Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Tesis de Maestría
en Ingeniería en Computación

Desarrollo basado en componentes para sistemas de gestión empresarial

Manuel Gayoso

Directora de Tesis: M. Sc. María E. Urquhart

Diciembre de 2002

Agradecimientos.

En primer término quiero agradecer a mi esposa Magdalena y mis hijos Julia, Elena, Joaquín y Martín por soportar mis largas ausencias, no solo durante la elaboración de este trabajo sino además mientras realizaba los cursos nocturnos que me privaron de estar a su lado. Confío poder, a mi turno, respaldarlos de la misma manera que hicieron ellos.

Mi agradecimiento especial para quienes me guiaron en este trabajo, para María Urquhart, quien, a pesar de no tratarse de un tema de su especialidad, dedicó un gran esfuerzo en interpretar mis ideas y me obligó a formalizarlas y a llegar a una expresión más coherente de las mismas y para Andrés Vignaga, quien me aportó mucho material y referencias acerca del tema, pero cuyo análisis y discusión del trabajo fueron mucho más valiosos, pues me permitieron clarificar y mejorar los conceptos y la manera de expresarlos.

Jorge Abin colaboró con referencias e información valiosa para mi trabajo y Gustavo Guido fue de gran ayuda para la comprensión de varios detalles prácticos de implementación; para ellos va también mi agradecimiento.

Y finalmente, pero no menos sentido, mi profundo agradecimiento a Nora Szasz, quien con su tesón y esfuerzo empujó y dio ánimo, no solo a mí sino a toda la generación, para que cumpliéramos con los requisitos curriculares y para que llegáramos a la culminación de nuestros respectivos trabajos.

Desarrollo basado en componentes para sistemas de gestión empresarial

Manuel Gayoso

Directora de Tesis: M. Sc. María E. Urquhart

Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo - Uruguay

Resumen.

Las prácticas de reingeniería de procesos de negocio, en particular cuando éstas se llevan adelante mediante planes de mejora continua, hacen necesario que los sistemas informáticos de información y gestión de las empresas puedan reaccionar rápidamente a los cambios, incorporación y eliminación de procesos y modificaciones en las reglas de negocio. Los sistemas tradicionales, monolíticos, no dan una respuesta adecuada a esta demanda. En los últimos años ha cobrado fuerza el paradigma de construcción de sistemas en base a componentes (Component Based Development, CBD) y a pesar de que aún no ha alcanzado su madurez, se lo ve como una solución al problema planteado, debido a sus características de adaptabilidad, flexibilidad y a la posibilidad de composición en forma dinámica de las soluciones. En este trabajo se realiza un estudio de los aspectos conceptuales y del estado actual de las metodologías de diseño y construcción de sistemas de este paradigma, los principios de diseño en que se basa y ejemplos de propuestas concretas. También se estudia el impacto en el ciclo de desarrollo de software y las características y realidad de un promocionado mercado de componentes que debiera jugar un rol de primer orden en la construcción de los sistemas basados en componentes. En particular se extraen conclusiones acerca de las posibilidades de aplicación para construir y modificar dinámicamente los sistemas informáticos que respalden los procesos de negocio y de que manera esto puede impactar en la planificación de la gestión de procesos de la empresa. Ante la carencia de un "diseño dominante" en el área, lo que dificulta la formación de un mercado de componentes de negocio, se propone el desarrollo de líneas de productos (familias de sistemas interrelacionados que comparten un conjunto importante de componentes), componibles y configurables, como forma de acompañar los cambios propuestos por la reingeniería de procesos de negocio y la mejora continua.

Contenido.

Ilustraciones.

Tablas.

1. Introducción.

Desde la década de los noventa, los analistas y consultores empresariales han descubierto la importancia de la (re)ingeniería de procesos de negocio (Business Process Reengineering, BPR) como herramienta para el mejoramiento de los procesos de la empresa. Se han hecho grandes esfuerzos para crear modelos que permitan analizar estos procesos y actuar sobre ellos de modo de mejorarlos y adecuarlos a las nuevas realidades que se plantean constantemente, estos esfuerzos trascienden el área de los procesos llegando inclusive a revisar los productos y objetivos mismos de las empresas. Una de estas concepciones es la de Negocios Basados en Componentes, que permite descomponer las distintas partes de un negocio en los procesos y entidades que lo forman y los vínculos entre cada una de esas partes [Veryard01].

Basándose en esta teoría los procesos empresariales resultan de la composición de diversos procesos de menor porte los cuales deben *articular* entre sí. Los procesos pueden involucrar diversas organizaciones (por ejemplo: cadena de suministros) o pueden promover el consorcio de organizaciones para un negocio común. Pueden involucrar recursos exclusivos de una organización o darse la situación en que una empresa provea la realización de ciertos procesos a otra (por ejemplo: producción tercerizada). En todos estos casos debe existir un mecanismo de articulación entre los subprocesos que forman un proceso principal y entre estos y los procesos de control requeridos.

En la visión de Negocios Basados en Componentes, quien establece las políticas y estrategias de la empresa podría resolver cambiar un proceso entero por otro, o decidir que sea realizado por terceros o unirse a otra organización combinando determinados procesos de cada una para lograr un negocio nuevo en común o disolver ese consorcio cuando no sea más conveniente. En todos los casos debe poder hacerlo “agregando”, “modificando” o “quitando” componentes. En todos esos casos se dice que está *componiendo* el negocio.

Cualquier concepción moderna de procesos empresariales va a requerir un soporte informático para llevarlos adelante. Hoy por hoy la mayoría de las organizaciones tienen sistemas informáticos en los que basan su gestión y toma de decisiones; estos sistemas incorporan explícita o implícitamente la base de conocimientos acerca de los procesos empresariales.

En general, los sistemas de gestión e información empresarial tienen una estructura monolítica y son de carácter propietario, frecuentemente las reglas y procesos del negocio están incluidos en el código y a lo sumo se cuenta con algún modelo de alto nivel que los describe [Kim99]. El esfuerzo que implica la modificación de estos sistemas y muchas veces la falta de documentación de los mismos, actúan como un freno a la implementación de nuevos procesos de la empresa, que puedan evolucionar rápidamente o interactuar con procesos de otras compañías.

La carencia de un software que pueda dar respuesta rápida a la incorporación de nuevos procesos o negocios representa una traba para la composición dinámica de los negocios, ya que se deberá esperar que los departamentos de informática realicen las modificaciones pertinentes en los programas o que el proveedor del sistema de gestión de la empresa ofrezca una solución aceptable. Una vez hecho eso, en general se debe pasar un período de inestabilidad de los sistemas, provocado por la magnitud de los cambios efectuados.

Muchas veces la necesidad de cambios se impone por cuestiones totalmente externas a las empresas; recientemente en nuestro país, por ejemplo, la liberalización de la cotización del dólar produjo que esa moneda dejara de tener el valor de referencia que había tenido por más de quince años, muchas empresas vieron entonces la necesidad de incorporar rápidamente mecanismos de ajuste por inflación a sus sistemas de información como forma de establecer una unidad de valor estable en el tiempo. La incorporación de estos mecanismos, que se integran

estrechamente con los procesos contables normales, no es fácil de llevar a cabo en sistemas monolíticos que no tengan capacidad de extensión.

Se hace necesario que los sistemas puedan incorporar rápidamente los cambios y nuevas formas de negocio y que queden abiertos a la evolución futura. La simple sustitución de un sistema legado por otro sistema de concepción monolítica que implemente los nuevos procesos, traerá como consecuencia que, algunos años después, el nuevo sistema se convierta, a su vez, en un sistema legado, quedando la empresa expuesta al mismo problema. Se hace necesario además que los conceptos que se manejan por los profesionales de informática sean los mismos que manejan quienes están a cargo de la dirección estratégica de la empresa, de manera de reducir la diferencia cultural entre ambos sectores y poder establecer planes comunes para implementar los cambios necesarios.

Por el lado de los profesionales de tecnología informática y principalmente del sector que impulsa las tecnologías orientadas a objeto se ha puesto mucho énfasis en diferentes modelos de conceptualización de las aplicaciones, con una fuerte vinculación con las entidades y procesos mismos de los negocios.

Se han estudiado y definido los Componentes de Negocio como correlato informático a los conceptos del negocio [Herzum98] [Sims98], se han desarrollado metodologías específicas para el análisis y el diseño de sistemas, que aplicadas al dominio de la gestión de empresas, mantienen los conceptos de negocio a lo largo de todo el proceso del software. Ha emergido además un nuevo paradigma de desarrollo de sistemas: los sistemas basados en componentes respaldados en metodologías de diseño y desarrollo como *Catalysis* [D'Souza98] o *RUP (Rational Unified Process)* [Rational02] y extensiones como la propuesta en [Levi02].

El desarrollo de componentes de negocio, como conceptos independientes de una aplicación en particular, origina componentes generales para un dominio, por lo cual se prevé y en algunas áreas ya se verifica, el surgimiento de un "mercado" de componentes de software, seleccionables y reemplazables, a partir de los cuales se puedan construir los sistemas de las organizaciones [Sparling00b] [Wilkes02].

Por otro lado, los fabricantes de los grandes sistemas estándar de gestión de empresas, los *Enterprise Resource Planning (ERP)*, han reaccionado a esta nueva corriente, dejando de lado sus estructuras iniciales monolíticas y propietarias, para admitir la conexión y cooperación con componentes de terceros y en algunos casos hasta se han re-programado enteramente en base a componentes [Kim99] [Kumar00] [Sprott00] [SAP02].

En otras palabras, hay un movimiento general, respaldado además por la evolución de las distintas tecnologías de interoperabilidad, hacia la construcción de sistemas basados en diferentes componentes independientes, que representan funcionalidades o conceptos propios del dominio de cada aplicación. Esos componentes independientes pueden implementar soluciones generales para distintos requerimientos de un dominio y por lo tanto poder ser usados para componer más de un sistema.

El esquema monolítico de sistemas de aplicación cede paso a la concepción de sistemas basados en componentes.

En este trabajo se busca determinar hasta que punto los sistemas basados en componentes pueden ser la solución requerida por quienes establecen las estrategias empresariales, en el sentido que les permita modelar y componer dinámicamente las aplicaciones de software para gestionar y llevar adelante los procesos de la organización, con la misma flexibilidad y adaptabilidad con que pueden ser diseñados usando concepciones como la referido [Veryard01].

Para esto, se hace una revisión del estado del arte actual del desarrollo de sistemas basados en componentes (Component Based Development, CBD), tanto en lo referente a conceptos generales como a aspectos de diseño, arquitecturas y ejemplos de realizaciones concretas Estas son evaluadas, en base a principios aceptados de diseño y en base al cumplimiento de las necesidades planteadas en el área de los sistemas de gestión de empresas.

Se estudia también el impacto que produce este paradigma de desarrollo de sistemas en el proceso del software y en particular una de las estrategias de quienes fabrican software de aplicación, que es la de la elaboración de líneas de producto para un dominio, ya que éstas pueden crear el marco más adecuado para el re-uso planificado de componentes de software y para los mecanismos de composición de aplicaciones específicas y su evolución de acuerdo a cambios que se verifiquen en las empresas.

Las líneas de productos son familias de sistemas de un dominio determinado, que tienen en común un conjunto de requerimientos y comparten una arquitectura y componentes de software, pero al mismo tiempo muestran variantes lo suficientemente significativas para diferenciarlos [Griss01][Bosch00][Batory99]. Ante la carencia de un "diseño dominante" en el área de los sistemas de gestión e información empresarial y el desarrollo todavía no verificado de un mercado de componentes de negocio en este dominio, el desarrollo de líneas de productos, aunque propietarias, parece el mecanismo más plausible para lograr los objetivos señalados ya que puede hacer las veces de diseño dominante donde se conectan los componentes elegidos, a la vez que provee esas opciones de componentes, sustituyendo un mercado que todavía no ha emergido.

El trabajo se desarrolla de la siguiente manera:

En la sección 2 se realiza una puesta al día de conceptos y definiciones de componentes y sus diferentes clasificaciones, se estudian además las diferentes construcciones a partir de componentes: sistemas abiertos y sistemas construidos en base a componentes de software y la formación de líneas de productos. Como parte del estado del arte se reseña la evolución de los grandes sistemas ERP (*Enterprise Resource Planning)*, hacia construcciones en base a componentes.

En la sección 3 se estudia el proceso de construcción de sistemas basados en componentes, desde el punto de vista de la ingeniería de software: qué nuevas ideas o procesos aparecen con este paradigma y cómo impacta en el proceso de desarrollo de sistemas, así como las ventajas y desventajas de su aplicación. Como elementos determinantes en el paradigma de desarrollo de sistemas en base a componentes, se estudia en particular, en la sub-sección 3.5 como se constituye, las características que tiene y lo que representa un "*mercado de componentes de software"* y en la sub-sección 3.6, el impacto en el proceso de software de la estrategia de desarrollo de líneas de producto, como alternativa a un mercado aún no establecido.

En la sección 4 se estudia en forma más detallada los aspectos vinculados con el desarrollo de componentes y de sistemas basados en componentes, rasgos y objetivos fundamentales, principios de diseño que guían ese desarrollo y ejemplos concretos de arquitecturas propuestos por diversos autores, la evaluación y la comparación de las mismas en base a los principios de diseño.

Por último en la sección 5 se analizan los distintos aspectos estudiados y se extraen algunas conclusiones y enseñanzas.

El trabajo se complementa con un glosario de los términos más importantes usados y de los significados de las siglas que aparecen en el mismo. Se incluye además la bibliografía de referencia y anexos aclaratorios de algunos detalles particulares.

2. Componentes y sistemas basados en componentes.

En esta sección se estudia el concepto de componente de software, los orígenes del mismo y definiciones académicas y de la industria, diferentes clasificaciones de componentes de acuerdo a variados criterios y las construcciones posibles: sistemas abiertos y sistemas basados en componentes y líneas de productos. Se desarrolla en particular la evolución que han tenido desde finales de los noventa los grandes sistemas de gestión e información de empresas, los ERP, hacia estructuras basadas en componentes y a la constitución de sistemas abiertos admitiendo extensión mediante desarrollos de componentes de terceros..

Los conceptos establecidos en esta sección son la base para el desarrollo posterior del trabajo, que en las siguientes secciones se interna en el estudio de la construcción de sistemas basados en componentes y de temáticas afines como es la de los mercados de componentes.

2.1. Conceptos de componentes de software.

Una característica de la mayoría de las disciplinas de la ingeniería es que los productos que se elaboran son realizados mediante conexión, unión o colaboración de partes ya existentes.

Esas partes cumplen roles preestablecidos dentro de un dominio, de acuerdo a conceptos que son compartidos por quienes tienen que ver con ese contexto. Incluso cuando una parte debe ser adaptada para un caso específico, el *tipo* de parte es un concepto claro y compartido: por ejemplo todos los automóviles usan ruedas y aunque las formas y tamaños y hasta la cantidad de las mismas sean específicas de cada marca o modelo, hay un tipo de parte que es “la rueda”. Un diseñador de automóviles puede optar por elegir un tipo de rueda ya existente en el mercado, logrando significativo ahorro en diseño y producción o diseñar una rueda de dimensiones especiales basándose en otros elementos de su consideración (estéticos, prestaciones, etc.).

Esos roles preestablecidos de cada parte están expresados a través de “diseños dominantes” en las distintas áreas, por ejemplo los motores de explosión o la arquitectura de un PC.

En software es difícil verificar este concepto: la mayoría de los desarrolladores crean nuevos sistemas de la nada, muchas veces sin siquiera haber visto o estudiado desarrollos similares dentro del dominio. Dado un tema de aplicación, es muy probable que existan programas o sistemas hoy en día, que cumplan con varios de los requerimientos propios del dominio de la aplicación, sin embargo quienes tienen que hacer el desarrollo probablemente ni siquiera conocen de su existencia y difícilmente los puedan usar. Esta es una de las razones por las que el desarrollo de software lleva tanto tiempo.

La posibilidad de composición es una de las características requeridas para un nuevo enfoque de desarrollo de sistemas, pero no es la única que promueve el desarrollo basado en componentes (Component Based Development, CBD). La posibilidad de crear y utilizar partes de programas que contribuyan a la solución de aspectos generales o particulares de un proyecto de software y luego puedan ser usadas en otro desarrollo diferente (dentro del mismo domino o no) ha sido y es una de las aspiraciones más frecuentes entre quienes producen software o desarrollan sistemas informáticos.

Por ejemplo, como muestra la Figura 1, en un esquema muy frecuente para sistemas de información empresarial, cuando se usan los sistemas operativos para soportar las aplicaciones o cuando se usan sistemas administradores de bases de datos (DBMS) para asegurar la persistencia y el acceso a la información, se están usando componentes que proveen servicios generales y repetitivos para casi cualquier tipo de aplicación. Más recientemente la aparición de

componentes de presentación GUI (Graphic User Interface) estándar constituye un indicio más palpable del uso de componentes.

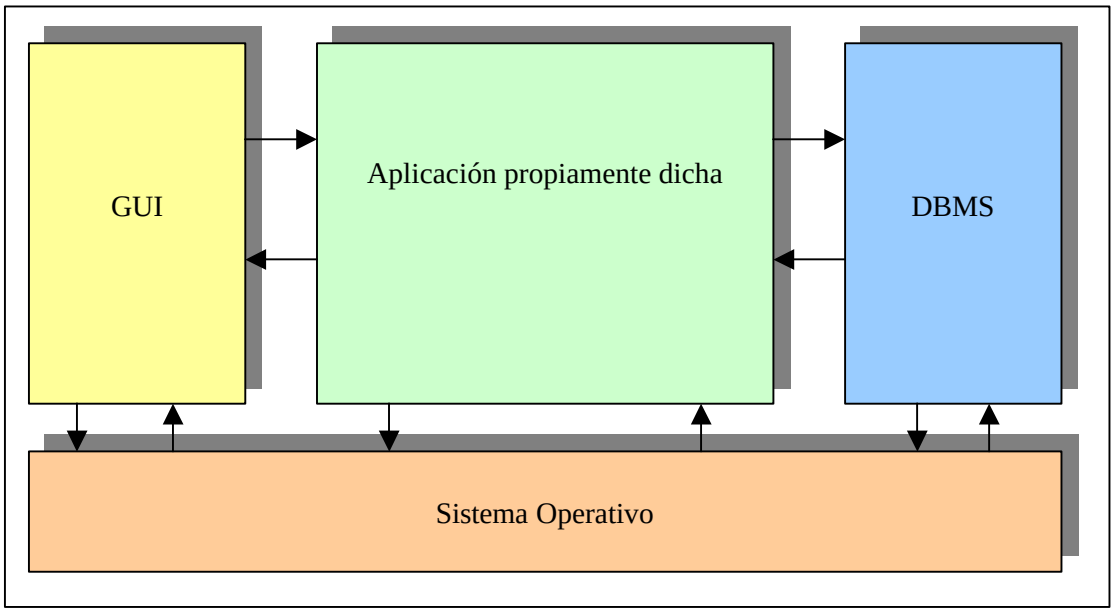

Figura 1. Los componentes básicos de los sistemas.

La intención de usar componentes o partes prehechas o prediseñadas como forma de acortar el ciclo de desarrollo del software no es nueva, más allá de ser una aspiración permanente de quines desarrollan software, hay una referencia formal de la palabra “componente”, en el sentido que se le da en este trabajo, en una conferencia científica de la OTAN de 1968, dónde se identificó el problema del costo del software, se definieron características para definir familias de componentes y hasta se estudiaron las características que debiera tener un mercado de componentes y el esfuerzo gubernamental que se debería realizar para hacerlo posible [Mc Ilroy68]. Asimismo, la descomposición de sistemas en módulos que “esconden” las decisiones de diseño, se puede encontrar en trabajos de principio de los setenta [Parnas72] y constituyen la base conceptual que permite la descomposición de sistemas en partes independientes entre sí.

Sin embargo para que el concepto cristalizara en el potencial que hoy tiene, se debió esperar a la adopción cada vez más generalizada de las tecnologías de orientación a objetos y en particular al desarrollo de mecanismos de interoperabilidad que se convirtieron en estándares de facto (COM, CORBA, Java/RMI). La preocupación por hacer explícita la arquitectura de los sistemas, constatada en la década de los noventa y la aparición de lenguajes para describirla contribuyeron a la formalización del desarrollo de sistemas basados en componentes.

El desarrollo de Internet, por su parte impuso la necesidad de creación de sistemas abiertos y posibilitó la interacción entre procesos de organizaciones distantes.

Aunque se promociona la independencia de lenguaje y de hecho un componente de software hasta podría estar escrito en un lenguaje procedural, dada la gran cantidad de conceptos compartidos, de dice que la noción de “componente” es un refinamiento de los conceptos que se han instruido como parte de la adopción de tecnologías orientadas a objetos [Hopkins00], los componentes pueden ser considerados como una extensión natural del modelo de objetos [Meyer99].

Varios autores han definido a su manera los componentes de software, de acuerdo a sus necesidades específicas para diferentes contextos. Esto hace que se manejen una gran cantidad

de definiciones, a pesar de lo cual se puede encontrar muchas coincidencias entre ellas, que permiten establecer el concepto usado en el desarrollo de este trabajo.

Un componente es:

- Según Grady Booch[1], *una parte física y reemplazable de un sistema, que conforma y provee la realización de un conjunto de interfaces. Representa el empaquetamiento físico de elementos que de otra forma serían abstracciones lógicas, tales como clases, interfaces y colaboraciones* [Booch99].

- Según Richard Veryard, *una relación dinámica de empaquetamiento entre un conjunto de servicios y un conjunto de capacidades (posiblemente embebidas en uno o más dispositivos)* [Veryard01].

- Para Microsoft, *una unidad de empaquetamiento, distribución o envío, que provee servicios sobre la base del encapsulamiento y la integridad de datos* [Sparling00a].

- Para Michael Sparling[2], *un paquete de servicios de software, implementados independientemente y con neutralidad de lenguaje, distribuidos en un contenedor reemplazable que los encapsula y que son accesibles por vía de una o más interfaces públicas* [Sparling00a].

- Según Clemens Szyperski[3], *una unidad binaria de producción, adquisición y despliegue independiente, que interactúa con otras para formar un sistema que funciona* [Szyperski97].

- En la versión de Jim Ning[4], *una pieza de software, encapsulada, distribuible y ejecutable que provee y recibe servicios a través de interfaces bien definidas* [Ning99].

Lo primero que surge en cualquier definición es que se trata de un elemento físico, tangible y no una abstracción o un patrón. En el entorno de este trabajo, cuando se habla de componentes se está hablando de piezas de software ejecutable (EXE, DLL, etc.), que cumplen con determinada funcionalidad en un contexto dado, más allá de que para otros autores el concepto se generaliza a diferentes etapas del ciclo de desarrollo y a diferentes tipos de objetos (documentos, diagramas UML, etc.) o a conceptos de análisis y diseño de la solución que luego se materializarán en varios ejecutables [Herzum98] [Sims98].

Otro concepto que se encuentra en forma reiterada en las definiciones es el de la provisión de servicios, dando a entender que el componente forma parte (compone) una solución general, aportando ciertos aspectos de procesamiento en los cuales se especializa.

El tercer elemento que aparece en las definiciones es el de interfaces. Interfaz se puede definir como un conjunto de servicios provistos o requeridos por cualquier componente que soporte ese conjunto[Allen01] o como una abstracción de todas las implementaciones posibles que pueden cumplir un cierto rol en el sistema compuesto [Fröhlich01]. De una manera u otra los conceptos de componente y de interfaz van de la mano ya que este último expresa básicamente cual es la funcionalidad que ofrece un componente o cual es la que requiere para su propio funcionamiento y actúa como medio de comunicación entre componentes.

---

[1] Grady Booch es uno de los autores de UML (Unified Modeling Language) y cofundador de Rational Software Inc.
[2] Michael Sparling es director de Tecnología en Castek, Canadá, una consultora en ingeniería de software
[3] Clemens Szyperski es Profesor Agregado en la Fac. de Tecnología de Información de la Universidad Queensland en Australia y es autor de un libro fundamental en el tema de componentes [Szyperki97]
[4] De Arthur Anderson Consulting, en el marco de la 21st. International Conference on Software Engineering, 1999

Un componente puede ofrecer una cierta funcionalidad, en cuyo caso se define una *interfaz provista* o *exportada*, que representa los servicios soportados o que brinda el componente.

Asimismo, un componente puede requerir cierta funcionalidad del entorno (otro componente asociado) y en ese caso se denomina *interfaz requerida* o *importada,* que representa los servicios que deben ser recibidos por el componente para poder implementar sus propias operaciones [Ning99].

Hay dos propiedades muy importantes inherentes al concepto de componente: una de ellas que se desprende de la propia definición es el *encapsulamiento de la implementación,* es decir que la provisión de servicios se realiza sin que el solicitante tenga que conocer como es que estos se llevan a cabo y la otra no tan evidente es la posibilidad de *re-uso*, dado que el componente implementa una interfaz, cualquiera sea el sistema o componente que solicite la realización de la misma, obtendrá el servicio provisto y por tanto el componente puede ser re-usado en diferentes contextos que lo requieran.

Bertrand Meyer[5] establece siete criterios que ayudan a definir lo que es un componente en el ambiente de un sistema y las cualidades que le dan valor en el marco de un proyecto de CBD (Component Based Development) [Meyer00]:

1. *Puede ser usado por otros elementos de software (clientes).*
2. *Puede ser usado por clientes sin la intervención del desarrollador.*
3. *Incluye una especificación de todas sus dependencias (plataformas de hardware y software, versiones, otros componentes).*
4. *Incluye una especificación precisa de todas las funcionalidades que ofrece.*
5. *Puede ser usado basándose solamente en esa especificación.*
6. *Se puede componer con otros componentes.*
7. *Puede ser integrado en un sistema en forma rápida y sin dificultad.*

El primer criterio diferencia los componentes de los programas de aplicación: estos no están pensados para ser usados por otros programas sino por personas. El segundo criterio refiere al encapsulamiento de la implementación. El tercero subraya que el componente no tiene sentido como cosa aislada sino que va a tener ciertas dependencias que deben ser especificadas. En los criterios cuatro y cinco se refiere a la especificación de interfaces del componente. En el criterio seis, se resalta la idea de que usualmente el componente va a formar parte de una estructura más amplia, más general, probablemente con una arquitectura claramente identificada. Finalmente, Meyer observa que todo esto sirve si su uso en la construcción de un sistema es sencillo y sin trastornos.

Desde otro punto de vista, el Software Engineering Institute de Carnegie Mellon University (SEI-CMU) [Oberndorf97], establece el concepto de COTS (Commercial Off The Shelf) relativo a componentes de software como *cosas que uno puede comprar, ya hechas, del estante virtual de algún fabricante. Va de la mano con el concepto de obtener, a un precio razonable, algo que ya hace el trabajo. Reemplaza la pesadilla de desarrollar componentes únicos de un sistema por la promesa de una rápida y eficiente adquisición de implementaciones de componentes baratos (o al menos más baratos).Las características más resaltadas de un COTS son:*

- *Existe a priori*
- *Está disponible al público interesado*
- *Puede ser comprado*

[5] Bertrand Meyer es el autor del lenguaje de programación orientado a objetos Eiffel y fundador de ISE Inc.

Esta visión del SEI-CMU introduce uno de los aspectos sobresalientes de los componentes y del desarrollo de sistemas basados en componentes: la posibilidad de comprar partes funcionales prehechas del sistema en vez de tener que desarrollar todas las partes. Efectivamente, la posibilidad de producción independiente de componentes de software para un dominio, aparece como una oportunidad de negocio para quienes se pueden especializar en ciertos aspectos y luego ofrecer esas funcionalidades específicas a quienes desarrollan o construyen sistemas de aplicación.

El SEI hace especial hincapié en los aspectos de mercado, accesibilidad y disponibilidad pública que deben tener los componentes. Esta disponibilidad es la que hace posible la composición sin necesidad de programar la funcionalidad específica y por tanto favorece el re-uso de los componentes. La disponibilidad no debe entenderse exclusivamente en el sentido de compra de los componentes, sino que puede estar circunscrita al ámbito de una sola organización, por ejemplo en una empresa de desarrollo de sistemas, que re-usa sus artefactos de software en el marco de uno o de varios proyectos [Bosch00].

En síntesis quedan establecidos los aspectos sobresalientes de los componentes:

- Son elementos de software ejecutable que son usados por uno o más programas a los cuales les aportan determinados servicios.
- Pueden ser re-usados por diferentes programas o en diferentes contextos
- Definen su funcionalidad y requerimientos a través de interfaces y especifican su contexto de aplicación y dependencias.
- Encapsulan (esconden) la implementación de la funcionalidad ofrecida.
- Pueden existir con anterioridad a las necesidades concretas de una aplicación y pueden ser obtenidos mediante adquisición a terceras partes.

Y una definición para usar en este trabajo:

Los componentes son unidades binarias reemplazables de despliegue, que implementan una o más interfaces bien definidas, dando acceso a un conjunto de funcionalidades interrelacionadas y que puedan adaptar su comportamiento en forma predefinida.

2.2. Diferentes clasificaciones de componentes.

En los trabajos de diversos autores referidos a componentes, estos son estudiados desde distintos puntos de vista, como son la facilidad de uso y adaptación, el tipo de funcionalidad que soportan o el ambiente en que pueden utilizarse. Aparecen adjetivaciones que de alguna manera clasifican o categorizan los componentes de acuerdo a los diferentes rasgos de estudio. Estas clasificaciones son independientes entre sí y se relacionan exclusivamente con el aspecto en particular con el que se quiere describir a cada componente.

En esta sub-sección se enumeran algunos de esos rasgos y las clasificaciones frecuentes en los mismos.

2.2.1. Adaptabilidad de los componentes.

Cada componente se fabrica para cumplir con requerimientos específicos y basándose en diferentes presunciones acerca del contexto. Podría ocurrir que para su uso en un caso particular fuera necesario realizar ciertas adaptaciones. Teniendo en cuenta el grado o necesidad de acceso

al código para realizar estas adaptaciones, los componentes pueden clasificarse según [Haines97] en:

- *Caja blanca ("White box"):* son aquellos componentes para los cuales se tiene acceso completo al código (se vende el código fuente), para que sea posible adaptarlos para trabajar en conjunto con otros componentes o modificar o ampliar sus funciones.
- *Caja gris ("Grey box"):* son aquellos componentes para los cuales no se puede acceder al código fuente, pero que proveen un lenguaje de extensión propio o APIs (Application Programming Interface) adecuadas.
- *Caja negra ("Black box"):* son aquellos componentes para los cuales solo hay disponible un ejecutable y no existe un lenguaje de extensión ni APIs, como resultado de lo cual deberá usarse tal como fue desarrollado originalmente.

Cada una de estos tipos tienen sus ventajas y desventajas, por ejemplo los componentes white box presentan una altísima adaptabilidad, sin embargo debido a la modificación del código fuente original, pueden traer tremendos problemas futuros y normalmente se pierde acceso a nuevas versiones del componente.

Hay técnicas y patrones específicos de programación [Haines97] [Gamma95] para lograr la adaptabilidad de los componentes grey box y black box:

- *Wrapping* es una técnica que agrega un nivel de indirección, haciendo que el componente se comunique exclusivamente con una pieza de software (*adaptador o wrapper)*, que a su vez conoce el contexto y puede comunicarse con él. Cambiando el wrapper se puede adaptar el mismo componente a diferentes contextos, sin necesidad de que los conozca.

- *Bridging* es una técnica que separa un componente en una parte que representa las abstracciones y otra que representa la implementación de éstas para contextos específicos. Por ejemplo en un componente para manejar diferentes presentaciones GUI se separa la abstracción de una "ventana" y sus diferentes elementos de la implementación de los mismos para una plataforma específica. Mediante un mecanismo de delegación el componente que representa la abstracción invoca al componente implementador. Esta técnica se puede aplicar si en el momento del desarrollo del componente fue previsto este mecanismo (grey box). En el futuro, el componente implementador puede ser extendido para abarcar otras especificaciones de contexto.

- *Mediating* es una técnica que evita el acoplamiento de diversos componentes que deben trabajar en una colaboración, haciendo que esos componentes se comuniquen con el *mediador* que conoce las diferentes interfaces y puede manejar la colaboración sin necesidad que cada componente conozca a los demás. Esta técnica permite adaptar los componentes en cada colaboración particular.

2.2.2. El rol de los componentes en los sistemas de aplicación.

De acuerdo al rol que ocupan dentro de los sistemas de aplicación también tendremos otras clasificaciones, por ejemplo en sistemas de información empresarial frecuentemente se pueden identificar [Allen01]:

- *Componentes GUI* - tales como botones, list boxes, imágenes, etc., en general independientes de cualquier aplicación en particular, proveen sin embargo un valor muy importante en lo que tiene que ver con la estandarización de interfaz con el usuario.

- *Componentes utilitarios* - usados para proveer servicios del sistema y de infraestructura, son típicamente conectividad a bases de datos, manejo de memoria, etc.
- *Componentes de negocio* – que encapsulan características de una aplicación en particular, tales como la lógica del negocio, los procesos del negocio, las reglas del negocio y la información del negocio. El gran potencial de los componentes de negocio es el conocimiento de la aplicación que encierra su desarrollo, son asimismo una vía de transmitir ese conocimiento.

Oliver Sims[6] refiriéndose a un componente de negocio (business component), dice que *es la realización de un concepto de negocio. Una construcción de análisis, diseño, elaboración, empaquetado y ejecución* [Sims98]. Esta definición de componente es más amplia que la que se usa en este trabajo ya que incluye aspectos de análisis y diseño, anteriores a un ejecutable.

Sin embargo se mantiene como característica del componente de negocios el ser una cosa física, tangible y en su última instancia puede ser un ejecutable (dll, exe, etc.). Puede ser el caso que un componente de negocio represente más de un concepto del negocio, por ejemplo un componente que maneje información acerca de empresas puede representar el concepto *cliente* y el concepto *proveedor*. Por otro lado puede requerirse más de un componente de software para representar un concepto del negocio, esto se ve, por ejemplo al implementar mediante el uso del patrón *model-view-controller* [Gamma95], dónde un componente se ocupa de los aspectos de presentación de una entidad, mientras que otro se encarga de las reglas (lógica) de comportamiento.

Dentro de los componentes de negocio se diferencian tres tipos [Sims98] [Herzum98]:

- los componentes que representan *procesos del negocio* como cadena de suministro, proceso de venta, etc.
- los componentes que representan *entidades del negocio* como cliente, vendedor, factura, etc.
- los componentes que representan *utilidades auxiliares*, tales como generadores de números de secuencia, validación y manejo de fechas y plazos, etc.

2.2.3. El modelo de componentes.

En función del modelo de componente utilizado, es decir con relación al software que habilita la interoperabilidad de los componentes, hay tres estándares de componentes fundamentales:

- CCM (CORBA Component Model, de OMG).
- COM (Component Object Model, de Microsoft), un subgrupo lo forman los componentes ActiveX (ocx), formados básicamente por controles GUI.
- JB Y EJB (Java Beans y Enterprise Java Beans, de Sun).

Esta clasificación se vuelve de especial importancia a la hora de elegir un componente en función de la plataforma CEE (Component Execution Environment) elegida para la aplicación. No obstante, por un lado los modelos y arquitecturas de componentes han evolucionado hacia la compatibilidad entre las diferentes propuestas [Sun99][Sun02] y por otro lado hay descritos varios wrappers (adaptadores) que permiten utilizar componentes de un modelo dado en otro modelo de CEE [Sun02].

[6] Oliver Sims, conocido como “padre de los Business Objects” es miembro de OMG e integra el OMG´s Architecture Board y el OMG´s Business Object Task Force.

2.3. Usos de los componentes.

Los componentes no tienen sentido como objetos aislados, sino que adquieren valor cuando se conectan a un sistema al cual le brindan servicios especializados o cuando se unen a otros componentes para construir una aplicación completa.

El primer caso lo constituyen los sistemas abiertos, los cuales usan el recurso de los componentes para extender su funcionalidad; en el segundo caso, los sistemas basados en componentes están constituidos enteramente por componentes, por lo que el recurso de componentes es usado para construir toda la estructura del sistema.

Los sistemas abiertos tienen como característica el uso de interfaces estándar para invocar componentes que realicen determinadas funcionalidades requeridas. La idea que se tiene de los sistemas abiertos es que presentan facilidades de "plug-and-play" para diversos componentes que no necesariamente fueron pensados para trabajar juntos. Asimismo los sistemas abiertos prometen una rápida adaptación a los cambios tecnológicos cuando estos emergen, porque es posible acompañar la nueva tecnología, ya sea reemplazando componentes o por extensión del sistema [Oberndorf97] .

Los componentes pueden conectarse tanto para brindar servicios al sistema como para obtener servicios del mismo, dependiendo del diseño de las interfaces.

Los sistemas abiertos y los componentes son cosas bien diferenciadas, sin embargo adquieren verdadero significado cuando se complementan en una aplicación, es decir que un sistema abierto llega a su verdadera expresión cuando son conectados los componentes que completan su funcionalidad. Del mismo modo los componentes dejan de ser funcionalidades abstractas, sin resultados de valor, para convertirse en parte integrante de una aplicación, a la cual aportan sus servicios.

Para SEI – CMU los sistemas abiertos son *una colección de software, hardware y recursos humanos que interactúan de acuerdo a un diseño, para satisfacer requerimientos establecidos, con una especificación de interfaces de componentes:*

- *completamente definida,*
- *pública*
- *y mantenida de acuerdo a consenso de grupo*

y con implementaciones de componentes que conformen esas especificaciones. [Oberndorf97]

Un ejemplo claro de esta forma de uso de componentes se puede encontrar en la versión actual del sistema de información y gestión empresarial SAP R/3 (Alemania), para el cual se puede obtener una lista bastante importante de componentes (desarrollados por terceras partes), que extienden la funcionalidad del sistema en una gran cantidad de casos o aplicaciones particulares [SAP02].

Por su parte, un sistema basado en componentes es por naturaleza un sistema abierto, pero uno en el que toda la estructura interna del sistema ha sido construida en base a componentes, por lo que estos no solamente extienden la funcionalidad del sistema sino que definen su funcionamiento entero.

Un sistema basado en componentes, requiere para su concreción la adhesión a una arquitectura de software definida explícitamente, que da estructura a toda la aplicación. Los componentes pueden ser reemplazables o no, se pueden re-usar o no, dentro del mismo sistema o en otras aplicaciones, pero es impensable que se pueda construir un sistema no trivial, basándose exclusivamente en el agregado de los componentes que se encuentren, en una suerte de re-uso oportunista, o que la misma estructura del sistema se determine como consecuencia de la unión

de varios componentes. Para construir un sistema robusto que cumpla los requerimientos planteados de nivel empresarial, se debería emplear un enfoque "top-down", definiendo primeramente la arquitectura de ese sistema, como guía para la construcción y luego seleccionando los componentes disponibles adecuados o construyendo componentes específicos de acuerdo a las necesidades. [Bosch00].

"La arquitectura de un sistema basado en componentes describe su organización estática en componentes interconectados a través de interfaces y define a nivel significativo como interactúan estos componentes entre sí" [Griss01].

2.3.1. Ideas rectoras en el diseño de un sistema basado en componentes.

Hay una serie de ideas rectoras que se deberían observar a la hora de diseñar sistemas basados en componentes y en el diseño mismo de los sistemas abiertos. Estos ideas no son independientes las unas de los otras y muchas veces para llevar adelante alguna es necesario que se cumplan las demás. [Kang99] [Meyer00] [Netdecisions01].

- *Encapsulamiento.* Se refiere a que todos los componentes de un sistema deben interactuar con éste exclusivamente a través de su interfaz. No debe haber otra forma de utilizar el componente que no sea a través de la interfaz y el componente no debe comunicarse ni interactuar con el sistema si no es a través de la misma.

  Es decir que el sistema no debe ver más allá de la interfaz ni ocuparse de cómo es que trabaja el componente. Mientras sea posible, el tamaño del componente debe ser tal que su interfaz exprese funcionalidades simples y coherentes.

- *Aislamiento.* Esta idea refiere a que, en la medida de lo posible, diferentes componentes no deberían interactuar entre sí. Dicho de otra manera propone el bajo acoplamiento o el acoplamiento indirecto de los componentes, de modo que los cambios impacten en lugares precisos del sistema y que no afecten varios componentes interrelacionados.

  Como resultado de la minimización de interdependencias, los sistemas son en general más fáciles de analizar y comprender. En un mundo de cambios constantes, esto constituye una virtud muy deseable.

- *Abstracción.* La funcionalidad y la estructura deben estar expresadas de la manera más simple y general posible.

  En términos prácticos la abstracción significa por ejemplo presentar la funcionalidad de los componentes como interfaces más generales o mediar el intercambio entre componentes a través de un protocolo neutral, estándar.

  El uso de middleware y modelos de componentes estándar es una expresión palpable de esta idea.

- *Re-uso.* En la construcción del sistema se debe intentar utilizar componentes ya hechos cuando sea pertinente y todo componente de software que se desarrolle debe ser hecho pensando en su posible re-uso.

  Procurar el re-uso no significa generalizar los componentes para que abarquen múltiples funcionalidades, por el contrario, componentes con funcionalidades más acotadas y concretas, de alta cohesión, son más fáciles de comprender, captar su utilidad y por lo tanto

estimulan el re-uso, aún cuando esto implique, para cada componente, solucionar solo aspectos parciales del problema.

El aseguramiento de la calidad tiene mucho que ver con el re-uso, componentes que sean re-usables en una multiplicidad de sistemas, deben ser confiables y seguros y esto también apunta a componentes más pequeños, con funcionalidades concretas. Por otra parte, el adecuado uso de componentes probados de alta calidad, va a contribuir a la construcción de sistemas más sólidos y confiables que si estos componentes fueran hechos desde cero.

- *Agilidad.* Los cambios son inevitables, sea en el propósito con que fue hecho un sistema, el ambiente en que fue desplegado o los componentes que lo integran.
  Las reglas de negocio, estrategias, arquitectura, diseño, implementación, despliegue y configuración deben anticipar los cambios. Esta idea de agilidad tiene que ver con la identificación de los lugares dónde se pueden producir cambios más que con intentar adivinar cuales van a ser. Con estos lugares identificados, se puede diseñar el sistema de modo que cuando ocurra algún cambio se conozcan rápidamente las partes que hay que adaptar y el impacto de esto, logrando de ese modo mayor velocidad de respuesta.

- *Mantenimiento.* La ingeniería de los sistemas debe prever su mantenimiento. Este concepto está ligado a la anticipación de la falibilidad, ya sea por caída o por error, de cada parte de los elementos que intervienen en la construcción, uso y mantenimiento del sistema. Obviamente las demás ideas rectoras que se detallan tienen mucho que ver con el mantenimiento de los sistemas, pero además con la trazabilidad, visibilidad y control que hacen que los sistemas sean verificables y mantenibles.

  En ambientes distribuidos esta cuestión es crítica, incluso aunque se pueda asegurar la calidad de los componentes utilizados, el comportamiento de estos componentes va a ser muy dependiente de factores que en general son considerados como externos al sistema. Ya sean caídas de hardware, fallas de comunicación o de infraestructura, los sistemas no deberían ser considerados en forma aislada de su ambiente de despliegue y en ese sentido estos tipos de fallas deberían estar consideradas y previstas por los componentes.

- *Diseño por contrato.* Los componentes del software deben publicar su funcionalidad o el servicio que brindan a través de una interfaz bien definida. La definición de esa interfaz constituye un contrato entre cualquier cliente del servicio y el componente que lo brinda.

  Esta idea lleva a que ya desde el diseño mismo del sistema se explicite cada contrato, al definir lo que se espera que haga y las interfaces de cada componente. La funcionalidad del componente debería mantenerse estrictamente dentro de la funcionalidad expresada por esas interfaces, ni agregar ni quitar servicios.

  El diseño por contrato establece un vínculo claro entre la metodología de análisis y diseño y la construcción del sistema. Constituye además un aspecto clave para el desarrollo o adquisición de componentes en paralelo.

- *Funcionalidad expresada como servicios.* Funcionalidades significativas que contribuyan a un resultado global de la aplicación no deberían considerarse como parte de ese resultado, sino como servicios que son utilizados para lograrlo.

  Con esta visión se logra que funcionalidades importantes en el contexto de una aplicación puedan ser re-usadas, sin cambios en el contexto de otra. Esta idea brinda además un elemento estratégico para subdividir funcionalidades, reduciendo la complejidad de las mismas y promoviendo la idea de agilidad del sistema.

- *Expresión única y explícita.* La funcionalidad y los datos deberían ser creados una sola vez y luego usados por quien lo requiera. Por ejemplo, las reglas de negocio deberían ser establecidas con claridad en un solo lugar y compartidas por todas las piezas del software, en vez de aparecer en cada componente que las use. Deberían además ser expresadas explícitamente y no implícitamente por el modo en que opera el código.

  Esto no significa que estén escritas una sola vez; cuestiones de comunicación y escalabilidad pueden requerir múltiples copias de datos o funcionalidades, pero se debería asegurar el mecanismo que los mantenga consistentes en forma automática.

- *Mínima diversidad.* Normalmente hay más de una forma de realizar una tarea, pero razonablemente se debe procurar ejecutarla de la misma manera que otras tareas similares. En ese sentido es recomendable el uso de patrones y estándares, lo que promueve el re-uso de diseño y facilita la comprensión del diseño.

  Esto apunta a la convergencia hacia arquitecturas estándar y en la medida que las aplicaciones representen soluciones a problemas similares, apunta hacia las "líneas de productos" [Bosch00], es decir, familias de sistemas dentro de un dominio, que comparten una arquitectura y tienen en común un conjunto de componentes.

2.3.2. Las líneas de productos.

Las líneas de producto se definen como *familias de sistemas para un dominio, que comparten una arquitectura y un conjunto de componentes, pero que además presentan variantes significativas que los diferencian* [Griss01].

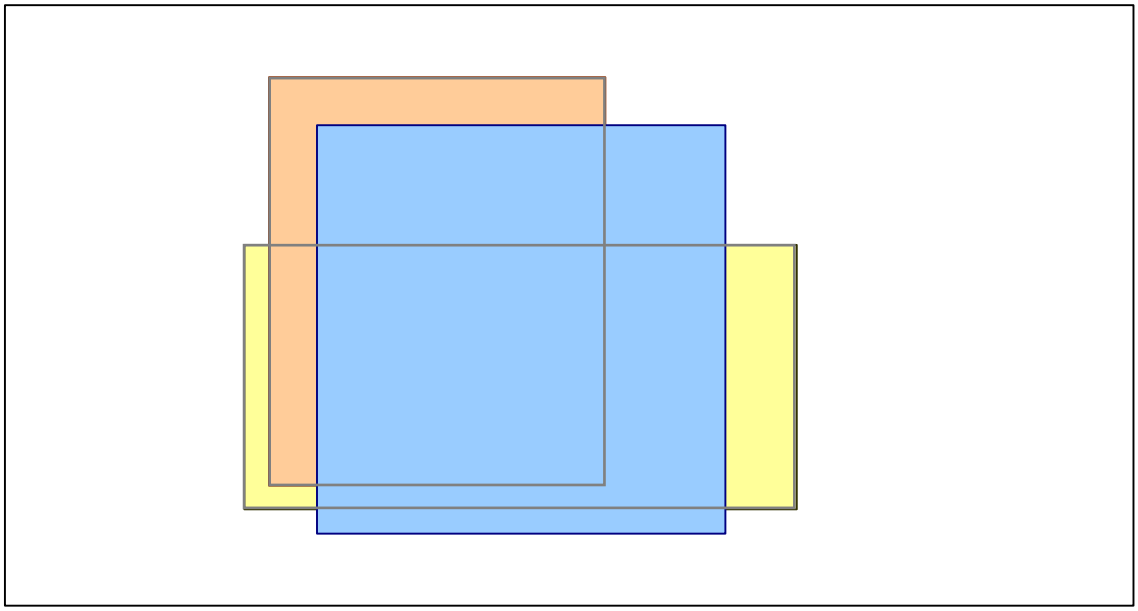

Figura 2. Distintos miembros de una línea de productos

En la Figura 2, se muestra en forma esquemática diferentes miembros de una línea de productos, representados por rectángulos, que comparten un conjunto de requerimientos y componentes (la zona superpuesta) y presentan diferencias, que en algunos casos representan funcionalidades exclusivas.

Los distintos miembros de una línea de productos tienen un conjunto de requerimientos en común. Los requerimientos comunes entre los miembros de las familias dan origen a un

conjunto de componentes compartidos. Cada miembro puede además orientarse hacia un sector particular del dominio (ya sea por la funcionalidad que implementa o por la forma en que lo hace), es decir que van a aparecer requerimientos diferenciados que a su vez definen los diferentes miembros de la familia. Las partes diferentes dan origen a distintas variantes de componentes, pero estas, por interactuar con la parte común, van a conformar las mismas interfaces

A diferencia de un conjunto de sistemas para un dominio, en el caso de las líneas de producto se ha hecho explícita la solución a los requerimientos comunes a través de un conjunto de componentes que son compartidos y de una arquitectura común a todos los sistemas.

"Una arquitectura de línea de productos es un diseño que identifica los componentes contenidos en los miembros de la familia, habilitando la síntesis de cualquier miembro en particular por composición de estos componentes" [Batory01].

La idea de las líneas de productos es la de ahorrar el costo de tener que materializar todas las variantes de los miembros de la familia, en cambio contar con la arquitectura y los diferentes componentes y construir el sistema adecuado, seleccionando los componentes que mejor se adapten a una aplicación dada, cuando esto sea requerido.

Cada miembro de la línea de productos instanciado de esa manera, es en sí un sistema basado en componentes. Lo que las líneas de producto aportan es la disponibilidad de los componentes para cada miembro que se desee implementar, en ese sentido se alinea con el enfoque que le da el SEI-CMU al tema [Oberndorf97].

Las líneas de producto proveen entonces el marco ideal para el re-uso de los componentes: estos ya fueron pensados para actuar en ese contexto, por lo que las necesidades de adaptación son bajas o inexistentes. Dado que las líneas de producto se definen en torno a una arquitectura específica, se logra además el re-uso de ésta.

El desarrollo de una línea de productos como estrategia para ofrecer soluciones generales y adaptables a casos particulares dentro de un dominio exige un enfoque top-down. La definición de la arquitectura de la línea de productos debe ser previa al desarrollo de los componentes específicos que implementan las diferentes opciones o funcionalidades.

Para que esto sea posible deben existir razones de negocio que justifiquen una inversión mayor: debido a su generalidad, el desarrollo de líneas de producto es más costoso que el desarrollo de un producto en particular y tiene que estar claramente justificado por la diversidad de productos finales y la parte común de los mismos [Bosch00]. Por ejemplo una empresa que desarrolla sistemas para el mercado va a obtener mejores resultados con la estrategia de líneas de producto que otra que desarrolla un sistema específico para su propio uso.

En el desarrollo de líneas de producto se recurre a la ingeniería y al análisis de dominio, para identificar requerimientos y elaborar arquitecturas y componentes re-usables que son un activo de la línea de productos. Luego a través del análisis de la aplicación particular, se puede materializar el producto adecuado para cada caso. Esto impacta en la estructura organizacional de la empresa de software, ya que deberán existir sectores especialistas en el dominio, a cuyo cargo estará el desarrollo y evolución de la arquitectura y los componentes re-usables y otros sectores especializados en el análisis y desarrollo de las aplicaciones particulares.

Cuando se trabaja en torno de una línea de producto, el proceso de desarrollo de software también se ve impactado. El solo hecho de que ya haya disponibles una arquitectura y un conjunto de componentes a la hora de iniciar un nuevo sistema, produce un cambio en el proceso del software. Por otra parte se deben contemplar procesos de software específicos para

el desarrollo y mantenimiento de los componentes reusables, como piezas independientes. En la sección 3 se estudia el impacto del CBD en el proceso de software y la incidencia de las líneas de productos y del mercado de componentes en el mismo.

2.4. La evolución de los sistemas ERP.

Un ejemplo interesante de la aplicación del concepto de componentes y sistemas abiertos lo da la evolución que han tenido los sistemas ERP (Enterprise Resource Planning).

Los sistemas ERP son aplicaciones configurables que brindan soluciones integradas a la mayoría de los aspectos de gestión de procesos y sistema de información para las empresas. Es reconocido el bagaje de conocimientos y reglas de negocio que acumulan y el conjunto de buenas prácticas de negocio que se obtienen implícitamente al adquirir e instalar el software. Tradicionalmente estos sistemas han constituido paquetes de tipo monolítico y propietario, aunque generalmente están preparados para correr en más de una plataforma.

La expectativa de los usuarios de trabajar con sistemas integrados que involucren las distintas áreas de la gestión de las empresas (finanzas, comercialización, suministros, producción, personal, etc.) ha llevado a que los distintos fabricantes de ERP extendieran estos sistemas a sectores que no eran su especialidad, para cubrir un espectro cada vez mayor. Por ejemplo, es reconocida la especialización de PeopleSoft en el área de administración de personal o la de SAP en el área de producción, sin embargo ambos fabricantes extendieron sus productos a todas las demás áreas de la empresa. Un enfoque de sistemas abiertos hubiera permitido unir soluciones de uno y de otro en las diferentes áreas aprovechando lo mejor de cada uno, sin embargo las estructuras propietarias y monolíticas (y las políticas de los fabricantes de ERP) lo hicieron imposible [Kim99].

En los últimos cuatro años, los grandes fabricantes de sistemas ERP (SAP, PeopleSoft, J.D. Edwards, BAAN, etc.) han reestructurado estas aplicaciones, volcándose hacia sistemas abiertos y en algunos casos estos han sido enteramente reprogramados en base a componentes [Sprott00].

Estos cambios se debieron a diversas razones [Sprott00], [Soh00], [Kumar00], entre las que se cuentan:

- Necesidad de integración de sistemas corporativos a lo ancho del mundo
- Necesidad de colaboración entre organizaciones que utilizan sistemas diferentes.
- Diferencias culturales significativas, especialmente cuando se intentó introducir estos sistemas en los mercados del lejano oriente (Japón, Malasia, etc.).
- Aparición y fortalecimiento de los nuevos paradigmas de negocio tales como e-business, e-commerce y b2b (business-to-business).

Algunos fabricantes como SAP, promocionan el uso de componentes propios y de terceros como forma de fortalecer las aplicaciones [Sprott00]. Por ejemplo, SAP cuenta con clasificaciones de tipos de software que pueden interactuar con sus sistemas a modo de componentes y la lista de fabricantes (certificados y no certificados) que los proveen. En general se cuenta con una descripción general del objetivo del componente y del tipo de interfaz, aunque no hay mucha información (al menos de acceso público) acerca de los modelos de componentes admitidos, en algunos casos aparecen referencias a COM/DCOM y en otros a EJB.

La Figura 3 muestra un dibujo que describe un componente para acceder a información de crédito de personas y empresas, en este caso particular manejada por una empresa de tipo clearing de informes, que es además quien fabrica y suministra el componente. Otras

organizaciones similares podrían fabricar componentes equivalentes para acceso a sus propias bases, usando la misma interfaz de SAP.

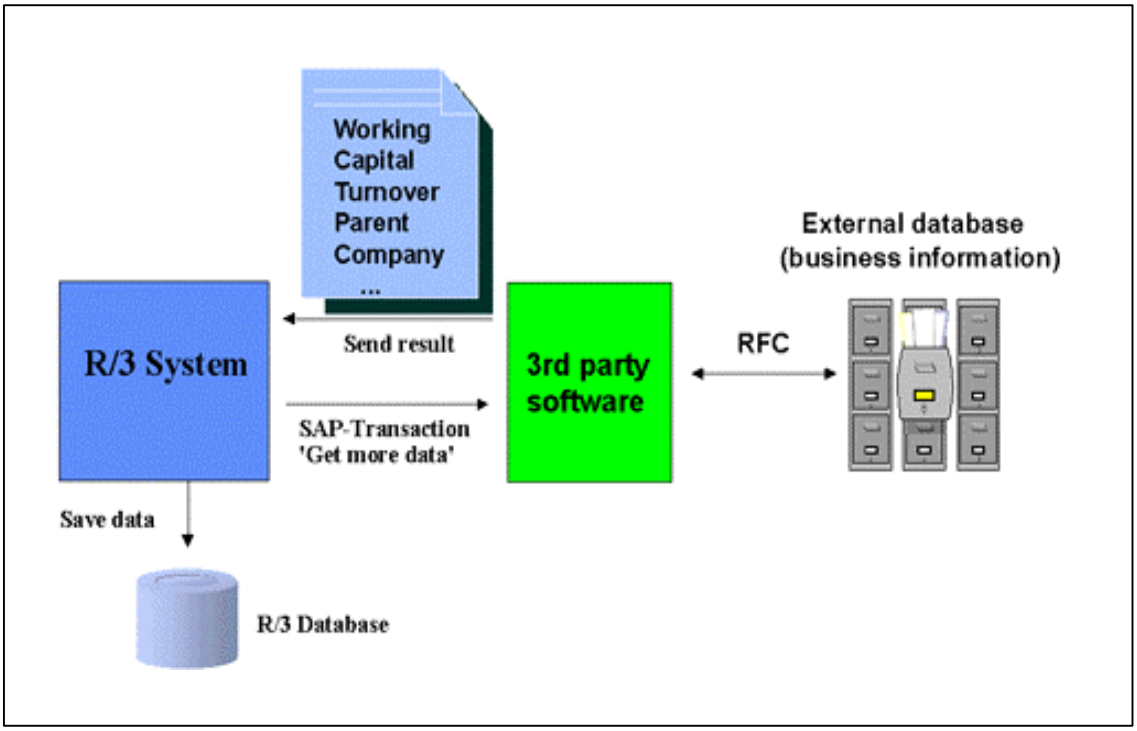

Figura 3. Descripción de componente Credit Management para SAP R/3

El componente de tercero se conecta a una interfaz descrita como FI-AR (Financial - Accounting Receivable) de SAP a través de una operación “Get more data”, realiza una conexión a la base de datos remota, dónde consulta la información de crédito y devuelve la información del cliente, la que es almacenada en las bases de SAP R/3 para su uso [SAP02].

En la medida que los ERP continúen y estimulen la estrategia de promover el uso de componentes (propios o de terceros) para la provisión de soluciones específicas para aspectos especiales de la aplicación, se creará en torno de ellos un conjunto importante de componentes y un buen número de organizaciones de software que los producen, en un fenómeno similar al que ocurrió en la industria automotriz, con GM (General Motors) especificando partes para que sean producidas por terceros [Kim99], pero manteniendo el liderazgo en base justamente a la especificación de las partes.

Los ERP mismos estarán en la categoría de sistemas abiertos o de sistemas basados en componentes, brindando una solución general integral para la gestión básica de las empresas y los usuarios (o sus departamentos de tecnología informática) podrán cumplir requerimientos o procesos específicos mediante la selección e incorporación de componentes adecuados. Sin embargo cada sistema ERP responde a diseños y arquitecturas específicas y propietarias, por lo que de alguna manera seguirán presentándose como soluciones monolíticos. No se espera, por ejemplo, que algún componente interno de BAAN pueda intercambiarse con componentes de SAP, por lo menos en corto plazo.

Los nuevos diseños basados en componentes apuntan más que nada a características de distribución o despliegue del software, pero aspectos como composición dinámica o reemplazo de componentes no son manejados.

No debe perderse de vista que los sistemas ERP (sobre todo los grandes ERP, como SAP, Baan, PeopleSoft, J.D. Edwards, etc.) encierran en sus reglas las mejores prácticas de negocio, avaladas por una cantidad muy grande de usuarios. Tampoco deben perderse de vista las características de alta configuración de estos sistemas, a través de las cuales se obtiene la flexibilidad y adaptabilidad. Estos sistemas tienden más bien a aumentar su funcionalidad que a permitir que nuevas funciones sean aportadas por componentes intercambiables.

Han habido también algunos traspiés para enfrentar situaciones nuevas, como por ejemplo el tener que manejar más de una moneda con tipos de cambio fluctuantes, cosa que ocurrió en oportunidad de la unificación monetaria en Europa y que ha ocurrido con frecuencia en implementaciones de países de Sudamérica.

3. El proceso de sistemas basados en componentes.

El desarrollo de sistemas basados en componentes presenta una manera diferente de construir sistemas, en oposición al proceso tradicional de desarrollo, en el cual todos los aspectos de la aplicación deben ser producidos "en casa" y que suele desembocar en un software monolítico. En esta nueva forma de construir los sistemas, la opción entre desarrollar y comprar adquiere un rol de primera importancia.

En esta sección se desarrollan las ideas fundamentales que sustentan el paradigma de desarrollo de sistemas basado en componentes (paradigma CBD). El cambio de mentalidad que implica, las nuevas actividades que aparecen en torno al desarrollo de sistemas, el impacto que esto produce en el ciclo de vida del software y las ventajas y desventajas aparejadas en su uso.

Vinculado al paradigma CBD y como condición fundamental para que este pueda ser efectivamente llevado adelante, surge el problema de la disponibilidad de componentes prehechos, resuelta a través de la eventual formación de mercados de componentes de negocio o la estrategia de elaboración de líneas de producto que proveen una arquitectura y un conjunto de componentes re-usables. Estas alternativas son estudiadas en las sub-secciones finales de esta sección.

3.1. Ideas que dan sustento al CBD.

El desarrollo de sistemas basados en componentes apunta a que el desarrollo de aplicaciones de gran escala se realice integrando componentes de software ya existentes. En la base de esta concepción está la idea de que algunas partes de los sistemas reaparecen con cierta regularidad y que esas partes debieran ser escritas una sola vez y debieran estar prehechas al momento de desarrollar una aplicación particular[Haines97].

La frase de Fred Brooks [7] "*Comprar versus construir. La solución más radical al problema de la construcción de software es no construirlo en absoluto.*" [Brooks87], establece la filosofía que soporta el desarrollo de sistemas basados en componentes y un cambio de mentalidad radical de los desarrolladores, en oposición al prejuicio tan habitual en la profesión informática: *"no hecho aquí"*.

En el enfoque basado en componentes para el desarrollo de sistemas, la noción de construir un sistema escribiendo software es reemplazada por la de construir un sistema mediante ensamblado e integración de componentes. En contraste con el desarrollo tradicional, en el cual la integración era una etapa tardía del proceso, en el enfoque CBD la integración constituye uno de los elementos centrales; la integración desplaza a la implementación como actividad protagónica.

Según Paul Allen hay tres grandes líneas de enfoque para esta forma de construir sistemas [Allen00]:

- *Components in advance*, es un enfoque top-down, centrado fuertemente en la arquitectura del sistema, que apunta a un sustancial mejoramiento de la adaptabilidad del software para acompasar los cambios en los procesos de negocios.
- *Components as you go*, se basa en la liberación incremental y en paralelo de soluciones de negocio y componentes. Cada incremento debe liberar mejoras de procesamiento para el negocio a la vez que contribuye en la creación de una arquitectura de componentes.

[7] Fred Brooks es el autor del libro "The Mytical Man-Month. Essays on Software Engineering" y profesor emérito en la Universidad de North Carolina

- *Components by opportunity*, se basa en la búsqueda de componentes prehechos que permitan ir ensamblando el sistema. La arquitectura del sistema será un resultado de los componentes seleccionados.

El primero de estos enfoques hace especial hincapié en el diseño estructural del sistema y la identificación (y especificación) de los componentes necesarios, generando un marco para el desarrollo de soluciones específicas o familias de soluciones (líneas de productos).

El segundo enfoque promueve la adaptación a nuevas necesidades en forma incremental, por vía de la extensión del sistema, donde la arquitectura es un resultado de los sucesivos incrementos y en todo caso específica para una problemática particular; es un enfoque del tipo "*bottom-up*".

El tercer enfoque se basa especialmente en el re-uso de componentes ya existentes, adecuando la arquitectura del sistema a estos y no tanto a las necesidades de la aplicación, en este caso no hay una arquitectura predefinida para el sistema, cuya concreción va a depender del éxito que se tenga en encontrar los componentes útiles.

Jan Bosch[8] dice, al respecto de estos enfoques, que la comunidad de desarrolladores ha aprendido dos lecciones, la primera es que el enfoque oportunista en la práctica no funciona y la segunda que el re-uso planificado de los componentes debe realizarse a través del enfoque "top-down" ya que el enfoque "bottom-up" tampoco funciona [Bosch00], aunque es bueno tener en cuenta que muchas veces las características de componentes prehechos que se quiera utilizar pueden tener incidencia en la arquitectura del sistema.

De todas formas, en cualquiera de los enfoques, cuando se ha identificado un componente, el camino para obtenerlo sería [Wilkes02]:

1º *re-usar,* primero re-usar lo que ya se tiene,
2º *comprar,* si no se tiene, entonces comprar uno ya hecho y
3º *construir,* como última instancia, si no está disponible o no existe entonces construirlo.

Se persigue un criterio de minimización de costos en la búsqueda de componentes.

3.2. Actividades específicas en el CBD.

De acuerdo a lo expresado por Haines en una ficha técnica del SEI [Haines97] y como esquematiza la Figura 4, el desarrollo basado en componentes (CBD) se caracteriza por cuatro actividades principales que lo distinguen y que se integran al ciclo de desarrollo de software. Estas son: (a) calificación de componentes, (b) adaptación de componentes, (c) ensamblado del sistema y (d) evolución del sistema y los componentes, a través de las cuales se logra la incorporación de componentes adecuados en un sistema y su posterior mantenimiento.

Específicamente estas actividades consisten en:

a) Calificación de componentes. Esta actividad implica el proceso de determinación de cuales componentes preexistentes se ajustan a las necesidades de un nuevo proyecto y a la selección de componentes útiles dentro de un mercado de componentes. Más allá de verificar la conformidad con los requerimientos para el componente, la calificación de

[8] Profesor de Ingeniería de Software en la Universidad de Groningen, Holanda, con muchos trabajos de investigación en el área de Arquitectura de Sistemas.

componentes puede incluso referir a las normas de calidad y confiabilidad aplicadas en el desarrollo del mismo.

La calificación de componentes consta de dos fases: *descubrimiento* y *evaluación.*

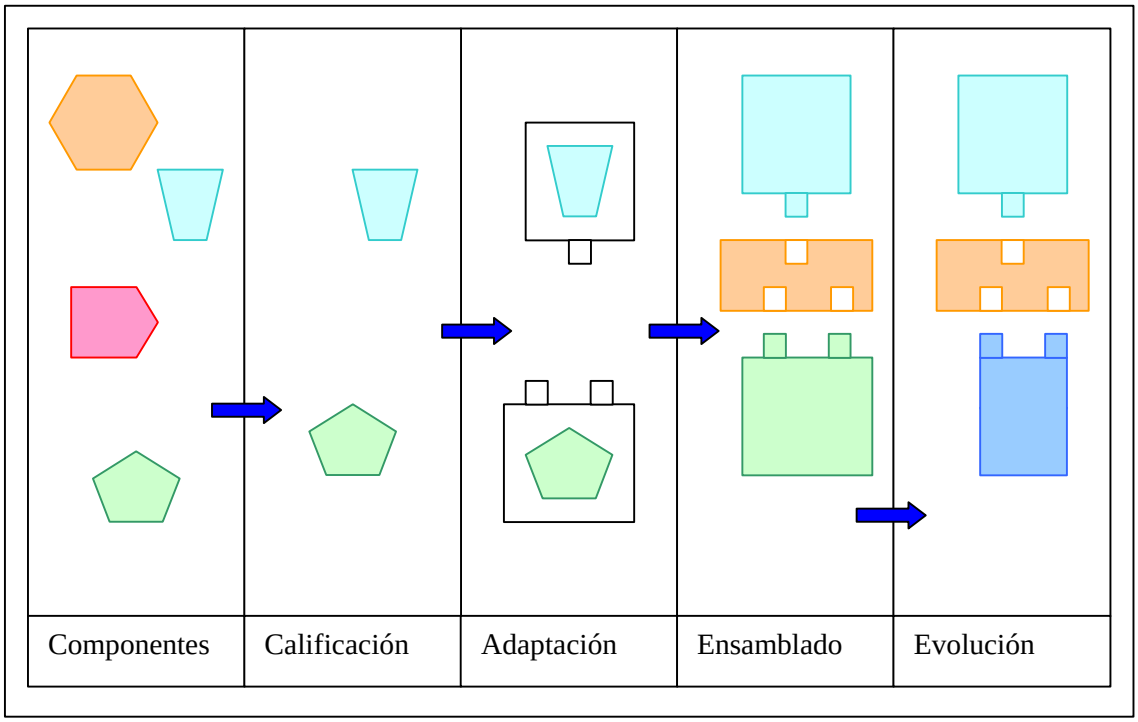

Figura 4. Actividades principales de CBD

En la fase “descubrimiento” se identifican las propiedades del componente, estas propiedades incluyen su funcionalidad y otros aspectos de su interfaz, como el modelo de componente estándar usado. También se pueden incluir, aunque más difíciles de evaluar, las características de calidad, confiabilidad, performance, etc. y características propias del fabricante como idoneidad, solvencia y estabilidad de la empresa, que darán cierta seguridad en cuanto a la evolución futura del componente.

En la fase “evaluación” se realizan las pruebas que permitan verificar si el componente cumple con su especificación y con los requerimientos planteados. En esta fase se incluyen típicamente pruebas “de escritorio”, consulta a otros usuarios del componente y eventualmente “benchmarks” y pruebas mediante prototipos.

b) *Adaptación de componentes.* Como los componentes son escritos para cumplir diferentes requerimientos y tomando ciertas presunciones en cuanto al contexto, frecuentemente deben ser adaptados para integrarlos a nuevos sistemas, diferentes de aquel para el que fueron hechos originalmente.

Dependiendo del tipo de adaptabilidad del componente (white box, grey box, black box) las técnicas de adaptación pueden variar desde la modificación del código original, la configuración de parámetros o el desarrollo de “wrappers” “bridges” o “mediadores” (cuyos mecanismos fueron descritos en la sub-sección 2.2.1), que permitan la integración.

c) *Ensamblado de componentes.* Esta actividad refiere a la forma en que van a interactuar los componentes con el sistema y entre sí. Los componentes deben ser ensamblados en

una infraestructura bien definida; desde el punto de vista estructural: que permita la interconexión de los componentes; desde el punto de vista conceptual: que construya un sistema de aplicación coherente al reunir componentes diversos.

Históricamente se han usado varias arquitecturas de ensamblado:

- *Database*, en la que la información que comparten los componentes se da a través de los mecanismos centralizados de almacenamiento.
- *Message bus*, en la que los componentes manejan diferentes bases de datos propias, coordinadas por un mecanismo de mensajería.
- *Object request broker*, en la que la tecnología de ORB provee los mecanismos para encontrar y activar los componentes.

d) Evolución del sistema. A primera vista parece que la evolución será fácil de sobrellevar ya que los componentes forman la unidad de cambio. Para reparar un error un componente es reemplazado por una corrección del mismo o para incluir una nueva funcionalidad requerida se puede hacer evolucionar el componente o reemplazarlo por otro.

Sin embargo esto puede ser una tarea muy ardua, dado que un componente nuevo difícilmente se va a comportar exactamente igual que el reemplazado y nuevas pruebas deben ser implementadas antes de su adopción. Pueden surgir incompatibilidades nuevas con los demás componentes y en general los adaptadores (wrappers, bridges, mediators, etc.) van a tener que ser escritos nuevamente.

En resumen, siguiendo estas actividades, los componentes son obtenidos y calificados a partir de un mercado (externo o interno), luego son adaptados mediante diversas técnicas a los requerimientos y contexto específicos del sistema y finalmente son ensamblados formando el sistema abierto que provee la solución. Se deben implementar mecanismos administrativos que permitan controlar el uso y versiones de componentes para facilitar la evolución del sistema.

3.3. Impacto en el ciclo de vida del software.

Las organizaciones, o sectores de las mismas, que desarrollan sistemas de aplicación también verifican cambios estratégicos con la adopción del paradigma CBD: de productores de software pasan a ser consumidores de software (o por lo menos en lo que respecta a los componentes). Esto implica el desarrollo de políticas y criterios administrativos y de control que permitan gerenciar la adquisición y el uso de los componentes.

El ciclo de vida del software se ve impactado por estas nuevas estrategias. Así como la adquisición de sistemas estándar completos impacta el ciclo de implementación de una aplicación [Sawyer01], del mismo modo el uso de componentes prehechos (propios o comprados) impacta en el ciclo de vida del software.

Dejando de lado la discusión acerca de la validez del modelo de ciclo de vida “en cascada”, dado que en él se establecen todas las actividades propias del desarrollo de software, es útil tomarlo como referencia a fin de mostrar los cambios que se producen en las mismas cuando se incorpora el paradigma CBD. La comparación de actividades del ciclo de vida tradicional y el correspondiente al CBD se esquematizan en la Tabla 1.

En algunas etapas las actividades están bien diferenciadas entre el desarrollo tradicional y el CBD y en otras no tanto, sin embargo en todas hay un cambio conceptual impuesto por el paradigma: es un proceso que está más orientado a adquirir y utilizar componentes preexistentes

que a programar todas las funcionalidades del sistema. En cierta medida se puede decir que es un proceso más alineado con las prácticas de ingeniería, dejando de lado algunos aspectos artesanales del desarrollo tradicional.

Ciclo del desarrollo tradicional	Ciclo del desarrollo con enfoque CBD
Planificación del Proyecto	Planificación del Sistema
Iniciación del Proyecto	Iniciación del Sistema
Análisis de Requerimientos • Detalle de requerimientos • Vincular requerimientos a especificaciones • Documento de especificaciones	Análisis de necesidades del Sistema • Necesidades de alto nivel • Características funcionales críticas • Identificación de arquitectura y componentes funcionales Análisis funcional de componentes • Identificación de mercado o línea de productos • Especificación de funcionalidad de los componentes • Comparación y selección de productos Determinación de desvíos • Entre componentes y organización • Entre componentes y sistema • Identificación de adaptadores
Diseño del sistema • Resolución algorítmica • Modularización	Diseño del sistema • Arquitectura del sistema • Modelo de componentes
Programación • Diseño de programas • Codificación	Programación en paralelo • Adaptadores de componentes re-usados • Desarrollo de prototipos • Programación de componentes nuevos • Prueba de componentes aislados
Integración y pruebas • Prueba de unidades • Prueba de integración • Prueba del sistema	Ensamblado y prueba del sistema (prototipos) Prueba funcional del sistema (componentes finales)
Instalación del sistema • Instalación del software • Capacitación a los usuarios	Despliegue del sistema • Distribución, despliegue del software • Capacitación a los usuarios: el nuevo sistema y los nuevos procesos
Mantenimiento del sistema	Mantenimiento • Del sistema • Administración de componentes

Tabla 1. Comparación de actividades de los ciclos de vida de desarrollo

No se hace hincapié en el concepto de proyecto (un proceso largo, con gran cantidad de horas hombre de desarrollo, coordinación de equipos de trabajo, etc.), sino en el de la arquitectura del sistema, la visión de la misma como integradora de toda la aplicación y la identificación y especificación de los componentes necesarios para cumplir los requerimientos.

El documento de especificaciones, base para el diseño y desarrollo de los programas en el proceso tradicional, es la fuente para la búsqueda y evaluación de componentes y a la identificación y mitigación de desvíos con respecto a las necesidades de la organización. En este sentido debe tenerse en cuenta que los componentes preexistentes fueron desarrollados no para

cumplir los requerimientos de un sistema en particular sino para los requerimientos de un mercado o para los que fueron identificados en el marco de una línea de productos del dominio.

El resultado del análisis funcional del sistema, define una arquitectura para el mismo, las interfaces que vinculan los componentes y los contratos entre estos (especificaciones semánticas de la funcionalidad de los componentes). El contraste entre requerimientos o casos de uso y la funcionalidad del sistema se realiza justamente utilizando esa arquitectura [Bosch00].

Uno de los aspectos de mayor impacto es que el diseño del sistema no se ve tan influido por la resolución de problemas algorítmicos, como ocurre en el desarrollo tradicional. El desarrollo basado en componentes permite concentrarse en la arquitectura del sistema durante el diseño, pudiéndose diferir la resolución de algoritmos para etapas posteriores o directamente incorporando las mismas embebidas en los componentes re-usables disponiobles.

En los casos en que se pueda contar con una línea de productos o exista un mercado de componentes del dominio, la programación en general va a implicar volúmenes de trabajo mucho menores que en el ciclo tradicional y básicamente estará orientada al desarrollo de adaptadores o funciones especiales. Aún en caso de ser necesaria la programación de componentes especiales, esta puede ser realizada en paralelo, basándose en la arquitectura del sistema, la especificación de interfaces y los contratos definidos para cada componente [Sparling00a], pero con independencia de la aplicación en particular.

Para el caso en que se demore la adquisición o desarrollo de determinados componentes, es posible el desarrollo de prototipos que emulen el comportamiento de los componentes requeridos, lo que permite acelerar las etapas de integración y pruebas del sistema. En este sentido los ciclos de vida del software de tipo iterativos e incrementales son especialmente adecuados para el desarrollo CBD [Pfleeger01].

Se debe tener especial cuidado ya que, a diferencia del proceso tradicional, en el cual se pude realizar el diseño y luego programar ajustándose estrictamente a esa definición, cuando se usan componentes prehechos, es probable que su incorporación impacte en la arquitectura del sistema, debiéndose realizar transformaciones en la misma, por lo que se deben prever los mecanismos de comunicación entre todos los equipos que trabajan en paralelo [Sparling00a], [Bredemeyer02], [Malan02a], [Malan02b]. El impacto puede incluir aspectos de manejo de las reglas de negocio o variaciones a los requerimientos del sistema por lo que se deberá renegociar con las personas afectadas por la aplicación o que tengan incidencia en la determinación de objetivos de la misma [Albert02].

Siguiendo el paradigma CBD, la prueba de unidades se realiza en oportunidad de la selección de los componentes prehechos, por lo tanto en la etapa que corresponde a pruebas, lo principal va a ser el ensamblado del sistema y la prueba funcional. El problema del “testing" de componentes y de sistemas basados en componentes fue un tema recurrente del 3er. y 4to. “Workshop on Component-Based Software Engineering” de ICSE 2000[9] e ICSE2001[10] respectivamente, y se puede encontrar profusa documentación en [ICSE00] e [ICSE01].

Como se dijo anteriormente, el desarrollo basado en componentes facilita el uso de prototipos que emulen ciertas funcionalidades todavía no implementadas y de ese modo acelerar las etapas de pruebas de performance y rendimiento del sistema.

[9] International Conference on Software Engineering 2000, Limerick, Irlanda, Junio de 2000
[10] International Conference on Software Engineering 2001, Toronto, Canadá, Mayo de 2001

En la etapa de instalación, sobre todo si se trata de un sistema de componentes distribuidos, se debe hacer especial énfasis en el despliegue del sistema, la ubicación de los componentes en los diferentes equipos que integran la red.

Asimismo, si se incorporaron componentes que incluyen sus propias prácticas de negocio, es posible que haya que instruir a los usuarios finales en el uso y aprovechamiento de las mismas.

Finalmente, el producto de un desarrollo basado en componentes no es exclusivamente el sistema desarrollado, sino además un conjunto de componentes re-usables, que constituyen un activo de la organización y como tal deberán preverse mecanismos para su administración y mejoramiento, lo que en definitiva constituye otros ciclos de desarrollo en paralelo.

3.4. Ventajas y desventajas del CBD.

Tradicionalmente se han señalado una serie de ventajas en la adopción del paradigma CBD, algunas de las cuales son resultado directo de los principios de calidad aplicados en el desarrollo.

- *Menor tiempo de producción*. Como resultado directo del re-uso es de esperar menores tiempos para el desarrollo de sistemas y por lo tanto implementaciones menos costosas y ajustadas a cronograma y presupuesto. Aún en casos en que los componentes deban ser desarrollados desde cero, la posibilidad de desarrollo en paralelo, basada en el principio de diseño por contrato (como se detalla en 2.3.1), ciertamente acelera los plazos de desarrollo (el "*time to market").*

- *Menor complejidad*. Dado que el uso de componentes preexistentes implica acceder a soluciones ya prontas para los problemas típicos de la aplicación, se requiere menor capacitación en detalles específicos del dominio, o en resolución de particularidades tecnológicas (GUI, conexión a bases de datos, etc.). Con el paradigma CBD se puede mantener una óptica más general y abstracta de la implementación, al no ser necesario manejar los diferentes temas a nivel de detalle de implementación. Siguiendo las ideas de David Parnas [Parnas72] las decisiones de diseño quedan "escondidas" en cada componente.

- *Mantenimiento*. Los sistemas elaborados en base a componentes cuentan con mayores facilidades de mantenimiento ya que los cambios efectuados en un componente solamente deberían afectar a ese componente, en la medida que las interfaces no se alteren (lo que de acuerdo a la definición de componentes debe cumplirse para conservar el componente).

- *Facilidad de integración*. Ya que los sistemas basados en componentes cuentan con interfaces estándar a través de las cuales se puede integrar partes nuevas o eventualmente comunicarse con otros sistemas. En última instancia la integración es uno de los propósitos de los sistemas basados en componentes.

- *Los sistemas CBD no son monolíticos*. Se pueden desplegar sistemas con distribución razonablemente libre de sus componentes, lo que facilita las arquitecturas modulares. Por otra parte los sistemas construidos en estas condiciones son fácilmente escalables, pudiéndose replicar los componentes críticos quedando a cargo del software de base o del middleware el balanceo de los servicios.

- *Los sistemas CBD no imponen límites artificiales.* Los sistemas desarrollados en base a componentes tienen gran facilidad de extensión a través de la sustitución o agregado de componentes. Un componente para control de líneas de crédito por ejemplo, puede al

principio acceder a una base de datos propia para obtener información, pero puede ser extendido por otro componente que además acceda a bases de datos remotas, quizás hasta manejadas por sistemas de otras compañías, como se muestra en la Figura 3, relativa a los componentes de SAP R/3.

- *Facilidad de evolución.* A través de la actualización de los componentes que integran la aplicación es más fácil mantenerse al día con los avances tecnológicos y también incorporar nuevas prácticas de negocio o modificar las ya existentes de acuerdo a los cambios en políticas o estrategias empresariales.

En resumen las principales ventajas del CBD apuntan a la rapidez de desarrollo, la menor complejidad y a la facilidad de mantenimiento y evolución.

Sin embargo, la adopción de este paradigma presenta algunos aspectos que pueden significar dificultades inesperadas:

- *Pérdida de autonomía.* El uso de componentes de terceros o de componentes propios que son re-usados en otras aplicaciones hace que decisiones acerca de los cambios que se puedan realizar en estos no dependan de quienes mantienen un sistema particular. En el caso de componentes de terceros, las iniciativas, motivaciones y orientaciones de esos cambios serán exclusiva potestad de los fabricantes y podrán tener rumbos desconocidos de antemano o divergentes de las políticas de los usuarios, incluso cambios propuestos por el consumidor de los componentes serán tenidos en cuenta o no, dependiendo de decisiones del fabricante. En el caso de componentes propios re-usados por otras aplicaciones o en el marco de una línea de productos, se deberá llegar a un acuerdo entre los diferentes responsables del mantenimiento antes de poder implementar ningún cambio.

- *Readaptación.* Ante cambios en componentes es muy probable que los adaptadores o wrappers deban ser reescritos. Se debe tener en cuenta que los cambios en los componentes pueden haber sido efectuados por iniciativas externas a las de los responsables de un sistema y aún así haya que incorporar esas nuevas versiones para no perder actualización.

- *Versiones de componentes.* Más allá de los problemas mencionados en los cambios en un componente, es necesario implementar algún sistema de administración de versiones de componentes usados. Puede ocurrir además que una versión más nueva de un componente no sea compatible con versiones anteriores de otros componentes y por tanto al hacer un cambio se produzca un efecto de bola de nieve que termine haciendo que cambien varios componentes más.

- *Componentes integrados.* Es habitual que los componentes evolucionen agregando nuevas funcionalidades a través de nuevas interfaces. En esta evolución es frecuente que un componente incluya funcionalidades que antes eran provistas por otros componentes: se deberá tomar la decisión de mantener los componentes anteriores y perder actualización o incorporar el componente nuevo y cambiar el flujo del sistema.

En general los inconvenientes expuestos son manejables a través de prácticas adecuadas de ingeniería de software, fundamentalmente las que tienen que ver con la especificación formal de los componentes y con configuración de versiones (Software Configuration Mangement, SCM) y administración de requerimientos y cambios (Requirements Management, RM).

Otra dificultad a considerar es la disponibilidad de componentes prehechos. Una de las premisas del paradigma de construcción de sistemas basados en componentes es que estos representan soluciones generales a diversos aspectos del dominio de aplicación y pueden ser independientes de una aplicación en particular. En estas condiciones, lo deseable a la hora de iniciar la

construcción de una aplicación particular es que, al menos, se encuentren disponibles los componentes que implementen los aspectos comunes del dominio. Una de las alternativas para esto es la formación de mercados de componentes, disponibles para toda la industria, la otra, menos ambiciosa es la estrategia de líneas de producto que proveen a la organización que las construye de una estructura general de la aplicación y el conjunto de componentes comunes para ese diseño. En las sub-secciones 3.5 y 3.6 se estudian cada una de esas alternativas.

3.5. El mercado de componentes.

La posibilidad de desarrollo de componentes en forma independiente de la construcción de aplicaciones concretas, abre las puertas a una nueva forma de negocio para organizaciones de software, basada en la especialización y desarrollo de determinadas funcionalidades específicas para un dominio y la oferta comercial de las mismas. Esto genera un mercado de componentes de software, cuyos principales clientes serán no ya los usuarios finales, sino otras organizaciones (o partes de estas) dedicadas al desarrollo de soluciones informáticas.

A continuación se estudian las características propias de un mercado de componentes de software, que ventajas trae aparejadas la formación de un mercado de esas características y cómo beneficia a los diferentes actores del mismo. Se estudia además la realidad de ese mercado en lo que tiene que ver específicamente con componentes que implementen la lógica de las aplicaciones, con la mira puesta en particular en el dominio de los sistemas de información y gestión empresarial. Este estudio está hecho en base a una pesquisa realizada en diversos sitios de internet, presentados como mercados de componentes y a opiniones recogidas de diversas fuentes que se detallan oportunamente.

3.5.1. Características, estructura y roles de un mercado de componentes de software.

Un mercado se puede definir como *"un grupo de compradores y vendedores que están en un contacto lo suficientemente próximo para que las transacciones entre cualquier par de ellos afecten las condiciones de compra o venta de los demás"* [Seldon67].

Los mercados son elementos básicos de la economía y de las relaciones entre empresas y entre personas. Aparecen cuando determinados sectores tienen necesidad de proveerse de algún bien o servicio y por diferentes motivos no pueden autoabastecerse. La necesidad de esos sectores es el incentivo que hace que otros sectores (los productores o proveedores) corran los riesgos de elaboración del bien o servicio requerido en procura de la rentabilidad de su comercialización [Sawyer01].

Hay tres elementos entonces sin los cuales no puede existir el mercado: el producto (bien o servicio), los consumidores y los productores. Los diferentes sistemas económicos pueden dar diferentes marcos a este concepto y de ahí pueden derivar muchas definiciones diferentes de lo que es un mercado, pero los tres elementos señalados aparecen invariablemente en todos los sistemas, ya que son la razón de su existencia. El mercado es un medio, no un fin en sí mismo.

Independientemente de las definiciones de mercado, hay algunas características que se verifican siempre:

- *Visibilidad.* Productores y consumidores se pueden encontrar entre sí con facilidad, los productos y quienes los fabrican o proveen pueden ser encontrados por quienes los necesitan, los proveedores pueden identificar los posibles consumidores y disponen de medios donde ofrecer sus productos.

- *Competencia.* Los mercados generan competencia, es decir que normalmente va a aparecer más de una oferta para satisfacer las necesidades de un cierto grupo de consumidores. Las diferencias entre una propuesta y otra pueden radicar en condiciones económicas, calidad de los productos o servicios ofrecidos u otros elementos que permitan distinguir y comparar una oferta con otra. La competencia brinda a los consumidores la posibilidad de optar por aquel producto que satisfaga más su necesidad de acuerdo a su escala de valores. También estimula a los proveedores a mejorar las características de sus productos.
- *Especialización.* La competitividad promueve la mejora de los productos y por lo tanto la especialización. En el área del software y particularmente de los componentes, la existencia de mercados dónde poder ofrecer los productos sin necesidad de que formen aplicaciones completas, permite que ciertas organizaciones puedan especializarse en esos componentes mejorando notablemente sus prestaciones. Esto puede verse claramente en el área de los manejadores de bases de datos (DBMS).
- *Asimetría de información.* Este concepto refiere a la dificultad que tiene una parte en adquirir los conocimientos que tiene otra parte y que le permiten la fabricación de un bien o provisión de un servicio. Los mercados de software son formados cuando entre consumidores y proveedores se verifica esta asimetría de información. La adquisición de componentes de software sustituye la transferencia de conocimientos por la transferencia de productos. Se obtienen así no los conocimientos sino el usufructo de los mismos. Esto es aplicable tanto a cuestiones técnicas propias del desarrollo de los componentes como a otras cuestiones como pueden ser reglas o prácticas de negocio embebidas en los componentes.
- *Generación de estándares.* Los mercados estimulan la creación de grupos de productores y consumidores que determinan estándares para los productos como forma de unificar las diferentes soluciones y establecer características técnicas y de calidad, lo que beneficia a unos y otros.

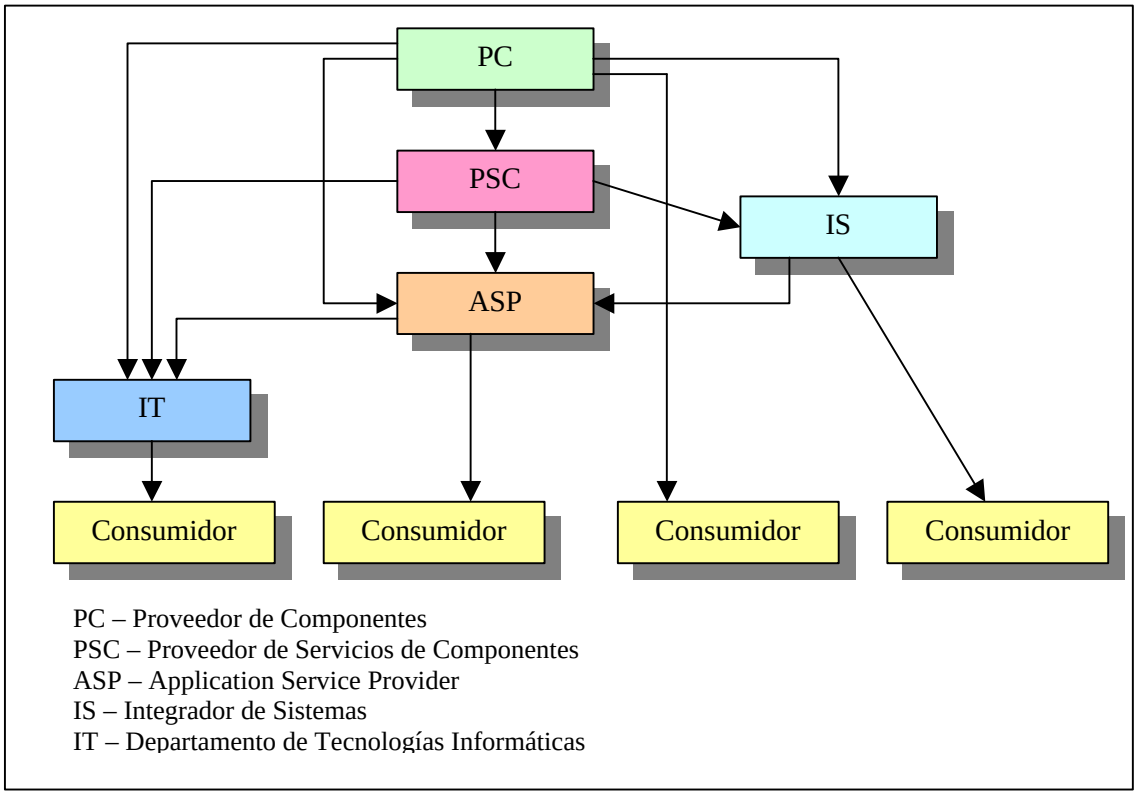

Figura 5. Estructura y Roles en un Mercado de Componentes

Sobre estas bases Sparling, por ejemplo, imaginó un esquema de mercado, dónde productores de componentes, proveedores de servicios, integradores de sistemas, departamentos de tecnología

de información y consumidores cumplen diferentes roles [Sparling00b], inclusive muchos de estos actores podrán comportarse como consumidores en determinado lugar de la cadena comercial y como productores o proveedores en otros.

La Figura 5 muestra los distintos actores y sus roles en un mercado de componentes. No hay mercado sin los consumidores, en este caso están considerados como quienes aprueban en definitiva los resultados del producto (o servicio) y quienes son responsables de la especificación de requerimientos. Algunos consumidores toman un producto como componente de lo que ellos a su vez ofrecerán.

Los proveedores de componentes son personas u organizaciones que desarrollan componentes aislados en función de requerimientos especificados por un consumidor o en base a su propio conocimiento de un dominio determinado.

Los proveedores de servicios de aplicación (ASP) son los herederos en épocas de internet del antiguo concepto de "*outsourcing*" de procesamientos de datos y aprovechan la necesidad de las organizaciones de liberar sus propios recursos tecnológicos contratando a terceras partes para el procesamiento de sus aplicaciones.

La forma de conexión de un usuario a la aplicación sería normalmente a través de un "browser" y eventualmente un "cliente fino". Lo que hace del CBD interesante para los ASP, es la facilidad de mantenimiento y flexibilidad que puede ofrecer un sistema basado en componentes, pensando en que se va a dar servicios a una diversidad de clientes con muchas cosas en común pero también con variantes significativas en aspectos determinados de la aplicación.

Los proveedores de servicios de componentes (PSC) brindan un servicio similar a los anteriores, pero en vez de tratarse de una aplicación completamente resuelta, se trata de la provisión de alguna funcionalidad especial en el marco de una aplicación mayor que la requiere. En este caso se espera que los clientes realicen su conexión por la vía de protocolos computador-computador, es decir que quien lo usa no es una persona sino un programa.

Los productores de componentes de software podrán venderlos a consumidores directos o a integradores de sistemas o a proveedores de servicios de componentes (PSC) o de aplicaciones (ASP). Estos a su vez podrán ofrecer sus servicios a consumidores finales o a los departamentos de IT de las organizaciones.

3.5.2. Importancia del mercado de componentes para el CBD.

La existencia de mercados de componentes promueve la sinergia entre productores y consumidores de componentes.

Los consumidores se ven beneficiados por la existencia del mercado en la medida que:

- *Impulsa el CBD.* Notoriamente la existencia de mercados hace más factible llevar adelante la estrategia de construcción de sistemas basados en componentes. Si bien esta es posible sin los mercados, obviamente las posibilidades de elección de componentes estarían limitadas a los desarrollos propios.

- *Acceso a componentes.* Los mercados bien organizados constituyen una forma rápida de acceder a los componentes requeridos.

- *Competencia.* La competencia promueve la mejora de la calidad de los productos, esto beneficia claramente a los consumidores. Por otra parte la existencia de propuestas

alternativas permite que los consumidores elijan la que mejor se adapta a sus necesidades específicas.

- *Guías de diseño.* La colección de componentes para un dominio específico puede ser una buena guía de diseño para los sistemas. Es notable por ejemplo, la estandarización de las interfaces interactivas con usuarios, logradas a partir de un conjunto de controles (botones, list box, etc.) que aparecen en los ambientes IDE (Integrated Development Environment) y que se pueden comprar a diversos proveedores.

- *Evolución tecnológica.* Los productores que compiten en el mercado van a tratar de permanecer dentro de los últimos avances tecnológicos, para no perder pie frente a sus colegas. La especialización que origina el mercado permite que los componentes se mantengan tecnológicamente actualizados.

- *Reducción de riesgos.* Los consumidores obtienen los componentes a precios razonables sin el riesgo que implica el desarrollo de los mismos. Hay quienes afirman que se trata de una transferencia de riesgo y no de una reducción del mismo [Haines97], pero desde el punto de vista de los consumidores esto es irrelevante.

Asimismo el CBD y la existencia de mercados favorecen a los productores de componentes de la siguiente manera:

- *Fuente de ingresos.* Los mercados establecen una opción clara para la comercialización de productos. Un buen estudio del mercado por parte de los productores, es la base para desarrollar los productos que detecten como necesarios a la vez que constituye el medio donde ofrecerlos. Con el desarrollo del CBD y los mercados de componentes, los proveedores pueden tener más claro en que productos vale la pena invertir y en cuales no, reduciendo así sus riesgos.

- *Especialización.* La existencia de mercados de componentes hace que los productores de software no tengan que presentar aplicaciones completas para la comercialización, sino que pueden especializarse en partes de estas. Esto implica que pueden dedicar un mayor esfuerzo en mejorar la calidad de esas partes y en mantenerlas actualizadas tecnológicamente como forma de mejorar su posicionamiento en el mercado.

- *Asociación entre productores.* El mercado de componentes es un excelente marco para la creación de soluciones integradas a partir de componentes de diversos proveedores o asociaciones de proveedores o "*joint ventures*", ofreciendo soluciones más completas a los consumidores.

3.5.3. La realidad del mercado de componentes.

Una búsqueda realizada en algunos sitios promocionados como "mercados de componentes" [Flashline] [Vbxtras] y [CompontSource] permite concluir que se pueden encontrar fácilmente componentes de GUI y utilitarios para diversas funciones (ayudas de desarrollo, conexión a bases de datos, etc.), pero en lo que respecta a lógica del negocio son muy pocas las cosas que se pueden ver, al menos en el área de los sistemas de gestión e información empresarial y lo poco que se encuentra, se trata, o bien de funciones aisladas vinculadas a algún sistema en particular o aplicaciones completas ofrecidas como componentes.

Esto puede significar que no hay componentes de mercado o que no hay visibilidad de los mismos (y por tanto no hay mercado).

En la búsqueda realizada, tampoco se pudo apreciar que se ofrecieran diferentes opciones (provistas por distintas compañías) para implementar una misma interfaz (diferentes componentes que ofrecieran el mismo servicio, con la misma característica de conexión), por lo que se carece en todo caso de la ventaja que daría un mercado real en cuanto a tener opciones diferentes de acuerdo a requerimientos variados.

Sea cual sea la causa, no se constata un mercado real de componentes de negocio.

El contraste entre la gran variedad de opciones de componentes GUI y las escasísimas ofertas de componentes de negocio, parece indicar que los fabricantes prefieren destinar sus esfuerzos en un área más redituable como es la de las interfaces gráficas. Pero cual puede ser la razón? Sólo con interfaces gráficas no se construye una aplicación.

La respuesta puede pasar por los modelos conceptuales y los modelos de referencia adoptados por las organizaciones que desarrollan sistemas de aplicación: mientras el modelo de interfaz gráfica es aceptado y compartido casi universalmente (a partir de las implementaciones primeras de Apple y las sucesivas versiones de Windows que impusieron un paradigma de comunicación con usuarios), no ocurre lo mismo con la estructura lógica de un sistema de aplicación para un dominio determinado, en particular el de los sistemas de información y gestión empresarial, donde no hay una "diseño dominante" que identifique los componentes esenciales del sistema.

Según Hopkins, *"... es lógico concluir que un mercado para componentes se desarrollará, a medida que la cantidad de proveedores aumente y la cantidad de consumidores constructores de sistemas aumente... Miles de aspectos enlentecen el surgimiento de un mercado de componentes entre los que se cuentan:*

- *Plataformas, un componente dado muchas veces solo puede ser usado en determinada plataforma o requiere múltiples implementaciones para cada plataforma diferente.*
- *Arquitectura, los componentes habitualmente son solamente útiles en el contexto de un sistema o framework más grande que les da estructura y semántica. Si un sistema no está basado en componentes o no tiene una arquitectura definida, los componentes disponibles tendrán menor valor o serán más problemáticos en la medida que el sistema intente conformar sus interfaces"*. [Hopkins00]

Por su parte en el documento de definición del perfil de UML para la arquitectura CCA (Component Collaboration Architecture), la OMG expresa "*La tecnología de componentes ha sido exitosa en sistemas de desktop e interfaz con usuarios. Pero esto no resuelve los problemas de la empresa. Recientemente los métodos y tecnologías para componentes de escala empresa han comenzado a aparecer. Esto incluye la 'sopa de letras'*[11] *de middleware como XML, CORBA, SOAP, Java, ebXML, EJB & .net. Lo que no ha emergido es la manera de hacer que con estas tecnologías se produzca una solución coherente para empresas y un mercado de componentes.* [OMG01b].

En el área de los sistemas de gestión e información empresarial (contabilidades, acreedores-deudores, inventarios, facturación, producción, etc.) no hay un "diseño dominante" y no están definidas ni hay consenso acerca de las partes que integran esas aplicaciones.

A modo de ejemplo el *Request for Proposal* de OMG (Object Management Group) de abril de 2001 [OMG01a], relativo a la definición del conjunto de interfaces de un módulo para manejo de Deudores y Acreedores para interactuar con sistemas de contabilidad central y sistemas de ventas, tiene al presente una única propuesta [OMG02] de marzo de 2002.

[11] Textual en el texto original

En el área contable se sigue manejando el modelo de partida doble descrito en el Renacimiento por Luca Pacioli cuya implementación informática (insólitamente) admite muchísimas variantes. Existe un modelo alternativo: REA (Resource-Event-Agent) [McCarthy82] que integra la contabilidad a otras necesidades del proceso empresarial, con un diseño más formal, utilizando elementos del modelo relacional, pero su difusión no se constata fuera del círculo del cual se extrajo la referencia.

No se cuenta con una definición compartida de clases de negocio, en el sentido de conceptos de negocio, que permita establecer con claridad los componentes que se deben manejar y su interrelación. En este estado de las cosas es muy poco probable que una organización de desarrollo de software se arriesgue a producir componentes sobre los cuales no hay definido un mercado de consumidores.

Paradójicamente, el área de los sistemas de gestión e información para empresas es una para la que más se han desarrollado “paquetes” comerciales con soluciones completas, existiendo un mercado real de las mismas. Es probable que los fabricantes estén migrando o hayan migrado sus productos hacia arquitecturas de componentes [Sprott00], por motivos de flexibilidad y adaptabilidad a los cambios, sin embargo insisten en vender sus aplicaciones en forma monolítica y no los componentes como funcionalidades independientes [Wilkes02].

Alrededor de esos sistemas, cuando las necesidades y las arquitecturas están bien definidas como en el caso de SAP, aparecen una gran cantidad de componentes de terceros. En ese caso, además, la propia compañía promueve desde su sitio web el uso de estos componentes, en definitiva promoviendo un mercado de componentes para su producto [SAP02].

3.6. El impacto de las líneas de producto en el CBD

El desarrollo de líneas de productos para proveer un marco de re-uso de componentes y de la arquitectura del sistema, representa una idea bastante menos ambiciosa que la del desarrollo de mercados generales de componentes, ya que mientras éstos están orientados a toda la industria de software, las líneas de producto en general no traspasan los límites de una organización que las produce y re-utiliza. Sin embargo representa una alternativa mucho más realizable en la práctica, especialmente en el dominio de los sistemas de información y gestión de procesos empresariales [Bosch00].

El concepto general de las líneas de producto fue descrito en la sub-sección 2.3.2 y los aspectos de diseño para construirlas y su utilización para sintetizar las diferentes aplicaciones particulares será desarrollada en la sección 4. En esta sub-sección se estudiará en particular el impacto de esta estrategia en el proceso de software de una aplicación particular y en la organización que produce el software.

La evolución de las líneas de productos se inscribe en un ciclo de madurez, en el que usualmente la primera versión corresponde o coincide con el desarrollo de una aplicación en particular, en la cual se hace una primera identificación de los aspectos comunes del dominio.

El desarrollo de nuevas aplicaciones particulares puede introducir variantes en el diseño original de la arquitectura de la línea de productos, en la medida que se identifiquen nuevos aspectos comunes del dominio, pero normalmente ya en esta etapa se va a hacer uso de los componentes comunes identificados en la primera instancia. En esta etapa del ciclo de madurez se desarrollan nuevos componentes específicos para las distintas variantes de los miembros de la línea de productos.

Finalmente se alcanza un estado de madurez en que las nuevas aplicaciones pueden ser enteramente sintetizadas a partir del conjunto de componentes comunes y las variantes para aplicaciones particulares ya desarrolladas, relacionadas entre sí a través de la arquitectura de la línea de productos.

En las etapas iniciales se trata de determinar las características y requerimientos generales de un dominio, para de ese modo realizar un diseño de solución general, que incluya una arquitectura

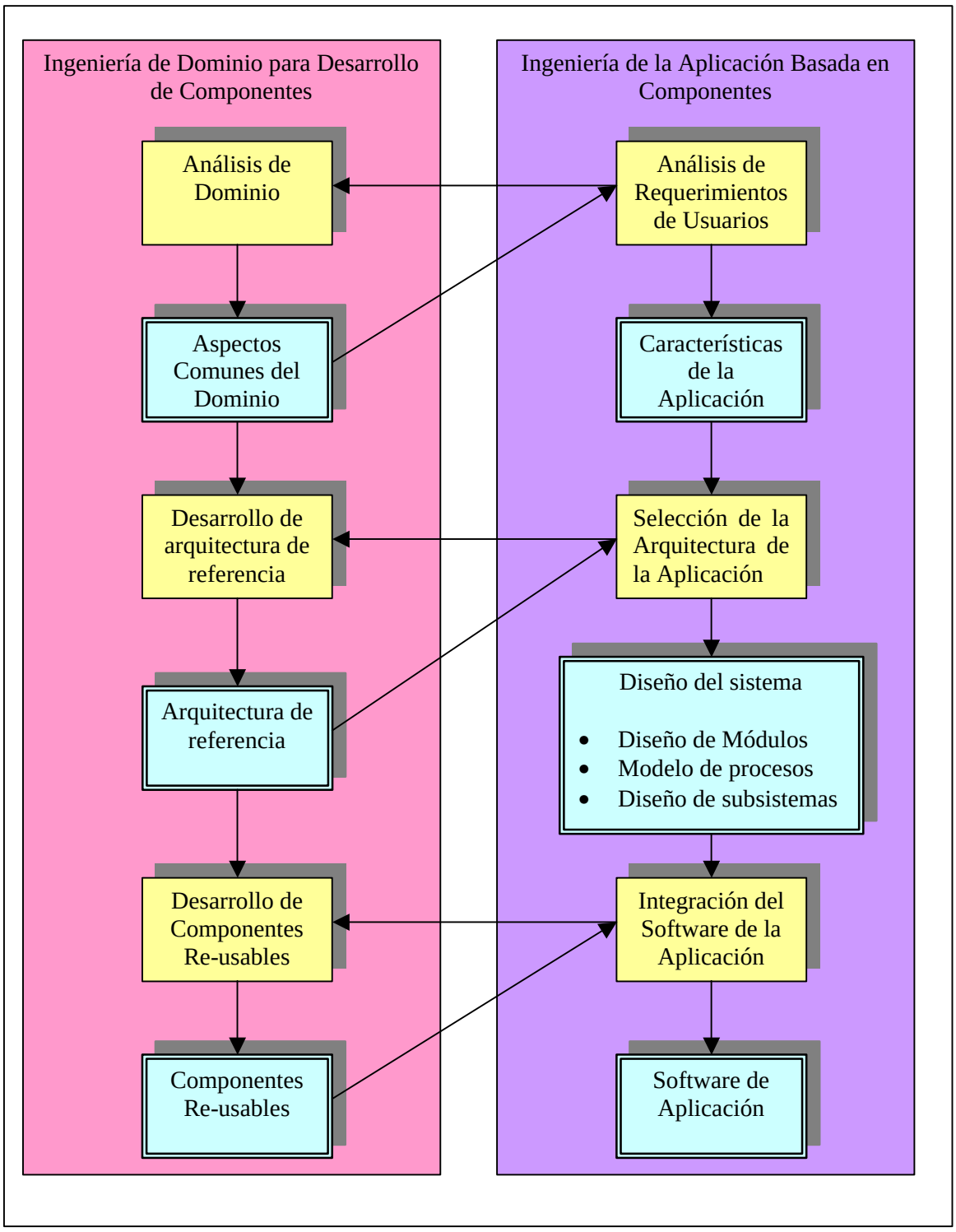

Figura 6. Proceso de Ingeniería de Dominio y de Ingeniería de Aplicación

de la línea de producto y la identificación del conjunto de componentes re-usables en el marco de esa arquitectura. Esta actividad corresponde a profesionales especialmente enfocados en el dominio de aplicación a través de disciplinas de ingeniería y análisis de dominio [Kean98].

En las etapas más avanzadas del ciclo de madurez de las líneas de producto, el equipo de profesionales especializados en el dominio, seguirá trabajando en aspectos de mantenimiento y evolución de la línea de producto, manteniendo su óptica orientada al dominio y no a una aplicación en particular.

A partir de estas consideraciones, se puede definir dos procesos de software que estarán interrelacionados: el primero, orientado al análisis de requerimientos comunes a partir del cual se elabora el diseño, desarrollo y mantenimiento de las arquitecturas y de los componentes que constituyen las líneas de producto y el segundo proceso, orientado a la construcción de aplicaciones particulares a partir de la línea de productos, eventualmente realizando adaptaciones particulares a los componentes existentes o extendiendo su funcionalidad. El segundo proceso de software se alinea con el proceso descrito en la sub-sección 3.3.

La Figura 6 muestra las correspondencias entre los procesos de software de ingeniería de dominio y de ingeniería de la aplicación particular. En el esquema las actividades se representan como cuadros simples y los subproductos como cuadros de marco doble. Las flechas representan las relaciones y dependencias entre procesos, el flujo de información y la utilización de artefactos de un proceso en el otro.
Lo que interesa resaltar son las relaciones y dependencias entre ambos procesos y los mecanismos mediante los cuales la organización puede soportarlos.

Cada etapa del proceso de la aplicación particular aporta nueva información que es elaborada en el proceso de la línea de productos, contribuyendo a ampliar el conocimiento acerca del dominio y plantear nuevas necesidades. A su vez, cada actividad del proceso de ingeniería de dominio elabora determinados artefactos (documentos de requerimientos comunes, especificación de arquitecturas, componentes re-usables, etc.), los que pueden ser usados en el proceso de la aplicación particular. Los artefactos que elabora el proceso de ingeniería de dominio son compartidos por todos los productos, mientras que los que elabora el proceso de ingeniería de la aplicación son propios de una aplicación en particular [Kang99].

Esta forma de encarar el desarrollo requiere de ciertos cambios organizacionales, en particular la formación de equipos con orientación al dominio, con capacidades para mantener los requerimientos comunes, diseñar y mantener una arquitectura de la línea de productos y desarrollar y mantener los componentes comunes.

Asimismo se deberían formar otros equipos de ingeniería de aplicación, capaces de interpretar y adaptar la arquitectura y componentes comunes a una aplicación en particular, en un proceso que implica fundamentalmente la integración y ensamblado de partes o la composición de la aplicación.

3.7. CBD, una alternativa alentadora.

La aplicación del paradigma CBD impacta en los procesos de elaboración de software y en la estructura organizacional.

Ya sea por la existencia de componentes prehechos, que hacen innecesario el desarrollo o por la forma misma del desarrollo de componentes, que se puede realizar en paralelo y con

independencia del proyecto en particular, los procesos y ciclo de vida del software se ven modificados y el *time to market* reducido considerablemente.

Esto es bueno, ya que frente a nuevas necesidades, se logran tiempos de desarrollo menores y mejor velocidad de respuesta. Adicionalmente, la flexibilidad de los sistemas basados en componentes, soportada por la posibilidad de reemplazo de piezas funcionales o por las facilidades de extensión del sistema, permiten una respuesta rápida o bastante rápida frente a la evolución de los negocios.

En ese sentido el paradigma CBD se presenta como una alternativa alentadora para el desarrollo de aplicaciones flexibles, configurables y componibles, que puedan acompañar los procesos de mejora continua de la empresa.

Sin embargo hay algunas cosas que deben estar en su sitio para que se pueda establecer el desarrollo basado en componentes.

En primer término, las estructuras de las organizaciones que producen el software deben ajustarse para asimilar el proceso en paralelo de los componentes. Se deben enfrentar problemas como la pérdida de visión global del proyecto o de objetivos particulares. Estos suelen ser subsanados por la figura del arquitecto de sistema que mantiene la visión global de la aplicación y apoya a los grupos de desarrollo para que se mantengan alineados con el proyecto [Sparling00a].

Asimismo en la organización se debe crear una cultura de re-uso, ya sea en el desarrollo de componentes como en su utilización. Algunas organizaciones han usado estrategias de estímulo del re-uso, mediante las cuales, por ejemplo, se premiaba económicamente a los programadores por cada invocación en su código a componentes prehechos y también a quien había hecho el componente, por cada vez que éste era invocado [Pfleeger01].

En segundo término, la verdadera potencia del paradigma CBD se alcanza cuando se puede verificar la existencia de componentes prehechos. En particular, componentes que encierren diferentes aspectos de la lógica de la aplicación. Estos deberían ser provistos por un mercado de componentes, que habilite el re-uso a todo un sector de la industria de software o por el desarrollo de líneas de productos, en las que el re-uso se limita a una organización y a los diferentes usuarios de la línea de productos (en el caso de líneas de productos de aplicaciones estandarizadas puede ser una cantidad muy grande de usuarios).

La segunda opción, si bien menos ambiciosa, parece mucho más alcanzable en la realidad, dado que no se verifica la existencia de un mercado de componentes para el dominio de los sistemas de información y gestión empresarial y que tampoco se identifica un “diseño dominante”, que promueva el desarrollo de componentes de mercado para el mismo.

En resumen, la solución al problema de la composición de procesos de la empresa, podría ser abordada por la estrategia de desarrollo de líneas de productos. Aunque la composición de los procesos del negocio no se logre directamente por quienes definen las estrategias de la empresa, al menos se puede contar con un mecanismo eficiente para dar respuestas rápidas a nuevos requerimientos.

Los resultados que se puedan obtener, van a depender, por un lado, de cómo la organización de software pudo estructurarse para absorber la construcción de componentes en paralelo y los procesos de ingeniería de dominio concurrentemente con los de ingeniería de aplicación.

Van a depender, por otro lado, de la calidad de diseño y desarrollo de las líneas de productos, en lo que tiene que ver con aspectos funcionales, pero también en cómo se interpretan las características de diseño requeridas para este tipo de sistemas.

En la sección 4 se estudia con mayor detenimiento las características más relevantes de los sistemas basados en componentes, los principios de diseño que aseguran el cumplimiento de esas características y un par de ejemplos de diseño concretos.

4. Diseño de sistemas basados en componentes.

La construcción de sistemas basados en componentes es una disciplina que se ha desarrollado con aceptación creciente en los últimos años (particularmente en los últimos seis) y sobre la cual se siguen realizando estudios y proponiendo diferentes puntos de vista para su puesta en práctica. Más allá de razones vinculadas a seguir las tendencias tecnológicas o "modas" en cuanto a métodos o paradigmas de desarrollo, el CBD tiene sus razones bien fundadas en que apunta a satisfacer algunas características requeridas para el desarrollo de software que han sido una constante histórica y un objetivo siempre perseguido, estableciéndolas como características propias del paradigma.

El hecho de que no se verifique un mercado real de componentes de negocio hace perder una de las ventajas teóricas importantes del CBD, pero hay otras ventajas no menos importantes que mantienen la vigencia de este paradigma. En particular la estrategia de desarrollo de líneas de productos posibilita contar con una arquitectura y un conjunto de componentes prehechos, a la hora de elaborar una aplicación particular. El desarrollo de líneas de producto implica el desarrollo específico de la casi totalidad de componentes y su alineación con una arquitectura definida.

En esta sección se estudian más en detalle los rasgos principales u objetivos que se persiguen con el desarrollo basado en componentes y cómo es que estos se logran, se estudian además los principios que guían el diseño de las arquitecturas de sistemas y de los componentes mismos y se presentan dos ejemplos de arquitecturas concretas, las que son evaluadas en base a esos principios y comparadas entre sí.

4.1. Características de los sistemas basados en componentes.

En primer término se repasan tres rasgos principales que se espera encontrar en un sistema basado en componentes y que en definitiva son los que motivan este tipo de desarrollo.

Estos son:

- Re-uso
- Extensibilidad
- Composición

Los tres rasgos son especialmente importantes ya que la manera en que sean resueltos por los diferentes modelos de desarrollo, determinará la posibilidad de armar una solución aplicativa para empresas a partir del conjunto dado de componentes. Estos rasgos son estudiados en las siguientes sub-secciones.

Hay otras características como flexibilidad y mantenimiento que en realidad pueden verse como consecuencia de una buena resolución para las tres precedentes.

4.1.1. Re-uso.

La posibilidad de re-uso de partes es una característica inherente al desarrollo CBD y una de sus motivaciones principales, pero no es exclusiva de este tipo de enfoque: el desarrollo orientado a objetos y las técnicas de análisis y diseño orientado a objetos apuntan a este mismo objetivo, al igual que sus antecesoras análisis y diseño estructurado y modular [Fröhlich01].

El concepto de *re-uso* se ha usado de formas muy diferentes, normalmente en toda situación en la que se puede aprovechar algo hecho anteriormente (código en forma de programas o subrutinas, pero también diseño y arquitecturas de software). Eso ha sido una práctica habitual no solamente en la ingeniería de software sino de prácticamente cualquier proceso de diseño y construcción. Sin dejar de notar lo importante (en ahorro de tiempo y costos) que puede ser aprovechar trabajos realizados anteriormente como base de nuevos desarrollos, interesa un concepto más estricto de re-uso: la utilización de un software prehecho, que se ajuste (o se pueda adaptar) a las necesidades concretas de un sistema, sin necesidad de meterse en aspectos internos de ese software.

Un componente de software es re-usable si y solo si, para usar esa pieza de software no es necesario tener acceso a su código fuente. Solo es necesario vincularse con la biblioteca estática o incluir la biblioteca dinámica que lo contiene. Cualquier cambio o mejora futura que se haga sobre ese componente, será accesible como una nueva versión del mismo y podrá utilizarse cuando se crea oportuno [Martin96c].

Los beneficios del re-uso son compartidos por todos, sobre todo desde el punto de vista del tiempo de desarrollo, mantenimiento de sistemas, adquisición de conocimientos y los costos involucrados, sin embargo hay algunos aspectos que dificultan su puesta en práctica, como lo señala Michael Sparling en [Sparling00a]:

- *El síndrome de diseño para re-uso*, desincentiva el uso de las piezas de software prehechas, por la razón de que para los programadores es más atractivo el diseño de nuevos componentes para ser re-usados, que el re-uso mismo de componentes hechos por otros. Por otro lado la medición de productividad en líneas de código también desincentiva el re-uso ya que crea la errónea idea de menor productividad, un programador puede escribir 10.000 líneas de código desarrollando un buen componente pero escribirá no más de 10 para usarlo (lo cual en realidad es más provechoso).

- *La idea falaz de que todo componente debe ser re-usable*, aunque la organización pueda identificar o no cuales van a ser los requerimientos futuros para el componente. Se produce un gran esfuerzo dirigido a la creación de componentes que manejen diversas variantes de un caso, sin que haya una razón de negocio que lo apoye. La regla del 80/20 se aplica bien: en el 80% de los casos sólo el 20% del código escrito es re-usado.

- *La complejidad de los componentes*, resultantes de querer abarcar todas las posibilidades en vez de apuntar a una funcionalidad específica y concreta, dificulta la comprensión de los mismos y hace mucho más difícil su re-uso.

También es interesante señalar al respecto, las diferentes escalas o alcances que puede adoptar el re-uso y su verificación en la práctica [Bosch00]:

- *En el ámbito de un proyecto*, es la forma de re-uso más habitual y el propio diseño de los sistemas en particular lo estimula ya sea en diferentes partes del mismo o en sucesivas versiones.

- *En el ámbito de una organización*, es la forma de re-uso que la industria intenta lograr en este momento, ya sea cuando una organización utiliza componentes de sistemas diferentes en el desarrollo de nuevas aplicaciones o cuando se diseñan y desarrollan "líneas de productos" que comparten un conjunto de requerimientos y a pesar de las variantes que las distinguen tienen un conjunto de componentes en común que son re-usados en las aplicaciones particulares..

- *En el ámbito de un domino*, por diferentes organizaciones, implementado a partir de un mercado de componentes, esta forma de re-uso se verifica casi en exclusividad con relación a algunos componentes tecnológicos (GUI, DBMS, etc.), pero no se verifica para componentes de negocio, aunque es uno de los principales objetivos del CBD y muchísimos autores se han explayado en el tema, ya que no existe un mercado real de componentes de negocio.

Sin duda esta última forma de re-uso es la que apuntaría a solucionar de mejor manera las hipótesis planteadas, en particular la posibilidad de que analistas de negocio puedan "armar" sus procesos informáticos mediante re-uso de "partes" obtenidas de un mercado.

Se debe agregar que para que un componente pueda ser efectivamente re-usado debe cumplir tres propiedades, debe ser: localizable, consumible y extensible. Para que esto se cumpla en especial debe tener una especificación completa de funcionalidad ofrecida y requerida, interfaces, contexto y modo de uso [Sparling00a]. Muchas veces sucede que quienes necesitan un determinado componente no hacen o tienen dificultad para hacer una especificación formal del mismo, en forma recíproca quienes lo ofrecen a veces también encuentran la misma dificultad.

En la experiencia personal del autor de este trabajo[12], el re-uso de componentes por un grupo de desarrolladores, se ha verificado con mayor éxito en los casos en que la funcionalidad del componente era concreta, documentada y bien adecuada al contexto. Asimismo, en la mayoría de los casos, el re-uso de los componentes respondía a especificaciones de diseño (top-down), en otros casos los componentes surgieron durante el desarrollo y luego por decisión del grupo de ingeniería de software se realizaron las modificaciones necesarias para que todas las aplicaciones dentro del sistema los usaran, sustituyendo desarrollos particulares previos. Este último caso obviamente estaría en contradicción con los beneficios del re-uso, pero favoreció el desarrollo de versiones posteriores, inscribiéndose estos componentes dentro del re-uso planificado.

4.1.2. Extensibilidad.

"Un sistema de software es considerado extensible, solamente si en su diseño incluye ciertos medios, bien documentados, de agregarle funcionalidad". [Fröhlich01].

En el paradigma de análisis estructurado esto es posible a través de variables de procedimiento y configuración, en el paradigma de orientación a objetos los mecanismos de herencia y polimorfismo constituyen el soporte por excelencia para la extensibilidad, sin embargo en todos estos casos quien realiza la extensión debe o bien formar parte del equipo que hizo el desarrollo original o al menos acceder al código fuente original. En estos casos la extensión, de hecho, se hace efectiva en tiempo de compilación.

El paradigma de desarrollo basado en componentes rompe con este concepto centralizado de extensión:

- Cualquier interesado puede realizar una extensión
- Nuevas extensiones se pueden incorporar en cualquier momento

[12] Desarrollo de sistemas de información y gestión para empresas en AT&G Informática, y en particular el sistema P-SIG, un ERP desarrollado con lenguajes procedurales, principalmente COBOL, actualmente en proyecto de reconversión con tecnologías de orientación a objetos, usando principalmente Delphi. En estos desarrollos trabajan concurrentemente unos doce desarrolladores (entre analistas y programadores) y a lo largo del tiempo han trabajado más de veinticinco personas diferentes.

Fröhlich denomina este tipo de extensibilidad como *extensibilidad distribuida*, otros nombres usados a veces son *extensibilidad independiente* y *extensibilidad dinámica* [Fröhlich01].

Uno de los problemas a resolver es el del mecanismo de extensión. En las tecnologías de orientación a objetos el mecanismo por excelencia de extensión es la herencia y el polimorfismo; estos mecanismos presentan el problema del *fragile base class* sintáctico y semántico [Persson00] [Fröhlich01], que puede ser resuelto en algunos casos por el lenguaje de programación y convenciones de diseño y eventualmente por análisis del código cuando ocurre. Un ejemplo del problema de "fragile base class", donde se hace ineludible al análisis del código, se presenta en el Anexo 1.

Estas soluciones crean problemas cuando tratamos con extensibilidad distribuida, ya que el código fuente no estará disponible y por tanto pueden surgir dudas acerca de cómo implementar la sobreescritura o no se sabrá si fueron observadas las convenciones de diseño que evitan el problema del fragile base class. El mecanismo propuesto en general es el de *forwarding o delegación* [Szyperski97] [Fröhlich01].

Este mecanismo propone que cuando un objeto recibe un mensaje invocando un método que no implementa, lo reenvía hacia otro objeto (normalmente el que hubiera sido su superclase en el mecanismo de herencia). En ese caso hay un *objeto exterior* que recibe inicialmente los mensajes y los que él no implementa los reenvía a un *objeto interior*[Szyperski97].

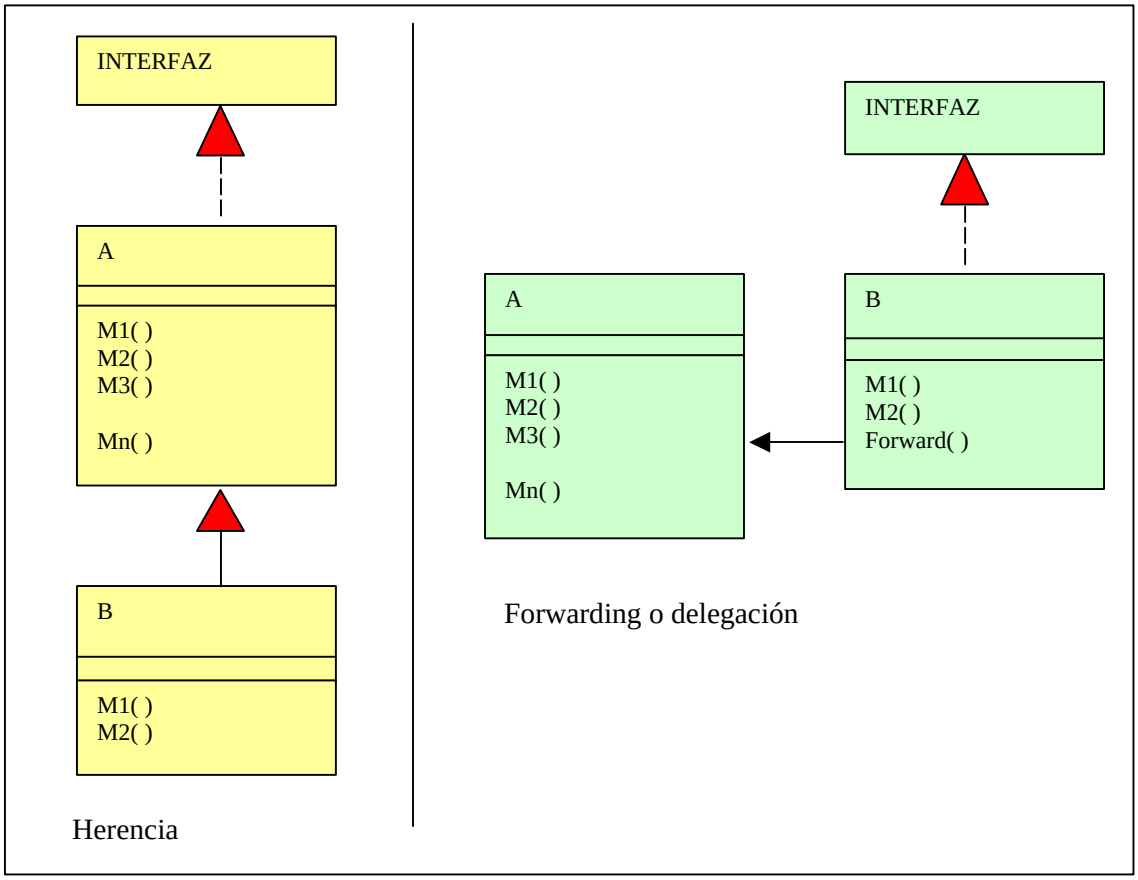

Figura 7. Extensión de los métodos M1 y M2 con mecanismos de herencia y forwarding

.

Así, como muestra la Figura 7, cuando se quiere extender el comportamiento de una clase, en vez de derivar una subclase con la nueva implementación, se define una clase nueva que implementa la función con el agregado que corresponda, probablemente invocando al objeto

original para la implementación básica del método. Los otros métodos del objeto original que no se quiera modificar simplemente se resuelven por forwarding o delegación a ese objeto.

El lenguaje orientado a componentes *LAGOONA* [Frölich01] ofrece una implementación particular del mecanismo de forwarding: la *delegación genérica* (*"generic forwarding"),* mediante el uso de un método por defecto, que se activa cuando un objeto recibe un mensaje que no reconoce. En la implementación de este método se puede hacer que el mensaje sea reenviado hacia otros objetos. De esta manera se pude extender métodos específicos de un componente sin necesidad de conocer todos los métodos que implementa.

Sin embargo puede haber dos problemas con el mecanismo de delegación por defecto: por un lado no se puede garantizar que todos los mensajes sean resueltos, ya que es posible implementar el método por defecto sin que haga el reenvío, o vacío como forma de ignorar todos los mensajes que lleguen al componente y que no sean implementados por éste. Por otro lado el mecanismo de delegación genérica puede resultar en una *"adquisición súbita"* o *"no planeada"* de funcionalidades que no se deseaban, cuando los componentes en forma automática reenvían cualquier mensaje que no reconocen, hacia otros objetos [Frölich01].

Las extensiones que se incorporan al comportamiento de los objetos, ya sea agregando nuevas funcionalidades o mejorando las ya provistas, suelen ser el resultado de la evolución misma de las aplicaciones, cuando están en etapa de producción o cuando se quiere aplicar un componente a un contexto diferente de aquel para el que fue pensado originalmente y aparecen nuevas necesidades.

Fred Brooks dice que *"es imposible para un cliente, aún trabajando con un ingeniero de software, especificar completamente, en forma precisa y correcta los requerimientos exactos de un producto de software moderno antes de haber utilizado algunas versiones del mismo."* [Brooks87]. En lo que refiere a los componentes, como toda pieza de software, van a incrementar y mejorar su funcionalidad a lo largo del tiempo a través del refinamiento de los requerimientos, pero también en función del re-uso y la aplicación en contextos diferentes.

No solo las extensiones, sino el mecanismo por el cual éstas son incorporadas va a ir variando a lo largo del tiempo. Se puede describir un ciclo de madurez de los componentes con relación a como se implementa la extensibilidad.

Un escenario típico en el ciclo de madurez de un componente sería el que muestra la Figura 8 [Collins00].

Al principio el componente se desarrolla y usa en un contexto particular, normalmente como parte del proyecto para el cual fue hecho. En esta etapa el componente suele ser muy poco adaptable, en el sentido que no interesa gastar recursos en otorgarle funcionalidad variable o puntos de variación.

La siguiente etapa ocurre cuando el componente es re-usado por primera vez, habitualmente habrá un cierto trabajo de adaptación para el nuevo contexto, de acuerdo a variantes en los requerimientos o dependencias, luego de lo cual el componente habrá ganado en generalidad y aumentarán sus probabilidades de re-uso. Para manejar diferentes casos que se presenten aparece un mecanismo de extensibilidad interno (compilado), probablemente como implementación del pattern *strategy* [Gamma95].

La etapa final ocurre cuando los diseñadores proveen al componente de mecanismos de extensión de manera que esta pueda ser efectuada sin necesidad de acceder al código, esto se hace en general mediante llamadas del componente a una clase de extensión que se puede

implementar como subclase de una clase provista por el paquete o por algún mecanismo de forwarding.

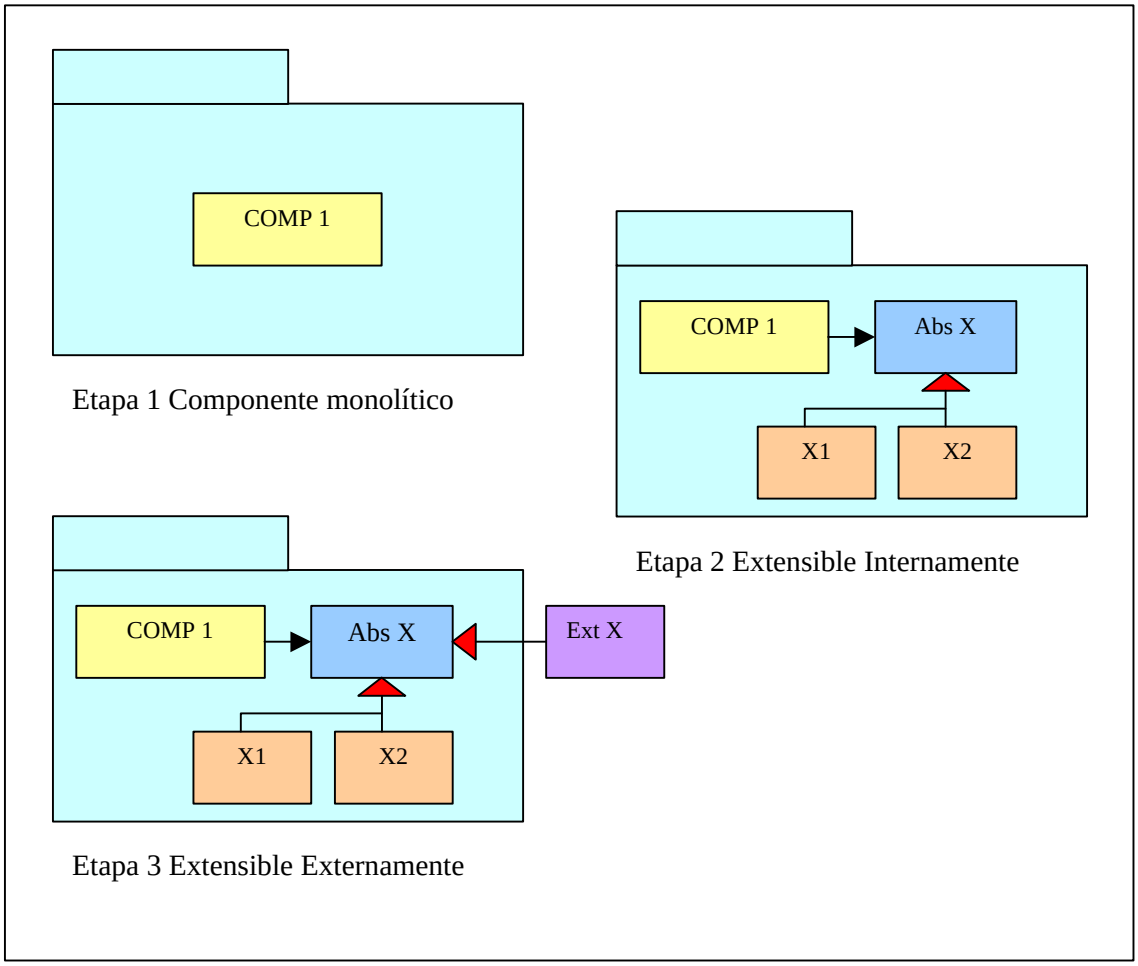

Figura 8. Etapas de madurez en la extensibilidad de un componente

Para llegar a esta etapa debe haber transcurrido un tiempo quizás prolongado, en que los diseñadores refinaron sus conocimientos acerca de los usos posibles para el componente. Un componente en este nivel de madurez estaría habilitado a trabajar en la casi totalidad de los contextos que requieran su funcionalidad. Estos puntos de extensión previstos en los componentes son conocidos como *puntos de variación* (*"variation points")*.

4.1.3. Composición.

"No estamos ni cerca de una comprensión universal de lo que son las técnicas de composición", Clemens Szyperski, agosto de 2002 [Szyperski02].

Este es el aspecto del desarrollo de sistemas basados en componentes (CBD) que más se relaciona con las inquietudes que motivaron este estudio. Las características vistas antes: re-uso y extensibilidad apuntan al desarrollo de sistemas menos costosos, de menor demora de desarrollo, más adaptables al contexto y a los cambios. En cambio la composición es la característica de los sistemas CBD que permite construir en forma lógica una aplicación basándose en una selección de piezas (los "building blocks" que aparecen con frecuencia en la literatura del tema) y la interconexión de las mismas.

La composición está directamente relacionada con el modelado de una solución para la aplicación. Si un analista empresarial quisiera armar un sistema para sus procesos de negocio, podría hacerlo, al menos conceptualmente, mediante composición; las características de re-uso y extensibilidad van a hacer que esto sea posible, pero la composición va a ser la actividad fundamental de ese proceso.

Las cosas no son tan sencillas como juntar ladrillos Lego[13] y ver que forma va adquiriendo el sistema, se necesita:

- Contar con una definición precisa de componentes, es decir que cosas van a ser las partes del sistema: Clases? Bibliotecas de funciones? Binarios ejecutables? Código fuente?
- Poder contar con componentes prehechos, ya sea catalogados en un repositorio o un mercado de componentes o por elaborar la aplicación particular a partir de una línea de productos, que ha desarrollado la organización.
- Que las piezas (componentes) puedan conectarse entre sí, para eso se cuenta con el middleware y las tecnologías de interoperabilidad, las cuales, optando por un modelo u otro, brindan el CEE (Component Execution Environment) para la conexión física de los componentes y así componer la solución.
- Definir una arquitectura del sistema a componer, usando el término arquitectura en el sentido de las interrelaciones estructurales de los componentes que hacen posible o forman un sistema [Collins01a]. Una arquitectura de software es un "template" usado para definir como es que los componentes "encajan" entre sí para formar un sistema [Batory94].

"La arquitectura de un software describe su organización estática en subsistemas interconectados a través de interfaces y define a nivel significativo como interactúan estos subsistemas entre sí" [Griss01].

Más allá del re-uso oportunista de piezas de software, un enfoque más disciplinado y basado en principios requiere que los componentes sean usados en el contexto de una arquitectura específica. Cuando una organización maneja una arquitectura en particular y realiza variaciones para producir diferentes productos interrelacionados para un dominio, se da origen a una línea de productos. En ese caso la composición se da en un ambiente estático, se puede prever la composición de un miembro de la línea de productos antes de su liberación y despliegue [Szyperski01].

"Una arquitectura de línea de productos es un diseño que identifica los componentes contenidos en los miembros de la familia, habilitando la síntesis de cualquier miembro en particular por composición de estos componentes" [Batory01]. En este caso la composición se realizaría seleccionando los componentes adecuados entre el conjunto de componentes que conforman la misma interfaz.

La idea de las líneas de productos es la de ahorrar el costo de tener que materializar todas las variantes de los miembros de la familia, en cambio contar con los diferentes componentes y mediante composición construir el sistema específico cuando este sea requerido.

Cada componente puede ser visto como una máquina virtual que implementa una interfaz (brinda un servicio). En el marco de una línea de productos puede haber muchos componentes que implementen la misma máquina virtual, es decir que conformen la misma interfaz. El conjunto de todos los componentes en una línea de productos que implementan la misma máquina virtual se denomina "realm" (reino) [Batory97] [Batory98].

[13] LegoSystems, juego de niños basado en ladrillos de encastre con los que se pueden armar muy diversas construcciones a partir de un conjunto no muy extenso de ladrillos diferentes.

Los realms se pueden expresar mediante una notación de conjuntos, dónde para cada realm se incluyen todos los componentes que implementan una máquina virtual específica:

S:={a, b, c}
T:={d(S), e(S), f(S)}
W:={n(W), m(W), p , q(T,S)}

S, T y W son todos ejemplos de realms que incluyen distintas versiones para la implementación de las máquinas virtuales que representan. En cada realm, los componentes incluidos son "plug compatibles" ya que implementan la misma interfaz. Por ejemplo, los componentes a, b y c son plug compatibles ya que todos pertenecen al realm S, es decir implementan la máquina virtual S. Lo mismo ocurre con los componentes correspondientes a los realms T y W.

Todos los componentes de un realm implementan una máquina virtual dada, es decir "exportan una interfaz", pero puede ocurrir que algunos de ellos requieren además funciones de otros componentes, es decir que dependen de otros componentes de acuerdo a una especificación de interfaz; esto se denota como un parámetro del componente. En el ejemplo todos los componentes del realm T dependen de una interfaz del realm S, mientras que en el realm W solo algunos componentes dependen de una "interfaz importada" y en el realm S no existen dependencias.

Una aplicación jerárquica se puede definir como una serie progresiva de máquinas virtuales cada vez más abstractas (unas importadas por otras) y su implementación se expresa como una composición de componentes denominada ecuación de tipos:

A1 = d(b)
A2 = f(a)

Siguiendo el desarrollo del ejemplo, en el marco de una línea de productos, las ecuaciones de tipo A1 y A2 implementan la misma máquina virtual y son plug compatibles, ya que están formadas por componentes d y f del realm T, que son a su vez plug compatible. En cada una de esas ecuaciones los componentes del realm T importan diferentes componentes del realm S.

Posiblemente los resultados de proceso van a ser diferentes (las diferencias pueden ser en cuanto a rendimiento, recursos requeridos o reglas de negocio aplicadas) pero en ambos casos se satisface la misma interfaz, en otras palabras: representan las variantes en una línea de productos.

En el realm W del ejemplo, se da una situación particular: algunos de los componentes (n y m) dependen de componentes del propio realm W. Este tipo de componentes se denomina componentes simétricos y con ellos se podrían definir ecuaciones de tipo del estilo:

A3 = n(p)
A4 = n(m(p))
A5 = m(n(p))
A6 = n(n(p))
A7 = m(m(p))

Las últimas dos muestran, además, que sería posible componer un componente con sí mismo. El uso de componentes simétricos, permite preparar mecanismos de extensibilidad, estos componentes son típicos del estilo de arquitecturas *pipe & filters*.

Como se ha dicho, cualquier componente del realm S del ejemplo, puede ser una instancia del parámetro de los componentes del realm T. Las ecuaciones de tipo pueden ser sintácticamente

correctas pero resultar semánticamente incorrectas. Como dice Bertrand Meyer, *"...También necesitamos especificaciones semánticas. Especificar exclusivamente el tipo es como si solo se especificara el diámetro de los conectores de corriente eléctrica. La ficha puede tener la forma y diámetro correctos, pero el voltaje puede ser inadecuado..."* [Meyer00].

En otros términos, hay restricciones propias del dominio que deben ser satisfechas adicionalmente para implementar una ecuación de tipo que tenga sentido. Estas restricciones se denominan *reglas de diseño* para un dominio y por lo tanto una línea de productos se puede expresar como el conjunto de realms y sus relaciones más el conjunto de reglas de diseño. Estas reglas de diseño se pueden establecer como *contratos* de los componentes: precondiciones y poscondiciones y prerrestricciones y posrestricciones [Batory99].

En definitiva, a través de esos mecanismos las líneas de productos presentan un marco estático adecuado, donde la composición responde a reglas de diseño predefinidas y dónde los productos resultantes se van a ajustar a una arquitectura de software preestablecida.
Otros enfoques, como los señalados en [Allen00], basados en composición oportunista o en enfoques bottom-up, no son tan rigurosos y presentan dificultados de llevar a la práctica al no existir un auténtico mercado de componentes de negocio.

4.2. Diseño de sistemas de componentes.

Uno de los requerimientos para aplicar las características señaladas es contar con una definición precisa del concepto de componentes, es decir de las partes que integran el sistema.

En este estudio se toma como definición de componente la siguiente:

> *Unidades binarias reemplazables de despliegue, que implementan una o más interfaces bien definidas, dando acceso a un conjunto de funcionalidades interrelacionadas y que puedan adaptar su comportamiento en forma predefinida.*

Esta definición resalta los componentes como unidades binarias de despliegue, es decir son ejecutables.

También se establece la necesidad de una definición precisa de las interfaces, como toda parte de cualquier sistema (una máquina, un puente, etc.) es imprescindible una especificación de su uso y relaciones con otras partes.

El componente debe ser comprendido por los clientes para poder ser usado (re-usado), uno de los aspectos que contribuyen especialmente es el de la identificación de un conjunto de funcionalidades claras e interrelacionadas que son provistas por el componente. Componentes que en su generalidad pretenden proveer diferentes funcionalidades con un espectro amplio, van a ser más difíciles de comprender y por tanto de re-usar.

Finalmente, componentes que no sean triviales, por lo general van a requerir de ciertos elementos de configuración que les permita adecuarse mejor al contexto de la aplicación y a los requerimientos particulares de la misma.

En todo caso, para poder abordar el desarrollo de un sistema basado en componentes y que estos sean realmente re-usables, se necesitan *buenos componentes*. Esta calificación, que aparece como bastante subjetiva, puede traducirse en algunas características específicas de los componentes. Por ejemplo se precisan componentes de alta cohesión, es decir que estén construidos por un conjunto de clases que colaboren estrechamente para implementar una

funcionalidad concreta; también se necesita que estos componentes tengan acoplamiento limitado, en otras palabras, mientras sea posible, que no requieran o presuman la presencia de algún otro componente específico para poder llevar a cabo su funcionalidad, en los casos en que esto no se pueda evitar se dice que el componente *depende* de otro.

Más allá de esas características siempre deseables, es necesario construir componentes que [Collins01a]:

- Implementen la funcionalidad correcta
- Tengan interfaces bien definidas
- Presenten una granularidad conveniente y
- Establezcan adecuadamente sus dependencias

En las siguientes sub-secciones se analizan estas cualidades establecidas como necesarias, definiendo con mayor precisión los alcances de las mismas y se presentan algunos principios de diseño que pueden ser usados como guía para lograr el cumplimiento de las mismas.

Los principios de diseño que se presentan son tomados en buena parte de los principios de diseño orientado a objetos, recopilados por Robert Martin en una serie de artículos de The C++ Report. Los restantes son principios de ingeniería de software presentados en la 21 Conferencia Internacional de Ingeniería de Software en una de las ponencias [Kang99]. Cada principio está referido a la cualidad de los componentes que se quiere alcanzar.

4.2.1. Funcionalidad correcta.

En principio un componente va a suministrar la funcionalidad correcta si brinda un servicio que cumple con los requerimientos especificados, en el marco de un sistema determinado.

La funcionalidad correcta de un componente, se establece en un contexto preciso dentro de un dominio y para una arquitectura de software determinada. Para lograr el re-uso de un componente, los aspectos en común de diversos sistemas deben ser descubiertos y representados de una manera que puedan ser usados al desarrollar un sistema similar, esto es el Principio de Orientación al Dominio (Domain Orientation Principle, DOP) [Kang99].

Esto está directamente relacionado con las disciplinas de Ingeniería de Dominio y Análisis de Dominio, a partir de las cuales se puede definir esa funcionalidad a implementar y a las cuales responde además la arquitectura del sistema para el cual se desarrolla el componente. La funcionalidad de los componentes debería ser determinada por los requerimientos del dominio de aplicación más que por las necesidades de una aplicación en particular. Esto apunta a aumentar las posibilidades de re-uso de los componentes.

Asimismo, las variantes que puedan ocurrir en la implementación de la funcionalidad dan origen a las líneas de producto, en tanto se mantenga una base común de requerimientos y componentes compartidos.

Otro principio a considerar es el de Separación de Intereses (Separation of Concerns Principle, SCP) [Kang99]: cada componente debe ser diseñado de manera que realice una única función claramente especificada (esto contribuye a la comprensión del componente y su potencial de re-uso) . Asimismo se desprende que cada componente funcional debe ser diseñado en forma independiente del mecanismo de interfaz por el cual se comunica con otros componentes. Se apunta al diseño lógico de los componentes con independencia de la plataforma.

4.2.2. Interfaces bien definidas.

Las interfaces deben ser bien definidas y fáciles de usar.

El Principio de Segregación de Interfaces (ISP, Interface Segregation Principle) [Martin96b], da una guía al respecto: *"Los clientes no deben ser forzados a depender de interfaces que no usan"*.

En otras palabras, para diferentes tipos de clientes se deberían proveer diferentes interfaces, aunque internamente al componente, todas esas interfaces pueden implementarse con el mismo conjunto de clases interrelacionadas. Cuanto más específica es la interfaz, en cuanto a los métodos que provee y su signatura, más fácil de comprender será su funcionalidad y por tanto se estimulará el re-uso.

Otro aspecto importante es que los componentes presenten interfaces como abstracciones limpias al usuario. Se deben usar conceptos, clases o tipos, que sean familiares al desarrollador de los componentes clientes y esconder detalles de implementación irrelevantes. En particular cuanto más básicos sean los tipos usados en la signatura de los métodos (Integer, string, etc.) más posibilidades de re-uso va a tener el componente.

El principio de Interfaz abstracta de Máquina Virtual (Abstract Virtual Machine Interface Principle, AVMIP) [Kang99], establece que *"La interfaz de un componente debe ser diseñada como una máquina virtual"*. Un componente debe proveer interfaces completas y no redundantes (mínimas), que además deben esconder los aspectos de implementación de tal manera que se puedan hacer diferentes implementaciones de la funcionalidad ofrecida.

Otro principio a considerar es el Principio de Sustitución de Liskov (LSP, Liskov Substitution Principle, por Bárbara Liskov, quien lo enunció) [Martin96a]. Este principio establece que *"Funciones que utilicen objetos de clases base deben poder usar objetos derivados de esas clases, sin enterarse"*.

Aplicado a los componentes significa que manteniendo la conformidad con la interfaz, cualquier implementación de la funcionalidad (componente) debería poder intercambiarse sin que el contexto se vuelva inconsistente. Este principio se refiere a la consistencia semántica de una interfaz en relación a todas sus implementaciones alternativas.

El principio LSP es también conocido como Principio de Diseño por Contrato, ya que los contratos (precondiciones, poscondiciones e invariantes) establecen los aspectos semánticos de las interfaces. Bertrand Meyer enunció sus reglas de precondiciones y poscondiciones de la siguiente manera: *"cuando se sobrescribe una rutina, solo podemos cambiar su precondición por otra más débil y su poscondición por otra más fuerte"* [Martin96a]. En ejemplos de violaciones del LSP, se puede apreciar como las precondiciones o poscondiciones no siguen esta regla, resultando la sustitución en sistemas inconsistentes. Se puede concluir que la consistencia de la sustitución está determinada por el entorno directo del componente (los clientes que lo usan), más que por aspectos intrínsecos del mismo. La observación de esta principio es fundamental en la implementación de puntos de variación o extensión. Un ejemplo de esto se puede ver en el Anexo 2.

4.2.3. Granularidad conveniente.

La granularidad de los componentes es todo un tema a resolver en el diseño de la arquitectura de un sistema. El término componente se ha aplicado a piezas de software que van desde sistemas enteros en el caso de integración de aplicaciones (EAI, Enterprise Application Integration) hasta

los componentes GUI más pequeños (botones, textbox, etc.) llamados *widgets (*enanos), pasando por los *Business Components* que representan conceptos del negocio [Sims98][Herzum98]. En definitiva el tema gira en torno a los módulos de funcionalidad y de distribución en que se va a descomponer un sistema.

Un elemento importante a tener en cuenta es que los componentes a lo largo del tiempo pueden tener variaciones, las que deben ser manejadas mediante algún mecanismo de control de versiones.

Es impensable o muy difícil que se pueda aplicar un re-uso real si no existe alguna forma de control de versiones. En cada liberación del software se adjudicará algún elemento que identifique de que versión se trata y en el mejor de los casos con que versiones de otros componentes puede interactuar.

El Principio de Equivalencia de Re-uso/Liberación (Reuse/Release Equivalence Principle, RREP) establece que: *"La granularidad de re-uso es la granularidad de liberación. Solo componentes que son liberados mediante un sistema de trazabilidad pueden ser efectivamente re-usados. Esa granularidad es el paquete"* [Martin96c]. En otras palabras la granularidad de un componente debe coincidir con el software contenido en la unidad de liberación, cada paquete que se libere por separado debe contener un componente y solo uno.

Otro aspecto a considerar es el enunciado en el Principio de Clausura Común (Common Closure Principle, CCP) [Martin96c], que dice que: *"Las clases incluidas en un paquete se deben ver afectadas por igual frente a un cambio. Un cambio que afecta al paquete, afecta a todas las clases de ese paquete"*. En otras palabras este principio dice que las clases o partes que es probable que se vean afectadas por los mismos cambios, deberían ser parte del mismo componente. En base a este principio, cuando hay un cambio, todo lo que éste afecta esta contenido en un componente y solamente se tendrá que cambiar y reinstalar ese componente.

Por otra parte es muy difícil que se pueda re-usar una clase en forma aislada; para lograr cumplir cierta funcionalidad, se va a requerir la *colaboración* de un conjunto de objetos y cada vez que se use esa funcionalidad se estarán usando todos esos objetos. El Principio de Re-uso Común (Common Reuse Principle, CRP) dice que: *"Las clases de un paquete se re-usan en conjunto. Si se re-usa una clase de un paquete se re-usan todas"* [Martin96c].

Dicho de otra manera, las clases que forman parte de una colaboración deben formar parte del mismo componente. Este principio tiene un corolario: las clases que no sean re-usadas en conjunto no deben estar en el mismo componente, ya que si esto no se cumple, podría ocurrir que un componente contuviera clases que no se precisan para una aplicación particular y si estas cambian (aunque las que sí se usan, no cambien) habría que pasar por el proceso de instalar una nueva versión del componente para mantenerlo actualizado, actividad que no aportaría ningún valor al sistema.

El hecho de colocar en el mismo componente todas las clases que forman parte de una colaboración, puede llevar a la formación de macro componentes (todo un sistema) ya que cada clase puede participar en varias colaboraciones, cada vez con un subconjunto diferente de clases. Esto va en contra del principio de re-uso común (CRP), pero se puede solucionar de dos maneras diferentes.

Una solución sería la de extraer el comportamiento común, es decir el conjunto de clases o colaboraciones que participan repetidamente y con ellas crear un componente de nivel más general o abstracto del cual dependen los otros conjuntos de colaboraciones, que a su vez formaría componentes independientes entre sí [Collins01a].

En la Figura 9, las clases C1, C2, C3, C4, C5 y C6 están contenidas en un componente. C1, C2, C3 y C4 colaboran para implementar una funcionalidad cualquiera (señalada con línea entera), mientras las clases C3, C4, C5 y C6 colaboran en una segunda funcionalidad (señalada con línea punteada). Para respetar el CRP, se debería cambiar la granularidad del componente, dividiéndolo en tres componentes (A, B y C). Las clases C3 y C4, que forman parte de las dos colaboraciones, se separan en un componente "B", que será un componente más general. Las restantes clases se dividen en componentes "A" y "C" que no se relacionan entre sí. De este modo se logra cumplir con CRP.

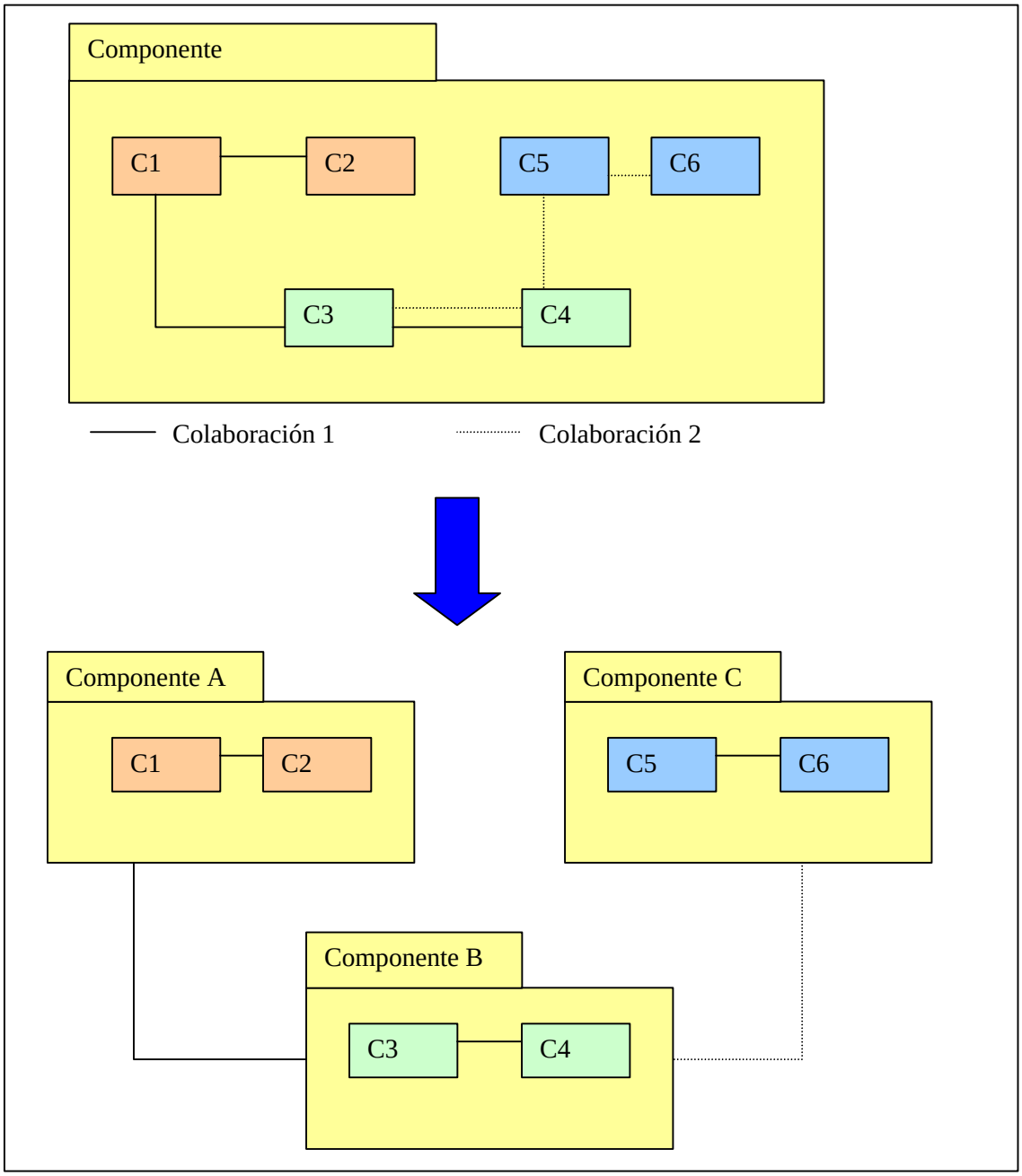

Figura 9. Cambio en la granularidad de componentes para respetar CRP

La otra solución pasa por el diseño basado en colaboraciones, que describe una aplicación como una composición de colaboraciones independientes. Cada objeto de la aplicación representa una colección de roles que describen acciones sobre datos comunes. Cada colaboración, por su

parte, es una colección de roles que encapsula relaciones a través de los objetos correspondientes. Estas colaboraciones son denominadas *refinamientos* [Batory99] [Smaragdakis99].

De modo que un refinamiento puede agregar a una clase nuevos atributos o miembros, nuevos métodos o sobrescribir métodos existentes. En lenguajes orientados a objetos esto se expresa fácilmente derivando subclases, que encapsulan las modificaciones del refinamiento, a partir de las clases originales, que podrían ser abstractas [Batory98] [Smaragdakis99]. Los diversos roles de las clases que forman parte de una colaboración están representados por esas subclases, que en conjunto forman el componente o refinamiento de la aplicación.

Como se ve en la Figura 10, la aplicación se puede ver como un conjunto de clases o como un conjunto de colaboraciones o refinamientos. Con esta idea, los componentes expresan colaboraciones y el sistema queda formado por capas de refinamientos, de hecho Don Batory[14] señala *componente, refinamiento y capa* como sinónimos.

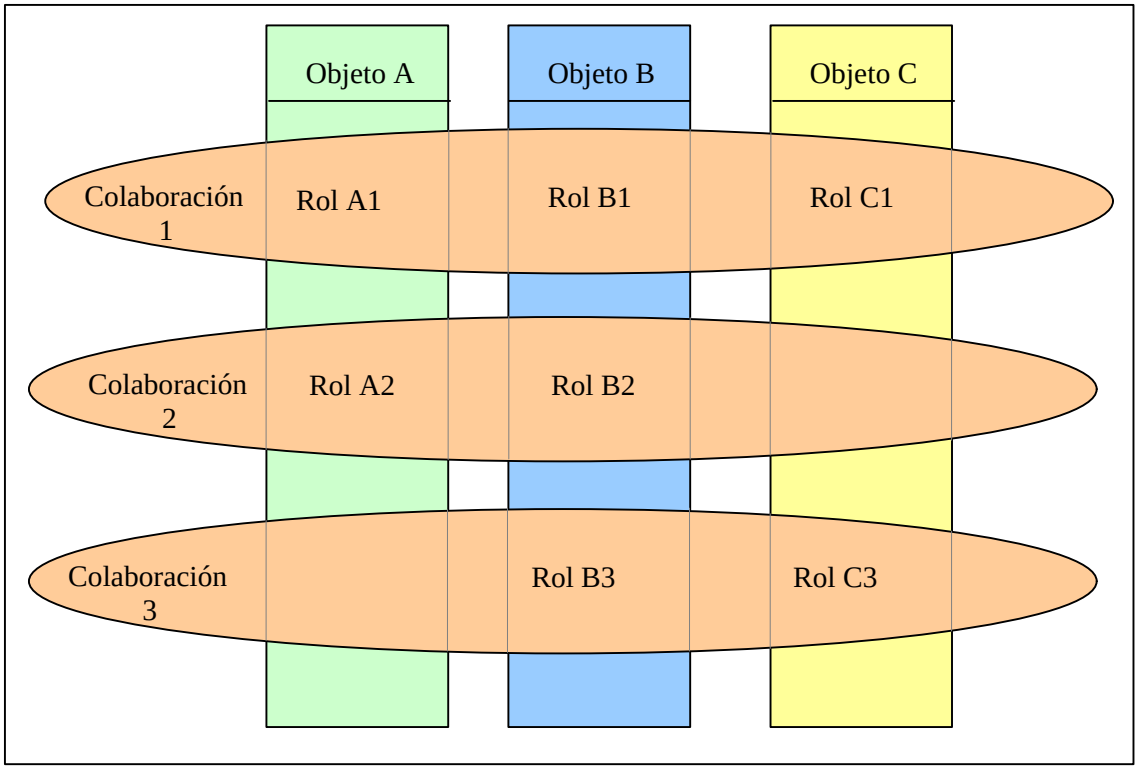

Figura 10. Descomposición de clases en roles de colaboraciones

Este enfoque tiene además un efecto importante en la composición de sistemas, ya que escribiendo distintas opciones de clases derivadas, se pueden obtener diferentes refinamientos y un mecanismo de composición, que pasa por elegir cual de los conjuntos de subclases instanciar en un caso particular. Esto es en particular beneficioso para el manejo de líneas de productos.[15]

[14] El Dr. Don Batory es profesor en el departamento de Ciencia de la Computación de la Universidad de Texas y consultor industrial en arquitecturas de líneas de productos y generadores de sistemas basados en componentes.

[15] En [Smaragdakis99] hay un trabajo muy interesante acerca de la implementación de estos conceptos a través de una construcción de C++: *Mixin Layers*, usando templates del lenguaje, herencia múltiple, anidación de clases y estandarización de nombres.

4.2.4. Dependencias entre componentes.

Los componentes, como fueron concebidos hasta ahora, deben implementar funcionalidades concretas. Para la implementación de la una funcionalidad específica es muy probable que los componentes "importen" una interfaz, es decir que requieran la realización de ciertos servicios especificados en la misma, por parte de un tercero. La implementación del servicio previsto en esa interfaz es a su vez realizada por otro componente y por esa razón se dice que el componente original tiene una dependencia con el segundo. Si bien el bajo acoplamiento entre componentes es una de las cualidades más buscadas, las dependencias son inevitables y por tanto se deben buscar reglas de diseño que impidan que estas afecten desfavorablemente al sistema.

El Principio de Dependencias Acíclicas (Acyclic Dependency Principle, ADP), establece que *"La estructura de dependencias entre paquetes debe formar un Grafo Orientado Acíclico, o sea que no debe haber ciclos en la estructura de dependencias"* [Martin96c]. Si se tienen dos componentes que dependen mutuamente uno del otro, como muestra la Figura 11, estas dependencias forman un ciclo: el primer componente requiere que esté presente el segundo y viceversa. Cabría la pregunta: ¿porqué dos componentes? Si siempre que se instale uno va a hacer falta instalar el otro, es mejor formar un solo componente con los dos, lo que al menos reportará ventajas al simplificar el despliegue del sistema.

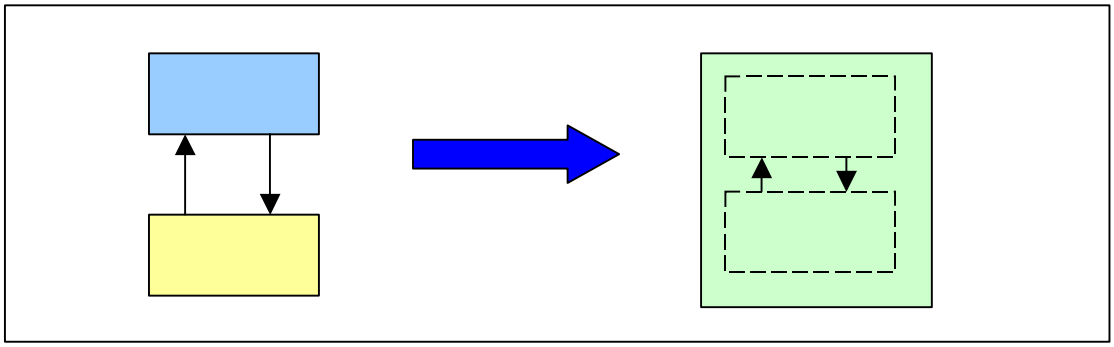

Figura 11. Eliminación de dependencias cíclicas para respetar ADP

La dependencia entre componentes tiene su punto crucial cuando alguno de los componentes es modificado. Es muy probable que aquellos componentes que dependan de éste tengan a su vez que ser modificados, o al menos habrá que verificar que esto no será necesario. El Principio de Dependencias Estables (Stable Dependencies Principle, SDP), da una guía con relación a este tema, estableciendo que: *"Las dependencias entre paquetes de un diseño, deben ir en la dirección de la estabilidad del paquete. Un paquete solamente debe depender de paquetes más estables que él"* [Martin97]. Si no se observa este principio y componentes que no tienen cambios frecuentes dependen de otros más inestables, cuando estos cambien se deberá revisar y eventualmente modificar a los componentes dependientes ajustándose a los cambios, aunque nada de su funcionalidad se haya modificado, se daría una situación absurda en la cual el componente más estable igualmente debe ser retocado constantemente.

Este principio sugiere la necesidad de definir cuales son los elementos estables en una aplicación. En general estos van a ser los elementos estables del dominio, conceptos o abstracciones básicas, denominadas *temas constantes del negocio (Enduring Business Themes, EBT)* por Mohamed Fayad[16], que también establece un nivel intermedio de estabilidad, los *objetos del negocio (Business Objects, BO)*, los cuales conceptualmente no tienen variaciones en cuanto a su significado, pero que sí admiten variaciones en su implementación y finalmente

[16] Profesor de J.D. Edwards en la Universidad de Nebraska y columnista habitual de Communications of the ACM

los *objetos industriales (Industrial Objects, IO)*, que pueden ser diferentes en cada implementación, estar presentes o no y variar de acuerdo a los cambios tecnológicos [Fayad01] [Fayad02a] [Fayad02b].

Estos temas remiten una vez más al área de la Ingeniería de Dominio y al Análisis de Dominio ya que los EBT y los BO, en general, no aparecen en el planteo del problema para una aplicación concreta, sino que están implícitos en el dominio de la misma y por esa razón son más difíciles de identificar. Los objetos industriales en cambio son los que aparecen más a menudo en la especificación del problema, son en general la cara visible del problema y por consiguiente más fáciles de descubrir. Una vez hecho esto y rastreando las razones de la existencia de esos IO se puede encontrar los BO y los EBT de la aplicación y del dominio. Tanto los EBT como los BO y los IO pueden ser implementados por componentes cuyas dependencias irán en el sentido de la estabilidad, acordes al principio de SDP.

4.2.5. Características del sistema y principios de diseño.

En las secciones precedentes se enunciaron una serie de principios de diseño, asociados a diferentes aspectos o cualidades de los componentes a desarrollar o de la línea de productos a construir. Estos principios también pueden ser usados para evaluar propuestas concretas de arquitecturas o diseños de sistemas basados en componentes.

En la Tabla 2, se resume la relación de los principios enunciados con respecto a cada aspecto o cualidad del sistema basado en componentes.

Cualidad	Principio	Sigla Usada	Recopilado por
Funcionalidad Correcta	Orientación al Dominio	DOP	[Kang99]
	Separación de Intereses	SCP	[Kang99]
Interfaces bien Definidas	Segregación de Interfaces	ISP	[Martin96b]
	Interfaz Abstracta de Máquina Virtual	AVMIP	[Kang99]
	Sustitución o Diseño por Contrato	LSP	[Martin96a]
Granularidad Adecuada	Equivalencia de liberación y re-uso	RREP	[Martin96c]
	Clausura Común	CCP	[Martin96c]
	Re-uso Común	CRP	[Martin96c]
Manejo de Dependencias	Dependencias Acíclicas	ADP	[Martin96c]
	Dependencias Estables	SDP	[Martin97]

Tabla 2. Principios de diseño asociados a las cualidades de los componentes

4.3. Arquitecturas de sistemas de componentes.

En las sub-secciones precedentes se estudió la aplicación de algunos principios generales de diseño para construir "*buenos componentes*". Para la construcción de un sistema de aplicación o formar parte de una línea de productos esos componentes deben interactuar entre sí de acuerdo a un diseño expresado por la arquitectura del sistema o la arquitectura de la línea de productos.

En este trabajo se ha tomado como definición de arquitectura de un sistema a la descripción de la organización estática de los componentes que lo forman (interconectados a través de interfaces) y la definición a nivel significativo de cómo interactúan estos componentes entre sí [Griss01].

Por supuesto que esa definición *a nivel significativo* depende de qué es lo que se quiere resaltar o cual va a ser el uso de la arquitectura. El concepto *arquitectura de software* está sobrecargado de significados, de acuerdo a diferentes aplicaciones del mismo: tendremos una arquitectura semántica, que describe las interrelaciones de los componentes en la construcción o modelo lógico del sistema, pero también tendremos una arquitectura de despliegue que describe en que lugar físico se instalarán esos componentes y con que otros nodos van a interactuar y una arquitectura de interoperabilidad, que describe el modelo de comunicación entre componentes (J2EE, .NET, etc.) y de manejo de su ciclo de vida. En definitiva las arquitecturas se deberían estudiar en conjunto con las metodologías y conceptos de diseño que las utilizan, a modo de ejemplo RUP [Rational02] define cinco vistas diferentes para la arquitectura de un sistema.

En dos ejemplos de arquitecturas concretas que se incluyen a continuación, se pueden ver diferentes motivaciones para la definición de la arquitectura: en la primera la arquitectura es usada para expresar la separación de responsabilidades en un sistema distribuido, en la segunda la arquitectura expresa la forma en que los componentes colaboran, siguiendo niveles de especificidad o generalidad (menor o mayor nivel de re-uso en el dominio), para llevar adelante los procesos del negocio.

En los dos casos se verá de que manera se observan los principios de diseño señalados en los puntos precedentes.

4.3.1. Arquitectura para Sistema de Componentes de Negocio, Sims y Herzum.

Esta ejemplo de arquitectura fue presentada por Oliver Sims y Peter Herzum en OOPSLA'98[17] [Herzum98] y se enmarca en una metodología de diseño basada en *componentes de negocio*, que expresan conceptos relevantes, entidades o procesos, de la aplicación o del dominio.

Un componente de negocio *"es la representación en un sistema de información, desde el análisis de requerimientos a la instalación y puesta en producción[18] de un concepto autónomo o un proceso autónomo del negocio. Consiste de todos los artefactos de software necesarios para expresar, implementar e instalar ese concepto autónomo del negocio como un elemento re-usable, autónomo también, en un sistema de información más grande"* [SIMS98].

Se trata de algo diferente al concepto de componentes que se ha definido en este trabajo, esta representación de conceptos de negocio tiene su origen en el análisis mismo de requerimientos y conserva su esencia a lo largo de todo el ciclo de desarrollo del software. Tiene manifestaciones físicas a lo largo de todo ese ciclo (a través de artefactos como: diagramas UML, modelo de entidad relación, código fuente, dll o exe, etc.).

El componente de negocio arranca como un concepto que se relaciona directamente con el dominio del negocio, se mantiene como una unidad coherente a lo largo del ciclo de vida de desarrollo, y emerge al final como un conjunto de componentes de software (componentes distribuidos) que pueden evolucionar en forma independiente en la medida que los requerimientos del negocio cambien o se desarrollen.

A modo de ejemplo, en una empresa se puede identificar la gestión de facturación como un concepto de negocio autónomo útil y relevante. Se podría considerar un componente de proceso de negocio, "Gestión de Facturación", como un componente de negocio útil para el sistema de información. Sin embargo ese componente de proceso aislado es insuficiente, se necesitan otras "cosas" para construir una aplicación efectiva, como las propias facturas, vendedores, etc.

[17] Object Oriented Programming, Systems, Languages and Applications, conferencia patrocinada por ACM.
[18] En el sentido de software activo, corriendo, en uso.

El Modelo de Sistema de Componentes de Negocio (BCSM, Business Component System Model), cuya representación gráfica se muestra en la Figura 12, expresa la totalidad de un concepto de negocio que se está desarrollando, en términos de los componentes de negocio necesarios para dotarlo de autonomía. Es una vista de alto nivel del concepto autónomo del negocio, haciendo abstracción de la arquitectura interna de cada componente de negocio y del detalle de las dependencias exactas entre estos.

La Figura 12 muestra una representación gráfica del sistema de componentes de negocio para un sistema de gestión de facturación sencillo. Cada cuadro representa un componente de negocio separado. En la parte superior aparece el componente de proceso "Gestión de Facturación" y debajo, todos y cada uno de los componentes de negocio que le son necesarios, desarrollados en forma independiente.

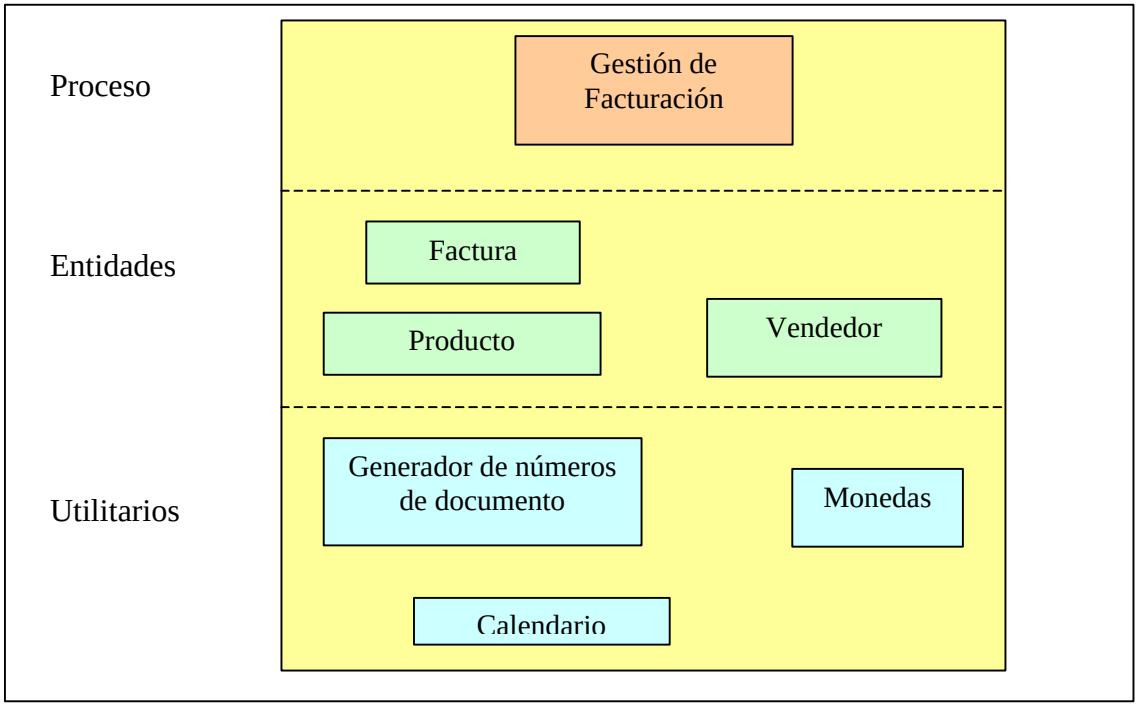

Figura 12. Sistema de Componentes de Negocio para el concepto Gestión de Facturación.

Se puede apreciar:

- el conjunto de componentes de negocio desarrollados independientemente y que colaboran para brindar la funcionalidad requerida
- que los componentes de negocio se ubican en diferentes niveles, indicados en la Figura 12 como *proceso*, *entidad* y *utilitarios*. Las dependencias entre componentes de negocio forman un grafo orientado sin ciclos, cada componente puede tener dependencias con componentes del mismo nivel o de niveles inferiores. Los componentes de negocio van a ser representados en tiempo de ejecución por componentes distribuidos, cuyas dependencias, del mismo modo representarán un grafo orientado sin ciclos, similar a lo que establecía el principio de dependencias acíclicas (ADP) visto en la sub-sección 4.2.4.
- los componentes se agrupan en categorías que reflejan los conceptos más relevantes desde el punto de vista del negocio. No se incluyen otras categorías (aunque necesariamente van a estar presentes), tales como gestión de la base de datos, calculadora, etc.

Los componentes de negocio de un mismo nivel tienden a tener algunas características en común, por ejemplo cuanto más bajo es el nivel, más relevantes son los aspectos de persistencia y menos relevantes los aspectos de proceso o de interfaz con el usuario. Y, por el contrario,

cuanto más alto es el nivel, menos relevantes los aspectos de persistencia y más relevantes los de procesamiento e interfaz con el usuario.

Sims y Herzum definen una arquitectura interna para los componentes de negocio que se basa en la separación de intereses, relevante en los sistemas distribuidos, de ese modo los componentes de negocio son diseñados desde el comienzo para implementarse como sistemas distribuidos. Esto se basa en dos aspectos fundamentales:

- un modelo estructural (arquitectura) que apunta directamente a los problemas de diseño de los sistemas distribuidos, descomponiendo los componentes de negocio en "componentes distribuidos" que se ubican en diferentes capas del modelo.
- una tecnología de componentes distribuidos que forma la base del desarrollo e instalación autónomos, la cual escapa al alcance de este trabajo, pero puede ser consultada en el resumen del Anexo 3, o en [Herzum98], [Sims98] y [OMG96].

4.3.1.1. El modelo estructural de Sims y Herzum.

En el modelo de Sims y Herzum, los componentes de negocio son soportados por una arquitectura de cuatro capas, que alberga los diferentes componentes distribuidos que los implementan.

Estas capas son: *capa usuario*, *capa de área de trabajo*, *capa empresa y capa de recursos*. En cada una se corresponde a cada área lógica de responsabilidad, un área de ocupación (en sistemas distribuidos) separada.

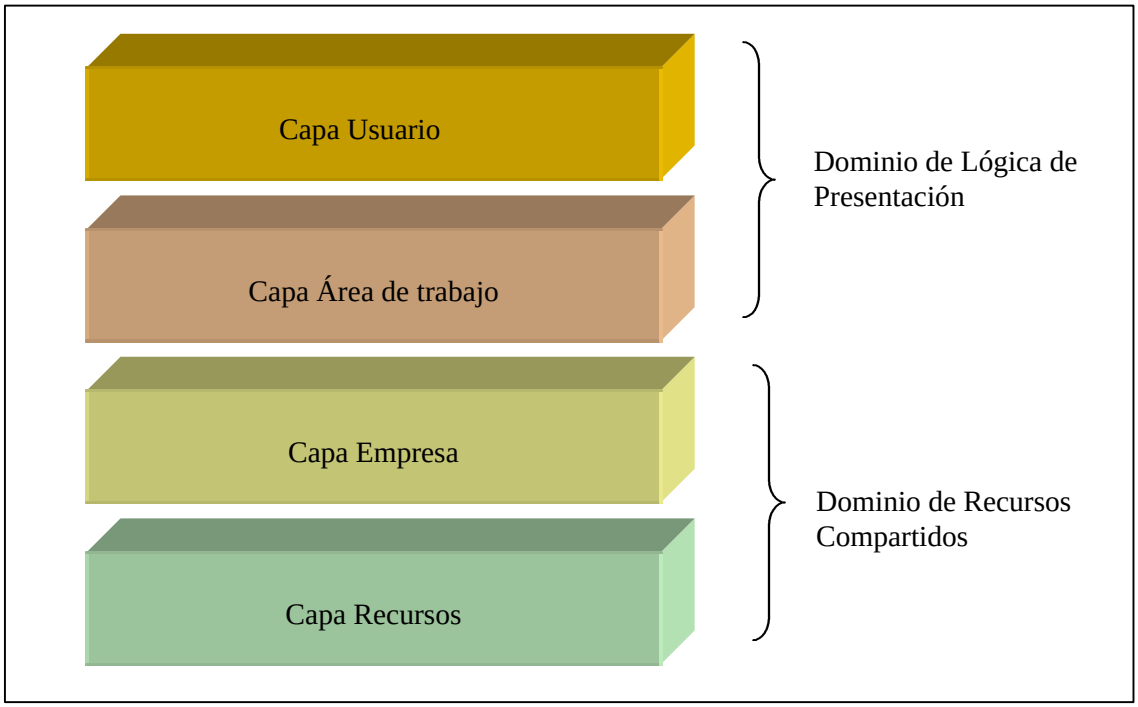

Figura 13. Arquitectura en cuatro capas de Sims y Herzum

La capa usuario es responsable de la presentación en pantalla de los componentes de negocio. No todos los componentes de negocio deben ser presentados en pantalla, por ejemplo la interfaz de usuario del componente de negocio "Factura" puede estar provista por la interfaz de usuario del componente de proceso "Gestión de Facturación".

Desde el punto de vista físico, esta capa se ubica lo más cerca posible del usuario final, en los PC o equipos "clientes" del sistema distribuido.

La capa de área de trabajo es responsable de dar soporte a la capa usuario, implementando la lógica local del negocio (para el uso concreto de un usuario) y conociendo con quien (que otros componentes distribuidos de otras capas) debe comunicarse para obtener o suministrar datos. Actúa como un buffer para un usuario exclusivo (es el área de trabajo de ese usuario) y puede tener estado y aspectos de persistencia no compartida con otros usuarios..

Esta capa, físicamente, puede estar ubicada en el equipo cliente o en un servidor departamental o central.

Las capas de usuario y de área de trabajo, en conjunto forman el *dominio de lógica de presentación*. Este es responsable de soportar a cada usuario a través de una sesión de trabajo, representa la visión o el modelo que tiene el usuario del componente de negocio. En este dominio se van a encontrar la mayoría de los componentes de software de mercado, ActiveX y JavaBeans y es el dominio de menor relevancia o de menor impacto en lo que hace a los sistemas de gran escala y misión crítica en la empresa: los cambios producidos en este dominio van a tener un efecto exclusivamente local.

La capa empresa es responsable por los aspectos más importantes de procesamiento de un componente de negocio. En esta capa se implementan la mayor parte de las reglas de negocio, la validación y la interacción con otros componentes del negocio, la gestión y la integridad del negocio. Esta capa es la más importante para los sistemas de gran escala y misión crítica y como ya fue dicho es la que tiene menor disponibilidad (o nula disponibilidad) de componentes de software de mercado.

Los componentes de software de esta capa son a menudo confundidos con el propio componente de negocio ya que encierran sus reglas, condiciones y características de procesamiento, sin embargo los autores insisten en que el componente de negocio incluye otros aspectos del concepto que se representa, como son la presentación o la persistencia y que son implementados por componentes distribuidos en las otras capas.

La capa de recursos es responsable del acceso a recursos compartidos del sistema. Por ejemplo, uno de los recursos más comunes es la información almacenada en bases de datos compartidas y por tanto esta capa es responsable de los aspectos de persistencia de los componentes de negocio. El acceso vía SQL a una base de datos particular es lanzado a partir de código de componentes de software de esta capa. Esta capa no tiene sentido sin la capa empresa a la cual presta servicios de acuerdo a sus requerimientos.

Las capas empresa y de recursos forman en conjunto el *dominio de recursos compartidos*, que es responsable por la integridad de esos recursos (en el marco de las reglas del negocio) y de proveer servicios, basados en esos recursos, a una multitud de clientes autorizados. En una implementación con tecnología Java, este dominio estaría típicamente poblado de EJB (EnterpriseJavaBeans).

La arquitectura para sistemas de componentes de negocio soluciona algunos aspectos recurrentes en los sistemas distribuidos[19], entre otros:

- Separa la base de datos (capa de recursos) de la lógica del negocio (capa empresa). Esto permite que los esquemas de base de datos y las tecnologías empleadas puedan cambiar sin que esto afecte necesariamente la lógica del negocio, ubicada en la capa empresa.

[19] Aunque corresponde aclarar que otros modelos, como el clásico de tres capas, también lo hacen.

- Resuelve al problema de ubicación de los objetos de negocio en los sistemas distribuidos, respondiendo a preguntas como: ¿Dónde se instancia el objeto que representa a un Comprador?. La estructura de componentes de negocio dice que van a haber dos objetos distribuidos que representan ese Comprador uno en el dominio de lógica de presentación y otro en el dominio de recursos compartidos. Estos dos objetos pueden o no usar el mismo código.

4.3.1.2. Evaluación de la arquitectura de Sims y Herzum en base a principios de diseño.

En la sub-sección 4.2 se señalaron cuatro aspectos de evaluación para determinar lo que en lenguaje llano se refirió como "buenos componentes" o un "buen sistema basado en componentes".

- Con relación a la *funcionalidad correcta* detallada en 4.2.1, se puede decir que la arquitectura de Sims y Herzum resulta de la aplicación del principio de Orientación al Dominio ya que desde las etapas más tempranas se trabaja con los conceptos del negocio. Asimismo se observa el principio de Separación de Intereses, en particular la independencia que se logra con relación a las tecnologías de interoperabilidad.
- Con respecto a contar con *interfaces bien definidas,* como fue tratado en la sub-sección 4.2.2, teniendo en cuenta que los componentes distribuidos se originan por la partición en capas de un concepto que es el componente de negocio, es presumible que la definición de interfaces entre las distintas capas que implementan ese componente de negocio sea adecuada. Con relación a la vinculación entre componentes de negocio (que implica una interfaz entre componentes distribuidos), la metodología no lo asegura. Acerca de los principios ISP y LSP tampoco se puede sacar ninguna conclusión, en cambio la observación del principio de Interfaz Abstracta de Máquina Virtual se desprende, por ejemplo, de la especificación de responsabilidades de los componentes del dominio de recursos compartidos.
- Acerca de la *granularidad conveniente* referida en 4.2.3, el método presenta dos aspectos: en la etapa de análisis y diseño una granularidad adecuada a las necesidades de los especialistas del dominio, dónde se puede componer una solución general basada en conceptos del negocio (el modelo de sistema de componentes de negocio). En cuanto a los componentes distribuidos, parece claro que hay un buen compromiso entre los principios de RREP, CRP y CCP. Cosas que pueden cambiar juntas, como algún elemento en el dominio de lógica de presentación, van a estar separadas en la capa de usuario y en la capa de área de trabajo, pero esto se debe a la observación de un principio más fuerte (RREP, equivalencia entre re-uso y liberación) para que los componentes se puedan instanciar en equipos separados.
- Los principios de *dependencias entre componentes* señalados en 4.2.4 son respetados por la regla de diseño del sistema de componentes de negocio: entre los componentes distribuidos que forman los diferentes componentes de negocio no hay dependencias cíclicas y en general las dependencias van hacia los componentes más generales del dominio o independientes del dominio de aplicación, que suelen ser los más estables (principios ADP y SDP). Las dependencias que se forman entre componentes distribuidos de un mismo componente de negocio, están orientadas de acuerdo a las capas de la arquitectura descrita: no hay ciclos y las dependencias en general son hacia partes más estables del concepto (por ejemplo puede cambiar la forma de presentar los datos de un Comprador, pero las reglas del negocio, validación y gestión de la persistencia del mismo van a ser mucho más estables e incluso usadas sin variaciones para diferentes formas de presentación de los datos).

4.3.2. Arquitectura de Collins-Cope y Matthews.

El ejemplo de arquitectura que se detalla a continuación fue presentado por Hubert Matthews y Mark Collins-Cope en ICSE2000[20] [Collins00]. En conjunto con otros artículos de los mismos autores [Collins01a] y [Collins01b], permite establecer criterios prácticos para el desarrollo de aplicaciones basadas en componentes en el área financiera y contable.

En este caso también se usa un estilo de arquitectura en capas; en particular se trata de cinco capas y está motivada en una mejor comunicación de la estructura general del sistema. La separación en capas clarifica los conceptos del sistema y estimula la separación de intereses. Por ejemplo, separa cuestiones técnicas de cuestiones propias del dominio; políticas de mecanismos y la decisión de si construir o comprar, quedan más expuestas o explícitas en esta arquitectura.

En particular las tecnologías de interoperabilidad y los diferentes modelos de componentes que proponen cada una, son dejados fuera de esta arquitectura, que resalta los aspectos lógicos o semánticos de la composición y el rol que cumple cada componente en el sistema.

Las capas en este modelo son contenedores lógicos de componentes, no componentes en sí mismos y por lo tanto el modelo no impide dependencias entre componentes de niveles no adyacentes. Se trata de un estilo de arquitectura en capas no estricto.

La estratificación de los componentes no está planteada como en el ejemplo anterior en cuestiones que tengan que ver con las responsabilidades lógicas en sistemas distribuidos, sino con la especificidad o generalidad de los componentes. Los componentes más específicos y por tanto menos re-usables son ubicados en las capas más altas de esta arquitectura, mientras que los componentes más re-usables, es decir más generales se ubican en las capas más bajas.

Si se tiene en cuenta que los componentes generales, que no van a ser exclusivos de una aplicación particular, sino propios del domino, tienen en general menor probabilidad de cambio que los componentes específicos, se verá que esta forma de estratificación lleva a sistemas más estables, ya que las dependencias irán “hacia abajo”[21] en el sentido de la estabilidad (SDP, en la sub-sección 4.2.4).

Las dependencias entre componentes pueden representar relaciones de asociación (referencia o uso de esos componentes) o de implementación de interfaces abstractas (extensibilidad distribuida o dinámica).

El eje vertical de la arquitectura marca la especificidad o generalidad de los componentes. Esto equivale a una graduación de la factibilidad de re-uso: los componentes más generales tenfrán un potencial de re-uso mayor y viceversa.

Es fácil observar que, al mismo tiempo, los componentes más generales van a poder ser usados (re-usados), no solo más veces sino en un número mayor de dominios de aplicación diferentes o en más aplicaciones distintas dentro de un dominio, por lo que el eje horizontal expresa la aplicabilidad de los componentes en relación con la cantidad de dominios en que pueden ser usados.

[20] 22nd. International Conference on Software Engineering, Limerick, Ireland, June 2000.

[21] La convención más comúnmente usada en presentación de arquitecturas en capas, ubica los componentes más específicos arriba y los más generales abajo. Esta convención es opuesta a la que generalmente se observa en diagramas de clases de UML, que presentan las clases derivadas (más específicas) “debajo” de las superclases (más generales), lo que puede llevar a alguna confusión si no se está atento al contexto [Szyperski97] [Collins01b].

4.3.2.1. Descripción de las cinco capas.

La arquitectura de Collins-Cope y Matthews se representa por un modelo de cinco capas, en las cuales, según muestra la Figura 14 se ubican diferentes tipos de componentes.

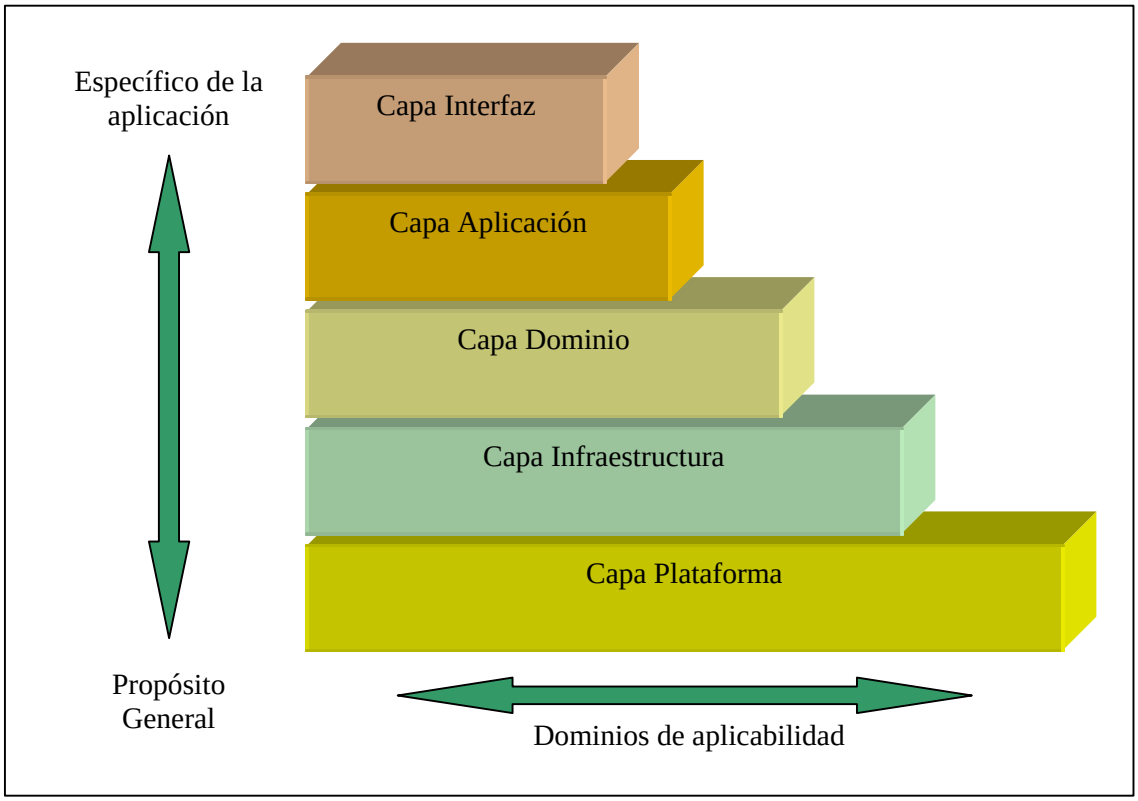

Figura 14. Arquitectura en cinco Capas de Collins-Cope y Matthews.

- En el nivel más alto la *capa interfaz*, que es la más específica en el sistema y también la que tiene menor rango de aplicabilidad a otras aplicaciones, aún del mismo dominio (a no ser otras versiones del mismo sistema). Esta capa es la responsable de manejar la comunicación entre el "mundo alrededor" y las capas de nivel más bajo. Esta comunicación con el mundo exterior puede ser con personas (a través de una interfaz gráfica) o con otros sistemas, a través de una interfaz electrónica (importación de archivos ASCII, documentos XML, etc.). En esta capa no se encuentran los componentes GUI (ActiveX, JavaBeans, etc.) de mercado (que son mucho más generales, no específicos de la aplicación) sino funcionalidades de comunicación con personas o sistemas, que probablemente dependan de (estén formados por) esos componentes o hayan sido derivados de estos.

- En un segundo escalón la *capa aplicación*, dónde se van a ubicar los objetos y componentes que encapsulan los procesos principales del negocio y las reglas de negocio específicas de la aplicación en particular. En esta capa se encuentran también muchos componentes que especializan o extienden interfaces dejadas abiertas en la capa de dominio (subsiguiente). Los objetos de esta capa no van a ser persistentes. Esta capa separa las políticas (ubicada en capas inferiores) de los mecanismos (procesos, ubicados en esta capa) y contiene el código que permiten la vinculación con esos componentes de nivel inferior. Distintos componentes de la capa de interfaz pueden depender del mismo componente en esta capa, por ejemplo el proceso que maneja una transacción puede ser el mismo aunque provenga de una interfaz con usuario o de un mensaje XML, manejados cada uno con los componentes correspondientes de la capa superior.

- A continuación, la *capa dominio*, está compuesta por componentes que encapsulan las interfaces de las clases del negocio, específicas del dominio de aplicación. En general estos componentes van a ser usados desde muchos lugares del sistema, suministrando las políticas y reglas generales del negocio (por ejemplo el soporte para partida doble en contabilidad). Asimismo suelen representar los componentes compartidos en una línea de productos. Las clases principales contenidas en los componentes de esta capa, representan entidades del dominio con estados persistentes.
- La *capa de infraestructura técnica*, provee, en el siguiente nivel, servicios técnicos de propósito general, tales como infraestructura para persistencia (el código SQL para acceder a las bases de datos, etc.), infraestructura de programación general (manejo de listas u otras estructuras de datos, etc.). Los componentes de esta capa pueden ser re-usados horizontalmente en cualquier aplicación o dominio.

- Finalmente y compuesta por los componentes más re-usables de todos, la *capa plataforma*, está compuesta por piezas de software que son incluidas para dar soporte a toda la aplicación. Los sistemas operativos, la infraestructura de interoperabilidad (COM, CORBA, etc.), manejadores de bases de datos y bibliotecas de GUI usadas en el desarrollo, son todos ejemplos del tipo de elementos de esta capa y salvo excepciones extraordinarias se trata de software adquirido (comprado a terceras partes).

De los elementos tecnológicos presentes en la capa de plataforma dependen componentes de las capas de interfaz y de infraestructura, no así los componentes de las capas de aplicación y de dominio, que representan la lógica del sistema. Es otra aplicación del principio de separación de intereses (la lógica del sistema no depende directamente de aspectos técnicos, podríamos cambiar el manejador de la base de datos o el ambiente gráfico y la lógica del sistema no se vería afectada).

4.3.2.2. Un ejemplo de uso de la arquitectura de Collins-Cope y Matthews.

Se verá un ejemplo para ayudar a clarificar los conceptos de aplicación de esta arquitectura:

En el dibujo de la Figura 15, las líneas terminadas en flecha rellena representan asociaciones entre clases (dentro de un componente o entre componentes de la misma o de diferente capa). Las líneas terminadas en flecha vacía representan implementaciones de interfaces abstractas. En todos los casos establecen dependencias en el sentido de la generalidad de los componentes, es decir en el sentido de la estabilidad.

Este dibujo esquematiza la arquitectura de un sistema de gestión financiera y en particular se toma como ejemplo la parte relativa al procesamiento de operaciones de débito a una cuenta.

En la capa de interfaz se ven los componentes que se vinculan con el mundo exterior, ya sea con personas, en el dibujo representado por el componente que incluye la clase “GUIdebit” o con otros sistemas informáticos, representado en el dibujo por el componente con la clase “Transfer”.

El componente que incluye “GUIdebit” se encarga de la comunicación a través de una interfaz gráfica, con las personas para que estas realicen sus operaciones. En el desarrollo de esa clase se usaron componentes de alguna biblioteca de componentes GUI (“GUIdial”) que se ubica en la capa de plataforma. La dependencia con estos componentes se expresa como una derivación de clases o como un mecanismo de delegación o por el uso directo de los mismos, incluidos tal cual en un formulario o ventana.

A su vez ambos componentes de la capa de interfaz requieren del componente de procesamiento que incluye la clase “ControlDebito”. Este componente se ubica en la capa de aplicación ya que provee lógica específica de la aplicación de débitos, ya sea que provengan de operaciones manuales o a través de comunicación con otros sistemas, por transferencia de operaciones.

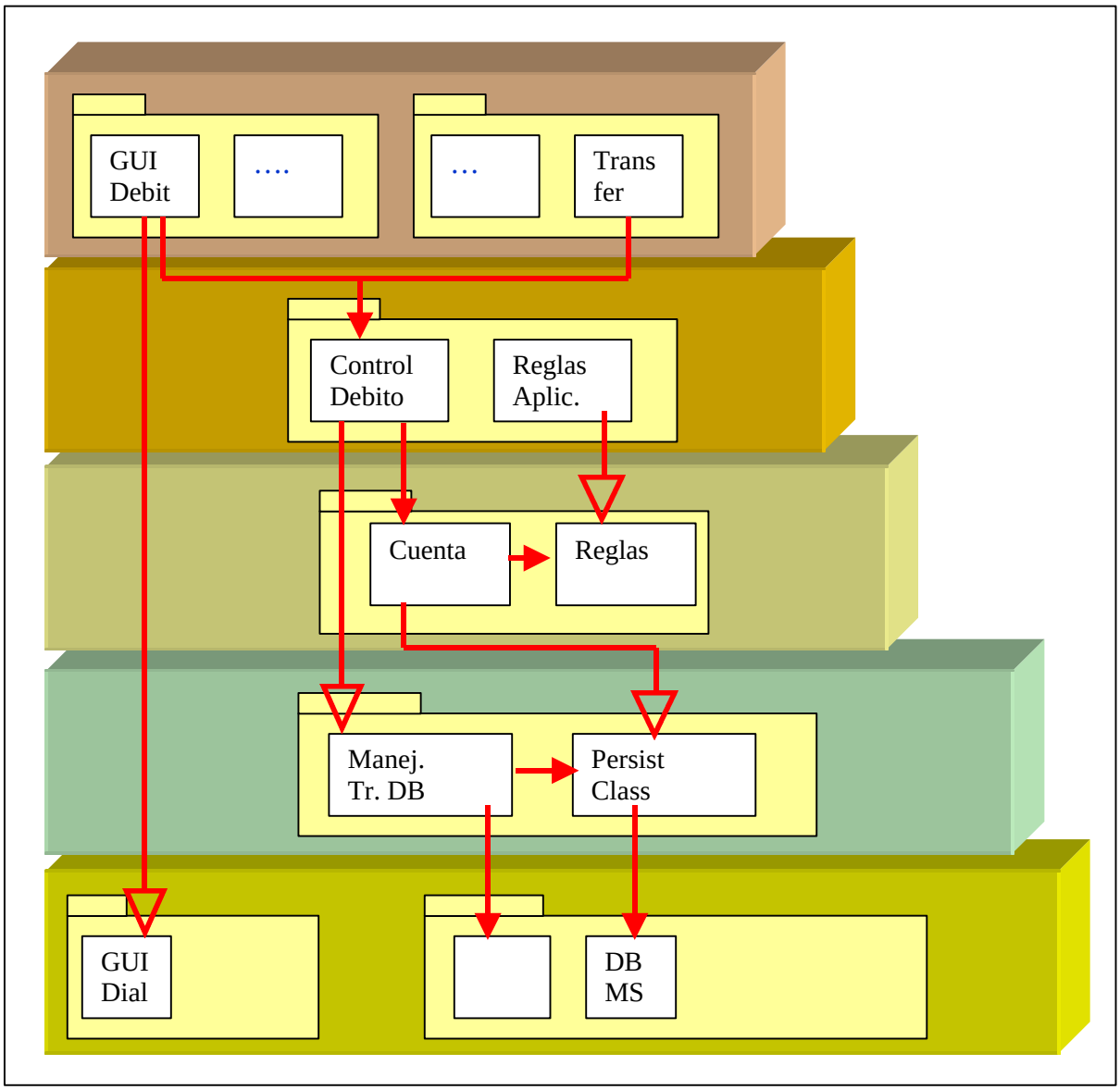

Figura 15. Ejemplo de aplicación de la arquitectura de Collins-Cope y Matthews.

Este componente dirige a la clase “Cuenta” de la capa dominio a través de una serie de invocaciones a métodos, la que en definitiva hace efectivas las operaciones de control de validez y registro.

La otra clase en el componente, “ReglasAplic” es derivada de la clase “Reglas” de la capa dominio y provee una implementación específica a las reglas de negocio para el tratamiento de las operaciones de débito, es por tanto de nivel aplicación.
La clase “ControlDebito” es a su vez derivada de una clase general que implementa el manejo de transacciones y que se ubica en la capa de infraestructura.

En la capa de dominio aparece un componente “Cuenta” que siguiendo las directivas de componentes del nivel inmediato superior, implementará los procedimientos requeridos para débitos (como en el ejemplo) y créditos que ocurran en la cuenta. Esta clase se apoya en una clase también propia del dominio para establecer las reglas de negocio, en particular debe ofrecer, además, una interfaz para implementar controles especializados a cada caso, como el que se hace a través de “ReglasAplic” del nivel superior.

En la capa de infraestructura aparecen componentes (“frameworks” probablemente) para el manejo de transacciones o para manejo de persistencia.

En la capa más baja aparecen los componentes más generales, sistema operativo, DBMS, componentes GUI, etc. de los cuales dependen todos los demás componentes de la aplicación.

Es de hacer notar que los componentes de esta capa, son básicamente componentes tecnológicos adquiridos a terceras partes. Obviamente los cambios tecnológicos se suceden con mayor rapidez que los cambios en las reglas y conceptos del dominio, sin embargo no hay dependencias directas de los componentes más comprometidos con la lógica de la aplicación(aunque sí hay una dependencia transitiva): las dependencias se establecen con componentes de la capa interfaz y de la capa de infraestructura, en otras palabras los cambios tecnológicos no deberían afectar a los componentes de la capa dominio ni de la capa aplicación, sino que se resuelven en las capas mencionadas.

4.3.2.3. Observaciones acerca del modelo.

Una serie de observaciones y criterios se extraen de la descripción del modelo de Collins-Cope y Matthews:

- Debería haber una relación 1:1 entre la estructura de componentes y la estructura de código ejecutable. Esta estructura puede ser usada como guía en el proceso del software, haciendo corresponder además sub-proyectos en paralelo para la implementación de cada componente.

- El nivel (la capa) en que se ubica un componente es el mismo que el nivel más alto de las clases que los forman.

- Reforzando lo anterior, los componentes no pueden cruzar capas.

- Las dependencias de los componentes deben ser o con componentes de niveles inferiores o con componentes de la misma capa.

- Las capas de aplicación y de dominio deben ser independientes de los componentes tecnológicos.

- No todos los componentes GUI residen en la capa interfaz, esta puede tener refinamientos específicos para la aplicación de componentes GUI de propósito general, los cuales residen en la capa plataforma.

- Hay quienes interpretan que una arquitectura en capas tiene la capacidad de sustituir una capa entera por otra y de ese modo modificar el comportamiento del sistema. En este caso las capas son una guía para determinar el contenido de los componentes y la sustitución podría ser para cada componente en particular, no necesariamente toda la capa. De hecho la primer interpretación trata a la capa entera como un componente (como en [Batory99] y [Smaragdakis01]).

4.3.2.4. Evaluación del modelo en base a principios de diseño.

Al igual que lo realizado para el modelo de Sims y Herzum (en 4.3.1.2), corresponde una evaluación de este modelo, en base a los criterios establecidos en la sub-sección 4.2.

- Con relación a implementar la funcionalidad correcta, no es mucho lo que se puede decir, ya que la metodología de análisis y de resolución de requerimientos no está especificada. Sin embargo algunos de los criterios son claramente respetados: el Principio de Orientación al Dominio, al identificar componentes específicos de dominio, independientes de aspectos tecnológicos e inclusive de aplicaciones en particular (para lo cual hay una capa específica). También es claro como se preserva el Principio de Separación de Intereses, a través de una estratificación que divide aspectos tecnológicos de aspectos de lógica y de interfaz con usuarios (personas o sistemas) y hasta componentes estándar de mercado (DBMS, GUI, etc.) de componentes desarrollados específicamente, evitando dependencias directas entre la lógica de la aplicación y los aspectos tecnológicos.

- Acerca de las interfaces, la propia estructura de la arquitectura favorece el principio de segregación de interfaces (ISP). En algún caso es explícita la definición de clases abstractas como interfaces a ser implementadas para extender la funcionalidad del sistema. Por supuesto que el buen uso de estos criterios depende del diseño final del sistema y no tanto de los aspectos estructurales.

- La granularidad de los componentes es uno de los aspectos que esta estructura orienta adecuadamente. La separación en capas de acuerdo a la especificidad de los componentes está muy de acuerdo con el principio de clausura común (CCP). Por otra parte el hecho de que cada capa no sea considerada como monolítica, contribuye a este principio (la capa no necesariamente se cambia en conjunto, sino solo aquellos componentes que sea necesario). Asimismo se mantiene el principio de equivalencia entre re-uso y liberación (RREP). La estratificación de acuerdo a especificidad, también resulta en una estratificación de acuerdo al potencial re-uso de los componentes y en ese sentido el principio de re-uso común (CRP) es claramente observado.

- Al igual que con la granularidad de los componentes, es notoria la observación de los principios para manejar una adecuada dependencia entre componentes: por definición (y por la semántica de la aplicación descompuesta de esta manera) no se forman ciclos en el grafo de dependencias (ADP) y al mismo tiempo estas están orientadas en el sentido de la estabilidad (SDP) hacia los niveles más bajos de la estructura, los más generales y re-usables.

4.3.3. Comparación de los modelos.

Sims y Herzum por un lado y Collins-Cope y Matthews por otro fundamentan sus arquitecturas en cuestiones diferentes.

La arquitectura de Sims y Herzum permite descomponer un concepto del negocio (el componente de negocio) en diferentes componentes físicos (ejecutables) de acuerdo a criterios de separación de responsabilidades en sistemas distribuidos. Por su parte Collins-Cope y Matthews diseñan su arquitectura para expresar los diferentes grados de re-uso potencial de los componentes o la generalidad / especificidad de cada componente en el contexto de un dominio de aplicación.

Sin embargo, si se analiza lo que se encuentra en cada capa, se puede apreciar una serie de similitudes en los resultados:

- Capa usuario vs. Capa interfaz. En ambos casos se incluyen los componentes que hacen posible la comunicación con el exterior (sea un usuario u otro sistema). Las reglas y lógica del negocio no aparecen en esta capa, sino solo el manejo que posibilita esa comunicación. Mientras que el modelo de Sims y Herzum incluye componentes GUI básicos, en el modelo de Collins-Cope y Matthews solo se incluyen derivaciones de estos (con dependencias a la capa plataforma donde residen). Conceptualmente el resultado es el mismo.

- Capa área de trabajo vs. Capa aplicación. En ambos casos se describen componentes de tipo proceso que llevan adelante la aplicación específica (originada en la capa precedente), usando o accediendo a los métodos y reglas contenidas en componentes de tipo entidad que encierran la lógica del negocio, ubicados en la capa siguiente. En la arquitectura de Sims y Herzum, quizás no quede tan claro que alguna extensión particular de estas entidades o reglas podría residir en la capa área de trabajo, pero es claro que esto forma parte de lo que se describe como "*implementa la lógica local del negocio"* [Herzum98].

- Capa empresa vs. Capa dominio. Más allá del nombre, que da idea de mayor generalidad en el segundo caso, sin duda esta capa contiene los componentes que describen las entidades del sistema, con sus lógicas y reglas de negocio asociadas y las cuales son "utilizadas" desde múltiples aplicaciones. En el dominio de los sistemas de gestión empresarial, las entidades y reglas propias de la contabilidad y la administración, si bien pueden ser seleccionadas por una empresa no son privativas de esta sino que responden en todos los casos a conceptos generales de dominio.

- Capa recursos vs. Capa infraestructura. En ambos casos está formada por los componentes o frameworks que hacen posible la comunicación con software específico de manejo de bases de datos, monitores transaccionales, o manejo de estructuras estándar (colas, listas, etc.). Se trata básicamente de la misma capa.

- Capa plataforma. Esta capa aparece solo en el modelo de Collins-Cope y Matthews. En el modelo de Sims y Herzum, si bien se hace mención a la (obvia) existencia de los componentes de este tipo, se aclara que no aparecen explícitos en su arquitectura. En el otro modelo, motivado en el potencial de re-uso y especificidad con relación a una aplicación, esta capa se puede explicitar sin dificultades (lo que es beneficioso, porque permite establecer las dependencias correspondientes).

La similitud ente los modelos no es sorprendente: en ambos casos se respetan en gran medida los principios de diseño reseñados, en particular los que tienen que ver con la granularidad y manejo de dependencias. En ambos casos hay una fuerte orientación al dominio y separación de intereses. En esas condiciones llegar a resultados similares era lo de esperar.

Es de resaltar, además, cómo el uso o la concordancia con los principios de diseño promueven arquitecturas dónde quedan expuestos claramente los aspectos comunes del dominio, representados por las capas "empresa" o "dominio", dónde se debiera verificar el mayor nivel de re-uso, dentro de una aplicación particular o a través de los diferentes miembros de una línea de productos.

Simultáneamente, se expone la posibilidad de variación cada vez mayor a medida que se acerca a la interfaz con el cliente o al manejo de la persistencia de la información. Esto es un aspecto a tener especialmente en cuenta en la definición de líneas de productos ya que lso puntos de extensión o variación quedan claramente identificados.

5. Conclusiones y trabajos futuros.

En este trabajo se ha realizado un estudio de los sistemas basados en componentes, sus propiedades y características fundamentales. El estudio se encaro desde el punto de vista de la construcción lógica de esos sistemas y de las diferentes estrategias para la elaboración de los mismos, en el marco de su aplicación para el dominio de los sistemas de gestión e información empresarial.

Las principales conclusiones se refieren a:

1. Si los sistemas basados en componentes son aptos para resolver los problemas de adaptabilidad que plantean la reingeniería de procesos de negocios y la mejora continua.
2. La importancia de los aspectos de diseño y de una metodología “top-down” para definir la estructura del sistema.
3. Las alternativas planteadas por los fabricantes de los sistemas ERP.
4. El impacto en el ciclo de vida del software y del desarrollo de áreas de ingeniería de dominio.
5. La realidad del mercado de componentes de negocio en el dominio de los sistemas de gestión e información de empresas.

Algunas elaboraciones, ideas y propuestas nuevas introducidas en el trabajo son:

6. Identificación de las razones que dificultan el establecimiento de un mercado de componentes y sus consecuencias.
7. Una propuesta para el desarrollo de sistemas de componentes, basada en la elaboración de líneas de productos, como forma de suplir la inexistencia de componentes de mercado.
8. Identificación de principios que pueden ser usados como guía para el diseño de los sistemas basados en componentes y su vinculación a las diferentes propiedades a verificar.

En las siguientes sub-secciones se resumen los principales aspectos de las conclusiones y aportes señalados y se proponen algunos temas vinculados sobre los que se podrían realizar estudios posteriores.

5.1. Sistemas de componentes y adaptabilidad.

Una de las inquietudes al iniciar este trabajo era la de determinar si los sistemas basados en componentes presentaban mecanismos de adaptabilidad que permitieran a los consultores de empresas, o a quienes resuelven acerca de las estrategias y políticas de las mismas, configurar los procesos de la organización en y con el software que los soporta, sin necesidad de reprogramar toda una aplicación.

Se puede decir que eso efectivamente se cumple. Esta conclusión se basa en las siguientes características de los sistemas basados en componentes:

- Los mecanismos de extensibilidad y de composición dinámica que fueron estudiados en las sub-secciones 4.1.2 y 4.1.3. Mediante estos mecanismos es posible modificar algún aspecto particular de la funcionalidad, sin alterar los demás, agregando funcionalidad por la vía de la extensión o modificando la existente por una variación en la composición del sistema.

- El estímulo al re-uso. Esto tiene el potencial de generar componentes prehechos que facilitan la extensión y la composición dinámica. Esta característica se ve incrementada por

la independencia de lenguaje de programación y de plataforma, soportada por los mecanismos de interoperabilidad existentes actualmente.

- La orientación al dominio. Como "extensión natural" de la orientación a objetos, los componentes tienden a representar auténticos conceptos del negocio, pero debido a su granularidad (mayor a la de los objetos), forman unidades de composición y despliegue que se relacionan mas adecuadamente con esos conceptos. Esta analogía de los componentes con los conceptos del negocio facilita la identificación de los aspectos a cambiar, en caso que así sea requerido, por las actividades de reingeniería de procesos de negocio y de la mejora continua.

En los ejemplos de arquitecturas presentados se aprecia de qué manera se pueden realizar modificaciones de la implementación, conservando los aspectos más estables de la misma y al mismo tiempo cómo los componentes que representan esos aspectos estables, pueden ser re-usados para colaborar en forma simultánea con otros, más variables, que representan procesos particulares.

5.2. La importancia del diseño y la metodología "top-down".

Sin duda todo lo precedente se apoya en que el diseño del sistema y su arquitectura aseguren la independencia de los componentes y la posibilidad de su reemplazo.

La granularidad de los componentes, que los relaciona con los conceptos del negocio, la especificación de sus interfaces, que establece las responsabilidades a cumplir en el sistema y un adecuado tratamiento de las dependencias entre componentes, que permite medir el impacto de su reemplazo o modificación, son todos aspectos que deben ser definidos al diseñar el sistema y para los cuales se han identificado un conjunto de principios de diseño que actúan como guías.

Sin un diseño que asegure un buen tratamiento de estos aspectos, las características señaladas en la sub-sección 5.1 difícilmente puedan verificarse en la práctica. Una metodología de tipo "top-down", en la que se identifiquen las características generales del dominio de aplicación y luego se pueda descomponer en los componentes que las implementan, favorece la aplicación de los principios de diseño y por ende asegura las facilidades de extensión y composición y la independencia de los componentes entre sí.

Esto descarta de plano los enfoques "oportunistas" de construcción del sistema en base a componentes preexistentes que se puedan encontrar. Está claro que en ese enfoque los componentes encontrados en primera instancia son los que van a definir la arquitectura del sistema y por lo tanto es probable que todo el sistema quede dependiente de ellos. En particular el autor de este trabajo opina que las denominaciones de "ladrillos de armar" ("building blocks") o las comparaciones con Lego, resultan más nocivas de lo que aparentan, en la medida que parece que omitieran la necesidad de un diseño del sistema.

Es cierto que la pretensión de uso de un componente prehecho puede hacer introducir cambios en la arquitectura del sistema y nadie va a dejar de usar un componente valioso porque afecte la arquitectura. En todo caso esto creará la necesidad de revisión de la estructura, lo cual será un proceso propio de diseño dónde se evaluarán los pro y los contra del cambio, tomándose una decisión ajustada al caso particular. Es, por lo tanto, admisible para algunos casos el enfoque "bottom-up", pero siempre que lleve a una decisión de diseño fundamentada y que no se pierda la visión global de la aplicación.

5.3. Las alternativas de los fabricantes de ERP.

Más allá de una intención de flexibilización, los fabricantes de ERP mantienen una política de sistemas propietarios. En el mejor de los casos presentan interfaces abiertas para la conexión de componentes de terceros con el objeto de obtener información del sistema o extender su funcionalidad.

Los ERP manejan la adaptabilidad y flexibilidad de las implementaciones a través de sus parámetros de configuración y facilidades para definir cadenas de procesos relacionados, lo cual es una alternativa también válida, sobre todo teniendo en cuenta la gran colección de mejores prácticas que encierran en sus aplicaciones, pero la solución que presentan no se basa en la composición dinámica o reemplazo de componentes.

A pesar de que en algunos casos se han re-programado en base a componentes, esto obedece más a razones de distribución o despliegue que a razones de composición dinámica de soluciones. En ese sentido no se alinean con las características que se han resaltado en este trabajo y a pesar de la evolución efectuada, semejan más sistemas monolíticos, que sistemas basados en componentes.

5.4. El impacto en el proceso de software.

La posibilidad de contar con componentes prehechos o, en caso contrario, de planificar el desarrollo paralelo de los mismos, impacta en la manera de hacer el software y en la estructura de la organización de desarrollo.

Por una parte surgirán sectores dedicados a la implementación de aplicaciones particulares, cuya actividad se orientará más a la integración y ensamblado de partes (composición) que al desarrollo propiamente dicho de esos componentes de software. El descubrimiento, selección, adaptación y ensamblado de los componentes adecuados para el caso concreto, en el marco de una arquitectura predefinida, será la actividad primordial de esos sectores.

Por otra parte surgirán sectores especializados en el dominio de aplicación, que se ocuparán de definir las arquitecturas de componentes adecuadas, la especificación de los componentes necesarios para conformarlas y eventualmente el desarrollo de los mismos.

En síntesis para la adopción de este paradigma será necesario adaptar los procesos de software con especificación de las actividades nuevas y anteriores que se integren a los ciclos de desarrollo.

También será necesario un cambio cultural, por ejemplo promover el re-uso (programación para el re-uso y re-uso en la programación) o redefinir la importancia del arquitecto de sistemas en el proyecto de desarrollo.

Inevitablemente se deberá atravesar una curva de aprendizaje y se deberán diseñar métodos y procesos propios y, como en todas las actividades, aprender de los errores cometidos, pero no reconocer de antemano que se deben efectuar estos cambios culturales, organizacionales y de los procesos puede conducir a situaciones caóticas (pérdida de visibilidad del sistema, costos excesivos en ingeniería de dominio, síndrome de re-uso etc.).

5.5. El mercado de componentes.

En la sub-sección 3.5.3 se establecieron algunas conclusiones respecto a la verificación de un mercado de componentes.

Se puede decir que hoy por hoy solo existe para componentes de tipo tecnológicos, componentes GUI, manejadores de bases de datos, componentes para encriptación / desencriptación de datos, por mencionar algunos. Si bien todos estos componentes son necesarios para desarrollar una aplicación, es claro que esta no se pude hacer contando solo con ellos.

Para el dominio de los sistemas de gestión de empresas no se verifica un mercado de componentes de lógica de negocio, que represente diferentes alternativa para implementar los procesos de la organización. Varios sitios de oferta de componentes de software visitados en Internet y las opiniones vertidas en documentos oficiales de OMG o de diversos autores confirman esta apreciación.

Si se descarta un mercado de componentes de negocio, se deben buscar mecanismos alternativos de provisión de componentes, lo que conduce a la propuesta efectuada de una estrategia de líneas de productos.

Sin embargo el estudio de las razones por las cuales no se verifica el mercado también puede reportar algunas ideas interesantes.

5.6. Razones que dificultan un mercado de componentes.

El área más preocupante en el desarrollo de sistemas de gestión de empresas debe ser, en la actualidad, la que corresponde a la implementación de la lógica y reglas de negocio (se asume que los aspectos de persistencia y representación gráfica están razonablemente solucionados). Esta área es justamente la que está más sujeta a cambios por razones de estrategia o política empresarial. ¿Porqué, entonces, no se ha desarrollado un mercado de componentes impulsado como oportunidad de negocio para los desarrolladores? Los clientes están presentes, los temas o funcionalidades a resolver son conocidos, sin embargo no se verifica ese mercado.

Uno de los motivos que pueden dificultar el surgimiento de esos mercados es que si bien las funcionalidades son conocidas y compartidas, lo que falta, es la definición del producto, esto es: la especificación, aceptada por los usuarios con generalidad, de los componentes requeridos y su interrelación, lo que es expresado a través de una arquitectura común o un “diseño dominante”.

Esta hipótesis podría ser objeto de comprobaciones futuras. De todos modos, se cuenta con ejemplos extraídos de otras disciplinas que refuerzan la idea: por ejemplo hasta que se definió una arquitectura de hardware dominante, la Motherboard de Intel, no había una estandarización ni especificación de compatibilidad para partes de computadoras. Una vez que se hubo impuesto esa arquitectura, el mercado entero de fabricantes de partes se vio revolucionado; todos sabían cual era el diseño que tenían que conformar y que éste había sido adoptado en forma masiva por el mercado. Una vez establecido ese estándar y con un gran número de fabricantes produciendo partes, el mismo modelo evolucionó hacia la facilidad de conexión con el computador (la composición de un hardware): el concepto de “plug-and-play”.

Es interesante observar que el diseño dominante no surgió de las arquitecturas de los grandes computadores, sino de los casi más pequeños, destinados en un principio al uso personal.

Un fenómeno similar puede ocurrir con relación al desarrollo de los sistemas de gestión de empresas. Si, por cualquiera sea la causa, un sistema de gestión basado en componentes se impone como diseño dominante, es muy probable que el mercado de componentes compatibles se establezca rápidamente y los mecanismos de composición dinámica se desarrollarán facilitando más aun la implementación de los procesos de la empresa.

5.7. La propuesta de elaboración de líneas de producto.

La descripción del concepto de líneas de productos y cómo estas impactan en el proceso de software ya fue tratado en las sub-secciones 2.3.2 y 3.6.

Ante la inexistencia de un "diseño dominante" para los sistemas de gestión e información empresarial, que frena el surgimiento de un mercado de componentes de negocio, la estrategia de desarrollo de líneas de producto ofrece la posibilidad de trabajar con una arquitectura predefinida (compartida por el grupo de usuarios de miembros de la línea de productos) y un conjunto de componentes prehechos para la misma.

Se debe tener claro que esta alternativa mantiene ciertas características propietarias del sistema de aplicación y de los componentes opcionales y su espectro de aplicación es mucho más limitado que el de un mercado de componentes. Sin embargo a través de esta vía se puede dar una buena respuesta a las necesidades de composición dinámica y constituye una base para el establecimiento de diseños dominantes sectoriales.

Para empresas que se dediquen al desarrollo de software para el mercado como actividad principal, apuntando a usuarios finales, constituye además una buena herramienta para promover la estandarización y el re-uso de sus componentes

5.8. Identificación de principios de diseño.

En el desarrollo de la sección 4.2, relativa al diseño de sistemas basados en componentes, se identificaron una serie de principios, generalmente aplicables al diseño y programación orientado a objetos, pero que constituyen una buena guía para la definición de algunos aspectos determinantes de los componentes. Los principios identificados fueron resumidos en la Tabla 2.

Dada la similitud existente entre los conceptos básicos del paradigma de orientación a objetos y los de los del CBD, el traslado de estos principio resulta casi directo, no obstante se entiende que el haberlo hecho explícito resulta un aporte interesante para quienes deben realizar el diseño y arquitecturas de sistemas basados en componentes.

5.9. Algunas líneas de estudio a realizar.

Una línea importante de estudio complementario es la que tiene que ver con la definición de los procesos de software y ciclos de vida de desarrollo para el paradigma CBD. Según Ivar Jacobson[22] [Jacobson01] el tema está resuelto con la aplicación de RUP [Rational02], sin embargo no deben descartarse los procesos incluidos en otras metodologías como Catalysis [D'Souza98] o extensiones como las propuestas en [Levi02].

[22] Ivar Jacobson es uno de los autores de UML (Unified Modeling Language) y cofundador de Rational Software Inc.

También vinculado al proceso de software basado en componentes, recientemente ha habido una serie de publicaciones y trabajos presentados en [ICSE00] e [ICSE01], acerca del testing, predicción de rendimiento y certificación de componentes. Estos aspectos tienen que ver con el aseguramiento de la calidad del software y por tanto se deberían realizar estudios complementarios, que ayuden a definir esa parte del proceso de desarrollo.

La estrategia de líneas de productos, en lo que tiene que ver con instanciar cada uno de los miembros, requiere el desarrollo de lenguajes de composición. Los lenguajes para especificación de arquitecturas (Architectural Description Language, ADL) apuntan a la formalización de los diseños, pero estos pueden ser complementados con lenguajes para la composición “plug-and-play”, de los componentes de la aplicación particular, basados en “ecuaciones de tipo” y “reglas de diseño” como las propuestas en [Batory97] o en perfiles de UML, como las propuestas de OMG y la notación para CCA (Component Collaboration Architecture) [OMG01b].

6. Glosario de términos.

Principales términos utilizados en este informe:

Análisis de dominio	Actividad incluida en la ingeniería de dominio, mediante la cual se identifica, captura y organiza la información que se usa para el desarrollo de sistemas en el dominio, para poder ser re-usada en el desarrollo de nuevas aplicaciones.
Arquitectura del sistema	Estructura organizacional de un sistema.
Componente de negocio	Realización, en todo el ciclo de desarrollo de un concepto del negocio. Una construcción de análisis, diseño, elaboración, empaquetado y despliegue.
Componente de software	Unidades binarias de despliegue, que implementan una o más interfaces bien definidas, dando acceso a un conjunto de funcionalidades interrelacionadas y que puedan adaptar su comportamiento en forma predefinida.
Composición dinámica	Estructuración o ensamblado de componentes en "tiempo de ejecución". En este trabajo se usa este término para denotar "sin necesidad de reprogramación".
Desarrollo de sistemas basado en componentes (CBD)	Paradigma de desarrollo de sistemas que se basa en el ensamblado de componentes preexistentes más que en el desarrollo tradicional específico de cada módulo del sistema.
Enterprise Resource Planning (ERP)	Paquetes de software de mercado, estándar y configurables que integran los sistemas de información empresarial con sistemas de gestión de procesos basados en esa información, dentro y a través de las distintas áreas funcionales de la empresa.
Ingeniería de dominio	Disciplina de software que incluye la definición de los alcances de un dominio, el análisis del mismo, la definición de la estructura de sistema para el dominio y la construcción de componentes independientes de las aplicaciones particulares.
Interfaz	Abstracción de todas las implementaciones posibles que pueden cumplir un rol determinado en un sistema compuesto.
Interfaz provista o exportada	Abstracción de los servicios ofrecidos o soportados por un componente.
Interfaz requerida o importada	Abstracción de los servicios requeridos a terceras partes por un componente, para complementar su funcionalidad.
Línea de productos	Familia de sistemas interrelacionados de un dominio, que comparten un conjunto de requerimientos, una arquitectura y un conjunto de componentes, pero presentan una serie de diferencias sustanciales que los justifican como productos distintos.

Mejora continua	Proceso sostenido de planificación de mejora de procesos, aplicación, evaluación y corrección del plan, como actividad permanente basado en el ciclo PDCA (Plan, Do, Check, Act).
Negocios Basados en Componentes	Modelo de representación de la empresa, que la descompone en diferentes procesos y productos que articulan entre sí para lograr resultados de valor.
Proceso	Conjunto de actividades realizadas con un propósito determinado.
Proceso de Software	Conjunto de actividades, métodos, prácticas y transformaciones usados por personas para desarrollar y mantener software y productos asociados.
Reingeniería de procesos de negocios	Rediseño de los procesos de una empresa, para conseguir mejoras sustanciales en áreas críticas, desde el punto de vista económico, productivo y de la calidad.
Sistema Abierto	Un sistema de aplicación que presenta interfaces bien definidas para extender su funcionalidad.
Sistema basado en componentes	Sistema construido por composición de partes reemplazables, de acuerdo a un diseño expresado por una arquitectura del sistema.
Sistemas de Gestión e Información Empresarial	Sistemas informáticos que soportan la información de la empresa (contabilidad, inventarios de deudores y acreedores, inventarios de existencias, etc.) y la definición, ejecución y control de los procesos principales (cadena de suministros, circuito producción, proceso de ventas, etc.).

7. Glosario de siglas.

Algunas siglas utilizadas en este informe.

ADL	Architectural Description Language
B2B	Business to business
CBD	Component Based Development
CBSE	Component Based Software Engineering
CCA	Component Collaboration Architecture (OMG)
CCM	CORBA Component Model
CEE	Component Execution Environment
CMU	Carnegie Mellon University
COM	Component Object Model (Microsoft)
CORBA	Common Object Request Broker Architecture (OMG)
COTS	Components Off The Shelf
DBMS	Data Base Management System
DCE	Distributed Computing Environment (OSF)
DCOM	Distributed COM
EAI	Enterprise Application Integration
EJB	Enterprise Java Beans
GUI	Graphic User Interface
ICSE	International Conference on Software Engineering
IDE	Integrated Development Environment
J2EE	Java 2 Enterprise Edition
OMG	Object Management Group, grupo de fabricantes que especifica estándares para software con orientación a objetos.
OOPSLA	Object Oriented Programming, Systems, Languages and Applications, conferencia anual.
ORB	Object Request Broker
OTAN	Organización del Tratado del Atlántico Norte
RMI	Remote Method Invocation (Java, JVM)
RFC	Remote Function Call (SAP)
RUP	Rational Unified Process
SEI	Software Engineering Institute (Carnegie Mellon University)
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
XML	eXtended Markup Language

8. Bibliografía

8.1. Bibliografía utilizada

Batory97 — Batory, Don y Geraci, Bart. *Composition Validation and Subjectivity in GenVoca Generators.* IEEE transactions on Software Engineering, Vol 23 Nro. 2, Feb 1997.

Booch99 — Booch, Grady et al.. *The Unified Modeling Language User Guide*. Addison Wesley, USA, 1999.

Bosch00 — Bosch, Jan. *Design and Use of Software Architectures.* Addison Wesley, USA, 2000.

Brooks87 — Brooks, Fred. *No Silver Bullet: Essence and Accidents of Software Engineering.* IEEE Computer, Vol. 20, No. 4, Abril 1987.

Fayad01 — Fayad, Mohamed y Altman, Adam. *An Introduction to Software Stability.* Communications of the ACM, Vol. 44 No. 9 pp. 95-98, Set. 2001.

Fayad02a — Fayad, Mohamed. *Accomplishing Software Stability.* Communications of the ACM., Vol. 45, Nro.1, pp 111-115, Enero, 2002.

Fayad02b — Fayad, Mohamed. *How to Deal with Software Stability.* Communications of the ACM, Vol. 45 Nro. 4, pp 109-112, Abr. 2002.

Gamma95 — Gamma, Erich et al. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, USA, 1995.

Hopkins00 — Hopkins, Jon. *Component Primer.* Communications of the ACM, Vol. 43, No. 10 pp. 27-30, Oct. 2000.

Kumar00 — Kumar, Kuldeep y van Hillegersberg, Jos. *ERP Experiences and Evolution.* Communications of the ACM, Vol. 43, No. 44 pp. 23-26, Abr. 2000.

Levi02 — Levi, Keith y Arsanjani, Ali. *A Goal Driven Approach to Enterprise Component Identification and Specification.* Communications of the ACM, Vol. 45, No. 10, pp. 45-52, Oct. 2002.

OMG96 — Object Management Group. *Common Facilities RFP-4. Common Business Objects and Business Object Facility.* OMG TC Document CF/96-01-04, 1996.

OMG01a — Object Management Group. *Account Receivable – Account Payable. AR/AP Facility. Request for Proposal.* OMG Document: finance/01-04-04, 2001.

OMG01b — Object Management Group. *A UML Profile for Enterprise Distributed Object Computing.* OMG Document ad/2001-06-09 – Part 1, 2001.

OMG02 — Aartsen, Amud et al. *Revised Submission in Response to OMG's Finance DTF RFP for an AR/AP Facility.* OMG DTC Document: finance/02-03-01, Marzo 2002.

Parnas72 Parnas, David. *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM, Vol. 15, Nro. 12, pp. 1053-1058, Dic. 1972.

Pfleeger01 Pfleeger, Shari. *Software Engineering: Theory and practice.* 2nd. Ed. Prentice Hall, New York, USA, 2001.

Sawyer01 Sawyer, Steve. *A Market-Based Perspective on Information Systems Development.* Communications of the ACM, Vol. 44, No. 11 pp. 97-102, Nov. 2001.

Seldon67 Seldon, Arthur y Pennance, F.G. *Diccionario de Economía.* Traducción de Casahuaga, Antonio. Oikos-tau Ediciones, Barcelona, España, 1967.

Sims98 Sims, Oliver y Eeles, Peter. *Building Business Objects*. John Wiley & Sons, USA, 1998.

Soh00 Soh, Christina et al. *Cultural Fits and Misfits: Is ERP a Universal Solution?* Communications of the ACM, Vol. 43, No. 4 pp. 47-51, Abr. 2000.

Sparling00a Sparling, Michael. *Lessons learned through six years of component-based development.* Communications of the ACM, Vol. 43, No. 10 pp. 47-53, Oct. 2000.

Sprott00 Sprott, David. *Componentizing the Enterprise Application Packages.* Communications of the ACM, Vol. 43, No. 4 pp. 63-69, Abr. 2000.

Szyperski97 Szyperski Clemens. *Component Software. Beyond Object-Oriented Programming.* ACM Press y Addison-Wesley, UK, 1997.

Veryard01 Veryard, Richard. *The Component-Based Business: Plug and Play*. Practitioner series Springer-Verlag, UK, 2001.

8.2. Artículos y páginas de internet consultados

Albert02 Albert, Cecilia y Brownsword, Lisa. *Evolutionary Process for Integrating COTS-Based Systems (EPIC): An Overview.* Technical Report CMU/SEI-2002-TR-009, Software Engineering Institute, Carnegie Mellon University, Julio de 2002.
http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr009.pdf
Ultima visita: 22/10/2002

Allen00 Allen, Paul. Texto extraído de *Realizing e-Business with Components.* Addison-Wesley, USA, 2000.
http://www.flashline.com
Ultima visita: 15/03/2002

Allen01 Allen, Paul. *The State of the Practice.* Component Development Strategies, Cutter Information Corp., Vol. XI, No 3, Marzo 2001.
http://www.cutter.com/cds/cds0103.html
Ultima visita: 17/04/2002

Batory94 — Batory,Don. *Products of Domain Models.* 1994
ftp://ftp.cs.utexas.edu/pub/predator/products.pdf
Ultima visita: 26/06/2002

Batory98 — Batory, Don. *Product-Line Architectures*. Conferencia de Smalltalk and Java in Industry and Practical Training, Erfurt, Alemania, Oct. 1998.
ftp://ftp.cs.utexas.edu/pub/predator/stja.pdf
Ultima visita: 26/06/2002

Batory99 — Batory, Don y Smaragdakis, Yannis. *Building Product Lines With Mixin Layers.* Workshop On Object Technology for Product Lines Architectures. European Software Institute, 1999.
http://www.esi.es/Projects/Reuse/Praise/pdf/ses1-3.pdf
Ultima visita: 31/07/2002

Batory01 — Batory, Don. *A Standard Problem for Evaluating Product-Line Methodologies.* Third International Conference on Generative and Component-Based Software Engineering, Erfurt, Alemania, Set. 2001.
ftp://ftp.cs.utexas.edu/pub/predator/gpl.pdf
Ultima visita: 26/06/2002

Bredemeyer02 — Bredemeyer Consulting. *Software Architecture in Context.* 2002
http://www.bredemeyer.com/where.htm
Ultima visita: 10/07/2002

Collins00 — Collins-Cope, Mark y Matthews, Hubert. *Components in Financial Systems.* 2000 International Workshop on Component Based Software Engineering. 22nd. International Conference on Software Engineering(ICSE2000), Limerick, Ireland, June 5-6, 2000
http://www.sei.cmu.edu/cbs/cbse2000/papers/01/01.pdf
Ultima visita: 21/07/2002

Collins01a — Collins-Cope, Mark. *Component Based Development and Advanced OO Design.* Oct. 2001.
http://www.ratio.co.uk/techlibrary.html
Ultima visita: 5/08/2002

Collins01b — Collins-Cope, Mark y Matthews, Hubert. *A Reference Architecture for Component Based Development.* Component Developer's and User's Forum 2001 - Frankfurt, Alemania, 2001.
http://www.ratio.co.uk/techlibrary.html
Ultima visita: 5/08/2002

ComponentSource — Mercado clasificado de componentes de software
http://www.componentsource.com
Ultima visita: 12/09/2002

D'Souza98 — D'Souza, Desmond. *Objects, Components and Frameworks with UML The Catalisys Approach.* Icon Computing, 1998.
http://www.mastersoft.com.br/ArtigosTecnicos/pagina_artigos.html
Ultima visita: 4/10/2002

Flashline — Mercado clasificado de componentes de software.
http://www.flashline.com
Ultima visita: 12/09/2002

Fröhlich01 — Fröhlich, Peter y Franz, Michael. *On Certain Basic Properties of Component-Oriented Programming Languages.* OOPSLA 2001, Florida, USA, Octubre de 2001.
http://www.ccs.neu.edu/home/lorenz/oopsla2001/23_Frohlich.pdf
Ultima visita: 3/07/2002

Griss01 — Griss, Martin. *CBSE Success Factors: Integrating Architecture, Process and Organization.* Capítulo 9 de Component-Based Software Engineering: Putting the Pieces Together, George Heineman y William Councill editores, Addison-Wesley, 2001.
http://www.hpl.hp.com/reuse/papers/cbse-success-factors.pdf
Ultima visita: 10/08/2002

Haines97 — Haines, Gary et al. *Component Based Software Development / COTS Integration*. Software Technology Review, SEI-CMU, última revision: Octubre de 1997.
http://www.sei.cmu.edu/str/descriptions/cbsd.html
Ultima visita: 17/04/2002

Herzum98 — Herzum, Peter y Sims, Oliver. *The Business Component Approach*. OOPSLA´98 Business Object Workshop IV, Vancouver, Canada, Octubre de 1998.
http://jeffsutherland.org/oopsla98/sims.html
Ultima visita: 16/11/2001

ICSE00 — 3rd Workshop On Component-Based Software Engineering, 22nd International Conference On Software Engineering, Limerick, Irlanda, Junio de 2000.
http://www.sei.cmu.edu/cbs/cbse2000/index.html
Ultima visita: 5/08/2002

ICSE01 — 4th Workshop On Component-Based Software Engineering, 23rd International Conference On Software Engineering, Toronto, Canadá, Mayo de 2001.
http://www.sei.cmu.edu/pacc/CBSE4-Proceedings.html
Ultima visita: 5/08/2002

Jacobson01 — Jacobson, Ivar y Griss, Martin. *The Long Awaited Promise of Component Reuse is About to be Achieved*. Approaching the Promised Land of Component Reuse. Application Development Trends, Junio 2001
http://www.hpl.hp.com/reuse/papers/adt-june-2001.pdf
Ultima visita: 10/08/2002

Kang99 — Kang, Kyo. *Issues in Component-Based Software Engineering*. 1999 Workshop on Component-Based Software Engineering, 21th. International Conference on Software Engineering, Los Angeles, USA, Mayo 1999.
http://www.sei.cmu.edu/cbs/icse99
Ultima visita: 1/10/2002

Kean98 — Kean, Liz. *Domain Engineering and Domain Analysis*. Software Technology Review, SEI-CMU, última revision: febrero de 1998.
http://www.sei.cmu.edu/str/descriptions/deda.html
Ultima visita: 17/04/2002

Kim99 — Kim, Bonn-Oh. *Component-Based ERP Design ins a Distributed Object Environment.* 1999 Workshop on Component-Based Software Engineering, 21th. International Conference on Software Engineering, Los Angeles, USA, Mayo 1999.
http://www.sei.cmu.edu/cbs/icse99
Ultima visita: 1/10/2002

Malan02a — Malan, Ruth y Bredemeyer, Dana. *Software Architecture: Central Concerns, Key Decisions.* Bredemeyer Consulting, 2002
http://www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF
Ultima visita: 10/07/2002

Malan02b — Malan, Ruth y Bredemeyer, Dana. *The Visual Architecting Process.* Bredemeyer Consulting, 2002.
http://www.bredemeyer.com/pdf_files/VisualArchitectingProcess.PDF
Ultima visita: 10/07/2002

Martin96a — Martin, Robert. *The Liskov Substitution Principle*. The C++ Report Vol. 8 Nro. 3, Mar 1996
http://www.objectmentor.com/resources/articles/lsp.pdf
Ultima visita: 30/09/2002

Martin96b — Martin, Robert. *Interface Segregation Principle*. The C++ Report Vol. 8 Nro. 8, Ago. 1996
http://www.objectmentor.com/resources/articles/granularity.pdf
Ultima visita: 30/09/2002

Martin96c — Martin, Robert. *Granularity*. The C++ Report Vol. 8 Nro. 10, Nov-Dic 1996
http://www.objectmentor.com/resources/articles/granularity.pdf
Ultima visita: 30/09/2002

Martin97 — Martin, Robert. *Stability*. The C++ Report, Vol. 9 Nro. 2, Feb 1997
http://www.objectmentor.com/resources/articles/stability.pdf
Ultima visita: 30/09/2002

McCarthy82 — McCarthy, William. *The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment.* The Accounting Review, Vol. LVII, No. 3, Julio de 1982.
http://www.msu.edu/user/mccarth4/McCarthy.pdf
Ultima visita: 26/03/2002

McIlroy68 — McIlroy, Douglas. *Mass-Produced Software Components.* NATO Conference on Software Engineering, Garmisch, Germany, 7 to 11 October ,1968.
http://www.ericleach.com/massprod.htm
Ultima visita: 16/04/2002

Meyer99 — Meyer, Bertrand. *The Significance of Components.* Software Development Magazine. Noviembre, 1999.
http://www.sdmagazine.com/columnists/meyer
Ultima visita: 17/09/2002

Meyer00 — Meyer, Bertrand. *What to compose.* Software Development Magazine, Marzo 2000.
http://www.sdmagazine.com/columnists/meyer
Ultima visita: 15/08/2002

Netdecisions01 — *Ten Principles of Frameworks Methodology.* Netdecisons Limited. UK. Octubre de 2001.
http://www.cbdiforum.com/bronze/frame_method.php3
Ultima visita: 15/05/02

Ning99 — Ning, Jim. *A Component Model Proposal.* 1999 Workshop on Component-Based Software Engineering, 21st. Iternational Conference On Software Engineering, Los Angeles, USA, Mayo 1999
http://www.sei.cmu.edu/cbs/icse99
Ultima visita: 1/10/2002

Oberndorf97 — Oberndorf, Tricia. *COTS and Open Systems – An Overview*. Software Technology Review, SEI-CMU, última revision Agosto de 1997.
http://www.sei.cmu.edu/str/descriptions/cots.html
Ultima visita: 17/04/2002

Persson00 — Persson, ,Erik. *Business Object Components?.* OOPSLA 2000 Business Object Component Workshop, Minnesota, USA, Octubre de 2000.
http://jeffsutherland.org/oopsla2000/persson/persson.htm
Ultima visita: 27/03/2002

Rational02 — Rational Software. *The Rational Unified Process for Systems Engineering.* Mayo de 2002.
http://www.rational.com/media/whitepapers/TP165.pdf
Ultima visita: 12/09/2002

SAP02 — Sitio Web de SAP.
http://www.sap.com
Indice de componentes para SAP de acuerdo a tipo de software
http://www.sap.com/partners/software/directory/directory.asp?softcat
Ultima visita: 20/05/02

Smaragdakis99 — Smaragdakis, Ioannis. *Implementing Large Scale Object Oriented Components.* Tésis de Doctorado en el departamento de Computer Science de la Universidad de Texas, 1999.
ftp://ftp.cs.utexas.edu/pub/predator/yannis-thesis.pdf
Ultima visita: 22/06/2002

Sparling00b — Sparling, M. *Is there a component market?.*
http://ww.cbd-hq.com/articles/2000/000606ms_cmarket.asp
Ultima visita: 6/12/2001.

Sun99 Sun Microsystems Inc. *What Is the Java™ 2 Platform, Enterprise Edition?*. 1999
http://java.sun.com/j2ee/sdk_1.2.1/techdocs/guides/j2ee-overview/Introduction.fm.html#7916
Ultima visita: 8/05/2002

Sun02 Sun Microsystems Inc. *Enterprise JavaBeans™ Components and CORBA Clients: A Developer Guide.* 2002
http://java.sun.com/j2se/1.4/docs/guide/rmi-iiop/interop.html
Ultima visita: 08/05/2002

Szyperski01 Szyperski, Clemens. *Components…What the heck?*. Trascripción de la charla presentada en la 4th. Workshop on Component-Based Software Engineering (CBSE4) de la International Conference on Software Engineering 2001, Toronto, Canadá, 2001.
http://www.sei.cmu.edu/pacc/CBSE4_papers/Szyperski-Keynote.pdf
Ultima visita: 23/09/2002

Szyperski02 Szyperski, Clemens. *Universe of Composition*. Software Development Magazine, Agosto, 2002.
http://www.sdmagazine.com/columnists/szyperski/
Ultima visita: 15/09/2002

Vbxtras Mercado de componentes COM y ActiveX
http://www.vbxtras.com
Ultima visita: 12/09/2002

Wilkes02 Wilkes, Lawrence. *Buying Software Components.* CBDi Forum, 2002
http://www.cbdiforum.com/public/sigs/buysell/buy_comp_report.php3
Ultima visita: 3/06/2002

Anexos.

Anexo 1. Ejemplo del problema de "fragile base class".

El problema de *"fragile base class"* semántico suele aparecer cuando hay invocaciones recursivas (self) a métodos de una clase. Las clases derivadas dependen de estas invocaciones recursivas de la superclase y a veces no es posible sobrescribir métodos sin conocer el código fuente de la clase base, más aún, modificaciones en la clase base pueden afectar los métodos sobrescritos en las clases derivados dejándolos inconsistentes.

A continuación se describe un ejemplo muy sencillo, casi trivial, pero que refleja la naturaleza del problema.

Tomando por ejemplo una clase que sea un contenedor de una cola, con métodos: ALTA(item:ANY), para agregar un ítem a la cola, BAJA( ) para eliminar el último ítem de la cola y BAJAMUL(n:INTEGER), para eliminar los n últimos ítems de la cola. Si queremos extender la funcionalidad, agregando un atributo que nos indique la cantidad de items en la cola podemos hacerlo derivando una subclase que sobrescriba los métodos: ALTA (item:ANY) además de dar de alta el ítem agregará uno a un contador, BAJA ( ) además de eliminar un ítem restará uno del contador. Cuando llegamos al método BAJAMUL (n:INTEGER), surge la dependencia con la implementación en la clase base: si fue implementado como n llamadas sucesivas al método BAJA ( ) el problema ya está resuelto y no hay que sobrescribir, en caso contrario sí hay que hacerlo. En el primer caso además, si el método BAJAMUL (n: INTEGER) es modificado en la clase base, por ejemplo no hace más una invocación al método BAJA ( ), el método en la subclase puede quedar inconsistente [Fröhlich01].

Este es un ejemplo trivial y muy fácil de reconocer y solucionar, el problema es que cuando ocurre el problema de "fragile base class" en situaciones más complejas, es mucho más difícil de identificar y por tanto mucho más nocivo.

Anexo 2. Ejemplo de incumplimiento de LSP.

La violación de LSP (Liskov Substitution Principle), ocurre cuando se sustituye un objeto por otro, derivado del primero, pero que no conforma su comportamiento.

Un ejemplo de violación de LSP es el caso de la derivación de una clase CUADRADO a partir de una clase base RECTÁNGULO, ambas representando las propiedades típicas de esas figuras geométricas. Si bien desde el punto de vista geométrico y por ser de la relación "*ES UN*" (el cuadrado *es un* caso particular del rectángulo), la derivación parece correcta, los problemas surgen cuando se quiere usar la clase derivada en lugar de la clase base.

Se asume, para este ejemplo, que la clase RECTÁNGULO tiene atributos Largo y Ancho (reales) y métodos SetLargo (l:REAL), SetAncho(a:REAL), para establecer sus lados, y una función Perímetro ( ):REAL que suministra el perímetro de rectángulo; se puede presumir que además tiene otros métodos y atributos propios de la naturaleza de la figura geométrica, pero que no interesan en este ejemplo.

Un módulo cliente cualquiera, usará los métodos Set... para establecer los lados de la figura. En el caso que se esté referenciando a un objeto de la clase CUADRADO en sustitución de un objeto de la clase RECTÁNGULO, surge un primer problema: al invocar, por ejemplo, SetLargo (5) el atributo de Largo quedará con ese valor, pero el Ancho no será alterado, por lo que sus lados no serán iguales y habrá perdido su naturaleza de CUADRADO, lo que resulta

inconsistente con el objeto. Analizando este problema, se puede concluir rápidamente que se deben sobrescribir los métodos de seteo, haciendo que el otro lado tome el mismo valor que el que se está seteando: los métodos SetLargo y SetAncho deben ser sobrescritos, asignando a los atributos de Largo y Ancho el mismo valor pasado como parámetro. Esto también se pude solucionar definiendo un invariante que fuerce a que los dos lados tengan el mismo valor. El sistema asi corregido parece consistente, pues cualquiera sea la invocación de seteo que se utilice, los objetos no perderán su naturaleza geométrica.

Sin embargo la consistencia de un sistema debe ser establecida por el uso del mismo y no por la consistencia interna de sus componentes, en otras palabras lo que importa es el comportamiento en las colaboraciones con otros objetos. Si en la nueva versión un cliente invocara los métodos SetLargo(10) y luego SetAncho (3), esperará que la función Perímetro devuelva un valor 26, pero si el objeto referenciado es un CUADRADO el valor devuelto será 12, siendo esto también una inconsistencia, ya que quien programó ese cliente esperaba encontrar un rectángulo.

Lo que ocurre es que no se han observado las reglas del *"diseño por contrato"*; por ejemplo, la poscondición de SetAncho(a) para el RECTÁNGULO podría ser: {(Ancho=a) & (Largo=old.Largo)} en cambio para CUADRADO la poscondición para la misma función sería: {(Ancho=a) & (Largo=a)}. Esta última poscondición es más débil que la primera ya que no incluye la cláusula (Largo=old.Largo) y en ese caso viola las reglas de Meyer y el principio de LSP.

Un cuadrado es un rectángulo, pero un objeto de clase CUADRADO no es un objeto de clase RECTÁNGULO, ya que su comportamiento es diferente y por lo tanto no lo puede sustituir.

Anexo 3. Tecnología de componentes distribuidos de Sims y Herzum.

El otro aspecto en que se basa la implementación de los componentes de negocio de Sims y Herzum como elementos autónomos de construcción y despliegue es una tecnología de componentes distribuidos.

El modelo que proponen Sims y Herzum se basa en la propuesta relacionada con la solicitud de especificación de OMG *Common Facilities RFP-4 Common Business Objects and Business Object Facility* [OMG96], presentada por Systems Software Associates (SSA) [Sims98] [Herzum98] y corresponde al esquema de la Figura 16.

Los componentes de negocio son descompuestos en las diferentes capas de la arquitectura y en cada una de ellas la funcionalidad correspondiente es implementada por *componentes distribuidos*[23] que son módulos ejecutables (dll, exe, etc.) desarrollados independientemente. La tecnología de componentes distribuidos oculta a los programadores muchos aspectos complejos de implementar, como manejo de threads, protocolos de comunicación, gestión de concurrencia, etc., a través de una infraestructura especializada, el Business Object Facility (BOF). Esta infraestructura forma una capa de separación que además hace que el componente distribuido sea técnicamente independiente del software de base, tal como el ORB usado o el sistema operativo en que corre.

En tiempo de ejecución y basándose en especificaciones de diseño, la infraestructura compone uno o más componentes distribuidos formando un *objeto distribuido* con visibilidad de red, cuya estructura se aprecia en la Figura 16. Es este el que implementa la funcionalidad del componente de negocio correspondiente a las responsabilidades de cada capa.

[23] En el sentido de que pueden ser reubicados físicamente.

Cada uno de esos objetos distribuidos tiene una interfaz definida en la etapa de diseño y los componentes distribuidos que lo componen implementan esa interfaz. Los objetos del lenguaje que forman cada uno de los componentes distribuidos sólo son visibles dentro de los confines de ese componente.

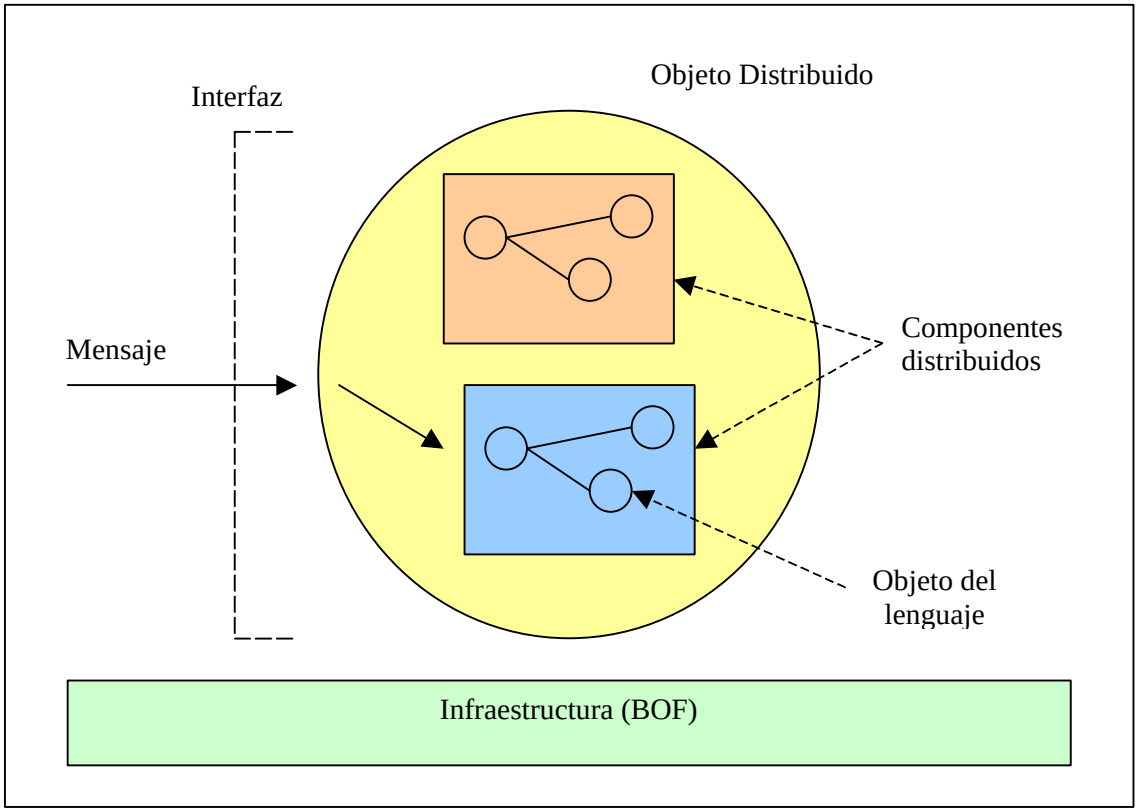

Figura 16. Estructura de un Objeto Distribuido.

En general la composición de más de un componente distribuido en un solo objeto distribuido se da por ejemplo cuando hay algún código común, independiente de la aplicación (gestión de persistencia, manejo de eventos, etc.) que no está incluido en el componente distribuido que representa la lógica del negocio y que por este mecanismo se pueden unir a él en tiempo de ejecución. También presenta una manera de adaptar los componentes distribuidos a un caso particular de aplicación, mediante extensión, con un mecanismo del estilo de la delegación.

Los mensajes enviados desde otros objetos distribuidos son orientados por la infraestructura (y el middleware de interoperabilidad) hacia el componente distribuido. Si hay más de un componente distribuido en el objeto distribuido, el mensaje es enviado al primero de una lista establecida en diseño, si este no lo maneja es enviado al segundo y así sucesivamente, implementando de ese modo un mecanismo de delegación automática que permite la extensión de los componentes originales sin tocar su código y aún mediante desarrollos en otros lenguajes de programación.

Asimismo como el objeto distribuido se compone en tiempo de ejecución, un componente independiente de la lógica de la aplicación puede ser usado para componer más de un objeto distribuido, favoreciendo el re-uso y la estabilidad de la aplicación.