

	<b>INGENIERÍA EN INFORMÁTICA – PLAN 2003</b> <b>DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE</b>	
	<b>APUNTE DE TEMAS DIVERSOS</b>	<b>VERSIÓN: 1.0</b> <b>VIGENCIA: 06-08-2009</b>

## INGENIERÍA DE SOFTWARE

### Introducción

La Ingeniería del Software es una disciplina relativamente reciente, sin embargo, a pesar de su relativa inmadurez, su evolución ha sido y sigue siendo vertiginosa. El desarrollo de software ha pasado de ser una práctica semiartesanal de codificación de programas línea a línea, implementados por lo general bajo demanda y a medida, a convertirse en un proceso mucho más sistemático, soportado por herramientas de alto nivel y metodologías más o menos formales que permiten explotar cualidades como la reutilización, la flexibilidad, o la portabilidad, entre otras muchas. Así, la tendencia actual es a desarrollar productos software estandarizados, fáciles de integrar con los ya existentes y que puedan configurarse de manera sencilla para adaptarse a las necesidades específicas de cada cliente.

La amplia variedad de sistemas gestionados actualmente mediante software ha disparado la complejidad de las aplicaciones que es necesario desarrollar. Por otra parte, dada la velocidad con la que se suceden los avances tecnológicos, sobre todo en áreas como la electrónica o las comunicaciones, resulta imprescindible dotar a las aplicaciones de la flexibilidad suficiente para que puedan asumir de manera ágil los continuos cambios impuestos por el mercado (nuevas plataformas hardware cada vez más flexibles, potentes y económicas, entornos distribuidos, cooperativos y online, nuevos protocolos de comunicaciones, etc.). Además, resulta igualmente esencial que el logro de esta mayor flexibilidad no sea a costa de perjudicar otros parámetros como la eficiencia, el coste o el tiempo de desarrollo de las aplicaciones.

En respuesta a estas nuevas demandas, en las últimas décadas han surgido nuevos paradigmas de desarrollo de software promovidos, no sólo desde el ámbito científico-académico, sino también fuertemente respaldados por muchas de las empresas líderes del sector informático (Sun Microsystems®, Microsoft®, etc.). Entre estas nuevas propuestas cabe destacar las siguientes:


- Desarrollo de Software Basado en Componentes (CBSD)
- Líneas de Productos Software (SPL)
- Generación Automática de Software (GP)
- Desarrollo de Software Orientado a Aspectos (AOSD)
- Arquitecturas Dirigidas por Modelos (MDA)

A pesar de la revolución que estos nuevos paradigmas han supuesto para la Ingeniería del Software, la falta de consenso en cuanto a las metodologías y notaciones que se deben emplear, así como la ausencia de herramientas que den un soporte integral al desarrollo de aplicaciones siguiendo una o más de estas nuevas propuestas hace que, a fecha de hoy, éstas no resulten tan efectivas en la práctica como cabría esperar.

Además, a la vista de los trabajos publicados sobre casos de estudio desarrollados con éxito siguiendo algunas de estas nuevas tendencias, se observa que son muy pocas las experiencias descriptas hasta el momento en el área de informática de sistemas en comparación con las llevadas a cabo en el ámbito de la informática de gestión. Esto puede deberse, entre otras, a las siguientes razones:

- (1) Las aplicaciones de sistemas suelen implementarse como rutinas optimizadas para resolver tareas muy específicas. Su enorme especificidad hace difícil su reutilización, del mismo modo que la necesidad de optimizar su rendimiento limita sensiblemente su grado de flexibilidad.
- (2) Por lo general, el software de sistemas resulta difícil de abstraer de la plataforma hardware para la que se construye y a la que se suele supeditar su desarrollo. Esto explica la actual concepción centrada en el hardware de este tipo de aplicaciones que de hecho, en muchos casos, suelen implementarse como sistemas mixtos con una parte hardware (código que se ejecuta sobre una plataforma de propósito específico) y otra parte de software (código que se ejecuta sobre un procesador convencional).
- (3) Además, el software que gestiona estos sistemas suele ser de pequeña o mediana envergadura, por lo que en la mayoría de los casos la aplicación de metodologías formales para su desarrollo suele considerarse un proceso más engorroso que útil.
- (4) Por último, y directamente relacionado con lo expuesto en el punto anterior, este tipo de aplicaciones las suelen implementar ingenieros de sistemas, expertos en programación a bajo nivel de dispositivos hardware, pero poco familiarizados con el uso de notaciones y metodologías semiformales de desarrollo de software, y por lo general, bastante reticentes a incorporarlas.

De acuerdo con todo lo anterior cabe preguntarse cómo y en qué medida puede resultar útil abordar la construcción de este tipo de sistemas mixtos Hardware/Software (HW/SW) desde una nueva óptica que permita mejorar su grado de flexibilidad y reutilización, teniendo en cuenta las restricciones que suelen guiar su construcción (eficiencia, bajo coste, alta fiabilidad, espacio y peso reducidos, corto tiempo de desarrollo, etc.).

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

Los procesos de análisis, definición y gestión de los requisitos resultan vitales a la hora de abordar, con una cierta garantía de éxito, el diseño y la implementación de cualquier producto software, más aún hoy en día cuando la complejidad de las aplicaciones es tan elevada. En los últimos años se han sucedido grandes cambios en el campo de la Ingeniería de Requisitos (ver cuadro de definición), siendo actualmente una de las disciplinas más activas dentro del campo de la Ingeniería del Software.

La ingeniería de requisitos estudia los servicios que el cliente demanda de un sistema y las restricciones bajo las que éste debe desarrollarse y operar [Sommerville 04].

La ingeniería de requisitos comprende el conjunto de actividades que se llevan a cabo a lo largo de todo el ciclo de vida del software y que buscan identificar las necesidades del cliente, documentarlas, verificar que son consistentes entre sí y analizarlas tratando de encontrar nuevos requisitos, así como el conjunto de procesos que dan soporte a estas actividades y los mecanismos de trazabilidad que permiten realizar su seguimiento [DoD 91].

Existen distintas formas de catalogar los requisitos asociados al software, aunque quizá una de las más intuitivas y ampliamente aceptadas sea la de Ian Sommerville quien propone la siguiente clasificación [Sommerville 04]:


- *Requisitos funcionales*: especifican los servicios que debe proporcionar el sistema, cuál debe ser su reacción ante las distintas entradas y cómo debe comportarse frente a las distintas situaciones que se puedan plantear.
- *Requisitos no funcionales*: establecen las restricciones bajo las que debe desarrollarse (legibilidad, documentación, etc.) y operar el sistema (tiempo de respuesta, recursos consumidos, obligatoriedad del uso de un determinado lenguaje de programación / sistema operativo / herramienta de desarrollo por motivos de compatibilidad, portabilidad o simplemente para ajustarse a la política de la empresa, etc.).
- *Requisitos propios del dominio*: características del sistema específicas del dominio de aplicación al que éste pertenece. Pueden ser nuevos requisitos funcionales, restricciones sobre requisitos previamente definidos, o especificaciones de cómo llevar a cabo ciertas operaciones.

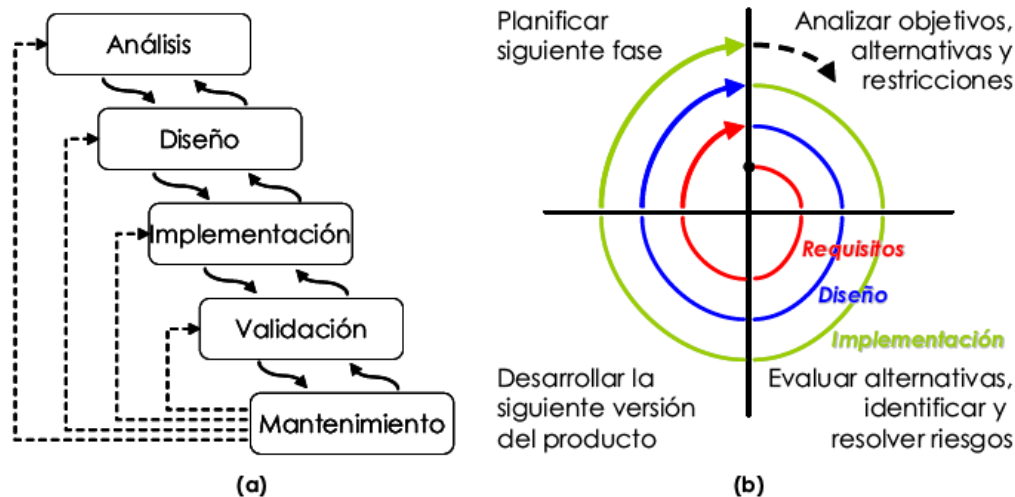
El cumplimiento de los requisitos funcionales de un determinado producto software está directamente relacionado con su eficacia.

Un sistema es eficaz o no (cumple o no con lo que se espera de él) y por lo tanto no cabe decir que un sistema es más eficaz que otro. La eficacia es un requisito ineludible y universal ya que todo sistema debe al menos cumplir con la finalidad para la que ha sido concebido.

En nuestro proceso de desarrollo es requerido seleccionar un ciclo de vida que se adecue al proyecto en cuestión de acuerdo a sus características más relevantes. A continuación se muestra una posible clasificación atendiendo al tipo de proceso que siguen sus ciclos de vida:

- *Metodologías descendentes (top-down)* en las que se busca manejar la creciente complejidad del software a través de abstracciones que permitan razonar acerca de la estructura y la lógica de los sistemas que se desea implementar. A partir de estas abstracciones se elaboran las soluciones que se van refinando y concretando sucesivamente hasta llegar a la implementación final. Este tipo de metodologías dan lugar a lo que se denomina desarrollo "para reutilización", entre las que encontramos:
  - *Metodologías en cascada (waterfall)*: Su ciclo de vida consta de una serie de etapas entre las que tradicionalmente se incluyen las siguientes: análisis, diseño, implementación, integración y mantenimiento, cada una de las cuales requiere un esfuerzo ingente de documentación. Estas etapas se realizan secuencialmente y de modo que hasta que no se completa una etapa no se pasa a la siguiente, aunque es posible volver atrás en el proceso para realizar algunas modificaciones (retroalimentación). Este tipo de metodologías suelen contemplar la posibilidad de validar los resultados intermedios utilizando prototipos ya sean evolutivos o desechables.
  - *Metodologías en espiral [Boehm 88]*: Se trata de un modelo inspirado en el anterior en el que se repiten todas las etapas del ciclo de vida tradicional del software de manera iterativa. Al finalizar cada etapa se produce un prototipo que, en cada iteración, se aproximará más a la solución final. Las metodologías en espiral suelen considerarse más flexibles aunque también más costosas de seguir que las anteriores.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
	APUNTE DE TEMAS DIVERSOS	



Ciclo de vida (a) en cascada y (b) en espiral.

- *Metodologías ascendentes (bottom-up)* en las que se elaboran las soluciones partiendo de pequeñas "piezas" de software, ya sean éstas de fabricación propia o adquiridas a terceros, que se van agregando de manera incremental hasta alcanzar la solución final, siguiendo un proceso similar al de una típica cadena de montaje industrial. Este tipo de metodologías dan lugar a lo que se denomina desarrollo "con reutilización", siendo el desarrollo de software basado en componentes (DSBC) uno de los máximos exponentes de este enfoque.

#### Paradigmas actuales de desarrollo de software

En el contexto de las ciencias de la computación un paradigma es una forma de hacer, esto es, el enfoque que se adopta para resolver los problemas de desarrollo del software. No debe confundirse el término paradigma con el de metodología; de hecho una metodología puede basarse en uno o varios paradigmas y distintas metodologías pueden sustentarse en el/los mismo/s paradigma/s.


Entre los paradigmas de desarrollo de software que actualmente cuentan con un mayor grado de aceptación cabe destacar algunos como los paradigmas de desarrollo de software orientados a objetos (OOSD, Object-Oriented Software Development), a aspectos (AOSD, Aspect-Oriented Software Development), los basados en componentes (CBSD, Component-Based Software Development), los basados en arquitecturas dirigidas por modelos (MDA, Model-Driven Architectures), los basados en líneas de productos (SPL, Software Product Lines) y el de Programación Generativa (PG).

#### **Desarrollo de software basado en componentes (DSBC)**

Desde sus comienzos, la Ingeniería del Software ha perseguido la idea de construir sistemas a partir de piezas ya existentes siguiendo el modelo de ensamblaje que utilizan muchas factorías industriales [Greenfield 04]. Para equiparar el proceso industrial con el de desarrollo de software sería necesario que los productos que han de ensamblarse estuvieran perfectamente especificados siguiendo unos ciertos estándares, tal y como ocurre en la industria. Sin embargo, los intentos de crear "factorías software" han fracasado reiteradamente por varios motivos. En primer lugar, el software en general no es un producto fácil de estandarizar, ya que existe una enorme variedad de metodologías de desarrollo, notaciones, lenguajes de programación, etc. Aunque la comunidad científica ha realizado varias propuestas tratando de definir estándares, por lo general éstos no han encontrado un apoyo decidido por parte de la industria; quizá la notación UML (Unified Modeling Language), estandarizada por el OMG (Object Management Group) sea una de las pocas excepciones a esta afirmación. Esta ausencia de estándares favorece que el software que se construye sea de naturaleza muy diversa de modo que los distintos productos obtenidos resultan, por lo general, difíciles de integrar entre sí.

Por otra parte, la creciente demanda de software cada vez más complejo obliga a las empresas a reducir a toda costa los precios y el tiempo de salida al mercado de sus productos a fin de mantener su competitividad. Para conseguir estos objetivos resulta imprescindible reutilizar al máximo cualquier software ya desarrollado, bien por la propia empresa en proyectos previos, bien el disponible de fuentes externas.

En este contexto, K. Wallnau [Wallnau 01] plantea dos visiones extremas acerca del proceso de desarrollo de software: la primera y quizá más cercana a la realidad actual es bastante "pesimista", mientras que la segunda, mucho más "optimista", muestra los ideales a los que se debería tender para aproximarnos al concepto de "factoría software".

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

La "visión pesimista" surge como consecuencia de considerar que:

- (1) El software está compuesto de elementos (componentes) vistos como cajas negras que ocultan muchas de sus propiedades haciendo difícil acceder a la información que contienen y a su comportamiento interno.
- (2) Cada fabricante utiliza su propia notación para definir sus esquemas de diseño y por lo tanto éstos no suelen ajustar a ningún estándar.
- (3) Las herramientas disponibles sólo permiten el desarrollo de productos para una determinada plataforma, limitando los lenguajes de programación a aquellos que son compatibles con ella. Por ejemplo, el entorno .NET de Microsoft® permite integrar componentes implementados utilizando varias de las herramientas de programación existentes para esta plataforma (Visual Basic, Visual C++, etc.), si bien estos componentes no son directamente compatibles con los desarrollados utilizando CORBA, EJB, etc.
- (4) Los componentes no suelen estar bien documentados lo que dificulta enormemente tanto su localización como su validación y, en consecuencia, su reutilización.

En contraposición, la "visión optimista" presupone que se cuenta con:

- (1) Elementos software (componentes) definidos mediante interfaces claras y correctamente especificadas a modo de contratos.
- (2) Esquemas de diseño estándar en los que los anteriores componentes encajan de manera sencilla.
- (3) Repositorios de componentes ya implementados y listos para ser reutilizados y en los que es sencillo encontrar el componente que mejor se ajusta a las necesidades de un determinado proyecto.
- (4) Herramientas de desarrollo estandarizadas que estén orientadas a la fabricación de soluciones mediante técnicas de composición y que permitan la integración de componentes heterogéneos (escritos utilizando distintos lenguajes de programación, para distintas plataformas, etc.).

#### *Definición de Componente Software*

Etimológicamente, el término "componente" procede de la palabra latina *cumponere* que significa "poner junto" (*cum* "junto" y *ponere* "poner").

Como puede fácilmente deducirse de la consulta de la literatura especializada, existe diversidad de opiniones sobre lo que puede considerarse o no un Componente Software (CS). Esta disparidad de opiniones ha dado lugar a numerosas definiciones de este término, algunas de las cuales quedan recogidas a continuación.

"Un componente es una unidad binaria de composición que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio" [Szyperski 98]


"Un componente es una implementación opaca de una determinada funcionalidad (caja negra) que puede ensamblarse con otros componentes siguiendo las reglas establecidas por el modelo de componentes adoptado" [Bachmann 00].

"Un componente es una implementación que: (a) realiza un conjunto de funciones relacionadas, (b) puede ser independientemente desarrollado, entregado e instalado, (c) tiene un conjunto de interfaces para los servicios proporcionados y otro para los servicios requeridos, (d) permite tener acceso a los datos y al comportamiento sólo a través de sus interfaces, (e) opcionalmente admite una configuración controlada." [IBM-WEBSPHERE].

Como puede observarse en las definiciones anteriores, todas ellas coinciden en el hecho de que los CS son unidades de implementación concebidas para ser reutilizadas, pudiendo ensamblarse con otros CS que, aun no siendo necesariamente de la misma procedencia, deben ser compatibles entre sí. La compatibilidad entre los CS requiere que éstos sean conformes con un determinado modelo de componentes que establecerá bajo qué condiciones pueden ensamblarse. El proceso de ensamblaje requiere que cada componente defina en su interfaz tanto los servicios que demanda (requisitos), como aquellos que ofrece al resto de componentes con los que interactúa.

El Desarrollo de Software Basado en Componentes (DSBC) puede considerarse una evolución del paradigma Orientado a Objetos (OO). La diferencia entre ambos enfoques parece no estar muy clara y a veces se tiende a confundir el concepto de componente con el de objetos. He aquí algunas claves para entender las diferencias entre ambos conceptos:

- (1) La interfaz de un objeto exhibe sólo los servicios que éste proporciona, mientras que la interfaz de un componente debe incluir además los servicios requeridos para su correcto funcionamiento.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

- (2) Existen diversas formas de interactuar con un componente, mientras que la única forma de interactuar con un objeto es mediante la invocación de alguno de sus métodos públicos (paso de mensajes).
- (3) Un componente puede contener uno o más objetos, clases, rutinas, e incluso otros componentes.
- (4) Los componentes de una misma aplicación pueden estar escritos en distintos lenguajes de programación, incluyendo lenguajes orientados a objetos.
- (5) Los componentes suelen estar empaquetados de manera más robusta que los objetos.

Según esto, un componente puede ser desde una subrutina de una librería matemática hasta una clase Java, un paquete Ada, un objeto COM, un JavaBean, o incluso una aplicación independiente con la que es posible interactuar desde otra a través de una determinada interfaz.

Es posible realizar distintas clasificaciones de los componentes atendiendo a su funcionalidad, visibilidad (caja negra, gris o blanca), coste (freeware, shareware o fullprice), modos de interacción, ámbito de aplicación, granularidad, etc. En particular, la granularidad y el ámbito de aplicación de un componente determinan en gran medida su utilidad, aplicabilidad y rendimiento. Así, por lo general, los componentes de grano grueso y de propósito general resultan de utilidad para la construcción de una amplia variedad de aplicaciones y consiguen un ahorro sustancial en los tiempos de desarrollo, dado el alto grado de reutilización que supone su incorporación. Sin embargo, este tipo de componentes suele proporcionar una funcionalidad a veces no requerida que puede complicar su integración y mermar la eficiencia global del sistema.

Por el contrario, cuanto más pequeño y específico es un componente mejor se ajustará a los requisitos que de él se demandan y más sencilla y eficiente resultará su integración en las distintas aplicaciones. Ahora bien, dada su especificidad, serán pocos los sistemas que puedan reutilizar este tipo de componentes así como tampoco supondrán un gran ahorro en el tiempo de desarrollo dado el poco código que se reutiliza.

#### *Desarrollo de Software basado en Componentes: Etapas*

Tradicionalmente los ingenieros del software han seguido un enfoque descendente (top-down) para el desarrollo de sus sistemas [Page-Jones 88]. Sin embargo, el DSBC apuesta por un enfoque ascendente (bottom-up) en el que los productos se construyen mediante el ensamblaje e integración de componentes software preexistentes.

A diferencia de las metodologías top-down cuyo ciclo de vida comprende las etapas de análisis, diseño e implementación, el DSBC divide el proceso en las siguientes fases [Brown 99] [Iribarne 03]:


- (1) Selección y evaluación de los componentes software que deberán satisfacer las necesidades del usuario y ajustarse al esquema de diseño de la aplicación.
- (2) Adaptación de los componentes cuando sea necesario.
- (3) Ensamblaje de los componentes como parte de la solución final.
- (4) Evolución del sistema si el usuario lo requiere.

En el siguiente cuadro se muestra un resumen comparativo de las fases comprendidas en el ciclo de vida del software siguiendo el enfoque tradicional y el basado en componentes.

Ciclo de vida	Etapas del desarrollo del software			
Clásico (top-down)	Análisis	Diseño	Implementación	Mantenimiento
DSBC (bottom-up)	Selección y evaluación	Adaptación	Ensamblaje	Evolución

A continuación se comentan brevemente cada una de las etapas comprendidas en el ciclo de vida del software según el enfoque del DSBC:

- (1) *Selección y evaluación de los componentes:* Esta etapa permite determinar qué componentes, de entre los disponibles, se ajustan mejor a las necesidades de una determinada aplicación. Para ello es necesario en primer lugar encontrarlos y, en segundo, evaluarlos a fin de comprobar que efectivamente cumplen con lo que se espera de ellos. Encontrar el componente adecuado requiere establecer con precisión los criterios de búsqueda. Entre éstos, suelen incluirse los relativos a la funcionalidad (servicios que debe proporcionar) aunque también es común incluir otros relacionados con el coste, el fabricante, compatibilidad con una determinada plataforma, etc. Esta fase puede resultar tediosa debido a que la abundante información disponible no siempre está correctamente clasificada. Además, los resultados de la búsqueda pueden resultar difíciles de cuantificar [Iribarne 02].

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

La fase de evaluación tampoco resulta sencilla, si bien existen algunas técnicas relativamente maduras que pueden aplicarse al efecto [ISO/IEC-9126 91]. Estas técnicas de evaluación se basan en encuestas realizadas a los clientes y en la construcción de prototipos con los que validar, de manera efectiva, el comportamiento de los componentes seleccionados.

- (2) *Adaptación de los componentes:* En el momento de crear nuevos componentes, los desarrolladores hacen ciertos supuestos sobre los sistemas en los que éstos se integrarán. Sin embargo, cuando estas suposiciones no se ajustan a las necesidades del cliente, como suele ocurrir, éste deberá realizar ciertas adaptaciones de forma que los componentes puedan encajar en el sistema. El procedimiento de adaptación depende de la accesibilidad del componente. Cuando éste se adquiere como una "caja negra" habrá que adaptar el sistema para permitir que encaje el componente, mientras que si el componente seleccionado es accesible, ya sea porque es de fabricación propia o porque se ha adquirido con ciertos privilegios para poder modificarlo, puede resultar más conveniente adaptar el componente al sistema.  
Las adaptaciones que puede requerir un componente pueden ser de tres tipos: para limitar su funcionalidad, para extenderla, o para hacerlo compatible (con otros componentes, con la plataforma, etc.). Uno de los mecanismos de adaptación más utilizados consiste en construir un componente de tipo wrapper que envuelva al que se quiere adaptar de forma que lo limite, extienda, o modifique como convenga.
- (3) *Ensamblaje o integración de los componentes:* El proceso de ensamblaje requiere de la existencia de un modelo de componentes, entendido como un conjunto de reglas que establecen qué componentes se pueden conectar con qué otros y cómo se comunican éstos entre sí. Dado que los componentes se comunican a través de sus interfaces, será necesario utilizar un lenguaje de definición de interfaces (IDL) que permita definirlos de una manera precisa. Además, cuando se precise conectar dos componentes situados en distintas localizaciones, será necesario proporcionar los mecanismos adecuados para su comunicación remota. Todo ello requiere además contar con una infraestructura o middleware que permita realizar el proceso de ensamblaje de los componentes. Entre los entornos middleware más utilizados cabe destacar algunos basados en tecnologías ORB (Object Request Broker) como .NET, EJB o CCM.
- (4) *Evolución del sistema:* A priori pudiera parecer que los sistemas contruidos a partir de componentes deberían poder evolucionar de manera sencilla como ocurre en los procesos industriales de montaje en los que la sustitución de piezas o su actualización con otras más modernas resulta una tarea casi trivial. Sin embargo, cuando se trata de productos software, la sustitución de un componente por otro suele ser una tarea compleja y tediosa ya que además de que el nuevo componente debe someterse a un proceso minucioso de validación para comprobar que se ajusta a lo que se espera de él, puede ser necesario plantear una modificación drástica de la estructura del sistema para permitir su integración. Adicionalmente, debe tenerse en cuenta que la evolución de un sistema basado en componentes depende en gran medida de cómo los fabricantes decidan evolucionar dichos componentes. De hecho, puede ocurrir que el fabricante no esté interesado en desarrollar una nueva versión del componente para cubrir la demanda de un cliente puntual o incluso que decida crear una nueva versión del producto dejando sin soporte la versión previa, lo que obligará al cliente a decidir si cambia de versión o de proveedor.

### Desarrollo basado en Líneas de Productos Software (LPS)

Hoy en día, la complejidad y el tamaño de los productos software está creciendo rápidamente y los clientes demandan productos cada vez de más calidad y más ajustados a sus necesidades. Para cubrir esta demanda la mayoría de las organizaciones productoras de software desarrollan y mantienen más de un producto simultáneamente. Esto es aplicable tanto a las empresas que desarrollan sistemas a medida para clientes individuales, como para aquellas que desarrollan productos para un mercado más amplio. Incluso las organizaciones que dicen vender un único producto invierten gran parte de su presupuesto en mejorarlo o adaptarlo a las necesidades de cada uno de sus clientes individuales, por lo que finalmente deben dar soporte a un conjunto de variantes de su producto [Muthig 02].

Por lo general, las empresas productoras de software se especializan en un determinado rango o dominio de aplicaciones por lo que sus productos suelen tener muchas características en común. A este conjunto de aplicaciones similares pertenecientes a un mismo dominio de aplicación se le suele denominar familia de productos. A veces, erróneamente, se asimila este concepto al de línea de productos, aunque como queda recogido en las siguientes definiciones existen ciertos matices que permiten diferenciar ambos conceptos.

"Línea de productos: conjunto de productos que comparten una serie de características comunes y que responden a las necesidades particulares de un determinado sector del mercado" [Whitney 96].

Por ejemplo, una línea de productos de higiene puede incluir, entre otros, geles de ducha, espumas de afeitado, productos de maquillaje o perfumes, etc.

"Familia de productos: conjunto de productos pertenecientes a un mismo dominio de aplicación que pueden construirse a partir de un conjunto de elementos básicos comunes." [Whitney 96].



	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

Siguiendo con el ejemplo anterior, podría crearse una familia de geles de ducha que incluyera distintas variedades, por ejemplo, para pieles sensibles, para mujeres, hombres, bebés, etc.

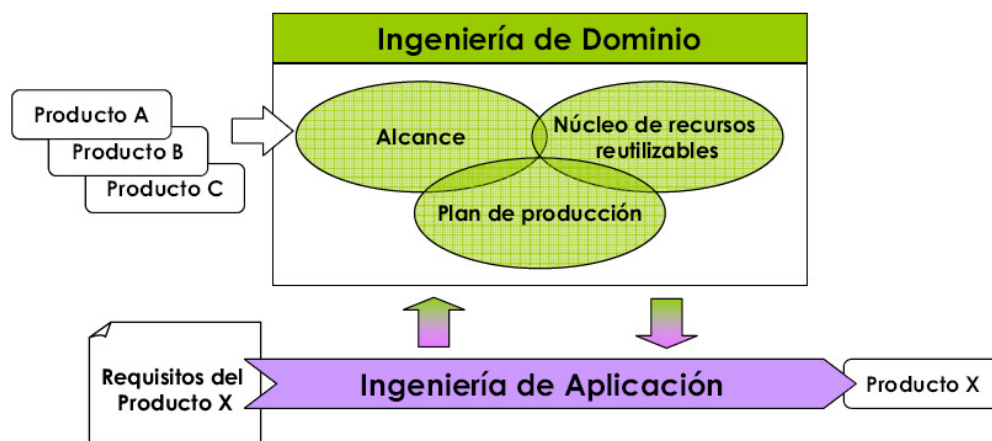
No obstante, una línea de productos puede diseñarse para desarrollar productos de una misma familia. Esto proporciona la ventaja de poder construir los distintos productos de la línea utilizando un conjunto de recursos comunes y válidos para todos ellos [Eisenecker 03]. Así, siguiendo con los ejemplos anteriores, tendría sentido hablar de una línea de productos de la familia "gel de ducha", de forma que todos los geles podrían fabricarse a partir de la misma base jabonosa o utilizando los mismos envases, diferenciándose unos de otros en el perfume añadido al jabón o en el color del etiquetado.

En el caso de las líneas de productos software (LPS) éstas siempre se utilizan para desarrollar sistemas de una misma familia o dominio de aplicación, tal y como queda recogido en la siguiente definición, en la que se ha resaltado en negrita las peculiaridades de estas frente a las utilizadas para desarrollar otros tipos de productos.

"Una línea de productos software es un conjunto de sistemas **intensivamente software** que comparten una serie de características comunes y que satisfacen las necesidades de un segmento particular del mercado, **pudiendo desarrollarse a partir de un conjunto de recursos comunes de acuerdo con un plan de producción previamente establecido.**" [Clements 02].

Las líneas de productos software no deben confundirse con: (1) aplicaciones que emplean, de manera casual y esporádica, mecanismos de reutilización de grano fino, (2) sistemas construidos reutilizando porciones de código extraídas a conveniencia de otros sistemas, (3) aplicaciones construidas a partir de una arquitectura genérica o utilizando una metodología de desarrollo basada en componentes y (4) un conjunto de versiones desarrolladas a partir de un mismo producto [Clements 02]. Así, si bien el desarrollo de software basado en líneas de productos puede considerarse una suma, entre otros, de varios de los conceptos aquí mencionados (reutilización, arquitectura software, componentes, etc.), estos no se utilizan de manera casual, opcional, parcial o interesada sino que, muy al contrario, todos ellos resultan esenciales y de su correcta integración, siempre previamente planificada, dependerá el éxito final de la LPS.

El desarrollo de software basado en líneas de productos requiere llevar a cabo dos actividades, la primera de ellas relacionada con lo que se ha dado en llamar ingeniería del dominio y la segunda relacionada con la denominada ingeniería de las aplicaciones.



Esquema de los procesos de ingeniería de dominio y de aplicaciones.

La ingeniería de dominio está inspirada en el concepto de "implementar para reutilizar". Esta disciplina se encarga de analizar las características comunes (commonalities) y diferenciales (variabilities) de las aplicaciones para, a partir de ellas, construir la infraestructura de la LPS consistente en un repositorio de artefactos reutilizables. Entre estos cabe destacar la especificación de los requisitos de las aplicaciones, su arquitectura software y el conjunto de componentes a partir de los que se construirán los distintos productos de la línea. El manejo de la variabilidad propia de las LPS requiere de tecnologías que den soporte a la identificación, creación y adaptación de este núcleo de artefactos reutilizables.

A diferencia de la anterior, la ingeniería de aplicaciones se inspira en el concepto de "implementación con reutilización" y es la responsable de la construcción de los distintos productos a partir de los artefactos desarrollados por la primera.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

Cada uno de estos productos se construirá (1) seleccionando aquellos recursos que resulten de utilidad, adaptándolos como convenga mediante los mecanismos de variación previamente establecidos (parametrización, especialización mediante herencia, etc.), (2) desarrollando nuevos recursos cuando sea necesario, y (3) ensamblándolos de forma que encajen en la arquitectura del sistema siguiendo las reglas establecidas en el plan de producción. De esta manera, construir un nuevo producto o sistema se convierte más en una tarea de ensamblaje o integración que de programación.

Con relación a la ingeniería de dominio es necesario hacer hincapié en las tres etapas que ésta comprende y que se comentan a continuación: (1) la definición del alcance de la LPS, (2) el desarrollo del núcleo de recursos reutilizables, y (3) la especificación del plan de producción.

#### *Definición del alcance*

El alcance de una LPS define qué aplicaciones pueden derivarse a partir de ella, especificando qué características tienen en común y cuáles las diferencian. De este modo, para cada aplicación incluida en el alcance de la LPS, se deberá especificar qué características (funcionalidades) debe ésta incorporar de manera obligatoria, opcional o alternativa (característica obligatoria seleccionada de entre un conjunto de opciones válidas).

Por lo general el alcance de una LPS suele presentarse utilizando una matriz bidimensional de productos vs. características a la que suele adjuntarse un diagrama de variabilidad en el que quedan recogidas todas las características identificadas así como las relaciones existentes entre ellas [Savolainen 01].

La correcta definición del alcance y, en particular, el modelado sistemático de la variabilidad resultan actividades esenciales para el éxito de cualquier LPS ya que de ellas depende su capacidad para anticipar y poder reaccionar, de manera adecuada, ante los posibles cambios que pudieran introducirse en el futuro [van Gurp 01].

#### *Desarrollo del núcleo de recursos reutilizables (core assets)*

El núcleo de recursos esenciales es la base de toda LPS. Estos recursos incluyen, entre otros, la arquitectura, los componentes software y hardware, los modelos de dominio, los requisitos, la documentación y las especificaciones que se van generando, los modelos de rendimiento, la planificación temporal de las actividades, el presupuesto, los casos de prueba, etc. Todos estos recursos ofrecen distintos niveles de reutilización durante el desarrollo del producto y quizá, de entre ellos, los más importantes sean la arquitectura y los componentes software. Puesto que a éstos últimos ya se ha dedicado uno de los apartados anteriores, parece conveniente realizar en este punto algunas consideraciones sobre las arquitecturas software.


- Arquitectura Software (AS): En la literatura es posible encontrar numerosas definiciones para este término entre las que, a continuación, se muestran las propuestas por Garlan y Bass:

"La arquitectura software de un sistema establece la estructura organizativa de sus componentes incluyendo cómo éstos se relacionan entre sí, así como los principios y líneas maestras que gobiernan su diseño y evolución" [Garlan 95].

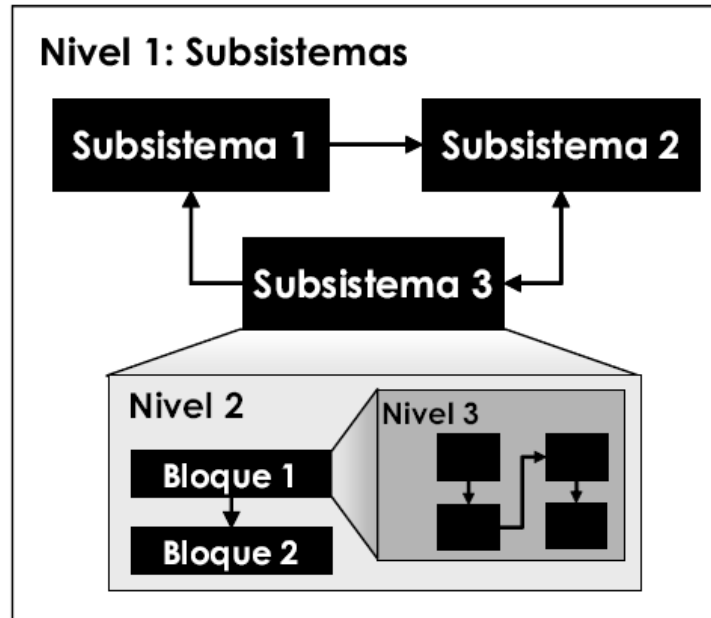
"La arquitectura de un sistema consiste en una vista abstracta que elimina los detalles de implementación y se concentra en el comportamiento e interconexión de los componentes vistos como cajas negras." [Bass 98].

En las definiciones anteriores, el término componente no debe entenderse como la "unidad binaria de composición" a la que se hizo referencia al hablar del desarrollo de software basado en componentes. En el caso de las AS, un componente es una entidad abstracta que sirve para modelar el sistema visto como un todo, los subsistemas que lo componen, los bloques funcionales en los que éstos pueden dividirse, etc. Así, la especificación de la AS posee distintos niveles de abstracción tal y como se muestra en la figura.



	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

## Nivel 0: Sistema



### Distintos niveles de abstracción de una AS.

A la hora de diseñar una AS es posible acudir a ciertos patrones o estilos arquitectónicos previamente establecidos [Gamma 95]. Por ejemplo, el patrón denominado "pipes and filters" modela las aplicaciones utilizando únicamente dos tipos de elementos que se encargan, unos de procesar la información (filters) y otros de transmitirla (pipes). El patrón establece además que sólo es posible conectar entre sí elementos de distinto tipo. En particular, existen algunos patrones de diseño específicos para definir AS destinadas a la construcción de productos de un determinado dominio, también denominadas arquitecturas específicas de dominio o arquitecturas de referencia. A esta categoría pertenecen las AS de las LPS.

A la hora de especificar la AS de una LPS se deberá recoger la variabilidad inherente a los distintos productos que se pretenden diseñar, esto es, la AS deberá capturar las similitudes que subyacen en la estructura y la lógica de todos ellos y, simultáneamente, ser lo suficientemente flexible como para incluir sus peculiaridades. Además, deberá evolucionar de manera sencilla a partir de los requisitos y ser independiente de la plataforma sobre la que se desarrollen las implementaciones particulares.

### Plan de producción de una LPS

El plan de producción describe cómo los distintos productos se construyen a partir del núcleo de recursos definido en la fase anterior. Para ello, cada uno de estos recursos deberá conocer cuál es su aportación al proceso de desarrollo (planes de producción parciales) y, adicionalmente, deberá definirse un plan de producción global en el que se establezca qué recursos de entre todos los disponibles son necesarios y cómo encajarlos para conseguir el producto final deseado.

Cada uno de los elementos que forman parte del plan de producción de una LPS, esto es, el conjunto de planes parciales y el plan global, se presentarán en un formato adecuado al tipo de usuario al que vayan destinados (analistas, diseñadores, integradores, clientes, etc.). Así, los planes de producción pueden variar desde especificaciones técnicas detalladas sobre el modelo de interoperabilidad de los componentes software disponibles, hasta documentos más informales que describan cómo utilizar los casos de prueba para validar cada producto.

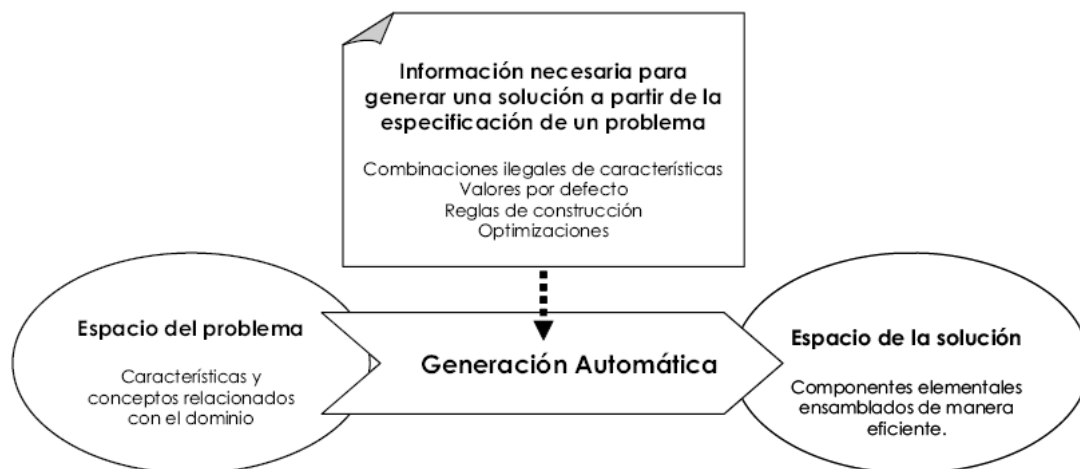
### Programación generativa

La Programación Generativa (PG) es un paradigma de desarrollo de software de reciente aparición que, como en el caso de las LPS, trata de desarrollar productos software pertenecientes a una misma familia. Para ello, este enfoque persigue que a partir de la especificación de unos ciertos requisitos, contemplados como opciones de configuración, y utilizando una serie de componentes de implementación elementales y reutilizables, se pueda derivar automáticamente un producto optimizado, ya sea éste intermedio o final [Czarnecki 00].

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

Según esta definición, el uso de técnicas de PG consigue, una vez que el programador de aplicaciones ha establecido en términos abstractos (utilizando un lenguaje textual o gráfico) las características del producto que desea, que sea un generador el que se encargue de construirlo automáticamente, enlazando adecuadamente los componentes necesarios. Para conseguir este objetivo tanto tiempo anhelado por la Ingeniería del Software es necesario (1) diseñar los componentes de implementación para que encajen en la arquitectura, (2) modelar el proceso de generación o traducción de los requisitos abstractos (espacio del problema) a un conjunto específico de componentes (espacio de la solución), y (3) conseguir que esta traducción la implemente automáticamente un generador.

Tal y como se muestra en la figura, pasar del espacio del problema al de la solución requiere dotar al generador de cierta información relativa a los recursos con los que cuenta y a las reglas de composición que le permitirán combinarlos de manera que consiga, de entre los resultados posibles, el mejor. Esta información deberá estar especificada de la manera más precisa y formal que sea posible, de modo que el generador pueda procesarla adecuadamente [Czarnecki 00]. La ventaja de utilizar especificaciones formales es que a partir de ellas el generador puede, aplicando una serie de reglas sencillas, llegar a una solución optimizada, mientras que esta misma tarea resultaría enormemente compleja y costosa para un programador. Como contrapartida debe considerarse el esfuerzo que supone formalizar la especificación del problema, lo que generalmente conlleva el uso de lenguajes formales como Maude [MAUDE] [Lucas 04] derivado de OBJ u OOOE [Alencar 91] [Nicolás 96] basado en la notación algebraica Z.




**Elementos de la PG.**

Desde cierto punto de vista, las herramientas de PG pueden considerarse compiladores de modelos o herramientas de metaprogramación. De manera similar a como en lingüística se utiliza un metalenguaje para discutir, describir o analizar otro lenguaje, un metaprograma puede definirse como aquel que se utiliza para generar otros programas. Desde este punto de vista, la PG puede considerarse una técnica de metaprogramación [Stuikys 02].

La generación automática de software no es un concepto nuevo ni exclusivo de la PG. Por ejemplo, los compiladores generan automáticamente código ejecutable a partir de código fuente, así como algunas herramientas CASE (Computer Aided Software Engineering) son capaces de generar automáticamente esqueletos de aplicaciones a partir de modelos UML.

Del mismo modo, otros de los conceptos que aparecen íntimamente ligados a la PG tampoco son nuevos, como por ejemplo el desarrollo de software basado en objetos, componentes y aspectos, la ingeniería de dominio, la programación intencional (IP, Intentional Programming), etc. La principal ventaja de la PG reside en que este paradigma de desarrollo de software, por primera vez, lejos de intentar diferenciarse de sus predecesores busca incorporar todo lo que cada uno de ellos tiene de positivo. De entre todos ellos, quizá los que más aportan a la PG sean el paradigma de desarrollo de software basado en LPS y el de programación automática [Balzer 85]; de hecho puede considerarse a la PG como una herramienta que permite, a partir de la especificación de una LPS, derivar distintas aplicaciones pertenecientes a un mismo dominio (el recogido en la LPS) de manera (semi)automática.

Lo reciente de la PG determina que este paradigma aún no haya alcanzado un grado madurez comparable al de sus predecesores. Por otra parte, es preciso tener en cuenta que, como en el caso de las LPS, su aplicación está limitada al desarrollo de familias de productos, lo que restringe el número de usuarios potenciales y, por lo tanto, su nivel de implantación.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

### La reutilización

Como ya vimos, lo que hoy el mercado nos exige es lograr un producto que se ajuste a las necesidades del cliente en el menor tiempo, costo, y esfuerzo posible, por lo que tendemos hacia la reutilización sistemática de software.

Los obstáculos para la reutilización son múltiples. Un repositorio de componentes reutilizables debe satisfacer la mayor demanda posible asegurando, a la vez, la fácil localización de sus componentes. La búsqueda en un repositorio se simplifica si está convenientemente organizado o consta de pocos componentes. Normalmente, para maximizar la oferta de un repositorio minimizando la cantidad de sus componentes, se persigue la generalidad y ortogonalidad de los componentes. De manera que, parametrizándolos, satisfagan necesidades particulares y, combinándolos, produzcan nuevos componentes.

La parametrización de un componente se facilita si su grado de ocultación es de "caja negra". Así, se evita la necesidad de conocer los detalles de implementación de un componente para parametrizarlo. La ortogonalidad de los componentes aumenta al incrementar la cohesión de cada componente y disminuir el acoplamiento entre componentes. Por otro lado, para que se pueda automatizar la reutilización de componentes, es imprescindible que estén especificados formalmente.

Como se verá a continuación, a medida que crece la abstracción de los componentes, mayor es la dificultad de especificarlos formalmente con sencillez y concisión y, por tanto, menor es la automatización de su reutilización. Por ejemplo, la parametrización y combinación automática de componentes de código como macros, funciones, clases, etc., está muy difundida, mientras que la reutilización automática de diseño es muy inferior y, la de análisis, aún menor.

### **La reutilización en el análisis**

El principal objeto de reutilización en la fase de análisis son las especificaciones de requisitos. Habitualmente, las especificaciones son informales y se expresan en lenguaje natural, lo que dificulta su parametrización y disminuye su capacidad para combinarse y formar nuevas especificaciones.

Para paliar estas carencias, existen algunas propuestas de uso de distintos grados de formalización:

- El modelado semiformal, cuyo representante más difundido es UML (Unified Modeling Language), suele ser expresivo y relativamente conciso. Sin embargo, las posibilidades que ofrece para la parametrización son bastante limitadas.

K. Czarnecki critica la capacidad de parametrización de los diagramas de clases de UML por las siguientes razones:

- No permiten expresar restricciones sobre el tipo de un parámetro.
- No permiten expresar los métodos que obligatoriamente deberá tener una clase utilizada como parámetro real.
- Un parámetro se puede usar con diversos fines, por ejemplo, para expresar el tipo de una superclase (herencia parametrizada), el tipo de un objeto asociado, el tipo del argumento de un método, etc. En UML es muy difícil, si no imposible, representar explícitamente el propósito de un parámetro.

Con el lenguaje de restricciones OCL (Object Constraint Language) se puede mejorar la precisión de los diagramas UML. Sin embargo, algunos autores desaconsejan su uso al considerarlo excesivamente costoso.


- Los lenguajes de especificación formal, como Z ó TUG (Tree Unified with Grammar), están libres de ambigüedad y permiten la definición de especificaciones genéricas que pueden particularizarse y combinarse. Sin embargo, como indica M. Chechik, el uso de los métodos formales está poco extendido porque suelen considerarse costosos y difíciles de usar.

Por otro lado, exceptuando los requisitos de seguridad, de rendimiento y, en general, los requisitos no funcionales, con frecuencia los requisitos recuperados de un repositorio sólo son parcialmente reutilizables y necesitan algún tipo de adaptación.

Estos problemas explican el escaso nivel de reutilización logrado en la fase de análisis.

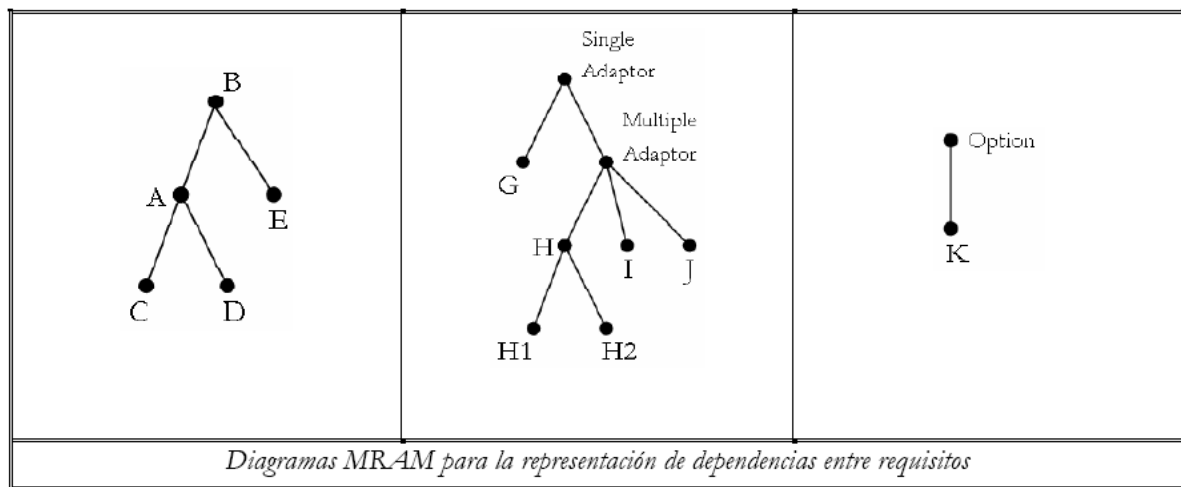
Muchas investigaciones asumen la naturaleza informal de los requisitos y se orientan a la mejora de la organización y las búsquedas en repositorios de especificaciones escritas en lenguaje natural o en algún lenguaje de modelado semiformal.

Para que los requisitos puedan reutilizarse independientemente, interesa que sean autocontenidos. Lamentablemente, esta situación no es frecuente. Si, por ejemplo, tres requisitos X, Y, Z hacen referencia a un cuarto requisito R, pueden tomarse varias decisiones:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

- Incluir R en cualquier especificación que reutilice a X, Y ó Z. De esta forma, X, Y, Z pasan a ser autocontenidos. Como inconveniente, la especificación pierde concisión e incluso puede ser redundante (si incluye a X e Y, contendrá dos veces a R).
- Para la reutilización de X, Y, Z se impone la reutilización explícita de R.

MRAM (Method for Requirements Authoring and Management) es una metodología que aborda la representación y gestión de las dependencias entre requisitos de forma arbórea. Los nodos de un árbol pueden ser requisitos o discriminantes de tres tipos: exclusión mutua (Single Adaptor), lista de alternativas (Multiple Adaptor) o rama opcional (Option). Por ejemplo, en la primera figura, el requisito B no tiene discriminantes. Si se reutiliza B, también se incluyen A, C, D y E. En la segunda figura se representa una exclusión mutua. Si se selecciona G, se descarta la otra rama, la cual permite la reutilización de H, I, J o cualquiera de sus combinaciones. Por último, la última figura representa la reutilización opcional del requisito K.




## La reutilización en el diseño

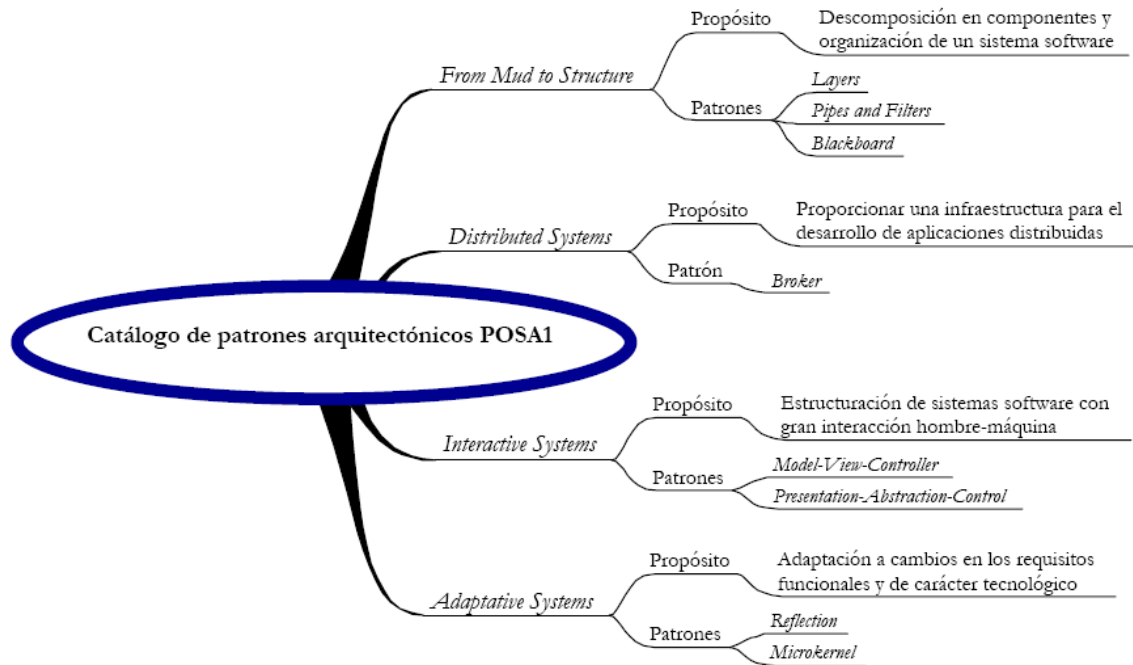
### Diseño arquitectónico

Los patrones o estilos arquitectónicos son soluciones de eficacia probada a problemas que aparecen con frecuencia en el diseño arquitectónico. El libro "Pattern-Oriented Software Architecture. A System of Patterns", conocido popularmente como POSA1, contribuyó de manera decisiva a la difusión de este tipo de patrones. En él, se enumeran los siguientes beneficios derivados del uso de patrones:

- Permiten la reutilización de soluciones arquitectónicas de calidad.
- Son de gran ayuda para controlar la complejidad de un diseño.
- Facilitan la documentación de diseños arquitectónicos.
- Proporcionan un vocabulario común que mejora la comunicación entre diseñadores.

POSA1 ofrece un catálogo de patrones arquitectónicos organizados, según su propósito, en cuatro categorías: From Mud to Structure, Distributed Systems, Interactive Systems y Adaptative Systems.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009	



*Catálogo de patrones arquitectónicos incluido en POSA1.*

En POSA1, los patrones se formulan informalmente según los siguientes apartados: nombre del patrón, sinónimos, ejemplo de aplicación, contexto o situaciones en las que el patrón puede utilizarse, propósito o tipo de problema que se resuelve, estrategia utilizada en el planteamiento de la solución, aspectos estructurales de la solución, comportamiento dinámico del patrón, implementación posible, variantes, usos conocidos, relación con otros patrones, beneficios y desventajas.


Posteriormente, se han publicado nuevos catálogos donde los patrones también se registran de manera informal. Ante la proliferación de patrones arquitectónicos, con el fin de automatizar la elección de los patrones más adecuados a problemas concretos, se han desarrollado sistemas de ayuda a la decisión e intentos de formalización del propósito de los patrones y de cómo su uso afecta a la calidad de una aplicación informática.

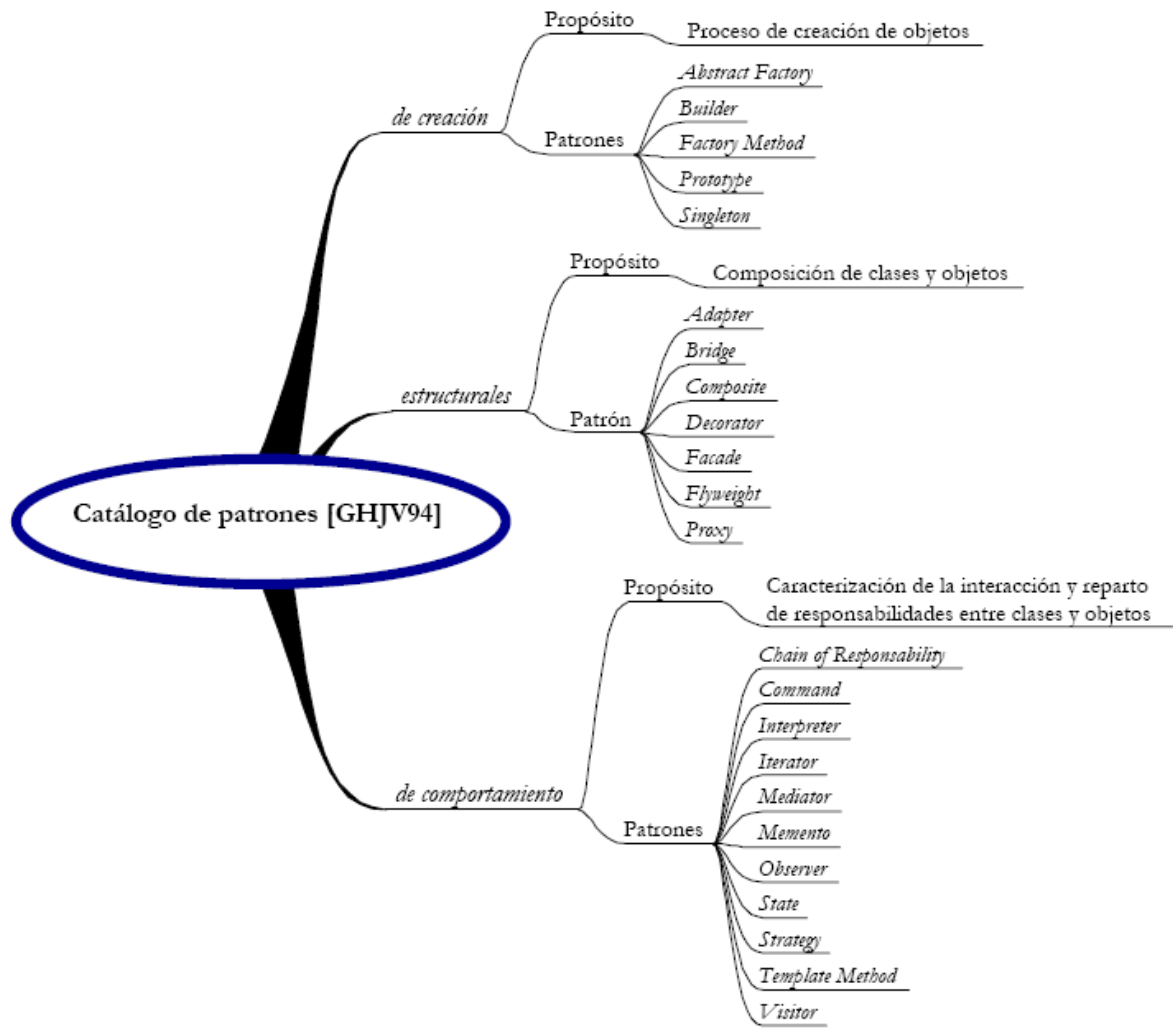
Reconocida la utilidad de los patrones arquitectónicos para la documentación y el mantenimiento de aplicaciones informáticas, se han producido investigaciones en ingeniería inversa para la extracción de patrones en código de lenguajes de programación.

#### *Patrones de diseño*

Análogos a los patrones arquitectónicos, los patrones de diseño son buenas soluciones de problemas que aparecen de forma recurrente en el diseño detallado. El catálogo "Design Patterns: Elements of Reusable Object-Oriented Software", cuyo germen fue la tesis doctoral de E. Gamma, marcó un hito en la reutilización del diseño. La extraordinaria difusión, ha llevado a que el término genérico "patrón de diseño" (que debería englobar a cualquier tipo de patrón "arquitectónico, de diseño detallado"), se utilice por defecto para hacer referencia al tipo de patrón orientado a objetos que recoge el citado catálogo.

Este catálogo contiene 23 patrones organizados según su propósito en tres categorías: de creación, estructurales y de comportamiento.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009



*Catálogo de patrones de diseño incluido en [GHJV94].*

Los patrones se especifican con la notación de modelado semiformal OMT (Object Modeling Technique) (diagramas de clases y diagramas de objetos para representar los aspectos estructurales, diagramas de interacción para expresar el comportamiento dinámico) acompañada de una descripción en lenguaje natural organizada según el nombre del patrón, propósito, sinónimos, motivación, aplicabilidad, participantes en los diagramas OMT, consecuencias de la utilización del patrón, consejos de implementación, código de ejemplo escrito en C++ y Smalltalk, usos conocidos y patrones relacionados.

Los patrones de diseño suscitan diversos problemas y controversias. A continuación, se presentan algunos de ellos.

- **Especificación:** La ambigüedad con que están expresados los patrones puede provocar su mal uso y limita su gestión automática mediante herramientas informáticas. Existen algunas propuestas de formalización basadas en la lógica, como el BPSL (Balanced Pattern Specification Language), donde los aspectos estructurales y dinámicos se especifican combinando la lógica de primer orden (FOL, First Order Logic) con acciones de lógica temporal (TLA, Temporal Logic Actions).
- **Ocultación:** Los patrones son unidades de reutilización de caja blanca, para usar el patrón no basta con conocer su propósito y "parámetros variables", sino que es imprescindible sumergirse en sus interioridades y adaptarlo manualmente a cada situación particular.

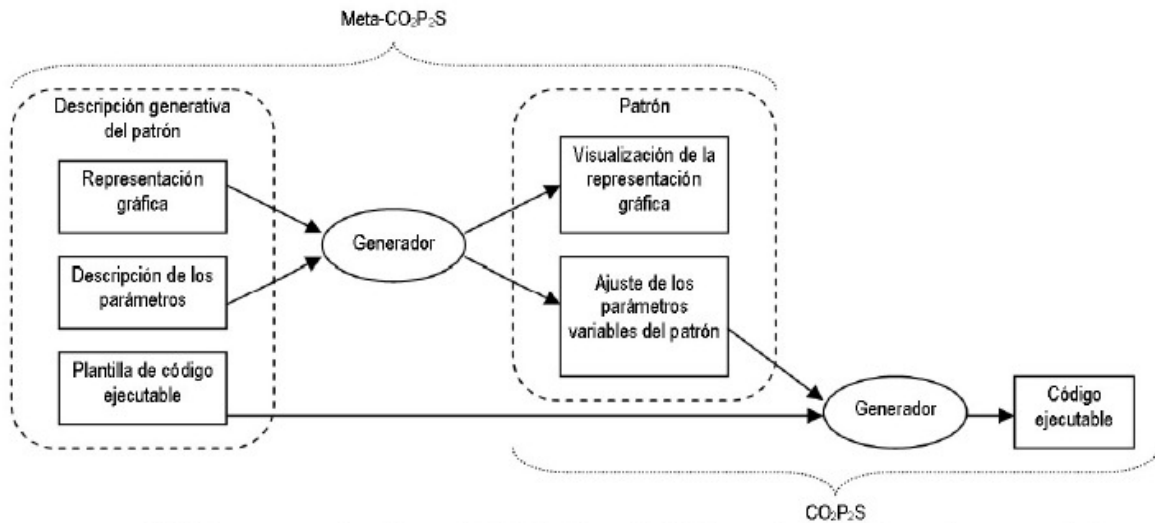
Cuando no es posible encapsular un patrón con un lenguaje de programación de propósito general, se puede utilizar una estrategia generativa. Se elabora un lenguaje específico de dominio (DSL, Domain Specific Language), cercano al usuario, para el ajuste de los parámetros variables y se genera automáticamente el



	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

código ejecutable correspondiente. S. MacDonald et al. han desarrollado dos herramientas (figura 2.7) que siguen este enfoque:

- CO<sub>2</sub>P<sub>2</sub>S (Correct Object-Oriented Pattern-based Programming System) ofrece una representación gráfica abstracta de diversos patrones almacenados previamente. El usuario configura de forma visual un patrón especificando exclusivamente sus parámetros y la herramienta aprovecha la utilidad Javadoc para generar código ejecutable a partir de plantillas prefabricadas.
- Meta-CO<sub>2</sub>P<sub>2</sub>S posibilita la implementación de nuevos patrones (representación gráfica parametrizable y generación de código ejecutable).



*Modelo generativo ofrecido por CO<sub>2</sub>P<sub>2</sub>S y Meta-CO<sub>2</sub>P<sub>2</sub>S para la encapsulación de patrones de diseño.*

- **Acoplamiento:** Como señalaron E. Yourdon y L. L. Constantine en 1979, el objetivo fundamental de cualquier diseño es conseguir un sistema mantenible. Sólo en casos excepcionales se sacrificará este objetivo para lograr una mayor velocidad de proceso o un menor tamaño de código. El logro de este objetivo pasa por alcanzar el mínimo acoplamiento entre sus componentes y la máxima cohesión de cada uno de ellos.

Sin embargo, el arquitecto C. Alexander, considerado de forma unánime como el padre del diseño basado en patrones, enunció lo siguiente:

"Es posible hacer edificios enlazando patrones, de un modo poco preciso. Un edificio construido así es una mezcla de patrones. No es denso. No es profundo. También es posible juntar patrones de modo que muchos patrones se solapen en un mismo espacio físico: el edificio es muy denso; tiene muchos significados representados en un espacio reducido; y a través de esa densidad, se hace profundo. "

Muchos autores interpretan esta postura afirmando que un diseño orientado a objetos que se limite a encadenar patrones (Stringing patterns) degenerará en una superpoblación de pequeñas clases triviales y redundantes. La tendencia opuesta, la imbricación profunda (Overlapping patterns), conlleva un alto acoplamiento.

Numerosas investigaciones se orientan hacia la composición eficaz de patrones de diseño. Por ejemplo, S. M. Yacoub y H. H. Ammar proponen la metodología POAD (Pattern-Oriented Analysis and Design) que, apoyándose en una definición de interfaz válida para lo que llaman "patrones constructivos", guía la reutilización sistemática de patrones y su composición manteniendo el control sobre el acoplamiento. Además, para dar soporte a la metodología, han desarrollado la herramienta POD (Pattern-Oriented Design Tool).

- **Mantenimiento de software y patrones de diseño:** Existe consenso sobre la contribución de los patrones de diseño a la mejora de la documentación del software y, de esta manera, a su mantenimiento. Sin embargo, muchos patrones incorporan cierto grado de complejidad para aumentar la adaptabilidad del software. Aunque algunos experimentos justifican esta adición, la mayoría de los estudios empíricos no permiten establecer una correlación entre el uso de patrones y una disminución de los cambios necesarios durante el mantenimiento.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

Admitiendo la utilidad de los patrones de diseño para la ingeniería inversa, se han producido numerosas investigaciones sobre la extracción de patrones en código de lenguajes de programación orientada a objetos, como C++. Inicialmente, los métodos de extracción eran manuales. Posteriormente, se han empleado métodos que transforman el código en representaciones intermedias que después se contrastan automáticamente con librerías de patrones.

- **Eficiencia:** Muchos de los patrones recogidos en [GHJV94] ofrecen la adaptabilidad de los diseños por medio de enlace dinámico (dynamic binding), lo que merma la eficiencia en tiempo de ejecución de sistemas que implementen un número considerable de estos patrones.

#### Marco de trabajo

Generalmente, la reutilización de código se consigue incorporando componentes prefabricados (clases, subprogramas, etc.) almacenados en librerías. El criterio que cohesiona los componentes de una librería es su aptitud para resolver cierto tipo de problemas. Es decir, su adecuación para un dominio específico.

Cuando existe una gran proximidad entre las aplicaciones informáticas de un dominio, se puede ir más allá y tratar de reutilizar el esqueleto común de las aplicaciones, técnicamente denominado "marco de trabajo". Un marco de trabajo representa las decisiones de diseño que son comunes a su dominio. Así, los marcos hacen hincapié en la reutilización del diseño frente a la reutilización del código.

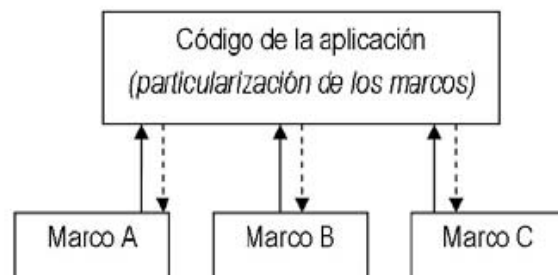
El uso de marcos de trabajo no sólo acelera la construcción de aplicaciones, además, hace que las aplicaciones tengan estructuras parecidas, por lo que son más fáciles de mantener y resultan más consistentes para los usuarios [GHJV94].

El desarrollo basado en marcos de trabajo consta fundamentalmente de dos procesos:

- (1) La construcción del marco de trabajo: Para el éxito de este proceso, debe determinarse con precisión el dominio. Las técnicas de análisis de dominio facilitan la detección de los aspectos comunes de todas las aplicaciones del dominio, que se implementarán en el marco, y de los puntos de variación, que serán parametrizados por los usuarios del marco. Acotado el dominio, se pasa a construir el marco que suele obtenerse generalizando software legado.
- (2) La construcción de aplicaciones particularizando el marco de trabajo: Los marcos de trabajo no son abstracciones de caja negra. Esto significa que el usuario deberá comprender la implementación del marco para particularizarlo, lo que implica una curva de aprendizaje considerable. Mientras que los marcos de trabajo se ubican en el "espacio de la solución", los usuarios pertenecen al "espacio del problema" y manejan puntos de variación abstractos propios de este espacio. Como la encapsulación del marco no es de caja negra, los usuarios se ven obligados a aventurarse en el espacio del problema y traducir manualmente sus puntos de variación conceptuales (mapping problem). Como inconveniente adicional, se liga la especificación de los programas a la implementación del marco. Si se altera dicha implementación, las especificaciones pasarán a ser inservibles.

#### Colisiones entre marcos de trabajo

Una aplicación compleja puede reutilizar varios marcos, lo que abre la posibilidad de solapamientos y colisiones. Por ejemplo, en la reutilización basada en marcos de trabajo, respecto a la basada en librerías de componentes, se da una inversión del flujo de control. El código nuevo de una aplicación no invoca a elementos prefabricados, sino que ocurre al revés, el marco de trabajo es quien invoca al nuevo código. La figura a continuación representa una aplicación que reutiliza los marcos A, B y C. El posible acceso simultáneo de los marcos al código de la aplicación que los particulariza plantea los problemas clásicos de la concurrencia (estados inconsistentes, interbloqueos, etc.).



*Inversión del flujo de control y colisiones en los marcos de trabajo.*

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

### La reutilización en la codificación

Quizás, la forma más primitiva de reutilización de código sea la expansión de texto basada en macros. Lamentablemente, esta técnica impone un fuerte acoplamiento entre la macro y el contexto desde el que se expande, pues su interacción se realiza mediante variables con el mismo identificador.

Las funciones que disponen de transparencia referencial, es decir, que transforman un mismo conjunto de entradas en el mismo conjunto de salidas independientemente del contexto desde el que se invocan, soslayan el problema de acoplamiento de las macros. Además, la reutilización intensiva de funciones lleva a organizarlas en librerías, aumentándose el grano de la reutilización.

En ocasiones, conviene que las funciones "tengan memoria" y puedan consultar resultados de invocaciones previas o información utilizada anteriormente por otras funciones. En los años 60 y 70 proliferaron librerías con funciones que compartían información mediante alguna estructura global de datos. Desgraciadamente, esta organización padece un acoplamiento análogo al que sufren las macros, ya que cualquier cambio en la estructura de datos se propaga a todas las funciones que la acceden.

Los tipos abstractos de datos y, posteriormente, la orientación a objetos superan el problema anterior encapsulando la estructura de datos junto con las funciones que la manipulan. Mientras se mantenga la interfaz de las funciones del tipo abstracto, los cambios en la estructura de datos sólo afectan a la implementación de dichas funciones, deteniéndose la propagación de las modificaciones.

Dada la gran envergadura de los programas, a mediados de los 90 surgió la tendencia de agrupar las clases en una entidad de grano mayor denominada componente software.

Como puede verse, el carácter formal del código ha posibilitado su reutilización efectiva desde los albores de la informática. A continuación, se resume una modalidad de reutilización de código novedosa, la orientación a aspectos.

#### *Programación orientada a aspectos*

Dada la capacidad limitada del cerebro humano para procesar información, una buena estrategia para manejar cuestiones complejas es dividir las en otras más simples que se tratan por separado. Este principio, conocido como "separación de las preocupaciones" (SOC, Separation Of Concerns), ha sido defendido desde antiguo por grandes investigadores como D. L. Parnas o E. W. Dijkstra.


La mayoría de las metodologías de análisis y de diseño, así como la práctica totalidad de los lenguajes de programación, proponen construcciones (subprogramas, clases, etc.) para organizar un sistema informático en unidades modulares. Aunque dichas construcciones facilitan la creación y encapsulación de las cuestiones funcionales que constituyen el núcleo de una aplicación, son insuficientes para capturar otro tipo de cuestiones que, por esta razón, suelen estar dispersas por toda la aplicación. Las cuestiones del primer tipo se denominan "centrales" (core concern), mientras que las del segundo tipo se llaman "transversales" (crosscutting concerns) o aspectos. Ejemplos de aspectos son la interacción entre componentes, la persistencia, la sincronización, los históricos de ejecución (logging), estrategias para el uso eficiente de la memoria, etc.

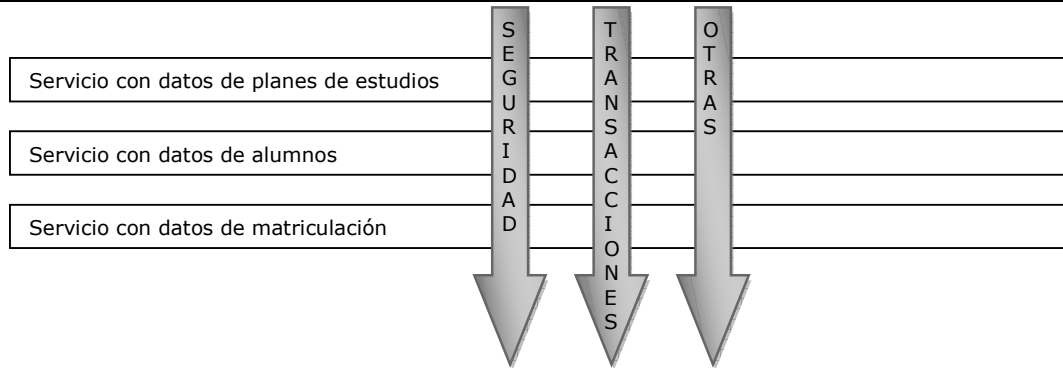
Entonces un aspecto transversal puede describirse como cualquier funcionalidad que afecta a múltiples puntos de una aplicación. La seguridad, por ejemplo, es un aspecto transversal porque muchos métodos de una aplicación pueden tener reglas de seguridad aplicadas. La figura a continuación nos ofrece una representación visual de los aspectos transversales.

La figura representa una aplicación típica que está dividida en módulos. Cada aspecto principal del módulo es ofrece en módulos. Cada aspecto principal del módulo es ofrecer servicios para su dominio particular. No obstante cada uno de esos módulos también requiere funcionalidades secundarias similares, como seguridad y gestión de transacciones.

Una técnica orientada a objetos común para la reutilización de una funcionalidad compartida es aplicar la herencia o delegación. Pero la herencia puede dar lugar a una jerarquía de objetos frágil si se utiliza la misma clase base a lo largo de una aplicación, y la delegación puede ser engorrosa porque puede requerir complicadas llamadas al objeto subordinado.

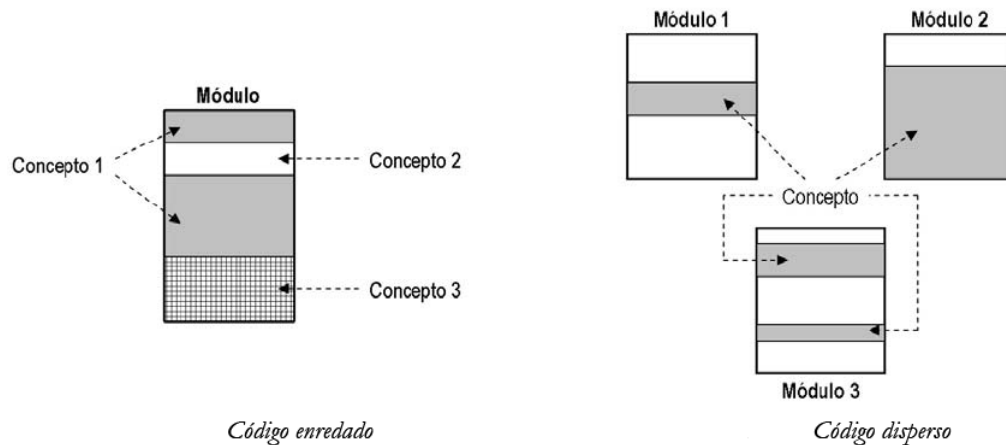
Los aspectos ofrecen una alternativa a la herencia y delegación que puede ser más clara en muchas circunstancias. Con la programación orientada a aspectos, aun puede definir la funcionalidad común en un lugar, pero puede definir de forma declarativa como y donde se aplica esa funcionalidad sin tener que modificar la clase a la que está aplicando la nueva característica. Los aspectos transversales ahora pueden modularizarse en objetos especiales llamados aspectos. Esto tiene dos ventajas. Primero, la lógica de cada aspecto está ahora en un sitio, en vez de estar esparcida por toda la base del código. En segundo lugar, nuestro módulo de servicio ahora está más limpio puesto que solo contienen el código de su tarea principal (o funcionalidad central) y las tareas secundarias se han cambiado a aspectos.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009



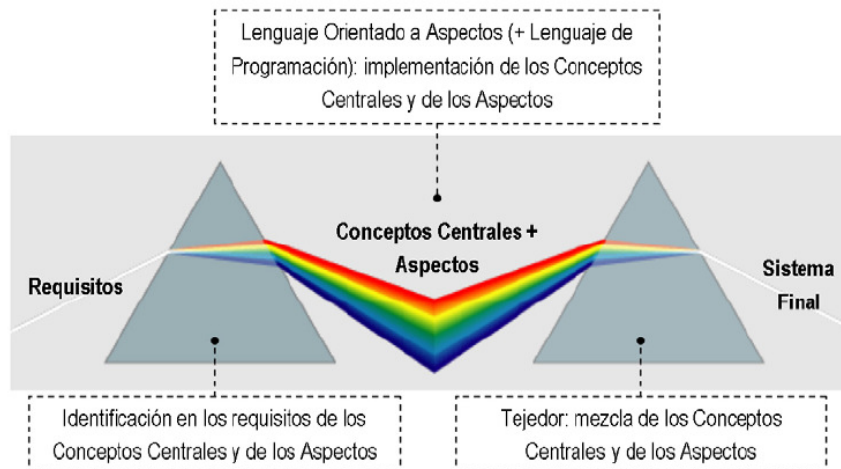
La falta de modularización de los aspectos puede percibirse cuando existe:

- Código enredado (code tangling): algunos módulos implementan más de un concepto.
- Código disperso (code scattering): un concepto aparece implementado en más de un módulo.




La encapsulación de los aspectos produce el aumento de la cohesión modular y la disminución del acoplamiento entre módulos, cualidades que redundan en una mejora de la productividad (desarrollo en paralelo), de la reutilización de módulos y del mantenimiento (localización de los cambios y control de su propagación).

El desarrollo orientado a aspectos se apoya en dos herramientas fundamentales: el lenguaje que facilita la implementación de los aspectos y el tejedor (weaver) que, para producir el sistema final, mezcla los aspectos con los conceptos centrales.



*Resumen del proceso de desarrollo orientado a aspectos.*

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 06-08-2009
APUNTE DE TEMAS DIVERSOS		

Para que el tejedor pueda realizar su trabajo automáticamente, además de la expresión del comportamiento de los aspectos, el lenguaje de implementación debe facilitar la descripción de la combinación entre aspectos y conceptos centrales. En general, los lenguajes orientados a aspectos se plantean como extensiones de lenguajes de programación. El comportamiento de los aspectos se describe con el lenguaje subyacente (Java en el caso de AspectJ e Hyper/J, C++ en el caso de AspectC++, C# en el caso de AspectC#, etc.). La parte novedosa de un lenguaje orientado a aspectos es la expresión de la combinación entre aspectos y conceptos centrales, cuyas cualidades deseables son:

- (1) Mínimo acoplamiento entre aspectos y conceptos centrales.
- (2) Capacidad de enlace (binding time) estático y dinámico entre aspectos y conceptos centrales.
- (3) Adición no invasiva de los aspectos. Los aspectos incorporan cualidades transversales a los conceptos centrales. Conviene que esta adición no implique la adaptación manual de los conceptos centrales. Cuando un arquitecto aborda la construcción de un sistema informático, se enfrenta al dilema del subdiseño frente al sobrediseño. Al comienzo de un proyecto, desconoce todos los "requisitos definitivos". Si opta por subdiseñar, el diseño inicial puede verse gravemente afectado ante un cambio en los requisitos. Por otro lado, si opta por sobrediseñar, corre el riesgo de complicar innecesariamente el diseño para anticiparse a modificaciones que nunca se producirán. Frente a los procesos formales que están orientados a la construcción de sistemas de gran tamaño, se sitúan los procesos ágiles, como eXtreme Programming o Scrum, que buscan la reacción efectiva frente a cambios en proyectos más reducidos. Los procesos ágiles defienden el subdiseño frente al sobrediseño, es decir, creen que la construcción de sistemas debe ser pragmática y ceñirse a los requisitos "en tiempo presente". R. Laddad opina que la orientación a aspectos está en sintonía con esta tendencia y que la adición no invasiva de aspectos puede ser una herramienta muy valiosa para adaptar un sistema evitando su modificación manual.

#### *AspectJ*

A continuación, se utilizará el lenguaje AspectJ para ilustrar los principales elementos de un lenguaje de aspectos.


En AspectJ, la superposición de los aspectos sobre los conceptos centrales (o sobre otros aspectos) se expresa mediante tres elementos:

- (1) Punto de unión (join points): Son los lugares de un programa donde puede insertarse un aspecto. Este punto puede ser una llamada a un método, una excepción o incluso la modificación de un campo. Estos son puntos donde el código del aspecto puede insertarse en el flujo normal de la aplicación para añadir un nuevo comportamiento. AspectJ ofrece un abanico muy amplio de tipos de join points, llamada a métodos, acceso a atributos, inicialización de clases y objetos, etc. Algunos join points se consideran conflictivos y no están disponibles para el programador. Los join points disponibles para el programador se denominan exposed join points.
- (2) Puntos de corte (pointcuts): Los puntos de corte ayudan a reducir los puntos de unión notificados por un aspecto. Si la notificación define el qué y el cuándo de los aspectos, entonces los puntos de corte (pointcuts) definen el dónde. Una definición de puntos de corte se corresponde con uno o más puntos de unión en los que debe incrustarse una notificación. A menudo se especifican estos puntos de corte utilizando nombres de clases y métodos específicos, o a través de expresiones normales que definen los patrones de correspondencia de clase y método.
- (3) Notificaciones (advices): El trabajo de un aspecto se llama notificación. Cuando el flujo de ejecución de un programa alcanza un pointcut, se produce un salto a su advice asociado, que contiene el código Java que implementa parte o la totalidad del comportamiento del aspecto. Este funcionamiento es muy similar al de la Programación Dirigida por Eventos, donde el trozo de código del programa original delimitado por un pointcut "hace de botón" que dispara un evento cuando el flujo de ejecución lo alcanza y el advice "hace de manejador del evento". Los advices se pueden ejecutar antes (before), después (after) o sobre (around) la activación del pointcut. Algunos join points tienen asociado un contexto accesible desde el advice correspondiente. Por ejemplo, el contexto de la llamada a un método contiene los objetos llamante y llamado.

Si se solapan los pointcuts de varios aspectos pueden producirse colisiones. Para resolver esta clase de conflictos, AspectJ permite fijar un orden de prioridad entre aspectos mediante la sentencia declare precedence.

Numerosos investigadores apoyan la orientación a aspectos, sin embargo, otros la critican pues consideran, por ejemplo, que la verificación de un programa orientado a aspectos se dificulta porque a) la introducción de aspectos complica el hilo de ejecución de los programas, y por tanto, la realización de pruebas de caja transparente; b) el comportamiento de un aspecto puede depender del contexto del programa sobre el que se combine, lo que impide la prueba aislada del aspecto.

Algunas investigaciones buscan una metodología que facilite la identificación sistemática de aspectos en los requisitos. Otras, orientadas hacia la ingeniería inversa, persiguen la detección automática de aspectos en código legado.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

Quizás, el campo de investigación más activo sea el desarrollo de nuevos lenguajes de aspectos y tejedores que reúnan las cualidades deseables de combinación entre conceptos centrales y aspectos mencionados anteriormente.

### La reutilización en la prueba

El auge de los procesos ágiles, que atribuyen una enorme importancia a la prueba del software, ha contribuido de forma decisiva al desarrollo y la difusión de marcos de trabajo para la automatización de las pruebas de unidades: JUnit para Java, NUnit para C#, CppUnit para C++, etc. Lamentablemente, como se indicó anteriormente, los marcos de trabajo no son abstracciones de caja negra. Existen algunas propuestas de aumento de la abstracción de marcos para pruebas de unidades mediante un enfoque generativo.

La formalización de las pruebas de unidades constituye un paso decisivo para su reutilización. Varias investigaciones respaldan que la reutilización de componentes posibilita, en gran medida, la reutilización de las pruebas de dichos componentes. Algunos autores consideran que la reutilización de pruebas puede incrementarse notablemente en dominios específicos, aprovechando la cercanía entre las aplicaciones de una familia de productos.

#### El desarrollo dirigido por modelos

A continuación, se citan algunas definiciones del término "modelo":

"Un modelo es una simplificación de la realidad. Un modelo proporciona los planos de un sistema. Los modelos pueden involucrar planos detallados, así como planos más generales que ofrecen una visión global del sistema en consideración. Un buen modelo incluye aquellos elementos que tienen una gran influencia y omite aquellos elementos menores que no son relevantes para el nivel de abstracción dado."

"Un modelo de un sistema es una descripción o especificación de dicho sistema y su entorno con algún propósito en particular. Con frecuencia, se representa combinando gráficos y textos. Estos últimos, pueden escribirse en lenguaje natural o en algún lenguaje de modelado. "

"Un modelo es un conjunto de elementos que describe una realidad física, abstracta o hipotética. Un buen modelo sirve como medio de comunicación; es más barato de construir que un sistema real; y sirve de soporte para la implementación. El grado de detalle de un modelo puede variar desde un boceto gráfico hasta un modelo totalmente ejecutable. "

Tradicionalmente, los modelos han servido como medio de comunicación entre clientes, analistas, diseñadores y programadores. Durante el análisis, los modelos se han utilizado para representar los conceptos de un dominio y para especificar las interfaces externas de un sistema. En el diseño, los modelos han servido para representar los planos que guían la implementación. Dada la variedad de personas, en cuanto a formación e intereses, que trabajan con modelos, los lenguajes de modelado, en general, han optado por sacrificar formalismos en aras de una mayor claridad. La primera versión de UML, el lenguaje de modelado más difundido en la actualidad, es un claro exponente de esta tendencia.


La utilización de modelos, ha recibido numerosas críticas, especialmente por los partidarios de los procesos de desarrollo ágiles:

- Los modelos suelen ser poco comprensibles para el cliente. En la comunicación cliente-analista es preferible emplear prototipos.
- Los lenguajes de modelado tipo UML son ambiguos, lo que impide su traducción automática a código ejecutable y fuerza a los programadores a tomar decisiones que no aparecen reflejadas en ningún diagrama.
- En ocasiones, durante las pruebas y el mantenimiento se retoca el código sin que se actualicen los modelos. Como resultado, los modelos son una fuente de inconsistencia que no contribuye a mejorar la documentación.
- En definitiva, el producto verdaderamente relevante del proceso de desarrollo es el código. Los esfuerzos de desarrollo deben centrarse en construir y mantener un código de calidad fomentando el uso intensivo de pruebas, refactorizando el código, etc. Sólo es justificable el uso de modelos sencillos en casos puntuales.

En contraposición a estas críticas, el desarrollo dirigido por modelos (MDD, Model Driven Development) considera que los modelos son productos de primera categoría que permiten el avance hacia mayores niveles de abstracción y reutilización. La arquitectura dirigida por modelos (MDA, Model Driven Architecture) del consorcio OMG (Object Management Group) es la principal representante del MDD y persigue solventar algunos de los problemas fundamentales del uso de modelos, entre los que cabe destacar:

- La eliminación de la ambigüedad en los modelos, posibilitándose la traducción automática de estos a código ejecutable.



	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

- La creación de un estándar que facilite la reutilización de modelos y su intercambio entre distintas herramientas de modelado.

Según el grado de abstracción, MDA establece tres puntos de vista para un sistema informático:

- (1) El modelo independiente de computación (CIM, Computation Independent Model): Representa el dominio de un sistema y sirve como medio de comunicación entre el cliente y el desarrollador.
- (2) El modelo independiente de plataforma (PIM, Platform Independent Model): Representa un sistema obviando su entorno de ejecución. Ejemplos de plataformas son Java, CORBA, .NET o los sistemas operativos Linux, Solaris, Microsoft, etc.
- (3) El modelo específico de plataforma (PSM, Platform Specific Model): Representa un modelo incluyendo los detalles de su entorno de ejecución. Este modelo es directamente traducible a código ejecutable.

Además, MDA contempla la descomposición en modelos dentro de un mismo nivel de abstracción. Incluso reconoce la posibilidad de definir y combinar modelos transversales en el sentido de la orientación a aspectos.

Dentro de los partidarios de MDA, existen dos corrientes: los traduccionistas, que abogan por modelos capaces de producir aplicaciones completas mediante un proceso de traducción automático, y los elaboracionistas, que apuestan por la producción parcial de aplicaciones. Para unos y otros, la primera opción para escribir modelos es UML.

Las versiones iniciales de UML padecían serias carencias para representar el comportamiento y definir la semántica de los modelos. La versión 2.0 supera muchas de estas carencias. Sin embargo, algunos autores consideran que esta nueva versión tiene un grado de complejidad excesivo que dificulta su manejo.

Cuando UML es insuficiente, MDA propone dos posibilidades:

- (1) Extender UML desarrollando un nuevo perfil (profile) por medio del lenguaje de restricciones OCL y los estereotipos (stereotypes) de UML. Un ejemplo notorio de esta posibilidad es el perfil Executable UML, que permite la definición de modelos computables y es de propósito general. Otros autores, con el fin de simplificar los modelos, optan por definir perfiles computables para dominios específicos.
- (2) Construir o metamodelar un nuevo lenguaje de modelado, generalmente específico para un dominio, utilizando el metalenguaje MOF (Meta-Object Facility) para describir su sintaxis abstracta. UML está descrito en MOF. Para facilitar la reutilización y el intercambio de modelos, se ha creado el estándar XMI (XML Model Interchange), que establece cómo derivar un esquema XML desde la sintaxis MOF de un lenguaje de modelado y cómo traducir los modelos escritos en ese lenguaje a documentos XML.

El traduccionista S. J. Mellor considera que MDA reconcilia los partidarios de UML con los partidarios de los procesos ágiles: "Si un modelo es ejecutable, es operacionalmente lo mismo que el código. Así, los principios ágiles para la construcción de código también son aplicables a la construcción de modelos ejecutables".

Además, propone el siguiente proceso de desarrollo MDA:

- (1) Delimitación del dominio y posible descomposición jerárquica del mismo en subdominios.
- (2) Formalización del conocimiento del dominio (o de cada subdominio).
  - a. Identificar los requisitos del dominio.
  - b. Abstractar el conocimiento del dominio en algún grupo de conceptos.
  - c. Seleccionar, o desarrollar si fuera necesario, un lenguaje de modelado apropiado para representar los conceptos.
  - d. Expresar formalmente los conceptos en este lenguaje.
  - e. Comprobar el modelo resultante del paso anterior.
- (3) Construir los puentes PIM-PSM (Platform Independent Model-Platform Specific Model).
  - a. Especificar las funciones de conversión
  - b. Si es necesario, añadir marcas de forma no invasiva al modelo.
  - c. Comprobar las funciones de conversión.
  - d. Examinar la completitud de los modelos.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO DE SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE TEMAS DIVERSOS	VERSIÓN: 1.0 VIGENCIA: 06-08-2009

e. Transformar los modelos.

Como puede comprobarse, los pasos 1, 2.a, 2.b, 2.c, 3.a y 3.c sirven para el desarrollo de una línea de productos, mientras que los pasos 2.d, 2.e, 3.b, 3.d y 3.e se ocupan del desarrollo de un producto particular de la línea.

Actualmente, existe una cantidad considerable de herramientas comerciales y de código abierto que soportan parcialmente MDA, por ejemplo, ArcStyler, OptimalJ, AndroMDA, NetBeans Metadata Repository, ModFact, Eclipse Modeling Framework, Rational Rose, EclipseUML, Poseidon, ArgoUML, CodaGen Architect, etc.

#### **BIBLIOGRAFÍA**

[Gil07] Dr. Rubén Heradio Gil (2007). "Metodología de Desarrollo de Software basada en el Paradigma Generativo. Realización mediante la transformación de ejemplares". España.

[Vicente05] Dra. Cristina Vicente Chicote (2005). "Desarrollo Integral de Sistemas de Procesamiento de Información Visual: Un Enfoque Multiparadigma basado en Líneas de Producto, Componentes y Generación Automática de Software". Colombia.

[Walls08] Craig Walls (2008). "Spring". Anaya Multimedia. España.