Lección 1
Desarrollo de Software Basado en Componentes

Lidia Fuentes, José M. Troya y Antonio Vallecillo
Dept. Lenguajes y Ciencias de la Computación. Universidad de Málaga.
ETSI Informática. Campus Teatinos, s/n. 29071 Málaga, Spain.
{lff,troya,av}@lcc.uma.es

Resumen

Esta lección presenta los modelos, conceptos y mecanismos fundamentales sobre los que se apoya actualmente el desarrollo de aplicaciones software basado en componentes reutilizables. En primer lugar, las arquitecturas software y los marcos de trabajo intentan ofrecer soluciones de diseño desde el punto de vista estructural de las aplicaciones, y de las relaciones entre sus componentes. A otro nivel se encuentra la programación orientada a componentes, un paradigma que propugna la construcción de componentes reutilizables en entornos abiertos y distribuidos, con el objetivo de lograr un mercado global de software. Basados en ellos, los modelos y plataformas de componentes proporcionan los mecanismos adecuados para tratar la complejidad de los problemas que aparecen en los sistemas abiertos y distribuidos. Finalmente, se describen las dificultades que encuentran las metodologías tradicionales para construir aplicaciones en estos nuevos ambientes, y los retos a los que se enfrenta la Ingeniería del Software para poder hablar realmente de "Ingeniería del Software Basada en Componentes".

1 Introducción

Los continuos avances en la Informática y las Telecomunicaciones están haciendo cambiar la forma en la que se desarrollan actualmente las aplicaciones software. En particular, el incesante aumento de la potencia de los ordenadores personales, el abaratamiento de los costes del hardware y las comunicaciones, y la aparición de redes de datos de cobertura global han disparado el uso de los sistemas abiertos y distribuidos. Esto ha provocado, entre otras cosas, que los modelos de programación existentes se vean desbordados, siendo incapaces de manejar de forma natural la complejidad de los requisitos que se les exigen para ese tipo de sistemas. Comienzan a aparecer por tanto nuevos paradigmas de programación, como pueden ser la coordinación, la programación orientada a componentes, o la movilidad, que persiguen una mejora en los procesos de construcción de aplicaciones software. En ellos se trabaja tanto en extensiones de los modelos existentes como en nuevos modelos, en la estandarización de sus interfaces y servicios, y la pertinaz búsqueda del cada vez más necesario mercado global de componentes software. Estos son parte de los nuevos retos con los que se enfrenta actualmente la ingeniería del software.

Uno de los enfoques en los que actualmente se trabaja constituye lo que se conoce como *Desarrollo de Software Basado en Componentes* (DSBC), que trata de sentar las bases para el diseño y desarrollo de aplicaciones distribuidas basadas en componentes software reutilizables. Dicha disciplina cuenta actualmente con un creciente interés, tanto desde el punto de vista académico como desde el industrial, en donde la demanda de estos temas es cada día mayor.

La presente lección pretende servir como una breve introducción a algunos de los conceptos y métodos fundamentales sobre los que se apoya el DSBC. En particular, nos centraremos en las arquitecturas software y los marcos de trabajo, la programación orientada a componentes, y en las plataformas de componentes distribuidas. Asimismo, discutiremos sobre lo que debería constituir una metodología para el DSBC. Por supuesto, aún queda mucho trabajo por hacer para poder hablar realmente de una *Ingeniería del Software Basada en Componentes*, pero sin duda las bases se están sentando para hacer de esta disciplina una realidad en un futuro cercano.

2 Conceptos Básicos

Comenzaremos con algunas definiciones básicas, que van a sentar las bases de los conceptos que manejaremos a lo largo de esta lección.

En primer lugar, entendemos por *sistema* a un conjunto de mecanismos y herramientas que permiten la creación e interconexión de componentes software, junto con una colección de servicios para facilitar las labores de los componentes que residen y se ejecutan en él.

Un sistema se denomina *independientemente extensible* [Szyperski, 1996] si puede ser dinámicamente extendido, y en donde pueden combinarse extensiones independientemente desarrolladas por distintas partes o entidades, sin conocimiento unas de otras.

Diremos que un sistema es *abierto* si es concurrente, reactivo, independientemente extensible, y permite a componentes heterogéneos ingresar o abandonar el sistema de forma dinámica. Estas condiciones implican que los sistemas abiertos son inherentemente evolutivos, y que la vida de sus componentes es más corta que la del propio sistema. No hemos querido incluir en nuestra definición de sistema abierto la propiedad de ser distribuido, puesto que consideramos que ambas características son independientes entre sí. Sin embargo, los sistemas objeto de nuestro estudio comprenden ambas propiedades, siendo tanto abiertos como distribuidos.

Como consecuencia de dichas características, el desarrollo de aplicaciones para este tipo de sistemas se ve afectado por una serie de problemas específicos, como son la gestión de la evolución del propio sistema y de sus componentes, la falta de una visión global del sistema, la dificultad para garantizar la seguridad y confidencialidad de los mensajes, la heterogeneidad de los componentes, o su dispersión, lo que puede implicar retrasos y errores en las comunicaciones.

El tratamiento de estas situaciones es lo que ha hecho ver la necesidad de nuevos modelos, pues la programación tradicional se ha visto incapaz de tratarlos de una forma natural. Así, la Programación Orientada a Objetos (POO) ha sido el sustento de la ingeniería del software para los sistemas cerrados. Sin embargo, se ha mostrado insuficiente al tratar de aplicar sus técnicas para el desarrollo de aplicaciones en entornos abiertos. En particular, se ha observado que no permite expresar claramente la distinción entre los aspectos computacionales y meramente composicionales de la aplicación, y que hace prevalecer la visión de objeto sobre la de componente, estos últimos como unidades de composición independientes de las aplicaciones. Asimismo, tampoco tiene en cuenta los factores de mercadotecnia necesarios en un mundo real, como la distribución, adquisición e incorporación de componentes a los sistemas.

A partir de estas ideas nace la *programación orientada a componentes* (POC) como una extensión natural de la orientación a objetos para los entornos abiertos [Nierstrasz, 1995][Szyperski y Pfister, 1997]. Este paradigma propugna el desarrollo y utilización de componentes reutilizables dentro de lo que sería un mercado global de software.

Sin embargo, disponer de componentes no es suficiente tampoco, a menos que seamos capaces de reutilizarlos. Y reutilizar un componente no significa usarlo más de una vez, sino que implica la capacidad del componente de ser utilizado en contextos distintos a aquellos para los que fue diseñado. En este sentido, uno de los sueños que siempre ha tenido la ingeniería del software es el de contar con un mercado global componentes, al igual que ocurre con otras ingenierías, o incluso con el hardware. De hecho, la analogía con los circuitos integrados (*IC*) llegó a poner de moda los términos *software IC*, *software bus* y *software backplane*, aunque nunca se haya podido llegar más allá de la definición de estos conceptos.

Para hablar de la existencia de un mercado de componentes software es necesario que los componentes estén empaquetados de forma que permitan su distribución y composición con otros componentes, especialmente con aquellos desarrollados por terceras partes. Esto nos lleva a la definición de componente:

> "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" [Szyperski, 1998].

Las interfaces de un componente determinan tanto las operaciones que el componente implementa como las que precisa utilizar de otros componentes durante su ejecución. Y los requisitos determinan las necesidades del componente en cuanto a recursos, como por ejemplo las plataformas de ejecución que necesita para funcionar. Obsérvese que los conceptos de objeto y componente son ortogonales, y que ninguno de los dos depende del otro.

Finalmente, a partir de los componentes reutilizables dentro de un mercado global de software nace el concepto fundamental en estos entornos, el de componente COTS (*commercial off-the-shelf*). Este concepto va a cambiar la idea tradicional del desarrollo de software en donde todo es propietario —o la reutilización se reduce a un ámbito local (el de la empresa que desarrolla el software)—, hacia nuevas metodologías de desarrollo de software en donde es posible contar con elementos externos. Esto se va a traducir, como veremos en la última sección, no sólo en nuevas formas de desarrollo e implementación de aplicaciones, sino también en alteraciones durante las fases de especificación y diseño para tener en cuenta este tipo de elementos. Aparecen también nuevos problemas, como el de la búsqueda y reconocimiento de los componentes que se necesitan, su posible adaptación, o la resolución de solapamientos entre las funciones y servicios que ofrecen.

Una vez disponemos del concepto de componente, un *modelo de componentes* define la forma de sus interfaces y los mecanismos para interconectarlos entre ellos (p.e. COM, JavaBeans o CORBA). Basada en un modelo de componentes concreto, una *plataforma de componentes* es un entorno de desarrollo y de ejecución de componentes que permite aislar la mayor parte de las dificultades conceptuales y técnicas que conlleva la construcción de aplicaciones basadas en los componentes de ese modelo [Krieger y Adler, 1998]. En este sentido, podemos definir una plataforma como una implementación de los mecanismos del modelo, junto con una serie de herramientas asociadas. Ejemplos de estas plataformas son ActiveX/OLE, Enterprise Beans y Orbix, que se apoyan en los modelos de componentes COM, JavaBeans y CORBA, respectivamente.

Por otro lado, un mercado global necesita también de *estándares* que garanticen la interoperabilidad de los componentes a la hora de construir aplicaciones. En el mundo de los sistemas abiertos y distribuidos estamos presenciando la clásica 'guerra' de estándares que sucede antes de la maduración de cualquier mercado. Todos los fabricantes y vendedores de sistemas tratan de convertir sus mecanismos en estándares, a la vez que pertenecen a varios consorcios que también tratan de definir estándares, pero de forma independiente. Pensemos por ejemplo en Sun, que intenta imponer JavaBeans como modelo de componentes, mientras que participa junto con otras empresas en el desarrollo de los estándares de CORBA. Actualmente la interoperabilidad se resuelve mediante pasarelas (*bridges*) entre unos modelos y otros, componentes especiales que se encargan de servir de conexión entre los distintos componentes heterogéneos. La mayoría de los modelos comerciales ofrecen pasarelas hacia el resto, pues reconocen la necesidad de ser *abiertos*.

3 Arquitecturas Software y Marcos de Trabajo

El disponer de componentes software no es suficiente para desarrollar aplicaciones, ya provengan éstos de un mercado global o sean desarrollados a medida para la aplicación. Un aspecto crítico a la hora de construir sistemas complejos es el diseño de la estructura del sistema, y por ello el estudio de la Arquitectura Software se ha convertido en una disciplina de especial relevancia en la ingeniería del software [Shaw y Garlan, 1996].

> Entendemos por *Arquitectura Software* la representación de alto nivel de la estructura de un sistema o aplicación, que describe las partes que la integran, las interacciones entre ellas, los patrones que supervisan su composición, y las restricciones a la hora de aplicar esos patrones.

En general, dicha representación se va a realizar en términos de una colección de componentes y de las interacciones que tienen lugar entre ellos. De esta forma aparecen las arquitecturas basadas en componentes y conectores: los primeros se dedican a labores computacionales, mientas que los conectores encapsulan los patrones de sincronización y coordinación entre los componentes. Este tipo de arquitecturas son completamente modulares y favorecen la reutilización de todos sus elementos, incluyendo los que definen las distintas relaciones entre ellos.

Una arquitectura software viene determinada por las diferentes instancias de cada tipo de componentes y conectores que la componen, y por una serie de *enlaces* (*bindings*) específicos que definen la unión de todas ellas formando una estructura. A esta estructura se le da el nombre de *configuración*, y suele considerarse insertada en una *jerarquía*, pues toda entidad software, independientemente de su granularidad, dispone de una estructura que puede ser descrita mediante una arquitectura software.

Cuando lo que interesa no es obtener una configuración concreta, sino extraer los patrones genéricos que definen a una familia de sistemas, se habla de *estilos arquitectónicos*. La definición de estilos tiene

una gran importancia desde el punto de vista de la reutilización, y es un aspecto fundamental en la arquitectura del software.

En general, la arquitectura software nace como una herramienta de alto nivel para cubrir distintos objetivos, entre los que destacamos:

1. Comprender y manejar la estructura de las aplicaciones complejas.
2. Reutilizar dicha estructura (o partes de ella) para resolver problemas similares.
3. Planificar la evolución de la aplicación, identificando sus partes mutables e inmutables, así como los costes de los posibles cambios.
4. Analizar la corrección de la aplicación, y su grado de cumplimiento respecto a los requisitos iniciales (prestaciones o fiabilidad).
5. Permitir el estudio de alguna propiedad específica del dominio.

Se han propuesto diversos modelos de arquitecturas software usando componentes y conectores que se basan en la especificación de la arquitectura del sistema. En esta línea podemos citar los trabajos que parten del modelo inicial de Allen y Garlan [Allen y Garlan, 1994], como UNICON [Shaw et al., 1995], AESOP [Garlan et al., 1994] [Garlan et al., 1995], δarwin [Magee et al., 1995][Magee y Kramer, 1996], Executable Connectors [Ducasse y Richner, 1997] y LEDA [Canal et al., 1997]. Todos ellos comparte la misma visión arquitectónica de los sistemas, aunque cada uno de ellos utilice notaciones y formalismos diferentes.

3.1 Lenguajes de Descripción de Arquitecturas

Todas las propuestas mencionadas anteriormente utilizan los *Lenguajes de Descripción de Arquitecturas* (LDAs) para expresar la estructura de las aplicaciones. Los LDAs proporcionan los modelos, notaciones y herramientas que permiten describir los componentes y conectores que forman una aplicación, así como sus enlaces específicos. Los LDAs deben ser capaces de:

1. Gestionar los diseños a alto nivel, adaptarlos a implementaciones específicas, y permitir la selección de patrones o paradigmas de arquitecturas especificados previamente.
2. Representar nuevos patrones de arquitecturas y nuevas formas de interacción entre los componentes, de forma que puedan ser reutilizados en diseños futuros.
3. Aportar un conjuntos de herramientas formales para demostrar propiedades sobre los sistemas expresados en ese lenguaje (seguridad o de viveza, por ejemplo).
4. Aportar otro conjunto de herramientas de desarrollo para realizar implementaciones parciales de los sistemas a partir de su descripción en un LDA concreto.

En cuanto al tipo de arquitectura que permiten describir los distintos LDAs, los hay que sólo permiten las arquitecturas estáticas [Tanir, 1996], en donde las conexiones entre los componentes no varían a lo largo de la vida del sistema; la utilidad de dichos LDA estáticos se pone de manifiesto para la descripción de sistemas hardware o sistemas eléctricos, aunque no se muestran muy adecuados para sistemas abiertos. En este tipo de sistemas es preciso que los LDAs permitan expresar los cambios en las conexiones e interacciones entre los componentes en tiempo de ejecución, pues van a depender, por ejemplo, de los recursos disponibles o del perfil de los usuarios. Los LDAs mencionados antes permiten la descripción de arquitecturas dinámicas, así como Rapide [Luckham et al., 1995].

Por otro lado, hay muchas propuestas a nivel de especificación de arquitecturas software basadas en formalismos cuyos únicos objetivos son el prototipado rápido de aplicaciones y la verificación formal de algunas de sus propiedades, entre los que podemos citar Wright [Allen y Garlan, 1997] y ACME [Garlan et al., 1997]. Aunque las bondades de estas propuestas son innegables, la mayoría de ellas se quedan a este nivel de diseño, no abordando la problemática de la generación de código a partir de las especificaciones. De esta forma dejan una laguna importante en el ciclo de desarrollo de las aplicaciones, perdiéndose la descomposición del sistema en componentes y conectores cuando se llega al nivel de la implementación [Allen y Garlan, 1997]. Conseguir metodologías que cubran todo el ciclo de vida de

las aplicaciones a partir de su arquitectura software es otro de los caminos de investigación abiertos actualmente.

Hasta ahora hemos hablado de LDAs de propósito general que, al menos en teoría, pueden usarse para cualquier dominio de aplicación. Sin embargo, cada dominio tiene sus peculiaridades específicas y de ahí surgen los *Lenguajes para Dominios Específicos* (LDEs), que incorporan conceptos y funciones particulares de un dominio concreto como parte del propio lenguaje. Los LDEs proporcionan grandes ventajas al arquitecto de sistemas, entre ellas su especialización y gran expresividad en su dominio de aplicación, aunque suelen presentar un serio problema de extensibilidad: apenas aparece un nuevo concepto relacionado con el dominio que no estaba previsto en el lenguaje, éste deja de ser útil [Bosch y Dittrich, 1997].

Además de los LDAs y LDEs, existen otras propuestas que, teniendo su mismo objetivo, ofrecen un punto de vista más composicional que arquitectónico para la construcción de sistemas basados en componentes. Nos referimos a los denominados *Lenguajes Composicionales* [Nierstrasz, 1995] [Lumpe et al., 1997]. Un lenguaje composicional es una combinación entre un LDA y un lenguaje de configuración (*scripting language*), que sirven como soporte para un entorno de composición visual. Los diferentes estilos arquitectónicos se definen a niveles de componentes y conectores, y la unión entre ellos (*glue*) se define mediante el lenguaje de configuración. El objetivo es definir, a partir de un lenguaje composicional, metodologías y entornos de desarrollo de aplicaciones software. Aunque ya se conocen las características que deben poseer estos lenguajes composicionales para su tratamiento en entornos arquitectónicos [Nierstrasz, 1995], todavía no existe ninguno consolidado, siendo quizá PICT [Nierstrasz et al., 1996] el primer intento. PICT es un lenguaje basado en π-cálculo que simplifica notablemente esta notación formal, pero permite utilizar la mayor parte de sus ventajas. PICCOLA [Lumpe et al., 1997] es una evolución de PICT creado como lenguaje de configuración con tipos dinámicos. PICCOLA trata de resolver las dos principales desventajas que ha mostrado PICT: su demasiado bajo nivel, y la ausencia de *hooks*, necesarios para especializar componentes mediante técnicas de reutilización de caja negra.

3.2 Marcos de Trabajo

La reutilización de arquitecturas software se define dentro un *marco de trabajo* (*framework*, o abreviadamente MT). En general, un MT se suele definir de la siguiente forma:

> "Un MT es un diseño reutilizable de todo o parte de un sistema, representado por un conjunto de clases abstractas y la forma en la cual sus instancias interactúan".

Otra forma de expresar este concepto es:

> "Un MT es el esqueleto de una aplicación que debe ser adaptado a necesidades concretas por el programador de la aplicación".

En ambas definiciones podemos ver que un MT encapsula el patrón de la arquitectura software de un sistema o de alguna de sus partes, que puede además estar parcialmente instanciada.

Las principales ventajas que ofrecen los MT son la reducción del coste de los procesos de desarrollo de aplicaciones software para dominios específicos, y la mejora de la calidad del producto final [Fayad y Schmidt, 1997]. Sin embargo, la utilización de MT presenta actualmente ciertas dificultades, aunque se suelen englobar todas en lo que comúnmente se denomina el problema de *la documentación de un MT* [Sparks et al., 1996]. Este problema combina las dificultades que presenta la identificación del MT y de su utilización (reutilización). En primer lugar, el desarrollador de una aplicación se enfrenta con la dificultad de que le es muy complicado, si no imposible, decidir si un MT es el adecuado para resolver su problema, y hasta qué grado es capaz de resolverlo. Y en segundo lugar, si realmente el MT es el apropiado, ha de saber cómo ha de utilizarlo (acomodándolo, especializándolo y posiblemente extendiéndolo) para integrarlo en su implementación concreta. A esa adaptación se le denomina comúnmente *extensión del MT*, y constituye la forma de reutilización de los marcos de trabajo.

Usualmente los MT se extienden mediante los denominados *puntos de entrada* (*hot spots* o *hooks*), componentes (o procedimientos) cuya implementación depende del sistema o entorno final en el que se ejecuta la aplicación, y que permiten adaptarla. Dichos puntos de entrada son definidos por el diseñador del MT, cuya arquitectura debe orientar para que estos sean la forma natural de extensión del marco [Schmidt, 1997b].

El problema de la documentación se debe principalmente a que no existe una forma clara de especificar los marcos de trabajo. De hecho, la documentación de mayor parte de los MT existentes no está unificada,

es pobre, o bien se encuentra dispersa entre mucha literatura heterogénea y dirigida a usuarios de muy distintos perfiles. La razón fundamental de estos hechos es que los marcos de trabajos existentes han nacido normalmente como generalización de aplicaciones ya realizadas, cuya estructura ha sido extraída y recompuesta como un MT para su posterior reutilización. Y en cuanto a los MT de nueva creación, estos se enfrentan a la falta de estandarización de su documentación, por lo que dependen en gran medida de la voluntad y conocimientos de sus desarrolladores.

Por tanto, sería preciso disponer de mecanismos y herramientas para describir de forma unificada y completa la arquitectura de un MT, los problemas para los que está diseñado, y la forma en la que hay que utilizarlo. Existen diferentes propuestas en este sentido, que pasamos a describir.

En primer lugar, algunos autores proponen la construcción de entornos gráficos que permiten 'navegar' por la arquitectura del MT, consultando en cada uno de sus puntos la información que el usuario considera relevante. La representación de la arquitectura se realiza de forma gráfica, ya sea mediante grafos [Meusel et al., 1997][Florijn et al., 1997] o mediante secuencias de mensajes [Lange y Nakamura, 1995]. Estas propuestas aportan un conocimiento más profundo del marco de trabajo, aunque no definen una metodología que permita utilizar ese conocimiento para adaptar el MT a la nueva aplicación. Su utilidad queda por tanto reducida a la fase de identificación del MT.

Los contratos de reutilización [Steyaert et al., 1996] pueden ser también utilizados en estos entornos para definir la forma en la que pueden cooperar los diseñadores del MT y los encargados de adaptarlo o extenderlo [Codenie et al., 1997]. Este enfoque define la información necesaria para documentar adecuadamente un MT. Así, se deben documentar cada una de las clases con sus métodos, indicando los que se consideran puntos de entrada al MT, y las relaciones de herencia, composición, etc. Por otro lado, las extensiones que se realicen a un MT deben documentarse siguiendo el mismo método, incluyéndose en la documentación final.

Los *Patrones de Diseño* (*Design Patterns*) son otro enfoque muy útil tanto para diseñar la arquitectura de un MT como para documentar el diseño realizado, según postulan diferentes autores [Lange y Nakamura, 1995][Odenthal y Quiebel-Cirkel, 1997][Meijler y Engel, 1997]. Un patrón de diseño ofrece una solución abstracta a un problema de diseño que aparece muy frecuentemente, expresada mediante una conjunto de relaciones e interacciones entre componentes [Gamma et al., 1995]. Sobre los patrones de diseño hablaremos con mayor detalle en el siguiente apartado.

Finalmente, el uso de herramientas visuales es uno de los enfoques de mayor aceptación actualmente. Dichas herramientas proporcionan notaciones visuales que permiten representar tanto a los componentes como a los conectores, y definir sus enlaces. Asimismo, permiten agregar nuevos componentes y definir nuevas relaciones entre ellos. Las principales ventajas de estas propuestas se basan en su facilidad de uso y la rapidez con la que permiten identificar la adecuación de un MT a una aplicación concreta, así como los trabajos necesarios para extenderlo y adaptarlo. Por otro lado, las principales desventajas que poseen estos métodos provienen de que no suelen permitir ningún tipo de verificación sobre el diseño realizado, puesto que no suelen poseer ningún tipo de lenguaje que sirva de soporte a la notación gráfica, y que permita especificar más formalmente la arquitectura de la aplicación y verificar sus propiedades. La unión de este tipo de notaciones gráficas con los LDAs descritos anteriormente es otro de los campos en donde se está trabajando actualmente.

3.3 Patrones de Diseño

Los patrones de diseño (PD) aparecen tras la tesis de E. Gamma publicados en un libro que recoge no sólo su definición, sino todo un catálogo de PDs aplicables a la programación básica orientada a objetos, y que cubren la mayor parte de los pequeños problemas recurrentes que se plantean en el desarrollo de cualquier programa [Gamma et al., 1995].

El trabajo de Gamma ha sido extendido por distintos autores, y actualmente existen numerosos catálogos de PDs aplicables a casi todas las disciplinas (programación, enseñanza, negocios, etc.), en cuya construcción se trabaja de forma muy activa. Casi todo el trabajo sobre PDs está tutorizado por un colectivo de investigadores y empresarios que se reúnen de forma periódica en las conferencias PLoP.

Los patrones de diseño y los marcos de trabajo son disciplinas complementarias, y por lo general un MT suele estar compuesto por una colección de PDs. En este sentido, los PDs suelen ser de una granularidad menor que la de un MT.

Pero los PDs también presentan algunos inconvenientes. La propuesta de 'diseñar documentando' se basa en que todo MT debe poder ser construido en base a PDs más o menos conocidos o estándares. Sin embargo, esto no es siempre cierto, pues cada dominio de aplicación presenta peculiaridades específicas,

que deben ser abordadas de forma particular. Otro de los inconvenientes de los PDs es su elevado número, que los está haciendo inmanejables; por eso uno de los retos que también trata de abordar la comunidad de PDs es su catalogación. Por otro lado, la complejidad de ciertos PDs que modelan relaciones complicadas entre un gran número de componentes hacen difícil su comprensión y entendimiento, y por tanto su uso. Y por último, tampoco está claramente definida la *composición* de PDs, por lo que se hace compleja la reutilización de PDs para construir arquitecturas en base a ellos.

3.4 Clasificación de los Marcos de Trabajo

Hemos definido los MT y hablado sobre ellos y sobre las ventajas que pueden proporcionar para construir aplicaciones, pero no hemos citado ejemplos concretos ni de MT, ni de sus dominios de aplicación. En primer lugar vamos a dividir los marcos de trabajo según su aplicabilidad en horizontales y verticales, de acuerdo al tipo de aplicación que acometan.

Entre los horizontales existen aquellos MT dedicados a modelar infraestructuras de comunicaciones [Schmidt, 1997a][Hüni et al., 1997], las interfaces de usuario [Taylor et al., 1996], los entornos visuales [Florijn et al., 1997] y, en general, cualquiera de los aspectos relacionados con el sistema subyacente [Bruegge et al., 1993].

Dentro de los MT horizontales merecen especial atención los denominados *Marcos de Trabajo Distribuidos* (*Middleware Application Frameworks*), diseñados para integrar componentes y aplicaciones software en ambientes distribuidos, permitiendo altos niveles de modularidad y reutilización en el desarrollo de nuevas aplicaciones, y aislando la mayor parte de las dificultades conceptuales y técnicas que conlleva la construcción de aplicaciones distribuidas basadas en componentes [Fayad y Schmidt, 1997]. En la terminología de la programación orientada a componentes, este término es sinónimo de *Plataforma de Componentes Distribuidos* (*Distributed Component Platform*). Ejemplos de estos marcos de trabajo distribuidos son CORBA [Siegel, 2000], ActiveX/OLE/COM [Box, 1998], JavaBeans [www.javasoft.com], ACE [Schmidt, 1994], Hector [Arnold et al., 1996] o Aglets [Lange y Oshima, 1997].

Por otro lado, los marcos de trabajo *verticales* son aquellos desarrollados específicamente para un dominio de aplicación concreto, y cubren un amplio espectro de aplicaciones, desde las telecomunicaciones (TINA [TINA-C, 1995]), la fabricación [Schmidt, 1995], o los servicios telemáticos avanzados y la multimedia (MultiTEL [Fuentes y Troya, 1999]). Por ser muy específicos y requerir un conocimiento muy preciso de los dominios de aplicación resultan los más costosos de desarrollar, y por tanto los más caros en caso de ser comercializados.

También destacaremos otros marcos de trabajo de especial relevancia dentro de la programación orientada a componentes, que son los denominados *Marcos de Trabajo para Componentes* (*Component Frameworks*). Se trata de MTs tanto verticales como horizontales, pero que están realizados exclusivamente para el desarrollo de aplicaciones basadas en componentes reutilizables, y por tanto presentan unas características especiales para tratar los problemas específicos que plantean dichos componentes (composición tardía, extensibilidad, etc.). En general, pueden definirse como una implementación concreta de uno o más patrones de diseño mediante componentes reutilizables, realizado a medida para un dominio de uso específico [Johnson, 1997]. Desarrollados sobre una plataforma de componentes concreta, estos marcos de trabajo suponen el siguiente nivel al de los componentes base para el desarrollo de aplicaciones. OpenDoc [www.omg.org] y BlackBox [Oberon Microsystems, 1997] son dos de los pocos marcos de trabajo para componentes visuales que existen actualmente.

3.5 Técnicas de Extensión de los Marcos de Trabajo

En este apartado discutiremos otra clasificación de los marcos de trabajo, esta vez atendiendo a la forma en la que permiten ser extendidos. De esta forma hablaremos de cuatro tipos de marcos de trabajo: de caja blanca, de caja de cristal, de caja gris y de caja negra. Esta taxonomía es la misma en la que se clasifican las diferentes formas de reutilización que permiten los objetos y componentes de un sistema, pues de hecho la extensión es la forma que ofrecen los MT para su reutilización.

Los MT de *caja blanca* se extienden mediante los mecanismos básicos que ofrece el paradigma de orientación a objetos. De esta forma, los puntos de entrada al MT se presentan como clases y métodos abstractos, que han de ser implementados para extender el MT. Además, se tiene acceso al código del MT y se permite reutilizar la funcionalidad encapsulada en sus clases mediante herencia y reescritura de métodos.

Los MT que permiten extensión por *caja de cristal* admiten la inspección de su estructura e implementación, pero no su modificación ni reescritura, salvo para sus puntos de entrada.

Los MT de *caja negra* se extienden mediante composición y delegación, en vez de utilizar herencia. De esta forma, el MT tiene definido una serie de interfaces que deben implementar los componentes que extienden el MT.

En general, los MT de caja blanca exigen al usuario un conocimiento muy profundo de su estructura interna y pueden llevar aparejados los problemas que acarrea la herencia, como pueden ser las anomalías de la herencia [Matsuoka y Yonezawa, 1993] o el problema de la clase base frágil [Mikhajlov y Sekerinski, 1998]. Por otro lado, los MT de caja negra se enfrentan a los típicos problemas de la programación orientada a componentes, como son la composición tardía [Szyperski, 1996] o la clarividencia (previsión de los posibles usos futuros del MT).

El término *caja gris* fue acuñado en [Büchi y Weck, 1997], y define una forma intermedia de reutilización, en donde sólo parte del interior de un MT (o componente) se ofrece de forma pública, mientras que el resto se oculta. Es una solución intermedia entre las cajas blancas y negras, y de ahí su nombre. Aunque el nombre de caja gris es menos conocido que los otros, es el que describe la forma de extensión más utilizada.

3.6 Composición de Marcos de Trabajo

Otro de los problemas que se plantea en la construcción de software mediante la utilización de marcos de trabajo es el de su composición. Este problema surge cuando el diseñador de un sistema necesita utilizar dos o más MT dentro de su arquitectura, y donde cada una resuelve un problema concreto (p.e. un MT vertical y dos horizontales). Existe una buena discusión de este problema en [Mattson y Bosch, 1997], en donde se identifican las principales dificultades de la composición de MT, sus causas más comunes, y un conjunto de soluciones que ayudan a resolver total o parcialmente estas situaciones.

Básicamente, los problemas que se plantean a la hora de componer marcos de trabajo son los siguientes:

1. Gestión del control de la aplicación. Cada MT suele tomar control de la aplicación cuando es ejecutado, invocando a las extensiones proporcionadas por el usuario. El problema es cómo hacer coexistir dos o más entidades que tratan de tomar control simultáneamente.

2. Adaptación de los servicios ofrecidos por cada uno de los MT. Cada MT ofrece servicios, que son utilizados por los componentes de los otros MT. Sin embargo, no suelen ser completamente compatibles los servicios ofrecidos con los demandados, por lo que es preciso realizar adaptaciones.

3. Falta de funcionalidad y servicios. A veces es necesario añadir elementos adicionales a la composición de marcos de trabajo, pues su unión no cubre completamente nuestras necesidades (existen *gaps* o lagunas). El problema fundamental es que estas extensiones han de ser compatibles con el resto de los servicios ofrecidos por los distintos marcos de trabajo.

4. Solapamiento de representaciones. Este problema ocurre cuando dos o más MT contienen la representación de la misma entidad del mundo real, pero modelada desde distintas perspectivas.

5. Solapamiento de funcionalidad. En determinadas ocasiones distintos MT ofrecen la misma funcionalidad, por lo que es preciso arbitrar quién la proporciona, y mantener actualizado el estado de todos los MT, que han de saber que la función ha sido efectuada. Para llevar a cabo la arbitración y la notificación suelen utilizarse soluciones basadas en los patrones de diseño *adapter* y *observer* [Gamma et al., 1995].

Obsérvese cómo la interoperabilidad entre distintas plataformas de componentes pueden contemplarse también desde la perspectiva de la composición de marcos de trabajo, en donde los problemas anteriores también aparecen. De las plataformas de componentes hablaremos con más detalle en la sección 6.

4 Paradigmas de Programación para Sistemas Abiertos

Hasta ahora hemos considerado la construcción de aplicaciones software desde su perspectiva arquitectónica, analizando su estructura y expresándola mediante LDAs que permiten representar las abstracciones de las entidades que las componen y las relaciones entre ellas. Sin embargo, existen otros enfoques

no arquitectónicos basados en distintos modelos y paradigmas de programación. Todos ellos persiguen el mismo objetivo, la construcción de aplicaciones en sistemas abiertos y distribuidos, aunque cada uno trata de realizarlo desde perspectivas diferentes.

Al principio sólo existía la programación secuencial, pues cada programa sólo se ejecutaba en una máquina monoprocesador y aislada. Posteriormente aparecieron las máquinas multiprocesador y los sistemas operativos multitarea, por lo que nacieron nuevos paradigmas y mecanismos de programación. En este sentido, la programación concurrente proporciona la noción de procesos, aportando una serie de primitivas para controlarlos y algunos mecanismos de comunicación y sincronización. Dichos mecanismos pueden estar basados en memoria compartida (semáforos y monitores) o en mensajes (rendez-vous y RPCs). Por último aparece la programación distribuida, que comenzó a tomar auge cuando la interconexión entre distintos computadores pudo hacerse de forma eficiente y económica. Las primeras soluciones estaban basadas en una extensión de los mecanismos que ofrecía la programación concurrente para tratar con los problemas de localización y dispersión que presentan las aplicaciones distribuidas (sockets, XWindows, RPCs, etc.), pero en seguida se vio que estas extensiones no eran suficientes. Incluso los paquetes comerciales existentes a este nivel (PVM, MPI, Horus o ISIS) han probado su efectividad en cuanto a la implementación de las aplicaciones, pero no a la hora de diseñarlas.

En primer lugar, esas extensiones son de demasiado bajo nivel para permitir el diseño y construcción de grandes aplicaciones distribuidas con el nivel de abstracción necesario. En este sentido, la programación paralela había desarrollado algunos *esquemas organizativos* (también llamados *skeletons*) de mayor nivel conceptual que los semáforos, monitores o sockets, y que deberían poder expresarse a nivel de diseño: esquemas como maestro-trabajadores o las granjas de procesos [Cole, 1989][Pelagatti, 1993]. Lo mismo ocurre con los esquemas desarrollados por la inteligencia artificial, como pueden ser las pizarras (*blackboards*) o las redes de contratos.

En segundo lugar, los sistemas abiertos introducen la necesidad de integrar programas heterogéneos para formar aplicaciones, así como de poder trabajar con aplicaciones ya existentes (*legacy systems*).

Y por último, y desde un punto de vista formal, cierta propiedades del comportamiento colectivo de los componentes (localidad, movilidad, seguridad, etc.) no son fácilmente manejables utilizando las lógicas y semánticas desarrolladas para lenguajes concurrentes convencionales.

Entre los principales modelos y paradigmas de programación que han surgido para el diseño e implementación de aplicaciones para sistemas abiertos y distribuidos se encuentran, entre otros la coordinación, los actores y la programación concurrente, la programación paralela computacional, la reflexión y la meta-programación, la programación orientada a componentes, la computación móvil, o los agentes.

Cada uno de estos paradigmas trata de solucionar los problemas mencionados anteriormente desde un enfoque distinto, no existiendo ningún paradigma lo suficientemente general y extendido como para abarcarlos todos. En esta lección vamos a centrarnos principalmente en la programación orientada a componentes (POC), los problemas que trata de solucionar, los conceptos sobre los que se fundamenta, y los mecanismos que ofrece.

5 La Programación Orientada a Componentes (POC)

La Programación Orientada a Componentes (POC) [Szyperski y Pfister, 1997] [Szyperski, 1998] aparece como una variante natural de la programación orientada a objetos (POO) para los sistemas abiertos, en donde la POO presenta algunas limitaciones; por ejemplo, la POO no permite expresar claramente la distinción entre los aspectos computacionales y meramente composicionales de la aplicación, no define una unidad concreta de composición independiente de las aplicaciones (los objetos no lo son, claramente), y define interfaces de demasiado bajo nivel como para que sirvan de contratos entre las distintas partes que deseen reutilizar objetos.

La POC nace con el objetivo de construir un mercado global de componentes software, cuyos usuarios son los propios desarrolladores de aplicaciones que necesitan reutilizar componentes ya hechos y probados para construir sus aplicaciones de forma más rápida y robusta.

Las entidades básicas de la POC son los componentes, en el mismo sentido que los hemos definido en la sección 2, cajas negras que encapsulan cierta funcionalidad y que son diseñadas para formar parte de ese mercado global de componentes, sin saber ni quién los utilizará, ni cómo, ni cuándo. Los usuarios conocen acerca de los servicios que ofrecen los componentes a través de sus interfaces y requisitos, pero normalmente ni quieren ni pueden modificar su implementación.

En el contexto de este documento consideraremos a la POC como un paradigma de programación que

se centra en el diseño e implementación de componentes, y en particular en los conceptos de encapsulación, polimorfismo, composición tardía y seguridad. No discutiremos aquí sin embargo otros aspectos de marketing que lleva asociado un mercado de componentes software, como cualquier otro mercado: distribución, comercialización, empaquetamiento, almacenamiento, publicidad, licencias, etc. Aunque son de especial relevancia para la POC, hemos preferido centrarnos solamente en los aspectos de técnicos de desarrollo y utilización de los componentes software.

5.1 El concepto de componente

Actualmente existe una gran discrepancia entre la mayoría de los ingenieros de software sobre el concepto de Componente Software. La polémica suscitada recientemente entre Clemens Szyperski y Bertrand Meyer en la columna "Beyond Objects"de la revista *Software Development* [disponible en la Web en la dirección //www.sdmagazine.com/features/uml/beyondobjects/] sobre qué es y qué no es un componente, sus propiedades y su naturaleza, podemos considerarla como uno de los ejemplos más recientes y representativos sobre la confusión que suscita el propio concepto de componente. Quizás, las causas podamos encontrarlas en la problemática existente en los modelos de componentes actuales, y que también ha destacado Jon Hopkins en el número especial de las Comunicaciones de ACM de Octubre de 2000 dedicado a marcos de trabajo basado en componentes: "Mientras que el concepto de componente software es conocido virtualmente desde los inicios de la ingeniería del software, la problemática y aspectos prácticos relativos a los mismos han evolucionado continuamente a lo largo del tiempo".

Actualmente existen distintas definiciones de componente, aparte de la ya mencionada anteriormente de Szyperski. Entre todas las definiciones hemos querido destacar las siguientes, que son quizá las más representativas y aceptadas por la comunidad de ingeniería del software:

- A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject by composition by third parties (agreed definition by the participants of WCOP'96 [Szyperski y Pfister, 1997]).
- A set of simultaneously deployed 'atomic' components. An atomic component is a 'module' plus a set of 'resources'. A module is a set of classes and possibly other non object-oriented constructs, such as procedures or functions. A resource is a frozen collection of typed items [that parametrize the component] ("technical definition"by Clemens Szyperski, [Szyperski, 1998]).
- The "seven criteria"definition [Meyer, 1999]:
  1. May be used by other software elements.
  2. May be used by clients without the intervention of the component's developer.
  3. Includes a specification of all dependencies.
  4. Includes a specification of the functionality it offers.
  5. Is usable on the sole basis of its specifications.
  6. Is composable with other components.
  7. Can be integrated into a system quickly and smoothly.
- A logically cohesive, loosely coupled module that denotes a single abstraction (Grady Booch, 1987)
- Prefabricated, pre-tested, self-contained, reusable software modules —bundles of data and procedures— that perform specific functions (Meta Group, 1994)
- A static abstraction with plugs [Nierstrasz, 1995]
- A piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability (Jed Harris, 1995 —also adopted by Orfali et al.)
- Anything that can be composed (agreed definition by the participants of WCOP'97 [Weck et al., 1997]).
- Self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and defined resuse status (James Sametinger, 1997).
- A system-independent binary unit that implements one or more interfaces (Kai Koskimies, , in [Broy et al., 1998]).

- A binary unit that exports and imports functionality using a standardized interface mechanism (Michael Stal, in [Broy et al., 1998]).
- A part of a kit, which defines a common set of protocols between its members. A generic chunk of software with robust, well-defined interfaces (Allan C. wills, in [Henderson-Sellers et al., 1999]).
- A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. It is a meta-subclass of Classifier (OMG UML 1.3 specification, 1999).

Por otro lado, existe también una confusión habitual entre el concepto de componente con el de clase, objeto, módulo, etc. Con el fin de distinguirlas entre sí, recogemos aquí las definiciones más aceptadas de otras entidades software, así como sus principales diferencias.

Clase Las clases no hacen referencia explícita a sus dependencias y requisitos. Las clases suelen construirse mediante herencia de implementación, y por tanto no suelen permitir una instanciación e instalación por separado de la clase base y de sus clases hijas. Esto puede causar problemas si se realiza composición tardía (pudiendo aparecer, por tanto, el problema de la clase base frágil).

Módulo Un módulo es un conjunto de clases, junto con opcionalmente otros elementos no orientados a objetos, como pueden ser procedimientos y funciones.

Paquetes/Librerías Un paquete es un conjunto de clases, usualmente agrupadas conceptualmente. Los paquetes no suelen ser ejecutables, y pueden ser consideradas como la versión orientada a objetos de las librerías tradicionales.

Subsistema Un subsistema es una agrupación de elementos de modelado que representa una unidad de comportamiento en un sistema físico. De acuerdo a la especificación de UML 1.3, un subsistema es una subclase de Classifier y de Package, y por tanto no tiene un comportamiento propio, sino que organiza elementos ejecutables que lo poseen (por ejemplo, componentes).

Recurso Un recurso es una colección no modificable de elementos con tipo [Szyperski, 1998].

Frameworks Los marcos de trabajo suelen estar compuestos de componentes, de los cuales unos están fijados por el propio marco, y otros son los que proporciona el usuario para especializar el marco de trabajo.

Objects Are apples oranges? [Szyperski, 2000].

5.2 Otros conceptos básicos de la POC

Aparte del propio concepto de componente software, existe otro conjunto de conceptos básicos que intervienen en la POC, y que permiten diferenciarla del resto de los paradigmas de programación. Entre ellos se encuentran los siguientes:

Composición tardía Dícese de aquella composición que se realiza en un tiempo posterior al de la compilación del componente, como puede ser durante su enlazado, carga o ejecución, y por alguien ajeno a su desarrollo, es decir, que sólo conoce al componente por su interfaz o contrato, pero no tiene porqué conocer ni sus detalles de implementación, ni la forma en la que fue concebido para ser usado.

Entornos Un entorno es el conjunto de recursos y componentes que rodean a un objeto o componente dado, y que definen las acciones que sobre él se solicitan, así como su comportamiento. Se pueden definir al menos dos clases de entornos para los componentes: el entorno de *ejecución* y el de *diseño*. En primero de ellos es el ambiente para el que se ha construido el componente, y en donde se ejecuta normalmente. El entorno de diseño es un ambiente restringido, que se utiliza para localizar, configurar, especializar y probar los componentes que van a formar parte de una aplicación, y en donde los componentes han de poder mostrar un comportamiento distinto a su comportamiento normal durante su ejecución.

Eventos Mecanismo de comunicación por el que se pueden propagar las situaciones que ocurren en un sistema de forma asíncrona. La comunicación entre emisores y receptores de los eventos se puede realizar tanto de forma directa como indirecta, siguiendo el mecanismo *publish-and-subscribe* que describiremos más adelante. Los eventos suelen ser emitidos por los componentes para avisar a los componentes de su entorno de cambios en su estado o de circunstancias especiales, como pueden ser las excepciones.

Reutilización Habilidad de un componente software de ser utilizado en contextos distintos a aquellos para los que fue diseñado (reutilizar *no* significa usar más de una vez). Existen 4 modalidades de reutilización, dependiendo de la cantidad de información y posibilidades de cambio que permita el componente a ser reutilizado: caja blanca, caja de cristal, caja gris y caja negra. Dichas modalidades coinciden con las que describiremos para los marcos de trabajo (sección 3.5).

Contratos Especificación que se añade a la interfaz de un componente y que establece las condiciones de uso e implementación que ligan a los clientes y proveedores del componente. Los contratos cubren aspectos tanto funcionales (semántica de las operaciones de la interfaz) como no funcionales (calidad de servicio, prestaciones, fiabilidad o seguridad).

Polimorfismo Habilidad de un mismo componente de mostrarse de diferentes formas, dependiendo del contexto; o bien la capacidad de distintos componentes de mostrar un mismo comportamiento en un contexto dado. Ambas acepciones representan los dos lados de una misma moneda. En POO el polimorfismo se relaciona con la sobre-escritura de métodos y la sobrecarga de operadores (polimorfismo *ad-hoc*). Sin embargo, en POC muestra tres nuevas posibilidades:

- La *reemplazabilidad*, o capacidad de un componente de reemplazar a otro en una aplicación, sin romper los contratos con sus clientes (también conocido por polimorfismo de *subtipos* o de *inclusión*).
- El polimorfismo *paramétrico*, o implementación genérica de un componente. Similar en concepto a los *generics* de Ada o a los *templates* de C++, el polimorfismo paramétrico no se resuelve como ellos en tiempo de compilación (generando la típica explosión de código) sino en tiempo de ejecución.
- Por último, el polimorfismo *acotado* combina los polimorfismos de inclusión y paramétrico para poder realizar restricciones sobre los tipos sobre los que se puede parametrizar un componente [Szyperski, 1998].

Seguridad Por seguridad en este contexto se entiende la garantía que debe ofrecer un componente de respetar sus interfaces y contratos, y forma el concepto básico sobre el que se puede garantizar la seguridad (en su acepción más amplia) de un sistema. Se puede hacer una distinción entre seguridad a nivel de *tipos* y a nivel de *módulos*. La primera se refiere a que la invocación de los servicios de un componente se realice usando parámetros de entrada de los tipos adecuados (o supertipos suyos: *contravarianza*) y que los servicios devuelvan también valores del tipo adecuado (o subtipos suyos: *covarianza*). La seguridad a nivel de módulo [Szyperski y Gough, 1995] se refiere a que sólo se utilicen los servicios ajenos al componente que hayan sido declarados por él, y no otros.

Reflexión La reflexión es la habilidad de una entidad software de conocer o modificar su estado. A la primera forma se le denomina *reflexión estructural*, y a la segunda *reflexión de comportamiento*. En la sección 5.5 hablaremos con más detalle de este concepto.

5.3 tendencias actuales de la POC

Dentro de la POC se está trabajando en varios frentes: la adecuación de los lenguajes de programación y modelos de objetos existentes para que incorporen estos conceptos; el diseño de nuevos lenguajes y modelos de componentes; la construcción de herramientas de desarrollo y marcos de trabajo para componentes; y la aplicación de técnicas formales para razonar sobre las aplicaciones desarrolladas a base de componentes.

En cuanto a los lenguajes de programación, sólo hay unos pocos que realmente incorporen conceptos suficientes para realizar una programación orientada a componentes: Oberon, Java, Ada95, Modula-3 y Component Pascal. Sin embargo, ninguno de ellos incorpora todos los conceptos, sino solamente unos pocos. Quizá Java y Component Pascal sean los más destacados: ambos permiten compatibilidad binaria

(considerando como *binarios* también a los Javabytes) y la composición tardía, son seguros tanto a nivel de tipos como de módulos, y permiten encapsulamiento y distintas clases de polimorfismo.

Por otro lado, muchos de los lenguajes que dicen ser 'orientados a componentes', son realmente lenguajes de configuración o entornos de desarrollo visuales, como es el caso de Visual Basic, Delphi, o C++ Builder. Ellos permiten crear y ensamblar componentes, pero separan totalmente los entornos de desarrollo y de utilización de componentes.

Más allá de estos meros entornos visuales de desarrollo de componentes, los *marcos de trabajo para componentes* (MTC) suponen el entorno natural en donde los componentes nacen, viven, y cumplen su misión. Los primeros MTC nacieron de la idea de los *documentos compuestos*, en donde la entidad básica pasa a ser el documento, en vez de la aplicación. Los usuarios dejan así de disponer de un conjunto de aplicaciones, cada una con una idea distinta de lo que es un documento, y pasan a disponer sólo de documentos. A su vez, un documento puede contener a su vez a otros documentos, y es la responsabilidad del sistema manejar a cada tipo de documento con su aplicación correspondiente.

- Por ejemplo en Visual Basic todo son *formularios*, que pueden contener *controles*, aunque la lista de los posible controles es completamente abierta. De hecho, la industria comenzó pronto a ofrecer desde hojas de cálculo a procesadores de texto como controles de Visual Basic para poder integrarlos fácilmente.

- En OLE se define el concepto de *contenedor* y se extiende el concepto de control, que pasa a denominar cualquier tipo de servidor de documentos. De esta forma los componentes pueden ser tanto contenedores como servidores de documentos simultáneamente, y es posible que, por ejemplo, un documento Word se incluya en una hoja de cálculo Excel, que a su vez forme parte de otro documento Word.

- Otro ejemplo de documento compuesto es la Web, en donde es posible incluir en páginas HTML multitud de objetos distintos, como por ejemplo los Applets de Java. Aunque más reciente que OLE, supone un paso atrás en cuanto al concepto que estamos tratando, pues los objetos que forman parte de las páginas Web no pueden ser a su vez contenedores, es decir, estamos frente a una tecnología similar a la que ofrecía Visual Basic con sus formularios y controles (aunque ya es posible en la última versión de Visual Basic definir controles que sean componentes ActiveX, que sí son contenedores de otros componentes).

- OpenDoc fue diseñado desde el principio como un MTC para documentos compuestos, y por tanto ofrece numerosas ventajas frente a Visual Basic y OLE. Sin embargo, no ha gozado de demasiado éxito comercial, aunque se espera que vuelva a renacer tras ser incorporado por la OMG como parte de las facilidades de CORBA.

5.4 Problemas típicos de la POC

POC es una disciplina muy joven y por tanto en la que los resultados obtenidos hasta ahora se centran más en la identificación de los problemas que en la resolución de los mismos. En ese sentido, destacaremos los siguientes retos y problemas con los que se enfrenta actualmente:

1. *Clarividencia.* Citado originariamente en el libro de B. Meyer [Meyer, 1997] y comúnmente aludido por Miguel Katrib en sus amenas conferencias (él lo denomina *Macumba*), este problema se refiere a la dificultad con la que se encuentra el diseñador de un componente al realizar su diseño, pues no conoce ni quién lo utilizará, ni cómo, ni en qué entorno, ni para qué aplicación; además, se encuentra también con la paradoja de "*maximizing reuse minimizes use*" [Szyperski, 1998]. Este problema está intrínsecamente ligado a la composición tardía y reusabilidad de los componentes.

2. *Evolución de los componentes.* La gestión de la evolución es un problema serio, pues en los sistemas grandes han de poder coexistir varias versiones de un mismo componente. Existen distintos enfoques para abordar este problema (desde la inmutabilidad de interfaces de COM a la integración de interfaces que propugna CORBA), aunque ninguno totalmente satisfactorio.

3. *Percepción del entorno* (*environment-awareness*). Esta es la habilidad de un objeto o componente de descubrir tanto el tipo de entorno en donde se está ejecutando (de diseño o de ejecución), como los servicios y recursos disponibles en él. La inspección y la reflexión estructural son dos mecanismos comúnmente utilizados para implementar esta habilidad.

4. *Particularización.* Cómo particularizar los servicios que ofrece un componente para adaptarlo a las necesidades y requisitos concretos de nuestra aplicación, sin poder manipular su implementación. La reflexión de comportamiento es la técnica comúnmente utilizada, componiendo el componente con envolventes (*wrappers*, *decorators* o *adapters*) que le proporcionan la funcionalidad apropiada. La composición simultánea de envolventes y el estudio de las propiedades del nuevo componente a partir de las del antiguo son campos de investigación actualmente abiertos.

5. *Falta de soporte formal.* Por otro lado, la POC también se encuentra con otro reto añadido, como es la dificultad que encuentran los métodos formales para trabajar con sus peculiaridades, como puede ser la composición tardía, el polimorfismo o la evolución de los componentes.

6. *El problema de la clase base frágil* (FBCP). Este problema ocurre cuando la superclase de una clase sufre modificaciones. El FBCP existe a dos niveles, sintáctico y semántico. A nivel sintáctico ocurre cuando las modificaciones de la superclase son puramente a este nivel, como por ejemplo sucede cuando se añade un método a una superclase, lo que en teoría debería obligar a recompilar todas sus clases hijas. El FBCP a nivel semántico ocurre cuando lo que se altera es la implementación de los métodos de la superclase, lo que puede llevar a que el comportamiento de las clases hijas se altere sustancialmente [Mikhajlov y Sekerinski, 1998].

7. *Asincronía y carreras de eventos.* En los sistemas abiertos y distribuidos, los tiempos de comunicación no están acotados inicialmente, ni se pueden despreciar los retrasos en las comunicaciones. Por consiguiente, es muy difícil garantizar el orden relativo en el que se distribuyen los eventos, sobre todo cuando los producen distintas fuentes. Asimismo, el proceso de difusión de eventos es complicado cuando tanto los emisores como los receptores pueden cambiar con el tiempo. La situación se complica cuando lo que se pretende es razonar formalmente sobre la aplicación a partir de sus componentes.

8. *Interoperabilidad.* Actualmente, los contratos de los componentes se reducen a la definición de sus interfaces a nivel sintáctico, y la interoperabilidad se reduce a la comprobación de los nombres y perfiles de los métodos. Sin embargo, es necesario ser capaces de referirnos y buscar los servicios que necesitamos por algo más que sus nombres, y poder utilizar los métodos ofrecidos en una interfaz en el orden adecuado. El estudio de la interoperabilidad de componentes a nivel de protocolos (orden entre los mensajes) o semántico ("significado" de las operaciones) es otro de los problemas abiertos.

Quizá sea POC la disciplina de la programación con mayor proyección a corto y medio plazo, no sólo por el planteamiento tan apropiado que realiza de la construcción de aplicaciones para sistemas abiertos, sino por la apremiante necesidad que existe en la industria de un paradigma de programación como el que POC proporciona. Actualmente es uno de los campos de investigación más activos de la comunidad software, tanto académica como industrial.

5.5 Reflexión y Metaprogramación

La reflexión es una propiedad que permite representar el estado de los objetos de un sistema como entidades de primera clase, y por tanto poder observarlos, manipularlos y razonar sobre ellos como elementos básicos. Fundamentalmente hay dos tipos de reflexión: la lógica y la computacional.

La reflexión lógica se aplica sobre los marcos formales lógicos de razonamiento, y permite a sus módulos ser accesibles como entidades de primera clase, de forma que operaciones como la composición o la transformación de módulos pueden ser definidas rigurosamente en la lógica y por tanto ser también ejecutadas. Otra consecuencia importante de la reflexión lógica es que permite representar unas lógicas en términos de otras, y de ahí poder hablar de interoperabilidad formal [Meseguer, 1998].

Por reflexión computacional se entiende la habilidad de una entidad software de conocer o modificar su estado. A la primera forma se le denomina *reflexión estructural*, y a la segunda *reflexión de comportamiento* [Manola, 1993]. En lo que sigue el término reflexión se referirá solamente a la reflexión computacional, en cualquiera de sus dos formas.

A la programación que utiliza técnicas de reflexión se le denomina Metaprogramación [Kiczales et al., 1991], y es la fuente de numerosos modelos de objetos que persiguen un tratamiento separado de los distintos requisitos de una aplicación. De entre todos ellos mencionaremos el modelo de Filtros Composicionales

de M. Aksit, el modelo de capas LayOM de Jan Bosch, y la Programación Orientada a Aspectos de George Kiczales y el grupo de Xerox.

Por ejemplo, el modelo de Filtros Composicionales [Aksit et al., 1993] es una extensión del modelo de objetos para permitir la separación entre la computación y el resto de los aspectos y requisitos de una aplicación. Este modelo se apoya en la visión de un objeto como una entidad que se comunica con su entorno a través de mensajes, mediante los cuales recibe las invocaciones de sus métodos y devuelve las respuestas. Los Filtros son entidades que capturan los mensajes entrantes y salientes de un objeto y los manipulan de acuerdo a la funcionalidad que trate de proporcionar cada filtro al objeto al que se asocia. De esta forma, cada requisito del sistema o de la aplicación se añade a los componente mediante distintos filtros. Existen filtros de entrada (*input filters*) y de salida (*output filters*) dependiendo del tipo de mensajes que capturan. Los filtros pueden componerse, constituyendo una cadena de filtros que han de atravesar los mensajes para llegar al objeto o salir de él. Cada uno de ellos viene definido por un *patrón de aceptación* y por la serie de acciones que el filtro realiza sobre cada mensaje, lo que constituye el *tipo* de filtro.

Por otro lado, la Programación Orientada a Aspectos se basa en descomponer los programas en diferentes *aspectos*, encargándose cada uno de estos aspectos de un requisito independiente del programa (prestaciones, seguridad, concurrencia, fiabilidad), y pudiendo estar desarrollado de forma separada de los demás, incluso en lenguajes diferentes [Kiczales et al., 1997]. Como analogía, piénsese en las distintos partes que componen la construcción de un edificio (electricidad, conducciones de agua, de aire, etc.). Lo normal es tener planos de cada uno de estos aspectos por separado, que afectan a todo el edificio, en vez de un plano de cada planta que contenga todos los distintos aspectos. Esto es lo que diferencia a los aspectos de otros paradigmas. Una herramienta especial, llamada *tejedor* (*weaver*) se encarga de juntar ordenadamente todos los aspectos entre sí, y con el núcleo computacional del programa.

6 Modelos y Plataformas de Componentes

Más que a describir los principales modelos, plataformas, y marcos de trabajo para componentes (PCD) que existen hoy en día, hemos preferido dedicar esta sesión a discutir las principales características que distinguen a las plataformas de componentes, abstrayendo sus elementos y servicios comunes con independencia de cómo se implementan en cada PCD particular.

Como mencionamos en la primera sección, un *Modelo de Componentes* es un estándar que define los interfaces de los componentes y los mecanismos para interconectarlos entre ellos, mientras que un *Marco de Trabajo Distribuido* es un MT diseñado para integrar componentes y aplicaciones software en ambientes distribuidos, permitiendo la modularidad y reutilización en el desarrollo de nuevas aplicaciones [Fayad y Schmidt, 1997]. En la terminología de la programación orientada a componentes, este término es sinónimo de *Plataforma de Componentes Distribuidos* (PCD). Este tipo de marcos ofrecen un entorno de desarrollo y de ejecución de componentes software que permite aislar la mayor parte de las dificultades conceptuales y técnicas que conlleva la construcción de aplicaciones basadas en componentes. Cada plataforma se apoya en los estándares y mecanismos definidos por su modelo de componentes base; en este sentido, podemos definir una plataforma como una implementación de los mecanismos del modelo, junto con una serie de herramientas asociadas. Ejemplos de estas plataformas son ActiveX/OLE, Enterprise Beans y Orbix, que se apoyan en los modelos de componentes COM, JavaBeans y CORBA, respectivamente.

En esta sección describiremos una serie de conceptos básicos sobre los que se apoyan las plataformas de componentes distribuidos: los componentes, sus interfaces, los contenedores, la meta-información, los servicios disponibles y los entornos de desarrollo integrados que proporcionan para desarrollar componentes. Todas las plataformas incorporan estos conceptos y ofrecen esos servicios a sus componentes, aunque fundamentalmente lo único que varía es la forma en la que los implementan.

6.1 Componentes e interfaces

El término *componente* ya fue definido anteriormente como 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" [Szyperski, 1998].

Las interfaces de un componente determinan tanto las operaciones que el componente implementa como las que precisa utilizar de otros componentes durante su ejecución. En los modelos de componentes

habituales cada interfaz va a venir determinada por el conjunto de atributos y métodos públicos que el componente implementa, y por el conjunto de eventos que emite. Los eventos especifican la forma en la que el componente notifica al exterior una respuesta a un estímulo externo o bien una cambio en una condición interna (p.e. la modificación del estado de una variable). En la interfaz de un componente se especifica tanto la signatura del evento como la condición que hace que éste se produzca, pero sin indicar ni el consumidor del evento ni la forma en la que se ha de tratar, por ser detalles que el componente ni puede, ni quiere conocer.

La interacción tradicional entre componentes está basada en RPCs para la invocación de los métodos públicos, y en el modelo de *publish-and-subscribe* para los eventos. En este modelo la comunicación entre emisores y receptores se puede realizar tanto de forma directa como indirecta. En el primer caso los receptores se registran en el emisor, quien envía los eventos a los receptores que tenga registrados en el momento de producirse el evento. Pero también pueden utilizarse *distribuidores*, entidades a las que el emisor envía el evento para que sea distribuido a los receptores potenciales. Los receptores pueden registrarse con los distribuidores, o bien estos últimos pueden utilizar estrategias de difusión total (*broadcast*) o parcial (*multicast*) para comunicar las ocurrencias de un evento a los receptores.

6.2 Contenedores

Los componentes suelen existir y cooperar dentro de *contenedores*, entidades software que permiten contener a otras entidades, proporcionando un entorno compartido de interacción. Se aplica sobre todo para objetos y componentes visuales, que contienen a su vez a otros objetos visuales. Por ejemplo, un control ActiveX puede ser un contenedor de otros controles ActiveX.

Normalmente la relación entre los componentes y sus contenedores se establece mediante eventos. Como ejemplo, pensemos en un contenedor, que se registra al ser creado en un escritorio o ventana (en general, un *drop site*, DS) para ser informado de cuándo alguien deposita ahí un componente. Cuando un usuario hace una operación que arrastra y suelta (*drag-and-drop*) un componente y lo deposita en el DS, el mecanismo de eventos del DS se lo notifica al contenedor, invocando el procedimiento que previamente éste registró, y pasándole como parámetro una referencia (*handle*) al componente que ha sido dejado en el DS. Ante esta notificación, el contenedor suele cambiar el aspecto del icono del componente para indicar que la operación de arrastre (*drag-and-drop*) ha terminado con éxito, y pasarle al componente una referencia a los servicios que ofrece el contenedor, para que el componente pueda utilizarlos.

6.3 Meta-información

Los nuevos estándares de componentes ya especifican el tipo de información que un componente debe hacer pública sobre sí mismo y sobre sus propiedades. Esa *meta-información* es la que permite a los contenedores, entornos y herramientas de desarrollo, y a otros componentes descubrir la funcionalidad que ofrece un componente, y poder manipularlo. A la acción de examinar la meta-información de un componente se le denomina *inspección*, y puede ser tanto estática, en tiempo de diseño, como dinámica, en tiempo de ejecución.

La forma usual de implementar la meta-información de los componentes es mediante técnicas reflexivas, que están cobrando cada vez más importancia pues constituyen la mejor forma de descubrir los recursos de un entorno dado y adaptarlos de forma flexible y modular para construir aplicaciones (sección 5.5). *Introspección* es la acción de examinar la propia meta-información, cuando el modelo de componentes soporta reflexión.

6.4 Entornos de Desarrollo Integrados

Un *Entorno de Desarrollo Integrado* (IDE) es una aplicación visual que sirve para la construcción de aplicaciones a partir de componentes. Por lo general todas ellas cuentan con los siguientes elementos:

- una o más 'paletas' para mostrar como iconos los componentes disponibles;
- un 'lienzo' o 'contenedor' en el cual se colocan los componentes y se interconectan entre sí;
- editores específicos para configurar y especializar los componentes;
- visores (*browsers*) para localizar componentes de acuerdo a ciertos criterios de búsqueda;

- directorios de componentes;
- acceso a editores, intérpretes, compiladores y depuradores para desarrollar nuevos componentes;
- y finalmente, acceso a algunas herramientas de control y gestión de proyectos y CSCW, esenciales para grandes proyectos software.

Ejemplos de IDEs son Visual Studio de Microsoft, VisualAge de IBM o VisualCafe de Symantec, complementados con lenguajes de configuración como VBScript y JavaScript.

6.5 Servicios y facilidades

La programación de aplicaciones distribuidas se basa en un conjunto de servicios que proporcionan a los componentes el acceso a los recursos compartidos de una forma segura y eficiente. Estos servicios suelen englobarse en las siguientes categorías básicas:

- *Comunicaciones Remotas*: proporcionan una serie de mecanismos para la comunicación remota entre componentes, como pueden ser los mensajes, RPCs, canales, etc.
- *Servicios de Directorio*: proporcionan un esquema de direccionamiento global para los recursos, servicios y componentes de un sistema, incluyendo la asignación de sus nombres, y su organización, localización y acceso.
- *Seguridad*: proporcionan el acceso seguro y autentificado a los recursos y servicios del sistema, así como protección frente a ataques externos o internos.
- *Transacciones*: proporcionan los mecanismos para coordinar las interacciones de los componentes cuando estos comparten datos críticos, de forma que siempre se pueda garantizar su coherencia.
- *Gestión*: proporcionan un conjunto de facilidades para la monitorización, gestión y administración de los componentes, recursos y servicios del sistema.

Es importante señalar que casi todas las PCD actuales implementan estos mecanismos y servicios, lo que nos ha permitido describirlos y estudiarlos de una forma general.

7 Bases para una metodología

Una vez definidos los conceptos y mecanismos fundamentales que constituyen las bases sobre las que se apoya el desarrollo de software basado en componentes (DSBC), en esta sección discutiremos el impacto que la POC y los componentes COTS introducen sobre las metodologías tradicionales de desarrollo de software.

En primer lugar, y desde la perspectiva de los componentes, su diseño ha de realizarse tratando que puedan ser reutilizados globalmente, dentro de un mercado global de software. Esto supone el primer compromiso, pues como mencionamos antes, "*maximizing reuse minimizes use*", y es difícil diseñar un componente sin tener en cuenta los posibles usos que puedan darse.

Por otro lado, el desarrollo de componentes no puede hacerse de forma arbitraria, sino de acuerdo a los modelos existentes, incorporando aquellos mecanismos que permitan a dichos componentes interoperar con otros. En este sentido, el desarrollo de componentes se ve ampliamente beneficiado por la existencia de dichos modelos y plataformas de componentes, así como de las pasarelas entre unos y otros. A un primer nivel, la interoperabilidad entre componentes (casi independientemente del modelo de componentes sobre el que se apoyen) podemos decir que está actualmente superada: las principales PCDs incorporan ya pasarelas que permiten a los componentes entenderse y trabajar independientemente de si son objetos CORBA, COM o simples beans. Y los nuevos modelos de componentes, como puede ser CCM (CORBA Component Model) [OMG, 1999] ya incorporan desde su origen la interoperabilidad entre componentes de distintos modelos. Sin embargo, una vez esa interoperabilidad básica está casi superada, nuevos retos se plantean al ser necesaria también una interoperabilidad entre componentes a un nivel más alto, como puede ser el semántico. En este sentido, la comunidad científica está trabajando en proporcionar estándares, mecanismos y herramientas que permitan extender las descripciones de los componentes para que puedan incorporar ese tipo de información, y que pueda ser manejada a la hora de construir aplicaciones y razonar sobre ellas.

Otro aspecto que aparece en estos ambientes es la necesidad de extender la funcionalidad de los repositorios de componentes y los servicios de directorio, así como la de los corredores de servicios (*traders*) que los explotan. Un mercado global de componentes COTS necesita disponer de mejores servicios para almacenar, localizar, distribuir y recuperar componentes.

Asociados a la recuperación de componentes, aparece también toda una serie de problemas en el momento en que el emparejamiento entre el componente que buscamos y el que esté disponible no sea uno-a-uno. Si necesitamos (y normalmente este será el caso) utilizar varios componentes COTS para obtener los servicios que hemos especificado, nos veremos obligados a tratar con problemas de solapamiento, lagunas, e incompatibilidades entre los componentes recuperados, además de los clásicos problemas que puede ocasionar su potencial adaptación. Es preciso por tanto disponer de mecanismos y procedimientos para identificar y resolver este tipo de problemas.

Asimismo, y aunque se aleja un poco del ámbito de este artículo, también hace falta definir toda una serie de estándares, herramientas y servicios para poder incorporar los aspectos de mercadotecnia básicos en un mercado global de componentes: licencias, políticas de precios y alquileres, documentación, publicidad, disponibilidad, demos, etc.

Eso es en lo relativo a los componentes y al mercado. Sin embargo, en donde la Ingeniería del Software se plantea sus mayores retos es en cuanto a metodología de diseño y construcción de aplicaciones basados en estos componentes COTS. La razón fundamental se debe a que las metodologías tradicionales utilizan un enfoque descendente: desde el análisis de requisitos a las fase de especificación, diseño, implementación y pruebas; cada uno de esos pasos se va refinando sucesivamente, consiguiendo en cada etapa una definición más precisa de los componentes finales de la aplicación, hasta su implementación definitiva. Todos los elementos que se obtienen de esta forma (desde la arquitectura de la aplicación hasta los componentes finales) se pretende que puedan ser reutilizados en otras aplicaciones, dentro de lo posible [Rumbaugh et al., 1999].

Esta situación cambia si queremos utilizar componentes COTS, pues su existencia debe ser considerada desde las primeras fases de la metodología, incluyendo por ejemplo el análisis de requisitos [Robertson y Robertson, 1999]. De esta forma se altera el proceso tradicional, pues aparecen en él una serie de restricciones muy importantes que condicionan todas sus fases (las impuestas al tener que considerar componentes ya existentes a la hora de especificar, diseñar e implementar las aplicaciones), así como ciertos procesos ascendentes (p.e. la búsqueda, identificación y recuperación de componentes COTS). De esta forma están apareciendo numerosas líneas de investigación que tratan de estudiar el impacto en las metodologías tradicionales de desarrollo de software en ambientes COTS.

Finalmente, sólo destacar la importancia que están tomando también los marcos de trabajo a la hora de desarrollar aplicaciones basadas en componentes. Aparte de las ventajas que hemos mencionado cuando los describíamos en la sección 3, se ha visto que sobre dichas estructuras es donde deben descansar con mayor peso las propiedades no funcionales de las aplicaciones (calidad de servicio, fiabilidad, eficiencia, escalabilidad, etc.). De esta forma se ha encontrado un equilibrio muy importante entre los componentes y los marcos de trabajo, haciendo descansar las propiedades funcionales de las aplicaciones sobre los primeros, y las no funcionales sobre los segundos.

8 Conclusiones

En esta lección hemos tratado de ofrecer una visión sobre lo que constituye el desarrollo de aplicaciones software basado en componentes reutilizables.

En primer lugar se han introducido las arquitecturas software y los marcos de trabajo, que intentan ofrecer soluciones de diseño desde el punto de vista estructural de las aplicaciones, y de las relaciones entre sus componentes.

A un nivel más bajo se encuentran la programación orientada a componentes, un paradigma que propugna la construcción de componentes reutilizables en entornos abiertos y distribuidos, con el objetivo de lograr un mercado global de software.

Basados en ella, los modelos y plataformas de componentes proporcionan los mecanismos adecuados para poder tratar la complejidad de los problemas que aparecen en la construcción de aplicaciones para sistemas abiertos y distribuidos, como pueden ser la heterogeneidad, la asincronía, la dispersión, fallos en la seguridad, retrasos y problemas en las comunicaciones, la composición tardía, etc.

Por último, hemos mencionado las dificultades con las que se encuentra actualmente la Ingeniería del Software para tratar de ofrecer metodologías de desarrollo de aplicaciones en estos ambientes. Si bien es

cierto que actualmente podemos hablar de "Desarrollo de Software Basado en Componentes", pensamos que aún queda cierto camino que recorrer antes de poder hablar con propiedad de una "Ingeniería del Software Basada en Componentes", capaz de ofrecer soluciones totales a los retos que plantea la construcción de grandes aplicaciones en sistemas abiertos y distribuidos.

Referencias

[Agha, 1986] Agha, G. A. (1986). *Actors: A Model of Concurrent Computation for Distributed Systems.* MIT Press.

[Aksit et al., 1993] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., y Yonezawa, A. (1993). Abstracting Object Interactions Using Composition Filters. En *Proc. of ECOOP'93*, núm. 791 de LNCS, págs. 152–184. Springer-Verlag.

[Allen y Garlan, 1994] Allen, R. y Garlan, D. (1994). Beyond Definition/Use: Architectural Interconnection. *SIGPLAN Notices*, 29(8).

[Allen y Garlan, 1997] Allen, R. y Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249.

[Arnold et al., 1996] Arnold, D. et al. (1996). Hector: Distributed Objects in Python. En *Proc. of the 4th International Python Conference*, Livermore, CA.

[Bosch, 1996] Bosch, J. (1996). Language Support for Component Communication in LayOM. En Mühlhäuser, M. (ed.), *Special Issues in Object-Oriented Programming. Workshop Reader of ECOOP'96*, págs. 131–138. Dpunkt Verlag.

[Bosch y Dittrich, 1997] Bosch, J. y Dittrich, Y. (1997). Domain-Specific Languages for a Changing World. (http://www.ide.hk-r.se/∼bosch/papers/dslinkw.ps).

[Box, 1998] Box, D. (1998). *Essential COM.* Addison-Wesley.

[Broy et al., 1998] Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M., y Szyperski, C. (1998). What characterizes a (Software) Component? *Software–Concepts and Tools*, 19:49–56.

[Bruegge et al., 1993] Bruegge, B., Gottschalk, T., y Luo, B. (1993). A Framework for Dynamic Program Analyzers. *ACM SIGPLAN Notices*, 28:65–82.

[Büchi y Weck, 1997] Büchi, M. y Weck, W. (1997). A Plea for Gray-Box Components. En *Proc. of the FSE'97 FoCBS Workshop*, págs. 39–49, Zurich.

[Calsen y Agha, 1994] Calsen, C. y Agha, G. A. (1994). Open Heterogeneous Computing in ActorSpace. *Journal of parallel and Distributed Computing*, págs. 289–300.

[Canal et al., 1997] Canal, C., Pimentel, E., y Troya, J. M. (1997). On the Composition and Extension of Software Systems. En *Proc. of the FSE'97 FoCBS Workshop*, págs. 50–59, Zurich.

[Carriero y Gelernter, 1992] Carriero, N. y Gelernter, D. (1992). Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107.

[Codenie et al., 1997] Codenie, W. et al. (1997). From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM*, 40(10):71–77.

[Cole, 1989] Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT Press.

[d'Hont et al., 1997] d'Hont, K., Lucas, C., y Steyaert, P. (1997). Reuse Contracts as Component Interface Descriptions. En *Proc. of the ECOOP'97 Workshop on Component Oriented Programming (WCOP'97)*, núm. 1357 de LNCS. Springer-Verlag.

[Ducasse y Richner, 1997] Ducasse, S. y Richner, T. (1997). Executable Connectors: Towards Reusable Design Elements. En *Proc. of ESEC'97.*

[Fayad y Schmidt, 1997] Fayad, M. E. y Schmidt, D. C. (1997). Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38.

[Florijn et al., 1997] Florijn, G., Meijers, M., y Winsen, P. V. (1997). Tool Support for Object-Oriented Patterns. En *Proc. of ECOOP'97*, núm. 1241 de LNCS, págs. 472–495. Springer-Verlag.

[Forman et al., 1995] Forman, I. et al. (1995). Release to Release Binary Compatibility in SOM. *ACM SIGPLAN Notices*, 30(10):426–438.

[Fuentes y Troya, 1999] Fuentes, L. y Troya, J. M. (1999). A Java Framework for Web-based Multimedia and Collaborative Applications. *IEEE Internet Computing*, 3(2):52–61.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., y Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Garlan et al., 1994] Garlan, D., Allen, R., y Ockerbloom, J. (1994). Exploiting Style in Architectural Design Environments. En *Proc. of SIGSOFT'94: Foundations of Software Engineering*. ACM Press.

[Garlan et al., 1995] Garlan, D., Allen, R., y Ockerbloom, J. (1995). Architectural Mismatch: Why Reuse is So Hard. *IEEE Software*, págs. 17–26.

[Garlan et al., 1997] Garlan, D., Monroe, R., y Wile, D. (1997). ACME: An Architectural Interchange Language. En *Proc. of the 19th IEEE International Conference on Software Engineering (ICSE'97)*, Boston.

[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*.

[Hüni et al., 1997] Hüni, H., Johnson, R., y Engel, R. (1997). A Framework for Network Protocol Software. *ACM SIGPLAN Notices*, 32.

[Johnson, 1997] Johnson, R. (1997). Frameworks = (Components + Patterns). *Communications of the ACM*, págs. 39–42.

[Kafura y Briot, 1998] Kafura, D. y Briot, J.-P. (1998). Actors and Agents. *IEEE Concurrency*, págs. 24–29.

[Kiczales et al., 1991] Kiczales, G., des Riviers, J., y Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

[Kiczales et al., 1997] Kiczales, G. et al. (1997). Aspect-Oriented Programming. En *Proc. of ECOOP'97*, núm. 1241 de LNCS, págs. 220–242. Springer-Verlag.

[Krieger y Adler, 1998] Krieger, D. y Adler, R. M. (1998). The Emergence of Distributed Component Platforms. *Computer Journal*, 41(3):43–53.

[Lange y Nakamura, 1995] Lange, D. B. y Nakamura, Y. (1995). Interactive Visualization of Design Patterns can help in Framework Understanding. *ACM SIGPLAN Notices*, 30(10):342–357.

[Lange y Oshima, 1997] Lange, D. B. y Oshima, M. (1997). *Programming Mobile Agents in Java –with the Java Aglet API*. IBM Research.

[Luckham et al., 1995] Luckham, D. C. et al. (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 *Proc. of the FSE'97 FoCBS Workshop*, págs. 178–187, Zurich.

[Magee et al., 1995] Magee, J., Eisenbach, S., y Kramer, J. (1995). Modeling Darwin in the π-calculus. En *Theory and Practice in Distributed Systems*, núm. 938 de LNCS, págs. 133–152. Springer-Verlag.

[Magee y Kramer, 1996] Magee, J. y Kramer, J. (1996). Dynamic Structure in Software Architectures. En *Proc. of ACM FSE'96*, págs. 3–14, San Francisco.

[Manola, 1993] Manola, F. (1993). MetaObject Protocol Concepts for a "RISC" Object Model. Informe Técnico núm. TR-0244-12-93-165, GTE Laboratories Inc.

[Matsuoka y Yonezawa, 1993] Matsuoka, S. y Yonezawa, A. (1993). Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. capítulo 4, págs. 107–150. MIT Press, Cambridge, MA.

[Mattson y Bosch, 1997] Mattson, M. y Bosch, J. (1997). Framework Composition: Problems, Causes and Solutions. En *Proc. of TOOLS USA'97*.

[Meijler y Engel, 1997] Meijler, T. D. y Engel, R. (1997). Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment. En Jazayeri, M. y Schauer, H. (eds.), *Proc. of ESEC/FSE'97*, núm. 1301 de LNCS, págs. 94–110. Springer-Verlag.

[Meseguer, 1998] Meseguer, J. (1998). Formal Interoperability. En *Proc. of the 1998 Conference on Mathematics in Artificial Intelligence*, Florida. (http://rutcor.rutgers.edu/∼amai/Proceedings.html).

[Meusel et al., 1997] Meusel, M., Czarnecki, K., y Köpf, W. (1997). A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. En *Proc. of ECOOP'97*, núm. 1241 de LNCS, págs. 498–510. Springer-Verlag.

[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. *Beyond Objects column, Software Development*, 7(11).

[Mikhajlov y Sekerinski, 1998] Mikhajlov, L. y Sekerinski, E. (1998). A Study of the Fragile Base Class Problem. En *Proc. of ECOOP'98*, núm. 1445 de LNCS, págs. 355–382. Springer-Verlag.

[Nierstrasz, 1995] Nierstrasz, O. (1995). Requirements for a Composition Language. En Ciancarini, P., Nierstrasz, O., y Yonezawa, A. (eds.), *Proc. of the ECOOP'94 Workshop on Object-Based Models and Languages for Concurrent Systems*, núm. 924 de LNCS, págs. 147–161. Springer-Verlag.

[Nierstrasz et al., 1996] Nierstrasz, O., Schneider, J.-G., y Lumpe, M. (1996). Formalizing Composable Software Systems - A Research Agenda. En Najm, E. y Stefani, J.-B. (eds.), *Proc. of FMOODS'96*, Paris. Chapman & Hall.

[Oberon Microsystems, 1997] Oberon Microsystems (1997). *BlackBox Developer and BlackBox Component Framework*. (http://www.oberon.ch).

[Odenthal y Quiebel-Cirkel, 1997] Odenthal, G. y Quiebel-Cirkel, K. (1997). Using Patterns for Design and Documentation. En *Proc. of ECOOP'97*, núm. 1241 de LNCS, págs. 511–529. Springer-Verlag.

[OMG, 1995] OMG (1995). CORBA 2.0 Interoperability. Informe Técnico núm. 95-03-10, Universal Networked Objects.

[OMG, 1999] OMG (1999). *The CORBA Component Model*. Object Management Group. http://www.omg.org.

[Pelagatti, 1993] Pelagatti, S. (1993). *A Methodology for the Development and the Support of Parallel Programs*. Tesis Doctoral, Università di Pisa-Genova-Udine.

[Pfister, 1996] Pfister, G. (1996). *In Search of Clusters: the Coming Battle of Lowly Parallel Computing*. Prentice-Hall.

[RM-ODP, 1997] RM-ODP (1997). Reference Model for Open Distributed Processing. Rec. ISO/IEC 10746-1 to 10746-4, ITU-T X.901 to X.904, ISO/ITU-T.

[Robertson y Robertson, 1999] Robertson, S. y Robertson, J. (1999). *Mastering the Requirement Process*. Addison-Wesley.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., y Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

[Schmidt, 1994] Schmidt, D. C. (1994). The ADAPTIVE Communication Environment. En *Proc. of the 12th Sun User Group Conference*, San Francisco, California.

[Schmidt, 1997a] Schmidt, D. C. (1997a). Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software. En Salus, P. (ed.), *Handbook of Programming Languages*, volumen 1. MacMillan Computer Publishing.

[Schmidt, 1997b] Schmidt, D. C. (1997b). Systematic Framework Design by Generalization. *Communications of the ACM*, 40(10):48–51.

[Schmidt, 1995] Schmidt, H. (1995). Creating the Architecture of a Manufacturing Framework by Design Patterns. *ACM SIGPLAN Notices*, 30(10):370–384.

[Shaw et al., 1995] Shaw, M. et al. (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 of an Emerging Discipline*. Prentice Hall.

[Siegel, 2000] Siegel, J. (2000). *CORBA 3. Fundamentals and Programming*. John Wiley & Sons. OMG Press.

[Sparks et al., 1996] Sparks, S., Benner, K., y Faris, C. (1996). Managing Object-Oriented Framework Reuse. *Computer Journal*, 39(9):52–61.

[Steyaert et al., 1996] Steyaert, P., Lucas, C., Mens, K., y d'Hondt, K. (1996). Reuse Contracts: Managing the Evolution of Reusable Assets. *ACM SIGPLAN Notices*, 31(10):268–285.

[Szyperski, 1996] Szyperski, C. (1996). Independently Extensible Systems —Software Engineering Potential and Challenges—. En *Proc. of the 19th Australasian Computer Science Conference*, Melbourne.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley.

[Szyperski, 2000] Szyperski, C. (2000). Component versus Objects. *ObjectiveView*, 1(5):8–16.

[Szyperski y Gough, 1995] Szyperski, C. y Gough, J. (1995). The Role of Programming Languages in the Lifecycle of Safe Systems. En *Proc. of the International Conference on Safety Through Quality (STQ95)*, págs. 99–114, Cape Canaveral, Florida.

[Szyperski y Pfister, 1997] Szyperski, C. y Pfister, C. (1997). Summary of the Workshop on Component Oriented Programming (WCOP'96). En Mühlhäuser, M. (ed.), *Special Issues in Object-Oriented Programming. Workshop Reader of ECOOP'96*. Dpunkt Verlag.

[Tanir, 1996] Tanir, O. (1996). *Modelling Complex Computer and Communication Systems: A Domain-Oriented Design Framework*. McGraw-Hill.

[Taylor et al., 1996] Taylor, R. et al. (1996). A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406.

[TINA-C, 1995] TINA-C (1995). *Overall Concepts and Principles of TINA*. (http://www.tinac.com/95/file.list.html).

[Weck et al., 1997] Weck, W., Bosch, J., y Szyperski, C. (1997). Summary of WCOP'97. En *Proc. of the ECOOP'97 Workshop on Component Oriented Programming (WCOP'97)*, núm. 1357 de LNCS. Springer-Verlag.

[Wegner, 1996] Wegner, P. (1996). Interoperability. *ACM Computing Surveys*, 28(1):285–287.

[Yellin y Strom, 1997] Yellin, D. M. y Strom, R. E. (1997). Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333.