifica que el componente funciona para los sistemas operativos WinNT y AIX, y para procesadores "x86" (normalmente PC) o procesadores "sparc" (estaciones de trabajo), y que se requiere el entorno de ejecución JRE. Además, para la implementación del componente se ha utilizado el lenguaje de programación Java, y se ofrece en un paquete `.jar` de Java (`OPB.jar`) cuyo punto de acceso al componente se hace en `OPB.OnePlaceBuffer`. Finalmente, el documento de empaquetamiento también desvela que el componente depende de un programa de la clase ORB, concretamente, el programa ORBacus.

2.6. Descripción de marketing <marketing>

La cuarta y última parte de un documento COTS tiene que ver con una descripción no técnica del componente y del fabricante del componente. En el modelo de documentación de componentes COTS, a esta clase de información se la denomina "descripción de marketing" (o requisitos de marketing).

En la figura 2.23 se ofrece la definición de esquema de una descripción de *marketing*. Esta descripción (como la de empaquetamiento y la extra-funcional) no es obligatoria en un documento de componente COTS (de ahí el atributo `minOccurs="0"` en el esquema).

Descripción de marketing: <marketing>

```
<xsd:element name="marketing" type="marketingType" minOccurs="0"/>
<xsd:complexType name="marketingType">
  <xsd:sequence>
    <xsd:element name="license" type="locationType"/>
    <xsd:element name="expirydate" type="xsd:string"/>
    <xsd:element name="certificate" type="locationType"/>
    <xsd:element name="vendor" type="vendorType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="vendorType">
  <xsd:sequence>
    <xsd:element name="companyname" type="xsd:string"/>
    <xsd:element name="webpage" type="xsd:uriReference" maxOccurs="unbounded"/>
    <xsd:element name="mailto" type="xsd:string" maxOccurs="unbounded"/>
    <xsd:element name="address" type="addressType" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 2.23: Definición de esquema de marketing de un documento COTS

El esquema de una descripción de *marketing* establece aquellos elementos de marketing que aparecen en los documentos o fichas no-técnicas de los componentes comerciales. No obstante, por las características del lenguaje XML-Schemas del W3C, la definición de esquema de un documento COTS (véase *Apéndice A*) puede ser extendida para definir (en un archivo independiente) nuevos elementos de marketing. Para utilizar estos nuevos elementos en un documento COTS, es necesario definir un espacio de nombres con una cláusula `xmlns` al inicio del documento que apunte hacia el archivo que contiene la extensión con los nuevos elementos de marketing. A modo de ejemplo, en la figura 2.24 se muestra un caso donde se ilustra el comportamiento anteriormente comentado. En la línea 8 se puede ver cómo se utiliza un nuevo elemento llamado `marketingB:price`, usado para conocer el precio del componente. Para este ejemplo, se ha incluido un enlace a un programa "servlet" que devuelve el precio del componente en tiempo real. La definición de esquema para un elemento de precio se establece en el archivo `miNuevoMarketing.xsd`, al que apunta el espacio de nombres `xmlns:marketingB` dentro de un documento `COTScomponent`[8].

Para concluir, el ejemplo de plantilla COTS mostrado en la figura 2.24 también sirve como punto de referencia para ilustrar el comportamiento de la descripción de marketing definida anteriormente.

[8]En el documento del ejemplo se ha omitido los dos espacios de nombres usuales: el que apunta a la gramática de tipos del W3C y el que apunta a la gramática del documento COTS. En la página 75 se muestra un documento con estos dos espacios de nombres.

```
1:  <COTScomponent name="OnePlaceBuffer" ...
2:          xmlns:marketingB="http://unsitio.es/miNuevoMarketing.xsd">
3:    ...
4:    <marketing>
5:      <license href=".../licencia.html" />
6:      <expirydate>2005-10-05</expirydate>
7:      <certificate href=".../lcard.pgp" />
8:      <marketingB:price href=".../servlet/prices.OPB.getPrice" />
9:      <vendor>
10:       <companyname>cotstrader</companyname>
11:       <address>La Cañada de San Urbano, 04120</address>
12:     </vendor>
13:   </marketing>
14: </COTScomponent>
```

Figura 2.24: Una instancia ejemplo de marketing de un documento COTS

2.7. Trabajos relacionados

La documentación de componentes software es posiblemente en la actualidad, una de las áreas más activas de la ingeniería del software basada en componentes. En estos últimos años han aparecido en la literatura muchas y variadas propuestas para la documentación de componentes software (en general). En lo que se refiere a propuestas de plantillas de especificación de componentes, destacamos el trabajo de Jun Han [Han, 2000] en un proyecto conjunto con Fujitsu Australia. Estas plantillas proporcionan información semántica "por encima" de la descripción estándar de signatura, usada para la selección de componentes.

También está el trabajo [Dong et al., 1999] que propone una plantilla para la especificación de componentes, útiles para desarrolladores y vendedores de componentes en fases de actualización de software y en procesos de selección y evaluación. Estas plantillas incluyen información funcional (como aspectos estructurales y de comportamiento del componente), propiedades extra-funcionales, y otra información adicional como la utilidad, los componentes relacionados o ejemplos de uso.

Los componentes de grano grueso se refieren a nivel de aplicaciones

Las plantillas *Advanced Components* de la arquitectura WSBC (*WebSphere Business Components*) de IBM [IBM-WebSphere, 2001], son un ejemplo de plantillas XML para la especificación de componentes de grano grueso. Estas plantillas permiten recoger ciertos aspectos funcionales del componente, como las interfaces proporcionadas y requeridas por medio de la definición de sus operaciones y de las características de las peticiones (entradas) y respuestas (salidas) de cada operación (definidas como esquemas), y de sus dependencias con otros programas. Sin embargo, la plantilla carece de estilo para recoger otro tipo de información muy importante en tareas de selección y evaluación de componentes, como es la información extra-funcional. También carece de estilo para representar, básicamente, información de protocolos y de comportamiento.

Otra propuesta de documentación basada en plantillas XML es la que ofrece el lenguaje WSDL (*Web Services Definition Language*) para la definición de servicios webs. Como vimos en el *Capítulo 1*, un servicio web es una aplicación (componente) que puede ser localizada e invocada a través de Internet y que se basa en estándares abiertos como XML, SOAP o UDDI. El lenguaje WSDL permite definir detalles de especificación a nivel de interfaz de un servicio web, como las signaturas y tipos de datos de entrada y salida de las operaciones, o la forma en la que se llama las operaciones y su devolución, y detalles

de conexión. Sin embargo, la especificación de WDSL, al igual que las plantillas *Advanced Components* de IBM, no contempla la posibilidad de representar información de protocolos, de comportamiento o extra-funcional.

Otros trabajos son aquellos que abordan aspectos puntuales de una documentación de componente. Por ejemplo, algunos autores, como en [Cho et al., 1998], [Bastide et al., 1999] o [Canal et al., 2003], proponen extensiones de un IDL para tratar la información de protocolos utilizando transiciones de estados, redes de Petri y π-cálculo, respectivamente. También está el trabajo descrito en [Rosa et al., 2001] que estudia cómo incluir la información extra-funcional en los componentes comerciales con el fin de relacionarlos con las especificaciones de arquitectura de las aplicaciones.

Los trabajos aquí mencionados han servido de inspiración para elaborar el modelo de documentación de componentes COTS propuesto en el presente capítulo. Existen otros muchos trabajos también relacionados con la especificación de componentes, algunos de ellos tratados en el *Capítulo 1*.

2.8. Resumen y conclusiones del capítulo

Uno de los aspectos clave en el desarrollo de software basado en componentes (DSBC) es la disponibilidad de especificaciones de componentes adecuadas y completas para llevar a cabo de forma efectiva los procesos de selección y evaluación de componentes. Tradicionalmente, la especificación de un componente software se ha centrado sólo en la descripción de sus interfaces a nivel de signaturas (nivel sintáctico). Se ha visto que, para las labores de selección y evaluación de componentes software, es necesaria otra clase de información adicional, como es la información sobre el comportamiento de las operaciones de las interfaces, o sobre los protocolos de interacción del componente. Asimismo, los requisitos extra-funcionales juegan también un papel importante y decisivo durante la toma de decisiones en los procesos de selección y evaluación de componentes. En los últimos años se han propuesto diferentes extensiones de la especificación tradicional de un componente software para soportar información extra-funcional, información de comportamiento, e información de protocolos.

Además, el carácter de "caja negra" de los componentes COTS obliga a disponer de suficiente información para su evaluación y consulta sin tener que recurrir a su código fuente. Por tanto es fundamental contar con una documentación para este tipo de componentes que permita realizar dichos procesos.

En este capítulo hemos presentado una propuesta de un modelo para la documentación de componentes COTS basado en plantillas XML. El modelo está soportado por un lenguaje para la definición de componentes COTS en la notación de esquemas (XML-Schemas) del W3C, notación utilizada además para la discusión del modelo propuesto a lo largo de este capítulo. En la línea de los trabajos actuales en especificación de componentes comerciales, el modelo de documentación presentado permite documentar un componente comercial a partir de cuatro clases de descripciones: (a) una descripción funcional, para definir las interfaces proporcionadas y requeridas por el componente, junto con el comportamiento de las operaciones de dichas interfaces, y aspectos de coreografía (protocolos) del componente; (b) una descripción extra-funcional, para atributos de calidad; (c) una descripción de empaquetamiento, para aspectos de implantación e implementación de componente; y (d) una descripción de *marketing*, para aspectos no técnicos del componente y de su fabricante.

Capítulo 3

Un Modelo de Mediación para Componentes COTS

Capítulo 3

Un Modelo de Mediación de Componentes COTS

Contenidos

El desarrollo de software basado en componentes (DSBC) está siendo reconocido como una de las tecnologías clave para la construcción de sistemas complejos a gran escala, evolutivos y de alta calidad. Una gran parte de las actividades del DSBC se centran en la búsqueda sistemática de unos componentes software con unas capacidades y supuestos internos particulares, y en el análisis de las incompatibilidades que surgen cuando estos se utilizan de forma conjunta. Aunque todo parece indicar que la figura de un mediador de componentes podría jugar un papel relativamente importante en este proceso, la realidad es que, en la mayoría de los métodos existentes para el desarrollo de software basado en componentes, no se utilizan de forma efectiva estos servicios de mediación, debido en parte a la falta de un modelo de mediación de componentes COTS que lo identifique. En este capítulo presentamos las características principales que debería tener un modelo de mediación de componentes COTS que funcione en sistemas abiertos. Y también presentamos algunos detalles de implementación de un servicio de mediación llamado COTStrader que sustenta el modelo de mediación de componentes COTS anunciado.

El capítulo está organizado en siete secciones. En ellas, comenzaremos analizando y justificando más detalladamente la necesidad de un modelo de mediación de componentes COTS en la sección 3.1. Seguidamente, en la sección 3.2 ofreceremos una perspectiva global del modelo de mediación, exponiendo las visiones de "servicio" y "tipo de servicio" de componente, las propiedades o requisitos que debe tener un modelo de mediación de componentes COTS, las características del repositorio de mediación, y las capacidades de exportación e importación presentes en una actividad de mediación. El proceso de mediación lo describiremos en la sección 3.3, donde analizaremos los algoritmos de selección y búsqueda desarrollados. En la sección 3.4 veremos algunos detalles de la implementación del servicio de mediación COTStrader, y luego analizaremos su adecuación al modelo de documentación presentado. Finalizando el capítulo, en la sección 3.6 discutiremos algunos trabajos relacionados con la propuesta del modelo de mediación de componentes COTS que aquí se presenta. Concluiremos el capítulo con un breve resumen y algunas conclusiones en la sección 3.7.

3.1. Introducción y conceptos relacionados

Como se adelantó en el *Prólogo* de esta memoria, las prácticas de desarrollo de software basado en componentes están ganando en interés en ingeniería del software debido a la utilización de software comercial (COTS). Esta aproximación está propiciando que las organizaciones cambien sus prácticas de ingeniería desde una perspectiva predominantemente basada en desarrollo de aplicaciones de software, a una perspectiva basada en ensamblaje de partes software reutilizables y comerciales. La construcción de una aplicación de software ahora conlleva la utilización de "piezas" software ya pre-fabricadas y desarrolladas en diferentes sitios y por diferentes personas, y posiblemente con diferentes perspectivas.

Esta aproximación está cambiando algunos de los actuales métodos y herramientas de la ingeniería del software. Por ejemplo, el tradicional método de desarrollo descendente —basado en sucesivos refinamientos de los requisitos del sistema hasta alcanzar una implementación concreta de los componentes de la aplicación final— no se puede aplicar al desarrollo de software basado en componentes (DSBC). En DSBC, cuando el diseñador del sistema elabora los requisitos iniciales, debe tener también en cuenta las especificaciones de unos componentes comerciales desarrollados por terceras partes y almacenados en repositorios de software [Mili et al., 1995] [Robertson y Robertson, 1999] [Wallnau et al., 2002].

Por tanto, existe un cambio significativo entre una aproximación de desarrollo de software dedicada exclusivamente a la construcción de componentes software, y una aproximación de desarrollo centrada en adquisición de componentes software para la construcción de aplicaciones de software [Tran y Liu, 1997]. En este último caso, los arquitectos, diseñadores y desarrolladores de sistemas deben aceptar la fuerte dependencia que existe entre tres aspectos principales:

(a) los requisitos de usuario,

(b) la arquitectura de software, y

(c) los productos COTS [Garlan et al., 1994] [Ncube y Maiden, 2000].

Bajo este nuevo entorno, el interés principal de cualquier desarrollo basado en componentes COTS ha estado en disponer de unos adecuados procesos de búsqueda y selección de componentes COTS. Sin embargo, estos procesos actualmente presentan serias limitaciones, generalmente debido a tres principales razones. En primer lugar, la información que se dispone de un componente software no es lo suficientemente expresiva como para garantizar una selección efectiva del mismo. En segundo lugar, los criterios de búsqueda y evaluación son normalmente muy simples en la práctica. Y finalmente, las actuales metodologías de desarrollo de software basadas en componentes no utilizan procesos intermediarios que busquen y localicen de forma efectiva aquellos componentes software que ofrecen los servicios requeridos.

En el primer caso, una de las características clave en DSBC es el uso de documentaciones (esto es, especificaciones) de componente COTS completas, concisas y que no contengan ambigüedades. En el caso de los componentes COTS, su naturaleza de "caja negra" dificulta la compresión de su comportamiento interno. Además, en los procesos de selección y de evaluación de componentes sólo se tienen en cuenta sus propiedades funcionales, ignorándose otra información crucial en la selección de componentes, como la información semántica y de protocolos [Vallecillo et al., 2000], o los requisitos extra-funcionales [Rosa et al., 2001] [Chung et al., 1999]. Por otro lado, la información que proporcionan los vendedores de componentes software suele ser escasa, y muy desorganizada. En este sentido, en el capítulo anterior presentamos un modelo de documentación de componentes COTS que cubría estas limitaciones.

En segundo término, los procesos de búsqueda y emparejamiento de componentes son (en teoría) delegados a procesos de mediación (*traders*). Sin embargo, los actuales procesos de mediación no son adecuados para trabajar con componentes COTS en sistemas abiertos independientemente extensibles (como por ejemplo Internet) [Vasudevan y Bannon, 1999].

Finalmente, los procesos de mediación no están completamente integrados en las actuales metodologías para un desarrollo efectivo basado en componentes software. Esto en parte se debe la falta de muchas de las potenciales ventajas que deberían proporcionar los procesos de mediación, como la recuperación de la información o la selección automatizada de componentes candidatos.

Por todas estas razones, pensamos que en DSBC es crucial la necesidad de un modelo para la mediación de componentes comerciales que establezca las bases o los requisitos mínimos para construir un servicio de mediación para esta clase de componentes. Las capacidades de un servicio de mediación se relacionan básicamente con tres aspectos:

(a) la recuperación de la información,

(b) la adquisición de componentes, y

(c) la compatibilidad y reemplazabilidad de componentes.

3.1.1. Recuperación de información

La recuperación de la información tiene que ver con aquellas tareas encargadas de recopilar información de software para almacenarla en un repositorio determinado. La idea surge con el concepto de "módulo software" (*asset*), tratado con anterioridad en la sección 1.2.2. Este término ha sido utilizado con frecuencia en el campo de la reutilización de software para referirse a diversas categorías de software, como componentes, objetos, modelos de diseño y de análisis de requisitos de software, esquemas de bases de datos, documentación, código, manuales, escenarios de pruebas, entre otros más [Sodhi y Sodhi, 1999]. De hecho, en los últimos años se han creado diversos repositorios de módulos software con el único objetivo de servir como fuente de información para los ingenieros desde donde poder consultar (o almacenar) prácticas de ingeniería en la forma anteriormente citada (componentes, código, objetos, etc.). Tres ejemplos de estos repositorios son:

Los assets *fueron una iniciativa promulgada en los años 80 y que no prosperó como en un principio se esperaba. La iniciativa intentaba recopilar y catalogar "trozos" de aplicaciones de software para su posterior reutilización*

— ASSET (*Asset Source for Software Engineering Technology*, `http://www.asset.com`) desarrollado conjuntamente por *Science Applications International Corporation* (SAIC) y el *Software Engineering Institute* (SEI), y que mantienen un repositorio de módulos software denominado WSRD (*World Software Reuse Discovery*) con más de 1000 productos catalogados por subareas de aplicación en diferentes dominios, como por ejemplo en inteligencia artificial, procesamiento distribuido, simulación, herramientas de desarrollo de software o interfaces de usuario, así hasta 35 dominios distintos.

— DCAC (*Digital Concepts and Analysis Center*, `http://arc_www.belvoir.army.mil/vert_libraries.html`), un repositorio de módulos software relacionados con el dominio de las imágenes digitales geográficas[1]. Este repositorio cataloga los módulos software en cinco categorías: software de visualización, software raster, software vectorial, software de conversión de imágenes geográficas y software geodésico.

— GAMS (*Guide to Available Methematical Software*, `http://gams.nist.gov`) es un proyecto del NITS (*National Institute of Standards and Technology*) que recopila técnicas de software matemático relacionadas con modelización matemática y análisis estadístico con más de diez mil módulos para la resolución de problemas en más de cien paquetes de software. El repositorio cataloga los módulos software en diversas categorías, como por ejemplo los módulos software relacionados con cálculo por elementos finitos, programación lineal, o métodos numéricos, entre otros.

En cualquiera de los casos, la información que almacena un repositorio viene dada en forma de documento de especificación del módulo software que se almacena. Además, para la consecución de las tareas de recuperación de la información se pueden seguir dos esquemas diferentes. Por un lado está el esquema "bajo demanda" o de "control reactivo", también conocido como modelo *push*, donde la tarea de recopilación, a petición de un cliente, acepta y almacena en el repositorio un documento de especificación de modulo software (*asset*) facilitado por el cliente. Otro esquema de recuperación de información es el de "extracción" o de "control pro-activo", también conocido como modelo *pull*. Aquí la tarea de recopilación consiste en buscar (explorar), detectar y extraer documentos de especificación de componente externos al repositorio. Esta tarea generalmente se asigna a procesos automáticos como los (ro)bots de Internet. Una vez detectado y extraído el documento, éste puede ser registrado en el repositorio utilizando un esquema por demanda.

[1] Con imágenes digitales geográficas hacemos referencia a imágenes de satélite, imágenes vectoriales geográficas (planos cartográficos digitalizados) y fotografías aéreas.

Para nuestros propósitos, tanto la presencia de un repositorio software (en este caso de componentes comerciales) como los esquemas de recuperación de información (reactivo y pro-activo) en el repositorio, han sido algunos de los elementos clave para la elaboración final del modelo de mediación de componentes COTS propuesto, y que analizaremos en este capítulo en próximas secciones.

3.1.2. Adquisición de componentes

La adquisición de componentes está relacionada con aquellas actividades del DSBC ligadas con la búsqueda y selección de componentes software reutilizables en uno o más repositorios [Goguen et al., 1996]. Estas tareas son originarias de los trabajos de búsqueda en repositorios de módulos software. A partir de unos criterios de consulta, una actividad de búsqueda recorre un repositorio de software extrayendo aquellos módulos software que cumplen los criterios de consulta impuestos. El proceso genera un nuevo repositorio, subconjunto del repositorio original. Esta actividad se puede llevar a cabo a diferentes niveles, refinando en cada uno de ellos las condiciones de la consulta anterior y generándose un nuevo repositorio a partir del anterior.

Los algoritmos de búsqueda en repositorios de componentes software son una extensión de los tradicionales algoritmos de búsqueda en repositorios de software, como por ejemplo los algoritmos de búsqueda: (a) basados en palabras clave (*keyworks*) [Matsumoto, 1987], (b) basados en facetas (*facets*) [Prieto-Diaz, 1991], donde el software se clasifica por categorías (facetas), (c) basados en conocimiento [Ostertag et al., 1992] [Henninger, 1994] que utilizan información estadística y basada en conocimiento para la selección de componentes, y (d) los basados en especificaciones. Para estos últimos está por ejemplo el trabajo de [Rollins y Wing, 1991] que utiliza operadores de emparejamiento a nivel de signaturas (las especificaciones) para seleccionar funciones de orden superior en una librería ML, y λProlog para casar las consultas de usuario con las signaturas de componente. Este trabajo es luego extendido en [Zaremski y Wing, 1995], donde las signaturas se consideran como funciones o como interfaces, y donde las operaciones de búsqueda utilizadas también pueden seleccionar módulos, además de funciones. Más tarde en [Zaremski y Wing, 1997] se extiende de nuevo el trabajo para soportar emparejamientos entre especificaciones, y usando el lenguaje de interfaces Larch/ML, para expresar pre y post condiciones en lógica de primer orden, y el probador de teoremas Larch, para verificar si los componentes candidatos satisfacen esas condiciones.

Para nuestros propósitos, el algoritmo de búsqueda y selección del servicio de mediación desarrollado (y que veremos en próximas secciones en este capítulo) está basado en especificaciones de componente, dadas por documentos COTS (descritos en el *Capítulo 2*).

3.1.3. Compatibilidad y reemplazabilidad de componentes

La reemplazabilidad de componentes es un aspecto ligado a las actividades de búsqueda y selección de componentes

Por último, otro aspecto importante en DSBC consiste en determinar cuándo un componente software es reemplazable por otro. Esto es útil, por ejemplo, cuando queremos conocer qué especificaciones concretas de componente (implementación de componente) encontradas por una actividad de búsqueda y selección, casan con las especificaciones abstractas de los componentes que definen (en su conjunto) la arquitectura de software del sistema, esto es: qué especificaciones concretas de componente pueden reemplazar los componentes abstractos de la arquitectura.

Para determinar si un componente A es reemplazable por otro B es necesario analizar si los dos componentes son compatibles entre sí. La compatibilidad entre dos componentes se puede determinar de muy diversas formas, dependiendo del nivel que se considere, o de

la clase de emparejamiento utilizado para llevar a cabo la compatibilidad (emparejamiento fuerte o débil). Por ejemplo, a nivel de signaturas se analiza la compatibilidad de los tipos (subtipado) y de los nombres de las signaturas de los dos componentes. A nivel de comportamiento, se analiza la compatibilidad de las pre y post condiciones de los dos componentes. A nivel de protocolos, se analiza si los mensajes que se envían y reciben entre los dos componentes son compatibles (interoperabilidad).

En nuestro caso, el modelo de mediación propone el uso de dos tipos de emparejamientos (fuerte y débil) y a diferentes niveles, determinado por la información que puede contener una especificación de componente (el documento COTS), a recordar: (a) interfaces (o nivel sintáctico), (b) comportamiento (nivel semántico), (c) protocolos, (d) información extra funcional, donde se permiten además búsquedas basadas en palabras clave, (e) información de implementación e implantación del componente, y (f) información no técnica del componente y de su fabricante.

3.2. Descripción del modelo de mediación

La mediación es el mecanismo natural que se define en los sistemas basados en objetos y componentes para búsqueda y localización de servicios. Un rol cliente que requiere un servicio de componente puede interrogar a un agente intermediario (un servicio de mediación) para que éste le conteste con las referencias de aquellos componentes que proporcionan la clase de servicio requerido. Los servicios de componente anunciados se denominan "exportaciones", mientras que a las consultas se las denominan "importaciones". Un mediador sólo proporciona "referencias a los posibles proveedores" del servicio buscado, sin proporcionar el servicio en sí.

La mediación no sólo es relevante para el DSBC, sino que también es muy útil en campos del software dependientes del contexto, como en la computación móvil o las aplicaciones auto-adaptativas. En estos casos, un servicio de mediación facilita la localización de servicios en entornos locales y permite la reconfiguración automática de aplicaciones. Además, los servicios de mediación extendidos para soportar calidad de servicio (QoS) pueden ser útiles para la auto-configuración de "aplicaciones multimedia". Aunque el modelo de mediación que discutimos en este capítulo sólo se centra en componentes comerciales, la mayor parte de las ideas aquí propuestas pueden ser también aplicadas a todas aquellas disciplinas donde se requiera la mediación.

El servicio de mediación de componentes COTS desarrollado lo hemos denominado COTStrader. El repositorio, que contiene especificaciones de componentes, soporta el modelo de documentación de componentes COTS presentado en el *Capítulo 2*. Cada especificación representa una plantilla de documento COTScomponent. Además, el servicio de mediación COTStrader soporta cuatro operaciones, $O = \{export, withdraw, replace, query\}$. Como veremos más adelante, el servicio de mediación ofrece las tres primeras operaciones para llevar a cabo acciones de exportación de servicios (para el papel de exportador). La última operación está reservada para acciones de importación de servicios (para el papel de importador). Por último, el servicio de mediación de componentes COTS está caracterizado por doce propiedades que lo identifican.

En las siguientes secciones analizaremos con más detenimiento estos tres aspectos. En primer término ofreceremos una definición para el concepto de "servicio", muy usual en el campo de la mediación. Luego, trataremos los tres aspectos citados anteriormente —y que caracterizan a un servicio de mediación— comenzando primero con una descripción de las propiedades de un servicio de mediación de componentes COTS, y siguiendo con una descripción del repositorio y de las operaciones asociados al servicio de mediación.

3.2.1. Servicios y tipos de servicios

Como podemos comprobar, los componentes ofrecen servicios, los clientes solicitan servicios y el intermediario tiene que ver con servicios. Por tanto, deberíamos comenzar definiendo qué entendemos por "servicio". Para ello adoptaremos la definición de servicio ISO dada por la especificación de la función de mediación de ODP [ISO/IEC-ITU/T, 1997] (la cual ya fue discutida en el *Capítulo 1*, sección 1.7).

Definición 3.1 (Servicio, [ISO/IEC-ITU/T, 1997]) *Un servicio es un conjunto de capacidades proporcionadas por un objeto a nivel computacional. Un servicio es una instancia de un tipo de servicio.*

En ODP, un "tipo de servicio" es un tipo de signatura de interfaz, un conjunto de definiciones de propiedad y un conjunto de reglas acerca de las propiedades de servicio. Las signaturas de interfaz describen la funcionalidad del servicio en términos de atributos y métodos, ya que ODP está orientado a objetos. Las propiedades de servicio son una terna (nombre, tipo de valor, modo) donde el nombre identifica la propiedad (por ejemplo, *permiteEncriptación*), el tipo de valor establece el tipo de valores permitidos por la propiedad (por ejemplo, *booleano*) y el modo especifica si una propiedad es de sólo lectura o de lectura-escritura, o si es el valor de la propiedad es opcional, o por el contrario es obligatorio que la propiedad tenga un valor asignado. Además, las propiedades pueden ser declaradas como dinámicas, lo cual significa que en lugar de tener asignado un valor fijo, pueden tener asociado un evaluador encargado de proporcionar dinámicamente el valor actual de la propiedad (p.e., la longitud de una cola).

Esta clase de "tipo de servicio" ODP es la más usada en la mayoría de los servicios de mediación actuales. Sin embargo, necesitamos extender esta definición para permitir descripciones de servicio más completas y para amoldarse a los requisitos particulares de los sistemas basados en componentes COTS.

En nuestro contexto, un tipo de servicio vendrá dado por las cuatro partes de un documento COTS (tratado en el *Capítulo 2*). La primera de ellas describe los aspectos funcionales (es decir computacionales) del servicio, incluyendo la información sintáctica (las signaturas) y la semántica. De forma contraria a un tipo de servicio de ODP, que contiene una única interfaz, nuestra definición funcional de servicio podrá mantener el conjunto de interfaces proporcionadas por el servicio, y el conjunto de interfaces que toda instancia de servicio puede requerir de otros componentes cuando se implemente sus interfaces soportadas. La información semántica puede ser descrita mediante pre/post condiciones, y también mediante protocolos de acceso al servicio (coreografía), los cuales especifican el orden relativo en el cual un componente espera que sus métodos sean llamados, y la forma en la que éste llama a otros métodos de los componentes.

La segunda parte describe aspectos extra-funcionales del servicio (como por ejemplo QoS, seguridad, y otros) en una forma similar a como lo hace ODP, es decir, mediante propiedades de servicio.

La tercera parte contiene la información de empaquetamiento acerca de cómo descargar, implantar e instalar el componente COTS que proporciona el servicio requerido, incluyendo detalles de implementación, restricciones del contexto y de la arquitectura, etc.

Finalmente, la información no técnica del servicio que ofrece el componente y de su fabricante, como por ejemplo información de licencia, precio, dirección de contacto, detalles del vendedor, etc.

3.2.2. Requisitos para un servicio de mediación de componentes COTS

A continuación presentamos una lista de características, requisitos o propiedades que debe tener un servicio de mediación de componentes COTS para entornos abiertos. Esta lista ha sido elaborada para poder cubrir las limitaciones detectadas en la función de mediación estándar de ODP para el caso de los componentes comerciales. Tanto el estudio como las limitaciones fueron discutidas en el *Capítulo 1* (sección 1.7).

Propiedad 1 (Modelo de componentes heterogéneo)
Un proceso de mediación no debería estar limitado a un modelo de objetos o componentes, sino que debería ser capaz de trabajar simultáneamente con distintos modelos de componentes y plataformas como CORBA, CCM, EJB, COM, .NET, entre otros. Los servicios de mediación heterogéneos también deberían ser capaces de mediar con múltiples protocolos de acceso al servicio y de ajustarse a la evolución de los actuales modelos y de aquellos nuevos que vayan apareciendo.

Propiedad 2 (Un mediador es más que un motor de búsquedas)
Aunque los servicios de mediación podrían asemejarse a los motores de búsquedas en la web, en realidad, los primeros realizan búsquedas más estructuradas. En un servicio de mediación, las heurísticas de emparejamiento necesitan modelar el vocabulario, las funciones de distancia y las clases de equivalencia en un espacio de propiedades específicas del dominio, además del emparejamiento independiente del dominio basado en palabras claves soportado por los motores de búsqueda de documentos.

Propiedad 3 (Federación)
Para la cooperación entre mediadores, se debería permitir la federación entre servicios de mediación utilizando diferentes estrategias. Por ejemplo, la estrategia tradicional de la federación "directa" obliga a que un servicio de mediación se comunique directamente con aquellos servicios de mediación con los que está federados. Si bien este esquema basado en federación es muy seguro y efectivo, la sobrecarga de la comunicación aumenta conforme lo hace también el número de servicios de mediación federados. Por otro lado, una estrategia de federación "basada en repositorio" permite que múltiples servicios de mediación puedan leer y escribir a la vez en el mismo repositorio, desconociendo cada uno de ellos la presencia de los demás dentro de la federación, permitiendo de esta forma que la aproximación sea escalable. Aunque el principal problema de esta estrategia es la implementación de un repositorio comúnmente accesible, esto no parece ser un inconveniente en Internet ya que los motores de búsquedas pueden trabajar con este tipo de repositorios.

Propiedad 4 (Composición y adaptación de servicios)
Los servicios de mediación actuales utilizan unos emparejamientos del tipo "uno a uno" entre las peticiones de servicio de los clientes y la disponibilidad real los servicios almacenados en los repositorios a los que tienen acceso estos servicios de mediación. Sin embargo, un servicio de mediación también debería proporcionar emparejamientos del tipo "uno a muchos", donde una petición de cliente se podría satisfacer mediante la composición de dos o más servicios disponibles en los repositorios, ofreciendo colectivamente las necesidades de los servicios requeridos. Además de ello, el emparejamiento "uno a uno" es también inadecuado en varias situaciones, por ejemplo, cuando se presentan desajustes de formato, de QoS o desajustes de rendimiento entre el cliente y las instancias de servicio detectadas. Este problema se podría resolver componiendo estas instancias de servicios y utilizando monitores de rendimiento para reducir estas diferencias.

Propiedad 5 (Múltiples interfaces)
En sistemas orientados a objetos los servicios se describen mediante interfaces, y cada objeto mediante una única interfaz (aunque puede ser obtenida a partir de otras mediante herencia múltiple). Sin embargo, puesto que en realidad los componentes software pueden ofrecer más de una interfaz a la vez, sus servicios deberían ser definidos más bien en términos de "conjuntos" de interfaces. De hecho, en actividades de "integración de componentes" es importante tener en cuenta esta visión de los componentes como "conjuntos" de interfaces, ya que en estas actividades de integración se pueden detectar conflictos entre distintos componentes que ofrecen interfaces equivalentes [Iribarne et al., 2002b].

Propiedad 6 (Emparejamientos débiles)
Los actuales procesos de búsqueda y selección de componentes utilizan operaciones de emparejamiento exactas demasiado restrictivas, siendo necesario en muchas ocasiones la utilización de criterios de emparejamiento más relajados. Para el caso de los procesos de mediación de servicios, sobre todo para los que funcionan para sistemas abiertos independientemente extensibles (como Internet) y donde los nombres de los métodos y operaciones hacen referencia a unos servicios ofrecidos, es muy importante considerar el tipo de emparejamiento impuesto (débil o exacto), ya que los servicios seleccionados se obtienen de una forma totalmente arbitraria, no estandarizada y sin procedimientos de consenso. Por esta razón, cuando se construye la lista de los componentes candidatos, un servicio de mediación debería permitir la utilización de emparejamientos parciales para seleccionar (de los repositorios) aquellos componentes comerciales que proporcionan un servicio requerido completo, o una parte del mismo.

Propiedad 7 (Uso de heurísticas y métricas — preferencias)
Un servicio de mediación debería permitir que los usuarios pudieran especificar funciones heurísticas y métricas cuando estos buscan componentes comerciales, especialmente para casos donde se realizan emparejamientos débiles. Así, por ejemplo, el servicio de mediación podría devolver los resultados ordenados según un criterio de búsqueda (p.e., ascendentemente por el precio del producto), o descartar algunas secuencias de búsqueda en repositorios donde éstas se llevan a cabo, o también para evaluar los resultados obtenidos a partir de dichas heurísticas o métricas consideradas (también conocidas como "preferencias").

Propiedad 8 (Extensión de los procedimientos de subtipado)
Los actuales servicios de mediación (basados en el de ODP) organizan los servicios en forma de una jerarquía de tipos de servicios con el fin de facilitar las actuaciones de los procesos de emparejamiento de tipos de servicios. Ligado al emparejamiento de tipos está la noción de subtipado. Un tipo A es un subtipo del tipo B ($A \leq B$) cuando las instancias del tipo A pueden sustituir las instancias del tipo B sin que los clientes perciban dicho cambio [Nierstrasz, 1995]. Actualmente el subtipado se realiza solamente a nivel de signaturas, y se deben definir extensiones para tratar también las comprobaciones a nivel de comportamiento [Leavens y Sitaraman, 2000], protocolos [Canal et al., 2001], QoS, etc.

Propiedad 9 (Extensible y escalable)
Un servicio de mediación debería contemplar cualquier tipo de información acerca de las interfaces de los componentes (o servicios), como las signaturas de las operaciones o de los métodos, o como la información de comportamiento, requisitos extra-funcionales, QoS, información de marketing y datos semánticos del componente. No obstante, un servicio de mediación también debería permitir que los usuarios pudieran incluir, de forma independiente, nuevos tipos de información para las interfaces de los componentes que ellos exportan

(registran). Y a su vez, el servicio de mediación también debería ser capaz de utilizar la funcionalidad y las capacidades de estos nuevos tipos de información, extendidos por el usuario como parte de la interfaz del servicio exportado. Además, independientemente de cómo estén federados y organizados los servicios de mediación, la escalabilidad también debería garantizarse para entornos distribuidos y abiertos a gran escala, como en Internet.

Propiedad 10 (Política "almacenamiento-y-reenvío")
Ante una petición de consulta de servicios por parte de un cliente, un servicio de mediación debería responder con un resultado tras procesar dicha petición de consulta. Dicho resultado, obtenido en una acción de "petición-respuesta" y que se corresponde con un comportamiento "automático" del servicio de mediación, se puede referir a una lista de servicios candidatos que cumplen con las necesidades de servicio impuestas en la petición de consulta, o también puede referirse a un aviso de "fracaso" en el caso de que la búsqueda no haya encontrado ningún servicio que cumpliera las características requeridas en la petición de consulta. Para este último caso, a un servicio de mediación también se le debería poder exigir que una petición de consulta tenga que ser obligatoriamente satisfecha, almacenando para ello la petición de consulta, en el caso de que ésta no pudiera ser satisfecha con la información disponible en ese momento, y posponiendo la respuesta hasta que uno (o varios) proveedor(es) de servicios realice(n) un registro (exportación) de un servicio con las características necesarias para que se cumpla la petición de consulta del cliente. A este comportamiento de "petición-respuesta" se le denomina comportamiento "en espera" o comportamiento de "almacenamiento-y-reenvío" (conocido también como *store and forward*).

Propiedad 11 (Delegación)
En relación con la propiedad anterior, un servicio de mediación también debería permitir la delegación de peticiones de consulta a otros servicios de mediación (conocidos) en caso de que estas no pudieran ser satisfechas por el servicio de mediación actual. También se debería permitir la delegación completa o parcial de peticiones de consulta.

Propiedad 12 (Modelos de almacenamiento por demanda y extracción)
Un modelo por demanda (modelo *push*), normalmente utilizado por los servicios de mediación basados en la función de mediación de ODP, es aquel donde los exportadores directamente contactan con el servicio de mediación para registrar sus servicios. Una alternativa para el registro de servicios, adecuada para sistemas de mediación que trabajan en entornos distribuidos y abiertos a gran escala, consiste en utilizar un modelo de almacenamiento por extracción (modelo *pull*). En este esquema los exportadores, en lugar de contactar directamente con los mediadores, publican los servicios en sus sitios web, para que luego los propios servicios de mediación se encarguen de "rastrear" continuamente la red en busca de estos nuevos servicios. Para ello, se puede hacer uso de procesos *bots* y motores de búsqueda para mejorar el comportamiento de registro de los actuales servicios de mediación.

3.2.3. Repositorios de mediación

Un proceso de mediación realiza las operaciones de exportación e importación de componentes sobre unos repositorios XML que almacenan especificaciones de componentes comerciales basadas en el modelo de documentación de componentes COTS. Como vimos en el *Capítulo 2*, este modelo utiliza el lenguaje COTScomponent para la definición de plantillas de especificación en XML.

El modelo de información de un repositorio de mediación se basa en tuplas de tres elementos cada una de ellas. El primer elemento de tupla contiene la fecha en la que se almacenó la especificación en el repositorio. El segundo elemento de tupla es un identificador de usuario del proveedor que registró la especificación. Para el identificador se puede utilizar alguna codificación estándar, como el UUID de COM. En nuestro caso, el servicio de mediación desarrollado (que presentaremos en la sección 3.4) utiliza un identificador de usuario (p.e., la dirección de correo electrónico). Por último, el tercer elemento de tupla contiene la especificación de componente en notación COTScomponent.

Repositorio: <COTSrepository>

```
<xsd:element name="COTSrepository">
  <xsd:complexType>
    <xsd:element name="COTSRegisterItem" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="COTSdate" type="xsd:date"/>
          <xsd:element name="COTSuserID" type="xsd:string"/>
          <xsd:element name="COTStemplate" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:complexType>
</xsd:element>
```

Figura 3.1: Definición de esquema del repositorio del servicio de mediación

Un servicio de mediación puede acceder a uno o más repositorios de especificaciones de componentes

En la figura 3.1 mostramos una definición de esquema del modelo de información del repositorio que soporta un servicio de mediación. En este esquema las tuplas se representan mediante elementos `COTSRegisterItem`, y el repositorio que las engloba, mediante el elemento contenedor `COTSrepository`. Cada registro o tupla `COTSRegisterItem` define otros tres elementos de esquema para almacenar: la fecha (`COTSdate`), el identificador de usuario (`COTSuserID`) y la plantilla de especificación del componente (`COTStemplate`). En la figura 3.2 mostramos una instancia de repositorio que presenta la estructura definida por el modelo de información anterior. En este ejemplo se muestra parte del contenido de un repositorio de mediador con dos servicios exportados (registrados), un servicio de agenda y un servicio de calendario, ambos definidos mediante plantillas `COTScomponent` (líneas 6–8 y 15–17 de la figura 3.2).

3.2.4. Operaciones de mediación

Las operaciones de mediación permiten controlar el acceso al repositorio, principalmente bajo dos acciones básicas, la acción de exportación de servicios y la acción de importación de servicios. Estas dos acciones son tenidas en cuenta por el modelo de mediación para la definición de dos interfaces de mediación, conocidas como la interfaz `Register` y la interfaz `Lookup`, las cuales están relacionadas con las acciones de exportación e importación de servicios, respectivamente. Estas dos interfaces son similares a las que utiliza el estándar de mediación de ODP (tratado en el *Capítulo 1*, sección 1.7).

En la figura 3.3 mostramos la definición IDL del servicio de mediación `COTStrader`. En total, el servicio de mediación soporta cuatro operaciones: `export()`, `withdraw()` y `replace()`, para acciones de exportación (interfaz `Register`), y `query()` para acciones de importación de servicios (interfaz `Lookup`). En los dos siguientes apartados de esta sección analizaremos estas dos interfaces por separado.

```
1:  <COTSrepository>
2:    <COTSRegisterItem>
3:      <COTSdate>November 14, 2002 4:38:56 PM CET</COTSdate>
4:      <COTSuserID>liribarne@ual.es</COTSuserID>
5:      <COTStemplate>
6:           <COTScomponent name="Agenda">
7:              ...
8:           </COTScomponent>
9:      </COTStemplate>
10:   </COTSRegisterItem>
11:   <COTSRegisterItem>
12:     <COTSdate>November 15, 2002 3:27:11 AM CET</COTSdate>
13:     <COTSuserID>liribarne@ual.es</COTSuserID>
14:     <COTStemplate>
15:          <COTScomponent name="Calendario">
16:             ...
17:          </COTScomponent>
18:     </COTStemplate>
19:   </COTSRegisterItem>
20:    ...
21: </COTSrepository>
```

Figura 3.2: Una instancia ejemplo de un repositorio de servicio de mediación

3.2.4.1. Exportación de servicios (interfaz `Register`)

La actividad de exportación de servicios de componente COTS en un servicio de mediación estará controlada por la interfaz `Register`. Esta interfaz define tres operaciones sobre el repositorio asociado: (1) registrar un nuevo servicio, operación `export()`, (2) eliminar un servicio existente, operación `withdraw()`, y (3) modificar un servicio existente, operación `replace()`. Las tres operaciones de la interfaz de exportación utilizarán documentos `COTScomponent` para llevar a cabo sus respectivas transacciones, y devolverán un valor `TRUE` si la operación tiene lugar satisfactoriamente, o `FALSE` en caso contrario. Para este último caso, las tres operaciones utilizarán el parámetro `results` para devolver una descripción del tipo de fallo que lo ha provocado (en caso de suceder). Una descripción para estas tres operaciones es la siguiente:

(a) Función `export()`. Almacena un servicio de componente en un repositorio de mediación. Esta función generará un registro `COTSRegisterItem` con sus tres elementos: la fecha (`COTSdate`) que deberá ser obtenida del sistema, la identificación de usuario (`COTSuserID`) que vendrá dada por el parámetro `userID`, y la especificación del servicio de componente (`COTStemplate`) en el parámetro `XMLCOTScomponentTemplate`.

(b) Función `withdraw()`. Elimina de un repositorio de mediación el servicio de componente `XMLCOTScomponentTemplate` del usuario dado por `userID`.

(c) Función `replace()`. Reemplaza un registro por otro en el repositorio de mediación. Esta función eliminará del repositorio aquel registro que contiene el servicio de componente `oldXMLCOTScomponentTemplate` y código de usuario `olduserID`, y lo sustituirá por el nuevo registro que se forma a partir del servicio de componente dado por el parámetro `newXMLCOTScomponentTemplate` y el código de usuario en `newuserID`.

```
module COTStrader {
    interface Register {
        boolean export  ( in  string  XMLCOTScomponentTemplate,
                          in  string  userID,
                          out string  results );
        boolean withdraw( in  string  XMLCOTScomponentTemplate,
                          in  string  userID,
                          out string  results );
        boolean replace ( in  string  oldXMLCOTScomponentTemplate,
                          in  string  olduserID,
                          in  string  newXMLCOTScomponentTemplate,
                          in  string  newuserID,
                          out string  results );
    };
    interface Lookup {
        boolean query   ( in  string  XMLCOTSqueryTemplate,
                          in  long    maxCandidates,
                          in  boolean storeAndForward,
                          out long    nHits,
                          out string  templates,
                          out string  results );
    };
};
```

Figura 3.3: Definición de las interfaces del servicio de mediación COTStrader

Por otro lado, por las propiedades que hemos definido para un modelo de mediación de componentes COTS, un mediador deberá permitir que los fabricantes de componentes COTS puedan exportar sus servicios usando un modelo de exportación basado en demanda (modelo *push*) o un modelo basado en extracción (modelo *pull*). El primer caso, el exportador directamente contactará (a petición propia) con el mediador a través de las operaciones de la interfaz `Register`. En el segundo caso (modelo *pull*) el exportador sólo tendrá que dejar el documento `COTScomponent` —con el servicio de componente que éste anuncia— en un lugar accesible por motores de búsquedas web, los cuales serán los encargados de localizar y extraer el documento del servicio anunciado para registrarlo luego en el repositorio del mediador. El objeto encargado de esta tarea, luego tomará el papel de cliente exportador para registrar el servicio usando un modelo de exportación por demanda.

3.2.4.2. Importación de servicios (interfaz `Lookup`)

La actividad de importación de servicios de componente COTS de un servicio de mediación estará controlada por la interfaz `Lookup`. Esta interfaz define una única operación sobre el repositorio, la operación `query()`. La operación `query()` permitirá buscar servicios de componentes COTS bajo unos criterios de consulta. Estos criterios serán establecidos en el parámetro `XMLCOTSqueryTemplate` mediante una plantilla especial de consulta llamada `COTSquery`. Describiremos esta plantilla de consulta en esta sección.

Además de los criterios de consulta, el usuario podrá fijar dos parámetros más en la consulta: (a) el número máximo de candidatos que deberá localizar el mediador en un repositorio (`maxCandidates`), y (b) una política de "almacenamiento-y-reenvío" (*store-and-forward*) para llevar a cabo la consulta (propiedad 10 de un modelo de mediación; véase página 105). Si el parámetro `storeAndForward` se establece con el valor `TRUE` entonces el mediador estará obligado a encontrar el número total de servicios de componente indicado en el parámetro `maxCandidates`. Si en el momento de la consulta el mediador no

puede encontrar el número total de servicios exigido, entonces éste (el mediador) pospondrá su respuesta hasta que los nuevos servicios de componente que se vayan registrando satisfagan el número total exigido. En cualquier caso, tras la consecución de una operación de importación (o consulta) el mediador devolverá el número total de servicios encontrados (`nHits`) y una cadena (`templates`) con una secuencia de plantillas `COTScomponent`, una por cada servicio de componente encontrado. En caso de establecerse `storeAndForward` a `TRUE`, el valor devuelto por `nHits` deberá coincidir con el valor del parámetro `maxCandidates`.

Para importar un servicio, el cliente necesitará proporcionar al mediador dos documentos XML. El primero de ellos, llamado `COTSquery`, contendrá los criterios de selección (de consulta) que deberá utilizar el mediador en las actividades de búsqueda de servicios. El segundo documento describirá dentro del elemento `COTSdescription` de la plantilla de consulta, las características principales del servicio requerido. En la figura 3.4 se muestra la definición de esquema de una plantilla de consulta `COTSquery`.

Lenguaje para definición de consultas: <COTSquery>

```
<xsd:element name="COTSquery">
  <xsd:complexType>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:sequence>
      <xsd:element name="COTSdescription" type="COTSdescriptionType"/>
      <xsd:element name="functionalMatching" type="functionalMatchingType"/>
      <xsd:element name="propertyMatching" type="propertyMatchingType"/>
      <xsd:element name="packagingMatching" type="locationType"/>
      <xsd:element name="marketingMatching" type="locationType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<!-- Declaración de tipos -->
<xsd:complexType name="COTSdescriptionType">
  <xsd:choice>
    <xsd:attribute name="href" type="xsd:uriReference" minOccurs="0"/>
    <xsd:element ref="COTScomponent"/>
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="functionalMatchingType">
  <xsd:sequence>
    <xsd:element name="interfaceMatching" type="matchingType"/>
    <xsd:element name="behaviorMatching" type="matchingType"/>
    <xsd:element name="choreographyMatching" type="matchingType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="matchingType">
  <xsd:sequence>
    <xsd:element name="exactMatching" type="referenceType" minoccurs="0"/>
    <xsd:element name="softMatching"  type="referenceType" minoccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="propertyMatchingType">
  <xsd:sequence>
    <xsd:element name="constraints" type="locationType"/>
    <xsd:element name="preferences" type="locationType"/>
  </xsd:sequence>
</xsd:complexType>
```

Figura 3.4: Definición de esquema para la generación de plantillas de consulta `COTSquery`

Un documento de consulta `COTSquery` contiene: un atributo para indicar el nombre de la consulta, y una secuencia de cinco elementos. El primero de ellos (`COTSdescription`) se refiere a la definición del servicio que se desea buscar, utilizando para ello una plantilla de documento usual (esto es, una plantilla `COTScomponent`). La descripción de un servicio (cuyo comportamiento viene dado por el tipo `COTSdescriptionType`) puede especificarse (a) bien mediante un enlace externo hacia el archivo que contiene el documento de servicio de componente (documento `COTScomponent`), en un atributo del elemento `COTSdescription`, o (b) bien directamente en la plantilla de consulta dentro del elemento `COTSdescription`, utilizando la sintaxis de un documento de componente `COTScomponent`[2].

Los cuatro elementos restantes de una plantilla de consulta `COTSquery` permitirán fijar los criterios de consulta sobre las cuatro partes del documento de componente que se desea buscar, cuya definición quedará establecida previamente por el elemento `COTSdescription` (como un enlace externo, o directamente en la plantilla). Para establecer criterios de consulta de la parte funcional del componente se utiliza el elemento `functionalMatching`. Para la parte extra-funcional se utiliza el elemento `propertyMatching`; para la parte de empaquetamiento el elemento `packagingMatching`; por último, para la parte de marketing se utiliza el elemento `marketingMatching`.

Para establecer una consulta sobre la parte funcional de un componente, es necesario indicar dos cosas, por un lado su descripción funcional y por otro lado la clase de emparejamiento que deberá realizar el mediador en las actividades de búsqueda. La descripción funcional se ofrece dentro de la sección `<functional>` del documento `COTSdescription`. Los operadores de emparejamiento pueden ser de dos tipos, aunque se pueden llevar a cabo a tres niveles diferentes, y que son el operador de emparejamiento fuerte (o exacto) y el operador de emparejamiento débil (o relajado), y los niveles a los que estos afectan son a nivel de interfaces, a nivel de comportamiento y a nivel de protocolos. En la plantilla de consulta `COTSquery` se hace uso de los elementos `interfaceMatching`, `behaviorMatching` y `choreographyMatching`, para especificar los niveles a los que afecta la consulta. Para cada uno de ellos se puede fijar el tipo de operador de emparejamiento que el mediador deberá utilizar para buscar servicios de componentes similares, y para lo cual se utiliza el operador `softMatching`, para establecer un emparejamiento débil, y el operador `exactMatching`, para establecer un emparejamiento fuerte.

En la figura 3.5 se muestra un ejemplo de plantilla de consulta que ilustra el comportamiento descrito hasta el momento para la parte funcional. En la figura se puede ver una plantilla de consulta `COTSquery` para buscar un servicio de componente concreto cuyas características están descritas en un archivo `Consulta1.xml`, dado por el elemento `COTSdescription` (línea 5). En la figura 3.6 mostramos el posible contenido del archivo. En él se puede observar que para definir las características de un servicio de componente buscado se puede hacer uso de una plantilla `COTScomponent` (discutida en el *Capítulo 2*). En la plantilla se puede observar además que el servicio de componente que hay que buscar deberá poseer una única interfaz ofertada y dos propiedades extra-funcionales.

Volviendo a la plantilla de consulta de la figura 3.5, tras la descripción del servicio a buscar (línea 5), en el ejemplo se han establecido criterios de búsqueda para las cuatro partes del documento COTS: criterios a nivel funcional (líneas 7 a 14), criterios a nivel extra-funcional (líneas 16 a 21), criterios a nivel de empaquetamiento (líneas 23 a 27) y criterios a nivel de marketing (líneas 29 a 31).

[2]La cláusula `ref="COTScomponent"` que aparece en la definición de esquema, se refiere a que el elemento `COTScomponent` ya ha sido definido en otro lugar dentro de la gramática. En nuestro caso la gramática completa —tanto para definir plantillas de especificación de componentes como plantillas de consultas— está definida en `http://www.cotstrader.com/COTS-XMLSchema.xsd` y mostrada en el *Apéndice A*

```
1:  <?xml version="1.0"?>
2:  <COTSquery name="UnaConsultaCliente"
3:                  xmlns="http://www.cotstrader.com/COTS-XMLSchema.xsd">
4:
5:    <COTSdescription href=".../Consulta1.xml" />
6:
7:    <functionalMatching>
8:      <interfaceMatching>
9:        <exactMatching href=".../exact2match.cgi" />
10:     </interfaceMatching>
11:     <choreographyMatching>
12:       <softMatching />
13:     </choreographyMatching>
14:   </functionalMatching>
15:
16:   <propertyMatching>
17:     <constraints notation="XQuery">
18:       (securityLevel = SAFE) and (isRunningNow = TRUE)
19:     </constraints>
20:     <preferences notation="ODP">first</preferences>
21:   </propertyMatching>
22:
23:   <packagingMatching notation= "XQuery">
24:     description/notation = "CCM-softpkg" and
25:     ( description/implementation/os/name= "WinNT" or
26:       description/implementation/os/name="Solaris" )
27:   </packagingMatching>
28:
29:   <marketingMatching notation="XQuery">
30:     vendor/address/country = "Spain"
31:   </marketingMatching>
32: </COTSquery>
```

Figura 3.5: Una plantilla de consulta `COTSquery`

A modo de ejemplo, para el caso de la parte funcional (descrita hasta el momento), en la plantilla de consulta se ha definido una sección `<functionalMatching>` donde se han establecido los criterios de emparejamiento para las interfaces y la coreografía. Para las interfaces, se ha establecido un operador de emparejamiento exacto con el elemento `<exactMatching>`. Como atributo, se ofrece también el nombre de un programa CGI que permite hacer emparejamientos sintácticos fuertes. Para el caso de la coreografía se exige un emparejamiento débil, pero sin embargo no se ha indicado ningún programa. Para resolver estos casos (como veremos más adelante), el mediador busca la existencia de un programa de emparejamiento dentro de las plantillas de componente del repositorio, o en su defecto, utiliza un programa de emparejamiento propio del mediador (en caso de disponer uno).

Para la parte extra-funcional se utiliza emparejamiento basado en propiedades, como lo hacen normalmente los servicios de mediación de ODP, esto es, utilizando "restricciones" y "preferencias". Las restricciones son expresiones lógicas (booleanas) consistentes en valores de propiedades de servicio, constantes, operadores relacionales (`<`, `>=`, `=`, `!=`), operadores lógicos (`not`, `and`, `or`) y paréntesis, que especifican los criterios de emparejamiento para

```
1:  <?xml version="1.0"?>
2:  <COTScomponent name="Consulta1"
3:            xmlns="http://www.cotstrader.com/COTS-XMLSchema.xsd">
4:    <functional>
5:      <providedInterfaces>
6:        <interface name="onePlaceBuffer">
7:          <description notation="IDL-CORBA">
8:             interface OnePlaceBuffer {
9:               void write(in long x); long  read();
10:            };
11:         </description>
12:       </interface>
13:     </providedInterfaces>
14:   </functional>
15:   <properties notation="W3C">
16:     <property name="securityLevel" priority=7>
17:       <type>xsd:string</type>
18:     </property>
19:     <property name="isRunningNow" priority=4>
20:       <type>xsd:boolean</type>
21:     </property>
22:   </properties>
23: </COTScomponent>
```

Figura 3.6: Una plantilla de descripción del servicio usado en una consulta `COTSquery`

incluir un componente en la lista de candidatos en las operaciones de búsqueda del mediador. Las restricciones son evaluadas por el mediador sustituyendo los nombres de las propiedades por sus valores actuales, y luego evaluando la expresión lógica. Los componentes cuyas restricciones son evaluadas a falso son descartados. En nuestro caso, para escribir las expresiones de emparejamiento hemos utilizado la notación definida en la propuesta XML Query Algebra del W3C (`http://www.w3.org/TR/query-algebra/`).

El álgebra del W3C recibe el nombre de lenguaje XQuery

Otro aspecto crítico en el proceso de selección (emparejamiento) es resolver los conflictos entre propiedades contradictorias. La asignación de prioridades es una de las posibles soluciones para tratar estos conflictos. Las prioridades pueden ser asignadas a cada propiedad de primer nivel en la etiqueta `properties`, utilizando una escala del 0 (muy baja) al 9 (muy alta), normalmente utilizada en procesos de toma de decisiones.

Las preferencias permiten ordenar la lista de componentes candidatos obtenida por el mediador acorde a unos criterios dados mediante los términos `first` (orden natural de la búsqueda), `random` (ordenados aleatoriamente), `min(expr)` (orden descendente) y `max(expr)` (orden ascendente), donde `expr` es una simple expresión matemática que involucra nombres de propiedades [ISO/IEC-ITU/T, 1997].

Como se observa en la figura 3.5, los criterios de búsqueda extra-funcionales se establecen en la etiqueta `<propertyMatching>`, y las restricciones y las preferencias en los elementos `constraints` y `preferences`, respectivamente. En ambos elementos se utiliza el atributo `notation` para indicar el tipo de notación empleada para describirlos. Para las restricciones se ha utilizado el álgebra del W3C. Para la preferencia de ordenación se ha utilizado la notación ODP (descrita antes). En el ejemplo, la plantilla de consulta obliga a que el mediador devuelva el listado de las plantillas encontradas en orden natural, esto es, en el mismo orden en las que el mediador las va encontrando.

3.3. El proceso de mediación de COTStrader

El proceso de mediación involucra dos actividades principalmente, la actividad de registrar o anunciar servicios de componente COTS y la actividad de buscar servicios de componente COTS con ciertas características. Cuando un proveedor de servicios desee publicar un servicio particular, éste se pondrá en contacto con un mediador a través de la interfaz `Register`, ofreciéndole una plantilla `COTScomponent` con la especificación del componente COTS que desea publicar. Luego, el mediador almacenará esta plantilla en algún repositorio asociado.

Por otro lado, el proceso de mediación como tal se lleva a cabo realmente en las actividades de consulta, a través de la interfaz `Lookup` del mediador. Cuando se recibe una consulta, el mediador intenta buscar aquellas descripciones de componente del repositorio, que casan con la descripción de componente requerida. Las condiciones de la consulta (en contenido y forma) y los criterios de búsqueda deberían estar establecidos adecuadamente en una plantilla de consulta `COTSquery`, ofrecida como argumento en el momento de hacer la llamada a la operación `query()` de la interfaz `Lookup`.

El algoritmo de búsqueda que sigue el mediador COSTtrader comprueba si una plantilla de consulta Q casa con una plantilla T residente en el repositorio asociado. El mediador recorre el repositorio completo y va almacenando aquellas plantillas T que van casando en una lista de candidatos. El algoritmo realiza los siguientes pasos:

Algoritmo de *Selección y búsqueda*

Entrada: Una plantilla de consulta Q y un repositorio β

Para cada plantilla T en el repositorio β **hacer**

Paso 1 Si una etiqueta `<marketingMatching>` está presente en Q entonces la expresión QueryAlgebra es evaluada para los campos de las etiquetas correspondientes de T referenciados en la expresión. Si la expresión se evalúa `FALSE` entonces la plantilla T es descartada. Puede suceder también que ninguno de los campos de la expresión no esté presente en T, en este caso la expresión se evalúa a `TRUE`.

Paso 2 Para la etiqueta `<packagingMatching>` de Q se repite el mismo proceso del paso 1.

Paso 3 Para la etiqueta `<propertyMatching>` de Q, se repite el mismo proceso del paso 1. La etiqueta no solo incluye las propiedades de calidad, sino también las palabras claves o atributos no funcionales. Para el caso de las propiedades dinámicas éstas deberían ser evaluadas antes de evaluar la expresión QueryAlgebra. El criterio descrito en la etiqueta `<preferences>` es usado para insertar T en la lista de candidatos, en el caso de que éste finalmente sea seleccionado.

Paso 4 Para la parte funcional, primero se comprueba si existe una etiqueta `<functionalMatching>` en Q, y luego la información funcional es casada utilizando los programas de emparejamiento correspondientes. Las signaturas (interfaces proporcionadas y requeridas, en ese orden) son casadas en primer lugar, luego los eventos en caso de que hayan sido indicados, luego las coreografías y finalmente los comportamientos. Esto sigue el patrón de "coste-menor" [Goguen et al., 1996], que filtra primero aquellos elementos funcionales fáciles de comprobar.

finPara

finAlgoritmo

En el caso de los elementos funcionales (del paso 4 del algoritmo anterior), las plantillas pueden especificar o no los programas de emparejamiento para cada elemento particular: interfaces, comportamiento o protocolo. Los eventos son emparejados sólo a nivel sintáctico. El algoritmo que selecciona el programa de emparejamiento que debe utilizar el mediador es el siguiente, independientemente del elemento funcional tratado (interfaz, comportamiento o protocolo).

Algoritmo de *Selección del programa de emparejamiento*

Entrada: Q y T, plantilla de consulta y plantilla para ser candidata

Para cada elemento E = `<interfaceMatching>`, `<behaviorMatching>` o `<choreographyMatching>` que aparezca en Q **hacer**

Paso 1 En caso de que se haya especificado un programa de emparejamiento m en la plantilla Q para un elemento particular E, y de que las notaciones en las que vienen descritas los elementos en Q y T sean compatibles, entonces el programa m será utilizado para hacer el emparejamiento de las descripciones para elemento E en cuestión.

Paso 2 Si no se ha detectado ningún programa de emparejamiento m especificado en la plantilla Q, entonces se procede a comprobar si hay uno en la plantilla T, y en caso de detectarse, éste es utilizado para hacer el emparejamiento entre las descripciones de los dos elementos de tipo E, suponiendo que sus notaciones en las que vienen representadas sendas descripciones coinciden.

Paso 3 Si no hay ningún programa de emparejamiento ni en la plantilla Q ni en la plantilla T, entonces se comprueba si el mediador tiene uno por defecto que pueda realizar el emparejamiento para la notación en la que han sido descritos los elementos que deben ser casados.

Paso 4 En cualquier otro caso, el componente es marcado como candidato "potencial", ya que las comprobaciones no han sido posibles para ese elemento particular.

finPara

finAlgoritmo

Tras repetir este proceso para todos los elementos en Q, la plantilla T es: (a) descartada si cualquiera de las pruebas ha fallado, (b) es incluida en la lista de candidatos si todas las pruebas se han llevado a cabo con éxito, o (c) es considerada como candidata "potencial" si todas las pruebas se han pasado satisfactoriamente porque no se ha encontrado un programa de emparejamiento para algún elemento funcional particular. La forma de tratar los candidatos "potenciales" dependerá del orden de emparejamiento seleccionado por el cliente. Además:

1. En caso de ser un emparejamiento débil (`softmatching`), T es incluida en la lista de candidatos cuando todas las partes han pasado las pruebas.
2. En caso de ser un emparejamiento fuerte (`exactmatching`), nuestro algoritmo compara sólo la información de signaturas, comprobando que:
   (a) todas las interfaces proporcionadas y eventos consumidos definidos en Q están presentes en T y ser sintácticamente iguales.
   (b) los conjuntos de las interfaces requeridas y los eventos emitidos en Q son un súper conjunto de aquellos en T, por medio de un emparejamiento sintáctico.

3.4. Implementación del modelo de mediación

COTStrader es un servicio de mediación implementado a partir del modelo de mediación de componentes COTS que hemos propuesto en este capítulo. La implementación del modelo —que discutiremos en la presente sección— ha sido desarrollada en Java, CORBA y XML. Para ello, se han utilizado las herramientas Orbacus [OOC, 2001] para implementar la parte CORBA, la API XML4J de IBM para implementar el repositorio de plantillas de componente COTScomponent, y la utilidad XQEngine (`http://www.fatdog.com`) que implementa una versión de la especificación del XML Query Algebra del W3C (`http://www.w3.org/TR/query-algebra/`).

Para la implementación del modelo, hemos partido de la definición IDL de un mediador COTS, ofrecida en la figura 3.3. Luego, hemos compilado la descripción IDL de COTStrader usando el compilador de IDL que ofrece la herramienta Orbacus para Java, de la forma:

```
$jidl COTStrader.idl
```

Esta operación genera la colección de clases CORBA necesaria para la implementación y ejecución del resto de las clases. En la figura 3.7 se muestra esta colección de clases y sus relaciones.

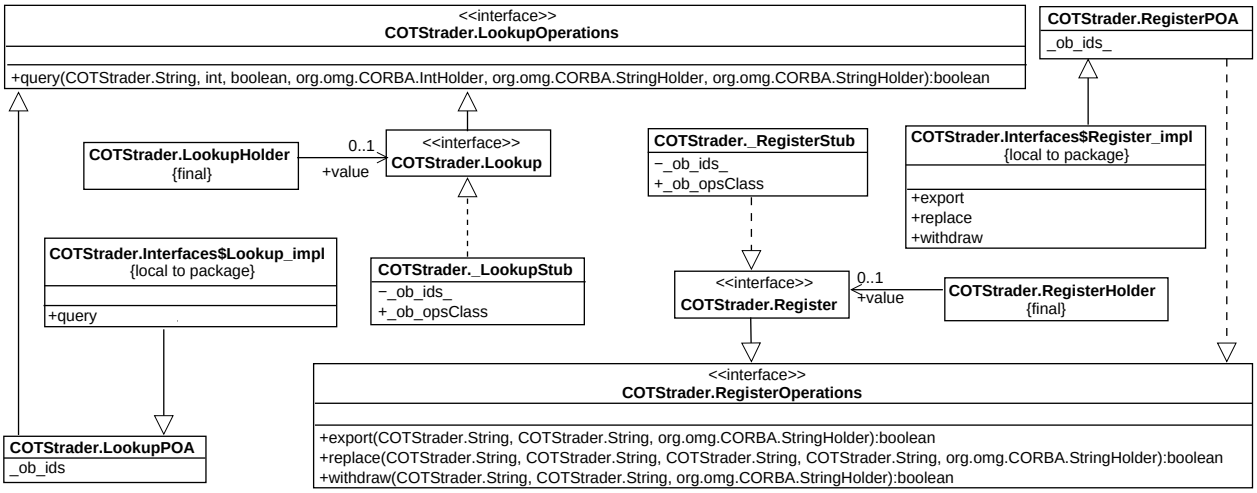

Figura 3.7: Modelando las interfaces de COTStrader

Para la implementación del servicio de mediación COTStrader han sido necesarias desarrollar 53 clases en 4 paquetes. Estos han sido:

— el paquete `com.costrader.utils` utilizado para dar un comportamiento especial a las funciones de utilidad general de fecha, formato y archivo (3 clases),

— el paquete `com.costrader.xml` para funciones particulares relacionadas con XML y no disponibles en el analizador usado (9 clases),

— el paquete `com.costrader.XQuery` para funciones particulares relacionadas con el álgebra de consulta y no disponibles en el paquete usado para ello (8 clases), y

— el paquete principal `COTStrader` con la funcionalidad del mediador (33 clases). De las 33 clases del paquete principal, 12 han sido generadas directamente por el compilador de CORBA (`jidl`), las mostradas en la figura 3.7, y el resto (21 clases) implementadas para ofrecer funcionalidad ligada a las clases `Register_impl` y `Lookup_impl`, las cuales establecen los puntos de entrada al objeto COTStrader.

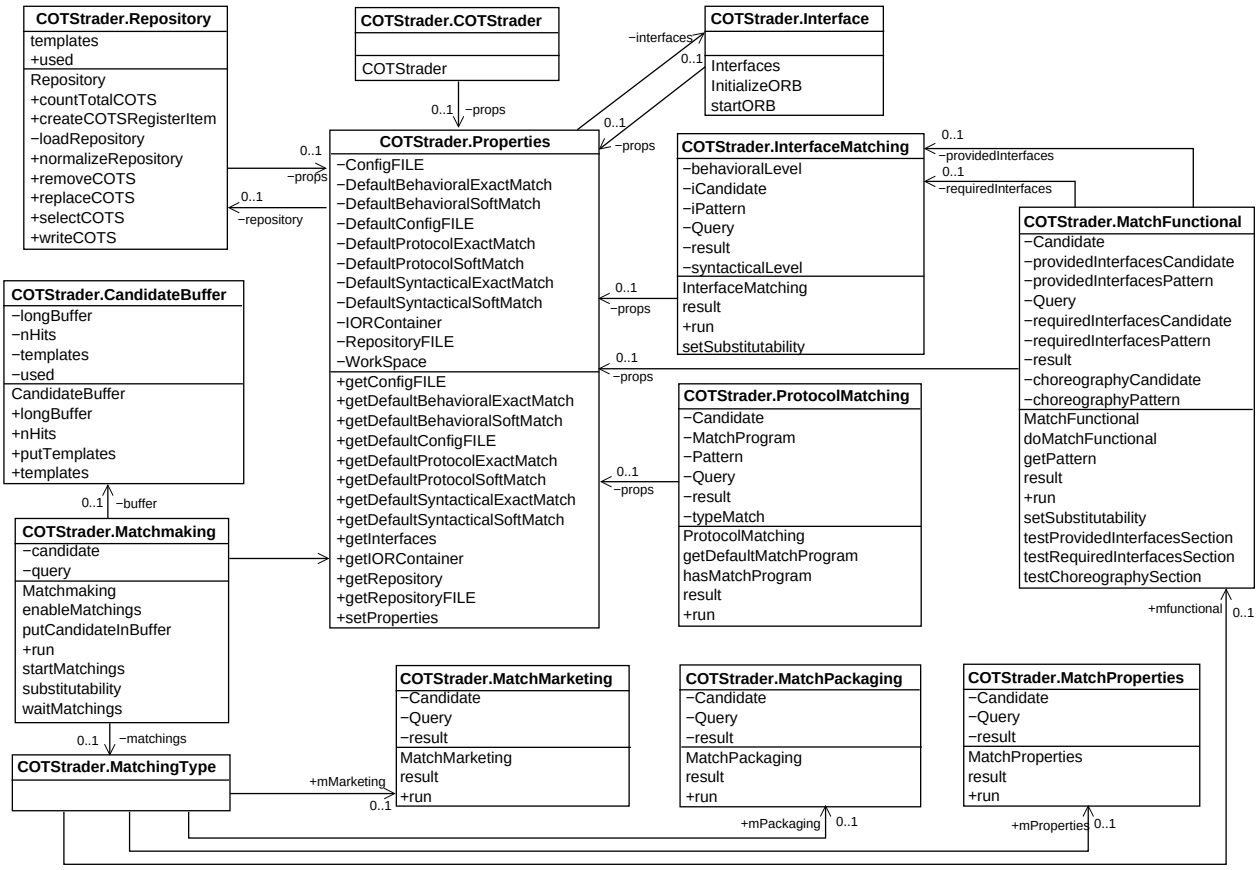

Figura 3.8: Diagrama de clases del servicio de mediación COTStrader

En la figura 3.8 mostramos un diagrama con la relación entre algunas de estas clases del paquete principal del servicio COTStrader. Usaremos este diagrama de clases como punto de referencia para describir a continuación parte de la implementación desarrollada del servicio de mediación. Para no complicar la exposición, en algunas ocasiones mostraremos el código Java de clase sin llegar a indicar todos los parámetros, atributos y operaciones que ésta implementa, ya que estos han sido mostrados en la figura 3.8.

En la figura 3.9 mostramos de forma resumida, y mediante un diagrama de bloques, las partes más importantes de la arquitectura general del servicio de mediación COTStrader, aunque hay otras muchas partes que no hemos incluido para no complicar el dibujo. Estas partes básicamente son, por un lado el repositorio del mediador (discutido en la sección 3.2.3), y por otro, las interfaces de mediación usadas para la exportación e importación (consulta) de servicios de componentes COTS (interfaces discutidas en la sección 3.2.4).

En la implementación del servicio de mediación COTStrader se hace uso de dos contenedores `InterfacesHandler` y `RepositoryHandler`, que mantienen el control concurrente de las interfaces de mediación y del repositorio. Ambos contenedores facilitan la comunicación entre sus partes mediante un bloque que se encarga de mantener el estado global del mediador. A este estado lo hemos denominado en la implementación como `Properties`. El estado del mediador mantiene información estática (establecida en diseño o cargada desde archivo en el momento que se activa el mediador) e información dinámica (establecida en tiempo de ejecución). Esta información se refiere a aspectos como por ejemplo, el nombre del archivo de configuración que contiene los enlaces a los programas de emparejamiento por omisión del mediador, el lugar donde CORBA registra las referencias de objeto IOR,

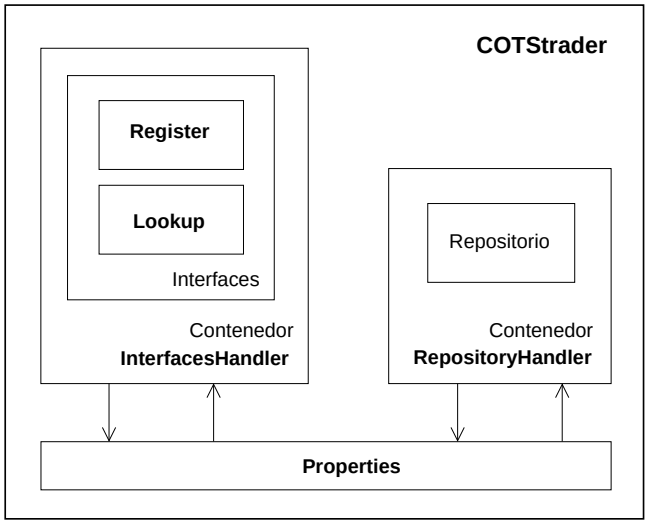

Figura 3.9: Arquitectura del servicio de mediación COTStrader

el lugar donde está el repositorio físico con las plantillas de componentes, o un espacio de memoria compartida requerida para algunas partes de la implementación.

En la implementación desarrollada, la clase principal es la clase `COTStrader`, desde donde se inicia el flujo de control del resto de las clases del servicio de mediación. La clase `COTStrader` está implementada como sigue:

```
public class COTStrader {
   private Properties props = new Properties(); // Estado común a toda clase
   COTStrader() {
      props.setProperties( props.getDefaultConfigFILE() );
   }
   COTStrader( String FILE ) {
      props.setProperties( FILE );
   }
}
```

La clase comienza declarando y creando como atributo privado un objeto `Properties` para mantener el estado general asociado al servicio de mediación. Como hemos adelantado antes, este estado contiene una serie de parámetros estáticos y dinámicos que son visibles y compartidos por el resto de las clases del paquete principal. Algunos de estos parámetros se establecen automáticamente desde un archivo de configuración, cuando se activa el servicio de mediación. Para activar el servicio de mediación se puede hacer de dos formas: (a) llamando directamente al servicio sin ningún parámetro, en este caso el servicio de mediación utilizará por defecto un archivo de configuración asociado, y (b) llamando al servicio con un archivo de configuración determinado, en tal caso el servicio de mediación utilizará las declaraciones de configuración del archivo ofrecido en la llamada. A modo de ejemplo, a continuación mostramos el archivo de configuración usado para activar el servicio de mediación.

```
# CONFIGURACION

REPOSITORY=COTStrader/COTSrepository.rep
IOR=COTStrader/imr/
```

```
# Variables para los programas de emparejamiento

notation                 = CORBA-IDL
 SYNTACTICAL_SOFTMATCH   = http://localhost:8080/servlet/syntacticalSoftMatch
 SYNTACTICAL_EXACTMATCH  = http://localhost:8080/servlet/syntacticalExactMatch
notation                 = JML
 BEHAVIORAL_SOFTMATCH    = http://localhost:8080/servlet/behavioralSoftMatch
 BEHAVIORAL_EXACTMATCH   = http://localhost:8080/servlet/behavioralExactMatch
notation                 = pi-protocol
 PROTOCOL_SOFTMATCH      = http://localhost:8080/servlet/protocolSoftMatch
 PROTOCOL_EXACTMATCH     = http://localhost:8080/servlet/protocolExactMatch

# Espacio de trabajo
WORKSPACE=COTStrader/tmp/
```

Como vemos en la configuración, primero se establece el parámetro `REPOSITORY` con el nombre del archivo donde se ubica el repositorio de plantillas COTScomponent asociado al servicio de mediación. Si existe un repositorio, éste es cargado en memoria y controlado por la clase `RepositoryHandler`. En caso contrario, el nombre del archivo es utilizado para permitir la persistencia del repositorio.

El parámetro de configuración `IOR` se utiliza para establecer el contenedor donde se debe registrar las referencias únicas de objeto (conocidas como IOR) que genera CORBA cada vez que se activa el servicio de mediación.

En la configuración del servicio de mediación se pueden establecer también los programas de emparejamiento por defecto que debe utilizar el mediador en las tareas de búsqueda y selección. Como vimos en la sección anterior, sólo se utilizan estos programas de emparejamiento si el mediador no detecta ninguno en la plantilla de consulta ni en las plantillas candidatas. En el ejemplo de configuración mostrado anteriormente se han declarado dos tipos de programas (uno fuerte y otro débil) para cada uno de los niveles de emparejamiento funcional: nivel sintáctico (o de signaturas de interfaces), nivel de comportamiento y nivel de protocolos (o coreografía). No obstante, aquí también se podrían indicar varios programas de emparejamiento para un mismo nivel, pero que funcionen para diferentes notaciones. Aunque en nuestro caso no hemos desarrollado ningún programa de emparejamiento, la implementación sí que está preparada para simplemente hacer una llamada al programa que se especifique en la configuración. A modo de ejemplo, en la configuración mostrada arriba, `behavioralSoftMatch` sería el nombre del programa usado por defecto por el mediador para hacer emparejamientos suaves entre dos descripciones de comportamiento de componente definidas en notación JML.

Por último, el parámetro de configuración `WORKSPACE` es necesario para especificar un nombre de directorio para las necesidades de espacio de trabajo requeridas por la implementación de COTStrader.

A continuación analizaremos algunos detalles de implementación para ciertas clases del paquete principal de COTStrader, concretamente para las clases `Properties`, `Repository`, `Interfaces`, `Register_impl`, `Lookup_impl` y `Matchmaking`.

3.4.1. La clase `Properties`

La clase `Properties` es el núcleo principal de la implementación de `COTStrader`, ya que ésta se encarga de mantener las variables comunes a todas las clases así como las variables de estado, mensajes, constantes de uso por defecto o zonas de acceso común. Respecto a esto último, algunas de las clases han sido implementadas para que sean ejecutadas concurrentemente, accediendo sus métodos simultáneamente a ciertos recursos compartidos

definidos en la clase `Properties`. Para la programación concurrente, estas clases extienden la clase `Thread` y sus métodos declaran la cláusula `syncronized` y usan las ordenes `wait()` y `join()` para sincronizar el flujo de control.

Como se ha visto, la clase `COTStrader` crea un objeto `Properties`, para iniciar el funcionamiento del mediador, mediante la orden `setProperties(`*`configuracion`*`)`, siendo el parámetro *`configuracion`* un nombre del archivo de configuración ofrecido en el momento de la llamada, o un archivo de configuración que tiene asignado por defecto el mediador. Por tanto, la función del objeto `Properties` es obtener los parámetros del archivo de configuración, fijar una serie de variables, mensajes y recursos compartidos, y activar tanto el repositorio como los dos objetos CORBA: (a) uno que encapsula la funcionalidad de la interfaz `Register` (para la exportación de servicios de componente COTS), y (b) otro que encapsula la funcionalidad de la interfaz `Lookup` (para la importación o consulta de servicios de componente COTS[3]). Parte del código Java de la clase `Properties` es como sigue:

```
package COTStrader;
import com.cotstrader.utils.file;
public class Properties extends Constants {
   // ESTADO: atributos de estado, mensajes, constantes y zonas compartidas.
   private Repository repository    = REPOSITORY_NOT_INITIALIZED;
   private Interfaces interfaces    = INTERFACES_NOT_INITIALIZED;
   private String RepositoryFILE    = VALUE_NOT_SPECIFIED;
   private String ConfigFILE        = VALUE_NOT_SPECIFIED;
   private String DefaultConfigFILE = "COTStrader/COTStrader.config";
   ...
   // MÉTODOS: operaciones de la clase
   public void setProperties( String Name ){
      ConfigFILE = Name;
      /** Se establecen los atributos de ESTADO */
      RepositoryFILE = tool.getVariable("REPOSITORY", ConfigFILE);
      IORContainer = tool.getVariable("IOR", ConfigFILE);
      WorkSpace = tool.getVariable("WORKSPACE", ConfigFILE);
      ...
      try {
         /** Se activan las interfaces y el repositorio */
         RepositoryHandler rep_thread = new RepositoryHandler(this);
         InterfacesHandler int_thread = new InterfacesHandler(this);
         rep_thread.start(); int_thread.start(); // se inicia la activación
         rep_thread.join(); int_thread.join();   // se espera mensajes OK
         repository = rep_thread.value;
         interfaces = int_thread.value;
      }
      catch ( InterruptedException exc ) {}
   }
   public Repository getRepository() { return repository; }
   public Interfaces getInterfaces() { return interfaces; }
   public String getRepositoryFILE() { return RepositoryFILE; }
   public String getIORContainer()   { return IORContainer; }
   ...
}
```

[3]Por las características de CORBA, aunque inicialmente se activan dos objetos para la exportación e importación de servicios, CORBA creará un hilo de ejecución para atender una petición de cliente cada vez que se realiza una llamada a una de las interfaces del componente.

En su declaración, la clase `Properties` extiende la clase `Constants`, en la cual se han definido las constantes de entorno. Al inicio del código se ha establecido la cláusula `package COTStrader` para indicar que la clase `Properties` pertenece al paquete `COTStrader` (uno de los cuatro citados anteriormente). El resto de las clases que discutiremos a continuación también comienzan por esta cláusula y otras del tipo `import` requeridas, aunque no las mostraremos en el código para simplificar.

La clase `Properties` comienza declarando los atributos o variables de estado, mensajes, constantes y zonas de acceso compartido (discutido anteriormente). Entre los métodos que implementa, el más importante es `setProperties`. En él primero se establecen todas las variables de entorno (del estado) obtenidas desde la configuración (determinada por el parámetro `Name` en la llamada), y luego se crean los contenedores `RepositoryHandler` y `InterfacesHandler` (véase figura 3.9) en donde se activa y mantiene respectivamente el repositorio y las interfaces del mediador. Estos dos contenedores se activan y ejecutan concurrentemente, e inician el control del mediador.

La clase `RepositoryHandler` contiene el repositorio que usa COTStrader, y es como se muestra en el código siguiente. Como vemos, el repositorio se inicia en el método `run()`, al cual se le pasa como parámetro una referencia al estado.

```
public class RepositoryHandler extends Thread {
   public Repository value;              // mantiene el repositorio
   private Properties sharedProperties;  // mantiene una referencia al estado
   /** Constructor de clase */
   RepositoryHandler( Properties propertiesREF ) {
      super("RepositoryHandler");
      sharedProperties = propertiesREF;
   }
   /** Acción que realiza el contenedor */
   public void run() {
      value = new Repository(sharedProperties);  // se inicia el repositorio
      System.out.println("Trader repository: Ok");
   }
}
```

Una implementación muy parecida se da para el contenedor `InterfacesHandler` que mantiene las otras dos partes del mediador, las interfaces de exportación e importación. Aquí también se inician las interfaces en el método `run()` con una referencia hacia el estado.

```
public class InterfacesHandler extends Thread {
   public Interfaces value;              // mantiene las interfaces
   private Properties sharedProperties;  // mantiene una referencia al estado
   /** Constructor de clase */
   InterfacesHandler( Properties propertiesREF ) {
      super("InterfacesHandler");
      sharedProperties = propertiesREF;
   }
   /** Acción que realiza el contenedor */
   public void run() {
      value = new Interfaces(sharedProperties);  // se inician las interfaces
      System.out.println("Interfaces: Ok");
   }
}
```

Veamos a continuación algunos detalles importantes para estas tres partes del mediador (el repositorio, la interfaz de importación y la interfaz de registro), comenzando por el repositorio (clase `Repository`).

3.4.2. La clase `Repository`

Para la implementación del repositorio se ha utilizado una API DOM para tratar con el formato XML de los registros del repositorio. La estructura del repositorio fue discutida en la sección 3.2.3. Como vimos allí, el repositorio del mediador queda identificado por un documento XML cuyo elemento raíz es `<COTSrepository>`, dentro del cual se mantienen los registros mediante elementos `<COTSRegisterItem>`. La clase que implementa la funcionalidad del repositorio es como sigue:

```
public class Repository extends Constants {
   private Properties props = PROPERTIES_NOT_INITIALIZED;
   static Document templates;                  // El repositorio
   public boolean used = false;                // Usado para sincronización
   Repository ( Properties propertiesREF ) {   // Constructor
      props = propertiesREF;                   // Una referencia al estado
      templates = loadRepository();            // Se carga el repositorio
   }
   private Document loadRepository() { ... }
   public  Element createCOTSRegisterItem ( ... ) { ... }
   public  Node selectCOTS ( int index ) { ... }
   public  boolean replaceCOTS(Element XML1, Element XML2) { ... }
   public  boolean removeCOTS(Element XML) { ... }
   public  boolean writeCOTS ( Element XML ) { ... }
   ...
}
```

La clase `Reposity` utiliza tres atributos en su inicio, uno (`props`) con una referencia al estado con las variables, mensajes, constantes y zonas compartidas por otras clases, el segundo (`templates`) para mantener la colección de registros DOM del repositorio, y el tercero (`used`) es necesario para aspectos de sincronización. En su activación, la clase `Properties` carga en memoria el árbol DOM completo `<COTSrepository>` con los registros del repositorio, si este ya existe; en caso contrario, la clase crea un repositorio vacío preparado para gestionar nuevos registros `<COTSRegisterItem>`. Esta gestión se lleva a cabo a través de los métodos que la clase implementa, los cuales permiten crear un nuevo registro (`createCOTSRegisterItem()`), seleccionar un registro (`selectCOTS()`), reemplazar un registro existente por otro (`replaceCOTS()`), eliminar un registro del repositorio (`removeCOTS`) y guardar un nuevo registro en el repositorio (`writeCOTS()`), entre otros métodos. Estos métodos del repositorio son los que luego utilizan las interfaces `Lookup` y `Register` para implementar sus propios métodos.

3.4.3. La clase `Interfaces`

La clase `Interfaces` implementa la funcionalidad para activar los objetos ORB de las interfaces de exportación (`Register`) y de importación o consulta (`Lookup`), y también implementa la funcionalidad de los métodos para ambas interfaces. Un esquema de la clase `Interfaces` es como sigue:

```
public class Interfaces extends Constants {
   public Properties props = PROPERTIES_NOT_INITIALIZED;
   Interfaces( Properties propertiesREF ) {          // Constructor
       props =  propertiesREF; initializeORB();
   }
   void initializeORB() { ... }                      // Se preparan los ORBs,
   void startORB ( org.omg.CORBA.ORB orb )           // y se activan:
```

```
        throws org.omg.CORBA.UserException { ... }
   class startLookup extends Thread { ... }           // un ORB para Lookup
   class startRegister extends Thread { ... }         // y otro para Register
   class Lookup_impl extends LookupPOA { ... }        // La lógica de Lookup
   class Register_impl extends RegisterPOA { ... }    // La lógica de Register
}
```

En su llamada, el constructor de la clase se encarga de activar tanto los canales ORB asociados a las dos interfaces como las propias interfaces. La inicialización del ORB se lleva a cabo en el método `initializeORB()` de la siguiente forma:

```
void initializeORB() {
   String args[]={""};                                    // necesario por el ORB
   java.util.Properties props = System.getProperties();  // propiedades de un ORB
   props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
   props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");
   try {
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props); // se inicia
      startORB(orb);                                                // se activa
   } catch(Exception ex) { ... }
}
```

La activación de los dos objetos ORB del mediador se lleva a cabo concurrentemente en el método `startORB()`, de la forma usual a como se hace en código CORBA. La interfaz de exportación se activa con `startRegister()` y la interfaz de importación se activa con `startLookup()`.

```
void startORB ( org.omg.CORBA.ORB orb )
   throws org.omg.CORBA.UserException {
   org.omg.PortableServer.POA rootPOA = org.omg.PortableServer.POAHelper.narrow
                                    (orb.resolve_initial_references("RootPOA"));
   org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();
   try {
      startLookup   thread_Lookup   = new startLookup(orb);
      startRegister thread_Register = new startRegister(orb);
      thread_Lookup.start(); thread_Register.start();
      thread_Lookup.join(); thread_Register.join();
   } catch ( InterruptedException exc ) {}
   manager.activate();
   orb.run();
}
```

3.4.4. La clase `Register_impl`

La implementación de la interfaz `Register` es recogida en la clase `Register_impl`[4], la cual contiene métodos para exportar, eliminar y reemplazar servicios de componente COTS en el mediador. Estos métodos utilizan tipos de datos propios del compilador CORBA utilizado, y que en nuestro caso ha sido Orbacus. Al compilar el IDL de COTStrader con la herramienta `jidl`, el compilador genera las clases `Register_impl` y `Lookip_impl` (entre

[4]La herramienta de compilación `jidl` que ofrece Orbacus (y otras en el mercado) genera una colección de archivos (véase figura 3.7) a partir de una definición "`.idl`". Entre estos archivos, la compilación también crea prototipos de archivos de clases vacíos `xxx_impl.java` por cada interfaz definida en el IDL. Aunque no es necesario seguir esta norma, en nuestro caso hemos respetado la nomenclatura `xxx_impl` para hacer referencia a la implementación de las interfaces `Register_impl` y `Lookup_impl`.

otras) para las interfaces `Register` y `Lookup`, respectivamente (interfaces declaradas por el IDL). El compilador traslada las operaciones declaradas en cada interfaz del IDL a prototipos de método para cada clase. En esta conversión, el compilador utiliza los tipos de datos propios del compilador CORBA Orbacus para el caso de los tipos de datos de los parámetros de salida de un método.

```
class Register_impl extends RegisterPOA {
   public synchronized boolean export (  String XMLCOTScomponentTemplate,
                     String userID, org.omg.CORBA.StringHolder comments ) {
      ...
   }
   public synchronized boolean withdraw ( String XMLCOTScomponentTemplate,
                  String userID, org.omg.CORBA.StringHolder comments) {
      ...
   }
   public synchronized boolean replace ( String oldXMLCOTScomponentTemplate,
          String olduserID, String newXMLCOTScomponentTemplate,
          String newuserID, org.omg.CORBA.StringHolder comments ) {
      ...
   }
}
```

De los tres métodos que implementa la clase, el primero de ellos (`export()`) se utiliza para llevar a cabo una exportación o almacenamiento de un servicio de componente COTS en el mediador. La implementación del método es como sigue:

```
public synchronized boolean export (  String XMLCOTScomponentTemplate,
                  String userID, org.omg.CORBA.StringHolder comments ) {
   boolean exit = false;
   try {
     Element XML = (props.getRepository()).createCOTSRegisterItem
                      (XMLCOTScomponentTemplate,userID);
     exit = (props.getRepository()).writeCOTS(XML);
     comments.value = props.generateExit(exit,"export");
   } catch ( Exception e ) { ... }
   notifyAll();  // notifica a todos que ha terminado
   return exit;
}
```

Como tratamos en su momento, para exportar un servicio de componente COTS es necesario ofrecer al mediador una plantilla de documento COTScomponent y el código del usuario que la exporta (que en nuestro caso será una dirección de correo electrónico). Estos dos datos son ofrecidos como entrada en el método `export()` en las variables `XMLCOTScomponentTemplate` y `userID`, respectivamente. En su implementación, el método comienza generando un registro `<COTSregisterItem>` a partir de los dos datos de entrada. En esta operación, internamente también se comprueba primero si la plantilla COTS que se desea registrar es correcta tanto sintáctica como semánticamente (si es un tipo de documento COTSdocument). Si se crea el registro correctamente (`XML != null`) entonces el mediador procede almacenarlo dentro del árbol DOM que mantiene la clase del repositorio (para acceder al repositorio previamente se obtiene una referencia al mismo de la forma `props.getRepository()`). El método devuelve un valor `true` o `false` si la operación de registro se ha llevado a cabo con éxito o se ha detectado algún error. El método también devuelve una cadena como parámetro con información extra (a nivel de comentarios) sobre el resultado producido.

Para eliminar un servicio de componente COTS se utiliza el método `withdraw()`, cuya implementación Java es como sigue:

```
public synchronized boolean withdraw ( String XMLCOTScomponentTemplate,
               String userID, org.omg.CORBA.StringHolder comments) {
   boolean exit = false;
   try {
     Element XML = (props.getRepository()).createCOTSRegisterItem
                     (XMLCOTScomponentTemplate,userID);
     exit = (props.getRepository()).removeCOTS(XML);
     comments.value = props.generateExit(exit,"withdraw");
   } catch ( Exception e ) { ... }
   notifyAll();  // notifica a todos que ha terminado
   return exit;
}
```

Para este método (al igual que `export()`) también es necesario indicar el servicio que se quiere eliminar y el código del usuario. De nuevo el método comienza creando un registro XML a partir de los dos datos de entrada, y luego elimina el servicio haciendo una llamada al método del repositorio asociado encargado de ello (`removeCOTS()`). También aquí se genera un valor de devolución `true` o `false` en función del resultado obtenido tras la operación de eliminación, y se genera un mensaje de texto en la salida.

Finalmente, la clase `Register_impl` implementa un tercer método llamado `replace()` para reemplazar un servicio de componente COTS mantenido por el mediador. La implementación del método es como sigue:

```
public synchronized boolean replace ( String oldXMLCOTScomponentTemplate,
       String olduserID, String newXMLCOTScomponentTemplate,
       String newuserID, org.omg.CORBA.StringHolder comments ) {
   boolean exit = false;
   try {
     Element oldXML = (props.getRepository()).createCOTSRegisterItem
                        (oldXMLCOTScomponentTemplate,olduserID);
     Element newXML = (props.getRepository()).createCOTSRegisterItem
                        (newXMLCOTScomponentTemplate,newuserID);
     exit = (props.getRepository()).replaceCOTS(newXML,oldXML);
     comments.value = props.generateExit(exit,"replace");
   } catch ( Exception e ) { ... }
   notifyAll();  // notifica a todos que ha terminado
   return exit;
}
```

El funcionamiento de este método es similar a los dos anteriores, solo que en este caso es necesario ofrecer el servicio que hay que reemplazar y el nuevo servicio, y los códigos de los usuarios (del servicio que hay que reemplazar y del nuevo). Generalmente los códigos de usuario serán el mismo, aunque con esta opción se permite modificar el existente por uno nuevo. En la implementación del método primero se generan los documentos DOM a partir de los datos de entrada, el primer documento (un elemento `COTSRegisterItem`) permitirá localizar el registro que hay que reemplazar dentro del árbol DOM (el repositorio) por el segundo documento. Esta operación se realiza mediante una llamada al método `replaceCOTS()` de la clase repositorio asociado.

3.4.5. La clase `Lookup_impl`

La implementación de la interfaz `Lookup` es recogida en la clase `Lookup_impl`, la cual contiene el método `query()` usado para buscar servicios de componente COTS en el mediador.

```
class Lookup_impl extends LookupPOA {
   public boolean query ( String XMLCOTSqueryTemplate, int maxCandidates,
          boolean storeAndForward, org.omg.CORBA.IntHolder nHits,
          org.omg.CORBA.StringHolder templates,
          org.omg.CORBA.StringHolder comments )  { ... }
}
```

Como sucede para la clase `Register`, el método `query()` utiliza tipos de datos propios del compilador CORBA utilizado, para el caso de los tipos de datos de los parámetros de salida. La implementación del método es como sigue:

```
public boolean query ( ... )  {
   boolean exit = false;
   /** Se crea un buffer para almacenar la lista de candidatos */
   CandidateBuffer container = new CandidateBuffer(maxCandidates);
   /** Se genera la plantilla de consulta */
   Node theQuery = ((new makeDocument()).fromString(XMLCOTSqueryTemplate))
                   .getDocumentElement();
   /** Si storeAndForward es true, hay que encontrar al menos 1 candidato */
   int minRequired = storeAndForward ? 1 : 0;
   while ( !exit ) { // Se inicia la búsqueda de los candidatos
      /** Se obtiene un posible candidato del repositorio */
      Node candidate = (props.getRepository()).selectCOTS();  // posible
      /** y luego se enfrenta a la plantilla de consulta */
      Matchmaking match = new Matchmaking(theQuery,candidate,container,props);
      match.start();
      /** Las condiciones de la parada de la búsqueda */
      if ((minRequired <= container.nHits()) &&
          (container.nHits() == maxCandidates)) exit = true;
   }
   /** Se preparan las salidas */
   templates.value = container.templates();            // lista de candidatos
   nHits.value = container.nHits();                    // total encontrados
   comments.value = props.generateExit(exit,"query");  // comentarios salida
   return exit;
}
```

En primer lugar se crea un buffer `container` (controlado por la clase `CandidateBuffer`) donde el mediador irá almacenando las especificaciones de servicio de componente que vaya encontrando y que casen con las restricciones de consulta especificadas en la plantilla `XMLCOTSqueryTemplate`. Como la plantilla de consulta `XMLCOTSqueryTemplate`, en su llamada, es del tipo cadena (`String`), primero se genera una plantilla DOM a partir de esta información, plantilla que será utilizada para hacer la búsqueda. Luego se establece el número mínimo de elementos que hay que devolver. Esto se puede interpretar a partir del valor ofrecido en la llamada para el parámetro `storeAndForward`. Si el valor de este parámetro es `true` entonces el mediador estará obligado a devolver como mínimo un servicio de componente COTS que cumpla con las restricciones impuestas en la consulta. También se puede imponer un número máximo de servicios devueltos por el mediador en el parámetro `maxCandidates`, aunque su valor no obliga a que éste tenga que alcanzar el número máximo indicado.

El proceso de búsqueda de candidatos es controlado por la variable `exit`, la cual es comprobada continuamente en un bucle `while` que funciona de forma indefinida. Aquí se van seleccionando uno a uno los servicios del repositorio y se van enfrentando a la plantilla de consulta. Los enfrentamientos (emparejamientos) se hacen de forma concurrente en la clase `Matchmaking`, creando un hilo de trabajo para cada uno ellos (como veremos más adelante, la clase `Matchmaking` extiende la clase `Thread` para permitir hilos múltiples). Esto quiere decir que el mediador empieza "lanzando" hilos de emparejamiento (`Matchmaking`) entre servicios de componente y la consulta (un hilo por cada servicio que le vaya pasando el repositorio) hasta que los resultados que estos generan concurrentemente activan la condición de parada. Para ello, cada vez que se crea un hilo de emparejamiento (`match`) se le pasa una referencia hacia el buffer `container` que mantiene la lista de candidatos. Ya que varios hilos acceden simultáneamente al mismo buffer (la lista de candidatos), cuando éste se crea, se le indica la longitud máxima que puede tener, para evitar así el desbordamiento en el número de resultados exigidos (y consumir memoria innecesaria). La longitud del buffer estará determinada por el número de candidatos que como máximo el mediador debe encontrar (`maxCandidates`).

Internamente a una operación de emparejamiento (controlado por la clase `Matchmaking`), si el emparejamiento se efectúa con éxito, entonces el mediador incorpora el servicio en cuestión como servicio candidato dentro del buffer `container`. Externamente a un emparejamiento, en el bucle de la búsqueda se testea continuamente el buffer para comprobar si el número de candidatos almacenados está entre la cantidad de servicios exigidos en la consulta, esto es, entre el mínimo exigido (`minRequired <= container.nHits()`) determinado por `storeAndForward`, y el máximo exigido (`container.nHits() == maxCandidates`), determinado por el parámetro `maxCandidates`.

Cuando el mediador encuentre las soluciones a la petición de consulta, éste devuelve la lista de los servicios candidatos (`templates`), y el número total de servicios encontrados (`nHits`, tal que este valor estará entre $1 \leqslant$ `nHits` $\leqslant$ `maxCandidates` si `storeAndForward` es `true`, o entre $0 \leqslant$ `nHits` $\leqslant$ `maxCandidates` si `storeAndForward` es `false`)

3.4.6. La clase `CandidateBuffer`

La clase `CandidateBuffer` mantiene la colección de los servicios de componente COTS del repositorio del mediador que han sido seleccionados en la operación de emparejamiento con las restricciones de una consulta dada. A esta colección de servicios de componente COTS es lo que hemos denominado antes "lista de candidatos". El mediador crea una lista por cada petición de consulta, esto es, se crea un buffer `CandidateBuffer` cada vez que se llama al método `query()`. La implementación de la clase `CandidateBuffer` es como sigue:

```
public class CandidateBuffer extends Constants {
   private String templates = S_NOT_INITIALIZED; // Funciones de buffer
   private int nHits        = NOT_INITIALIZED;   // Num. elementos incluidos
   private int longBuffer   = NOT_INITIALIZED;   // Tamaño máx. soporta el buffer
   private boolean used     = false;             // Sincronización
   CandidateBuffer( int maxValue ) { // Constructor
      longBuffer = maxValue;         // Establece el tamaño máximo del buffer
   }
   public synchronized void putTemplate( String value ) {
      if ( nHits < longBuffer ) {  // Primero se comprueba que hay sitio
         while(used) {             // luego se comprueba si está ocupado
            try { wait(); } catch (InterruptedException e) {}
         }
         used = true;              // Se marca que el buffer está en uso,
```

```
            templates += value;    // se incluye el valor en el buffer,
            nHits++;               // se incrementa el total de elementos,
         used = false;             // se quita la marca de ocupado y
      }
   }
   public String templates()  { return templates; }
   public int    nHits()      { return nHits; }
   public int    longBuffer() { return longBuffer; }
}
```

La clase está controlada por cuatro atributos y cuatro métodos (además del constructor). Entre los atributos están el buffer `templates` que almacena las plantillas candidatas (ya en formato cadena, tal y como hay que devolver en el método `query()`), el número de elementos incluidos en el buffer, la longitud máxima que éste puede tener, y un *flag* usado para la sincronización. Como vemos, en su creación el constructor de la clase establece el tamaño del buffer.

Entre los métodos de la clase están los que permiten incluir un elemento en el buffer (`putTemplate`()), obtener todos los elementos del buffer en una cadena (`templates()`), conocer el número total de elementos que hay en el buffer (`nHits()`), y por último, conocer la longitud máxima del buffer (`longBuffer()`).

3.4.7. La clase `Matchmaking`

Esta clase efectúa las actividades de emparejamiento entre una plantilla de consulta y una plantilla de componente COTS. El emparejamiento se lleva a cabo para las cuatro partes de un documento COTS: funcional, extra-funcional, empaquetamiento y marketing. La implementación de la clase de emparejamiento es como sigue (para los métodos véase el diagrama de la figura 3.8):

```
public class Matchmaking extends Thread {
   private Properties props;              // Una referencia al estado,
   private CandidateBuffer buffer;        // la lista de candidatos,
   private Node query = null;             // la plantilla de consulta,
   private Node candidate = null;         // la plantilla candidata, y
   private MatchingType matchings = null; // los cuatro tipos de emparejamiento.
   Matchmaking ( Node XMLQuery, Node theCandidate,
                 CandidateBuffer theBuffer, Properties theProper ) { ... }
   public void run () {
      enableMatchings(props);     // Prepara los 4 niveles de emparejamiento,
      startMatchings();           // los activa a la vez,
      waitMatchings();            // y espera hasta que todos ellos finalicen.
      if ( substitutability() )   // Luego comprueba el emparejamiento,
         putCandidateInBuffer();  // y si es correcto se almacena el candidato.

   }
   // resto de los métodos de la clase
   ...
}
```

Como podemos ver, el constructor de la clase acepta una serie de parámetros en su creación, y que son: por un lado, dos plantillas XML que hay que enfrentar, y que se corresponde, la primera de ellas, con la consulta del cliente en formato COTSquery, y la segunda, a una especificación de componente COTS del repositorio, en formato COTScomponent. Por otro lado, el constructor también acepta una referencia a un buffer que realiza las

funciones de lista de candidatos, y donde la clase deberá almacenar la plantilla de especificación COTScomponent, en caso de que esta haya sido detectada como candidata en el proceso de emparejamiento implícito a la clase, esto es, la plantilla cumple las condiciones de la consulta. Por último, el constructor también acepta una referencia al estado interno del mediador, donde se definen variables globales, mensajes, constantes, funciones comunes, entre otros aspectos. Esta referencia al estado es necesaria por requisitos de implementación en gran parte de las clases del paquete principal `COTStrader`.

La actividad principal de la clase se realiza en el método `run()`, donde se llevan a cabo las comprobaciones de emparejamiento entre la consulta y el candidato para los cuatro niveles de información (funcional, extra-funcional, empaquetamiento, y marketing). Estos cuatro niveles están controlados por `matchings`, una instancia de la clase `MatchingType` que declara la funcionalidad para realizar los emparejamientos para los cuatro niveles.

```
public class MatchingType {
   public MatchFunctional mFunctional;
   public MatchProperties mProperties;
   public MatchPackaging  mPackaging;
   public MatchMarketing  mMarketing;
}
```

En la actividad de emparejamiento (llevada a cabo en el método `run()`) primero se prepara la información para efectuar los emparejamientos en los cuatro niveles posibles (`enableMatchings()`). Una vez preparada la información, se da la orden para realizar los emparejamientos (`startMatchings()`) y se espera a que estos terminen (`waitMatchings()`). Tras los emparejamientos, se comprueban los resultados (`substitutability()`) para decidir si la plantilla candidata debe ser incluida o no en la lista de candidatos, controlada por el buffer (`putCandidateInBuffer()`). Para cada nivel, el emparejamiento puede devolver los valores `"True"`, `"False"` o `"Unknown"`. El valor `"True"` se da cuando el nivel de plantilla candidata casa con el de la consulta; `"False"` se da cuando no casa; y `"Unknown"` cuando no se ha podido realizar el emparejamiento, bien porque no se ha encontrado un programa para hacer emparejamiento o bien porque las notaciones en las que vienen expresadas los niveles no coinciden (por ejemplo, una definición de comportamiento en OCL y otra en JML), y no se puede garantizar si estas partes de la especificación casan o no. Por tanto, la comprobación del resultado del emparejamiento (`substitutability()`) acepta una plantilla como candidata para ser incluida en lista de candidatos, si los emparejamientos de las cuatro partes han sido `"True"` o `"Unknown"`.

Todas estas comprobaciones de emparejamiento, así como la determinación del programa de emparejamiento que el mediador debe usar, han sido tratados en los algoritmos "Selección y búsqueda" y "Selección del programa de emparejamiento" en la sección 3.3 (páginas 113 y 114, respectivamente).

3.5. Adecuación de COTStrader al modelo de mediación

Para desarrollar el servicio de mediación COSTtrader hemos tenido en cuenta las características (propiedades, véase página 103) del modelo de mediación de componentes COTS que hemos definido. Exceptuando aquellas características del modelo relacionadas con organizaciones entre servicios de mediación, como la federación y delegación de consultas entre servicios de mediación (propiedades 3 y 6), y la consideración de heurísticas y métricas, el resto de ellas se justifican como sigue:

- **Modelo de componentes heterogéneo.** El servicio de mediación utiliza las plantillas COTS como documentos de especificación de componentes comerciales para cualquier modelo de componentes (p.e., CORBA, EJB, COM, etc.).

- **Un mediador es más que un motor de búsquedas.** El tipo de consultas que permite el servicio de mediación es más amplio y estructurado que el de un motor de búsquedas convencional, pues aquel permite consultas sobre el dominio de componentes a distintos niveles: sintáctico, semántico, protocolos, extra-funcional, implantación y no técnica.

- **Composición y adaptación de servicios.** El servicio de mediación construido facilita la actuación de procesos externos sobre la lista de componentes candidatos para labores de composición y adaptación de servicios.

- **Múltiples interfaces.** Las especificaciones basadas en el modelo de documentación de componentes COTS, como las que soporta el servicio de mediación implementado, permiten la consideración de componentes software con múltiples interfaces, recogidas como colecciones de interfaces proporcionadas y requeridas.

- **Emparejamientos débiles.** Se ha dejado preparado el escenario adecuado para incorporar e integrar sin problemas programas de emparejamiento exacto y suave entre interfaces. La implementación del servicio de mediación COTStrader establece su entorno de ejecución completo a través de un archivo de configuración, desde donde se pueden indicar los nombres de estos programas de emparejamiento.

- **Extensión de los procedimientos de subtipado.** Se ha extendido el tradicional operador de reemplazabilidad entre interfaces para el caso de los componentes con múltiples interfaces, válido a cualquier nivel de especificación (p.e., sintáctico, semántico, protocolos). El desarrollo de la extensión del operador de reemplazabilidad se lleva a cabo en el siguiente capítulo.

- **Extensible y escalable.** Puesto que el modelo de información que utiliza el servicio de mediación está soportado por el lenguaje de esquemas XMLSchemas del W3C, es posible extender con relativa facilidad las plantillas XML de los componentes para que estos incluyan nuevas funcionalidades. Además, aprovechandonos de la capacidad que posee el lenguaje de esquemas del W3C de trabajar con tipado y subtipado[5] de plantillas XML y esquemas, el servicio de mediación funciona correctamente para plantillas de componente COTScomponent extendidas. Para ello, ha sido necesario desarrollar un analizador gramatical de esquemas propio[6].

- **Política de "almacenamiento y reenvio".** Esta política está soportada por la interfaz `Register` de COTStrader.

- **Modelos de almacenamiento "bajo demanda" y "por extracción".** Para soportar un esquema de almacenamiento por extracción (modelo *pull*), se ha desarrollado un componente *bot* llamado ServiceFetcher asociado al servicio COTStrader que

[5]En este caso, con el término "tipado" queremos decir cuándo una plantilla XML es una instancia válida de un esquema dado, y con "subtipado" cuándo un esquema es creado a partir de otro (u otros).

[6]Los analizadores XML existentes hasta la fecha en el momento que se desarrolló el nuestro —como por ejemplo las APIs XML4J y JAXP de IBM y Sun, respectivamente— sólo permitían realizar análisis sintácticos sobre plantillas XML, sin llegar a hacer comprobaciones semánticas entre estas plantillas XML y sus esquemas. Durante ese mismo año, el W3C recibió varias propuestas de analizadores gramaticales similares a la que nosotros habíamos desarrollado.

rastrea sistemáticamente la red buscando plantillas de componente en XML. Para verificar si una plantilla XML concreta —localizada por el componente ServiceFetcher— es una instancia de plantilla válida de la gramática COTScomponent (`http://www.cotstrader.com/COTS-XMLSchema.xsd`), se utiliza el analizador gramatical de esquemas anterioremente comentado. El esquema de almacenamiento "bajo demanda" lo soporta la interfaz `Register` de COTStrader.

3.6. Trabajos relacionados

La mayoría de los mediadores existentes ([Bearman, 1997] [OOC, 2001] [Kutvonen, 1996] [Beitz y Bearman, 1995] [Müller-Jones et al., 1995] [PrismTech, 2001] [Shmidt, 2001]) presentan similares características, ventajas y desventajas, ya que todos ellos se basan en el modelo de mediación de ODP [ISO/IEC-ITU/T, 1997] (adoptado también por OMG) y que discutimos en el *Capítulo 1* (sección 1.7.7).

Uno de los trabajos en el área de la mediación que parece diferenciarse de los demás, es WebTrader [Vasudevan y Bannon, 1999], un trabajo que proclama un modelo de mediación para el desarrollo de software basado en Internet, y que utiliza XML para describir servicios de componente. Sin embargo, el principal problema de esta aproximación es que la información que maneja acerca de los componentes está muy limitada, ya que utiliza la misma que mantiene la función de mediación de ODP (vista en la sección 1.7, página 43).

Por otro lado, también debemos destacar el proyecto Pilarcos[7] que proclama, entre otras características, la necesidad de un servicio de mediación basado en ODP/OMG para el desarrollo de soluciones intermedias (*middleware*) para la gestión automática de aplicaciones federadas. Como sucede en WebTrader, el principal inconveniente de esta propuesta es la ausencia de especificaciones de componentes adecuadas que puedan ser aplicadas para la mediación de componentes comerciales.

Otro trabajo relacionado es el servicio de directorios de servicios web de UDDI (*Universal Description, Discovery and Integration*, `http://www.uddi.com`), descrito en el *Capítulo 1* en la sección 1.8. Como vimos, las compañías que desean anunciar sus servicios web deben registrarse en un UBR (*UDDI Business Registry*), un repositorio usado por UDDI para localizar servicios web. Un repositorio UBR contiene información acerca de las compañías registradas, estructurada en páginas blancas, amarillas y verdes. Las páginas blancas contienen información acerca de la compañía que ofrece un servicio; las páginas amarillas catalogan las compañías acorde a la clase de servicios que estas proporcionan; y finalmente las páginas verdes contienen las referencias a la descripción WSDL de los servicios proporcionados por las compañías, esto es, la descripción técnica de los servicios web ofrecidos. A partir de estas descripciones técnicas, las búsquedas se centran sólo en aspectos de localización, enlace y comunicación de los servicios webs.

Precisamente, una de las limitaciones de UDDI viene del hecho de que estas descripciones técnicas radican en los WDSL, los cuales sólo permiten capturar información funcional acerca de los servicios descritos, y sólo a nivel de signaturas (no permite ni descripciones de comportamiento ni de protocolos). Además, la información extra-funcional y atributos de calidad acerca de los servicios tampoco pueden ser capturados con WSDL, dificultado esto la validación y selección de servicios web basados en requisitos extra-funcionales y restricciones arquitectónicas.

En este sentido, UDDI proporciona un potente y completo servicio de directorio, pero no puede ser considerado realmente como un mediador: un servicio de mediación es pare-

[7] Pilarcos es un proyecto de investigación del National Technology Agency TEKES de Finlandia, del Departamento Computer Science de la Universidad de Helsinki y Nokia, SysOpen y Tellabs.

cido a un servicio de directorio avanzado que permite búsquedas basadas en atributos [Kutvonen, 1995]. Además, como servicio de directorio que es, UDDI no implementa actualmente dos de las características comunes en el estándar de mediación: la federación y la propagación de consultas. Aunque la especificación de UDDI permite el concepto de "afiliación", no contempla realmente la posibilidad de federación entre repositorios UDDI, y la propagación entre diferentes UBR no está permitida tampoco por tanto.

3.7. Resumen y conclusiones del capítulo

El DSBC se dirige hacia la construcción de sistemas de software mediante la búsqueda, selección, adaptación e integración de componentes software comercial (COTS). En un mundo donde la complejidad de las aplicaciones está en continuo crecimiento, y donde la cantidad de información disponible empieza a ser considerablemente grande para que sea gestionada por intermediarios humanos, los procesos de mediación automatizados juegan un papel muy importante.

En este capítulo hemos analizado las características de un modelo de mediación para sistemas abiertos, y para el cual hemos discutido nuestra visión de servicio y tipo de servicio de componente comercial, junto con los aspectos de un repositorio de componentes COTS para la mediación, y las capacidades de exportación e importación presentes en una actividad de mediación. Para el modelo también se ha discutido cual ha sido el proceso de mediación y los algoritmos de selección y búsqueda desarrollados para ello. Finalmente hemos presentado COTStrader, una implementación del modelo de mediación para componentes COTS como una solución a la heterogeneidad, escalabilidad y evolución de los mercados COTS.

Capítulo 4

Análisis de las Configuraciones

Capítulo 4

Análisis de las Configuraciones

Contenidos

Una importante actividad del DSBC empleada en la construcción de aplicaciones de software con componentes reutilizables, es la encargada de realizar adecuadas combinaciones de componentes a partir de una colección de "componentes candidatos", con el fin de localizar aquellas cuyas especificaciones de componente cumplan, colectivamente mejor, las especificaciones de los componentes de la aplicación: requisitos impuestos a nivel de diseño en la arquitectura de software. Los componentes candidatos son una colección de especificaciones de componentes reutilizables, desarrollados por terceras partes y fuera de las estrategias de la organización. Estas especificaciones se caracterizan porque cada una de ellas, por separado, cumplen una parte de las especificaciones de los componentes de la arquitectura de la aplicación, y por tanto, podrían llegar a intervenir en una (o varias) de estas combinaciones. La colección de los componentes candidatos puede ser creada por un proceso de mediación como el que hemos visto en el capítulo anterior, buscando y seleccionando en uno o varios repositorios originales aquellas especificaciones de componente que, una a una, cumplen algunos de los requisitos de los componentes de la aplicación.

A una combinación de componentes también se le denomina configuración

Tradicionalmente, las tareas que enfrentan las dos colecciones anteriores (componentes candidatos y componentes de la aplicación) utilizan operadores de emparejamiento entre componentes del tipo "uno-a-uno" durante el cálculo de las combinaciones de componente. Esta clase de emparejamiento se debe a que normalmente los componentes software solían ofrecer un único servicio, especificado por separado en una única interfaz. Sin embargo, esta clase de emparejamiento no es adecuada para el caso de los componentes comerciales, ya que —como vimos en el *Capítulo 1*— suelen estar compuestos por diferentes servicios disponibles en múltiples interfaces, y también necesitan los servicios de otros componentes para funcionar correctamente. Según esto, los actuales procesos de DSBC deberían (re)considerar algoritmos de desarrollo que soporten también componentes software cuyas especificaciones contemplen múltiples interfaces (como los componentes comerciales), y con nuevas operaciones de emparejamiento entre las interfaces de los componentes.

En el presente capítulo se estudian los problemas que aparecen cuando se combinan especificaciones de componentes con múltiples interfaces, en la construcción de aplicaciones de software con componentes comerciales. Los problemas que aparecen son básicamente dos: las *lagunas* y *solapamientos* entre interfaces. En el capítulo también se extienden los tradicionales operadores de reemplazabilidad y compatibilidad entre componentes para el caso de múltiples interfaces, y se presenta un algoritmo que utiliza estos operadores extendidos, junto a otros definidos en el capítulo, para calcular combinaciones de componentes a partir de un conjunto de componentes candidatos y de una definición de arquitectura de la aplicación. A cada una de estas combinaciones las denominamos "configuraciones", y al algoritmo que las calcula "algoritmo de las configuraciones".

El capítulo se organiza en siete secciones, la primera de ellas con una introducción a los conceptos previos relacionados con el tema. En la sección 4.2 analizaremos la composición de componentes con múltiples interfaces, y también el operador de reemplazabilidad de componentes tradicional. En la sección 4.3 ofreceremos una extensión del operador de reemplazabilidad para componentes con múltiples interfaces. Para ello, en esa misma sección definiremos una colección de operadores que serán necesarios para el funcionamiento de un algoritmo que calcula "configuraciones" de aplicación, que presentaremos en la sección 4.4. Discutiremos también algunas consideraciones del algoritmo de configuración presentado en la sección 4.5. Finalizaremos el capítulo describiendo algunos de los trabajos relacionados con el propuesto, y con un breve resumen y conclusiones del capítulo, en las secciones 4.6 y 4.7 respectivamente.

4.1. Introducción y conceptos relacionados

La generación de configuraciones de aplicación, esto es, combinaciones de componentes software (obtenidas de un repositorio) que satisfacen los requisitos de una arquitectura de software de una aplicación objeto, está relacionada básicamente con dos aspectos del DSBC, como son la composición y la reemplazabilidad de componentes software. La composición de componentes software puede ser entendida a su vez de dos formas, según [Mili et al., 1995]:

- Dado un conjunto de componentes software y un esquema para componerlos (una arquitectura de software), la composición de componentes sirve para comprobar si tal composición es posible (*verificación*) y si se satisface el conjunto de los requisitos (*validación*). En este caso, a la composición de componentes también se le denomina "problema de la verificación y validación de composiciones" o "combinación de componentes" [Heineman y Councill, 2001] [Wallnau et al., 2002].

- Dado un conjunto de requisitos, la idea consiste en encontrar un conjunto de componentes dentro de una librería de componentes (repositorio) cuyo comportamiento combinado satisfaga estos requisitos. Para este caso, a la composición de componentes también se le denomina "problema del diseño ascendente" según [Mili et al., 1995].

Los estudios sobre la composición de componentes tienen sus orígenes en la especificación de módulos software y emparejamientos de tipos (*type matching*). En el contexto de la reutilización del software, una especificación ayuda a "comprobar" si un componente satisface las necesidades de un sistema. A este proceso de comprobación de componentes se le conoce con el nombre de "reemplazabilidad de componentes" o *substitutability*. El proceso de reemplazabilidad consiste en estudiar la compatibilidad y/o la interoperabilidad entre las especificaciones de dos componentes. En el primer caso, el problema de la compatibilidad de tipos consiste en decidir si un componente software puede ser un subtipo de otro [Nierstrasz, 1995] o si es compatible en su comportamiento respecto a otro [Zaremski y Wing, 1997]. En el segundo caso, el problema de la interoperabilidad consiste en identificar las relaciones entre dos componentes software y determinar si estos pueden interactuar [Cho y Krause, 1998].

Como se puede ver, la especificación de un componente juega un papel muy importante en las tareas de reemplazabilidad y composición. Como discutimos en el *Capítulo 1*, tradicionalmente la especificación de un componente puede quedar identificada por su interfaz. En general, una interfaz puede presentarse de muy diversas formas, dependiendo del nivel de detalle que queramos describir, o de la perspectiva desde donde la miremos. Normalmente, para estudiar la interoperabilidad entre objetos suelen distinguirse hasta tres niveles de información en la interfaz de un componente:

(a) **Nivel de signaturas**. En los actuales modelos de componentes las interfaces están formadas por el conjunto las signaturas de las operaciones proporcionadas y requeridas por un componente. Las primeras de ellas determinan las operaciones que implementa un componente, mientras que las segundas se refieren a las operaciones que precisa de otros componentes durante su ejecución. Sin embargo, esta información sintáctica de las signaturas no es del todo suficiente para construir aplicaciones de software [Zaremski y Wing, 1997] [Yellin y Strom, 1997], es necesaria también la información a nivel de protocolos y de comportamiento.

(b) **Nivel de protocolos**. Un protocolo examina el orden parcial en el que se espera que se invoquen los servicios de un objeto, y también el orden en el que éste invoca

los servicios externos de otros objetos. Para especificar este nivel existen diferentes propuestas: máquinas de estados finitas [Yellin y Strom, 1997], redes de Petri [Bastide et al., 1999], lógica temporal [Han, 1999] o pi-cálculo [Canal et al., 2003]. A los protocolos entre las interfaces de un componente también se los denominan roles.

(c) **Nivel semántico**. La información semántica trata de describir el comportamiento de las operaciones de las interfaces [Vallecillo et al., 2000]. En este caso nos encontramos con formalismos que van desde las pre/post condiciones e invariantes (p.e., utilizando Larch) [Zaremski y Wing, 1997] [Dhara y Leavens, 1996], ecuaciones algebraicas [Goguen et al., 1996] o incluso el cálculo de refinamiento [Mikhajlova, 1999].

Estos tres niveles de información también se contemplan en el modelo de documentación de componentes comerciales, propuesto en el *Capítulo 2*. A continuación estudiaremos el tratamiento que tienen estos tres niveles de información en la composición de componentes con múltiples interfaces.

4.2. Composición con múltiples interfaces

Independientemente de la notación y del nivel de interoperabilidad utilizado para describir las interfaces de un componente, cuando se construyen aplicaciones software a partir de componentes reutilizables existen unas operaciones normalmente usadas por los procesos de búsqueda de componentes que tienen especial interés en los algoritmos de composición de componentes: como los algoritmos de configuración. A continuación se describen algunos de estos operadores y más tarde se expondrá un algoritmo de configuración que hace uso de ellos, en la sección 4.4.2.

4.2.1. Operadores de composición

El primer operador de composición se denomina "reemplazabilidad", y expresa la habilidad de un componente de reemplazar a otro sin que los clientes del primero conozcan la existencia de este cambio. Este operador define un orden parcial entre componentes, y normalmente se denota como "$\leqslant$". A partir de este operador, dada una aplicación y dos componentes C y D, $D \leqslant C$ significa que podemos reemplazar C con D en la aplicación con la garantía de que la aplicación continuará funcionando sin problemas.

Operador de reemplazabilidad $\leqslant$

- A nivel de signaturas, la reemplazabilidad $D \leqslant C$ permite comprobar si todos los servicios que ofrece el componente C también los ofrece D, expresado normalmente como la contra-varianza de los tipos de los parámetros soportados y la co-varianza de los tipos de los parámetros requeridos. En sistemas de objetos a esto se le conoce con el nombre de "subtipado". En el caso de que necesitemos tener en cuenta las operaciones requeridas, deberíamos exigir además que el componente D no utilice ningún método externo no usado por C.

- A nivel de protocolos, además de comprobar si el componente C puede ser reemplazado por el componente D, necesitaremos comprobar dos cosas: (a) que todos los mensajes que acepta C (esto es, las operaciones que soporta) también los acepta D, y que los mensajes que ofrece D (esto es, las operaciones de respuesta y las requeridas) son un subconjunto de los mensajes que ofrece C; y (b) que el orden relativo de los mensajes soportados y requeridos por ambos componentes son consistentes [Canal et al., 2001] [Yellin y Strom, 1997].

- Finalmente, la reemplazabilidad semántica entre componentes es conocida en este contexto como "subtipado de comportamiento" o *behavioral subtyping*. Pierre America fue el primero en hablar sobre este concepto [America, 1991]. En sistemas de objetos está relacionado con la reemplazabilidad polimórfica de los objetos: donde se dice que el comportamiento de las instancias de una subclase es consistente con el comportamiento de las instancias de la superclase. El comportamiento de un componente determina la utilización semántica de los atributos de las operaciones y el comportamiento de sus tipos.

Basándonos en el operador de reemplazabilidad "$\leqslant$" también se puede definir una nueva relación de equivalencia entre dos interfaces, independientemente del nivel que se haya utilizado para definirlas. Dos interfaces R_1 y R_2 son *equivalentes*, denotado por el operador de equivalencia en la forma $R_1 \equiv R_2$, si se cumple que $R_1 \leqslant R_2$ y $R_2 \leqslant R_1$, es decir, si todas las operaciones de R_1 son las mismas de R_2 y viceversa.

Operador de equivalencia $\equiv$

Un tercer operador es aquel que determina cuándo dos componentes son *compatibles* para interoperar, denotado normalmente como "$\bowtie$". A nivel de signaturas esto significa que los dos componentes conocen todos los mensajes que se intercambian entre ellos. A nivel de protocolos esto significa que los dos componentes respetan sus protocolos de interacción [Yellin y Strom, 1997] [Canal et al., 2001]. A nivel semántico, esto significa que el comportamiento que tiene un componente se corresponde adecuadamente con las expectativas de comportamiento de los clientes del componente. Esto quiere decir que la compatibilidad semántica verifica si las pre-condiciones de los métodos de un componente se cumplen cuando son llamados por otro componente, y si las post-condiciones satisfacen las expectativas de estos componentes que los llaman. Esto ha sido la base de la disciplina del desarrollo del "diseño por contratos" [Meyer, 1997]

Operador de compatibilidad $\bowtie$

Todas estas operaciones son muy importantes en DSBC ya que son las que normalmente utilizan los procesos de selección para buscar componentes en repositorios y localizar aquellos que realmente pueden sustituir las especificaciones de los componentes requeridos y definidos en la arquitectura de software de una aplicación que se desea construir.

En lo que resta del capítulo, el operador "$\leqslant$" se referirá a un operador de reemplazabilidad que funciona a cualquier nivel (signaturas, protocolos o semántica) ya que todas las definiciones que presentaremos a continuación son válidas para todos ellos, y la sobrecarga de este operador simplifica bastante la notación y hace más legible el capítulo. No obstante, debemos resaltar que la complejidad del operador "$\leqslant$" puede ser muy diferente en función del nivel de que se trate: tiene una complejidad de $O(1)$ a nivel de signaturas, mientras que es exponencial en los otros dos niveles [Canal et al., 2001] [Dhara y Leavens, 1996].

4.2.2. Servicios de componente

Los componentes COTS son componentes de grano grueso que integran diferentes servicios y ofrecen múltiples interfaces. Pensemos por ejemplo en componentes como un navegador de Internet o un procesador de textos, los cuales ofrecen distintos servicios a la vez, como un editor de páginas Web, un corrector ortográfico, etc. Además de los servicios proporcionados, un componente comercial también necesita los servicios de otros componentes comerciales para su correcto funcionamiento.

Aunque la noción de componente comercial ya ha sido definida en otros capítulos por un modelo de documentación, en la exposición que nos sigue emplearemos una definición particular de componente comercial con el fin de facilitar las definiciones de los operadores y del algoritmo de configuración que vamos a introducir. Por todo esto, dado que hemos supuesto que el operador de reemplazabilidad funciona igual para cualquier nivel

de información de un componente (sintáctico, protocolos y semántico), consideraremos la especificación de un componente comercial como dos colecciones de definiciones de servicios: los que proporciona y los que requiere de otros para poder funcionar. Una definición de componente algo más formal y necesaria para nuestro estudio, podría ser la siguiente:

Definición 4.1 (Componente) *Un componente C estará determinado por dos conjuntos de interfaces $C = (\mathcal{R}, \overline{\mathcal{R}})$, el primero de ellos con las interfaces* soportadas *por el componente, y que lo denotaremos de la forma $\mathcal{R} = \{R_1, ..., R_n\}$, y el segundo de ellos con las interfaces* requeridas *por el componente, y que lo denotaremos de la forma $\overline{\mathcal{R}} = \{\overline{R}_1, ..., \overline{R}_m\}$.*

Para simplificar, escribiremos $C.\mathcal{R}$ y $C.\overline{\mathcal{R}}$ para referirnos a los dos conjuntos de interfaces de un componente dado C. Igualmente, escribiremos R_i y $\overline{R}_j$ para referirnos a interfaces particulares de las colecciones.

En el caso de la interoperabilidad a nivel de signaturas, las interfaces R_i y $\overline{R}_j$ representan interfaces estándares, como por ejemplo, las tradicionales interfaces de CORBA o COM, compuestas por un conjunto de atributos y métodos públicos. A nivel de protocolos, las interfaces R_i o $\overline{R}_j$ expresan un "rol", por ejemplo un protocolo en π-calculus como el que se mostró en la figura 1.8 del *Capítulo 1.* A nivel semántico, estas interfaces se corresponden con una descripción de interfaz con información semántica, por ejemplo una definición mediante pre/post condiciones en JavaLarch como la que se mostró en la figura 1.7 del *Capítulo 1.*

Por último, la construcción de una aplicación con componentes software normalmente se inicia con la definición de su arquitectura, a partir de la cual comienza el ciclo de vida en su desarrollo. Una vez más, como ha sucedido para el caso de la definición de componente, aunque la noción de "arquitectura de software" ya ha sido introducida en el *Capítulo 1*, en lo que sigue usaremos una definición particular para este concepto con el fin de facilitar también las definiciones de los operadores y del algoritmo, y la legibilidad en su exposición. En nuestro caso, a la arquitectura la vamos a denominar "arquitectura de aplicación" y la definimos de la siguiente forma:

Definición 4.2 (Arquitectura de aplicación) *Una arquitectura de aplicación $\mathcal{A}$ está determinada por un conjunto de especificaciones abstractas $\mathcal{A} = \{A_1, \ldots, A_n\}$, tal que A_i puede ser descrito como un componente C que respeta la definición 4.1.*

Un algoritmo de configuración calcula soluciones de implementación a partir de componentes comerciales para una aplicación deseada

En la siguiente sección introduciremos un sencillo ejemplo de aplicación de software con componentes comerciales que nos servirá luego de ayuda para ilustrar el funcionamiento de un algoritmo de configuraciones y para desarrollar el conjunto de operadores que éste utiliza.

4.2.3. Un ejemplo de aplicación con componentes

El ejemplo de aplicación que se presenta a continuación se centra en una aplicación personalizada de componentes al estilo de un Escritorio. Esta aplicación de escritorio (E) estará formada por cuatro componentes: una calculadora (CAL), un calendario (CIO), una agenda (AG) y un *Meeting Scheduler* (MS). Haciendo uso de la definición de arquitectura anterior, la aplicación de escritorio E se establece como sigue:

$$E = \{CAL,\ CIO,\ AG,\ MS\}$$

En la aplicación E el usuario puede acceder directamente a cada unos de los componentes del Escritorio, aunque internamente, unos dependen de los otros para funcionar de

forma correcta. Estas interdependencias entre los componentes vendrán impuestas por la especificación abstracta de la arquitectura de componentes de la aplicación. En la figura 4.1 se ilustra un esquema de las interdependencias para los cuatro componentes de la aplicación Escritorio[1]. En la arquitectura, los requisitos de los servicios externos de cada componente —servicios que necesita de otros componentes para funcionar— se representan con una casilla blanca, mientras que los servicios ofrecidos se expresan con una casilla negra.

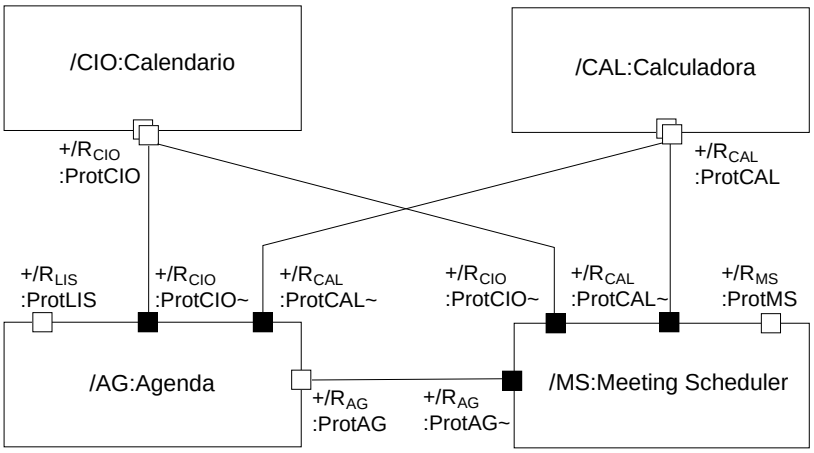

Figura 4.1: Componentes de la aplicación Escritorio

A partir de la arquitectura de componentes de la figura anterior se desprenden las restricciones de interdependencias entre los cuatro componentes de la aplicación, a saber:

(a) El Calendario (*CIO*) implementa el propio servicio de calendario R_{CIO} y no requiere ningún otro componente para funcionar.

(b) La Calculadora (*CAL*) también implementa el propio servicio de calculadora R_{CAL} y tampoco requiere otro componente.

(c) La Agenda (*AG*) implementa un servicio de agenda R_{AG} y un servicio de un listín telefónico R_{LIS}, y necesita un servicio de calculadora R_{CAL} y un servicio de calendario R_{CIO}.

(d) Por último, el componente abstracto *Meeting Scheduler MS* implementa el propio servicio R_{MS} y necesita la presencia de los servicios de una agenda R_{AG}, una calculadora R_{CAL} y un calendario R_{CIO} para poder funcionar.

E: Arquitectura (Especificaciones abstractas)
$CIO = \{R_{CIO}\}$
$CAL = \{R_{CAL}\}$
$AG = \{R_{AG}, R_{LIS}, \overline{R}_{CAL}, \overline{R}_{CIO}\}$
$MS = \{R_{MS}, \overline{R}_{AG}, \overline{R}_{CAL}, \overline{R}_{CIO}\}$

Tabla 4.1: Los componentes de la aplicación Escritorio

En la tabla 4.1 los componentes se muestran haciendo uso de la notación establecida en la definición 4.1: una colección de servicios ofrecidos y una colección de servicios requeridos

[1]Para la definición de la arquitectura de la aplicación se ha utilizado la notación UML-RT, introducida brevemente en el *Capítulo 1*.

por el componente. En dicha tabla se muestran los cuatro componentes de la arquitectura de Escritorio de la figura 4.1, y que representan en nuestro caso las especificaciones abstractas de los componentes.

4.2.4. El problema de las lagunas y los solapamientos

La definición de colecciones de componentes interconectados esbozan, a un nivel abstracto, la estructura de una aplicación software que hay que construir. Las definiciones de estos componentes abstractos son utilizadas para buscar en repositorios de componentes candidatos, posibles componentes cuyas especificaciones sean similares a las que se están buscando (y que en su conjunto identifican y dan forma a la futura aplicación). Tras una búsqueda dentro del repositorio de candidatos, el proceso extrae un componente del repositorio por cada definición de componente abstracto especificado en la estructura de la aplicación. De forma individual, cada componente extraído ofrecerá una solución para cada definición de componente abstracto, pero cuando los componentes extraídos se combinan siguiendo los criterios establecidos en la definición arquitectónica de la aplicación, pueden ofrecer ciertos problemas en su conjunto. Básicamente es posible que aparezcan dos tipos de problemas: (a) las lagunas y (b) los solapamientos.

El problema de las lagunas

Por un lado puede ocurrir que, tras efectuar la búsqueda de los componentes abstractos en el repositorio y llevar a cabo todas las combinaciones entre los posibles componentes encontrados, una solución de arquitectura no cubra todos los servicios de componente de la arquitectura; a este problema lo denominamos "problema de las *lagunas* de servicios". Para el caso de la aplicación de Escritorio, imaginemos por ejemplo una posible solución de arquitectura a partir de los cuatro componentes de la tabla 4.2.

$C_1 = \{R_{CIO}\}$	$C_2 = \{R_{CAL}\}$
$C_3 = \{R_{AG}, R_{CIO}, \overline{R}_{CAL}\}$	$C_4 = \{R_{LIS}\}$

Tabla 4.2: Cuatro componentes que intervienen en una solución de implementación de E

Si observamos con detenimiento estos componentes, podemos comprobar que una combinación de ellos realmente ofrece una solución que cubre todos los servicios de la arquitectura de la aplicación E, salvo para el servicio R_{MS}. Para este caso entonces se dice que la combinación calculada presenta una laguna para el servicio R_{MS}. No obstante, esto no implica que la solución calculada no vaya a ser considerada finalmente por el desarrollador del sistema final. De las soluciones encontradas, el desarrollador evaluará todas ellas y seleccionará aquella que mejor se ajuste a las necesidades impuestas en la arquitectura. Puede suceder incluso que una combinación que presente lagunas (como la que hemos adelantado antes) sea finalmente la solución más optima y la que tenga que utilizar el desarrollador del sistema, a costa de tener que implementar los servicios no incluidos por la solución encontrada (las lagunas).

El problema de los solapamientos

Por otro lado, también puede ocurrir que el proceso de combinación de componentes obtenga una colección de componentes donde un mismo servicio (o varios) se proporcione simultáneamente por varios componentes de la colección. A este problema lo denominamos "problema de los *solapamientos* entre servicios". Volviendo al caso de la aplicación de Escritorio, si de nuevo consideramos una combinación entre los cuatro componentes de la tabla 4.2 podremos comprobar que existe un solapamiento entre los componentes C_1 y C_3, ya que ambos ofrecen el servicio R_{CIO}. Para resolver un problema de solapamiento entre servicios es útil disponer de un operador que oculte el servicio "solapado" en cada uno

de los componentes de la solución que lo contenga, salvo en uno de ellos, que será el que finalmente aporte dicho servicio dentro de la solución.

A continuación introducimos un conjunto de operadores —entre ellos un operador de ocultación de servicios— útil para un algoritmo de combinaciones entre componentes.

4.3. Extensión del operador de reemplazabilidad

Los algoritmos o procesos de búsqueda y emparejamiento de componentes en repositorios de software utilizan una serie de operadores para llevar a cabo sus funciones, siendo uno de los más importantes el operador de reemplazabilidad. Tradicionalmente, el operador de reemplazabilidad que utilizan estos algoritmos, funciona para componentes que sólo ofrecen una única interfaz, con los métodos que estos soportan, y para los tres niveles de una especificación: nivel sintáctico, protocolos y semántico. Sin embargo, los componentes comerciales se caracterizan (entre otras cosas) por poseer dos colecciones de servicios o interfaces, los ofertados y los requeridos por el componente.

Por todo esto, ha sido necesario extender el operador de reemplazabilidad de componentes tradicional para tratar el caso de los componentes software con múltiples interfaces (como los componentes comerciales) y las dos colecciones de interfaces de un componente: las interfaces proporcionadas y las requeridas. Esto permitirá definir procesos de búsqueda y emparejamiento más realistas.

A continuación definimos un conjunto de operadores que funcionan sobre las interfaces de un componente, como son el operador de inclusión de interfaces, el operador de intersección de interfaces, el operador de composición de componentes, el operador de ocultación de interfaces y el operador extendido de reemplazabilidad. La definición de estos operadores se basa en la notación tradicional de conjuntos.

4.3.1. Operador de inclusión de conjuntos de interfaces ($\mathcal{R}_1 \subseteq \mathcal{R}_2$)

Uno de los operadores más usuales utilizado por el algoritmo de configuraciones es aquel que determina si un conjunto de interfaces está incluido en otro conjunto (es un subconjunto del primero). Este operador queda definido de la siguiente forma:

Definición 4.3 (Inclusión de conjuntos de interfaces) *Sean $\mathcal{R}_1$ y $\mathcal{R}_2$ dos conjuntos de interfaces definidos de la forma $\mathcal{R}_1 = \{R_1^1, ..., R_1^s\}$ y $\mathcal{R}_2 = \{R_2^1, ..., R_2^t\}$. Se dice que $\mathcal{R}_1$ está incluido en $\mathcal{R}_2$, y lo denotaremos de la forma $\mathcal{R}_1 \subseteq \mathcal{R}_2$, si para todo $R_1^i \in \mathcal{R}_1$ existe una interfaz $R_2^j \in \mathcal{R}_2$ con $R_2^j \leqslant R_1^i$.*

Téngase en cuenta en esta definición que el operador "$\leqslant$" tiene que ver con el operador de reemplazabilidad para interfaces simples, sin importar el nivel de interoperabilidad al que se refiera (signaturas, protocolos o semántica).

4.3.2. Operador de intersección de conjuntos de interfaces ($\mathcal{R}_1 \cap \mathcal{R}_2$)

Otro operador necesario para el algoritmo de configuraciones es aquel usado para conocer la colección de interfaces comunes a dos conjuntos de interfaces, esto es, un operador de intersección. De forma parecida a como lo hemos hecho antes, podemos definir un operador de intersección en la forma natural como sigue:

Definición 4.4 (Intersección de conjuntos de interfaces) *Sean $\mathcal{R}_1$ y $\mathcal{R}_2$ dos conjuntos de interfaces definidos de la forma $\mathcal{R}_1 = \{R_1^1, ..., R_1^s\}$ y $\mathcal{R}_2 = \{R_2^1, ..., R_2^t\}$. Se define la*

intersección de interfaces $\mathcal{R} = \mathcal{R}_1 \cap \mathcal{R}_2$ *como el conjunto* $\mathcal{R} = \{R^1, \cdots, R^u\}$ *tal que para todo* $R^i \in \mathcal{R}$ *existen dos interfaces* $R_1^j \in \mathcal{R}_1$ *y* $R_2^k \in \mathcal{R}_2$ *tales que* $R^i \equiv R_1^j$ *y* $R^i \equiv R_2^k$.

El operador $\equiv$ es el operador de equivalencia entre interfaces introducido en la página 138. Como hemos adelantado, estos operadores funcionan para los tres niveles de información de una interfaz: nivel sintáctico, nivel de protocolos y nivel semántico.

4.3.3. Operador de ocultación de servicios ($C - \{\mathcal{R}\}$)

Uno de los problemas que se puede presentar a la hora de combinar componentes candidatos para la búsqueda de una solución de arquitectura, es el problema de los solapamientos. Este problema se presenta cuando un mismo servicio (o más) es ofrecido por dos (o más) componentes candidatos dentro de una solución encontrada. Un operador de ocultación permite (como su nombre indica) esconder uno o más servicios de componente de la colección de servicios ofrecidos por el componente en cuestión. Por tanto, podemos definir un operador de ocultación de servicios de la siguiente forma:

Definición 4.5 (Ocultación) *Sea* C_1 *un componente de la forma* $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$ *y sea* $\mathcal{R}$ *un conjunto de interfaces. El operador de* ocultación *"−" se define como* $C_1 - \{\mathcal{R}\} = (\mathcal{R}_1 - \mathcal{R}, \overline{\mathcal{R}}_1)$*, permitiendo esto ocultar en el componente* C_1 *todas las interfaces ofrecidas en el conjunto* $\mathcal{R}$.

La implementación del operador de ocultación se puede hacer desarrollando un "envolvente" software[2] que defina un nuevo componente a partir del original, y donde sus interfaces no incluyan una definición para los servicios ocultados. A nivel de implantación (*deployment*) e integración, esto implica que la implementación completa del componente comercial tenga que estar presente en el entorno de instalación del componente, debido a la naturaleza binaria de un componente software [Szyperski, 1998]. No obstante, aunque las implementaciones tengan que estar en el entorno de instalación, realmente permanecerían inhabilitadas en tiempo de ejecución al no estar contempladas en las interfaces de los envolventes.

4.3.4. Operador de composición de componentes ($C_1 \mid C_2$)

Por otro lado, los componentes deben componerse para construir otros nuevos, o nuevas aplicaciones. Para ello, definimos un operador de composición de componentes de la siguiente forma:

Definición 4.6 (Composición de componentes) *Sean* $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$ *y* $C_2 = (\mathcal{R}_2, \overline{\mathcal{R}}_2)$ *dos componentes. Se define la composición de los componentes* C_1 *y* C_2*, que denotaremos por* $C_1 \mid C_2$*, como un nuevo componente* $C_3 = (\mathcal{R}_3, \overline{\mathcal{R}}_3)$*, tal que:*

$$\mathcal{R}_3 = \begin{cases} \mathcal{R}_1 \cup \mathcal{R}_2 & \textit{sii } \mathcal{R}_1 \cap \mathcal{R}_2 = \varnothing \\ \textit{indefinido} & \textit{sii } \mathcal{R}_1 \cap \mathcal{R}_2 \neq \varnothing \end{cases}$$

$$\overline{\mathcal{R}}_3 = \begin{cases} \overline{\mathcal{R}}_1 \cup \overline{\mathcal{R}}_2 - \{\mathcal{R}_1 \cup \mathcal{R}_2\} & \textit{sii } \mathcal{R}_1 \cap \mathcal{R}_2 = \varnothing \\ \textit{indefinido} & \textit{sii } \mathcal{R}_1 \cap \mathcal{R}_2 \neq \varnothing \end{cases}$$

Téngase en cuenta que la composición de componentes es tanto conmutativa como asociativa. La hemos definido como una operación "parcial" para evitar los conflictos que pudieran aparecer en una aplicación cuando dos o más de sus componentes ofrecieran el mismo servicio (esto es, un problema de solapamientos entre servicios).

[2]Código adicional, conocido como *wrapper*, que extiende el funcionamiento de un componente software.

4.3.5. Operador de reemplazabilidad entre componentes ($C_1 \leqslant C_2$)

En este punto ya estamos en condiciones de poder extender el tradicional operador de reemplazabilidad entre interfaces para tratar con componentes que ofrezcan (y requieran) varias interfaces. Este nuevo operador de reemplazabilidad entre componentes queda definido de la siguiente forma:

Definición 4.7 (Reemplazabilidad) *Sean* $C_1 = (\mathcal{R}_1, \overline{\mathcal{R}}_1)$ *y* $C_2 = (\mathcal{R}_2, \overline{\mathcal{R}}_2)$ *dos componentes. Diremos que* C_1 *puede ser* reeemplazado (*o* sustituido) *por* C_2*, y lo denotaremos de la forma* $C_2 \leqslant C_1$*, si y solo si* $(C_1.\mathcal{R}_1 \subseteq C_2.\mathcal{R}_2) \wedge (C_2.\overline{\mathcal{R}}_2 \subseteq C_1.\overline{\mathcal{R}}_1)$.

Esto significa que el componente C_2 ofrece todos los servicios (o más) ofrecidos por el componente C_1, y además requiere los mismos (o menos) servicios de otros componentes.

4.4. Estrategias de composición con múltiples interfaces

El objetivo de una estrategia de composición es conseguir combinaciones de componentes a partir de un repositorio que, de forma conjunta, resuelvan las necesidades de los servicios ofertados y requeridos de los componentes definidos por una aplicación $\mathcal{A}$. A cada una de estas combinaciones que ofrecen una solución de arquitectura, la vamos a denominar "configuración". Una definición algo más formal de una configuración es la siguiente:

Definición 4.8 (Configuración) *Dada una arquitectura de aplicación* $\mathcal{A}$ *y un repositorio de componentes* β*, definimos una* configuración S *como un conjunto de componentes software de* β *que cumple las siguientes dos condiciones: (a) el conjunto de los servicios ofrecidos por los componentes de* S *debe coincidir con el conjunto de los servicios ofrecidos por los componentes de* $\mathcal{A}$ *(esto es, no se presenta* lagunas *de servicios), y (b) dos componentes cualesquiera de* S *no ofrecen una misma interfaz.*

Veamos a continuación más detalles acerca de cómo se lleva a cabo la generación de configuraciones. Para producir el conjunto de todas las configuraciones válidas para una aplicación dada $\mathcal{A}$ se han definido las tres siguientes fases:

(a) selección de componentes candidatos,

(b) generación de configuraciones, y

(c) cierre de las mismas.

Describamos cada una de ellas con más detalle.

4.4.1. Selección de componentes candidatos

En primer lugar, el algoritmo de configuraciones genera las combinaciones de componentes a partir de un repositorio de componentes cuya restricción es que éste almacene componentes que ofrezcan, cada uno de ellos, al menos un servicio ofrecido por los componentes abstractos definidos por la arquitectura de la aplicación. Por este motivo, antes de llevar a cabo el algoritmo de configuraciones es necesario seleccionar del repositorio original β aquel conjunto de componentes $\{B_1, \cdots, B_m\}$ que puedan potencialmente formar parte de la aplicación $\mathcal{A}$ al ofrecer alguno de sus servicios. Este proceso lo puede llevar a cabo un servicio de mediación, como el que hemos presentado en el capítulo anterior. Al conjunto

que obtiene este proceso lo hemos denominado conjunto de componentes candidatos, y lo hemos denotado como $C_\beta(\mathcal{A})$.

Para su obtención, primero se considera la arquitectura de la aplicación $\mathcal{A}$ como un nuevo componente, obtenido por la composición de los componentes abstractos que la definen, y considerando solamente los servicios ofrecidos, sin tener en cuenta los que se requieren. Luego se recorre el repositorio de componentes original β una sola vez, decidiendo para cada uno de sus elementos si se incluyen en la colección $C_\beta(A)$ o no. Si $m = card(\beta)$ es el número de componentes en el repositorio original, y L es la complejidad del operador de reemplazabilidad (constante para las comprobaciones de signaturas, y exponencial para las comprobaciones de los protocolos y semántica), entonces la complejidad del proceso de selección sería $O(mL)$.

Una definición algo más formal de un conjunto de componentes candidatos es como sigue:

Definición 4.9 (Componentes candidatos) *Considérese $\mathcal{A}$ como un componente $\mathcal{A} = A_1 \mid A_2 \mid \cdots \mid A_n$, donde $\mathcal{A}.\mathcal{R}$ y $\mathcal{A}.\overline{\mathcal{R}}$ representan los conjuntos de interfaces ofrecidos y los requeridos por la aplicación. Se define la* colección de componentes candidatos *de la aplicación $\mathcal{A}$ respecto al repositorio β como $C_\beta(\mathcal{A}) = \{B \in \beta \mid A.\mathcal{R} \cap \mathcal{B}.\mathcal{R} \neq \varnothing\}$, esto es, los componentes del repositorio β que ofrecen algún servicio que también ofrece $\mathcal{A}$.*

A modo de ejemplo, volvamos al ejemplo de la aplicación de Escritorio para describir este proceso. Como se ha indicado, lo primero de todo es considerar la arquitectura de la aplicación como un único componente con servicios ofrecidos y requeridos, y para el caso del Escritorio esto significa:

$$E = CIO \mid CAL \mid AG \mid MS$$

La composición de estos cuatro componentes de Escritorio nos lleva a decir que la aplicación de Escritorio es un componente E la siguiente forma:

$$E.\mathcal{R} = \{R_{CIO}, R_{CAL}, R_{AG}, R_{LIS}, R_{MS}\},\ E.\overline{\mathcal{R}} = \{\}$$

A partir de estos servicios del componente Escritorio, recorremos los componentes que hay en el repositorio original, seleccionando aquellos que ofrecen al menos uno de los servicios que soporta E. Supongamos que un posible resultado tras llevar a cabo el enfrentamiento entre E y β, es la colección de componentes candidatos que se muestra en la tabla 4.3. Para este caso se ha supuesto que el proceso de selección ha encontrado solamente seis componentes candidatos. Como una particularidad, se puede observar que el componente C_6 requiere un servicio externo definido por la interfaz R_P. En el caso de que este componente sea finalmente incluido en una configuración, sería necesario cerrarla para que la aplicación pudiera funcionar correctamente con dicho componente. El último paso lleva a cabo esta tarea, efectuando una operación de cierre transitivo entre conjuntos, en este el conjunto de las configuraciones frente al conjunto del repositorio original β.

4.4.2. Generación de configuraciones

La segunda fase trata de construir el conjunto $\mathcal{S}$ de todas las posibles configuraciones con los componentes candidatos. Como hemos adelantado, las "configuraciones" son subconjuntos de combinaciones de componentes candidatos que satisfacen las especificaciones de la aplicación completa y que no presentan solapamientos entre sus servicios (para componerlos sin problemas). Un posible algoritmo que calcula las configuraciones consiste en

$C_\beta(E)$: Componentes candidatos (Especificaciones concretas)
$C_1 = \{R_{CIO}\}$
$C_2 = \{R_{CAL}\}$
$C_3 = \{R_{AG}, R_{CIO}, \overline{R}_{CAL}\}$
$C_4 = \{R_{LIS}\}$
$C_5 = \{R_{MS}, R_{AG}, \overline{R}_{CIO}\}$
$C_6 = \{R_{CAL}, R_{LIS}, \overline{R}_P\}$

Tabla 4.3: Una lista de componentes candidatos encontrados para la aplicación Escritorio

ir generando combinaciones entre los servicios de los componentes candidatos y escoger aquellas soluciones $Sol = \{S_1, \cdots, S_l\}$ con $\mathcal{A}.\mathcal{R} \subseteq Sol.\mathcal{R}$.

En la tabla 4.4 se muestra un algoritmo de vuelta atrás que genera el conjunto $\mathcal{S}$ de todas las configuraciones posibles a partir de la aplicación $\mathcal{A}$ —considerada como un componente— y de los componentes que hay en la colección de componentes candidatos $C_\beta(A) = \{C_1, \cdots, C_k\}$. La llamada inicial del algoritmo es:

- $Sol = \varnothing$
- $\mathcal{S} = \varnothing$
- `configs`$(1, Sol, \mathcal{S})$

```
1  function configs(i, Sol, S)
2    // 1 ≤ i ≤ size(C_β(A)) recorre el repositorio
3    // Sol es la configuración que se está construyendo
4    // S es el conjunto de todas las configuraciones válidas A hasta el momento
5    if i ≤ size(C_β(A)) then
6      for j := 1 to size(C_i.R) do // para todo servicio en C_i
7        // se intenta incluir el servicio C_i.R_j en Sol
8        if {C_i.R_j} ∩ Sol.R = ∅ then // C_i.R_j ∉ Sol.R?
9          Sol := Sol ∪ {C_i.R_j};
10         if A.R ⊆ Sol.R then // ¿Es Sol una configuración?
11           S := S ∪ {Sol}; // si es así, entonces se incluye en S
12         else // pero si aun hay servicios pendientes ...
13           configs(i, Sol, S); // entonces se busca en C_i ...
14         endif
15         Sol := Sol − {C_i.R_j};
16       endif
17     endfor
18     configs(i + 1, Sol, S); // Siguiente en C_β(A).
19   endif
20 endfunction
```

Tabla 4.4: Algoritmo que genera configuraciones válidas para una arquitectura software

	C_1	C_2	C_3	C_4	C_5	C_6	Configuraciones	R	C
–	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	–	–	NO: falta R_{MS} (laguna)	No	No
1	R_{CIO}	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	–	C_1, C_2, C_3–$\{R_{CIO}\}, C_4, C_5$–$\{R_{AG}\}$	Si	Si
2	R_{CIO}	R_{CAL}	R_{AG}	–	R_{MS}	R_{LIS}	C_1, C_2, C_3–$\{R_{CIO}\}, C_5$–$\{R_{AG}\}, C_6$–$\{R_{CAL}\}$	Si	No
3	R_{CIO}	R_{CAL}	–	R_{LIS}	R_{MS}, R_{AG}	–	C_1, C_2, C_4, C_5	No	Si
4	R_{CIO}	R_{CAL}	–	–	R_{MS}, R_{AG}	R_{LIS}	C_1, C_2, C_5, C_6–$\{R_{CAL}\}$	Si	No
5	R_{CIO}	–	R_{AG}	R_{LIS}	R_{MS}	R_{CAL}	C_1, C_3–$\{R_{CIO}\}, C_4, C_5$–$\{R_{AG}\}, C_6$–$\{R_{LIS}\}$	Si	No
6	R_{CIO}	–	R_{AG}	–	R_{MS}	R_{CAL}, R_{LIS}	C_1, C_3–$\{R_{CIO}\}, C_5$–$\{R_{AG}\}, C_6$	No	No
–	R_{CIO}	–	R_{AG}	–	–	R_{LIS}	NO: faltan R_{CAL} y R_{MS} (lagunas)	No	No
7	R_{CIO}	–	–	R_{LIS}	R_{MS}, R_{AG}	R_{CAL}	C_1, C_4, C_5, C_6–$\{R_{LIS}\}$	No	No
8	R_{CIO}	–	–	–	R_{MS}, R_{AG}	R_{CAL}, R_{LIS}	C_1, C_5, C_6	No	No
–	R_{CIO}	R_{CAL}	–	–	–	–	NO: faltan R_{AG}, R_{LIS} y R_{MS} (lagunas)	No	No
9	–	R_{CAL}	R_{AG}, R_{CIO}	R_{LIS}	R_{MS}	-	C_2, C_3, C_4, C_5–$\{R_{AG}\}$	No	Si
10	–	R_{CAL}	R_{AG}, R_{CIO}	–	R_{MS}	R_{LIS}	C_2, C_3, C_5–$\{R_{AG}\}, C_6$–$\{R_{CAL}\}$	No	No
–	–	R_{CAL}	R_{AG}	R_{LIS}	R_{MS}	–	NO: falta R_{CIO} (laguna)	No	No
11	–	R_{CAL}	R_{CIO}	R_{LIS}	R_{MS}, R_{AG}	-	C_2, C_3–$\{R_{AG}\}, C_4, C_5$	No	Si
12	–	R_{CAL}	R_{CIO}	–	R_{MS}, R_{AG}	R_{LIS}	C_2, C_3–$\{R_{AG}\}, C_5, C_6$–$\{R_{CAL}\}$	No	No
–	–	R_{CAL}	–	–	R_{MS}	–	NO: faltan R_{CIO}, R_{AG} y R_{LIS} (lagunas)	No	No
13	–	–	R_{AG}, R_{CIO}	R_{LIS}	R_{MS}	R_{CAL}	C_3, C_4, C_5–$\{R_{AG}\}, C_6$–$\{R_{LIS}\}$	No	No
14	–	–	R_{AG}, R_{CIO}	–	R_{MS}	R_{CAL}, R_{LIS}	C_3, C_5–$\{R_{AG}\}, C_6$	No	No
15	–	–	R_{CIO}	R_{LIS}	R_{MS}, R_{AG}	R_{CAL}	C_3–$\{R_{AG}\}, C_4, C_5, C_6$–$\{R_{LIS}\}$	No	No
16	–	–	R_{CIO}	–	R_{MS}, R_{AG}	R_{CAL}, R_{LIS}	C_3–$\{R_{AG}\}, C_5, C_6$	No	No
–	–	–	–	–	–	R_{CAL}, R_{LIS}	NO: faltan R_{CIO}, R_{AG} y R_{MS} (lagunas)	No	No

Tabla 4.5: Algunos resultados del algoritmo `configs()` para la aplicación Escritorio

Como se puede ver, el algoritmo explora todas las posibilidades y va construyendo una lista de configuraciones válidas (línea 11). Cada configuración individual (línea 9) se genera al recorrer todos los servicios de los componentes de la colección candidata, incorporando en la solución parcial aquellos servicios del componente que no están en la misma —en la forma $C_i.R_j$— y descartando aquellos que ya están (líneas 8 y 10). Al finalizar, $\mathcal{S}$ contiene todas las configuraciones válidas. Luego, para cada una de ellas, es necesario ocultar los servicios de componente que ya están presentes en la misma, haciendo uso para ello del operador de ocultación. Una implementación de este algoritmo es COTSconfig disponible en `http://www.ual.es/selectingCOTS/home.html` y presentado en la 28 Conferencia de Euromicro [Iribarne et al., 2002b].

Por la forma en la que se ha presentado el proceso, no se producen solapamientos entre servicios ni se obtienen configuraciones "parciales", sólo las que ofrecen los servicios de A. Respecto al orden de complejidad del algoritmo, expresado en términos de operaciones de reemplazabilidad, su tiempo de ejecución en el peor de los casos, es de orden $O(2^n)$, siendo n el número total de servicios del conjunto de componentes candidatos $C_\beta(A)$.

A modo de ejemplo en la tabla 4.5 se muestran algunos de los resultados producidos por el algoritmo `configs()` para la aplicación de Escritorio E. En este caso el algoritmo genera $2^9 = 512$ combinaciones, ya que el total de servicios proporcionados por todos los componentes candidatos fue 9. De estas 512 posibles combinaciones, sólo 16 de ellas son válidas, mostradas en la tabla junto con algunas otras configuraciones que son descartadas.

4.4.3. Cierre de las configuraciones

Una vez obtenidas las configuraciones, ahora es preciso cerrarlas para formar una aplicación completa, y evitar así la existencia de lagunas en el caso de no estar presentes los servicios requeridos por algún componente de la configuración concreta. Para ello, como se ha dicho

antes, es suficiente utilizar cualquiera de los algoritmos existentes para el cierre transitivo de un conjunto (en este caso una configuración) con respecto a otro más grande, en este caso con respecto a los componentes del repositorio β.

Definición 4.10 (Aplicación cerrada) *Sean C_1, C_2, ..., C_n un conjunto de componentes y $A = C_1 \mid C_2 \mid ... \mid C_n$ uno nuevo obtenido por la composición de ellos. Se dice que el componente A es* cerrado *si se cumple $\bigcup C_i.\overline{\mathcal{R}} \subseteq \bigcup C_i.\mathcal{R}$.*

Siguiendo con nuestro ejemplo de Escritorio, en la tabla 4.5 las columnas 2 a la 7 muestran los servicios proporcionados por cada componente en cada configuración. La columna 8 describe la configuración en términos de sus componentes constituyentes (ocultando los servicios apropiados). Y en las columnas 9 y 10 se muestra con una "R" y una "C" si la configuración respeta la estructura de la aplicación o si la configuración es cerrada, respectivamente. Por ejemplo, la configuración 1 contiene todos los componentes candidatos salvo C_6, y cada componente proporciona sólo un servicio. Esta configuración es cerrada y respeta la estructura de la aplicación.

De las 16 configuraciones encontradas, cuatro de ellas son cerradas, y otras cuatro respetan la estructura. Ahora es decisión del diseñador del sistema seleccionar aquella configuración (de entre la lista de todas las configuraciones válidas) que mejor se ajuste a las necesidades impuestas en la arquitectura, o incluso puede darse el caso que el diseñador tenga que refinar los requisitos de la arquitectura original tras evaluar los resultados obtenidos, y volver a recalcular la lista de configuraciones hasta lograr encontrar alguna que se adecue correctamente.

Es necesario tener en cuenta que los procesos aquí descritos han sido definidos para el caso de aplicaciones completas. Sin embargo, esto podría ser utilizado para sólo algunas partes de una aplicación, y no para toda ella en su conjunto. En este sentido, podríamos permitir que el diseñador del sistema decidiera qué partes del conjunto de la aplicación quisiera implementar con componentes COTS desde repositorios y qué partes no, aplicando por tanto el proceso descrito a las partes seleccionadas.

4.5. Consideraciones sobre las configuraciones

Una vez establecida una estrategia para producir una lista de configuraciones a partir de la especificación abstracta de una aplicación y de un repositorio de componentes, en esta sección discutimos algunas consideraciones sobre las configuraciones.

4.5.1. Métricas y heurísticas

En general, el proceso que aquí se propone ofrece al diseñador de aplicaciones un conjunto de configuraciones posibles desde donde poder seleccionar aquella que se ajuste mejor a sus necesidades. Sin embargo, y puesto que el algoritmo que las construye es exponencial, sería muy interesante disponer de métricas capaces de asignar "pesos" a cada una de las configuraciones. Por ejemplo, se podrían considerar factores como el precio de cada componente, su complejidad (por ejemplo, en función del número de interfaces que soportan), su fabricante, etc. Esto permitiría, entre otras cosas, presentar las soluciones encontradas ordenadas según un criterio definido por el usuario, y si se dispone de pesos, también se podría modificar el algoritmo para extenderlo a uno de ramificación y poda, para eliminar aquellas opciones que no conduzcan a configuraciones deseables, y rebajar también la complejidad del algoritmo de configuración.

4.5.2. Cumplimiento con la arquitectura de software

Las configuraciones se han construido a partir de los servicios que la aplicación ofrece, sin tenerse en cuenta para nada su estructura interna. Cabría preguntarse también si es posible generar configuraciones que respeten esa estructura, en el sentido que se mantenga la "división" en los componentes que se especifican en la definición $A = \{A_1, \cdots, A_n\}$. Para ello, se pueden generar las configuraciones siguiendo el proceso anterior, y descartar aquellas que no sean "compatibles" con la arquitectura definida para A. El concepto de compatibilidad en este contexto se define como:

Definición 4.11 *Sea* $A = \{A_1, \cdots, A_n\}$ *una aplicación y* $S = \{S_1, \cdots, S_m\}$ *una configuración para A, esto es, un conjunto de componentes del repositorio cuya composición ofrece los mismos servicios que A, y que no presenta solapamientos entre ellos. Se dice que S* respeta la estructura *de A si* $\forall\, i \in \{1..m\}, \forall\, j \in \{1..n\} \bullet S_i.\mathcal{R} \cap A_j.\mathcal{R} \neq \varnothing \Rightarrow (S_i \leqslant A_j) \vee (A_j \leqslant S_i)$

Esta definición obliga a que los componentes de la configuración estén "contenidos" o "contengan" a las especificaciones abstractas de los componentes de la aplicación, respetando las "fronteras" definidas en la arquitectura de la aplicación.

4.6. Trabajos relacionados

Trabajos relacionados con el análisis de las configuraciones podemos encontrar varios. En [Rolland, 1999] por ejemplo se propone una técnica para razonar semánticamente durante los procesos de selección y ensamblaje sobre aquellos componentes que cumplen los requisitos del sistema. Estos requisitos, son recogidos mediante unos diagramas de transiciones llamados mapas que se basan en cuatro modelos: el modelo "As-Is", el modelo "To-Be", el modelo COTS, y el modelo de emparejamiento integrado. Sus inconvenientes: (a) es una propuesta muy abstracta; (b) no hace una propuesta de especificación COTS; y (c) no describe detalles de cómo se llevan a cabo los emparejamientos sintácticos y semánticos para comprobar la reemplazabilidad, interoperabilidad e incorporabilidad de componentes, entre otros.

Por su parte, Jun Han [Han, 1999] hace una propuesta bastante interesante basada en la especificación de las signaturas de la interfaz de un componente. Además el autor impone restricciones semánticas a estas interfaces y establece configuraciones para el caso de los protocolos basados en roles para capturar las relaciones entre distintos componentes.

Otro trabajo es el de [Zaremski y Wing, 1997], donde los autores hacen distinción entre información tipo (signaturas) e información de comportamiento (especificaciones), y define unas funciones para el emparejamiento (*matching*) de las especificaciones de dos componentes a partir de las signaturas y las *pre/post* condiciones de comportamiento. Algunas de estas funciones de emparejamiento son:

- $match_{E-pre/post}(S, Q) = (Q_{pre} \Leftrightarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post})$
- $match_{plug-in}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$
- $match_{plug-in-post}(S, Q) = (S_{post} \Rightarrow Q_{post})$
- $match_{weak-post}(S, Q) = S_{pre} \Rightarrow (S_{post} \Rightarrow Q_{post})$

El autor define en total 12 tipos de funciones de emparejamiento (*match*), reglas de composición, propiedades y una teoría formal de especificación alrededor de este marco de

trabajo. Desarrolla un lenguaje llamado Larch/ML para la especificación de componentes y realiza las implementaciones en LP (*Larch Prover*), un probador de teoremas basado en un subconjunto de la lógica de primer orden. Estas funciones pueden ser útiles en:

(a) *recuperación* de componentes desde una librería o repositorio [Mili et al., 1995],

(b) *clasificación* de componentes en una librería o repositorio de componentes,

(c) *reemplazabilidad* de componentes [Nierstrasz, 1995] en arquitecturas COTS, e

(d) *interoperabilidad* de componentes [Canal et al., 2003].

Las principales desventajas de la propuesta de Zaremski son la dificultad de hacer especificaciones de componentes con Larch/ML; además, los componentes tienen que estar implementados en un lenguaje propietario como LP, no permitiéndose otros modelos de componentes como CORBA, EJB o COM; por último, las funciones de emparejamiento propuestas no son adecuadas para componentes más complejos que los que utiliza su autor para validar su propuesta.

4.7. Resumen y conclusiones del capítulo

Para el desarrollo de aplicaciones basadas en componentes COTS son necesarias unas tareas que combinen las descripciones "concretas" de los componentes comerciales con el objetivo de encontrar soluciones, totales o parciales, a las necesidades impuestas en la arquitectura de software del sistema que hay que construir. Estas tareas utilizan operadores de emparejamiento que no son adecuadas para el caso de los componentes comerciales, ya que estos operadores funcionan sobre interfaces simples, y no para componentes con múltiples interfaces (como es el caso de los componentes comerciales). Además, para esta clase de componentes también hay que tener presentes las interfaces que estos requieren de otros, y no sólo las que ofrecen. Sin embargo, para un proceso que lleve a cabo combinaciones entre componentes con múltiples interfaces, esto podría acarrear ciertos problemas.

En este capítulo hemos extendido los tradicionales operadores de reemplazabilidad y equivalencia de componentes para el caso de que los componentes ofrezcan y requieran *múltiples interfaces*, complementando los estudios tradicionales, en los que los componentes sólo ofrecían una interfaz y no se tenían en cuenta los servicios que necesitan de otros componentes para poder funcionar. En el capítulo también se han estudiado los problemas que aparecen al combinar componentes con múltiples interfaces, como los *solapamientos* o las *lagunas* de las interfaces. Asimismo, se ha presentado un algoritmo para la "generación de configuraciones" que calcula combinaciones de componentes comerciales para desarrollar una aplicación, tomando como referente los componentes abstractos definidos en su arquitectura de software.

Capítulo 5

Integración con Metodologías de Desarrollo de Software basadas en Componentes COTS

Capítulo 5

Integración con Metodologías de Desarrollo de Software basadas en Componentes COTS

Contenidos

El desarrollo de software basado en componentes (DSBC) está cambiando algunos de los actuales métodos y herramientas de la Ingeniería del Software. Los tradicionales métodos de desarrollo de software ascendente y descendente no son directamente aplicables al campo del DSBC ya que el diseñador del sistema también debe de tener en cuenta las especificaciones de los componentes comerciales disponibles en repositorios de software, y que son considerados en todas las fases del proceso de desarrollo [Mili et al., 1995] [Robertson y Robertson, 1999] [Cheesman y Daniels, 2001]. Además, los constructores, arquitectos y diseñadores de sistemas deben aceptar la fuerte dependencia de tres aspectos muy importantes en el DSBC: (a) los requisitos, (b) la arquitectura del sistema, y (c) los productos COTS [Garlan et al., 1994] [Ncube y Maiden, 2000]. Las actuales soluciones tratan estos aspectos durante la construcción de aplicaciones de software. Estas soluciones normalmente se basan en metodologías en espiral donde el nivel de detalle de los requisitos, de las especificaciones de arquitectura y de los diseños del sistema, va aumentando progresiva y sistemáticamente a lo largo del ciclo de vida de desarrollo del sistema. Aunque una metodología en espiral parece ser la mas adecuada cuando se trata de construir aplicaciones de software con componentes COTS, en realidad presenta algunas limitaciones para esta clase de componentes.

El modelo Twin Peaks [Nuseibeh, 2001] *es un ejemplo de metodología en espiral*

En primer lugar, un proceso de construcción con componentes comerciales requiere una rápida realización de prototipos de arquitectura de sistema que contemple los continuos cambios (refinamientos) de los requisitos de los componentes y de sus interconexiones que supone la utilización de un modelo de espiral. Sin embargo, aunque son numerosas y conocidas hoy día las propuestas de lenguajes para la definición de arquitecturas de software, como lo son por ejemplo Acme [Garlan et al., 2000], Rapide [Luckham et al., 1995], Wright [Allen y Garlan, 1997] o LEDA [Canal et al., 2001], entre otras[1], son pocas las propuestas que permiten trabajar con componentes COTS y elaborar prototipos de arquitecturas de software de forma gráfica, rápida y fácil, como se requiere en un modelo en espiral con componentes comerciales.

En segundo lugar, la identificación y recopilación de los requisitos de un sistema en una arquitectura de software es una tarea ardua, y más complicada aún para el caso de los componentes comerciales por su naturaleza de caja negra. La falta de una información de especificación para esta clase de componentes dificulta que sus características puedan ser analizadas rápida y fácilmente con el fin identificar algunos requisitos de mercado (como información semántica, sintáctica o de protocolos, entre otra).

Por último, las actuales soluciones de desarrollo basadas en metodologías en espiral carecen hoy día de unos procesos automáticos que medien entre las arquitecturas de software y los componentes comerciales, esto es, unos procesos que actúen ante la demanda de unas necesidades arquitectónicas de sistema y la oferta de unos componentes de mercado residentes en repositorios que podrían satisfacer tales necesidades arquitectónicas.

Las propuestas hasta el momento realizadas en los capítulos precedentes, como el modelo de documentación de componentes COTS, el modelo de mediación de componentes COTS, y el análisis de las configuraciones, pueden integrarse con éxito en las metodologías de DSBC actuales. En particular, en este capítulo se ve cómo puede integrarse con una metodología en espiral como es la de [Nuseibeh, 2001].

El capítulo se compone de siete secciones. En la sección 5.1 ofreceremos una introducción y algunos conceptos relacionados con el ámbito del capítulo. En la sección 5.2, expondremos

[1] En el *Capítulo 1*, sección 1.6.3, ya tratamos algunas de estas propuestas de Lenguajes para la Definición de Arquitecturas, conocidos como LDAs.

un ejemplo de una aplicación de software basada en componentes COTS que usaremos para ilustrar el funcionamiento del método de elaboración en espiral que presentaremos inmediatamente después en la sección 5.3. Seguidamente, en la sección 5.4, analizaremos las etapas que componen el método de desarrollo propuesto y luego discutiremos algunas de sus consideraciones en la sección 5.5. Finalizando el capítulo, en la sección 5.6, veremos algunos trabajos relacionados con el método de desarrollo aquí presentado, y concluiremos el capítulo con un breve resumen y algunas conclusiones, en la sección 5.7.

5.1. Introducción y conceptos relacionados

El ámbito del presente capítulo se centra en la construcción de aplicaciones de software con componentes COTS a partir del paradigma de desarrollo de software basado en componentes (DSBC). Como vimos en el *Capítulo 1* (donde tratamos más extensamente el paradigma DSBC), la construcción de aplicaciones con componentes software está inevitablemente ligada al concepto de requisito de software, y su concepción en el área de la ingeniería del software, junto con las características peculiares de los componentes comerciales, están cambiando la forma de construir las aplicaciones de software de hoy día.

Analicemos a continuación dos de los factores que están favoreciendo dicho cambio y que están estrechamente relacionados con el propósito del presente capítulo: el modelo de desarrollo en espiral, y el problema de la conexión diseño-implementación.

5.1.1. El modelo de desarrollo en espiral

Inicialmente, el concepto de requisito de software fue concebido como algo obligatorio y necesario, y su significado fue la base del proceso de desarrollo en cascada —conocido también como modelo *waterfall*— muy utilizado por las organizaciones para la construcción de sistemas durante muchos años. La idea del modelo se sustenta en la identificación y recopilación de una colección de los requisitos que debe tener el sistema de software que se pretende construir. Sin embargo, este paradigma centrado en requisitos no tardó en provocar ciertos problemas en las organizaciones.

Aunque los ingenieros utilizaban tareas y técnicas especiales de especificación de requisitos con el fin de identificar y recopilar el mayor número de requisitos posible, estas especificaciones eran realmente incapaces de cubrir todos los requisitos del sistema. Esto se debía en parte a que los usuarios del sistema obviaban muchos detalles técnicos que los ingenieros luego no llegaban a detectar con las técnicas de requisitos. Además, muchos de los requisitos iniciales del sistema sufrían cambios durante el proceso de desarrollo, y otros nuevos iban apareciendo. Por tanto, era extremadamente complicado conocer el conjunto completo de los requisitos del sistema durante las etapas iniciales del sistema. La inevitable conclusión fue que, particularmente para el desarrollo de grandes sistemas, las especificaciones de los requisitos contenían muchos detalles, a menudo con inconsistencias y con información contradictoria, lo cual demandaba algún tipo de gestión de requisitos a lo largo del ciclo de vida del desarrollo del sistema.

Una solución a este problema fue el modelo en espiral de Barry Boehm [Boehm, 1988], quien afirmaba que el conjunto de los requisitos del sistema no podía ser determinado completamente al inicio de su construcción, y todo intento para establecerlos probablemente fallaría. Para hacer frente a este dilema, el proceso de desarrollo debería hacer uso de frecuentes prototipos de arquitectura del sistema, requerir la presencia permanente de los usuarios finales del sistema durante todo el desarrollo, y realizar progresivos refinamientos de los prototipos y objetivos del sistema, entre otros aspectos.

El modelo del ciclo de vida en espiral resuelve algunas de los inconvenientes de un modelo en cascada, proporcionando un proceso de desarrollo incremental, en donde los desarrolladores continuamente evalúan los sucesivos prototipos de la arquitectura y el riesgo que conlleva cambiar el proyecto para controlar los requisitos inestables.

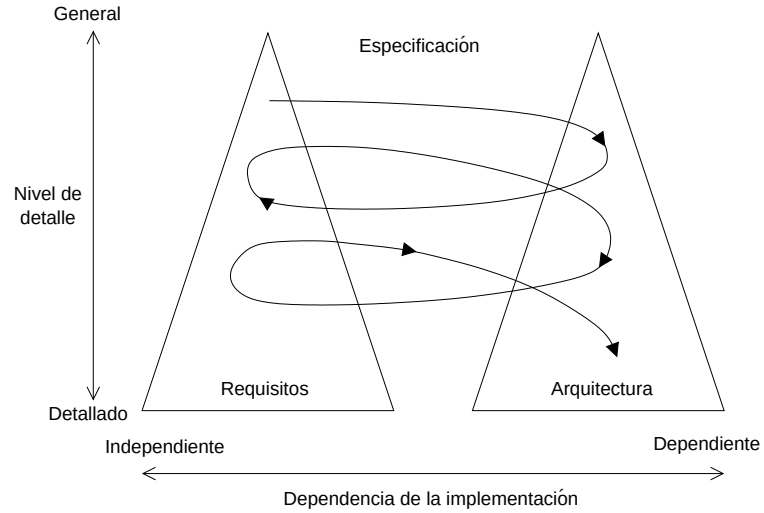

Figura 5.1: Un modelo en espiral ([Nuseibeh, 2001], página 116)

Como se puede comprobar en la figura 5.1, inicialmente las especificaciones de los requisitos se recogen a nivel general y se van detallando sistemática y progresivamente a medida que se va construyendo el sistema. Durante todo este proceso, simultáneamente se realizan rápidos prototipados de la arquitectura del sistema que son continuamente revisados y que pueden alterar el estado de la colección de los requisitos, modificando o desapareciendo requisitos y/o dependencias entre requisitos existentes, o incluyendo algunos nuevos. El modelo en espiral permite que tanto los ingenieros de requisitos como los arquitectos del sistema puedan trabajar concurrentemente de forma repetida para describir las partes que desean producir. Este proceso permite que los desarrolladores del sistema puedan comprender mejor los problemas arquitectónicos, y desarrollar y adaptar las arquitecturas basadas en requisitos.

Por otro lado, aunque el modelo en espiral parece ser el más idóneo para procesos DSBC, especialmente para el caso de los componentes comerciales, no han proliferado métodos, técnicas y herramientas estándares en DSBC que faciliten su aplicación de forma rápida y óptima. Además, muchas cuestiones están aún sin contestar, como por ejemplo, (a) cuáles son los estilos arquitectónicos más estables ante la presencia de cambios en los requisitos, y cómo seleccionarlos, o (b) qué clases de requisitos son más estables que otros, y cómo identificarlos, o (c) qué clases de cambios suelen generarse en un sistema, y cómo debemos gestionar los requisitos y las arquitecturas para minimizar el impacto de estos cambios [Nuseibeh, 2001]. Lo que es cierto es que las respuestas a estas y otras cuestiones favorecerán la emergencia de un contexto de desarrollo de software que incluya (1) líneas de productos y familias de productos que requieren arquitecturas estables que toleren cambios en los requisitos, (2) sistemas COTS que requieren la identificación de requisitos existentes que casen con los requisitos de la arquitectura, y (3) sistemas heredados (*legacy systems*) que pueden incorporar en las especificaciones de requisitos restricciones acerca de unos sistemas ya existentes.

5.1.2. El problema de la conexión diseño-implementación

Lo que sí parece estar claro es que un proceso de construcción de aplicaciones de software basado en componentes COTS sigue un paradigma DSBC, donde se parte de un prototipado rápido de la arquitectura del sistema, hecho a partir de unas especificaciones de componente y arquitectónicas que recogen los requisitos de los componentes abstractos y de sus interconexiones.

Una situación ideal del paradigma DSBC llevado a su máxima expresión nos lleva a suponer que estas especificaciones de componentes abstractos son luego utilizados por procesos automáticos que buscan y seleccionan especificaciones de unos componentes comerciales (componentes de mercado) que residen en repositorios públicos perfectamente conocidos por estos procesos automáticos. Como resultado, se obtienen colecciones de especificaciones de componentes concretos que casan y cumplen con aquellos requisitos de componente y requisitos arquitectónicos que dan solución parcial o totalmente a una configuración de componentes de la aplicación deseada. En otros casos, puede que estas colecciones de especificaciones de componentes concretos no ofrezcan soluciones de mercado para uno o varios componentes abstractos definidos en la arquitectura de software. En este caso, el desarrollador del sistema puede tomar las siguientes decisiones:

(1) evaluar de nuevo los requisitos de los componentes abstractos en función de la información de los requisitos de los componentes concretos obtenidos, re-estructurando o detallando cada vez más los requisitos de los componentes abstractos y volviendo a repetir el proceso hasta llegar a obtener la configuración que mejor se ajuste a las necesidades impuestas; o

(2) tras sucesivas iteraciones del paso anterior, si no se llega a la solución óptima, el desarrollador deberá quedarse con la mejor solución obtenida hasta el momento, y luego desarrollar una implementación para aquellos componentes abstractos que no hayan sido cubiertos por una solución de componente comercial concreto, respetando su especificación y sus dependencias con el resto de los componentes de la arquitectura.

Sin embargo, la realidad es bien diferente al proceso anteriormente comentado. Actualmente existe una desconexión entre las arquitecturas y los repositorios: entre las especificaciones de componentes abstractos definidos a nivel de diseño en la arquitectura de software de la aplicación, y las especificaciones de componentes que presentan una implementación concreta y que residen en repositorios de software. A este problema se le conoce como el "problema de la conexión diseño-implementación" o el problema del *Gap Analysis*. Este problema, aún sin resolver completamente en el campo de la ingeniería del software, se debe principalmente a la falta de tres factores importantes que deben estar presentes en el paradigma DSBC con componentes comerciales (DSBC-COTS), como en la situación ideal anteriormente comentada: (a) un modelo de especificación de componentes comerciales, (b) una notación adecuada para el prototipado rápido y visual de arquitecturas de software, y (c) un servicio para la mediación de componentes comerciales. Los dos últimos factores (arquitectura y mediación) están directamente relacionados con el primero (requisitos).

Como hemos adelantado al comienzo de este capítulo, en siguientes apartados discutiremos un método de desarrollo basado en un modelo en espiral como una propuesta de solución al problema de la conexión diseño-implementación. Pero antes de abordar este método, en la siguiente sección introduciremos las bases de un ejemplo para elaborar una aplicación de software con componentes COTS y que iremos desarrollando a lo largo del capítulo. Este ejemplo nos servirá para ilustrar el comportamiento de las etapas del método que proponemos.

5.2. Un ejemplo de aplicación con componentes COTS

En la presente sección vamos a desarrollar un ejemplo para construir una aplicación de software con componentes COTS. Para ello, seguidamente vamos a elaborar un marco de trabajo donde expondremos el ámbito de la aplicación del ejemplo seleccionado dentro del campo de la informática, describiremos también la colección de los componentes comerciales que conforman la aplicación del ejemplo, y usaremos un diagrama de secuencias para explicar las relaciones entre estos componentes. Veamos pues el planteamiento del ejemplo seleccionado.

5.2.1. Descripción de una aplicación GTS

En muchos sistemas informáticos es frecuente encontrarnos con componentes software que llevan a cabo tareas de conversión entre formatos de datos. En la mayoría de los casos, esta conversión es necesaria por las propias necesidades del sistema, ya que puede requerir el uso de componentes que trabajen con diversas representaciones internas (diferentes formatos). Este es el caso por ejemplo de los Sistemas de Información Geográficas (SIG), donde se trabaja con una gran variedad de información, como información alfanumérica, información de imágenes de satélite, imágenes aéreas (tomadas por vuelo aéreo, en avión), imágenes vectoriales, entre otros. Debido a esta gran variedad y cantidad de información, es muy frecuente encontrarnos con SIG desarrollados para que puedan funcionar de forma distribuida. La toma de datos en muchos casos se realiza de forma descentralizada, y almacenando esta información en bases de datos locales. Esta información distribuida luego puede ser accedida y consultada por componentes software que permiten y facilitan la interoperabilidad entre los diferentes formatos de bases de datos.

Por centrarnos solo en un pequeño caso, un buen ejemplo de este comportamiento se da en los componentes software de conversiones entre formatos de imágenes vectoriales. En algunos módulos de SIG es muy frecuente efectuar conversiones entre formatos de imágenes para realizar cálculos de distancias entre elementos espaciales, áreas de polígonos, altura de elementos, entre otros casos. Por ejemplo, imaginemos que una determinada zona geográfica de la Tierra ha sido digitalizada a partir de un plano cartográfico de la zona. Esta operación se puede llevar a cabo mediante un Plotter y algún programa que permita la georreferenciación de planos cartográficos y su digitalización, por ejemplo AutoCad. Normalmente, este proceso de digitalización genera un archivo con extensión DWG, con las coordenadas espaciales (coordenadas vectoriales) introducidas desde el Plotter por el operador. Esta información digital luego puede ser almacenada en una base de datos DWG, para su posterior incorporación como módulo del SIG.

Los SIG suelen requerir la presencia de múltiples componentes distribuidos

Supongamos ahora, que necesitemos llevar a cabo un pequeño cálculo a partir de la imagen vectorial de la zona anteriormente digitalizada. A modo de ejemplo, imaginemos que deseamos calcular a qué distancia se encuentran las autovías más cercanas a partir de un núcleo de población con el propósito de llevar a cabo futuros proyectos de construcción urbana. Para esto es necesario tener la imagen en un formato determinado: formato DXF.

Por tanto, si consideramos un SIG como un gran sistema, distribuido y donde sus módulos requieren el intercambio de información con diversos formatos, entonces podríamos pensar en tener un componente software, ubicado en alguna parte del sistema, que responde ante las peticiones de otros componentes software que únicamente desean convertir formatos entre imágenes, como por ejemplo conversiones del tipo DWG/DXF, GIF/TIF, JPG/GIF, entre otras conversiones posibles.

El planteamiento que se hace a continuación nos sirve para ilustrar el comportamiento del método de DSBC-COTS que proponemos en la siguiente sección. El ejemplo por

tanto, consiste en un servicio de conversión de imágenes espaciales, también conocidas con el nombre de imágenes geográficas (GTS, *Geographic Translator Service*) que funciona remotamente en un sitio de la red.

En la figura 5.2 podemos ver el entorno donde se ubica nuestra aplicación GTS. Para este caso, CE (componente emisor) es un componente que trabaja con un determinado tipo de imagen y que tiene que pasarle una imagen al componente CR (componente receptor) con un formato establecido. Para ello, CE delega el proceso de conversión al componente GTS, pasándole la información necesaria para que éste pueda llevar a cabo con éxito la tarea de conversión de imágenes.

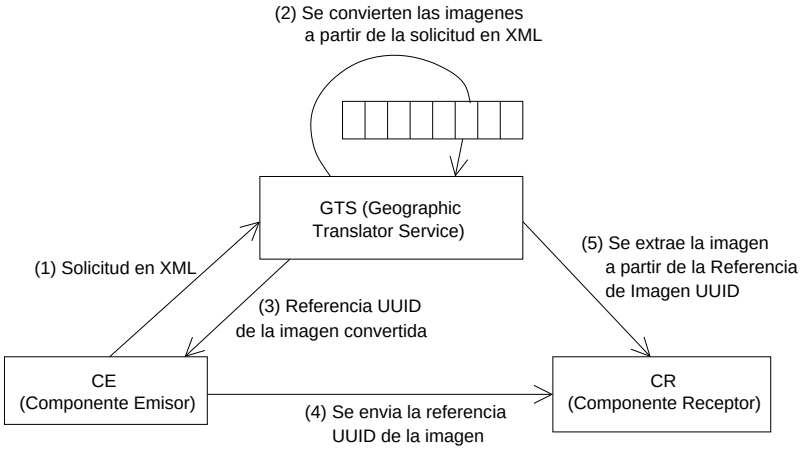

Figura 5.2: Entorno de una aplicación GTS

Para descargar el tráfico de la información por la red, suponemos que el componente CE no envía la imagen al GTS, sino que es el GTS el que extrae el archivo desde el sitio CE mediante un modelo por extracción (modelo *pull*) ofreciendo su ubicación previamente en la primera conexión CE/GTS (paso 1 en la figura 5.2). Además, para optimizar el proceso de descarga por el GTS, el archivo de imagen se deja comprimido en algún formato típico, como “zip”, “tar”, “gzip”, u otros formatos.

El componente GTS lleva a cabo la conversión entre formatos de imagen, estos también indicados por el componente CE en la llamada al GTS (paso 2). Una particularidad de esto es que, por razones de optimización de tráfico, nuestro GTS no devolverá al componente CE el archivo de imagen convertido. El GTS convertirá el archivo de imagen objeto, le asignará un identificador único UUID (similar a un *Universal Unique IDentifier* del modelo COM) y luego lo almacenará en un buffer, desde donde más tarde el archivo podrá ser descargado por otro componente haciendo uso de su UUID asignado. Por tanto, ante una llamada de conversión del CE, el GTS sólo devolverá el UUID asignado al archivo de imagen convertido (paso 3). Cada celda del buffer estará compuesta por la pareja `<UUID,href>`, siendo `href` una referencia física a la ruta donde se encuentra realmente almacenado el archivo de imagen convertido y comprimido. Supondremos además que la representación interna del buffer se realizará en XML.

Seguidamente el componente CE llama al componente CR, ofreciéndole el UUID de la imagen con la que éste último debe operar (paso 4). A modo de ejemplo, este CR podría ser un componente que calcula distancias, áreas o alturas entre elementos espaciales, tal y como hemos adelantado antes cuando explicamos el entorno de los SIG.

Finalmente, el componente CR accede al GTS con el UUID de la imagen, y éste le responde con la dirección de enlace desde donde hacer una descarga del archivo de imagen convertido y en un formato comprimido (paso 5).

Supondremos también que el componente GTS acepta los mensajes de petición de conversión de imagen, empaquetados en formato XML, siendo estos mensajes de la forma que se muestra a continuación en la figura 5.3.

```
<image url="http://www.unlugar.com/">
  <name input="RiverImage" output="RiverImage2" />
  <format input="DWG" output="DXF" />
  <compression input=".zip" output=".tar" />
</image>
```

Figura 5.3: Un ejemplo de mensaje en XML que acepta el componente GTS

Concretamente, para este ejemplo, el componente GTS acepta la imagen `RiverImage` en formato DWG ubicada en el sitio `http://www.unlugar.com/` y comprimida en formato ZIP. Tras la conversión, el GTS generará un archivo en formato DXF con el nombre `RiverImage2`, y comprimido en formato TAR. Como hemos indicado anteriormente, el componente GTS almacenará la imagen convertida `RiverImage2.tar` en una celda del buffer interno asociado al GTS, desde donde luego otro componente podrá descargar la imagen con el UUID que el componente GTS generó al almacenarla en el buffer.

5.2.2. Componentes de la aplicación GTS

Para no complicar en exceso el ejemplo que estamos planteando, sólo nos centraremos en el comportamiento interno del componente GTS sin tener en cuenta los componentes emisores y receptores. Consideremos por tanto el componente GTS como una sub-aplicación constituida por la composición de una colección de otros componentes comerciales. En la figura 5.1 se muestra una descripción de las interfaces de estos componentes. Estas definiciones nos ayudarán a comprender mejor el comportamiento interno del GTS, y serán útiles para construir más tarde la arquitectura de software de la aplicación GTS.

```
//Componente Translator
interface Translator {
  boolean translate
   (in string request,
    out long uuid);
  boolean get
   (in long uuid,
    out string url);
};
```

```
//Componente XMLBuffer
interface XMLBuffer {
  boolean pull
   (in string location,
    out string uuid);
  boolean get
   (in string uuid,
    out string href);
};
```

```
//Componente ImageTranslator
interface imageTranslator {
  boolean translate
   (in string source,
    in string iformat,
    in string target,
    in string oformat);
};
```

```
//Componente FileCompressor
interface fileCompressor {
   boolean zip
    (in string source,
     in string target,
     in string format);
   boolean unzip
    (in string source,
     in string format);
};
```

```
//Componente XDR
interface XDR {
   boolean unpack
    (in string template,
     out string iname,
     out string oname,
     out string oformat,
     out string icompress,
     out string ocompress,
     out string url);
};
```

Tabla 5.1: Las definiciones IDL de los componentes del GTS

Como podemos comprobar, la aplicación GTS estará compuesta por cinco componentes comerciales. Entre estos, la definición abstracta del componente `Translator` acepta dos métodos: `translate` y `get`. El método `translate` acepta una petición XML realizada por un componente de tipo emisor para convertir un archivo de imagen geográfica. Por otro lado, el método `get` acepta peticiones `UUID` por parte de componentes de tipo receptor para extraer desde el buffer XML archivos de imágenes convertidas.

El subsistema de componentes `GTS` también utiliza un componente conversor de archivos convencional, llamado `fileCompressor` en la figura, y que trabaja de una forma similar a como lo hacen las tradicionales herramientas de compresión/descompresión de archivos tipo "zip/unzip" (por ejemplo, similares a los programas *Winzip* o *pkunzip*). Como podemos comprobar en la figura, la interfaz de este componente tiene dos métodos, necesarios para la compresión y descompresión de archivos. Por ejemplo, un mensaje de tipo `zip` podría ser de la siguiente forma `zip("RiverImage","RiverImageCompressed","ZIP")`. En este ejemplo hemos introducido directamente los valores de los parámetros dentro del mensaje.

Por otro lado, el subsistema de componentes `GTS` utiliza un componente conversor de imágenes de satélite, llamado `ImageTranslator`, similar a los programas comerciales *Geographic Translator* o *MapObjects* usados para este fin. La interfaz de este componente puede convertir formatos tales como `DWG`, `DXF`, `DNF`, entre otros formatos parecidos. Por ejemplo, un mensaje `translate` podría ser `translate("RiverImage","DXF","RiverImage","DWG")`.

Para desempaquetar las peticiones especiales en XML realizadas externamente por los componentes emisores y receptores, la aplicación GTS considera la utilización de un nuevo componente, llamado `XDR` en la figura. En este caso, una petición XML puede ser establecida en el parámetro `template` del método `unpack`. Tras una llamada, este método devuelve siete parámetros, obtenidos todos ellos a partir del mensaje de petición en XML. Luego, los valores de estos parámetros son utilizados por el resto de los componentes del GTS.

Los archivos de imágenes geográficas son mantenidos en un buffer XML y controlados por un componente llamado `XMLBuffer` en la figura. La interfaz de este componente tiene dos métodos: `pull` y `get`. El método `pull` descarga un archivo de imagen de satélite comprimida desde una localización referenciada. Una vez obtenida, la imagen es descomprimida, convertida y de nuevo comprimida por el componente `Translator`, llamando a los métodos correspondientes de las interfaces de los componentes `FileCompressor` y `ImageTranslator`. Finalmente, el componente almacena la imagen espacial en un buffer, asignándole un identificador (UUID) y que es devuelto por el método `pull`. Por otro lado, usando el código de la referencia de imagen (UUID), el método `get` devuelve una localizacion en forma `href` donde se almacena el archivo de imagen convertido. Esta localización será la que luego utilizará el componente receptor para descargar la imagen de satélite en el formato prestablecido.

5.2.3. Funcionamiento de la aplicación

Una vez definidos los componentes de la aplicación GTS por separado, veamos a continuación cómo sería el funcionamiento conjunto de todos ellos, y en definitiva, el comportamiento del componente GTS. Para describir dicho funcionamiento haremos uso del diagrama de secuencias mostrado en la figura 5.4, el cual ilustra el orden de las llamadas entre los métodos de las interfaces de los cinco componentes vistos anteriormente. En la parte superior del diagrama de secuencias se muestran cinco instancias de componente, una para cada componente de la aplicación GTS (definidos en la figura 5.1). Así pues, `/trans:Translator` representa a un componente denominado `trans` que es una instancia del tipo de interfaz de componente `Translator`, y de forma similar para el resto de los cuatro componentes de la figura.

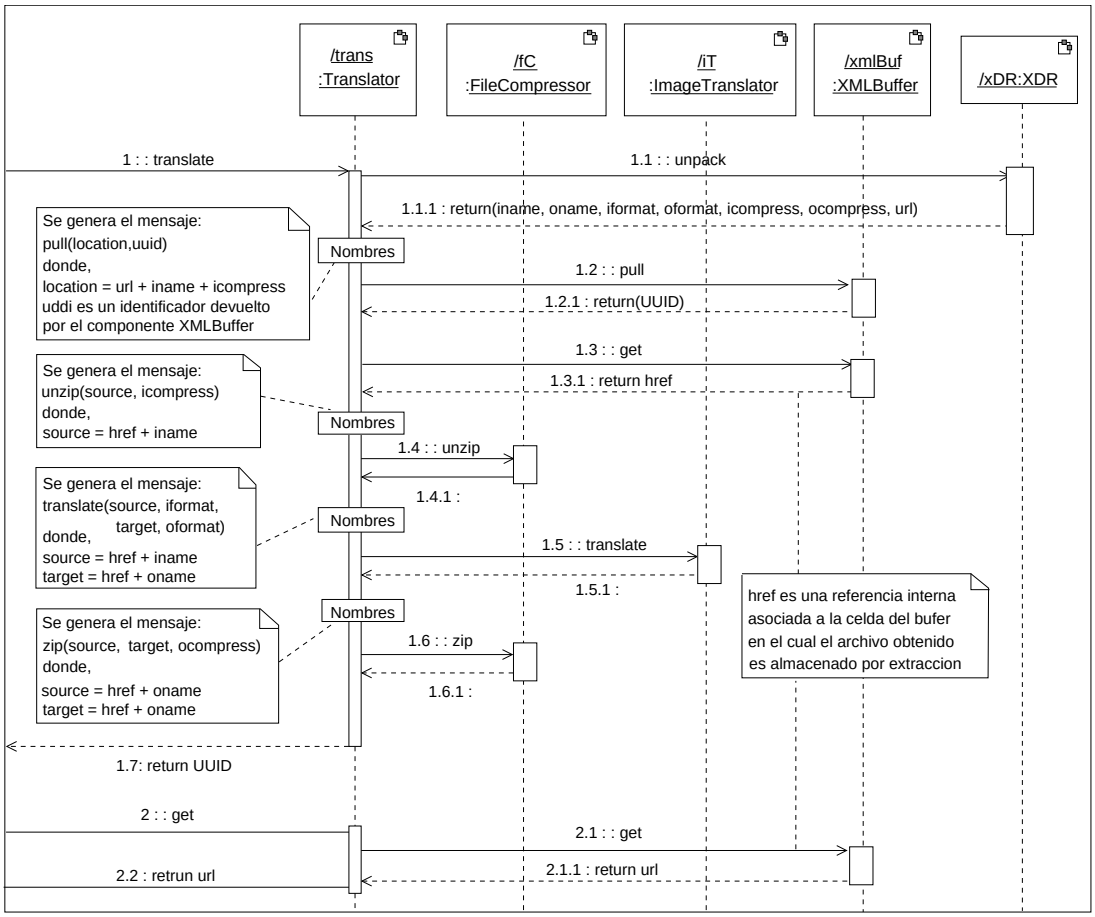

Figura 5.4: Secuencia de métodos de los componentes del GTS

La secuencia del funcionamiento de la aplicación GTS se inicia con una llamada externa de un componente emisor al componente GTS para convertir una imagen de satélite. Esta llamada queda recogida como un mensaje de petición de conversión sobre el método `translate` del tipo de componente `Translator` (secuencia 1 en la figura 5.4). Para facilitar la exposición, supongamos que el mensaje en cuestión se corresponde con el mostrado en la figura 5.3. Una vez recibida la petición de conversión en XML, el tipo de componente `Translator` empieza a requerir los servicios de los demás componentes de la colección.

En primer lugar, el componente accede a un servicio de desempaquetamiento, ofrecido por el tipo de componente `XDR`. Esta operación se lleva a cabo mediante una llamada al método `unpack` (paso 1.1), al cual se le pasa el mensaje que hay que desempaquetar . Como resultado del desempaquetamiento (paso 1.1.1), se extraen las siete variables necesarias para que el componente de conversión pueda realizar su tarea. La variable `iname` es el nombre que tiene el archivo de imagen origen (p.e., `RiverImage`), `oname` es el nombre que debe tener el archivo de imagen de salida una vez ya convertido (p.e., `RiverImage2`), `iformat` es el tipo de formato que tiene el archivo de imagen de entrada (p.e., `DWG`), `oformat` es el tipo de formato que debe generarse para el archivo de imagen de salida (p.e., `DXF`), `icompress` es el tipo de compresión del archivo de imagen de entrada (p.e., `.zip`), `ocompress` es el tipo de compresión requerido por el componente receptor para el archivo de imagen de salida convertido (p.e., `.tar`), y por último `url` es la dirección web donde reside el archivo de imagen de entrada y desde donde se debe extraer (p.e., `http://www.unlugar.com`).

Para nuestro ejemplo se está considerando el mensaje XML de la figura 5.3

Seguidamente, el componente `Translator` deberá extraer el archivo de imagen des-

de su ubicación. El servicio de descarga de un archivo es ofrecido en el tipo de componente `XMLBuffer`, por el método `pull`. Según esto, `Translator` hace una llamada al servicio (paso 1.1) pasándole el mensaje creado a partir de `url`, `iname` y `icompress` (p.e., `http://www.unlugar.com/RiverImage.zip`). Como resultado, el componente `XMLBuffer` descarga el archivo de imagen del sitio indicado y lo almacena en un buffer XML interno, al cual le asigna un código único `UUID` usado en la devolución (paso 1.2.1).

Usando el código `UUID` obtenido en el paso anterior, ahora el componente `Translator` solicita al componente `XMLBuffer` que le pase el archivo de imagen con el código UUID (paso 1.3). Como resultado (paso 1.3.1), devolverá una referencia a la localización (`href`) donde se encuentra físicamente almacenado el archivo de imagen requerido (p.e., `/usr/local/xml/`).

Antes de hacer la conversión de la imagen, es necesario descomprimir el archivo de imagen fuente, ya que originariamente, dicho archivo se descarga comprimido en un formato (p.e., `RiverImage.zip`). Para hacer esto, ahora se requiere de un servicio de descompresión de archivos, ofrecido en este caso por el tipo de componente `FileCompressor` en el método `unzip`. Llegado a este punto, el componente `Translator` primero genera la ruta completa —donde reside el archivo de imagen fuente comprimido— a partir de las variables `href`, `iname` y `icompress`, y a continuación efectúa la llamada al servicio de descompresión (paso 1.4). Como resultado (paso 1.4.1), se extrae el archivo de imagen de tipo `iformat` (p.e., `DXF`). La descompresión se realiza en la misma localización que el archivo comprimido (p.e., `/usr/local/xml/`), y se devuelve `true` o `false` para indicar el éxito o el fracaso de la operación de descompresión.

Una vez que se ha obtenido la imagen de satélite fuente (p.e., `RiverImage.dwg`) ahora se trata de convertirla al formato de imagen especificado en el mensaje original (p.e., `DXF`). Para ello, esta vez se hace uso de un servicio de conversión de imágenes de satélite ofrecido por el método `translate` del tipo de componente `ImageTranslator` (paso 1.5). Esta vez el componente `Translator` genera el mensaje de llamada a partir de las variables `href` (p.e., `/usr/local/xml`), `iname` (p.e., `RiverImage`), `iformat` (p.e., `DWG`), `oname` (p.e., `RiverImage2`) y `oformat` (p.e., `DXF`). Como resultado (paso 1.5.1), el conversor genera la nueva imagen en la misma localización que el archivo origen (p.e., `/usr/local/xml/`). Igual que antes, se devuelve `true` o `false` para el resultado de la operación de conversión.

Finalmente, antes de concluir la funcionalidad de la aplicación GTS es necesario comprimir la imagen convertida en el formato de compresión establecido en el mensaje original XML (p.e., `.tar`, según el mensaje de la figura 5.3). De nuevo aquí se requiere un servicio de compresión/descompresión de archivos. Como se ha visto antes, este servicio es ofrecido en el tipo de componente `FileCompressor`, pero esta vez por el método `zip`. La compresión de un archivo requiere el nombre del archivo que hay que comprimir (p.e., `RiverImage2.dxf`, el nombre del archivo comprimido (p.e., `RiverImage2`), y el tipo de formato de compresión que hay que realizar (p.e., `.tar`). Como resultado (paso 1.5.1), el archivo de imagen convertida y comprimida (p.e., `RiverImage2.tar`) permanece en la localización indicada, la cual está asociada al código único `UUID` de un buffer XML generado anteriormente por el componente `XMLBuffer` (en el paso 1.1). También aquí se devuelve `true` o `false` para indicar el éxito o el fracaso de la operación de compresión.

La aplicación GTS devuelve el código `UUID` como resultado final de la operación de conversión de una imagen de satélite (paso 1.7). Este código es el que utiliza un componente receptor para descargar la imagen de satélite convertida y comprimida desde el GTS. Este servicio de descarga lo ofrece el tipo de componente `XMLBuffer`, el cual prepara el archivo de imagen para la descarga y genera una dirección de enlace (paso 2.1.1) para que el componente receptor (paso 2.2) efectúe dicha descarga siguiendo un modelo por extracción.

5.3. Integración con una metodología en espiral

Hasta el momento ya hemos presentado las bases de un ejemplo de aplicación de software con componentes comerciales (la aplicación GTS) y que usaremos como ejemplo para ilustrar el funcionamiento de un método de desarrollo de software para la construcción de aplicaciones con componentes COTS, que introducimos a continuación en la presente sección. Para la exposición del método, en primer lugar haremos un planteamiento general del problema que intentamos resolver con dicho método; en segundo lugar estudiaremos por separado cuales son los actores o entidades que lo constituyen; y finalmente, expondremos su comportamiento global, indicando las secuencias de interconexión entre las entidades anteriormente identificadas. Más tarde en la sección 5.4 estudiaremos con más detalle la secuencia de estas interconexiones (en forma de etapas) para llevar a cabo la construcción de la aplicación GTS del ejemplo desarrollado en la sección 5.2.

5.3.1. Planteamiento

La "ingeniería de componentes software" es una sub-disciplina de la ingeniería del software que sigue principalmente dos prácticas de desarrollo para la elaboración de partes software: (a) por un lado está la práctica de desarrollo de software "orientado" a componentes software (DSOC) —también conocida como desarrollo de software "centrado" en componentes (DSCC)— la cual utiliza métodos, lenguajes y herramientas para el desarrollo exclusivo de componentes de software; y (b) por otro lado está el desarrollo de software basado en componentes (DSBC), empleado para la construcción de aplicaciones a partir de componentes software ya desarrollados. El estilo de "desarrollo orientado" es utilizado en las organizaciones para desarrollar elementos de software concretos dentro de un proyecto de construcción, probablemente siguiendo prácticas DSCC. Habitualmente, en su elaboración, las especificaciones de estos elementos de software y sus implementaciones son almacenados en repositorios de software especiales y vinculados a la propia organización.

El campo de la Ingeniería de componentes se sustenta en las disciplinas de desarrollo "centradas" y "basadas"

Es usual también que estas organizaciones utilicen prácticas DSBC en la elaboración de sus nuevos proyectos, (re)utilizando para ello muchos de los componentes almacenados en los repositorios asociados a la organización, y que ya fueron desarrollados, probados y validados en otros proyectos previos. Sin embargo, como vimos en el *Capítulo 1*, esta práctica de reutilización de software es difícilmente aplicable en la realidad, ya que por regla general las organizaciones no llevan a cabo exhaustivas especificaciones de los componentes que deben desarrollar, ni tampoco cuentan con repositorios de software donde almacenarlas; y además, a esto tenemos que añadir el continuo interés por parte de las organizaciones por considerar el uso de componentes COTS para la elaboración de aplicaciones de software como prácticas DSBC-COTS.

Sin embargo, hoy día existe una falta de métodos, técnicas y herramientas que ofrezcan soporte para una práctica realista DSBC-COTS. En las primeras fases del ciclo de vida de un producto software (especialmente en fases de diseño de la arquitectura de software) se deben considerar unas implementaciones concretas de unos componentes software elaboradas por terceras partes y ajenas a las estrategias de desarrollo de la organización. La falta de (a) un modelo para la especificación de componentes comerciales (requisitos), (b) una notación adecuada para el prototipado de arquitecturas de software, y un (c) proceso que medie entre las arquitecturas de software (componentes abstractos) y los repositorios de software (componentes concretos) a la hora de buscar y seleccionar los componentes comerciales, son factores que han dificultado la consolidación de prácticas DSBC-COTS en el campo de la ingeniería del software. Como vimos en la sección 5.1.2, este último factor (c) es conocido como el problema de la conexión diseño-implementación.

5.3.2. Entidades y herramientas

En lo que sigue presentamos una estrategia para el desarrollo de aplicaciones de software con componentes COTS, basada en una metodología en espiral que integra las propuestas introducidas en capítulos anteriores. El método de desarrollo presentado básicamente está controlado por la interacción de tres actores o roles:

(a) el **arquitecto**, que representa al desarrollador del sistema y que define los requisitos de los componentes y sus interconexiones en una arquitectura de software;

(b) el **mediador**, que representa a un objeto que intercede entre las necesidades del arquitecto (los componentes abstractos de la arquitectura de software) y el repositorio con las especificaciones de los componentes COTS; y

(c) el **configurador**, que representa a un objeto que intercede entre el mediador y el arquitecto, y que realiza combinaciones entre los componentes candidatos (obtenidos por el mediador) que ofrecen soluciones a las necesidades del arquitecto (soluciones de arquitectura).

Ligados a estos roles hay cuatro entidades principales en el método, y que son: (1) la arquitectura de software, (2) un servicio mediación de componentes, (3) un servicio de integración, y (4) un modelo para la especificación de componentes COTS. Estas cuatro entidades, en su conjunto, resuelven la falta de los tres factores (citados anteriormente) que dificultan la consolidación de prácticas DSBC-COTS: los requisitos, la arquitectura y la búsqueda y selección de componentes comerciales. Desvelemos a continuación algunos aspectos de estas cuatro entidades, empezando por el modelo de especificación.

Especificación de componentes COTS

La utilización de documentos unificados que ayuden a describir el contenido de un componente COTS, son muy útiles para facilitar las tareas de búsqueda, selección y ensamblaje de componentes. Para estas labores, es necesario que dicha documentación contenga información de tipo funcional —signaturas, protocolos y comportamiento— y también información no funcional y no técnica. Basándonos en esta clase de información, en el *Capítulo 2* se introdujo un modelo de documentación de componentes COTS a través de sus plantillas COTScomponent, y que hemos adoptado para describir los requisitos de componente de una arquitectura software, y como fuente de información para el funcionamiento del servicio de mediación de componentes COTS, sobre el cual gira todo nuestro método DSBC-COTS.

Sólo a modo de ejemplo, en la figura 5.5 se muestra un ejemplo de un documento COTS que se corresponde con la especificación del componente `Translator` de la aplicación GTS. Como tratamos en el *Capítulo 2*, un documento COTS recoge la especificación de cuatro partes de un componente comercial. La parte `functional` describe aspectos funcionales, como descripción de las interfaces ofertadas y requeridas, pre/post y protocolos. La parte `Properties` describe aspectos no funcionales de la forma `(nombre,valor)`. La parte `packaging` es información de empaquetamiento, implantación e instalación. La parte `marketing` recoge información no técnica, como licencia, precio o detalles del vendedor.

Arquitectura de software

En relación a la arquitectura de software (AS), son numerosas las propuestas existentes de lenguajes para la definición de arquitecturas (LDA). Ejemplos de estos lenguajes son Acme, LEDA, MetaH, Rapide, SADL o Wright, entre otros (véase sección 1.6.3). No obstante, los

```
<COTScomponent name="Translator"
               xmlns="http://www.cotstrader.com/COTS-XMLSchema.xsd" ...>
  <functional>
    <providedInterfaces>
      <interface name="Translator">
        <description notation="CORBA-IDL">...</description>
        <behavior notation="JavaLarch"><description>...</description></behavior>
      </interface>
    </providedInterfaces>
    <requiredInterfaces>
      <interface name="FileCompressor">...</interface>
      <interface name="ImageTranslator">...</interface>
      <interface name="XDR">...</interface>
      <interface name="XMLBuffer">...</interface>
    </requiredInterfaces>
    <serviceAccessProtocol>
       <description notation="pi-protocols" href=".../trans.pi"/>
    </serviceAccessProtocol>
  </functional>
  <properties notation="W3C">
    <property name="capacity"><type>int</type><value>20</value></property>
    <property name="formatConversion" composition="AND">
      <property name="compress"><type>string</type><value>ZIP,TAR</value></property>
      <property name="image"><type>string</type><value>DWG,DXF</value></property>
    </property>
  </properties>
  <packaging><description notation="CCM-softpkg" href=".../trans.csd"/></packaging>
  <marketing>
    <license href="..."/><expirydate>...</expirydate><certificate href="..."/>
    <vendor><companyname>Geographic Features corp.</companyname>
      <webpage>http://www.gtranslator.es</webpage><mailto>sales@gt.es</mailto>
    </vendor><description>Geographic Translator Component</description>
  </marketing>
</COTScomponent>
```

Figura 5.5: Especificación del componente `Translator` de la aplicación GTS

actuales LDAs carecen de expresividad suficiente para poder desarrollar AS con COTS. Aunque estos LDAs permiten definir elementos arquitectónicos, como puertos, conectores y protocolos (entre otros), una vez más, el problema radica en que éstos no pueden recoger especificaciones COTS con información como la que hemos visto en el apartado anterior.

En recientes trabajos hemos observado una tendencia a utilizar notación UML para la descripción de AS [Garlan et al., 2001] [Hofmeister et al., 1999] [Medvidovic et al., 2002]. Las extensiones de UML, como las restricciones, valores etiquetados, los estereotipos y las notas, pueden utilizarse para representar elementos arquitectónicos como conectores, protocolos o propiedades, y también capturar información de especificación (como la de un documento COTS) complementándose por ejemplo con diagramas de casos de uso, o diagramas de comportamiento, entre otros. No obstante, para sistemas complejos, donde el número de componentes comerciales es elevado, la utilización de muchas clases (diagrama de clases) podría complicar la descripción de la AS.

Para el método, hemos adoptado la notación UML-RT de Bran Selic [Selic, 1999] para modelar la descripción de una arquitectura de software. Una de las ventajas que tiene este tipo de LDA, frente a las demás, es que dispone de un conjunto de símbolos que ayudan a crear rápidamente una vista de componentes de la AS, eliminando vistas con muchas clases UML. En el *Capítulo 1* ya presentamos una breve introducción a esta notación, y

también justificamos las ventajas de su utilización. Seguidamente volvemos a resumir estas justificaciones que nos llevan a utilizar UML-RT como LDA para el método DSBC-COTS que estamos proponiendo:

(1) Como la tendencia actual es a utilizar UML para describir arquitecturas, y dado que UML-RT también lo es, también es posible las representaciones de modelado tradicionales, como los diagramas de clases, diagramas de estados o diagramas de secuencias, entre otros.

(2) UML-RT adopta la notación original de ROOM (*Real-time Object Oriented Modelling*), también desarrollada por Bran Selic en [Selic et al., 1994]. ROOM (y por extensión UML-RT) utiliza un conjunto reducido de notaciones gráficas que cubren todas las necesidades para hacer una representación visual de una arquitectura de software en poco tiempo.

(3) Los sistemas basados en componentes COTS suelen requerir un prototipado rápido de la arquitectura de software para permitir continuos refinamientos de la misma. UML-RT permite hacer un prototipado rápido de una arquitectura de software.

Servicio de mediación

Un servicio de mediación es un proceso que intercede entre las necesidades del desarrollador o del arquitecto del sistema, necesidades que son definidas en una arquitectura de software con los requisitos de los componentes que conforman en su conjunto el sistema que hay que construir, y el/los repositorio/s con las especificaciones de unas implementaciones comerciales que se corresponden con componentes software desarrollados por tercera partes. Un servicio de mediación para componentes comerciales resuelve el problema de la conexión diseño implementación ya que busca y selecciona las implementaciones concretas de los componentes software residentes en sitios perfectamente accesibles (repositorios) que cumplen con las definiciones de diseño de los componentes abstractos (arquitectura).

Para el método DSBC-COTS proponemos la herramienta COTStrader como servicio de mediación para búsqueda y selección automática de componentes COTS, la cual fue discutida en el *Capítulo 3*. Como vimos, el servicio de mediación COTStrader está basado en un modelo de mediación para componentes comerciales (introducido en el mismo capítulo) y que extiende el modelo de mediación de ODP para soportar esta clase de componentes.

Servicio de integración

Un servicio de integración es un proceso que ofrece soluciones de implementación para una arquitectura de software predeterminada. A diferencia del servicio de mediación, que enfrenta una a una las especificaciones de los componentes abstractos de la arquitectura con las especificaciones de los componentes comerciales residentes en repositorios de software, el servicio de integración enfrenta de forma conjunta, todas las especificaciones abstractas de la arquitectura con una colección especial de especificaciones de componentes comerciales (componentes candidatos), obteniendo en dicho enfrentamiento múltiples combinaciones de componentes concretos que ofrecen soluciones (total o parcialmente) a los requisitos de los componentes de la arquitectura. A estas combinaciones las hemos denominado como "configuraciones" de arquitectura en el *Capítulo 4*. Para el método DSBC-COTS proponemos la herramienta COTSconfig como servicio de integración de arquitecturas, introducido también en dicho capítulo.

5.3.3. Descripción del método de DSBC-COTS

Las prácticas DSBC están cambiando algunas de las herramientas y métodos de la ingeniería del software, y las actuales soluciones de desarrollo con componentes se basan normalmente en metodologías en espiral que producen progresivamente requisitos, especificaciones arquitectónicas y diseños del sistema más detallados mediante repetidas iteraciones.

El método de desarrollo propuesto está basado en un modelo en espiral y utiliza las cuatro herramientas citadas anteriormente para la construcción de aplicaciones de software con componentes COTS. Además, como ya hemos adelantado, éste método ofrece una propuesta de solución para el problema de la conexión diseño-implementación.

Para construir una aplicación de software, el método comienza definiendo la arquitectura de software preliminar del sistema a partir de los requisitos del usuario. Estos requisitos definen la estructura del sistema a alto nivel, exponiendo su organización como una colección de componentes interconectados. Estos componentes de arquitectura ("componentes abstractos") se enfrentan a la colección de componentes COTS ("componentes concretos") disponibles en repositorios de software. Este proceso de enfrentamiento se corresponde con un proceso de búsqueda y selección de componentes que produce una lista de componentes candidatos que pueden formar parte de la aplicación: bien porque proporcionan algunos de los servicios requeridos o bien porque cumplen algunos de los requisitos de usuario extra funcionales, como por ejemplo el precio, o limitaciones de seguridad, entre otros.

A partir de esta lista con los posibles componentes candidatos que podrían formar parte de la aplicación, otro proceso se encarga de realizar todas las posibles combinaciones entre estos componentes para construir el sistema. Estas diferentes combinaciones (configuraciones) deben ser generadas y mostradas al diseñador del sistema para que éste tome ciertas decisiones sobre: qué configuración es la que mejor casa con las restricciones impuestas en la arquitectura, qué componentes abstractos no han sido solucionados y cuáles de ellos necesitarían ser implementados, y cómo debería cambiarse la arquitectura de software (y en qué casos esto es factible) para incorporar los componentes COTS encontrados.

Luego, los requisitos del sistema son casados con los requisitos de la configuración de arquitectura, obtenida y seleccionada en el paso anterior. Si es necesario, estos requisitos de sistema podrían ser revisados adecuadamente para ajustarlos a las nuevas necesidades del sistema y ser enfrentados de nuevo contra los repositorios de componentes concretos. El ciclo descrito se repite de forma indefinida hasta obtener una configuración de arquitectura de software que cumpla todos los requisitos de usuario de la aplicación, o porque el propio arquitecto del sistema lo decida.

Como se puede comprobar, la arquitectura de software es redefinida (re-ajustada) en cada iteración. En las etapas iniciales, en lugar de establecer las especificaciones de los componentes abstractos a priori, estas se obtienen tras analizar antes algunos de los componentes software disponibles en el mercado de componentes, y que son similares a los que se necesitan. La primera vez que se realiza el proceso de búsqueda y selección de componentes (un proceso de mediación), éste se lleva a cabo sobre características principales de los componentes, utilizando para ello solamente palabras claves (previamente establecidas) y emparejamientos suaves (*soft*). A medida que la arquitectura de software se va refinando cada vez más en cada iteración, las búsquedas también se van haciendo cada vez más exactas.

Veamos a continuación las etapas en las que se descompone el método anteriormente descrito para construir aplicaciones de software con componentes comerciales. A modo de ejemplo, durante la exposición iremos construyendo la aplicación GTS del ejemplo.

5.4. Etapas del método de desarrollo

El diagrama de bloques que se muestra en la figura 5.6 ilustra la secuencia de etapas que definen el comportamiento del método. Como se puede comprobar, dicho método está basado en un modelo en espiral que integra diversas metodologías de la ingeniería del software en su desarrollo: arquitecturas de software, especificación de componentes, búsqueda de componentes, configuración y composición de componentes, o evaluación de componentes. En su recorrido, en el método intervienen las cuatro entidades y sus herramientas, enumeradas en la sección 5.3.2, a recordar: (a) la especificación de los componentes comerciales usando las plantillas COTScomponent, (b) la arquitectura de software usando la notación UML-RT, (c) un proceso de búsqueda y selección de componentes comerciales usando el servicio de mediación COTStrader, (d) un proceso de composición de componentes usando el servicio de integración COTSconfig.

Como se puede comprobar, el método de desarrollo propuesto se compone de seis etapas, a saber: **etapa 1**, definición de la arquitectura de software en notación UML-RT, **etapa 2** generación de plantillas de componentes a partir de los requisitos de componente de la arquitectura, **etapa 3** búsqueda y selección de los componentes necesitados, dentro del repositorio de software, **etapa 4** integración de configuraciones de arquitectura a partir de los componentes candidatos extraídos, **etapa 5** cierre de las configuraciones de arquitectura obtenidas, y **etapa 6** evaluación de los resultados obtenidos. Veamos a continuación una descripción más detallada para cada una de las etapas del método presentado.

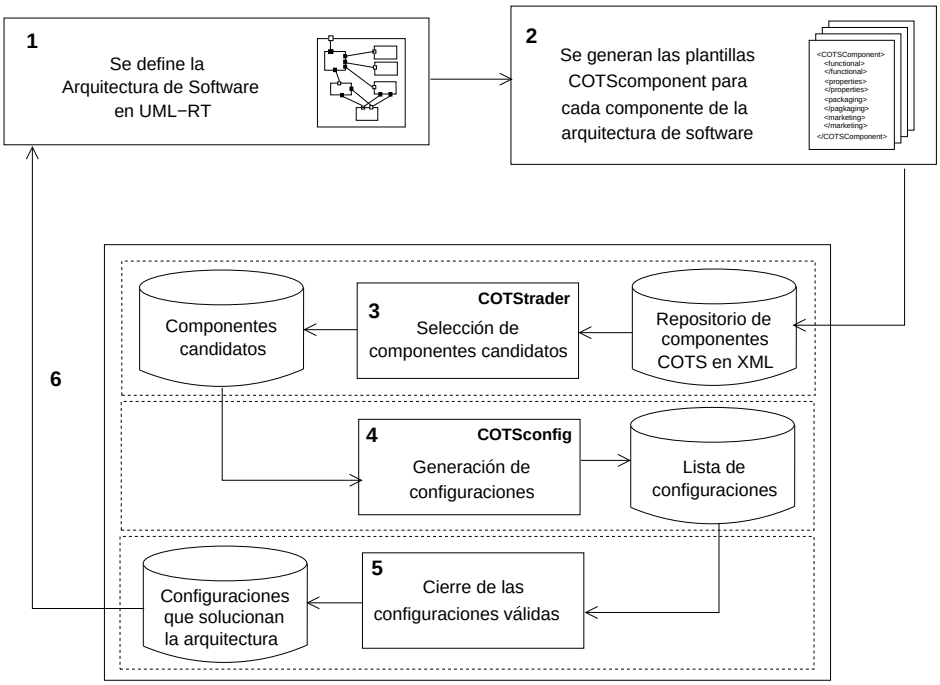

Figura 5.6: Un método de desarrollo basado en mediación

5.4.1. Definición de la arquitectura de software

Primero se define la estructura de la aplicación que hay que construir mediante una arquitectura de software AS. Para escribir la vista de componentes de AS el método propone la notación UML-RT. Concretamente, para construir la arquitectura de software de la aplicación GTS que mostramos en la figura 5.7, hemos usado la herramienta comercial Rational

Rose RealTime (`http://www.rational.com/support`), por ser ésta la que actualmente integra la versión original UML-RL de Selic y Rumbaugh [Selic y Rumbaugh, 1998].

Como se puede comprobar en la descripción UML-RT de la arquitectura GTS, los componentes se describen mediante cajas llamadas "cápsulas", y que a su vez pueden contener otras. Así por ejemplo, la cápsula `/trans:Translator` significa que `trans` es una instancia del componente `Translator`. Cada cápsula (componente) puede tener una o varias interfaces ("puertos") que pueden ser de dos tipos, ofertadas (cuadro pequeño de color blanco) y requeridas (negro). Un puerto se denota con un nombre y un protocolo, que especifica el orden en el que se envían los mensajes. Por ejemplo, el puerto `+/FileCompressor:ProtComp` es una interfaz llamada `FileCompressor` y un protocolo `ProtComp`, y `ProtComp~` es el protocolo dual. Por último, los "conectores" son líneas que unen dos puertos. Si une más de un puerto, estos se deben duplicar en la cápsula que lo requiera—mostrado con doble cuadro pequeño, como sucede en las interfaces `Document` y `Element` del componente `DOM`.

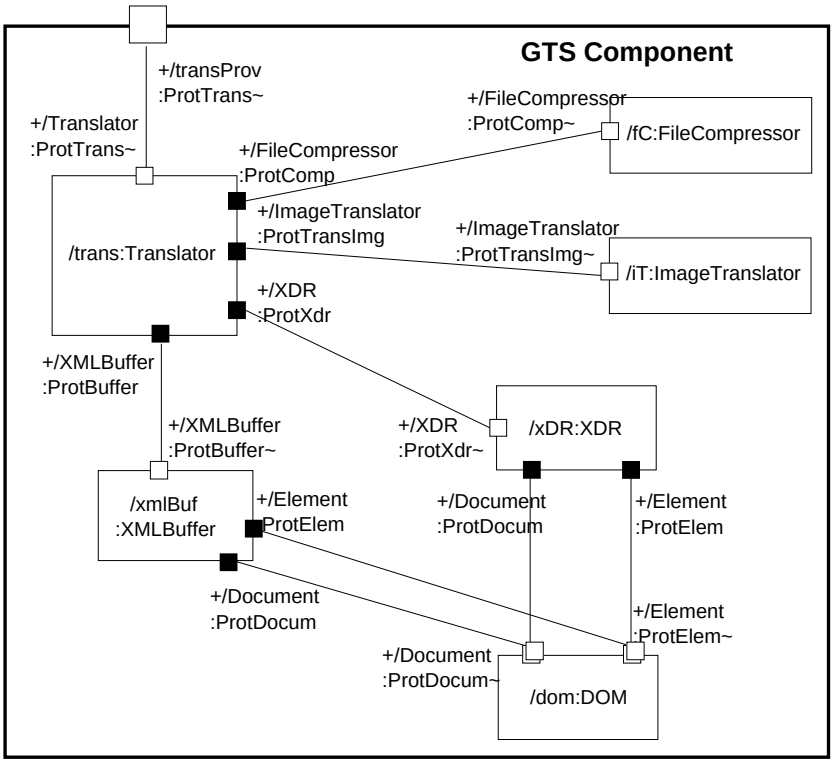

Figura 5.7: Arquitectura de software del componente GTS en notación UML-RT

En la vista de componentes de la figura 5.7, `trans:Translator` es el componente principal, con una interfaz `Translator` y otras cuatro requeridas: (a) un compresor de archivos `fC:FileCompressor`, estilo zip/unzip; (b) un conversor de imágenes geográficas `iT:ImageTranslator` como *Geographic Translator* o *MapObjects*; (c) un componente `xDR:XDR` para representación de datos en XML; (d) un buffer XML `xmlBuf:XMLBuffer` que almacena `<UUID,href>`; y (e) dos interfaces DOM para trabajar en XML para el XDR y el Buffer, disponibles en paquetes como XML4J de IMB o JAXP de Sun.

Los requisitos de un componente se fijan a partir del tipo información que puede acomodar un documento COTS (como el que se ha visto a modo de ejemplo en la figura 5.5, y tratado el *Capítulo 2*), esto es: requisitos funcionales, requisitos extra-funcionales, requisitos de empaquetamiento (implantación e implementación del componente) y requisitos de marketing (precio, condiciones de licencia, disponibilidad, etc.).

En nuestra propuesta, una cápsula (componente) de la arquitectura puede ser extendida mediante notas, estereotipos y valores etiquetados para acomodar los requisitos de un componente abstracto. A modo de ejemplo, usaremos el componente de la figura 5.8 para analizar cómo se recogen los requisitos de un componente. Esta figura muestra la vista interna de una cápsula que se corresponde con la del componente `ImageTranslator`. Como se puede comprobar, la descripción de una interfaz de componente es incluida en una nota UML que comienza por el estereotipo `<<interface>>`, para indicar: que el contenido de la nota es una descripción de interfaz y para diferenciarla también de otras notas que pueda contener la cápsula. Asociada a la nota se incluye un valor etiquetado externo para referirse al tipo de notación usada en la descripción (p.e., `{notation="CORBA IDL"}`)[2].

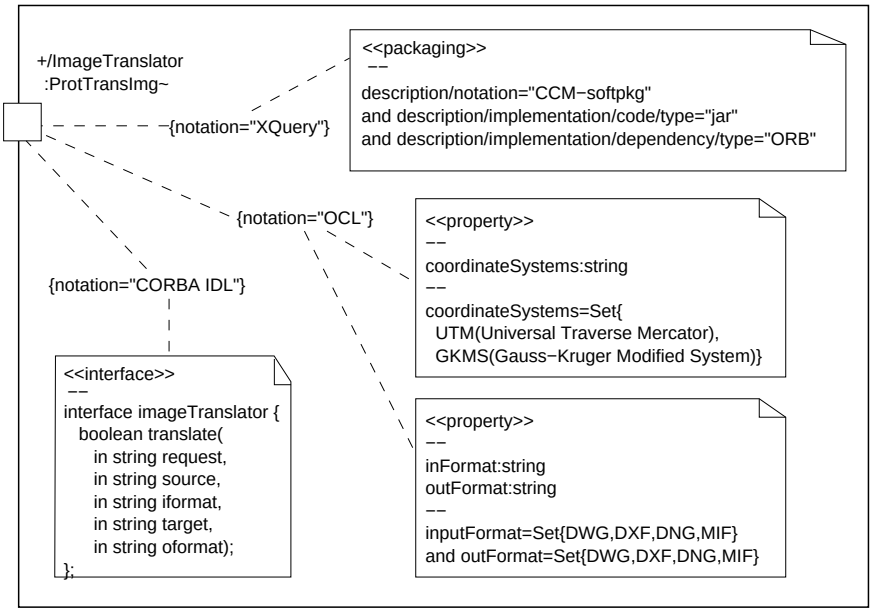

Figura 5.8: Definición de los requisitos del componente ImageTranslator

Los nombres de los estereotipos coinciden con los nombres de los elementos de una plantilla de componente COTScomponent

Las propiedades de un componente (esto es, información extra-funcional en un documento COTS) se pueden describir también en notas independientes. Una nota de propiedad comienza con el nombre de estereotipo `<<property>>`, seguido por la descripción de la propiedad en una notación particular. De forma similar a las notas de interfaz, la notación de la descripción es representada mediante un valor etiquetado externo asociado a la nota. Por ejemplo, obsérvese cómo para el componente `ImageTranslator` se han definido dos propiedades en dos notas independientes, utilizando notación OCL y expresado por el valor etiquetado `{notation="OCL"}`. Obsérvese también que se utiliza el símbolo "`--`" para separar las distintas partes de una nota de propiedad: (a) la cabecera con el estereotipo `<<property>>`, (b) la declaración de los tipos usados en la descripción OCL, y (c) la declaración OCL en sí.

El método también permite imponer restricciones de empaquetamiento y de marketing utilizando la notación XQuery (`http://www.w3.org/TR/query-algebra`), expresado mediante el valor etiquetado externo `{notation="XQuery"}`. Concretamente, para el caso de la información de empaquetamiento de un documento COTS, tal y como lo tratamos

[2]Aunque RoseRT define las interfaces dentro de los puertos, en nuestro caso también incluimos la descripción de la interfaz como un comentario adicional al puerto a través de una nota UML para facilitar así un prototipado rápido visual de la arquitectura (condición indispensable para un método DSBC que se base en un modelo en espiral, como lo es el que aquí proponemos)

en el *Capítulo 2*, se utiliza la notación "softpackage" de CCM para describir esta clase de información. A modo de ejemplo, en la figura 5.9 se muestra una información de empaquetamiento, expresada en la notación CCM, de un posible componente comercial o componente de mercado.

Sin embargo, en una nota de empaquetamiento (como la que se muestra en la figura 5.8) la notación XQuery es utilizada para establecer las restricciones de empaquetamiento CCM que debería tener un documento COTS de un componente comercial concreto. Esta información, como veremos más adelante, es utilizada por el proceso mediador para llevar a cabo la búsqueda y selección de documentos COTS cuyas definiciones CCM de empaquetamiento cumplan dichas restricciones impuestas en notación XQuery. Por supuesto, también se podría haber incluido directamente la descripción CCM en una nota UML y haberle asociado un valor etiquetado externo `{notation="CCM-softpkg"}`, en lugar de usar notación XQuery.

```
<?xml version="1.0"?>
<softpkg name="TranslatorService" version="1.0">
  <pkgtype>CORBA Component</pkgtype>
  <idl id="IDL:vendor1/Translator:1.0">
     <link href="http://.../Translator.idl"/>
  </idl>
  <implementation>
    <os name="WinNT" version="4.0"/><os name="AIX"/>
    <processor name="x86"/><processor name="sparc"/>
    <runtime name="JRE" version="1.3"/>
    <programminglanguage name="Java"/>
    <code type="jar">
      <fileinarchive name="Translator.jar"/>
      <entrypoint>TranslatorService.translator</entrypoint>
    </code>
    <dependency type="ORB"><name>ORBacus</name></dependency>
  </implementation>
</softpkg>
```

Figura 5.9: Un documento de paquete en notación CCM de un componente comercial

Por todo esto, la nota de empaquetamiento asociada al componente `ImageTranslator` (vista en la figura 5.8) significa que, para construir la aplicación GTS, se requiere un componente comercial cuyo documento COTS asociado tenga los siguientes requisitos de empaquetamiento: (a) que la sección de empaquetamiento del documento COTS esté descrita utilizando una notación "CCM-softpkg", (b) el componente requerido debe ser ofrecido e implantado en un paquete ".jar", (c) el componente requerido debe funcionar bajo alguna herramienta "ORB", como por ejemplo Orbacus de IONA.

Para los demás componentes (cápsulas) de la arquitectura de software de la aplicación GTS se impondrían sus requisitos de forma similar a como se ha descrito para el componente `ImageTranslator`.

Para concluir esta sección, en la figura 5.10 mostramos una imagen donde se puede ver el entorno de trabajo en Rational Rose RT que hemos utilizado para analizar y desarrollar la arquitectura de software del ejemplo GTS. Como se puede comprobar, para describir la arquitectura en UML-RT puede ser necesario estudiar y complementar la estructura del sistema utilizando otros diagramas UML, como por ejemplo diagramas de clases o diagramas de secuencias, como ha sucedido en nuestro caso. Para el método propuesto solamente es necesario el diagrama resultante con la arquitectura de software en UML-RT.

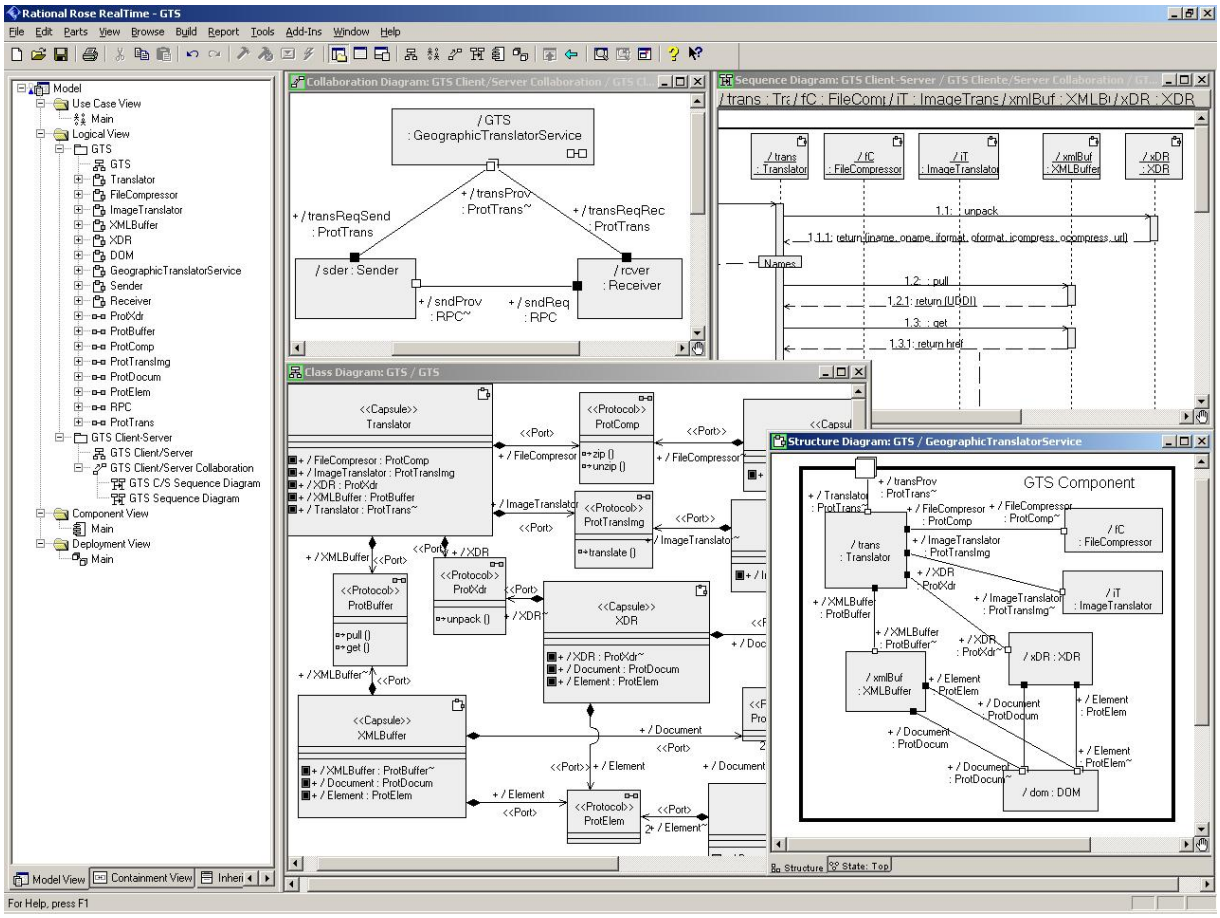

Figura 5.10: Entorno de trabajo en RationalRose-RT para modelar la arquitectura GTS

5.4.2. Generación de plantillas de componente

Una vez definida la arquitectura de software de la aplicación, seguidamente se deberá extraer del diagrama UML-RT la información de los componentes, los servicios que estos requieren y ofrecen, y sus propiedades. Para este propósito, el método sugiere la utilización de un proceso que analice el archivo de arquitectura que crea Rational Rose RealTime (un archivo RTMDL) y que luego genere una lista de plantillas COTScomponent (como la mostrada en la figura 5.5) con la descripción de los componentes encontrados en dicha arquitectura. Igualmente, por cada plantilla de componente se debe generar también una plantilla COTSquery necesaria para interrogar más tarde al servicio de medicación cuando se lleven a cabo las labores de búsqueda y selección de los componentes (etapa siguiente).

Con el fin de generalizar y no depender de una herramienta de diseño gráfica concreta (como RoseRT), y por tanto de unos formatos de archivos particulares a la herramienta seleccionada (RTMDL), el método sugiere la utilización de alguna herramienta gráfica que almacene los diseños arquitectónicos en formato XMI, y de otra herramienta genérica que procese estos archivos XMI y genere plantillas COTScomponent[3].

[3]Dado que la herramienta gráfica finalmente seleccionada para realizar nuestras pruebas ha sido Rational Rose RT, ha prevalecido la idea de convertir los archivos RTMDL a plantillas COTScomponent, ya que en la actualidad la herramienta Rose RT todavía no permite la conversión a formato XMI. No obstante, este formato está siendo considerado por Rational para ser incluido en posteriores versiones, como una sugerencia y discusión que mantuvimos con algunos miembros del equipo de trabajo de Rational Europa.

5.4.3. Invocación del servicio de mediación (COTStrader)

Una vez obtenida la lista de las plantillas COTScomponent con los servicios que deben tener los componentes de nuestro sistema GTS, el siguiente paso consiste en llamar al servicio de mediación COTStrader. A partir de estas plantillas, que describen los componentes abstractos, el servicio de mediación COTStrader produce una lista de los componentes candidatos que están disponibles para implementar el sistema. Este proceso ya fue descrito en el *Capítulo 3*.

Como hemos mencionado anteriormente, las operaciones de emparejamiento comienzan con emparejamientos suaves, principalmente buscando sólo por palabras clave, para llegar a ser cada vez más exactos en cada iteración. Los niveles de emparejamiento típicos son: palabras claves, información de marketing y de empaquetamiento (sistemas operativos, modelo de componentes, etc.), propiedades de calidad, nombres de interfaces, operaciones de interfaces, e información de comportamiento y semántica. Aunque estos últimos emparejamientos son en teoría bastante útiles, nuestra experiencia evidencia la dificultad de ir más allá de las búsquedas a nivel de propiedades de calidad. Los vendedores de software no suelen incluir los nombres de las interfaces desde donde se proporcionan sus servicios, y ni que decir de la semántica.

Volvamos de nuevo al ejemplo de nuestra aplicación GTS. Para nuestro ejemplo contamos con un repositorio de componentes —asociado y mantenido por el servicio de mediación— construido a partir de la información ofrecida por diferentes vendedores de componentes comerciales[4]. A partir de este repositorio, el mediador genera una lista de candidatos con 8 componentes que proporcionan uno (o más) de los 7 servicios definidos en los 6 componentes abstractos especificados en la arquitectura de software.

En la tabla 5.2 se muestra la lista de los componentes candidatos obtenida para el ejemplo GTS (tercera columna de la tabla, denotados de la forma C_i). Por comodidad en la exposición, para describir dicha tabla y referirnos a los componentes de la misma, hemos utilizado la nomenclatura empleada en el *Capítulo 4*, correspondiente al análisis de las configuraciones (la siguiente etapa del método).

Nombre de componente	GTS: Arquitectura (Especificaciones abstractas)	$C_{\mathcal{B}}(GTS)$: Componentes candidatos (Especificaciones concretas)
FileCompressor	$FC = \{R_{FC}\}$	$C_1 = \{R_{EL}, R_{DO}\}$
ImageTranslator	$IT = \{R_{IT}\}$	$C_2 = \{R_{EL}, R_{DO}, \overline{R}_{TL}\}$
XDR	$XD = \{R_{XD}, \overline{R}_{EL}, \overline{R}_{DO}\}$	$C_3 = \{R_{FC}\}$
XMLBuffer	$BF = \{R_{BF}, \overline{R}_{EL}, \overline{R}_{DO}\}$	$C_4 = \{R_{FC}, \overline{R}_{EL}, \overline{R}_{DO}\}$
DOM	$DM = \{R_{EL}, R_{DO}\}$	$C_5 = \{R_{IT}\}$
Translator	$TR = \{R_{TR}, \overline{R}_{FC}, \overline{R}_{IT}, \overline{R}_{XD}, \overline{R}_{BF}\}$	$C_6 = \{R_{IT}, R_{FC}, \overline{R}_{FL}\}$
		$C_7 = \{R_{BF}, \overline{R}_{EL}, \overline{R}_{DO}\}$
		$C_8 = \{R_{XD}, \overline{R}_{EL}, \overline{R}_{DO}\}$

Tabla 5.2: Los componentes de la arquitectura GTS y los componentes candidatos

En la primera columna de la tabla se muestran los nombres de los 6 componentes que intervienen en la arquitectura de software de la aplicación GTS. En la segunda columna se muestran los mismos componentes, pero ahora como colecciones de servicios ofrecidos y requeridos. Por simplicidad, usaremos sólo dos caracteres para nombrar a los componentes y servicios ofertados y requeridos. Así por ejemplo, el componente `FileCompressor` será referido como FC, y un servicio ofrecido por éste será referido como R_{FC}. Hecha esta

[4]Más tarde, en la sección 5.5 discutiremos algunas valoraciones acerca de este repositorio y de los componentes comerciales que hemos creado.

puntualización, volvamos a la tabla. En la segunda columna se pueden ver además los 7 servicios definidos por los componentes de la arquitectura GTS: R_{FC}, R_{IT}, R_{XD}, R_{BF}, R_{EL}, R_{DO} y R_{TR}. Estos 7 servicios de arquitectura son los que se han enfrentado al repositorio del mediador para generar la lista de candidatos, mostrada en la tabla 5.2.

Obsérvese en la tabla que tanto el componente candidato C_2 como el C_6 requieren los servicios externos $\overline{R}_{TL}$ y $\overline{R}_{FL}$, respectivamente. El primero de ellos ($\overline{R}_{TL}$) representa un servicio de utilidad (*Tool*, *TL*) con una colección de métodos para transformar imágenes de satélite, como por ejemplo, métodos de redimensionamiento, rotación, o "morphig", entre otros. El segundo de ellos ($\overline{R}_{FL}$) ofrece un servicio de filtrado (*Filter*, *FL*) llamado así porque proporciona una colección de métodos para aplicar efectos especiales sobre imágenes de satélite, como por ejemplo efectos de ruido, segmentación, y ocultación, entre otros efectos. Para el ejemplo de la aplicación GTS que estamos desarrollando, hemos considerado estos dos componentes particulares (C_2 y C_6) con el objetivo de cubrir todos los casos posibles que se nos puedan presentar a la hora de hacer las distintas combinaciones entre los componentes comerciales candidatos en la siguiente etapa.

5.4.4. Generación de configuraciones (COTSconfig)

Una vez que el servicio de mediación ha obtenido la lista de los componentes candidatos con los componentes que en un principio podrían formar parte de la futura aplicación (al implementar uno o varios servicios de la arquitectura del sistema), la siguiente etapa consiste en combinar estos componentes de la lista para encontrar soluciones a la arquitectura de software. Esta etapa la lleva a cabo el servicio de integración de configuraciones COTSconfig, discutido en el *Capítulo 4*. Recordemos que una configuración es una combinación de componentes de la lista de candidatos que cumple todos (o una parte) de los requisitos de la arquitectura del sistema.

Dado que los componentes de la lista de candidatos se refieren a componentes complejos, como lo son los componentes comerciales, ofreciendo o requiriendo más de un servicio a la vez, y dado que en la lista de candidatos puede existir más de un componente ofreciendo servicios similares, el proceso de combinación puede generar más de una configuración de componentes, como soluciones alternativas a la arquitectura AS. Si embargo, tal y como vimos en el *Capítulo 4* (cuando tratamos el servicio de integración), no todas las configuraciones obtenidas por el generador de configuraciones son válidas para construir el sistema. Se puede dar el caso donde el proceso de generación no consiga encontrar implementaciones de servicio concretas (ofrecidos por los componentes candidatos) para las necesidades de servicio de los componentes de la arquitectura ("lagunas" de servicios). Por contra, también se puede dar el caso donde, para una misma configuración, varios componentes candidatos ofrezcan el mismo (o varios) servicio(s) necesitado(s) en la arquitectura de software ("solapamientos" entre servicios).

La tarea de producir configuraciones "válidas" a partir de una lista de candidatos, no es una tarea fácil, especialmente cuando los componentes pueden ofrecer y requerir más de un servicio a la vez. La idea consiste en explorar todas las posibles alternativas de configuración y descartar aquellas con problemas de solapamientos y lagunas de servicios.

Siguiendo con nuestro ejemplo de la aplicación GTS, la tabla 5.3 muestra un resumen con algunos de los resultados obtenidos por el servicio de integración de configuraciones COTSconfig a partir de la arquitectura GTS y de la lista de componentes candidatos mostrada en la tabla 5.2, previamente obtenida por el servicio de mediación COTStrader en la etapa anterior. De las 256 combinaciones posibles (2^8, siendo 8 el número de componentes que hay en la lista de candidatos), sólo 24 de ellas dieron lugar a configuraciones válidas, mostradas todas ellas en la tabla.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	Configuraciones	R	C
1	R_{EL}, R_{DO}	–	R_{FC}	–	R_{IT}	–	R_{BF}	R_{XD}	C_1, C_3, C_5, C_7, C_8	Si	Si
2	R_{EL}, R_{DO}	–	R_{FC}	–	–	R_{IT}	R_{BF}	R_{XD}	C_1, C_3, C_6–$\{R_{FC}\}, C_7, C_8$	Si	No
3	R_{EL}, R_{DO}	–	–	R_{FC}	R_{IT}	–	R_{BF}	R_{XD}	C_1, C_4, C_5, C_7, C_8	Si	Si
4	R_{EL}, R_{DO}	–	–	R_{FC}	–	R_{IT}	R_{BF}	R_{XD}	C_1, C_4, C_6–$\{R_{FC}\}, C_7, C_8$	Si	No
5	R_{EL}, R_{DO}	–	–	–	R_{IT}	R_{FC}	R_{BF}	R_{XD}	C_1, C_5, C_6–$\{R_{IT}\}, C_7, C_8$	Si	No
6	R_{EL}, R_{DO}	–	–	–	–	R_{IT}, R_{FC}	R_{BF}	R_{XD}	C_1, C_6, C_7, C_8	No	No
–	R_{EL}	R_{DO}	R_{FC}	–	–	–	–	–	NO: faltan R_{IT}, R_{BF} y R_{XD} (lagunas)	No	No
7	R_{EL}	R_{DO}	R_{FC}	–	R_{IT}	–	R_{BF}	R_{XD}	C_1–$\{R_{DO}\}, C_2$–$\{R_{EL}\}, C_3, C_5, C_7, C_8$	Si	No
8	R_{EL}	R_{DO}	R_{FC}	–	–	R_{IT}	R_{BF}	R_{XD}	C_1–$\{R_{DO}\}, C_2$–$\{R_{EL}\}, C_3, C_6$–$\{R_{FC}\}, C_7, C_8$	Si	No
9	R_{EL}	R_{DO}	–	R_{FC}	R_{IT}	–	R_{BF}	R_{XD}	C_1–$\{R_{DO}\}, C_2$–$\{R_{EL}\}, C_4, C_5, C_7, C_8$	Si	No
10	R_{EL}	R_{DO}	–	R_{FC}	–	R_{IT}	R_{BF}	R_{XD}	C_1–$\{R_{DO}\}, C_2$–$\{R_{EL}\}, C_4, C_6$–$\{R_{FC}\}, C_7, C_8$	Si	No
11	R_{EL}	R_{DO}	–	–	R_{IT}	R_{FC}	R_{BF}	R_{XD}	C_1–$\{R_{DO}\}, C_2$–$\{R_{EL}\}, C_5, C_6$–$\{R_{IT}\} C_7, C_8$	Si	No
12	R_{EL}	R_{DO}	–	–	–	R_{IT}, R_{FC}	R_{BF}	R_{XD}	C_1–$\{R_{DO}\}, C_2$–$\{R_{EL}\}, C_6, C_7, C_8$	No	No
13	R_{DO}	R_{EL}	R_{FC}	–	R_{IT}	–	R_{BF}	R_{XD}	C_1–$\{R_{EL}\}, C_2$–$\{R_{DO}\}, C_3, C_5, C_7, C_8$	Si	No
14	R_{DO}	R_{EL}	R_{FC}	–	–	R_{IT}	R_{BF}	R_{XD}	C_1–$\{R_{EL}\}, C_2$–$\{R_{DO}\}, C_3, C_6$–$\{R_{FC}\}, C_7, C_8$	Si	No
15	R_{DO}	R_{EL}	–	R_{FC}	R_{IT}	–	R_{BF}	R_{XD}	C_1–$\{R_{EL}\}, C_2$–$\{R_{DO}\}, C_4, C_5, C_7, C_8$	Si	No
16	R_{DO}	R_{EL}	–	R_{FC}	–	R_{IT}	R_{BF}	R_{XD}	C_1–$\{R_{EL}\}, C_2$–$\{R_{DO}\}, C_4, C_6$–$\{R_{FC}\}, C_7, C_8$	Si	No
17	R_{DO}	R_{EL}	–	–	R_{IT}	R_{FC}	R_{BF}	R_{XD}	C_1–$\{R_{EL}\}, C_2$–$\{R_{DO}\}, C_5, C_6$–$\{R_{IT}\}, C_7, C_8$	Si	No
18	R_{DO}	R_{EL}	–	–	–	R_{IT}, R_{FC}	R_{BF}	R_{XD}	C_1–$\{R_{EL}\}, C_2$–$\{R_{DO}\}, C_6, C_7, C_8$	No	No
–	–	R_{EL}, R_{DO}	–	–	–	–	R_{BF}	R_{XD}	NO: faltan R_{FC} y R_{IT} (lagunas)	No	No
19	–	R_{EL}, R_{DO}	R_{FC}	–	R_{IT}	–	R_{BF}	R_{XD}	C_2, C_3, C_5, C_7, C_8	Si	No
20	–	R_{EL}, R_{DO}	R_{FC}	–	–	R_{IT}	R_{BF}	R_{XD}	C_2, C_3, C_6–$\{R_{FC}\}, C_7, C_8$	Si	No
21	–	R_{EL}, R_{DO}	–	R_{FC}	R_{IT}	–	R_{BF}	R_{XD}	C_2, C_4, C_5, C_7, C_8	Si	No
22	–	R_{EL}, R_{DO}	–	R_{FC}	–	R_{IT}	R_{BF}	R_{XD}	C_2, C_4, C_6–$\{R_{FC}\}, C_7, C_8$	Si	No
23	–	R_{EL}, R_{DO}	–	–	R_{IT}	R_{FC}	R_{BF}	R_{XD}	C_2, C_5, C_6–$\{R_{IT}\}, C_7, C_8$	Si	No
24	–	R_{EL}, R_{DO}	–	–	–	R_{IT}, R_{FC}	R_{BF}	R_{XD}	C_2, C_6, C_7, C_8	No	No
–	–	–	–	–	–	–	R_{BF}	R_{XD}	NO: faltan R_{EL}, R_{DO}, R_{FC} y R_{IT} (lagunas)	No	No

Tabla 5.3: Algunas configuraciones que solucionan la arquitectura GTS

En la tabla anterior, las columnas 2 a la 9 muestran los servicios que proporcionan los 8 componentes en cada combinación. Así por ejemplo, la configuración número 1 es una combinación que contiene todos los componentes candidatos, excepto los componentes C_2, C_4 y C_6, y cada componente proporciona sólo un servicio, excepto el componente C_1 que ofrece dos. Otras configuraciones requieren que parte de sus componentes oculten algunos de sus servicios al estar estos proporcionados por otros componentes que intervienen en la misma configuración. Esto se puede ver por ejemplo en la configuración número 2 (entre otros), donde el componente C_6 oculta el servicio R_{FC} porque ya está incluido como servicio en el componente C_3. En la tabla se usa la nomenclatura del operador de ocultación (C_6–R_{FC}) discutido en el *Capítulo 4*. Por último, existen otras combinaciones de componentes candidatos que no conducen a configuraciones válidas por la falta de algunos de los servicios (expresado en la tabla como "lagunas").

5.4.5. Cierre de las configuraciones

Una vez obtenida la colección de todas las configuraciones posibles, ahora es necesario cerrarlas para obtener una aplicación completa. Una configuración "cerrada" se caracteriza porque ésta no requiere de ningún otro componente para poder funcionar: todos los servicios requeridos por sus componentes ($\overline{R}$) quedan satisfechos por otros servicios proporcionados (R). Obsérvese además, que no todas las configuraciones válidas tienen porqué ser cerradas: aunque una configuración resultante puede que no presente lagunas de servicios con

respecto a la arquitectura, puede que aquella (la configuración) contenga un componente COTS que requiera un servicio externo no contemplado en el diseño original de la arquitectura. Esta situación es bastante común en situaciones reales, como sucede por ejemplo cuando intentamos instalar una aplicación en nuestro PC y descubrimos que durante la instalación se requiere la presencia de otra aplicación (aparentemente no relacionada) que debería estar ya instalada para que la primera funcione correctamente.

Volviendo una vez más a nuestro ejemplo, de las 24 configuraciones obtenidas para el ejemplo GTS, sólo 2 de ellas son completamente cerradas: la configuración 1 y la 3. Estas dos configuraciones son las que precisamente no incluyen los componentes C_2 y C_6, que requieren los servicios R_{TL} y R_{FL} respectivamente (los servicios extras para la transformación y efectos especiales sobre imágenes de satélite extraídos del repositorio por el servicio de mediación en la etapa 3). Por consiguiente, como se ya ha indicado, ahora sería necesario un proceso que cierre las 22 configuraciones restantes, aún no cerradas. Esto se puede hacer invocando de nuevo al servicio COTStrader para que busque los servicios externos hasta que las configuraciones se cierren (en caso de conseguirse).

5.4.6. Evaluación de los resultados

Las condiciones de salida del servicio de mediación se pueden establecer como políticas internas al mismo, o directamente en la petición de consulta. Las soluciones generadas por el mediador (configuraciones válidas cerradas) son devueltas al arquitecto de la aplicación para que éste las evalúe. En el método no contemplamos ninguna técnica de decisión y evaluación concreta, dejando esto al arquitecto del sistema. Así por ejemplo, el arquitecto podría detectar qué componentes de la aplicación debe desarrollar, al no existir componentes comerciales para ellos, o cuales de ellos debe implantarse en su sistema. Además, si lo requiere, el arquitecto puede refinar la AS inicial para incorporar o eliminar nuevos requisitos (paso 1) en el sistema. De nuevo, los componentes de AS son trasladados a plantillas XML y ofrecidos al servicio de mediación (etapas 2 y 3) para que busque y genere otras soluciones, ahora en base a las nuevas restricciones. Este recorrido cíclico se repite de forma indefinida hasta que todas las restricciones impuestas en AS se hayan cumplido o hasta que el propio arquitecto lo decida.

5.5. Algunas valoraciones de la experiencia

En esta sección discutiremos algunos aspectos del impacto que podría tener actualmente la aplicabilidad del método propuesto en entornos comerciales. El primer aspecto se refiere a la documentación de componentes. Es cierto que en la actualidad muchos autores están proclamando una mejor documentación de los componentes, utilizando para ello diferentes notaciones y estrategias [Cheesman y Daniels, 2001]. La mayoría de las propuestas parten de información básica que necesita ser capturada para construir sistemas basados en componentes. Sin embargo, son pocas las propuestas que están soportadas por herramientas, y probablemente ninguna de ellas está ampliamente aceptada por la industria para documentar componentes software comerciales.

Para validar nuestro método hemos intentado construir un ejemplo de aplicación GIS (descrita en la sección 5.2) a partir de componentes COTS. En primer lugar, hemos investigado los sitios Web de diferentes proveedores de componentes software (tales como IBM, Sun, ComponentSource, Flashline, y OpenSource RedHat Community), intentando rellenar nuestras plantillas de componentes con la información disponible en aquellos componentes que se venden o licencian. El estudio se realizó durante la primera mitad del año 2002, y el

repositorio con estos ejemplos de especificaciones de componentes comerciales, puede encontrarse en la dirección `http://www.cotstrader.com/samples/templates/repository`.

Los resultados evidencian una clara separación entre (a) lo que realmente clama la comunidad investigadora, sobre cual es la clase de información necesaria para la reutilización de componentes (y especialmente si queremos tener mecanismos automatizados), y (b) la escasa información proporcionada por los vendedores de software. Más concretamente, en nuestros estudios hemos podido comprobar que sólo entre el 25 % al 40 % de la información descrita en nuestras plantillas era información realmente disponible por los vendedores; y esto en las plantillas se interpreta de la siguiente forma:

- Sección `<marketing>`. Normalmente, la mayor parte de esta información sí suele ser ofrecida por el vendedor del componente.
- Sección `<packaging>`. Principalmente hay disponibles características de implantación, como por ejemplo CPUs, sistemas operativos, lenguajes, etc.
- Sección `<properties>`. La información extra-funcional es realmente muy costosa de encontrar, a no ser que sea información a nivel de características soportadas, o algunas características muy específicas. Por ejemplo, en el caso de los componentes de conversión de imágenes de satélite (como son OpenMap de BBN o MapObject de ESRI) había información disponible sobre los formatos de conversión soportados (DXF, DWG, MIF, etc.), mapa de proyecciones (Ortográfica, Policónica, Azimutal, etc.), o incluso los tipos de sistemas de coordenadas permitidos (UTM, GKM, ECEF, etc.). Sin embargo, realmente no hemos encontrado mucha información relacionada con otros aspectos, como por ejemplo, de calidad de servicio.
- Sección `<functional>`. Aparentemente, esta información es la más "técnica" y en teoría la más fácil de proporcionar. La mayoría de las propuestas académicas en DSBC contemplan útiles soluciones que se basan en este tipo de información (interfaces de componente y comportamientos semánticos), y sin la cual una práctica DSBC (automatizada) parecería impensable. Sin embargo, sorprendentemente, esta información ha sido precisamente la más costosa de encontrar. Los vendedores, en caso de ofrecer algún tipo de información funcional, sólo se limitan a describir los nombres de algunas interfaces soportadas, y nada acerca de sus operaciones, o de sus protocolos o semántica.

Estos resultados evidencian la necesidad de disponer de mejores documentaciones de componentes, para el caso de querer llevar a cabo efectivos procesos de búsqueda y selección de componentes comerciales. Como también hemos visto, el llamado problema de la conexión diseño-implementación (o problema del *gap analysis*) describe el problema de cómo casar unas descripciones de componentes "abstractos" producidos por métodos y procesos DSBC, con descripciones "concretas" de los componentes disponibles en repositorios. Nuestra evidencia demuestra que aún estamos lejos de todo esto, ya que en realidad no se cuenta con información funcional, necesaria para comprobar si un componente COTS dado puede ser incorporado en una arquitectura de software descrita por las interfaces y los nombres de los métodos de los componentes que la constituyen.

Finalmente, otro aspecto que también hemos experimentado durante nuestras pruebas es la dificultad de implementar una arquitectura de software general con componentes COTS encontrados por Internet. Personalmente creemos que en muchas situaciones reales la arquitectura de un sistema no puede ser completamente implementada a partir de componentes COTS localizados desde repositorios de software, a no ser que estemos trabajando

en ambientes muy específicos, y/o usando líneas de productos arquitectónicos [Bosch, 2000] que dependen de repositorios de componentes propietarios y desarrollados anteriormente para aplicaciones similares.

5.6. Trabajos relacionados

El método de desarrollo presentado en este capítulo está directamente relacionado con las prácticas de DSBC para la construcción de aplicaciones de software. Aunque hoy día existe una gran cantidad de trabajos relacionados con esta práctica de desarrollo, no es tan extenso (aunque sí intenso) para el caso de las metodologías de construcción con componentes comerciales, donde sus herramientas y técnicas no están aún muy extendidas y parecen estar en sus primeras fases de desarrollo. No obstante, podemos destacar algunas líneas de investigación que tienen que ver con los trabajos de construcción de aplicaciones con componentes comerciales. Este es el caso del trabajo de Robert Seacord *et al.* [Seacord et al., 2001] que proponen unos procesos para la identificación de componentes basada en el conocimiento de las reglas de integración del sistema. Aunque la propuesta carece de una forma concreta para documentar componentes comerciales, es uno de los pocos trabajos que tratan con ejemplos reales de componentes comerciales. Este trabajo se desarrolla dentro del *COTS-Based Systems (CBS) Initiative* del *Software Engineering Institute* (SEI).

En segundo lugar destacamos la propuesta de los proyectos CAFE y ESAPS[5], ambos disponibles en el *European Software Institute* (ESI). Aunque son varias sus líneas de interés, destacamos la de Cherki *et al.* [Cherki et al., 2001], que describen una plataforma llamada Thales para la construcción de sistemas software basados en partes COTS. Esta propuesta utiliza herramientas Rational para definir la arquitectura software, y utiliza un diagrama de clases, en lugar de UML-RT como proponemos en el método aquí expuesto. Además, el trabajo carece de procesos automáticos para la mediación de componentes COTS.

5.7. Resumen y conclusiones del capítulo

El DSBC es una práctica de desarrollo de software dentro del campo de la ingeniería del software que aboga por la construcción de sistemas software mediante la búsqueda, selección, adaptación e integración de componentes software comerciales (COTS). En un mundo donde la complejidad de las aplicaciones está en continuo crecimiento, y donde el volumen de información disponible empieza a ser excesivamente elevado como para ser mantenido por intermediarios humanos, los procesos de mediación automáticos jugarán un papel bastante importante para el planteamiento de nuevos métodos de DSBC empleados para la construcción de sistemas software con componentes COTS y siguiendo metodologías de desarrollo en espiral (como parece ser la tendencia actualmente).

En este sentido, en el capítulo hemos visto cómo nuestras propuestas (modelo de documentación de componentes COTS, el modelo de mediación de componentes COTS, y el de análisis de las configuraciones) se pueden integrar con éxito en metodologías de DSBC —como la metodología en espiral de [Nuseibeh, 2001]— para la construcción de aplicaciones de software con componentes COTS. El método de desarrollo presentado intenta servir de propuesta de solución al problema de la conexión diseño-implementación, que generalmente se presenta en estilos de desarrollo ascendentes al tratar de juntar aspectos

[5]Disponibles en `http://www.esi.es/Cafe` y `http://www.esi.es/esaps`

de implementación (componentes comerciales) con características de diseño de un sistema (arquitecturas de software).

El método presentado se compone de seis etapas, y para su práctica se propone el uso de las distintas herramientas que hemos desarrollado a lo largo del presente documento: (a) plantillas COTScomponent para documentar los componentes comerciales, (b) el servicio de mediación COTStrader y (c) el servicio de integración de configuraciones de arquitectura, COTSconfig. Además, para la elaboración de la arquitectura de software del sistema, el método propone usar la notación UML-RT, extendiendo las representaciones gráficas de las cápsulas mediante notas, estereotipos y valores etiquetados, para modelar la recogida de los requisitos de los componentes del sistema.

Por último, para hacer la exposición del método DSBC presentado, hemos desarrollado un ejemplo de una aplicación GTS, la cual lleva a cabo conversiones de imágenes de satélites en entornos GIS distribuidos. La aplicación GTS, constituida por seis componentes comerciales, ha sido desarrollada en el capítulo para ilustrar el funcionamiento de cada una de las etapas que componen del método propuesto.

Capítulo 6

Conclusiones Finales

Capítulo 6

Conclusiones Finales

Contenidos

Es un hecho constatable el crecimiento en la utilización de prácticas de desarrollo de software reutilizable para la construcción de sistemas distribuidos y abiertos. Aunque en realidad la reutilización del software no representa nada nuevo —ya que es un tema ancestral en el área de la ingeniería del software— este crecimiento se debe en cierta medida al interés que hay latente hoy día en la organizaciones por los componentes comerciales (COTS), los cuales están impulsando aún más que se precipite el resurgir de este acontecimiento. Este interés por los componentes comerciales viene suscitada por el propósito de las organizaciones de recuperar las señas de identidad que subyacen en la reutilización del software: reducir los tiempos y costes de desarrollo del software, tratando de mejorar su fiabilidad y seguridad al permitir el reciclaje de componentes software basados en la experiencia, esto es, componentes que han sido suficientemente probados, depurados y validados con anterioridad dentro o fuera de la organización.

Aunque realmente es muy complicado predecir y cuantificar el impacto que podría tener en un futuro la utilización de las prácticas de desarrollo basadas en componentes COTS en las organizaciones, sí que puede cambiar la forma de pensar de los ingenieros a la hora de afrontar sus proyectos, ya que este planteamiento está haciendo que la ingeniería del software se enfrente a nuevos estilos de desarrollo de software, ascendentes y siguiendo algún modelo en espiral (p.e., [Nuseibeh, 2001]), frente al desarrollo tradicional, descendente y en cascada. Para éste último, los requisitos del sistema se van desglosando (refinando) sucesivamente en otros más simples hasta obtener los componentes finales. En el desarrollo ascendente, parte de los requisitos iniciales del sistema pueden ser directamente satisfechos por componentes software que ya han sido elaborados por terceras partes, y que se encuentran almacenados en repositorios de componentes públicos [Mili et al., 1995].

Una de las principales consecuencias de este estilo de desarrollo —ascendente y basado en componentes comerciales— es el acercamiento fehaciente de tres áreas importantes de la ingeniería del software (figura 6.1), como son las arquitecturas de software (A), los servicios de mediación (S) y las especificaciones de componentes (E). Este acercamiento ha venido propiciado por la inherente naturaleza del estilo de desarrollo ascendente: la necesidad de contemplar sistemáticamente las características de implementación de los componentes en fases de diseño del sistema. Además, este acercamiento también parece ser especialmente constatable en las "fronteras" de las áreas, aunque no en la misma proporción:

- **Arquitecturas y especificaciones (A-E).** Los componentes comerciales deben proporcionar adecuadas especificaciones para que estos puedan ser considerados en los diseños de arquitecturas de software, y por otro lado, estos diseños deben conducir a prototipos rápidos de arquitecturas que permitan considerar con facilidad estas especificaciones. En el caso de las arquitecturas de software, la tendencia parece encaminarse hacia prototipos de arquitecturas diseñadas mediante diagramas UML [Cheesman y Daniels, 2001] o usando notaciones como UML-RT [Selic et al., 1994]. Para el caso de las especificaciones de los componentes comerciales esto es hoy día un tema de debate abierto en el campo de la investigación.

- **Servicios de mediación y especificaciones (S-E).** Los actuales servicios de mediación (p.e., [OOC, 2001] [PrismTech, 2001] [Shmidt, 2001]) se basan en la función de mediación de ODP [ISO/IEC-ITU/T, 1997]. Esta función presenta serias limitaciones para el caso de los componentes comerciales, por ejemplo, que sólo funciona para objetos CORBA, que trabaja solamente sobre la información sintáctica de los

objetos, o que permite sólo interfaces simples, entre otra limitaciones. Estas limitaciones evidencian la necesidad de mejorar los actuales servicios de mediación para soportar especificaciones de componentes comerciales.

- **Arquitecturas y servicios de mediación (A-S).** Este acercamiento se ha dado en menor grado que los dos anteriores. Si bien los servicios de mediación no son considerados en las prácticas de diseño de las arquitecturas de software, sí que se utilizan motores de búsqueda convencionales para localizar las fichas técnicas de componentes comerciales en fases de evaluación. Sin embargo, en estas actividades interviene directamente el factor humano, sin llegar a existir una conexión automatizada entre los requisitos de diseño de la arquitectura y las características de implementación de los componentes comerciales. A esto se le conoce como "el problema de la conexión diseño-implementación" [Cheesman y Daniels, 2001], y en la actualidad sigue siendo un problema no resuelto por completo.

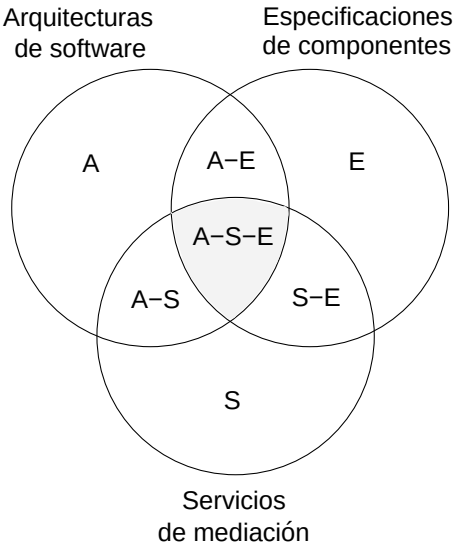

Figura 6.1: Situación de nuestro trabajo

La motivación ha sido buscar una solución para el Problema de la conexión diseño-implementación

Una de las principales motivaciones del presente trabajo ha sido la de tratar de acercar aún más estas tres áreas de la ingeniería del software (perspectiva A-S-E en la figura 6.1) con la intención de ofrecer una propuesta de solución al problema de la conexión diseño-implementación, anteriormente citado. Por tanto, el trabajo aquí presentado ofrece sus contribuciones dentro del paradigma del desarrollo de software basado en componentes, y en particular en los campos de (a) los componentes comerciales, (b) los modelos de mediación y (c) los modelos de documentación de componentes, en donde su principal aportación es la definición de un modelo de mediación de componentes COTS para la construcción de aplicaciones de software en entornos abiertos y distribuidos.

6.1. Aportaciones

Las principales aportaciones de este trabajo son las siguientes:

- En primer lugar, se ha identificado el escenario de trabajo, sus actividades y sus actores, sobre el cual se que han establecido las bases mínimas para el posterior desarrollo de las propuestas restantes [Iribarne y Vallecillo, 2000c].

- En segundo lugar, se ha realizado un estudio de la actual función de mediación de ODP [ISO/IEC-ITU/T, 1997] y se han identificado una serie de limitaciones para trabajar con componentes comerciales. Este estudio ha sido clave para el diseño posterior de un modelo de mediación para componentes COTS [Iribarne et al., 2001a] (siguiente aportación).
- En tercer lugar, se define un modelo de mediación de componentes diseñado para construir aplicaciones de software a partir de componentes comerciales. A partir de este modelo se ha desarrollado una implementación de un servicio de mediación, denominado COTStrader [Iribarne et al., 2001b].
- También se define un modelo para la documentación de componentes COTS, y un lenguaje en la notación XMLSchemas (W3C) que lo sustenta [Iribarne et al., 2001b] [Iribarne et al., 2001c]. Este lenguaje permite definir especificaciones de componentes comerciales en plantillas XML, denominadas COTScomponent.
- En cuarto lugar, se estudia y ofrece una solución a ciertos problemas que aparecen en la construcción de aplicaciones de software cuando se combinan especificaciones de componentes con múltiples interfaces: los problemas de las *lagunas* y *solapamientos* entre interfaces. También se extienden los tradicionales operadores de reemplazabilidad y compatibilidad de componentes para el caso de múltiples interfaces, y se define un algoritmo de configuración que utiliza estos operadores extendidos para calcular la lista de combinaciones de especificaciones de componente a partir de una colección de componentes candidatos dada. En relación a este último punto, también se ha desarrollado una implementación del algoritmo de configuración, denominado COTSconfig [Iribarne et al., 2002b].
- Para definir una arquitectura de software con información de componentes COTS, se realiza una propuesta que utiliza la notación UML-RT extendiendo las representaciones gráficas de las cápsulas mediante notas, estereotipos y valores etiquetados, para modelar la captura de los requisitos de los componentes del sistema [Iribarne et al., 2003b].
- Se ha definido un método de desarrollo de software semi-automatizado para la construcción de aplicaciones con componentes comerciales [Iribarne y Vallecillo, 2002], y que pone de manifiesto cómo se pueden integrar nuestras propuestas en algunos tipos de metodologías en espiral del DSBC, como la de [Nuseibeh, 2001].
- Se ha desarrollado un caso ejemplo completo de una aplicación de software con componentes comerciales, en el campo de los sistemas de información geográficos (SIG) [Iribarne et al., 2002a]. La aplicación ejemplo es un subsistema de conversión de imágenes geográficas, integrado por cinco componentes comerciales. El ejemplo ha servido para validar la integración de las propuestas en una metodología en espiral.
- Por último, también se ha desarrollado el sitio web `http://www.cotstrader.com` desde donde se ofrece soporte para la mediación de componentes COTS por Internet. Para facilitar las actividades de mediación desde el sitio web, se han desarrollado otras tres herramientas: (a) un visualizador "applet" de plantillas COTScomponent, (b) un esquema de estilos W3C (XSL) para la visualización de plantillas COTScomponent desde un navegador web, y (c) un analizador gramatical de esquemas W3C para comprobar si una plantilla XML es una instancia válida del lenguaje de documentación COTScomponent definido. Este analizador es utilizado por el servicio de mediación COTStrader en sus actividades de exportación de componentes.

6.2. Limitaciones y líneas de investigación abiertas

Si bien es cierto que la principal contribución de nuestro trabajo es que ofrece una propuesta de solución al problema de la conexión diseño-implementación a través del servicio de mediación en componentes comerciales, también presenta algunas limitaciones:

- En primer lugar, aunque el servicio de mediación desarrollado cumple algunos de los requisitos del modelo de mediación de componentes COTS definido, otros aspectos permanecen aún abiertos, como la composición y adaptación dinámica de servicios, el uso de funciones heurísticas, o la delegación de consultas en entornos federados. La propuesta tampoco permite la mediación "semántica", ya que no trata ni con ontologías ni con mediación basada en conocimiento.

- En segundo lugar, la propuesta necesita herramientas que automaticen las operaciones de emparejamiento entre componentes a nivel de protocolos y a nivel semántico. Es muy importante contar con un soporte automatizado para las comprobaciones de compatibilidad y reemplazabilidad de servicios, ya que estas son dos aspectos claves en el DSBC. Aunque en la actualidad existen varias herramientas para ciertas notaciones formales, que llevan a cabo comprobaciones automáticas sobre operaciones de compatibilidad y reemplazabilidad, estas (las herramientas) no están disponibles por Internet para que puedan ser ampliamente utilizadas.

- En tercer lugar, el servicio de mediación COTStrader no soporta los actuales avances en el campo Web, como los "servicios webs" o los "webs semánticos", siendo interesante su integración con otras notaciones (WSDL, WSFL, RDF), recursos y repositorios disponibles por Internet.

Por otro lado, es evidente que el trabajo presentado en esta memoria tiene unas connotaciones eminentemente prácticas en el área de la ingeniería del software basada en componentes, influenciado en parte por varios motivos:

(a) En primer lugar por las exigencias tecnológicas de la disciplina sobre la que se sustenta este trabajo (DSBC).

(b) En segundo lugar, por el fuerte auge y aceptación que ha tenido (y sigue teniendo) en estos últimos años el campo de los sistemas basados en componentes comerciales.

(c) Y en tercer lugar, y no menos importante, por el escenario donde se aplica este trabajo: los sistemas distribuidos y abiertos en Internet.

Además, es necesario añadir que todas estas suposiciones están fuertemente sugestionadas por la evolución continua de una masa tecnológica que está motivada más bien por intereses de mercado, y no científicos. Sin embargo, son también precisamente estos intereses de mercado los que están favoreciendo la paulatina consolidación de un nuevo mercado de componentes, basados en estándares y marcos teóricos que están propiciando la emergencia de la disciplina de la ingeniería del software basada en componentes (ISBC) [Heineman y Councill, 2001] [Wallnau et al., 2002].

6.3. Trabajo futuro

Tomando como referencia las limitaciones y líneas de investigación abiertas reseñadas en el apartado anterior, consideramos de especial interés el acometimiento de los siguientes temas de trabajo para proseguir la labor científica que aquí hemos presentado:

- En primer lugar, el estudio de aquellos aspectos que han quedado abiertos para la adecuación del servicio de mediación con el modelo de mediación, esto es:
  (a) estudiar y desarrollar la composición y adaptación dinámica de servicios,
  (b) estudiar, definir e integrar el uso de funciones heurísticas, y
  (c) estudiar y definir la delegación de consultas y la federación de servicios de mediación.
- En segundo lugar, creemos que es interesante el estudio y la integración de otras notaciones (como WSDL, WSFL o RDF), recursos y repositorios disponibles por Internet con el servicio de mediación COTStrader.
- En tercer lugar, también creemos que puede ser interesante la construcción de pasarelas hacia otros servicios de mediación (especialmente a aquellos CORBA) para soportar con esto entornos federados, en la forma parecida a como lo hacen los UBR sobre los repositorios UDDI.

Apéndice A

Sintaxis BNF y XML-Schema De Un Documento COTS

Apéndice A

Sintaxis BNF y XML-Schema de un Documento COTS

Contenidos

El presente apéndice contiene la definición completa del lenguaje para la documentación de componentes COTS, tratado en el *Capítulo 2*. En primer lugar se ofrece la definición del lenguaje de documentación usando la notación BNF, luego se presenta su traducción a la notación XMLSchemas del W3C para facilitar así su utilización en plantillas XML.

A.1. Sintaxis BNF

A.1.1. Sintaxis de un documento COTScomponent

COTScomponent ::= `<COTScomponent` [*template-attributes*] `>`
COTScomponent-body `</COTScomponent>`
COTScomponent-body ::= *functional-element*
[*properties-element*] [*packaging-element*] [*marketing-element*]

A.1.1.1. Sintaxis de la parte funcional

functional-element ::= `<functional>` *functional-body* `</functional>`
functional-body ::= *providedInterfaces-element*
[*requiredInterfaces-element*] [*serciveAccessProtocol-element*]
[*consumedEvents-element*] [*producedEvents-element*]
providedInterfaces-element ::= `<providedInterfaces>` *providedInterfaces-body*
`</providedInterfaces>`
providedInterfaces-body ::= *interface-list*
interface-list ::= *interface-element* [*interface-list*]
interface-element ::= `<interface` *name-attribute*`>` *interface-body* `</interface>`
interface-body ::= *description-element* [*behavior-element*]
behavior-element ::= `<behavior` *notation-attribute*`>` *behavior-body* `</behavior>`
behavior-body ::= *description-element*
[*exactMatching-element*] [*softMatching-element*]
requiredInterfaces-element ::= `<requirededInterfaces>` *requiredInterfaces-body*
`</requiredInterfaces>`
requiredInterfaces-body ::= *interface-list*
serciveAccessProtocol-element ::= `<choreography>`
serciveAccessProtocol-body `</choreography>`
serciveAccessProtocol-body ::= *description-element*
[*exactMatching-element*] [*softMatching-element*]
consumedEvents-element ::= `<consumedEvents>` *event-list* `</consumedEvents>`
producedEvents-element ::= `<producedEvents>` *event-list* `</producedEvents>`
event-list ::= *event-element* [*event-list*]
event-element ::= `<event>` *string-literal* `</event>`

A.1.1.2. Sintaxis de la parte no funcional

properties-element ::= *properties-included* | *properies-hrefered*
properties-hrefered ::= `<properties` *notation-attribute href-attribute* `/>`
properties-included ::= `<properties` *notation-attribute*`>`
property-element-list `<properties>`
property-element-list ::= *property-element* [*property-element-list*]

property-element ::= *property-simple* | *property-composed*
property-simple ::= <property *name-attribute*> *property-body* </property>
property-body ::= *type-element value-element* [*implementedBy-element*]
type-element ::= <type> *string-literal* </type>
value-element ::= *value-included* | *value-hrefered*
value-included ::= <value> *string-literal* </value>
value-hrefered ::= <value *href-attribute*/>
implementedBy-element ::= <implementedBy> *string-literal* </implementedBy>
property-composed ::= <property *property-composed-attributes*>
property-composed-body </property>
property-composed-attributes::= *name-attribute composition-attribute*
[*priority-attribute*]
property-composed-body ::= *property-simple property-simple*
[*property-composed*] | *property-simple* [*property-composed*]
composition-attribute ::= composition = "*composition-value*"
composition-value ::= AND | OR
priority-attribute ::= priority = *priority-value*
priority-value ::= *digit*

A.1.1.3. Sintaxis de la parte de empaquetamiento

packaging-element ::= <packaging> *description-element* </packaging>

A.1.1.4. Sintaxis de la parte de marketing

marketing-element ::= *marketing-included* | *marketing-hrefered*
marketing-hrefered ::= <marketing *href-attributed*/>
marketing-included ::= <marketing> *marketing-body* </marketing>
marketing-body ::= [*license-element*] [*expirydate-element*]
[*certificate-element*] [*vendor-element*]
[*description-element*] [*more-elements*]
license-element ::= *license-included* | *license-hrefered*
license-hrefered ::= <license *href-attribute*/>
license-included ::= <license> *literal-string* </license>
expirydate-element ::= <expirydate> *expirydate-body* </expirydate>
expirydate-body ::= [*day-element*] *month-element year-element*
day-element ::= <day> *day-value* </day>
month-element ::= <month> *month-value* </month>
year-element ::= <day> *year-value* </year>
certificate-element ::= *certificate-included* | *certificate-hrefered*
certificate-hrefered ::= <certificate *href-attribute*/>
certificate-included ::= <certificate> *literal-string* </certificate>
vendor-element ::= <vendor> *vendor-body* </vendor>
vendor-body ::= *companyname-element*
[*webpage-list*] [*mailto-list*] [*address-list*]
companyname-element ::= <companyname> *literal-string* </companyname>
webpage-list ::= *webpage-element* [*webpage-list*]
webpage-element ::= <webpage> *url* </webpage>
mailto-list ::= *mailto-element* [*mailto-list*]
mailto-element ::= <mailto> *mail-reference* </mailto>

address-list ::= *address-element* [*address-list*]
address-element ::= `<address>` *address-body* `</address>`
address-body ::= *zip-element street-element city-element country-element*
zip-element ::= `<zip>` *literal-string* `</zip>`
street-element ::= `<street>` *literal-string* `</street>`
city-element ::= `<city>` *literal-string* `</city>`
country-element ::= `<country>` *literal-string* `</country>`
more-elements ::= others XML elements defined by the importer-client

A.1.2. Sintaxis del servicio de importación: COTSquery

COTSquery-template ::= `<COTSquery` [*template-attributes*] `>`
COTSquery-body `</COTSquery>`
COTSquery-body ::= *COTSdescription-element* [*query-constraints*]
COTSdescription-element ::= *COTSdescription-hrefered*
| *COTSdescription-included*
COTSdescription-hrefered ::= `<COTSdescription` *href-attribute* `/>`
COTSdescription-included ::= `<COTSdescription>` *COTScomponent* `</COTSdescription>`
query-constraints ::= *functionalMatching-element* [*propertyMatching-element*]
[*packagingMatching-element*] [*marketingMatching-element*]

A.1.2.1. Sintaxis de emparejamiento funcional

functionalMatching-element ::= `<functionalMatching>`
functionalMatching-body `</functionalMatching>`
functionalMatching-body ::= *interfaceMatch-element* [*sapMatch-element*]
| [*interfaceMatch-element*] *sapMatch-element*
interfaceMatching-element ::= `<interfaceMatching>` *matching-type* `</interfaceMatching>`
sapMatching-element ::= `<choreographyMatching>`
matching-type `</choreographyMatching>`
matching-type ::= *exactMatching-type* | *softMatching-type*

A.1.2.2. Sintaxis de emparejamiento no funcional

propertyMatching-element ::= *constraints-element* [*preferences-element*]
constraints-element ::= *default-constraints-notation* | *other-constraints-notation*
default-constraints-notation ::= `<constraints` *default-notation-attribute*`>`
default-contrains-body `</constraints>`
default-constraints-attribute ::= `notation="`*default-constraints-value*`"`
default-constraints-value ::= `XMLQuery`
default-contraints-body ::= *expression*
other-constraints-notation ::= `<constraints` *notation-attribute*`>`
other-contraints-body `</constraints>`
other-contraints-body ::= *literal-string*
preferences-element ::= *default-preferences-notation* | *other-preferences-notation*
default-preferences-notation ::= `<preferences` *default-preferences-attribute* `>`
default-preferences-body `</preferences>`
default-preferences-attribute ::= `notation="`*default-preferences-value*`"`
default-preferences-value ::= `ODP`
default-preferences-body ::= `first` | `random` | *min-preference* | *max-preference*

min-preference ::= `min(`*expression*`)`
max-preference ::= `max(`*expression*`)`
other-preferences-notation ::= `<preferences` *notation-attribute* `>`
other-preferences-body `</preferences>`
other-preferences-body ::= *literal-string*

A.1.2.3. Sintaxis de emparejamiento de empaquetamiento

packagingMatching-element ::= *default-packMatch-notation*
| *other-packMatch-notation*
default-packMatch-notation ::= `<packagingMatching` *default-packMatch-attribute*`>`
default-packMatch-body `</packagingMatching>`
default-packMatch-attribute ::= `notation="`*default-packMatch-value*`"`
default-packMatch-value ::= `XMLQuery`
default-packMatch-body ::= XMLQuery syntax
other-packMatch-notation ::= `<packagingMatching` *notation-attribute*`>`
other-packMatch-body `</packagingMatching>`
other-packMatch-body ::= *literal-string*

A.1.2.4. Sintaxis de emparejamiento de marketing

marketingMatching-element ::= *default-marketMatch-notation*
| *other-marketMatch-notation*
default-marketMatch-notation ::= `<marketingMatching`
default-marketMatch-attribute`>` *default-marketMatch-body* `</marketingMatching>`
default-marketMatch-attribute ::= `notation="`*default-marketMatch-value*`"`
default-marketMatch-value ::= `XMLQuery`
default-marketMatch-body ::= XMLQuery syntax
other-marketMatch-notation ::= `<marketingMatching` *notation-attribute*`>`
other-marketMatch-body `</marketingMatching>`
other-marketMatch-body ::= *literal-string*

A.1.3. Sintaxis de elementos en general

A.1.3.1. Sintaxis de emparejamientos Hard y Soft

exactMatching-type ::= *exactMatching-omitted* | *exactMatching-element*
softMatching-type ::= *softMatching-omitted* | *softMatching-element*
exactMatching-omitted ::= `<exactMatching/>`
exactMatching-omitted ::= `<softMatching/>`
exactMatching-element ::= `<exactMatching` *href-attribute* `/>`
softMatching-element ::= `<softMatching` *href-attribute* `/>`

A.1.3.2. Sintaxis de atributos

template-attributes ::= *name-attribute* [*namespace-attibute-list*]
name-attribute ::= `name="`*name-value*`"`
namespace-attibute-list ::= *namespace-attribute* [*namespace-attibute-list*]
namespace-attribute::= `xlmns` [`:` *namespace*] `=` *namespace-value*

A.1.3.3. Sintaxis del elemento `description`

description-element ::= *description-included* | *description-hrefered*
description-included ::= `<description` *notation-attribute*`>` *description-body* `</description>`
description-body ::= *literal-string*
description-hrefered ::= `<description` *notation-attribute href-attribute* `/>`
href-attribute ::= `href="`*href-value*`"`
notation-atribute ::= `notation="`*notation-value*`"`
notation-value ::= *literal-string*
href-value ::= [*url-header*] *url-body*
url-body ::= *alpha-symbols* [*url-body*]
url-header ::= `http://` | `file://`

A.1.3.4. Elementos terminales

expression ::= `(` *expression* [*operator expression*] `)` | *literal-string*
operator ::= *relational-operators* | *logical-operators*
relational-operators ::= `<` | `>` | `<=` | `>=` | `=` | `!=`
logical-operators ::= `and` | `or` | `not`
literal-string ::= *alpha-symbols* [*literal-string*]
alpha-symbols ::= *special-symbols* | *lower-character* | *digit*
special-symbols ::= `/` | `?` | `#` | `.` | `:` | `@`
lower-character ::= `a` through `z`
upper-character ::= `A` through `Z`
digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

A.2. XML-Schema de un documento COTS

```
<!--
################################################################################
#  File name   : COTS-XMLSchema.xsd                                            #
#  Document    : The COTS Component specification template                     #
################################################################################
-->

<!-- ********************************************************************** -->
<!--                          XML Schema namespace                          -->
<!-- ********************************************************************** -->

<xsd:schema  targetNamespace="http://www.cotstrader.com"
             xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">

<!-- ********************************************************************** -->
<!--                                Annotations                             -->
<!-- ********************************************************************** -->

<xsd:annotation>
 <xsd:documentation>
   An W3C XMLSchema template for COTS components
   Please report errors to <liribarne@ual.es>
 </xsd:documentation>
</xsd:annotation>

<!-- ********************************************************************** -->
```

```
<!--                            Global complex types                          -->
<!-- ********************************************************************** -->

<xsd:complexType name="locationType">
 <xsd:attribute name="notation" type="xsd:string" minOccurs="0"/>
 <xsd:choice>
  <xsd:attribute name="href" type="xsd:uriReference"/>
  <xsd:simpleContent>
   <xsd:extension base="xsd:string"/>
  </xsd:simpleContent>
 </xsd:choice>
</xsd:complexType>

<xsd:complexType name="referenceType">
 <xsd:attribute name="href" type="xsd:uriReference" minOccurs="0"/>
</xsd:complexType>

<!-- ********************************************************************** -->
<!--                            COTS Component elements                       -->
<!-- ********************************************************************** -->

<!-- Exporting: Component description -->
 <xsd:element name="COTScomponent" type="exportingType"/>

<!-- Importing: Component query -->
 <xsd:element name="COTSquery" type="importingType"/>

<!-- ###################################################################### -->
<!--                             Exporting description                        -->
<!-- ###################################################################### -->

<xsd:complexType name="exportingType">
 <xsd:attribute name="name" type="xsd:string"/>
 <xsd:sequence>
  <xsd:element name="functional" type="functionalType"/>
  <xsd:element name="properties" type="propertiesType" minOccurs="0"/>
  <xsd:element name="packaging" type="packagingType" minOccurs="0"/>
  <xsd:element name="marketing" type="marketingType" minOccurs="0"/>
 </xsd:sequence>
</xsd:complexType>

<!-- ******************** Functional description ********************** -->

<xsd:complexType name="functionalType">
  <xsd:sequence>
    <xsd:element name="providedInterfaces" type="InterfaceList"/>
    <xsd:element name="requiredInterfaces" type="InterfaceList" minOccurs="0"/>
    <xsd:element name="consumedEvents" type=eventType"" minOccurs="0"/>
    <xsd:element name="producedEvents" type=eventType"" minOccurs="0"/>
    <xsd:element name="choreography" type="behaviorType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complezType>

<xsd:complexType name="InterfaceList">
 <xsd:sequence>
  <xsd:element name="interface" type="InterfaceType" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="InterfaceType">
 <xsd:attribute name="name" type="xsd:string"/>
 <xsd:sequence>
  <xsd:element name="description" type="locationType"/>
  <xsd:element name="exactmatching" type="referenceType"/>
  <xsd:element name="softmatching" type="referenceType"/>
  <xsd:element name="behavior" type="behaviorType"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="behaviorType">
 <xsd:attribute name="notation" type="xsd:string" minOccurs="0"/>
 <xsd:sequence>
  <xsd:element name="description" type="locationType"/>
  <xsd:element name="exactmatching" type="referenceType"/>
  <xsd:element name="softmatching" type="referenceType"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="eventType">
  <xsd:element name="event" type="xsd:string" maxOccurs="unbounded"/>
</xsd:complexType>

<xsd:complexType name="choreographyType">
 <xsd:sequence>
  <xsd:element name="description" type="locationType"/>
  <xsd:element name="exactmatching" type="referenceType"/>
  <xsd:element name="softmatching" type="referenceType"/>
 </xsd:sequence>
</xsd:complexType>

<!-- ********************** Non Functional description ***************** -->

<xsd:complexType name="propertiesType">
 <xsd:attribute name="notation" type="xsd:string"/>
 <xsd:choice>
  <xsd:attribute name="href" type="xsd:uriReference"/>
  <xsd:element name="property" type="propertyType" maxOccurs="unbounded"/>
 </xsd:choice>
</xsd:complexType>

<xsd:complexType name="propertyType">
 <xsd:attribute name="name" type="xsd:string"/>
 <xsd:attribute name="composition" minOccurs="0">
  <xsd:simpleType>
   <xsd:restriction base="xsd:string">
    <xsd:enumeration value="OR"/>
    <xsd:enumeration value="AND"/>
   </xsd:restriction>
  </xsd:simpleType>
 </xsd:attribute>
 <xsd:attribute name="priority" minOccurs="0">
  <xsd:simpleType>
   <xsd:restriction base="xsd:positiveInterger">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="9"/>
   </xsd:restriction>
```

```
  </xsd:simpleType>
 </xsd:attribute>
 <xsd:sequence>
  <xsd:element name="type" type="xsd:datatype"/>
  <xsd:element name="value" type="valueType" minOccurs="0"/>
  <xsd:element name="implementedBy" type="xsd:string" minOccurs="0"/>
  <xsd:element name="implementedIn" type="xsd:string" minOccurs="0"/>
  <xsd:element href="property"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="valueType">
 <xsd:choice>
  <xsd:attribute name="href" type="xsd:uriReference"/>
  <xsd:simpleContent>
   <xsd:extension base="xsd:string"/>
  </xsd:simpleContent>
 </xsd:choice>
</xsd:complexType>

<!-- ****************** Packaging/Architectural description ************* -->

<xsd:complexType name="packagingType">
 <xsd:element name="description" type="locationType"/>
</xsd:complexType>

<!-- ********************** Marketing description ******************** -->

<xsd:complexType name="marketingType">
 <xsd:sequence>
  <xsd:element name="license" type="locationType"/>
  <xsd:element name="expirydate" type="xsd:string"/>
  <xsd:element name="certificate" type="locationType"/>
  <xsd:element name="vendor" type="vendorType"/>
  <xsd:element name="description" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="vendorType">
 <xsd:sequence>
  <xsd:element name="companyname" type="xsd:string"/>
  <xsd:element name="webpage" type="xsd:uriReference" maxOccurs="unbounded"/>
  <xsd:element name="mailto" type="xsd:string" maxOccurs="unbounded"/>
  <xsd:element name="address" type="addressType" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="addressType">
 <xsd:sequence>
  <xsd:element name="street" type="xsd:string"/>
  <xsd:element name="zip" type="xsd:int" minOccurs="0"/>
  <xsd:element name="city" type="xsd:string"/>
  <xsd:element name="country" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>

<!-- ######################################################################## -->
<!--                          Importing description                           -->
```

```
<!-- ########################################################################## -->

<xsd:complexType name="importingType">
 <xsd:attribute name="name" type="xsd:string"/>
 <xsd:sequence>
  <xsd:element name="COTSdescription" type="COTSdescriptionType"/>
  <xsd:element name="functionalMatching" type="functionalMatchingType"/>
  <xsd:element name="propertyMatching" type="propertyMatchingType"/>
  <xsd:element name="packagingMatching" type="locationType"/>
  <xsd:element name="marketingMatching" type="locationType"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="COTSdescriptionType">
 <xsd:choice>
  <xsd:attribute name="href" type="xsd:uriReference" minOccurs="0"/>
  <xsd:element ref="COTScomponent"/>
 </xsd:choice>
</xsd:complexType>

<xsd:complexType name="functionalMatchingType">
 <xsd:sequence>
  <xsd:element name="interfaceMatching" type="matchingType"/>
  <xsd:element name="choreographyMatching" type="matchingType"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="matchingType">
 <xsd:sequence>
  <xsd:element name="exactMatching" type="referenceType" minoccurs="0"/>
  <xsd:element name="softMatching"  type="referenceType" minoccurs="0"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="propertyMatchingType">
 <xsd:sequence>
  <xsd:element name="constraints" type="locationType"/>
  <xsd:element name="preferences" type="locationType"/>
 </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

Glosario

Glosario

Active Server Page (ASP)
Es una tecnología Internet para la programación estilo CGI de Microsoft, usado en IIS. Véase *CGI, IIS, JSP*

Architecture Definition Language (ADL)
Lenguaje de definición de arquitecturas de software.

arquitectura de software Véase *ADL*
Definición de [Bass et al., 1998]: La arquitectura de un programa o sistema de computación es la estructura o estructuras del sistema, que están compuestas de componentes software, de las propiedades visibles de esos componentes, y las relaciones entre ellos.

artefacto
1. Módulo software. **2.** Hace referencia a una parte software que puede ser desde un código ejecutable, hasta una especificación de una interfaz, una arquitectura de software, una aplicación completa o un componente individual. Véase *asset*

ASP
Véase *Active Server Page*.

asset
Elemento software que puede referirse a código, documentos, herramientas, procesos, entre otros. Este término algunas veces es usado para referirse a un componente. Véase *artefacto*

bot
1. Abreviatura del término *robot* que aglutina variantes de motores de búsqueda, como *spiders* o *crawlers*, entre otros. **2.** Un motor de búsqueda inteligente especializado en la localización y selección automática y autónoma de información concreta. Véase *search engine*

cápsula
Elemento arquitectónico usado en *UML-RT* para modelar un componente software. Véase *UML-RT*

CBD
Component-Based Development. Véase *desarrollo basado en componentes*

CCM
Modelo de componentes de CORBA. Véase *CORBA*

COM
Véase *Component Object Model*.

Commercial Off-The-Shelf (COTS)
Componente software diseñado, elaborado y comercializado por terceras partes, generalmente de bajo precio o de libre distribución, utilizado por los desarrolladores de software como partes preexistentes que puede incorporar en su arquitectura sin crearlos.

Common Object Request Broker Architecture (CORBA)
Véase *OMG*, *IDL*, http://www.corba.org
Es un modelo de componentes estándar diseñado para permitir la interoperabilidad entre componentes de diferentes plataformas, lenguajes, autores o vendedores. En el mercado existe una gran variedad de implementaciones, por ejemplo Orbix y Orbacus de IONA, ObjectBroker de Bea Systems, o VisiBroker de Visigenic/Borland.

Component Adquisition
1. Adquisición de componentes. **2.** Véase sección 3.1.2

Component-Based Development (CBD)
Véase *Desarrollo de Software Basado en Componentes*.

Component Object Model (COM)
Modelo de componentes de Microsoft. Define un estilo arquitectónico para componentes software de Microsoft.

componente
Definición de [Szyperski, 1998]: Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio.

conector
Una conexión entre puertos de una colección de componentes que dan forma a un producto software. Un conector impone los requisitos de conexión sobre los puertos que conecta, y generalmente afecta a los protocolos de interacción.

CORBA
Véase *Common Object Request Broker Architecture*.

COTS
Véase *Commercial Off-The-Shelf*.

Data Type Definition (DTD)
Véase *DTD*, *schema*
Solución tradicional de documentos XML heredada de SGML. Las DTD se suelen usar para crear un modelo de datos para los documentos XML, lo que permite validar estos documentos. Actualmente las DTD se usan como *Schemas*, creados también por el W3C.

DCOM
Véase *Distributed Component Object Model*.

DTD
Véase *Data Type Definition*.

Desarrollo de Software Basado en Componentes (DSBD) Véase *artefacto*
Una perspectiva del desarrollo de software en donde todos los "artefactos" se pueden construir mediante ensamblaje, adaptación e integración de componentes existentes y en una variedad de configuraciones.

Distributed Component Object Model (DCOM)
Véase *Component Object Model (COM)*.

EJB
Véase *Enterprise Java Beans*.

Enterprise Java Beans (EJB)
Es el modelo de componentes del lado servidor de Sun Microsystems.

especificación
Un documento que prescribe de forma completa, precisa y fiable los requisitos, diseño, comportamiento o características de un sistema o de un componente.

especificación de requisitos
Un documento que contiene los requisitos de un componente. La especificación define un componente y puede ser usado como un contrato para construir el componente o para ser integrado en una arquitectura de software.

esquema
Un esquema es una gramática que establece la forma en la que se puede escribir un documento XML específico a partir de los elementos definidos por ésta.

estándar
Un documento disponible públicamente que define las especificaciones para las interfaces, servicios, procesos, protocolos o formatos de datos, y que es establecido y mantenido por consenso por un grupo. Ejemplo de estándar son los que emite la organización ISO. Véase *ISO*

eXtensible Markup Language (XML)
Lenguaje de marcas extensible heredado de SGML. Ampliamente usado para modelización de datos, intercambio de información, formato para el almacenamiento de información en repositorios, entre otras aplicaciones. Lenguaje propuesto por el W3C. Véase *W3C*

función de mediación
1. *trading function*. **2.** Especificación de un mediador. Véase *mediador*

IDL
Véase *Interface Definition Language*.

Ingeniería de requisitos
Definición de [Jordon y Davis, 1991]: La ingeniería de requisitos es el uso sistemático de principios, técnicas, lenguajes y herramientas que hacen efectivos el análisis, la documentación, la evolución de las necesidades de usuario y las especificaciones del comportamiento externo para cumplir aquellas necesidades de usuario.

Internet Information Server (IIS)
Es el servidor web de Microsoft con normas de seguridad.

Interface Definition Language (IDL)
Lenguaje para la definición de interfaces. Un IDL ofrece una descripción precisa de los tipos, métodos y excepciones de una interfaz de componente. Los objetos CORBA pueden tener múltiples interfaces compuestas o heredadas de otras.

International Standard Organization (ISO)
Organización internacional en emisión de estándares y documentos de estandarización.

ISO
Véase *International Standard Organization*.

Java API for XML Processing (JAXP)
Soporte de Sun Microsystems para la programación Java-XML del lado servidor.

JAXP
Véase *Java API for XML Processing*.

Véase *CGI, ASP, servlet*

Java Server Page (JSP)
Es una tecnología Internet para la programación estilo CGI de Sun Microsystems. Permite mezclar código java dentro de una página HTML.

JSP
Véase *Java Server Page*.

legacy systems
Sistemas heredados.

mediador
= intermediario

1. *trader*. **2.** Objeto software que media entre objetos que ofertan ciertas capacidades, que se denominan servicios, y otros objetos que demandan la utilización dinámica de estas capacidades. **3.** Un objeto software que busca dinámicamente servicios distribuidos.

middleware
Véase *software de interconexión*.

modelo de referencia
Es una descripción de los componentes software, servicio de componente (funciones) y las relaciones entre ellos (cómo se integran estos componentes y cómo interaccionan).

Véase *bot*

motor de búsquedas
1. *search engine*. **2.** Programa que rastrea, localiza y extrae información por Internet.

.NET
Entorno de servicios web de Microsoft para redes de área local, extensa e Internet.

Object Management Group (OMG)
Organización que emite recomendaciones para la elaboración de productos software para computación de objetos distribuidos.

Véase en http://www.omg.org

ORB
Véase *Object Request Broker*.

Object Request Broker (ORB)
Software de interconexión (*middleware*) que controla la comunicación y el intercambio de los datos de forma transparente entre objetos de un entorno heterogéneo distribuido.

Véase *middleware*

Object Management Architecture
Una arquitectura de gestión de objetos que está construyendo el OMG.

OMA
Véase *Object Management Architecture*.

OMG
Véase *Object Management Group*.

protocolo
1. Un conjunto de reglas sintácticas y semánticas para el intercambio de información. **2.** Orden en el que se deben llamar los métodos de una interfaz de componente software.

QoS
Véase *Quality of Service*.

Quality of Service (QoS)
Calidad de servicio de un componente. Hace referencia a atributos tales como el tiempo medio de respuesta del servicio, el tiempo máximo, la precisión de la respuesta, portabilidad, o la adaptabilidad, entre otros atributos.

Remote Procedure Call (RPC)
Llamada a procedimiento remoto. Es una infraestructura cliente/servidor que incrementa la interoperabilidad, portabilidad y flexibilidad de una aplicación permitiendo que éste pueda ser distribuida sobre múltiples plataformas heterogéneas.

requisito funcional
Algo que el componente software debe hacer.

requisito no funcional
Una propiedad o cualidad que un componente debe tener.

repositorio
Una base de datos o un servicio de almacenamiento distribuido común para componentes, modelos y sistemas. Normalmente es usado en entornos de desarrollo distribuido.

retrieval information
1. Recuperación de la información. **2.** Véase sección 3.1.1

RPC
Véase *Remote Procedure Call*.

search engine
Véase *motor de búsquedas*.

servicio
Definición de [ISO/IEC-ITU/T, 1997]: Un servicio es un conjunto de capacidades a nivel funcional proporcionadas por un objeto. Un servicio es una instancia de un tipo de servicio.

servicio web
Véase *WSDL, SOAP, UDDI*
Servicio Web de [W3C-WebServices, 2002]: Un servicio web en un sistema software identificado por un URI (un identificador de recursos uniforme) [Berners-Lee et al., 1998], cuyas interfaces públicas y enlaces están definidos y descritos en XML, y su definición puede ser localizada por otros sistemas de software que pueden luego interactuar con el servicio web en la manera preestablecida por su definición, utilizando mensajes basados en XML transmitidos por protocolos de Internet.

servicio de mediación
Véase *mediador*
1. *trading service*. **2.** Una implementación de la especificación de un mediador.

servlet
Véase *ASP, CGI, JSP*
Es una tecnología Internet para la programación estilo CGI de Sun Microsystems para Java. Permite desde java generar una página HTML en respuesta a una petición del lado cliente.

SGML
Véase *Standard Generalized Markup Language*.

Simple Object Access Protocol (SOAP)
Véase *WSDL, UDDI, servicio web*
SOAP es una especificación para usar documentos XML como mensajes. La especificación SOAP contiene una sintaxis para definir mensajes XML, un modelo para el intercambio de mensajes, un conjunto de reglas para representar los datos dentro de un mensaje, una guía para el transporte de mensajes SOAP sobre HTTP, y un convenio para usar llamadas RPC usando mensajes SOAP.

sistema abierto
Una colección de implementaciones de componentes software y hardware diseñados para satisfacer ciertas necesidades y con interfaces de componente bien definidas y mantenidas públicamente.

SOAP
Véase *Simple Object Access Protocol*.

software de interconexión
1. *middleware*. **2.** Sistema software que proporciona interoperabilidad entre servicios de aplicaciones, como computación de objetos distribuidos, y resuelve algunos aspectos de hardware y diferencias de sistemas operativos en un entorno heterogéneo y distribuido.

SOM
Véase *System Object Model*.

Standard Generalized Markup Language (SGML)
Lenguaje estándar de marcas.

System Object Model (SOM)
Implementación de IBM del modelo de objetos CORBA.

trader
Véase *mediador*.

trazabilidad
Forma de relacionar requisitos con elementos de implementación.

UBR
Véase *UDDI*
1. *UDDI Business Registry*. **2.** Afiliación de repositorio UDDI.

UDDI
Véase *Universal Description, Discovery, and Integration*.

UML
Véase *Unified Modeling Language*.

UML-RT
Lenguaje visual para la modelización de arquitecturas de software. Este lenguaje aparece como unión entre UML estándar y el lenguaje ROOM, usado para modelizar sistemas complejos y concurrentes en tiempo real.
Véase *UML*

Unified Modeling Language
Lenguaje de modelización de sistemas software desarrollado por OMG.
Véase *OMG*

Universal Description, Discovery, and Integration (UDDI)
Es una especificación de una función de directorio para la publicación, localización y enlace de servicios web. En su elaboración participan IBM, Microsoft y Ariba.

W3C
Véase *World Wide Web Consortium*.

Web Services Description Language (WSDL)
Lenguaje de definición de servicios web propuesto conjuntamente por Microsoft e IBM.

World Wide Web Consortium (W3C)
Organización que se dedica a la implementación y a la emisión de recomendaciones. Algunas propuestas del W3C son XML, SOAP o los *Schemas*, entre otras.
Véase en `http://www.w3.org`

WSDL
Véase *Web Services Description Language*.

XQuery

Véase *W3C, XML*

Lenguaje de consulta para XML desarrollado por W3C. Es una especificación de un lenguaje de consulta sobre plantillas XML propuesta en 2001. Actualmente existen muy pocas implementaciones de la especificación, pero la más conocida es XQEngine, disponible en W3C.

XML

Véase *eXtensible Markup Language*.

Acrónimos

Acrónimos

ADL	Architecture Description Language
API	Application Programming Interface
ASP	Active Server Page
CBD	Component-Based Development
CBSE	Component-Based Software Engineering
CCM	CORBA Component Model
CGI	Common Gateway Interface
COM	Common Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CMM	Capability Maturity Model
DSBC	Desarrollo de Software basado en Componentes
DCBSE	Distributed Component-Based Software Engineering
DCOM	Distributed Common Object Model
DCE	Distributed Computing Environment
DOM	Document Object Model
DTD	Document Type Definition
DSOM	Distributed System Object Model
EDOC	Enterprise Distributed Object Computing
EJB	Enterprise JavaBeans
GTS	Geographic Translator System
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IIS	Internet Information Servlet
ISBC	Ingeniería del Software Basada en Componentes
ISBC	Ingeniería del Software Basada en Componentes Distribuidos
ISO	International Standard Organization
ITU	International Telecommunications Union
JAXP	Java API for XML Processing
JRMP	Java Remote Method Protocol

JML	Java Modeling Language
JSP	Java Active Page
JVM	Java Virtual Machine
KBSE	Knowledge-Based Software Engineering
LDA	Lenguaje de Definición de Arquitecturas
MOM	Message-Oriented Middleware
MOTS	Modifiable Off-The-Shelf
NDI	Non-Developmental Item
NFR	Non-Functional Requirements
OCL	Object Constraint Language
ODP	Open Distributed Processing
OMA	Object Mangement Architecture
OMG	Object Mangement Group
OOSE	Object-Oriented Software Engineering
ORB	Object Request Broker
ORPC	Object Remote Procedure Call
OSF	Open Systems Foundation
POO	Programación Orientada a Objetos
QoS	Quality of Service
RMI	Remote Method Invocation
RM-ODP	Reference Model for Open Distributed Processing
RPC	Remote Procedure Call
ROOM	Real-Time Object-Oriented Modeling
RTMDL	Real-Time Model Definition Language
RUP	Rational Unified Model
SEI	Software Engineering Institute
SGML	Standard Generalized Markup Language
SIG	Sistemas de Información Geográficos
SOAP	Simple Object Access Protocol
SOM	System Object Model
UBR	UDDI Business Registry
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
UML-RT	UML Real Time
UUID	Universal Unique IDentifier
W3C	World Wide Web Consortium
WSBC	WebSphere Business Components

WSDL	Web Services Description Language
XDR	eXchange Data Representation
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language

Bibliografía

Bibliografía

[Abowd et al., 1993] Abowd, G., Allen, R., y Garlan, D. (1993). Using style to understand descriptions of software architecture. En *SIGSOFT'93: Foundations of Software Engineering*, páginas 71–80.

[Acherman et al., 1999] Acherman, F., Lumpe, M., Schneider, J., y Nierstrasz, O. (1999). Piccola—A Small Composition Language. `http://www.iam.unibe.ch/~scg/Research/Piccola/`.

[Addy y Sitaraman, 1999] Addy, E. A. y Sitaraman, M. (1999). Formal Specification of COTS-Based Software: A Case Study. En *Fifth Symposium on Software Reusability*, páginas 83–91.

[Agha et al., 1993] Agha, G., Frolund, S., Kim, W. Y., Panwar, R., Patterson, A., y Sturman, D. (1993). Abstraction and Modularity Mechanisms for Concurrent Computing. En Agha, G., Wegner, P., y Yonezawa, A. (eds.), *Research Directions in Concurrent Object-Oriented Programming*, páginas 3–21. MIT Press, London.

[Alencar y Goguen, 1992] Alencar, A. y Goguen, J. (1992). OOZE. En Stepney, S., Barden, R., y Cooper, D. (eds.), *Object Orientation in Z*, páginas 158–183. Springer-Verlag: Cambridge CB2 1LQ, UK. Workshops in Computing.

[Alford, 1985] Alford, M. W. (1985). SREM at the Age of Eight: The Distributed Computing Design System. *IEEE Computer*, 18(4):36–46.

[Allen y Garlan, 1997] Allen, R. y Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.

[America, 1991] America, P. (1991). Designing an Object-Oriented Programming Language with Behavioral Subtyping. En der Bakker, J., de roever, W., y Rozenberg, G. (eds.), *Foundations of Object-Oriented Languages, 1990 REX School/Workshop, Noordwijkerhout, The Netherlands*, número 489 de LNCS, páginas 60–90. Springer-Verlag.

[Attiya y Welch, 1998] Attiya, H. y Welch, J. (1998). *Distributed Computing: Fundamentals, Simulations and Advenced Topics.* McGraw-Hill. ISBN: 0-07-7093526.

[Azuma, 2001] Azuma, M. (2001). SquaRE the next generation of the ISO/IEC 9126 and 14598 international standards series on software product quality. *ESCOM (European Software Control and Metrics conference).* `http://www.escom.co.uk/conference2001/papers/azuma.pdf`.

[Bachman et al., 2000] Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., y Wallnau, K. (2000). Technical Concepts of Component-Based

Software Engineering. Informe Técnico número CMU/SEI-2000-TR-00, Software Engineering Institute (SEI). `http://www.sei.cmu.edu`.

[Bailin, 1994] Bailin, C. (1994). Object Oriented Requirements Analysis. En Marciniak, J. J. (ed.), *Encyclopedia of Software Engineering*, páginas 740–756. New York: John Wiley & Sons.

[Barnes, 1997] Barnes, J. (1997). *High Integrity Ada: the SPARK Approach*. Addison-Wesley. ISBN: 0-20-11751-77.

[Basili y Boehm, 2001] Basili, V. R. y Boehm, B. (2001). COTS-Based Systems Top 10 List. *IEEE Computer*, 30(5):91–93.

[Bass, 2001] Bass, L. (2001). *Component-Based Software Engineering. Putting the Pieces Together*, capítulo Software Architecture Design Principles, páginas 389–403. Addison-Wesley.

[Bass et al., 1998] Bass, L., Clements, P., y Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley. ISBN: 0-201-19930-0.

[Bastide y Sy, 2000] Bastide, R. y Sy, O. (2000). Towards Components that Plug AND Play. En Vallecillo, A., Hernández, J., y Troya, J. M. (eds.), *ECOOP'2000 Workshop on Object Interoperability (WOI'00)*, páginas 3–12.

[Bastide et al., 1999] Bastide, R., Sy, O., y Palanque, P. (1999). Formal Specification and Prototyping of CORBA Systems. En *ECOOP'99*, número 1628 de LNCS, páginas 474–494. Springer-Verlag.

[Bearman, 1997] Bearman, M. (1997). *Tutorial on ODP Trading Function*. Faculty of Information Sciences Engineering. University of Canberra. Australia.

[Beitz y Bearman, 1995] Beitz, A. y Bearman, M. (1995). An ODP Trading Service for DCE. En *First International IEEE Workshop on Services in Distributed and Networked Environment*, páginas 34–41.

[Bellwood et al., 2002] Bellwood, T., Clément, L., Ehnebuske, D., Hately, A., Hondo, M., Husband, Y. L., Januszewski, K., Lee, S., McKee, B., Munter, J., y von Riegen, C. (2002). UDDI Version 3.0. Publicación de especificación. `http://www.uddi.org/pubs/uddi-v3.00-published-20020719.pdf`.

[Berners-Lee et al., 1998] Berners-Lee, T., Fielding, R., y Masinter, L. (1998). Uniform Resource Identifiers (URI): Generic Syntax. Informe Técnico número RFC 2396, IETF. `http://www.ietf.org/rfc/rfc2396.txt`.

[Bertoa y Vallecillo, 2002] Bertoa, M. y Vallecillo, A. (2002). Quality Attributes for COTS Components. En *Proceeding of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002)*, Málaga, Spain.

[Binns et al., 1995] Binns, P., Engelhart, M., Jackson, M., y Vestal, S. (1995). Domain-Specific Software Architectures for Guidance, Navigation, and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227.

[Bjoerner, 1987] Bjoerner, D. (1987). On the Use of Formal Methods in Software Development. En *9th International Conference on Software Engineering*, páginas 17–29. IEEE Computer Society Press.

[Boehm, 2000] Boehm, B. (2000). Requirements than handle IKIWISI, COTS, and Rapid Change. *IEEE Computer*, 33(7):99–102.

[Boehm, 1988] Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72.

[Booch et al., 1999] Booch, G., Rumbaugh, J., y Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

[Bosch, 2000] Bosch, J. (2000). *Design & Use of Software Architectures*. Addison Wesley.

[Botella et al., 2002] Botella, P., Burgués, X., Carvallo, J. P., Franch, X., y Quer, C. (2002). Using Quality Models for Assessing COTS Selection. En *Workshop on Requirements Engineering (WER2002)*, Valencia, Spain.

[Botella et al., 2001] Botella, P., Burgués, X., Franch, X., Huerta, M., y Salazar, G. (2001). Modeling Non-Functional Requirements. En *Actas de las Jornadas de Ingeniería de Requisitos Aplicada*, Sevilla.

[Boubez et al., 2002a] Boubez, T., Hondo, M., Kurt, C., Rodriguez, J., y Rogers, D. (2002a). UDDI Data Structure Reference V1.0. Publicación de especificación. `http://www.uddi.org/pubs/DataStructure-V1.00-Published-20020628.pdf`.

[Boubez et al., 2002b] Boubez, T., Hondo, M., Kurt, C., Rodriguez, J., y Rogers, D. (2002b). UDDI Programmer's API 1.0. Publicación de especificación. `http://www.uddi.org/pubs/ProgrammersAPI-V1.01-Published-20020628.pdf`.

[Brooks, 1987] Brooks, F. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19.

[Brown, 1999] Brown, A. (1999). *Constructing Superior Software*, capítulo Building Systems from Pieces: Principles and Practice of Component-based Software Engineering. Macmillan Technical Publishing. ISBN: 15-787-01473.

[Brown y Wallnau, 1998] Brown, A. W. y Wallnau, K. C. (1998). The Current State of CBSE. *IEEE Software*, 15(5):37–46.

[Broy et al., 1998] Broy, M., Deimel, A., Henn, J., Koskimies, K., Plásil, F., Pomberger, G., Pree, W., Stal, M., y Szyperski, C. (1998). What characterizes a (software) component? *Software – Concepts and Tools*, (19):49–56.

[Burger, 1995] Burger, C. (1995). Cooperation policies for traders. En *The Third IFIP Conference on Open Distributed Processing*, páginas 191–201.

[Burgués et al., 2002] Burgués, X., Estay, C., Franch, X., Pastor, J., y Quer, C. (2002). Combined Selection of COTS Components. En Dean, J. y Graval, A. (eds.), *First International Conference on COTS-Based Software Systems*, volumen 2255 de *LNCS*, páginas 54–64, Orlando, USA. Springer-Verlag.

[Cameron, 1986] Cameron, J. R. (1986). An Overview of JSD. *IEEE Transactions on Software Engineering*, 12(2):222–240.

[Canal, 2000] Canal, C. (2000). *Un Lenguaje para la Especificación y Validación de Arquitecturas de Software*. Tesis Doctoral, Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga.

[Canal et al., 2001] Canal, C., Fuentes, L., Pimentel, E., Troya, J. M., y Vallecillo, A. (2001). Extending CORBA Interfaces with Protocols. *The Computer Journal*, 44(5):448–462.

[Canal et al., 2003] Canal, C., Fuentes, L., Pimentel, E., Troya, J. M., y Vallecillo, A. (2003). Adding Roles to CORBA Objects. *IEEE Transactions on Software Engineering*, 29(3):242–260.

[Carney, 1999] Carney, D. (1999). Requirements and COTS-Based Systems: A Theory Question Indeed.

[Carney y Long, 2000] Carney, D. y Long, F. (2000). What do you mean by COTS? Finally, a usefull answer. *IEEE Software*, 17(2):83–86.

[Cauldwell et al., 2001] Cauldwell, P., Chawla, R., Chopra, V., Damschen, G., Dix, C., Hong, T., Norton, F., Ogbuji, U., Olander, G., Richman, M. A., Saunders, K., y Zaev, Z. (2001). *Professional XML Web Services*. Wrox Press Ltd. ISBN: 1-861005-09-1.

[Cheesman y Daniels, 2001] Cheesman, J. y Daniels, J. (2001). *UML Components. A Simple Process for Specifying Component-Based Software*. Addison-Wesley. ISBN: 0-201-70851-5.

[Cherki et al., 2001] Cherki, S., Kaim, E., Farcet, N., Salicki, S., y Exertier, D. (2001). Development Support prototype for system families based on COTS. Informe Técnico número TH-WP2-May-01-01, ESAPS Project. `http://www.esi.es/esaps`.

[Cho y Krause, 1998] Cho, I.-H. y Krause, L. (1998). Interoperable Software Modules. En *TOOLS'98*. `http://cyclone.cs.clemson.edu/~ihcho/`.

[Cho et al., 1998] Cho, I.-H., McGregor, J. D., y Krause, L. (1998). A Protocol-based Approach to Specifying Interoperability Between Objects. En *TOOLS'26*, páginas 84–96. `http://www.cs.clemson.edu/~ihcho/protocol/icm.ps`.

[Chung et al., 1999] Chung, L., Nixon, B., Yu, E., y Mylopoulos, J. (1999). *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers. ISBN: 07-923-86663.

[COCOMO-II, 1999] COCOMO-II (1999). Center for Software Engineering (CSE), University of Southern California, USA. `http://sunset.usc.edu/research/COCOMOII/`.

[COCOTS, 1999] COCOTS (1999). Center for Software Engineering (CSE), University of Southern California, USA. `http://sunset.usc.edu/research/COCOTS/`.

[Cook y Daniels, 1994] Cook, S. y Daniels, J. (1994). *Designing Objects Systems*. Prentice Hall. ISBN: 0-13-203860-9.

[Corbin, 1991] Corbin, J. R. (1991). *The Art of Distributed Applications*. Springer-Verlag. ISBN: 35-409-72471.

[Cottman, 1998] Cottman, B. (1998). ComponentWare: Component Software for the Enterprise. *Engineering Institute. SEI Interactive*. `http://www.i-kinetics.com/wp/cwvision/CWvsion.htre`.

[Cuesta, 2002] Cuesta, C. (2002). *Arquitectura de Software Dinámica Basada en Reflexión*. Tesis Doctoral, Universidad de Valladolid.

[Davidrajuh, 2000] Davidrajuh, R. (2000). *Automating Supplier Selection Procedures*. Tesis Doctoral, Narvik Institute of Technology. Norwegian University of Science and Technology.

[Dean y Vigder, 1997] Dean, J. y Vigder, M. (1997). System Implementation Using Commercial Off-The-Shelf (COTS) Software. En *9th Annual Software Technology Conference (STC'97)*, Salt Lake City, Utah, USA. `http://seg.iit.nrc.ca/English/abstracts/NRC40173.html`.

[Dhara y Leavens, 1996] Dhara, K. K. y Leavens, G. T. (1996). Forcing Behavioral Subtyping Through Specification Inheritance. En *18th International Conference on Software Engineering (ICSE-18)*, páginas 258–267, Berlin, Germany. IEEE Press.

[Dodd, 2000] Dodd, J. (2000). Sterling Software Component-Based Development Method. `http://www.sterling.com/cool` y `http://www.ca.com/products`.

[Dong et al., 1999] Dong, J., Alencar, P., y Cowan, D. (1999). A Component Specification Template for COTS-Based Software Development. En *Workshop Ensuring Successfull COTS Development*.

[Dorfman y Flynn, 1984] Dorfman, M. y Flynn, R. F. (1984). ARTS–an Automated Requirements Traceability System. *Journal of Systems and Software*, 4(1):63–74.

[Dürr y Plat, 1994] Dürr, E. y Plat, N. (1994). VDM++ Language Reference Manual. Utrecht, The Netherlands: Cap Volmac.

[D'Souza y Wills, 1999] D'Souza, D. F. y Wills, A. C. (1999). *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley. ISBN: 0-201-31012-0.

[Duke et al., 1995] Duke, R., Rose, G., y Smith, G. (1995). Object-Z: A Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533.

[Durán, 2000] Durán, A. (2000). *Un Entorno Metodológico de Ingeniería de Requisitos para Sistemas de Información*. Tesis Doctoral, Universidad de Sevilla.

[Franch, 1997] Franch, X. (1997). The Convenience for a Notation to Express Non-Functional Characteristics of Software Components. En *Proceeding Foundations of Component-Based Systems Workshop*, páginas 101–109, Zurich, Switzerland.

[Franch, 1998] Franch, X. (1998). Systematic Formulation of Non-Functional Characteristics of Software. En *Proceedings of the 3rd IEEE International Conference on Requirements Engineering*, páginas 174–181, Colorado, USA.

[Franch y Botella, 1998] Franch, X. y Botella, P. (1998). Putting Non-Functional Requirements into Software Architectures. En *Proceedings of the 9th IEEE International Workshop on Software Specification and Design*, páginas 60–67, Ise-shima, Japan.

[Franch y Carvallo, 2002] Franch, X. y Carvallo, J. (2002). A Quality-Model-Based Approach for Describing and Evaluating Software Packages. En *Proceeding of the 10th Joint International Conference on Requirements Engineering (RE'02)*, páginas 104–111, Essen, Germany.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., y Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN: 0-201-63361-2.

[Gao et al., 2000] Gao, J., Eugene, Y., y Shim, S. (2000). Tracking Component-Based Software. En *Continuing Collaboration for Successful COTS Development*, Limerick, Ireland. COTS Workshop in ICSE2000.

[Garlan, 2000] Garlan, D. (2000). Software Architecture: a Roadmap. *The Future of Software Engineering, ACM Press*, páginas 91–101.

[Garlan, 2001] Garlan, D. (2001). *Software Architecture*, capítulo Encyclopedia of Software Engineering. John Wiley & Sons, Inc.

[Garlan et al., 1994] Garlan, D., Allen, R., y Ockerbloom, J. (1994). Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, páginas 17–26.

[Garlan et al., 2001] Garlan, D., Cheng, S., y Kompanek, A. (2001). Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Science of Computer of Computer Programming Journal*.

[Garlan et al., 2000] Garlan, D., Monroe, R., y Wile, D. (2000). *Foundations of Component-Based Systems*, capítulo Acme: Architectural Description of Component-Based Systems, páginas 47–68. Cambridge University Press.

[Goguen et al., 1996] Goguen, J., Nguyen, D., Meseguer, J., Luqi, Zhang, D., y Berzins, V. (1996). Software Component Search. *Journal of Systems Integration*, 6:93–134.

[Goschinski, 1993] Goschinski, A. (1993). Supporting User Autonomy and Object Sharing in Distributed Systems: The RHODOS trading service. En *The First IEEE Symposium on Autonomous Descentralized Systems*.

[Grünbacher y Parets-Llorca, 2001] Grünbacher, P. y Parets-Llorca, J. (2001). A Framework for the Elicitation, Evolution, and Traceability of System Requirements. En *Actas de las Jornadas de Ingeniería de Requisitos Aplicada*, páginas 53–66, Sevilla.

[Han, 1998] Han, J. (1998). Characterization of Components. En *International Workshop on Component-Based Software Engineering (CBSE'98)*, páginas 39–42, Kyoto, Japan.

[Han, 1999] Han, J. (1999). Semantic and Usage Packaging for Software Components. En Vallecillo, A., Hernández, J., y Troya, J. M. (eds.), *Object Interoperability. ECOOP'99 Workshop on Object Interoperability*, páginas 25–34, Lisbon, Portugal.

[Han, 2000] Han, J. (2000). Temporal Logic Based Specifications of Component Interaction Protocols. En Vallecillo, A., Hernández, J., y Troya, J. M. (eds.), *ECOOP'2000 Workshop on Object Interoperability (WOI'00)*, páginas 43–52.

[Heineman y Councill, 2001] Heineman, G. T. y Councill, W. T. (2001). *Component-Based Software Engineering. Putting the Pieces Together*. Addison-Wesley. ISBN: 0-201-70485-4.

[Henderson, 1997] Henderson, P. (1997). Formal Models of Process Components. En *Proceedings of the 1st Workshop on Component-Based Systems*, European Software Engineering Conference (ESEC) y el ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). `http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html`.

[Henderson-Sellers et al., 1999] Henderson-Sellers, B., Pradhan, R., Szyperski, C., Taivalsaari, A., y Wills, A. C. (1999). Are Components Objects? En *OOPSLA'99 Panel Discussions*.

[Henninger, 1994] Henninger, S. (1994). Using iterative refinement to find reusable software. *IEEE Software*, páginas 48–59.

[Hofmeister et al., 1999] Hofmeister, C., Nord, R. L., y Soni, D. (1999). Describing Software Architecture with UML. En *First Working IFIP Conference on Software Architecture*, páginas 145–160. Kluwer Academic.

[Holland, 1993] Holland, I. (1993). Specifying Reusable Components using Contracts. En *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volumen 615 de *LNCS*, páginas 287–308, Utrech, The Netherland.

[IBM-WebSphere, 2001] IBM-WebSphere (2001). Large Grained Components. `http://www.ibm.com/software/components`.

[IEEE, 1990] IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. IEEE Std610.12, IEEE Computer Society Press.

[IEEE, 1993] IEEE (1993). IEEE Recommended Practice on the Selection and Evaluation of CASE Tools. IEEE Std. 1209-1992. New York, NY: Institute of Electrical and Electronics Engineers.

[Intalio, 2001] Intalio (2001). The OpenORB web site. Distributed Object Group. `http://dog.intalio.com/trading.html`.

[Iribarne et al., 2001a] Iribarne, L., Troya, J. M., y Vallecillo, A. (2001a). Study of the RM-ODP Trading Function. Informe Técnico número TR-CBSE-01, Departamento de Lenguajes y Computación. Universidad de Almería.

[Iribarne et al., 2001b] Iribarne, L., Troya, J. M., y Vallecillo, A. (2001b). Trading for COTS Components in Open Environments. En *27th Euromicro Conference*, páginas 30–37, Warsaw, Poland. IEEE Computer Society Press.

[Iribarne et al., 2002a] Iribarne, L., Troya, J. M., y Vallecillo, A. (2002a). A Case Study on Building a GTS System Using COTS Components. Informe Técnico número TR-2002-09-01, Departamento de Lenguajes y Computación, Universidad de Almería.

[Iribarne et al., 2002b] Iribarne, L., Troya, J. M., y Vallecillo, A. (2002b). Selecting Software Components with Multiple Interfaces. En *28th Euromicro Conference*, páginas 27–33, Dortmund, Germany. IEEE Computer Society Press.

[Iribarne et al., 2003a] Iribarne, L., Troya, J. M., y Vallecillo, A. (2003a). A Trading Service for COTS Components. *The Computer Journal*. Enviado para revisión.

[Iribarne et al., 2003b] Iribarne, L., Troya, J. M., y Vallecillo, A. (2003b). Describing Specifications and Architectural Requirements of COTS Components. En Lau, K.-K. (ed.), pendiente de su aparición, The Book Series on Component-based Software Development. World Scientific.

[Iribarne et al., 2003c] Iribarne, L., Troya, J. M., y Vellecillo, A. (2003c). *The Development of Component-Based Information Systems*, capítulo Trading for COTS Components to Fulfil Architectural Requirements. Enviado para revisión.

[Iribarne y Vallecillo, 2000a] Iribarne, L. y Vallecillo, A. (2000a). Searching and Matching Software Components with Multiple Interfaces. En *CBD Workshop at TOOLS Europe'2000. France*. `http://apolo.lcc.uma.es/~av`.

[Iribarne y Vallecillo, 2000b] Iribarne, L. y Vallecillo, A. (2000b). Sobre la búsqueda y emparejamiento de componentes COTS con múltiples interfaces. En *JISBD'2000*. Jornadas de Ingeniería del Software y Bases de Datos. `http://www.ual.es/~liribarn`.

[Iribarne y Vallecillo, 2000c] Iribarne, L. y Vallecillo, A. (2000c). Una metodología para el desarrollo de software basado en COTS. En *1er taller de Ingenieria del Software basada en Componentes Distribuidos IScDIS00. Universidad de Extremadura*.

[Iribarne y Vallecillo, 2002] Iribarne, L. y Vallecillo, A. (2002). Construcción de aplicaciones software a partir de componentes COTS. En *Actas del JISBD'02: VII Jornadas de Ingeniería del Software y Bases de Datos*, El Escorial (Madrid).

[Iribarne et al., 2001c] Iribarne, L., Vallecillo, A., Alves, C., y Castro, J. (2001c). A Non-Functional Approach for COTS Components Trading. En *Workshop on Requirements Engineering (WER2001)*, páginas 124–138, Buenos Aires, Argentina.

[ISO/IEC-9126, 1991] ISO/IEC-9126 (1991). Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use. International Standard ISO/IEC 9126, Geneve, Switzerland: International Standards Organization/International Electrochemical Commision.

[ISO/IEC-ITU/T, 1997] ISO/IEC-ITU/T (1997). Information Technology – Open Distributed Processing – Trading function: Specification. ISO/IEC 13235-1, UIT-T X.950.

[Jacobson et al., 1999] Jacobson, I., Booch, G., y Rumbaugh, J. (1999). *The Unified Development Process*. Addison-Wesley. ISBN: 0-201-57169-2.

[Jacobson et al., 1992] Jacobson, I., Christerson, M., Jonsson, P., y ÖVergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Longman. ISBN: 0-201-54435-0.

[Jewell y Chappell, 2002] Jewell, T. y Chappell, D. (2002). *Java Web Services*. O'Reilly. ISBN: 0-596-00269-6.

[Jordon y Davis, 1991] Jordon, K. y Davis, A. (1991). Requirements Engineering Metamodel: An Integrated View of Requirements. En *Fifteenth Annual International Computer Software and Applications Conference*, páginas 472–478. IEEE Computer Society Press.

[Kerry, 1991] Kerry (1991). Go-Between: a Prototype Trader. Centre of Expertise in Distributed Information Systems.

[Kiely, 1998] Kiely, D. (1998). Are Components the Future of Software? *Computer*, 32(2):10–11.

[Konstantas, 1995] Konstantas, D. (1995). Interoperation of Object Oriented Applications. En Nierstrasz, O. y Tsichritzis, D. (eds.), *Object-Oriented Software Composition*, páginas 69–95. Prentice-Hall.

[Kontio et al., 1996] Kontio, J., Caldiera, G., y Basili, V. (1996). Defining Factors, Goals and Criteria for Reusable Component Evaluation. En *Proceedings of CASCON'96 Conference*, páginas 17–28, Toronto, Canada.

[Krieger, 1998] Krieger, D. (1998). The Emergence of Distributed Component Platforms. *IEEE*, páginas 43–53.

[Kruchten, 1995] Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50.

[Kutvonen, 1995] Kutvonen, L. (1995). Achieving Interoperability through ODP Trading function. En *Second International Symposium on Autonomous Decentralized systems (ISADS'95)*, páginas 63–69, Arizona. IEEE Computer Society.

[Kutvonen, 1996] Kutvonen, L. (1996). Overview of the DRYAD Trading System Implementation. En *The IFIP/IEEE International Conference on Distributed Platforms*, páginas 314–326. Chapman and Hall.

[Lano et al., 1997] Lano, K., Bicarregui, J., Maibaum, T., y Fiadeiro, J. (1997). Composition of Reactive Systems Components. En *Proceedings of the 1st Workshop on Component-Based Systems*, European Software Engineering Conference (ESEC) y el ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). `http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html`.

[Lawlis et al., 2001] Lawlis, P. K., Mark, K. E., Thomas, D. A., y Courtheyn, T. (2001). A Formal Process for Evaluating COTS Software Products. *IEEE Computer*, 30(5):58–63.

[Lea y Marlowe, 1995] Lea, D. y Marlowe, J. (1995). Interface-Based Protocol Specification of Open Systems Using PSL. En *Proc. of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, páginas 374–398, Aarhus, Dänemark.

[Leavens et al., 1999] Leavens, G. T., Baker, L., y Ruby, C. (1999). *Behavioral Specifications of Businesses and Systems*, capítulo JML: A Notation for Detail Desing. Kluwer Academic. `http://www.cs.iastate.edu/~leavens/JML.html`.

[Leavens y Sitaraman, 2000] Leavens, G. T. y Sitaraman, M. (2000). *Foundations of Component-Based Systems*. Cambridge University Press. ISBN: 0-521-77164-1.

[Leffingwell, 2001] Leffingwell, D. (2001). Features, Use Cases and Requirements. En *The Rational Edge*. Rational. `http://www.therationaledge.com/splashpage_dec.html`.

[Luckham et al., 1995] Luckham, D., Augustin, L., Kenny, J., Veera, J., Bryan, D., y Mann, W. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355.

[Lumpe et al., 1997] Lumpe, M., Schneider, J., Nierstrasz, O., y Achermann, F. (1997). Towards a Formal Composition Language. En *Proceedings of the 1st Workshop on Component-Based Systems*, European Software Engineering Conference (ESEC) y el ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). `http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html`.

[Lynch, 1996] Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann. ISBN: 1-55860-348-4.

[Magee y Kramer, 1996] Magee, J. y Kramer, J. (1996). Dynamic Structure in Software Architectures. En *ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering*, páginas 3–14.

[Maiden y Ncube, 1998] Maiden, N. A. y Ncube, C. (1998). Acquiring COTS Software Selection Requirements. *IEEE Software*, 15(2):46–56.

[Maiden et al., 1997] Maiden, N. A., Ncube, C., y Moore, A. (1997). Lessons Learned During Requirements Acquisition for COTS Systems. *Communications of the ACM*, 40(12):21–25.

[Maier et al., 2001] Maier, M. W., Emery, D., y Hilliard, R. (2001). Software Architecture: Introducing IEEE Standard 1471. *IEEE Computer*, 34(4):107–109.

[Matsumoto, 1987] Matsumoto, Y. (1987). A software factory: An overall approach for software production. En Freeman, P. (ed.), *Tutorial on Software Reusability*, páginas 155–178.

[Medvidovic et al., 1996] Medvidovic, N., Oreizy, P., Robbins, J., y Taylor, R. (1996). Using object-oriented typing to support architectural design in the C2 style. En *ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering*, páginas 24–32.

[Medvidovic et al., 2002] Medvidovic, N., Rosenblum, D. S., Robbins, J. E., y Redmiles, D. F. (2002). Modeling Software Architectures in the Unified Modeling Lenguage. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57.

[Medvidovic y Taylor, 2000] Medvidovic, N. y Taylor, R.Ñ. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.

[Merz et al., 1994] Merz, M., Muller, K., y Lamersdorf, W. (1994). Service Trading and Mediation in Distributed Computing Systems. En *14th International Conference on Distributed Computing Systems*, páginas 450–457. IEEE Computer Society Press.

[Meyer, 1992] Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall. ISBN: 0-13-247925-7.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction. 2nd Ed.* Series on Computer Science. Prentice Hall.

[Meyer, 1999] Meyer, B. (1999). The Significance of Components. *Software Development*, 7(11):56–57.

[Meyer et al., 1998] Meyer, B., Mingins, C., y Schmidt, H. (1998). Trusted Components for the Software Industry. *Computer*, 31(4). `http://www.trusted-components.org/`.

[Meyers y Oberndorf, 2001] Meyers, B. C. y Oberndorf, P. (2001). *Managing Software Acquisition: Open Systems and COTS Products*. The SEI Series in Software Engineering. Addison-Wesley. ISBN: 0-201-70454-4.

[Mikhajlova, 1999] Mikhajlova, A. (1999). *Ensuring Correctness of Object and Component Systems*. Tesis Doctoral, Åbo Akademi University.

[Mili et al., 1995] Mili, H., Mili, F., y Mili, A. (1995). Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562.

[Mili et al., 2000] Mili, R., Desharnais, J., Frappier, M., y Mili, A. (2000). Semantic Distance Between Specifications. *Theorical Computer Science*, 247(1–2):257–276.

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency*. Prentice Hall. ISBN: 0-131-15007-3.

[Müller-Jones et al., 1995] Müller-Jones, K., Merz, M., y Lamersdorf, W. (1995). The TRADEr: Integrating trading into DCE. En *The 3rd IFIP Conference on Open Ditributed Processing*, páginas 459–470.

[Monge et al., 2002] Monge, R., Alves, C. F., y Vallecillo, A. (2002). A Graphical Representation of COTS-based Software Architectures. En *Proc. of the Fifth Iberoamerican Conference on Requirements Engineering and Software Environments (IDEAS'02)*, páginas 126–137, La Habana, Cuba.

[Moriconi et al., 1995] Moriconi, M., Qian, X., y Riemensschneider, R. (1995). Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372.

[Ncube y Maiden, 2000] Ncube, C. y Maiden, N. (2000). COTS Software Selection: The Need to make Tradeoffs between System Requirements, Architectures and COTS Components. En *Continuing Collaborations for Successful COTS Development*, Limerick, Ireland. COTS Workshop in ICSE2000.

[Nierstrasz, 1995] Nierstrasz, O. (1995). Regular Types for Active Objects. En Nierstrasz, O. y Tsichritzis, D. (eds.), *Object-Oriented Software Composition*, páginas 99–121. Prentice-Hall.

[Nuseibeh, 2001] Nuseibeh, B. (2001). Weaving Together Requirements and Architectures. *IEEE Computer*, 34(3):115–117.

[Oberndorf, 1997] Oberndorf, P. (1997). COTS and Open Systems – An Overview. `http://www.sei.cmu.edu/activities/str/descriptions/cots_body.html`.

[Ochs et al., 2000a] Ochs, M., Pfahl, D., Chrobok-Diening, G., y Nothhelfer-Kolb, B. (2000a). A COTS Acquisition Process: Definition and Application Experience. En *Proceedings of the 11th ESCOM Conference*, páginas 335–343, Shaker, Maastricht.

[Ochs et al., 2000b] Ochs, M., Pfahl, D., Chrobok-Diening, G., y Nothhelfer-Kolb, B. (2000b). A Method for Efficient Mesurement-based COTS Assessment and Selection. En *Proceedings IEEE 7th International Software Metrics Symposium*, páginas 285–296, London, England.

[OIM, 1995] OIM (1995). Metadata Corporation Open Information Model (OIM). `http://msdn.microsoft.com/repository/oim` y `http://www.mdcinfo.com/OIM/documents.html`.

[OMG, 1999] OMG (1999). *The CORBA Component Model*. Object Management Group. `http://www.omg.org`.

[OMG, 2000] OMG (2000). Trading Object Service Specification. Versión 1.0. `http://www.omg.org/`.

[OMG, 2001] OMG (2001). *A UML Profile for Enterprise Distributed Object Computing V1.0*. Object Management Group. OMG document `ad/2001-08-19`.

[OOC, 2001] OOC (2001). ORBacus Trader. ORBacus for C++ and Java. `http://www.ooc.com/ob`.

[OSF, 1994] OSF (1994). *OSF DCE Application Development Guide*. Cambridge, MA. `http://www.opengroup.org`.

[Ostertag et al., 1992] Ostertag, E., Hendler, J., Prieto-Diaz, R., y Braun, C. (1992). Computing similarity in a reuse library system. *ACM Transaction on Software Engineering and Methodology*, páginas 205–228.

[Popescu-Zeletin et al., 1991] Popescu-Zeletin, R., Tschammer, V., y Tschichholz, M. (1991). 'Y' Distributed Application Platform. En *Computer Communication*, número 6, páginas 366–375.

[Poston y Sexton, 1992] Poston, R. y Sexton, M. (1992). Evaluating and Selecting Testing Tools. *IEEE Software*, 9(3):29–35.

[Prieto-Diaz, 1991] Prieto-Diaz, R. (1991). Implementing faceted classification for software reuse. *Communication of the ACM*, páginas 89–97.

[Prieto-Diaz y Neighbors, 1986] Prieto-Diaz, R. y Neighbors, J. (1986). Module Interconnection Languages. *The Journal of Systems and Software*, 6(4):307–334.

[PrismTech, 2001] PrismTech (2001). Trading Service – White Paper. PrismTech OpenFusion, Enterprise Integration Services. `http://www.prismtechnologies.com`.

[Puder et al., 1995] Puder, A., Markwitz, S., Gudermann, F., y Geihs, K. (1995). AI-Based Trading in Open Distributed Environments. University of Frankfurt.

[Purtilo, 1994] Purtilo, J. (1994). The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174.

[Raynal, 1988] Raynal, M. (1988). *Distributed Algorithms and Protocols*. John Wiley & Sons, Inc.

[Reilly, 1990] Reilly, J. R. (1990). Entity-Relationship Approach to Data Modeling. En Thayer, R. y Dorfman, M. (eds.), *System and Software Requirements Engineering*. IEEE Computer Society Press.

[Riemenschneider y Stavridou, 1999] Riemenschneider, R. y Stavridou, V. (1999). The Role of Architecture Description Languages in Component-Based Development: The SRI Perspective. `http://www.sei.cmu.edu/cbs/icse99/papers/42/42.htm`.

[Robertson y Robertson, 1999] Robertson, S. y Robertson, J. (1999). *Mastering the Requirements Process*. Addison-Wesley. ISBN: 0-201-36046-2.

[Rolland, 1999] Rolland, C. (1999). Requirements Engineering for COTS Based Systems. *Elsevier, Information and Software Technology*, 41(14):985–990.

[Rollins y Wing, 1991] Rollins, E. y Wing, J. (1991). Specifications as search keys for software libreries. En *Proceedings of the Eighth International Conference on Logic Programming*.

[Rosa et al., 2001] Rosa, N. S., Alves, C. F., Cunha, P. R. F., y Castro, J. F. B. (2001). Using Non-Functional Requirements to Select Components: A Formal Approach. En *Fourth Iberoamerican on Requirements Engineering and Software Environments (IDEAS'2001). San José, Costa Rica.*

[Ross, 1977] Ross, D. T. (1977). Structured Analysis (SA): A Language for Communicating Ideas. *IEEE Transactions on Software Engineering*, 3(1):16–33.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., y Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley. ISBN: 0-201-30998-X.

[Rumpe et al., 1999] Rumpe, B., Schoenmakers, M., Radermacher, A., y Schürr, A. (1999). UML + ROOM as a Standard ADL? En Titsworth, F. (ed.), *Engineering of Complex Computer Systems, ICECCS'99*. IEEE Computer Society Press. `http://www4.in.tum.de/~rumpe/papers/RRSS99/RRSS99.html`.

[Saiedian y Dale, 2000] Saiedian, H. y Dale, R. (2000). Requirements Engineering: Making the Connection between the Software Developer and Customer. *Information and Software Technology. Elsevier*, 42:419–428.

[Sayani, 1990] Sayani, H. (1990). PSL/PSA at the Age of Fifteen: Tool for Real-Time and Non-Real Time Analysis. En Thayer, R. y Dorfman, M. (eds.), *System and Software Requirements Engineering*, páginas 403–417. IEEE Computer Society Press.

[Seacord et al., 2001] Seacord, R. C., Mundie, D., y Boonsiri, S. (2001). K-BACEE: Knowledge-Based Automated Component Ensemble Evaluation. En *27th Euromicro Conference*, páginas 56–62, Warsaw, Poland. IEEE Computer Society Press.

[Sedigh-Ali et al., 2001] Sedigh-Ali, S., Ghafoor, A., y Paul, R. A. (2001). Software Engineering Metrics for COTS-Based Systems. *IEEE Computer*, páginas 44–50.

[Selic, 1999] Selic, B. (1999). UML-RT: A Profile for Modeling Complex Real-Time Architectures.

[Selic, 2001] Selic, B. (2001). On Modeling Architectural Structures with UML. En *UML'2001 Workshop on Software Architectures and Requirements Engineering (STRAW'01)*, Toronto, Canada. `http://www.rational.com/events/ICSE2001/ICSEwkshp/SELIC_Modeling_Archit%ecture.pdf`.

[Selic et al., 1994] Selic, B., Gullekson, G., y Ward, P. (1994). *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc. ISBN: 0-471-59917-4.

[Selic y Rumbaugh, 1998] Selic, B. y Rumbaugh, J. (1998). Using UML for modeling complex real-time systems. `http://www.rational.com/media/whitepapers/umlrt.pdf`.

[Shaw, 1996] Shaw, M. (1996). Truth vs. Knowledge: The Difference Between What a Component Does and What We Know it Does. En *Eighth International Workshop on Software Specification and Design*, páginas 181–185, Paderborn, Germany. IEEE Computer Society Press.

[Shaw et al., 1995] Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., y Zelesnik, G. (1995). Abstractions for Software Architecture and Tools to Support them. *IEEE Transactions on Software Engineering*, 21(4):314–335.

[Shaw y Garlan, 1996] Shaw, M. y Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. ISBN: 0-13-182957-2.

[Shmidt, 2001] Shmidt, D. C. (2001). The Adaptative Communication Environment (TAO): AceORB. University of California. `http://www.theaceorb.com`.

[Sodhi y Sodhi, 1999] Sodhi, J. y Sodhi, P. (1999). *Software Reuse. Domain Analysis and Design Process*. McGraw-Hill. ISBN: 0-07-057923-7.

[Spivey, 1992] Spivey, J. (1992). *The Z Notation. A Reference Manual. 2nd Ed.* Prentice Hall. ISBN: 13-978529-9.

[Stafford et al., 1998] Stafford, J., Richardson, D., y Wolf, A. (1998). Aladdin: A Tool for Architecture-Level Dependence Analysis of Software. Informe Técnico número CU-CS-858-98, University of Colorado.

[Svodoba, 1990] Svodoba, C. (1990). Tutorial on Structured Analysis. En Thayer, R. y Dorfman, M. (eds.), *System and Software Requirements Engineering*. IEEE Computer Society Press.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley. ISBN: 0-201-17888-5.

[Thayer y Dorfman, 1999] Thayer, R. y Dorfman, M. (1999). *Software Requirements Engineering*. IEEE Computer Society Press. ISBN: 0-8186-7738-4.

[Thompson, 1998] Thompson, C. (1998). Workshop on Compositional Software Architectures: Workshop Report. `http://www.objs.com/workshops/ws9801/report.html`.

[Tran y Liu, 1997] Tran, V. y Liu, T. (1997). A Procurement-Centric Model for Engineering Component-Based Software Systems. En *Fifth International Symposium on Assesment of Software Tools*.

[Valetto y Kaiser, 1995] Valetto, G. y Kaiser, G. (1995). Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments. En *7th IEEE International Workshop on CASE*, páginas 40–48. IEEE Computer Society Press.

[Vallecillo et al., 1999] Vallecillo, A., Hernández, J., y Troya, J. M. (1999). Object Interoperability. En *Object-Oriented Technology: ECOOP'99 Workshop Reader*, número 1743 de LNCS, páginas 1–21. Springer-Verlag.

[Vallecillo et al., 2000] Vallecillo, A., Hernández, J., y Troya, J. M. (2000). New Issues in Object Interoperability. En *Object-Oriented Technology: ECOOP'2000 Workshop Reader*, número 1964 de LNCS. Springer-Verlag.

[Vasudevan y Bannon, 1999] Vasudevan, V. y Bannon, T. (1999). WebTrader: Discovery and Programmed Access to Web-Based Services. En *Poster at the 8th International WWW Conference (WWW8)*, Toronto, Canada. `http://www.objs.com/agility/tech-reports/9812-web-trader-paper/WebTrade%rPaper.html`.

[Vigder, 1998] Vigder, M. (1998). An Architecture for COTS based Software Systems. Informe Técnico número NRC41603, National Research Council of Canada (NRC). `http://seg.iit.nrc.ca/papers/NRC41603.pdf`.

[Voas, 1998] Voas, J. (1998). Certifying Off-The-Shelf Software Components. *IEEE Computer*, 31(6):16–19.

[Voas, 2001] Voas, J. (2001). Faster, better, cheaper. *IEEE Software*, páginas 96–97.

[W3C-WebServices, 2002] W3C-WebServices (2002). Web Services Glosary. W3C Working Draft. `http://www.w3.org/TR/2002/WD-ws-gloss-20021114/`.

[Wallnau et al., 2002] Wallnau, K. C., Hissam, S. A., y Seacord, R. C. (2002). *Building Systems from Commercial Components*. The SEI Series in Software Engineering. Addison-Wesley. ISBN: 0-201-70064-6.

[Ward y Mellor, 1985] Ward, P. T. y Mellor, S. J. (1985). *Structured Development Techniques for Real-Time Systems*. Prentice-Hall: Englewood Cliffs, NJ, 3 edición.

[Warmer y Kleppe, 1998] Warmer, J. y Kleppe, A. (1998). *The Object Constraint Language — Precise Modeling with UML*. Addison-Wesley. ISBN: 0-201-37940-6.

[Weck y Büchi, 1998] Weck, W. y Büchi, M. (1998). Compound Types: Strong Typing for Architecture Composition. En *First Nordic Software Architecture Workshop*. `http://www.abo.fi/~mbuechi/publications/CTforAC.html`.

[Yellin y Strom, 1997] Yellin, D. M. y Strom, R. E. (1997). Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333.

[Zaremski y Wing, 1995] Zaremski, A. M. y Wing, J. M. (1995). Signature Matching: A Tool for Using Software Libraries. *ACM Trans. on Software Engineering and Methodology*, 4(2):146–170.

[Zaremski y Wing, 1997] Zaremski, A. M. y Wing, J. M. (1997). Specification Matching of Software Components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369.

Este documento ha sido generado
con la versión 3.14 de LaTeX

COTSTRADER.COM es un sitio web registrado
en el gestor de nombres NOMINALIA.COM
y ha sido creado con fines totalmente
de investigación y sin ánimo de lucro.

Todas las figuras y tablas contenidas en
el presente documento son originales.

Un Modelo de Mediación para el Desarrollo de Software basado en Componentes COTS

Luis F. Iribarne Martínez
Departamento de Lenguajes y Computación
Universidad de Almería
Almería, 14 de Julio de 2003
http://www.ual.es/~liribarn