Informe Técnico – Technical Report
DPTOIA-IT-2002-004
Marzo, 2002

Líneas de Productos, Componentes, Frameworks y Mecanos

Francisco José García Peñalvo
Juan-Antonio Barras
Miguel Ángel Laguna Serrano
José Manuel Marqués Corral

Departamento de Informática y Automática
Universidad de Salamanca

Revisado por:

Dra. María N. Moreno García

Departamento de Informática y Automática

Universidad de Salamanca

mmg@usal.es

Dra. Yania Crespo González-Carvjal

Departamento de Informática

Universidad de Valladolid

yania@infor.uva.es

Aprobado en el Consejo de Departamento de 18 de marzo de 2002.

Información de los autores:

Dr. Francisco José García Peñalvo

Área de Ciencias de la Computación e Inteligencia Artificial
Departamento de Informática y Automática
Facultad de Ciencias - Universidad de Salamanca
Plaza de la Merced S/N – 37008 - Salamanca
fgarcia@usal.es

Juan-Antonio Barras

Consejería de Fomento de la Junta de Castilla y León
barras@tiscali.es

Dr. Miguel Ángel Laguna Serrano

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática - Universidad de Valladolid
fgarcia@usal.es

Dr. José Manuel Marqués Corral

Área de Ciencias de la Computación e Inteligencia Artificial
Departamento de Informática - Universidad de Valladolid
jmmc@infor.uva.es

Este trabajo ha sido parcialmente financiado por el proyecto Dolmen (CICYT TIC2000-1673-C06-05).

Resumen

La reutilización del software es una de las disciplinas que más beneficios promete a la hora de desarrollar nuevos productos software con un enfoque de Ingeniería del Software. Sin embargo, a lo largo de su historia estos beneficios parece que no acaban de llegar. Una de las principales causas ha sido la granularidad los elementos software a reutilizar, demasiado pequeña para ser realmente efectiva, por ejemplo el módulo o la clase. Para solucionar el problema de la granularidad en la reutilización surgen diferentes alternativas de reutilización basadas en elementos reutilizables de grano grueso.

En este trabajo se van a repasar los principales elementos reutilizables de grano grueso, relacionándolos con los métodos de ingeniería de dominio y con las líneas de productos. En este sentido interesa remarcar la idoneidad que presenta el Modelo de Mecano para dar soporte a las líneas de productos.

Abstract

Software reuse is one of the most promising disciplines related to Software Engineering due to its potential benefits. However, through its history these benefits seem they are not real. One of the main reasons could be the granularity of the reusable elements that it was too finer to be really effective, examples of this kind of reusable units was the module in seventies or the class in the eighties. To solve the granularity in reusable elements problem, several reuse alternatives based on coarse-grained reusable elements arise.

In this work the main coarse-grained reusable elements proposals are reviewed, relating then to the domain engineering methods and the product lines. In this way, the eligibility of the Mecano Model to support the product line concept is underlined.

Tabla de Contenidos

1. Introducción

Para que la reutilización del software ofrezca los beneficios que de ella se esperan, se debe producir un acercamiento sistemático e institucional en su implantación [Griss, 1993], [Griss, 1996b] huyendo de una aproximación oportunista, que no dejará más que algunos beneficios muy localizados que no repercutirán en la organización global. Tras numerosos intentos de introducir la reutilización en el desarrollo de sistemas software, se ha llegado a una serie de conclusiones [Kang et al., 1998b]:

- *La reusabilidad debe ser una característica inherente al software*. La reutilización de software que no ha sido desarrollado para su reutilización es técnicamente más difícil y costosa que la reutilización del software desarrollado para ese fin.
- *El desarrollo de elementos software reutilizables requiere de la determinación de arquitecturas*. Una arquitectura define cómo los elementos software se integran para crear un sistema, de forma que contar con arquitecturas estandarizadas permite definir los contextos en los que los elementos reutilizables pueden ser desarrollados.
- *El desarrollo de arquitecturas estándares y de elementos software reutilizables requiere la comprensión de las características comunes y de las que pueden variar en los sistemas que conforman un dominio.*

Así pues, la ingeniería de dominio surge como una evolución de la reutilización sistemática del software basada en modelos donde las arquitecturas software juegan un papel ampliamente destacado.

2. Ingeniería de dominio

La reutilización del software dentro de un dominio de aplicación pasa por el descubrimiento de elementos comunes a los sistemas pertenecientes a dicho dominio. Cuando se utiliza este enfoque se está produciendo un cambio de un desarrollo orientado a un único producto software a un desarrollo centrado en varios productos que comparten unas características formando una familia. Esto provoca una reestructuración del proceso software de forma que surgen dos procesos distintos: la *ingeniería de dominio* y la *ingeniería de aplicación*.

La ingeniería de dominio se centra en el desarrollo de elementos reutilizables que formarán la familia de productos, mientras que la ingeniería de aplicación se orienta hacia la construcción o desarrollo de productos individuales, pertenecientes a la familia de productos, y que satisfacen un conjunto de requisitos y restricciones expresados por un usuario específico, reutilizando, adaptando e integrando los elementos reutilizables existentes y producidos en la ingeniería de dominio.

La ingeniería de dominio se puede definir como el proceso clave que se necesita para el diseño sistemático de una arquitectura y de un conjunto de elementos software reutilizables que pueden ser usados en la construcción de una familia de aplicaciones relacionadas o subsistemas. Es el proceso sistemático que incorpora criterios de negocio y produce un soporte racional, modelos y arquitecturas que permiten tomar mejores decisiones, llevar un registro del dominio, obtener nuevas versiones y mejorar el proceso de desarrollo gracias al conocimiento que se tiene del sistema [Griss, 1996a], [Griss et al., 1998].

El objetivo fundamental de la ingeniería de dominio es la optimización del proceso de desarrollo del software en un espectro de múltiples aplicaciones que representan un negocio común o problema de dominio [Simos et al., 1996].

El propósito de la ingeniería de dominio es el desarrollo de elementos software dentro de un dominio para que puedan ser usados (y reutilizados) en la construcción de aplicaciones para un dominio concreto.

Así, la ingeniería de dominio ofrece un conjunto de modelos de referencia que describen el dominio, identificando las arquitecturas y los componentes que permitirán el desarrollo de aplicaciones dentro de él. Además, ofrece la base de conocimiento adecuada para comprender el espacio del problema definido por el software presente en el dominio. Es, por tanto, una parte clave en la consecución de una arquitectura y un conjunto de elementos reutilizables que reúnan la calidad suficiente para confiar en ellos.

La ingeniería de dominio tiene sus orígenes en los trabajos de James M. Neighbors sobre reutilización basada en generación a través del paradigma Draco [Neigbors, 1984] a principios de la década de los ochenta. Desde entonces se han publicado varias aproximaciones que incluyen un amplio rango de métodos y procesos, tanto formales como informales, para llevar a cabo las diferentes actividades en que se descompone la ingeniería de dominio, para capturar y representar información sobre sistemas que comparten un conjunto común de características y datos. Entre las diversas propuestas existentes en la ingeniería de dominios cabe destacar como las más representativas a FODA (*Feature-Oriented Domain Analysis*) [Kang et al., 1990], OODA (*Object Oriented Domain Analysis*) [Cohen y Northrop, 1998], ODM (*Organization Domain Modeling*) [Simos, 1995], [Simos et al., 1996], DAGAR (*Domain Architecture-based Generation for Ada Reuse*) [Klingler y Solderitsch, 1996], FORM (*Feature-Oriented Reuse Method*) [Kang, 1998], [Kang et al., 1998a], [Lee et al., 2000], FeatureRSEB (*Feature Reuse-Driven Software Engineering Business*) [Griss et al., 1998] y FODAcom [Vici y Argentieri, 1998].

La diferencia entre estas propuestas está en la manera en la que identifican el dominio de forma efectiva y hacen uso de la máxima experiencia sobre el dominio, la arquitectura y los sistemas existentes. Algunos métodos se centran en cómo conseguir ejemplares existentes para hacer un análisis detallado de los mismos, mientras que otros están orientados a cómo encontrar, representar y agrupar conjuntos de características (*features*).

Diferentes aplicaciones en la misma familia de aplicaciones o dominio de problema se comparan o se distinguen gracias a sus *features*. Cuando se desarrolla una arquitectura y un conjunto de componentes para su reutilización es importante comprender cuáles de ellas se incluyen como parte de los elementos reutilizables, cómo se relacionan y cuáles son obligatorias u opcionales para las diferentes aplicaciones.

Una característica o *feature* es una abstracción funcional distintiva e identificable que puede ser empaquetada, implementada, probada, distribuida y mantenida. Son, por tanto, objetos de primera clase en el desarrollo de software orientado a un dominio [Kang, 1998].

Existen diferentes tipos de *features* que son tenidos en cuenta dependiendo del interés que se tenga puesto en el desarrollo del sistema. Tanto los usuarios, como los analistas y los desarrolladores están involucrados en el desarrollo pero con diferentes perspectivas. Los usuarios están más preocupados por los servicios o funciones que debe ofrecer el sistema; los analistas y diseñadores se centran en las tecnologías del dominio; y los desarrolladores se interesan especialmente por las técnicas de implementación. Las aplicaciones no podrán llevarse a cabo sin un consenso de todas las partes que permita conseguir un conjunto de *features* consistente.

Así, las *features* pueden clasificarse en cuatro categorías [Lee et al., 2000], como se muestra en la Figura 1:

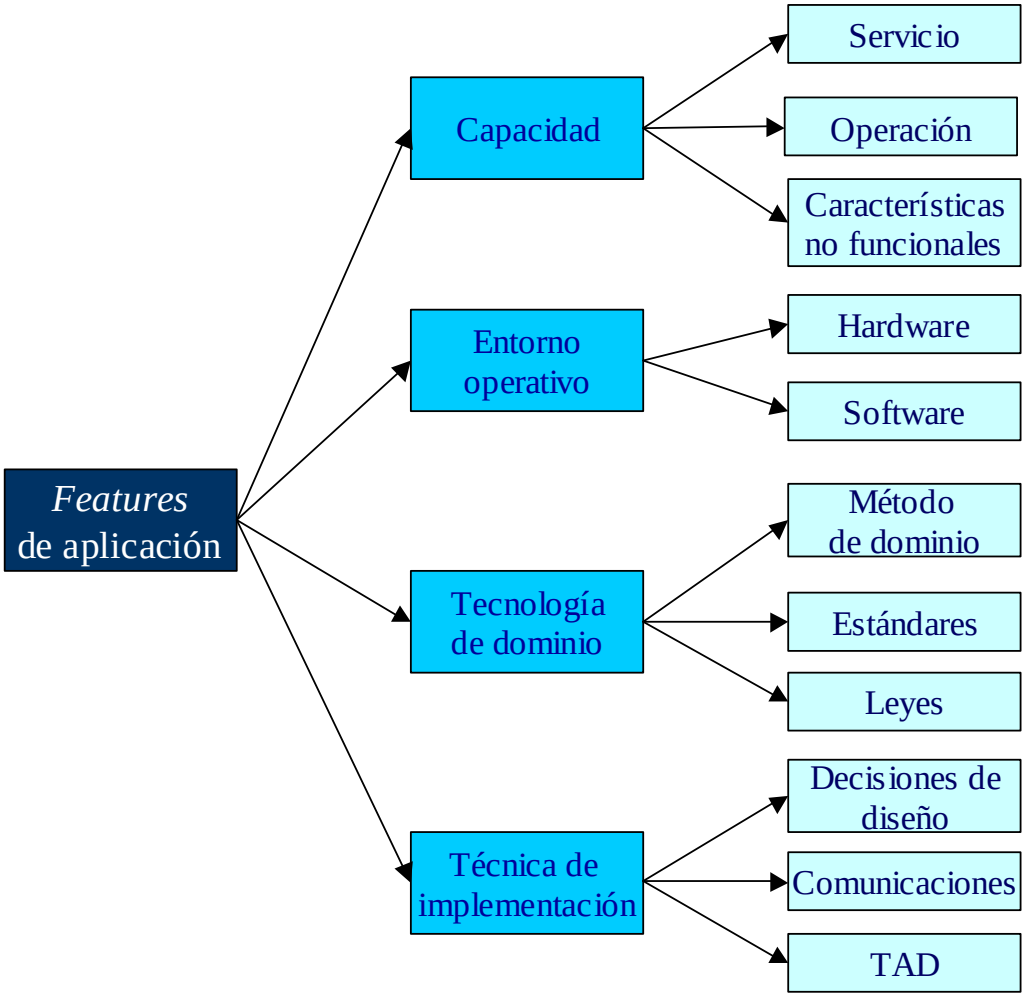


Figura 1. Clasificación de las *features* [Lee at al., 2000]

La diversidad en las propuestas trae consigo una diversidad en la terminología, existiendo una falta de consenso en la misma. Normalmente, por ingeniería de dominio se hace referencia a todo el proceso de creación de los elementos reutilizables propios del dominio. En lo que ya no existe uniformidad es en el nombre de las fases que componen el proceso. Típicamente existen dos fases muy diferenciadas: una primera de análisis del dominio [Arango y Prieto-Díaz, 1991], donde debe adquirirse el conocimiento sobre el dominio (búsqueda de elementos comunes y diferencias en la familia de sistemas que conforman el dominio) y modelarse, y una segunda fase de ingeniería de componentes de dominio, en la que los elementos software reutilizables propios del dominio son creados.

La forma más efectiva de organizar la ingeniería de dominio es hacerlo en el contexto de líneas de productos específicas. Dentro de una línea de productos, los dominios son analizados y la información sobre éstos capturada y organizada en modelos del dominio y en elementos software reutilizables (*assets*), facilitando además su evolución [Klingler y Solderitsch, 1996]. Las líneas de productos serán objeto de estudio en el apartado tercero.

A continuación se van a explicar someramente los fundamentos de ODM, DAGAR, FODA, FORM y RSEB.

2.1. ODM

ODM (*Organization Domain Modeling*) [Simos, 1995], [Simos et al., 1996] afronta la ingeniería de dominio con el argumento de que los dominios están conformados por el contexto

de múltiples sociedades superpuestas de uso, desarrollo, mantenimiento, adaptación y aplicación. ODM no sólo trata con dominios, sino con dominios de Organización. ODM utiliza un modelado explícito del contexto social y organizacional para concretar completamente el ciclo de vida de modelado. Se desarrolla a lo largo de tres fases: **Planificación del Dominio**, establece los objetivos, selecciona y enfoca el dominio para el proyecto, de acuerdo con las necesidades generales de la organización. Define las reglas básicas de pertenencia al dominio y modela las relaciones de este dominio con otros adyacentes; **Modelado del Dominio**, describe las características comunes y variables de los sistemas que formarán parte del dominio, con unas bases formales para modelar las variantes entre características. La idea clave es el *concepto*: idea abstracta generalizada desde instancias particulares. Los atributos de un concepto son denominados *características*; **Desarrollo de la Base de *Assets***, parte del modelo de conceptos desarrollado durante la fase Modelado del Dominio y se centra en definir, especificar y organizar la arquitectura de un conjunto de *assets* de base que soporten los requisitos específicos para ese dominio.

El conjunto final de especificaciones para los módulos reutilizables que provee ODM es el punto de entrada para DAGAR.

2.2. DAGAR

DAGAR (*Domain Arquitecture-based Generation for Ada Reuse*) [Klingler y Solderitsch, 1996] es un método que parte de los productos obtenidos con ODM. Incluye el desarrollo de una arquitectura de dominio, la implementación de *assets* dentro de la arquitectura y un proceso para la elaboración de aplicaciones a partir de una selección del conjunto de *assets*.

La implementación de los conceptos de variabilidad y similitud se lleva a cabo dentro de DAGAR definiendo *reinos*. Un reino identifica un módulo y su interfaz dentro de la estructura en capas de la arquitectura del dominio. Cada reino representa un núcleo de servicios ofrecido por los *assets* del dominio, y contiene una *especificación del reino* y unos *agujeros*, que son ubicaciones donde la implementación del reino puede ser distinta.

Un *componente* es una instancia de la implementación de un reino. La *especificación de un componente* dice cómo se han de llenar los agujeros en el reino. El *cuerpo de un componente* es la implementación de las funciones y procedimientos del reino. La variabilidad queda expresada en el dominio cuando un reino origina varios componentes, cada uno de los cuales tiene implementaciones del mismo conjunto de servicios pero en formas diferentes.

2.3. FODA

FODA (*Featured-Oriented Domain Analysis*) [Kang et al., 1990] se dedica a explorar sistemas de software relacionados para descubrir y explotar características comunes. Recolecta y representa información sobre sistemas software que comparten un conjunto como datos y capacidades. El concepto clave en FODA, es el de característica (*feature*): abstracción funcional identificable y distinguible. FODA propone este proceso de tres fases: **Análisis de contexto:** define los límites del dominio que está siendo analizado; **Modelado del dominio:** describe los problemas dentro del dominio que pueden ser resueltos mediante aplicaciones software; **Modelado de la arquitectura:** crea la/s arquitectura/s que implementa una solución a los problemas de ese dominio.

2.4. FORM

FORM (*Featured-Oriented Reuse Method*) [Kang, 1998], [Kang et al., 1998a], [Lee et al., 2000] extiende FODA a las fases de diseño e implementación software, y prescribe cómo el modelo de características es utilizado para desarrollar arquitecturas de dominio y componentes para reutilización.

1. Identificación del dominio y del alcance.- La mayoría de las aplicaciones consisten ene varios subsistemas o subproblemas distintos y reconocibles, aunque sólo algunos de ellos son económicamente reutilizables. Por lo tanto es importante decidir qué partes son válidas para un futuro tratamiento. En FODA se construye un modelo de contexto, mientras que en RSEB el alcance está relacionado con la descomposición en sistemas de componentes y un modelo de casos de uso de alto nivel que describe la arquitectura del sistema y el contexto.
2. Selección y análisis de ejemplos, necesidades y tendencias.- Hay un delicado equilibrio entre la reutilización reactiva y proactiva. Un conjunto de elementos reutilizables debe anticiparse a las necesidades futuras. Dado que esto es difícil de hacer, y es la razón por la que construir software reutilizable es más caro que construir software convencional. El proceso de reutilización debe encontrar los elementos esenciales comunes y la variabilidad, así como dar prioridad a las partes que deben ser tenidas en cuenta en la reutilización. La mayoría de las aproximaciones seleccionan ejemplos clave y extraen sus conjuntos de *features* esenciales. Los ejemplos relacionan el ámbito del dominio con la evaluación de las necesidades de los usuarios, el mercado, la tecnología y las tendencias del negocio.
3. Identificación, recolección y agrupación de los conjuntos de *features*.- Utilizando modelos de análisis, tablas y/o gráficos, las *features* que aparecen juntas (AND) o son variantes que seleccionar (OR, XOR) se estructuran en un marco de decisión, de esta forma la terminología del dominio es almacenada. En FODA un modelo de información describe las entidades de dominio y las relaciones entre ellas, mientras que RSEB se construye un modelo de casos de uso y un modelo de objetos. En FODA un modelo de comportamiento describe las relaciones dinámicas entre las entidades del dominio el cual se corresponde con los diagramas de secuencia e interacción de RSEB. El modelo funcional de FODA, relacionado con los modelos de objetos de RSEB, describe el flujo de datos entre las entidades. El modelo de *features* mantiene todos los modelos juntos estructurando y relacionando los conjuntos de *features*.
4. Desarrollo de un modelo y una arquitectura de dominio o genéricos.- A partir de estos conjuntos de *features*, un modelo de dominio resume las características esenciales de todas o algunas de las aplicaciones del dominio. También se desarrolla una arquitectura del sistema que relaciona los mecanismos principales las *features*, los subsistemas y las variantes. La arquitectura cubre los subsistemas y las aplicaciones resultantes, define los servicios principales, especifica las interfaces de forma precisa y sirve como modelo de referencia y como anteproyecto funcional. La arquitectura de dominio de FODA está muy cercana a la capa arquitectónica de RSEB.
5. Representación de las partes comunes y la variabilidad.- Se identifican los subsistemas, módulos y funciones genéricas, relacionándose entre ellas mediante generalizaciones, especializaciones o alternativas.
6. Explotación de las partes comunes y la variabilidad.- Se emplean notaciones y mecanismos para especificar diferentes clase de productos genéricos o parametrizados.
7. Implementación, certificación y empaquetado de los elementos reutilizables.- El subconjunto más importante de *assets* candidatos se implementan y se distribuyen como elementos reutilizables certificados, bajo una política de gestión de la configuración. Los *assets* utilizan diferentes tecnologías, desde la composición a la generación, empleando lenguajes y *kits* [Griss y Wentzel, 1994], [Griss y Wentzel, 1995]. Otros elementos reutilizables serán implementados cuando se necesiten.

Tabla 1. Pasos de la ingeniería de dominio en FODA y RSEB [Griss et al., 1998]

Este método de ingeniería de dominio se basa en que las características son muy similares a los objetos software de una aplicación en términos de sus definiciones [Kang et al., 1998].

El interés que presenta FORM se debe a que es un método sistemático para realizar ingeniería de dominio; propone unas tareas bien definidas y un conjunto de diagramas suficiente para poder representar los productos obtenidos durante la ejecución de cada tarea.

FORM comienza con un análisis de la similitud y variabilidad entre aplicaciones en un dominio en concreto en términos de servicios, entornos de operación, tecnologías de dominio y técnicas de implementación. El modelo construido durante este análisis es llamado *modelo de características*, y en él se representan tanto las características obligatorias, como las alternativas que pueden ser seleccionadas para las diferentes aplicaciones del dominio.

A continuación, este modelo se utiliza para definir un conjunto de *arquitecturas de referencia*, que se descomponen en tres niveles jerárquicos: subsistemas, procesos y módulos, de forma que cada uno de ellos se corresponda con las características definidas en el modelo de características.

Los módulos obtenidos son la base para la creación de componentes reutilizables listos para ser empleados durante el desarrollo de aplicaciones.

2.5. RSEB

RSEB (*Reuse-Driven Software Engineering Business*) [Griss, 1996a], [Jacobson et al., 1997] es un proceso de reutilización sistemático dirigido por casos de uso, que utiliza ampliamente conceptos extraídos de FODA y ODM (en la Tabla 1 se resumen los principales pasos que promueven FODA y RSEB en la ingeniería de dominio). La arquitectura y los sistemas reutilizables son descritos por casos de uso y luego transformados en modelos de objetos que pueden ser trazados desde estos casos de uso. Éstos son primordiales para definir arquitecturas, subsistemas y objetos reutilizables.

RSEB se basa en el método OOSE (*Object-Oriented Software Engineering*) [Jacobson et al., 1992] y la ingeniería de negocio orientada a objetos OO BPR (*Object-Oriented Business Process Reengineering*) [Jacobson et al., 1994].

RSEB define varios procesos de desarrollo software dirigidos por modelos*: Architecture Family Engineering* (desarrolla una arquitectura por capas), *Component System Engineering* (desarrolla sistemas de componentes reutilizables), y *Application System Engineering* (desarrollo de aplicaciones).

La variabilidad se trata estructurando los casos de uso y los modelos de objetos utilizando los llamados *puntos de variación*.

3. Líneas de productos

La reutilización del software es uno de los objetivos fundamentales dentro de la ingeniería del software. Ya en los últimos años de la década de los sesenta, la idea de construir sistemas mediante la composición de componentes software fue presentada como solución a la afamada crisis del software por M. D. McIlroy [McIlroy, 1976]. Durante la década de los setenta se propugnó la reutilización de los módulos, mientras que en los años ochenta la influencia del paradigma orientado a objetos hizo que la clase se convirtiera en la unidad de reutilización. Sin embargo, todas estas tendencias fallaban en conseguir un enfoque sistemático de reutilización porque daban lugar a iniciativas individuales, frecuentemente realizadas a pequeña escala.

Con estos enfoques se deja de lado la reutilización de elementos software de mayor grano, que pueden corresponderse con partes significativas de los sistemas a construir, y además suelen necesitar de una adaptación de muchos de sus aspectos. Para solucionar este apartado surgen los *frameworks* [Wirfs-Brock y Johnson, 1990] y la programación orientada a componentes [Szyperski, 1998] como aproximaciones a la reutilización del software.

De todas las experiencias anteriores se pueden obtener dos lecciones básicas:

1. La reutilización oportunista no es efectiva en la práctica, de lo que se deduce que para que un programa de reutilización sea efectivo debe estar previamente planificado.
2. Un enfoque únicamente ascendente de reutilización, esto es, la composición arbitraria de elementos software reutilizables para la construcción de un sistema software no funciona en la práctica. Se requiere el empleo de una técnica

descendente, por ejemplo un conjunto de componentes que se integren en una estructura de alto nivel definida como una arquitectura software.

Una arquitectura de línea de productos es una arquitectura común a un conjunto de productos o sistemas relacionados. Las líneas de producto se entienden como una de las formas más prometedoras para llevar a buen término un plan de reutilización debido, fundamentalmente, a su forma sistemática de obtener y organizar todo entorno a una arquitectura de dominio, consiguiéndose así un incremento en la productividad, en la competitividad en el mercado y en la calidad de los productos software finales.

3.1. Definición de línea de productos

Ya se ha visto en una primera aproximación que una línea de productos es una la arquitectura que comparten una serie de productos software relacionados. Informalmente una línea de productos se puede ver como un conjunto de productos que están estrechamente relacionados (por su funcionalidad), que son vendidos a los mismos grupos de compradores, que son comercializados a través del mismo tipo de distribución, o que caen en el mismo rango de precios. Para encontrar una definición más adecuada al marco de la reutilización se puede recurrir a la bibliografía especializada, donde existen diversas definiciones, algunas de las cuales se recogen a continuación.

- Grupo o familia de productos relacionados realizados por el mismo proceso y para el mismo propósito, diferenciándose sólo en el estilo, modelo o tamaño. Una línea de producto agrupa aplicaciones relacionadas en familias de aplicaciones tomando ventaja de los elementos comunes de la familia. Dado que los productos están relacionados (tienen funcionalidad o requisitos de usuario similares) hay un alto grado de aspectos comunes en una línea de producto [Sonnemann, 1995].
- Colección de productos software que recogen un conjunto de requisitos de sistema comunes, y están organizadas alrededor de una actividad específica de negocio [Cohen et al., 1995].
- Grupo de productos que comparten un conjunto de características comunes gestionadas, que satisfacen las necesidades específicas de un sector concreto del mercado [Bass et al., 1997].
- Una línea de productos consiste en una arquitectura de línea de productos y en un conjunto de elementos software reutilizables que han sido diseñados para su incorporación en la arquitectura de línea de productos. Adicionalmente, la línea de productos incluye los productos que han sido desarrollados utilizando los *assets* mencionados [Bosch, 2000a].
- Conjunto de productos que comparten un conjunto común de requisitos, pero que exhiben una variabilidad significativa en sus requisitos [Griss, 2000].
- Conjunto de sistemas software que comparten un conjunto de características común y gestionado, que satisface las necesidades específicas de un segmento de mercado y que son desarrollados a partir de un conjunto central de *assets* de una forma establecida [SEI, 2001].

En la Figura 2 se esquematiza de forma gráfica el concepto de línea de productos desde la perspectiva de sus *assets* principales, esto es, una arquitectura base, un conjunto de componentes y los productos resultantes.

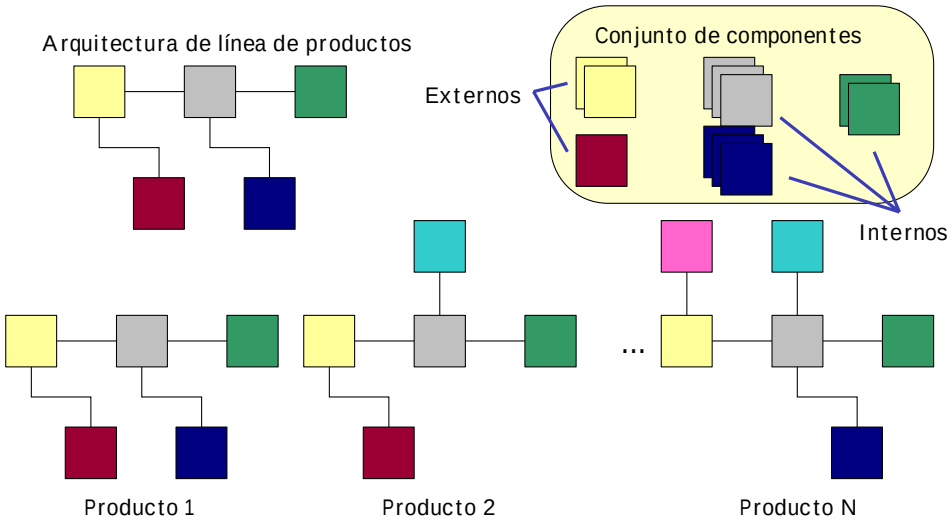


Figura 2. Esquema de los *assets* principales de una línea de productos [Bosch, 2000b]

Un concepto que está muy íntimamente relacionado con las líneas de productos es el de familia de productos, noción que ya ha aparecido directa o indirectamente en las definiciones anteriores. Así, una familia de productos se puede definir como: "el conjunto de productos diferentes que pueden ser producidos desde un diseño común, *assets* compartidos y mediante un proceso de ingeniería de aplicaciones. La pertenencia a este conjunto depende de la abstracción que unifica los *assets* en un sistema que funciona: una arquitectura, las reglas físicas o de negocio, o la plataforma hardware" o como "el conjunto de productos que comparten una plataforma común, pero tiene características y funcionalidad requeridas por el cliente".

Una línea de productos no necesita construirse como una familia de productos, aunque es la forma de obtener los mayores beneficios. Además, una familia de productos no necesita constituir una línea de productos si los productos resultantes tienen poco en común en términos de mercado o de sus características.

Una línea de productos no es un grupo de productos producidos por una única unidad de negocio. Puede esperarse que haya una gran correlación entre las líneas de productos y las unidades de negocio, pero conceptualmente son cosas distintas. Una unidad de negocio se forma por razones de organización o financieras y puede ser responsable de una o más líneas de productos. Mientras que una línea de productos comparten y gestionan un conjunto común de características que satisfacen las necesidades específicas de un mercado seleccionado. Teóricamente, líneas de productos provenientes de diferentes unidades de negocio pueden mezclarse para formar una "super línea de productos".

Siguiendo con las diferencias, una línea de productos tampoco es sinónimo de dominio. Mientras que un dominio es un cuerpo especializado de conocimiento, un área de experiencia o una colección de funcionalidad relacionada, la línea de productos hace referencia a los sistemas software que comparten las características particulares de un sector de mercado. Por ejemplo, en el caso de las telecomunicaciones, el dominio de las telecomunicaciones es el conjunto de problemas relacionados con las telecomunicaciones, que puede estar formado por otros subdominios tales como protocolos, telefonía, redes... Por otro lado, una línea de productos de telecomunicaciones es un conjunto específico de sistemas que tratan algunos de esos problemas.

3.2. Beneficios y costes de una línea de productos

Las líneas de productos potencian la reutilización estratégica. Los *assets* componentes de una línea de productos van más allá de una mera reutilización de código. Cada producto de la línea

de productos toma la ventaja del análisis, diseño, implementación, planificación, prueba... realizados en cada uno de los productos desarrollados previamente en la línea de productos.

Las líneas de productos tienen su origen en las escuelas de negocio en la década de los ochenta, con un claro objetivo económico mediante el desarrollo sinérgico de productos [Knauber y Succi, 2001]. Diversos beneficios se derivan de una estrategia basada en líneas de productos, tales como la reducción de costes, el descenso del tiempo de mercado, y la mejora en la calidad. Además, se pueden esperar beneficios no puramente técnicos como resultado de la calificación de productos y de la compartición de costes.

Sin embargo, para cada posible beneficio derivado de la reutilización existe un coste asociado. En [Clements et al., 1998] se presentan los elementos que afectan tanto a la línea de productos como a los nuevos productos, junto a los beneficios y los costes asociados.

Asset	Beneficio	Costes
Arquitectura, especificación de la arquitectura y evaluación de la arquitectura	La arquitectura representa una inversión de diseño para la organización. Llevar esta inversión a todos los productos de la línea de productos implica que la actividad más importante del diseño esté prácticamente realizada	La arquitectura debe soportar las variaciones inherentes a la línea de productos, que imponen restricciones adicionales
Componentes software	Los elementos derivados de la arquitectura software se comparten por los diferentes productos de la línea de productos	Los componentes deben ser diseñados de forma robusta para que puedan aplicarse en diferentes contextos, lo que complica su diseño. Con frecuencia deben diseñarse de forma más general, incluyendo los puntos de variación, sin pérdida de rendimiento
Análisis y modelado de rendimiento	Ventajas en la reutilización de análisis y modelos en el campo de los sistemas de tiempo real y sistemas distribuidos	La reutilización del análisis puede imponer restricciones sobre la movilidad de procesos entre procesadores, sobre la creación de nuevos procesos o sobre la sincronización entre procesos existentes
Pruebas, plan de pruebas, procesos de pruebas, casos de pruebas	Obtención de unidades de pruebas completas para los productos	La distinción entre la línea de productos y sus instancias significa que la solución de un problema puede tener repercusiones más allá del sistema concreto que fue corregido. Las pruebas deben considerar que todas las instancias de la línea de productos pueden verse afectadas por una modificación particular
Plan de proyecto	Los presupuestos y los calendarios pueden reutilizarse de proyectos previos. Además, con la experiencia previa éstos serán mejor estimados	En la práctica los beneficios derivados de la reutilización provocarán que cada producto instancia de la línea de productos tenga su propio presupuesto y calendario

Asset	Beneficio	Costes
Herramientas y procesos para el desarrollo de software, proceso para la realización de cambios	Herramientas del tipo de gestión de configuración, *workflows* se utilizan en todos los productos, ganándose en experiencia y amortizando su coste a través de la línea de productos completa	Todos estos elementos deben ser más robustos para diferenciar la gestión de la línea de productos de la gestión de un producto concreto
Personal, habilidades, formación	Debido a los elementos comunes entre las aplicaciones, se facilita la movilidad del personal entre proyectos. Su experiencia se puede aplicar a lo largo de toda la línea de productos. Su productividad crece de forma significativa	El personal debe ser formado más allá de las técnicas clásicas de ingeniería del software y de las técnicas corporativas para asegurar que entienden y pueden utilizar los *assets* asociados con la línea de productos. Deben crearse materiales de formación relacionados con la línea de productos. Debe producirse una transición controlada de las técnicas en ingeniería del software a las técnicas de ingeniería de dominio

Tabla 2. Beneficios y costes de las líneas de productos [Clements et al., 1998]

3.3. Descomposición de las líneas de productos

En [Bosch, 2000a] se describen tres dimensiones en las que se pueden descomponer los conceptos involucrados en las líneas de productos, estas dimensiones son:

1. Arquitectura, componente y sistema
2. Negocio, organización, proceso y tecnología
3. Desarrollo, instanciación y evolución

La primera dimensión divide el dominio de la línea de productos de acuerdo a sus *assets* principales que son parte del desarrollo basado en reutilización. Esta aproximación se presenta también en [Jacobson et al., 1997], recibiendo el nombre de *ingeniería de familia de aplicaciones, ingeniería del sistema de componentes* e *ingeniería de aplicación* respectivamente.

- **Arquitectura software:** La arquitectura software es *la estructura o estructuras de un sistema, consistente en unos componentes software, las propiedades externamente visibles de dichos componentes y en las relaciones entre ellos* [Bass et al., 1998b]. Es, por tanto, el principal *asset* de la línea de productos. La actividad principal es el diseño de una arquitectura software que de cobertura a todos los productos de la línea de productos e incluya todas las características que se comparten entre los productos.
- **Componente:** El segundo conjunto de *assets* está formado por los componentes que han sido identificados en la arquitectura. Deben reflejar la funcionalidad requerida, pero además deben soportar la variabilidad identificada en la arquitectura de la línea de producto. Son, generalmente, componentes de grano grueso, cercanos al concepto de *framework*.
- **Sistema:** El conjunto final de *assets* está formado por los sistemas construidos sobre la base de la arquitectura y los componentes de la línea de productos. Esta actividad requiere la adaptación de la arquitectura de la línea de producto para

ajustarse a la arquitectura de sistema, que puede requerir eliminar o añadir componentes o relaciones entre ellos, desarrollar extensiones a los componentes existentes, configurar los componentes y desarrollar elementos software específicos para el sistema.

La segunda dimensión está relacionada con las diferentes vistas de una organización. En diferentes reuniones de investigación, relacionadas con las líneas de productos, se ha utilizado la descomposición de esta dimensión [Clements, 1997], [Bass et al., 1997], [Bass et al., 1998a], [Bass et al., 1999], [Bass et al., 2000].

- **Negocio:** Todas las actividades que tienen lugar en una organización deben ser justificadas en términos de beneficios para el negocio. Con respecto a las líneas de productos, un aspecto sumamente importante es la más que considerable inversión que se requiere para convertir un proceso de desarrollo orientado a un producto cada vez, a un proceso de desarrollo que esté basado en la reutilización del software y en líneas de productos.
- **Organización:** En las aproximaciones más tradicionales a las líneas de productos se sugiere la creación en el organigrama de la organización de una unidad de ingeniería de dominio que sea la responsable del desarrollo y de la evolución de la arquitectura de la línea de productos y de sus *assets*. Existen alternativas que prescinden de esta unidad de ingeniería de dominio, pero siempre tienen repercusiones en la estructura de la organización al necesitarse personal cualificado y expertos en el dominio.
- **Proceso:** La existencia de una arquitectura de línea de productos y de un conjunto de *assets* relacionados con ella, conlleva considerables efectos en los procesos de desarrollo de un producto o un sistema. Adicionalmente, hay, entre otros, procesos específicos de la línea de productos, dedicados al diseño de la arquitectura base y de los *assets* propios de ésta.
- **Tecnología:** Se refiere a los aspectos tecnológicos que se utilizan para soportar el desarrollo del software para y con reutilización.

La tercera dimensión se centra en el ciclo de vida de cada uno de los *assets* de la organización.

- **Desarrollo:** Para cada *asset* principal existe un momento en el que es desarrollado inicialmente desde cero o basándose en una minería de otros *assets* existentes. Existen varios aspectos que son específicos a esta fase en el ciclo de vida de estos *assets*.
- **Instanciación:** La reutilización de una línea de productos no implica la mera duplicación de los *assets* involucrados. La arquitectura de la línea de productos necesita ser adaptada para convertirse en la arquitectura software del producto y los elementos reutilizables deben configurarse, adaptarse y extenderse para satisfacer los requisitos del producto concreto.
- **Evolución:** Los tres tipos de *assets* principales de una línea de productos evolucionan porque los requisitos de los productos evolucionan también. El cambio de los requisitos tiene repercusiones lógicas en los requisitos de la línea de productos, que implican su evolución, y esto conlleva la consiguiente evolución de la arquitectura y de los elementos reutilizables.

3.4. Actividades esenciales en una línea de productos

El concepto de línea de productos que utiliza una arquitectura base y un conjunto de elementos reutilizables es directamente aplicable a las organizaciones que desarrollan y comercializan productos o sistemas. También es posible obtener ventajas de la puesta en marcha de una línea de productos en organizaciones más orientadas al proyecto o incluso en los departamentos de informática que dan soporte a los sistemas informáticos de las organizaciones.

Existen tres puntos fundamentales para determinar la aplicabilidad de una aproximación basada en líneas de productos [Bosch, 2000a]:

1. La cantidad de elementos comunes entre los sistemas que se van a incluir en la potencial línea de productos.
2. La capacidad de negociación de los requisitos.
3. La propiedad de los *assets*.

En cualquiera de los casos las líneas de productos no aparecen de forma accidental, sino que requieren un esfuerzo considerable y explícito por parte de la organización interesada.

En esencia, una línea de productos involucra un desarrollo o adquisición de los *assets* base, fase normalmente conocida como ingeniería de dominio, y un desarrollo o adquisición de productos utilizando los *assets* base, fase más conocida como ingeniería de aplicación [Clements et al., 1998]. Existe una gran realimentación entre los *assets* base y los productos construidos con ellos, los *assets* base son renovados con los nuevos productos, subiendo el valor de los *assets* cuantas más veces son reutilizados. Como resultado, los *assets* base son realizados de forma más genérica para poder ampliar el rango de productos en los que ser reutilizados. De esta forma, se tiene que ambas fases son iterativas por naturaleza, como se refleja en la Figura 3, tanto en el desarrollo de los *assets* base como en la de los productos y en la coordinación de ambos.

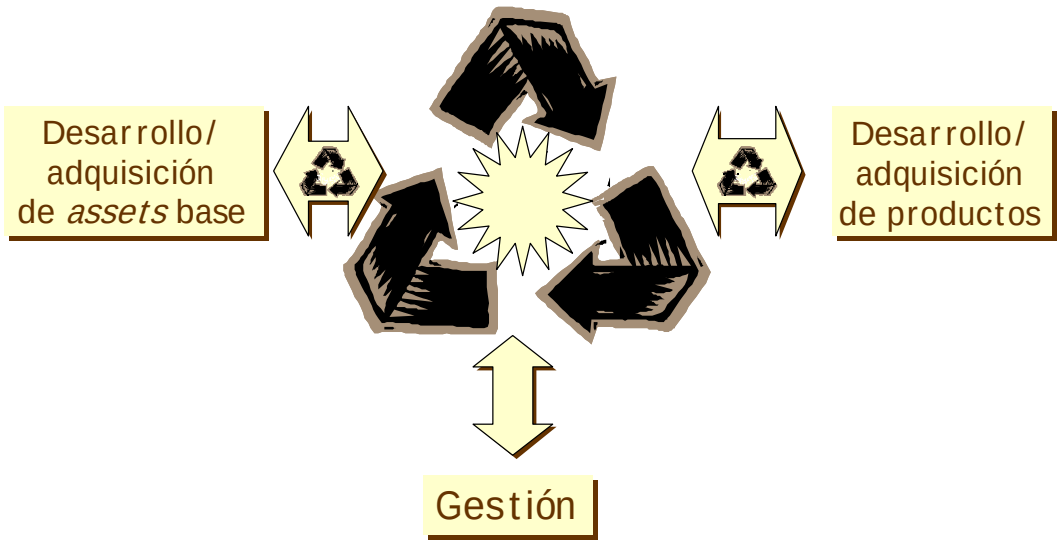


Figura 3. Iteración en las actividades de las líneas de productos

A la hora de poner en marcha una línea de productos la organización puede decantarse por una introducción de los procesos evolutiva o por una más revolucionaria. Además, la aproximación de línea de productos puede aplicarse a una familia de productos ya existente o a una nueva familia con la que la organización quiere expandir su mercado. Los posibles casos que se dan en la puesta en marcha de una línea de productos se recogen en la Tabla 3, cada uno de ellos lleva asociados unos riesgos y unos beneficios [Bosch, 2000a].

	Evolución	Revolución
Familia de productos existente	Desarrollo de una arquitectura de línea de productos basada en la arquitectura de la familia de productos. Desarrollo de componentes evolucionando componentes existentes.	La arquitectura de la línea de productos y los componentes se desarrollan basándose en un superconjunto de los requisitos de los miembros de la familia de productos y una serie de posibles requisitos futuro.
Línea de productos nueva	La arquitectura de la línea de productos y los componentes evolucionan sobre la base de los requisitos establecidos por los nuevos miembros	La arquitectura de la línea de productos y los componentes se desarrollan para cumplir los requisitos de todos los miembros esperados de la línea de productos.

Tabla 3. Posibilidades en la puesta en marcha de una línea de productos

El objetivo de la actividad de desarrollo y/o adquisición de los *assets* base es producir o adquirir [Clements et al., 1998], [SEI, 2001]:

1. *El espacio del producto:* Es la descripción de los productos iniciales que componen la línea de productos. Esta salida consiste en la descripción del espacio de productos que la línea de productos es capaz de incluir.
2. *Assets base:* Son la base para la producción de productos en la línea de productos. Incluye la arquitectura de la línea de productos y los componentes que son diseñados para su reutilización sistemática a través de la línea de productos.
3. *Plan de producción:* Describe cómo los productos son construidos a partir de los *assets* base.

Las entradas para esta actividad son:

1. *Restricciones del producto:* ¿Cuáles son los elementos comunes y variantes entre los productos que constituyen la línea de productos? ¿Cuáles son los requisitos de comportamiento y de calidad? ¿Qué características serán beneficiosas en un futuro?
2. *Restricciones de la producción:* ¿Qué estándares son necesarios aplicar en los productos de la línea de producción? ¿Qué componentes comerciales deben usarse? ¿Qué componentes heredados pueden ser reutilizados?
3. *Estilos, patrones y Frameworks:* ¿Cuáles son los bloques arquitectónicos relevantes que pueden aplicarse?
4. *Estrategia de producción:* ¿Construcción descendente o ascendente? ¿Cómo van a ser repartidos los costes? ¿Los productos se generarán de forma automática o se compondrán?
5. *Inventario de assets pre-existentes:* ¿Qué *assets* de la organización pueden utilizarse?

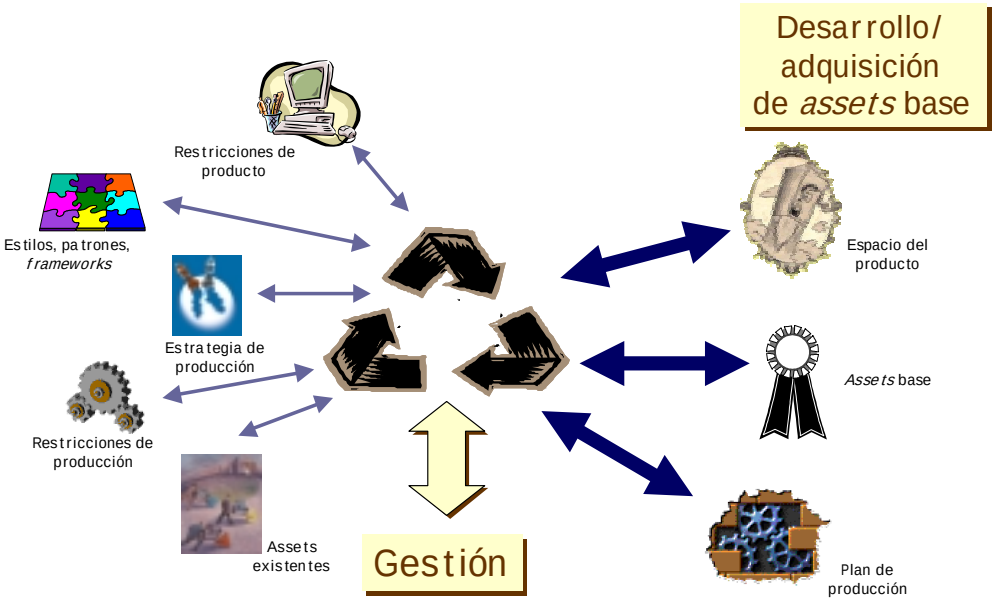


Figura 4. Entradas y salidas en el desarrollo/adquisición de los *assets* base

La actividad del desarrollo de productos es el objetivo último, los *assets* base son desarrollados como el camino para llegar a este fin. Esta actividad toma como entradas las salidas de la actividad de desarrollo/adquisición de *assets* base (espacio del producto, *assets* base y plan de producción), más los requisitos de los productos individuales, como se muestra en la Figura 5.

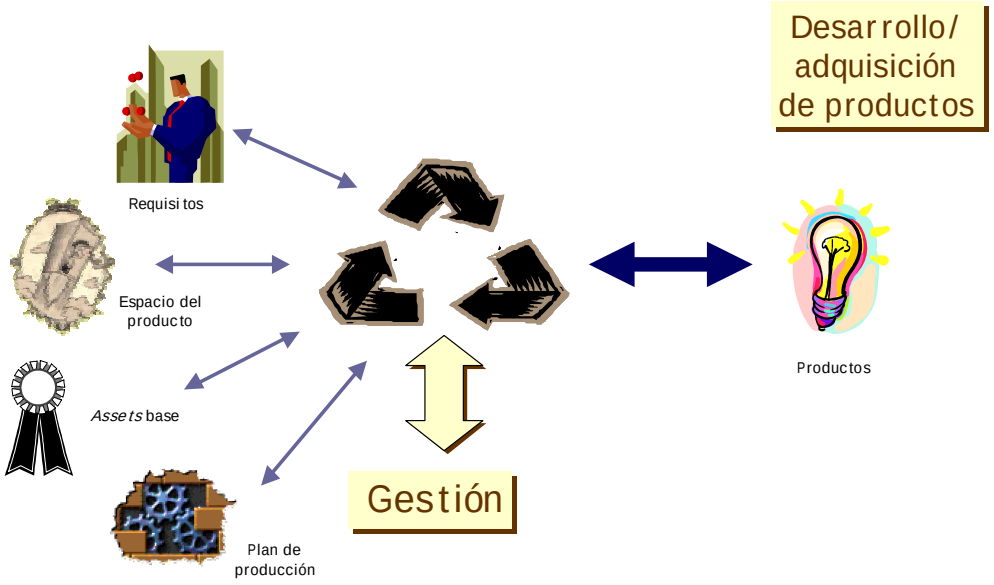


Figura 5. Entradas y salidas en el desarrollo/adquisición de productos

La gestión juega un papel fundamental para el establecimiento de una línea de productos. Las actividades deben ser abastecidas de los recursos necesarios, coordinadas y supervisadas. En el área de la gestión se incluyen, entre otros, la planificación, la consecución de la estructura de organización correcta, la fundación, la construcción y comunicación de los casos de negocio, la formación, la gestión de riesgos, las operaciones, el desarrollo y comunicación de una estrategia de adquisición, la gestión de la interfaz entre el cliente y el suministrador...

3.5. Elementos software de una línea de productos

Como resultado de los procesos descritos en la sección previa, se desarrollan varios elementos software. Éstos se dividen en dos categorías principales: los elementos compartidos por los productos miembros de la línea de productos y los elementos específicos de los productos [Bosch, 2000a], como se refleja en la Figura 6.

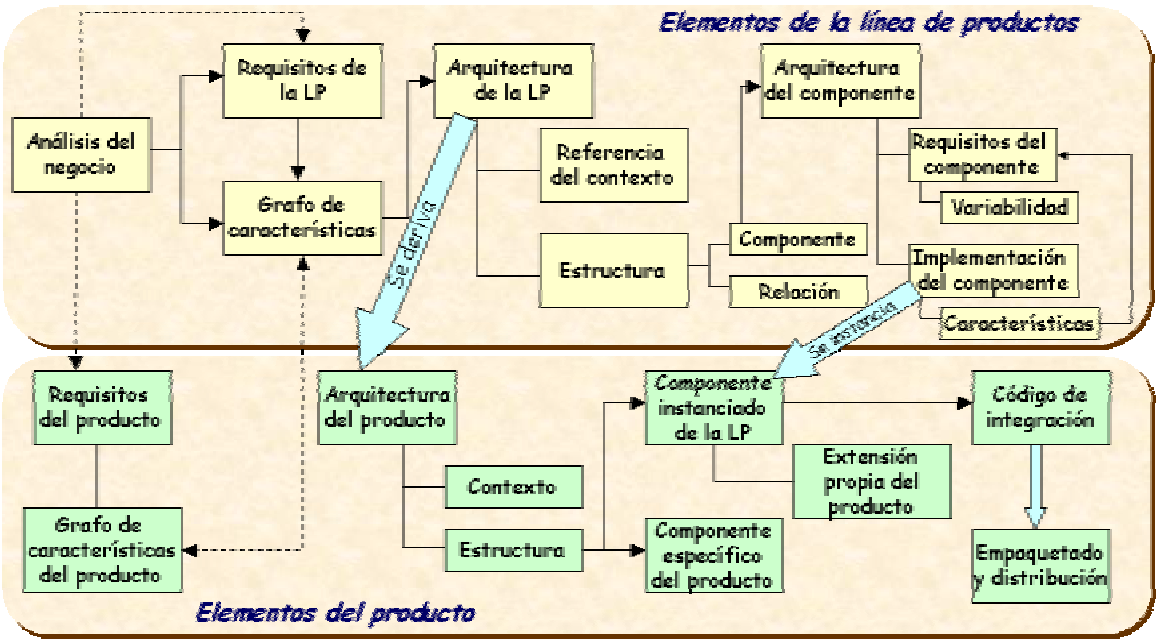


Figura 6. Elementos software de una línea de productos [Bosch, 2000a]

Dentro de los elementos software que configuran la línea de productos se incluyen los resultados de un análisis del negocio, los requisitos de la línea de productos, un grafo de características (*features*), una arquitectura de línea de productos, una o más referencias de contexto y la estructura de la arquitectura software expresada en término de sus componentes y sus relaciones. Cada componente presenta unos requisitos y, opcionalmente, una o varias implementaciones. Por cada implementación de un componente tiene asociada una serie de características que describen su adecuación para los productos particulares.

Como elementos software propios de los productos concretos de la línea de productos se pueden destacar los requisitos del producto, un grafo de características del producto, la arquitectura software del producto (derivada de la arquitectura de la línea de productos) que incluye la descripción de su contexto y su estructura, uno o más instanciaciones de las implementaciones de los componentes y, opcionalmente, uno o más componentes específicos para el producto. Por último, el producto final se completa con el código necesario para la integración del producto, su empaquetamiento y formato de distribución.

4. Componentes

El desarrollo basado en componentes (CBD – *Component Based Development*) ofrece a las organizaciones una forma de desarrollo de soluciones software, a escala de empresa, flexibles y con facilidad de incluir nuevas demandas de una forma eficiente en cuanto al coste y al tiempo.

En el marco de trabajo que ofrece el CBD, las partes de una aplicación software se diseñan de acuerdo a unas especificaciones predefinidas, que pueden ensamblarse para dar lugar a aplicaciones completas. Estas partes se conocen como componentes.

Se puede definir CBD como el *enfoque al desarrollo del software en el que todos los elementos software – desde el código ejecutable a las especificaciones de las interfaces, arquitecturas y modelos de negocio; y a todas las escalas desde aplicaciones y sistemas completos hasta las partes más pequeñas – pueden ser construidas ensamblando, adaptando y conectando componentes existentes en una variedad de configuraciones* [D'Souza y Wills, 1999].

Otro término que se utiliza para referirse a las actividades relacionadas con el desarrollo basado en componentes es CBSE (*Component-Based Software Engineering*). Concierne al ensamblado rápido de sistemas a partir de componentes donde: a) los componentes y frameworks tienen propiedades certificadas; y b) estas propiedades certificadas ofrecen las bases para predecir las propiedades de los sistemas construidos con los componentes [Bachman et al., 2000].

Existen diversos hechos significativos que indican una tendencia muy considerable hacia una ingeniería del software basada en componentes [Brown, 1999]:

- Un empuje económico que lleva muchas organizaciones al uso de soluciones comerciales a gran escala. En el mercado de las tecnologías de la información esto se ha reflejado en la subcontrata de servicios informáticos y en las ventajas aparentes de utilizar aplicaciones comerciales en lugar de aplicaciones software realizadas a medida. Ejemplos de esta situación son SAP o BAAN. Esto ha conducido a la popularización de los componentes comerciales, ya sean para el público en general, más conocidos como COTS (*Commercial Off-The-Shelf*), o para uso exclusivo del gobierno, conocidos como GOTS (*Government Off-The-Shelf*) [Oberndorf, 1998].
- Como resultado, muchas aplicaciones software combinan paquetes comerciales (típicamente con adaptaciones y configuraciones locales), datos legados, soluciones propias y código de integración. La única forma de diseñar, ensamblar y mantener estas aplicaciones es considerar todo ello como colecciones de piezas modeladas como componentes capaz de interaccionar.
- El estilo y arquitectura de las aplicaciones ha cambiado de forma muy significativa. Se ha pasado de aplicaciones centralizadas a aplicaciones distribuidas y multicapa, que son remotamente accesibles desde una gran variedad de plataformas cliente a través de *intranets* o de Internet. Las arquitecturas software, una vez definidas, especifican los componentes que las conforman, y para cada componente es necesario establecer sus interfaces y sus atributos de calidad.
- Desde el punto de vista tecnológico se cuenta con un amplio soporte que cuenta con la estandarización de varios modelos de componentes. Así, en primer lugar cabe destacar la importancia del paradigma orientado a objetos como la base sobre la que se ha construido la mayor parte de la tecnología de componentes[1]. Como iniciativas de relevancia dentro de los modelos de componentes cabe destacar las aportaciones de Microsoft con COM (*Component Object Model*) [Microsoft, 1995], DCOM (*Distributed Component Object Model*) [Thai y Oram, 1999] y el *framework* .NET [Microsoft, 2001], de SUN con los EJB (*Enterprise JavaBeans*) [SUN, 2000], o de OMG (*Object Management Group*) con el CCM (*CORBA Component Model*) [OMG, 1999] o del W3C (*World Wide Web Consortium*) con XML (*eXtensible Markup Language*) [Bray et al., 2000].

[1] Lo cual lleva a veces a confundir los términos objeto y componente. Las diferencias entre ambos conceptos quedan plasmadas en [Szyperski, 2000].

Hay dos factores fundamentales en el desarrollo basado en componentes: la **reutilización** – la facilidad para reutilizar componentes existentes en la creación de sistemas más complejos – y la **evolución** – crear un sistema altamente formado por componentes facilita su mantenimiento, ya que si el diseño es correcto los cambios estarán muy localizados y los efectos tendrán poca o ninguna propagación en el resto de los componentes del sistema.

Los dos factores mencionados imponen una serie de condicionantes. En primer lugar, deben existir componentes para su reutilización. Debe existir una provisión de componentes bien diseñados y listos para su utilización, que sean fácilmente localizables, adquiribles y utilizables. Además, debe existir un modelo de componentes que soporte el ensamblado y la interacción de componentes, esto es, debe existir un marco de referencia estándar en el que los componentes puedan existir y comunicarse. Finalmente, es necesario un proceso y unas arquitecturas que den soporte al CBD.

Centrando la atención en la faceta de reutilización se puede distinguir tres tipos de reutilización de componentes [Bosch, 2000a]:

- *Versiones de un sistema:* Es el nivel más bajo de reutilización es utilizar componentes sobre una serie de versiones de un sistema. El componente únicamente necesita cumplir los requisitos del producto y estar preparado para los futuros requisitos del producto. Este tipo de reutilización no es considerada como tal por algunos autores, como por ejemplo J. S. Poulin [Poulin, 1997].
- *Línea de productos software:* El segundo nivel de reutilización es donde los componentes son utilizados en una serie de versiones de productos y en una familia de productos que contienen funcionalidad relacionada, pero no idéntica. Un componente en este contexto debe estar preparado para incorporar fácilmente los nuevos requisitos de las siguientes versiones, pero además cubrir las diferencias en los requisitos de los componentes existentes entre varios productos. Esta propiedad recibe el nombre de variabilidad.
- *Componentes para terceros:* Es el nivel más alto de reutilización porque involucra versiones de productos, familias de productos y diferentes organizaciones. A este tipo de componentes se les suele conocer como COTS.

4.1. Definición de componente

La noción básica de componente como unidad elemental de composición para la construcción de aplicaciones software mediante su adaptación y ensamblado es comúnmente aceptada. Sin embargo, existen diferentes enfoques de cómo deben ser dichas unidades, en parte por los diferentes contextos en los que la palabra componente se utiliza cotidianamente. Como destaca Hopkins [Hopkins, 2000]: "*Mientras que el concepto de componente software es conocido virtualmente desde los inicios de la ingeniería del software, la problemática y aspectos prácticos relativos a los mismos han evolucionado continuamente a lo largo del tiempo*".

A continuación se recogen diferentes definiciones del término *componente*, aunque siempre desde la perspectiva del desarrollo basado en componentes y en relación con la reutilización del software.

- Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio [Szyperski y Pfister, 1997].
- Un componente es una unidad de composición con sus interfaces contractuales especificadas y únicamente con dependencias explícitas del contexto. Un

componente software puede ser distribuido de forma independiente y puede ser objeto de composición por terceras partes [Weck et al., 1997].

- Un componente es un paquete software que ofrece servicios a través de sus interfaces [Microsoft, 1997].
- Un conjunto de componentes atómicos simultáneamente distribuidos. Un componente atómico es un 'módulo' más un conjunto de 'recursos'. Un módulo es un conjunto de clases y posiblemente otras construcciones no orientadas al objeto, tales como procedimientos o funciones. Un recurso es una colección estática de elementos tipados (que parametrizan al componente) [Szyperski, 1998].
- Un componente debe cumplir los criterios siguientes [Meyer, 1999]:
    - Puede ser usado por otros elementos software.
    - Puede ser usado por clientes sin intervención del desarrollador del componente.
    - Incluye una especificación de todas sus dependencias.
    - Incluye una especificación de la funcionalidad que él ofrece.
    - Es usable sólo en la base de sus especificaciones.
    - Se puede componer con otros componentes.
    - Se puede integrar en un sistema de forma rápida y sin conflictos.
- Un componente es un empaquetamiento coherente de elementos software que pueden ser desarrollados independientemente y distribuidos como una unidad que puede ser integrada, sin necesidad de ser modificada, con otros componentes para construir sistemas más grandes [D'Souza y Wills, 1999].
- Un componente de código es un empaquetamiento coherente de implementación software que: a) puede ser desarrollado y distribuido independientemente, b) tiene interfaces explícitas y bien definidas para los servicios que ofrece, c) tiene interfaces explícitas y bien definidas para los servicios que espera de otros, y d) puede ser compuesto con otros componentes, quizás personalizando algunas de sus propiedades, pero sin modificar el componente en sí [D'Souza y Wills, 1999].
- Un componente es una implementación opaca de funcionalidad, un sujeto para la composición de terceras partes y cumple un modelo de componente [Bachman et al., 2000].
- Un componente software es una unidad de composición que ofrece interfaces explícitamente especificadas, requiere de unas interfaces e incluye unas interfaces de configuración, incluyendo además atributos de calidad [Bosch, 2000a].
- Paquete de servicios software que es neutral al lenguaje e independientemente implementado, que se distribuye en un contenedor encapsulado e intercambiable y al que se puede acceder mediante una o más interfaces públicas. Aunque un componente puede tener la capacidad de modificar una base de datos, no debe esperarse que mantenga información de estado. Un componente no está restringido por una plataforma ni está limitado por una aplicación [Sparling, 2000].
- Un componente CCM es una entidad binaria, de despliegue, con una interfaz bien definida que especifica los servicios que ofrece y los requisitos que necesita de otros componentes (protocolo *provide/uses*), diseñada y desarrollada para la composición con otros componentes [Sevilla, 2000].

- Un componente software es un empaquetamiento físico de software ejecutable con una interfaz bien definida y publicada [Hopkins, 2000].
- Parte modular, distribuible e intercambiable de un sistema, que encapsula implementación y presenta un conjunto de interfaces. Un componente está especificado típicamente por uno o más clasificadores (ejemplo, clases de implementación) que residen en él, y puede ser implementado por uno o más elementos software (ejemplo, ficheros binarios, ejecutables o de *script*) [OMG, 2001].

De las definiciones anteriores se desprende una gran controversia en el concepto de componente, normalmente suscitada por las restricciones que imponen algunas de ellas, como el caso de que tengan que ser binarios o que sólo sirvan para ser compuestos por terceros. Quizás la definición de [Szyperski y Pfister, 1997] sea de las más citadas y extendidas, aunque es difícil de comprender en su totalidad, así como de analizar en todas sus repercusiones (para un análisis detallado de esta definición se recomienda la consulta de [Szyperski, 2000]).

Reflejo de la confusión reinante se tiene la discusión mantenida en la columna *Beyond Objetcs* de la revista *Software Development* entre Bertrand Meyer y Clements Szyperski. Sin embargo, si se toma como base la definición de los siete criterios de Meyer [Meyer, 1999], éstos pueden resumirse en cinco, existiendo gran afinidad con la definición de Szyperski [Szyperski y Pfister, 1997]. En la Tabla 4 se presentan los puntos de la definición de Meyer, y al lado la misma idea utilizando el vocabulario de Szyperski.

Bertrand Meyer	Clements Szyperski
Puede ser usado por clientes sin que éstos necesiten la intervención del desarrollador	Independencia tiempo-espacio
Debe incluir especificación de sus dependencias: hardware, software, versiones y otros componentes	Conjunto de requisitos
Debe incluir la especificación de las funcionalidades que ofrece y será utilizable en base únicamente a esta especificación	Conjunto de interfaces
Componible con otros componentes	Unidad de composición
Integración en un sistema de forma rápida y sin conflictos	Incorporación al sistema

Tabla 4. Comparación entre las definiciones de Meyer y Szyperski [García-Molina et al., 2000]

En otros trabajos las características de un componente se resumen agrupándolas desde tres perspectivas [Cheesman, 1998], [Brown, 1999]:

- *Perspectiva de empaquetamiento:* El componente se considera como una unidad de empaquetamiento y distribución. Es un concepto de organización, centrado en la identificación de un conjunto de elementos que pueden ser reutilizados como una unidad. El énfasis está puesto en la reutilización. Se pueden distinguir varios tipos de elementos software que pueden ser considerados como un componente. Un tipo especial de componente, bajo esta perspectiva, es aquél que está pensado para distribuirse en un formato binario.
- *Perspectiva de servicio:* Considera al componente como una entidad que ofrece servicios a sus clientes (ver la definición de componente de [Microsoft, 1997]). El diseño e implementación de aplicaciones involucra comprender como una colección de componentes va a colaborar intercambiando peticiones de servicios. Bajo esta perspectiva se remarca la importancia de los contratos entre los componentes. Los servicios se agrupan en unidades contractuales coherentes, conocidas como interfaces. La interfaz sirve a la metáfora del contrato en el sentido de que la interfaz describe todo lo que un cliente potencial espera encontrar de los

servicios del componente, y además es la única manera que tienen los clientes de acceder a los servicios. La perspectiva de servicio presenta una noción lógica del componente porque es el diseñador el que decide como concretar la funcionalidad buscada en un conjunto representativo de componentes de servicio.

- *Perspectiva de integridad:* Con la perspectiva de servicio no se consigue ver al componente como una unidad reemplazable, cuando lo que se busca es llegar a un componente independiente, esto es, a una unidad reemplazable de comportamiento. La perspectiva de integridad define al componente como una cápsula de implementación, que encierra el software que de forma colectiva mantiene la integridad de los datos que en él se gestionan, y, por tanto, es independiente de la implementación de otros componentes. Este criterio es necesario para conseguir la sustitución de los componentes.

Desde un punto de vista arquitectónico, muy ligado con la granularidad de los componentes software, se pueden identificar dos tendencias claramente diferenciadas.

- En primer lugar está la perspectiva más cercana al nivel de abstracción de implementación, donde la mayor preocupación está en como construir, empaquetar y distribuir los componentes conformando algún modelo de componentes en concreto. Se busca construir un mercado de componentes que implementen una funcionalidad y una coordinación de forma única para los productos software.
- Por otro lado, se presentan los componentes como abstracciones arquitectónicas. Se requieren para implementar una o más interfaces que prescriben cómo los componentes pueden interactuar con otras restricciones arquitectónicas. Este es un enfoque más próximo al contexto definido por las líneas de productos, donde los componentes pueden ser de un grano mucho mayor, utilizando *frameworks* orientados a objetos como los componentes de una línea de productos, donde, gracias al alto nivel de configuración que tienen los *frameworks*, los componentes ofrecen una funcionalidad mucho más abierta que se puede aprovechar en los diferentes productos que poblarán la familia de productos.

En resumen, se distinguen tres partes esenciales en un componente: la interfaz, la implementación y su forma de distribución. Además, un componente debe poseer cuatro propiedades básicas: estar encapsulado, ser descriptivo, ser reemplazable y ser extensible [McInnis, 2000].

4.2. Interfaces de un componente

Los componentes no son elementos aislados, sino que interactúan con su entorno, estando su entorno formado por otros componentes. Los componentes invocan a otros componentes con peticiones de datos o de servicios. Sin embargo, los componentes se desarrollan de forma independiente con la intención de ser combinados con otros. Se requiere que las implementaciones de los componentes no estén relacionadas entre sí. Esto conduce a un interesante problema: por una parte se necesita que los componentes interactúen y por otro lado que sean independientes entre sí.

La solución está en el uso de las interfaces. Una interfaz define un contrato entre un componente que requiere una funcionalidad determinada y otro componente que ofrece dicha funcionalidad. La interfaz representa una especificación de la funcionalidad que debe ser accesible a través de ella. La especificación de la interfaz es independiente del componente o componentes que la implementan.

Según [Bosch, 2000a] se identifican tres tipos de interfaces en un componente: las interfaces de los servicios que ofrece, las interfaces de los servicios que requiere y las interfaces de configuración.

4.2.1 Interfaces de los servicios ofrecidos

Durante el diseño de una arquitectura se identifican los componentes que la forman y las relaciones entre ellos. De esta situación se desprende que frecuentemente un componente se relaciona con dos o más componentes. La relación de un componente con otro puede representar la interfaz de un servicio requerido, la interfaz de un servicio ofrecido o una combinación de ambas.

De acuerdo con esto, un componente puede presentar más de una interfaz de los servicios que ofrece. Cada relación con otro componente donde el primero ofrece una funcionalidad al segundo debe representarse como una interfaz de servicio ofrecido. Así, por cada tipo de interacción debe haber una interfaz de servicio ofrecido, aunque varios componentes puedan hacer uso de la misma interfaz.

Una interfaz está compuesta de un identificador, una lista de operaciones y una lista de identificadores de interfaz. Las operaciones de la interfaz pueden utilizar tipos base o identificadores de interfaz como argumentos formales y tipos de retorno.

En conclusión, un componente debe tener al menos una interfaz, pero potencialmente más interfaces, de servicios ofrecidos. Estas interfaces publican identificadores de interfaz que se refieren a otras interfaces definidas, pero un componente sólo puede publicar los identificadores de interfaz que él define internamente o que obtiene mediante una interfaz de servicio requerido.

4.2.2 Interfaces de servicios requeridos

Cada interfaz de servicio ofrecido por un componente es enlazada con una o más interfaces de servicios requeridos. La relación entre las interfaces ofrecidas y las requeridas no es de una a una, dado que una interfaz de servicios ofrecidos puede servir a varios componentes.

Una interfaz de servicios requeridos tiene los mismos elementos que una interfaz de servicios ofrecidos, con la diferencia de que la interfaz es requerida en lugar de ser ofrecida por el componente.

Como las interfaces se definen de forma independiente de los de componentes, una interfaz requerida se especifica referenciando únicamente el identificador de interfaz.

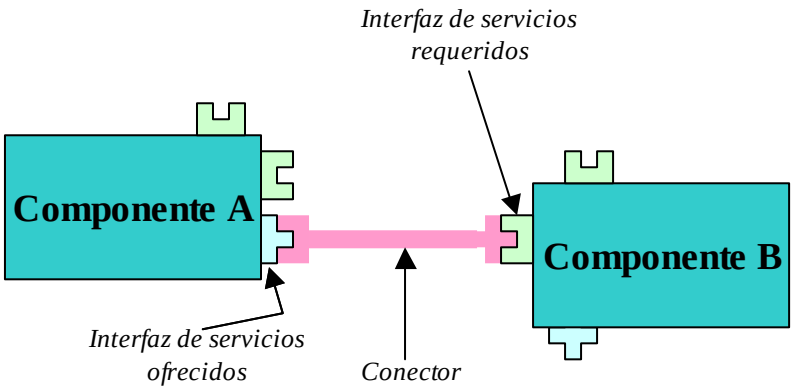


Figura 7. Interfaces y conectores [Bosch, 2000a]

Cuando se utilizan los componentes en una aplicación o en un producto, puede haber malos emparejamientos sintácticos entre las interfaces requeridas y ofrecidas, aunque la semántica de las interfaces coincida. Esto requiere la adaptación de al menos uno de los componentes, o bien

adaptar el conector que se utiliza entre los componentes. Gráficamente esto se puede apreciar en la Figura 7.

4.2.3 Interfaces de configuración

Si un componente contiene un conjunto de puntos de variación, esto es, puntos donde se puede modificar el comportamiento del componente. A cada punto de variación se le asocia una interfaz de configuración, que ofrece un punto de acceso para que el usuario del componente configure la instancia del componente a sus necesidades.

Para cada punto de variación, se emplea un mecanismo para su configuración. Algunos de los mecanismos más frecuentemente usados son:

- *Herencia:* Se asume que el componente se implementa como un conjunto de clases en un lenguaje orientado a objetos, y la herencia se utiliza como una técnica de caja blanca para especializar el componente para el contexto de su utilización. La herencia tiene una serie de desventajas, especialmente debido al hecho de que la funcionalidad que pertenece a una entidad lógica se encuentra dividida en múltiples componentes, lo que complica la comprensión y prueba de los componentes.
- *Extensiones:* Una extensión es un punto de variación donde el usuario del componente ofrece una de varias variante para un comportamiento. El patrón de estrategia [Gamma et al.,, 1995] es un claro ejemplo del uso de extensiones. La idea subyacente es separar la funcionalidad que puede variar de la que es estable, modelándose la funcionalidad variable como una entidad independiente.
- *Configuración:* Se incluyen todas las variantes y todos los puntos de variación en el componente y se ofrece una interfaz al usuario del componente. Esta interfaz permite al usuario establecer los parámetros adecuados para seleccionar la variante adecuada para un punto de variación.
- *Instanciación de plantillas:* Los componentes pueden requerir ser configurados con tipos específicos. Una técnica útil para estos casos es la utilización de plantillas que pueden ser instanciadas con tipos particulares.
- *Generación:* Representa la noción de generador de componentes. El uso típico de esta aproximación es que el usuario del componente prepare una especificación en algún lenguaje (utilizando algún lenguaje específico del dominio por ejemplo). Esta especificación servirá de entrada a un generador o compilador de alto nivel, que traducirá la especificación a código fuente, por norma general, que se incorporará a la aplicación o al producto final.

Generalmente, una interfaz de configuración se compone de una parte de documentación y de una parte técnica. El balance entre las dos partes depende del mecanismo de configuración que se utilice.

4.3. Modelo de componentes

El concepto de modelo de componentes es otro de los que ofrece cierta confusión en la bibliografía, especialmente en su relación con la noción de *framework de componentes*[2]. Así, por ejemplo W. Weck distingue entre ambos conceptos, utilizando el término *framework* de componentes para referirse a los estándares y convenios que pueden o no incluir servicios de

[2] En este contexto *framework* debe tomarse como marco de trabajo para los componentes, en lugar de como *framework* orientado a objetos.

soporte [Weck, 1997]. En [D'Souza y Wills, 1999] se presenta el concepto de *component kit* de forma similar al *framework* de componentes de [Weck, 1997], utilizándose el término modelo de componente para referirse a las interfaces y demás elementos de un componente individual, esto es, cada componente tiene su propio modelo de componente.

Salvando estas excepciones parece existir un consenso en que los sistemas basados en componentes se establecen sobre unos estándares y unos convenios bien definidos (modelo de componentes) y sobre una infraestructura de soporte (*framework* de componentes).

En resumen, un modelo de componentes especifica los estándares y convenios impuestos a los desarrolladores de componentes. Cumplir este modelo de componentes es una de las propiedades que distingue a los componentes de otras formas de software empaquetado [Bachman et al., 2000], mientras que un *framework* de componentes es una implementación de servicios que soporta o refuerza un modelo de componentes [Bachman et al., 2000].

Con la imposición de estándares se pretende conseguir una composición uniforme, unos atributos de calidad apropiados y una distribución de los componentes y aplicaciones. Con estas motivaciones los modelos de componentes imponen los siguientes estándares y convenciones [Bachman et al., 2000]:

- *Tipos de componentes:* Un tipo de componente puede definirse en términos de las interfaces que implementa. Si un componente implementa tres interfaces diferentes A, B y C, entonces es de tipo A, B y C. Además, un componente que implementa A, B y C es polimórfico con respecto a estos tipos, puede jugar el papel de A, B y C en diferentes ocasiones. Un modelo de componentes requiere que el componente implemente una o más interfaces, y en este sentido un modelo de componentes puede verse como la definición de uno o más tipos de componentes. Diferentes tipos de componentes pueden jugar distintos papeles en los sistemas software, y participar en diferentes tipos de esquemas de interacción.
- *Esquemas de interacción:* Los modelos de componentes especifican cómo se localiza a los componentes, qué protocolos de comunicación se usan y cómo se consigue la calidad de servicio (seguridad y transacciones). También puede describir cómo interactúan entre sí, o cómo interactúan con el *framework* de componentes. Los esquemas de interacción pueden ser comunes a todos los tipos de componentes o únicos para un tipo de componente particular.
- *Enlace con los recursos:* El proceso de composición es la forma de enlazar componentes con uno o más recursos. Un recurso es tanto un servicio ofrecido por un *framework* o por algún componente incluido en el *framework*. El modelo de componentes describe qué recursos están disponibles para los componentes, y cómo y cuándo los componentes se enlazan con dichos recursos. Un *framework* ve a los componentes como recursos que ha de manejar. Así, la distribución es el proceso por el que los *frameworks* son unidos a los componentes, y un modelo de componentes describe la forma en que los componentes son distribuidos.

4.3.1 *Framework* de componentes

Una analogía ampliamente utilizada para comprender el concepto de *framework* de componentes es verlo como un mini sistema operativo, siendo los componentes al *framework* lo que los procesos al sistema operativo.

El *framework* gestiona los recursos que comparten los componentes y ofrece los mecanismos subyacentes que permiten la comunicación e interacción entre los componentes. Estos *frameworks* están especializados para soportar un rango limitado de tipos de componentes y la interacción entre ellos.

La tendencia en la tecnología de componentes parece que camina hacia *frameworks* implementados independientemente de la implementación de los componentes. Algunos ejemplos pueden ser, la especificación de EJB que define un *framework* de *servidores* y *contenedores* que soportan el modelo de componentes de EJB, o el *framework* de Microsoft Visual Basic, especializado en la composición de componentes visuales, donde el *framework* es el intérprete para el lenguaje de *script* y la composición, todo ello acoplado con el modelo de componentes COM y los servicios de comunicación ofrecidos por el sistema operativo.

La consecución de un mercado robusto en el campo de los componentes software requiere de la existencia de modelos y *frameworks* de componentes estándares. Sin embargo, las experiencias reales han demostrado que diferentes dominios de aplicación tienen diferentes requisitos de rendimiento, seguridad, así como de otros atributos de calidad, lo que conduce a pensar en la necesidad de más de un estándar de modelo o *framework* de componentes. Las tecnologías que soportan la adaptación de los atributos de calidad de los *frameworks* pueden servir para reducir la necesidad de diversificar la oferta de *frameworks*. Por otro lado, las tecnologías que soportan la adaptación de los componentes a diferentes *frameworks* pueden reducir las consecuencias negativas de la fragmentación del mercado de *frameworks* [Bachman et al., 2000].

4.3.2 Patrón de diseño basado en componentes

El desarrollo de sistemas software basados en componentes es el resultado de la adopción de una estrategia de diseño basado en componentes. Una estrategia de diseño está muy próxima a un estilo arquitectónico, esto es, un patrón de diseño de alto nivel descrito por los tipos de componentes que soporta el sistema y sus patrones de interacción.

La tecnología de componentes refleja este patrón de diseño, que se muestra gráficamente en la Figura 8.

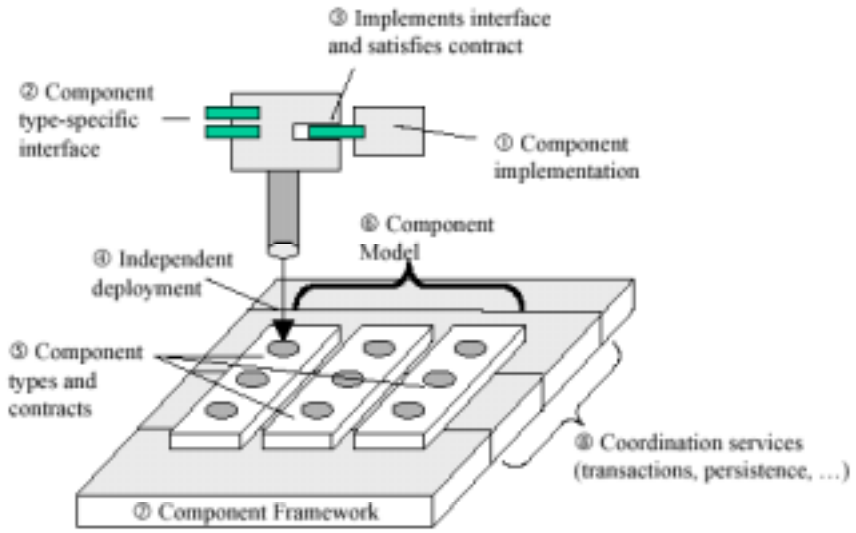


Figura 8. Patrón de diseño basado en componentes [Bachman et al., 2000]

Como se refleja en la Figura 8, un componente es una implementación software que puede ser ejecutada en un dispositivo físico o lógico (1). Un componente implementa una o más interfaces (2). Esto refleja que el componente satisface ciertas obligaciones, que se describen como contratos (3). Estas obligaciones contractuales aseguran que los componentes desarrollados de forma independiente obedecen ciertas reglas por lo que los componentes pueden interaccionar (o pueden no hacerlo) de forma predecible, y pueden ser distribuidos en entornos estándares (4). Un sistema basado en componentes se basa en un número pequeño de

tipos de componente, cada uno de los cuales juega un papel especializado en el sistema (5) y está descrito por una interfaz (2). Un modelo de componente (6) es el conjunto de tipos de componente, sus interfaces, y, adicionalmente, una especificación del patrón de interacción entre tipos de componentes. Un framework componente (7) ofrece una serie de servicios en tiempo de ejecución (8) para dar soporte y reforzar el modelo de componente. En muchos aspectos los frameworks de componentes son como sistemas operativos de propósito especial, aunque operando a un nivel de mucha mayor abstracción.

4.4. Composición

Composición es el término utilizado en el CBD para referirse a cómo los sistemas software se ensamblan. Los componentes ensamblados pueden interaccionar y, como ya se mencionó previamente, un modelo de componentes especifica los tipos de componentes y sus patrones de interacción.

Del modelo de referencia expuesto en la Figura 8 se distinguen dos clases de entidades que son compuestas: los componentes y los *frameworks*. Existen tres tipos principales de interacciones en los sistemas basados en componentes [Bachman et al., 2000]:

- **Componente – Componente (C – C):** Composición que posibilita la interacción entre componentes. Implica la funcionalidad de las aplicaciones, y por tanto los contratos que especifican estas interacciones pueden clasificarse como contratos de nivel de aplicación.
- ***Framework* – Componente (F – C):** Composición que posibilita la interacción entre un *framework* de componentes y sus componentes. Estas interacciones permiten a los *frameworks* gestionar los recursos de los componentes, y por tanto los contratos que especifican estas interacciones pueden clasificarse como contratos de nivel de sistema.
- ***Framework* – *Framework* (F – F):** Composición que permite las interacciones entre *frameworks*. Estas interacciones posibilitan la composición de componentes que se distribuyen en *frameworks* heterogéneos, y los contratos pueden clasificarse como contratos de interacción.

Los diferentes tipos de contratos que existen en la composición reciben el nombre genérico de formas de composición. En [Bachman et al., 2000] se distinguen seis formas de composición básicas:

- **Distribución de componentes:** Los componentes deben ser incluidos en un *framework* antes de ser compuestos o ejecutados. Los contratos de distribución (1) describen la interfaz que el componente debe implementar para que el *framework* pueda gestionar sus recursos

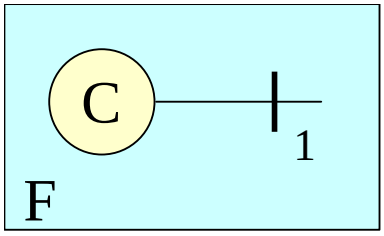


Figura 9. Distribución de componentes

- **Distribución de *frameworks*:** Los *frameworks* pueden ser distribuidos dentro de otros *frameworks*. Por ejemplo, la especificación de EJB lleva a la práctica parcialmente esta idea con los contenedores EJB incluidos en los servidores EJB. El contrato (1) es análogo al contrato expuesto en la distribución de componentes.

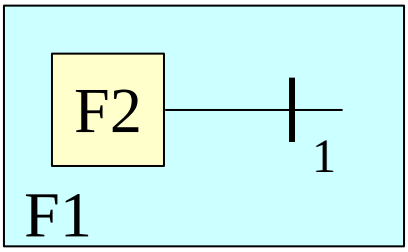


Figura 10. Distribución de *frameworks*

- **Composición simple:** Los componentes distribuidos en el mismo *framework* pueden ser compuestos. El contrato de composición (1) expresa funcionalidad específica del componente y de la aplicación. Los mecanismos de interacción para soportar el contrato los ofrece el *framework*.

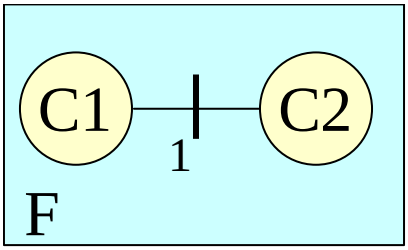


Figura 11. Composición simple

- **Composición heterogénea:** El soporte de *frameworks* por capas implica una composición de componentes a través de *frameworks*, ya sean éstos jerárquicos, como se ilustra en la Figura 12, o no. En cualquier caso se necesitan contratos de puente (1) a parte de los contratos de composición (2).

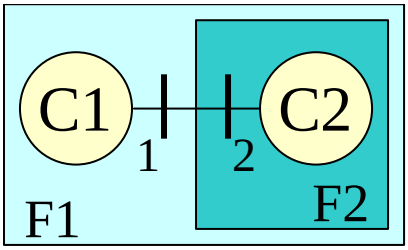


Figura 12. Composición heterogénea

- **Extensión del *framework*:** Los *frameworks* pueden tratarse como componentes, y pueden componerse con otros componentes. Esta forma de composición suele darse mediante la parametrización del comportamiento de los *frameworks* mediante *plug-ins*. Contratos estándares de *plug-ins* (1) para proveedores de servicios son muy comunes en los *frameworks* comerciales.

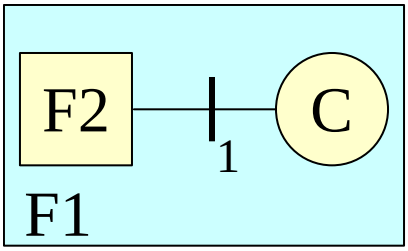


Figura 13. Extensión del *framework*

- **Composición transitiva:** Un componente puede ser a su vez un compuesto. La habilidad de predecir las propiedades de los componentes sugiere una capacidad similar para los componentes del componente. El contrato (1) se utiliza para componer C1 y C3, que contiene a su vez uno o más componentes. La cuestión que surge es si C2 es visible fuera de C3 y si puede ser distribuido independientemente.

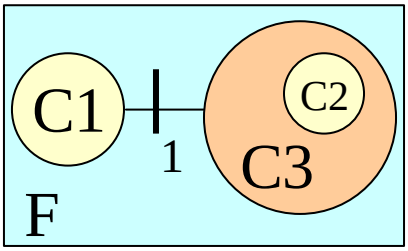


Figura 14. Composición transitiva

5. *Frameworks* orientados a objetos

Los *frameworks* orientados a objetos son el máximo exponente de la reutilización en las tecnologías de objetos. Éstos han sido utilizados con mucho éxito en diferentes ocasiones, aunque existen todavía grandes lagunas que hacen que sólo una mínima parte de los proyectos de construcción de *frameworks* acaben con éxito, principalmente debido a la complejidad y el esfuerzo que requiere su diseño y a que su construcción no tiene el soporte metodológico con el que cuenta la construcción de otros elementos software.

Los *frameworks* enlazan con la ingeniería de dominio con los llamados *frameworks* de aplicación o *frameworks* de dominio, donde se convierten en un tipo de componente especial, de un grano grueso, muy flexibles y adecuados para soportar la evolución, con lo que se convierten en unos componentes arquitectónicos idóneos para su aplicación en las líneas de productos, por más que inicialmente no se pensasen para este fin.

Los *frameworks* son una técnica de reutilización propia de la orientación a objetos. Comparten gran cantidad de características con las técnicas generales de reutilización del software, y potencian en gran medida las capacidades de reutilización que encierra el modelo de objetos, consiguiendo aumentar el ámbito de la reutilización de la pequeña escala que se consigue con la reutilización de unas clases, a una gran escala al facilitar la reutilización subsistemas o arquitecturas software.

Un *framework* asiste a los desarrolladores en la creación de soluciones a problemas dentro de un dominio y al mejor mantenimiento de dichas soluciones. Ofrece una infraestructura bien diseñada, de forma que cuando se crean nuevos elementos software, éstos pueden sustituir a otros elementos del *framework* con un mínimo impacto en el resto del *framework*.

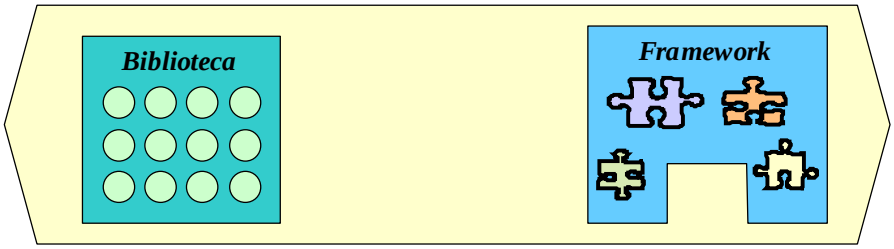
Un *framework* orientado a objetos ofrece un diseño abstracto y una implementación para un dominio particular. Las aplicaciones se construyen tomando el *framework* como base, y extendiéndolo con la funcionalidad propia de la aplicación. Un *framework* consta de una arquitectura, compuesta de un conjunto de clases abstractas y, posiblemente, clases concretas derivadas de las primeras que ofrecen implementaciones reutilizables.

Lo más interesante de un *framework* es que están descritas las interacciones que ocurren entre los objetos de las clases que lo componen, de forma que queda plasmada una solución genérica al conjunto de problemas de diseño soportados por el *framework*.

El diseño de un *framework* difiere en gran medida del diseño de una aplicación convencional.

- Los *frameworks* se emplean para ofrecer soluciones genéricas para un conjunto de problemas similares y relacionados dentro de un dominio, mientras que las aplicaciones dan una solución concreta para un problema concreto.
- Los *frameworks* son de naturaleza incompleta. Mientras que el diseño de una aplicación presenta todos los componentes que necesita, el diseño de un *framework* incluye puntos que requieren de su extensión e instanciación añadiendo soluciones concretas para el problema específico.
- Los *frameworks* no cubren toda la funcionalidad requerida por el dominio, sino que abstraen la funcionalidad común requerida por muchas aplicaciones, incorporando un diseño común y dejando que la variabilidad funcional sea incorporada por el usuario del *framework*[3].

Existen importantes diferencias entre un *framework* y una biblioteca de clases, aunque algunas bibliotecas muestran un comportamiento al estilo de los *frameworks*, y algunos *frameworks* se utilizan como si fueran bibliotecas de clases. Se puede ver esta situación como un intervalo donde las bibliotecas de clases tradicionales están en un extremo y los *frameworks* más sofisticados en el otro [Taligent, 1994], tal y como se refleja en la Figura 15.



- Conjunto de clases instanciadas por el cliente
- El cliente llama a las funciones
- Flujo de control no predefinido
- Interacción no predefinida
- No hay comportamiento por defecto

- Permite adaptación
- Llama a las funciones cliente
- Controla el flujo de la ejecución
- Define la interacción de objetos
- Ofrece un comportamiento por defecto

Figura 15. Diferencias entre una biblioteca de clases y un *framework* [Taligent, 1994]

En general, las mayores ventajas de la utilización de *frameworks* se pueden resumir en las tres siguientes [Taligent, 1993]:

- *Ofrecen una infraestructura y una guía arquitectónica:* Los *frameworks* no son una mera colección de clases. Ofrecen una funcionalidad muy rica y unas interacciones muy fuertes entre los objetos que dan la infraestructura a los usuarios del *framework*. Al ofrecer gran parte de la funcionalidad que necesita el usuario del *framework*, se reduce la necesidad de codificar y probar. Además, los *frameworks* ofrecen un diseño de gran calidad, guiando a sus usuarios para utilizar la tecnología de objetos de una manera más efectiva. Las aplicaciones realizadas con *frameworks* tienden a ser más mantenibles y reutilizables.
- *Ofrecen un mecanismo para extender la funcionalidad de una forma fiable:* Los *frameworks* ofrecen las interfaces para extender su funcionalidad a semejanza de

[3] Esto es generalmente cierto a lo largo del tiempo. Sin embargo, en un instante concreto, disponer de un buen *framework* de caja negra implica tener toda la funcionalidad disponible, habiendo "únicamente" que combinar las opciones.

cómo lo hacen las clases y los objetos, pero a un nivel de granularidad mucho mayor.

- *Reducen el mantenimiento:* Los cambios se localizan en lugares concretos, por lo que se minimiza la introducción de errores por parte de los usuarios del *framework*.

5.1. Estado del arte de los frameworks

El concepto de *framework* orientado a objetos fue desarrollado en la segunda mitad de la década de 1980. Los primeros ejemplos del concepto de *framework* que se encuentran en la bibliografía se tienen en el entorno de *Smalltalk* [Goldberg y Robson, 1989], con la aplicación del *framework* MVC (*Model-View-Controller* – Modelo-Vista-Controlador), y en *Apple Computer* [Schmucker, 1986].

Los *frameworks* se hicieron más populares cuando estuvieron disponibles algunos *frameworks* para la construcción de interfaces gráficas de usuario, como por ejemplo *Interviews* [Linton et al., 1989] o ET++ [Weinand et al., 1989].

Los *frameworks* no están limitados al área de las interfaces de usuario, encontrando representantes en muchos otros dominios, como los sistemas operativos [Russo, 1990], la supercomputación, con EPEE [Jezequel y Pacherie, 2000], los sistemas de fabricación, con SEMATECH [SEMATECH, 1998], [Doscher y Hodges, 2000] u OSEFA [Schmid, 2000], o los sistemas de alarmas [Molin y Ohlsson, 1998], por citar algunos.

Un hecho significativo en la historia de los *frameworks* fue la formación de *Taligent* en 1992. *Taligent* pretendía desarrollar un sistema operativo orientado a objetos basado en el concepto de *framework*. La compañía desarrolló, bajo el nombre de *CommonPoint*, un conjunto de herramientas para el desarrollo rápido de aplicaciones, que consistían en más de un centenar de *frameworks* orientados a objetos [Andert, 1994], [Cotter y Potel, 1995]. Con la aproximación de *Taligent* se cambia el centro de atención, puesto hasta entonces en los *frameworks* monolíticos, a varios *frameworks* de grano fino integrados.

Los trabajos relacionados con los *frameworks* están muy orientados a la captura de la información de un determinado dominio de aplicación, sin embargo existen menos aportaciones en campos generales relacionados con los *frameworks*, tales como métricas, métodos de uso o pruebas de *frameworks*.

En el campo de la documentación de *frameworks*, los patrones han sido el método más utilizado y recomendado [Johnson, 1992], [Hüni et al., 1995], así como para describir la lógica de diseño del propio *framework* [Beck y Cunningham, 1989].

En [Roberts y Johnson, 1996] se describe la evolución de un *framework* desde un *framework* de caja blanca, esto es un *framework* que se reutiliza fundamentalmente aplicando técnicas de herencia, hasta el desarrollo de una *framework* de caja negra, esto es un *framework* que se reutiliza fundamentalmente por parametrización y composición. La evolución se presenta como lenguaje de patrones que describen el proceso desde el diseño inicial del *framework* de caja blanca, hasta el *framework* de caja negra.

Otros trabajos relacionados con los *frameworks* son los que versan sobre la reestructuración (refactorización) de *frameworks*. Opdyke propone un conjunto de refactorizaciones destinadas a preservar el comportamiento, ayudando a eliminar múltiples copias de un código similar sin cambiar el comportamiento [Opdyke, 1992]. La refactorización puede utilizarse para la reestructuración de las jerarquía de herencia y de componentes [Johnson y Opdyke, 1993], [Opdyke y Johnson, 1993], [Crespo, 2000].

5.2. Definición de framework

La mayoría de los autores coinciden en que un *framework* orientado a objetos es una arquitectura software reutilizable que incluye diseño y código, pero no existe una definición generalmente aceptada, porque es difícil distinguir entre aspectos específicos de un framework o de una aplicación.

Algunos autores diferencian entre *framework* (arquitectura del *framework*) e implementación del *framework* (cualquier elemento software que soporte la implementación de aplicaciones basadas en el *framework*). Sin embargo, otros autores unen ambos conceptos bajo la noción de *framework.*

A continuación se recogen algunas de las definiciones más referenciadas en la bibliografía.

- Conjunto de clases que encierra un diseño abstracto para soluciones a una familia de problemas relacionados, soportando la reutilización con una granularidad mayor que las clases [Johnson y Foote, 1988].
- Conjunto de clases que cooperan y forman un diseño reutilizable para un tipo específico de software. Un *framework* ofrece una guía arquitectónica partiendo el diseño en clases abstractas y definiendo sus responsabilidades y sus colaboraciones. Un desarrollador personaliza el *framework* para una aplicación particular mediante herencia y composición de instancias de las clases del *framework* [Gamma et al., 1995].
- Conjunto de clases diseñadas para trabajar juntas en la resolución de un problema o para ofrecer facilidades [Sparks et al., 1996].
- Diseños reutilizables de todas las partes de un sistema software descrito por un conjunto de clases abstractas y la forma de colaborar de sus instancias [Roberts y Johnson, 1996].
- Infraestructura software que crea un entorno común para integrar aplicaciones e información compartida dentro de un dominio dado [SEMATECH, 1998].
- Diseño reutilizable de un sistema que describe cómo el sistema se descompone en un conjunto de objetos que interaccionan. Algunas veces el sistema es una aplicación completa; otras simplemente un subsistema. El *framework* describe tanto los objetos componentes como sus interacciones. Describe la interfaz de cada objeto y el flujo de control entre ellos. Describe cómo se reparten las responsabilidades del sistema entre los objetos [Fayad et al., 1998].
- Es una aplicación semicompleta que contiene componentes estáticos y dinámicos que pueden ser personalizados para obtener aplicaciones de usuario específicas [Fayad et al., 1999].

5.3. Tipos de frameworks

Hay varios tipos de *frameworks* en el mercado, desde *frameworks* de bajo nivel, que ofrecen servicios software básicos (comunicaciones, impresión, soporte al sistema de ficheros...), hasta *frameworks* de alto nivel muy especializados (interfaces de usuario, elementos software multimedia...). Sin embargo, aunque los principios subyacentes son independientes de los dominios en que se aplican, se puede hacer una clasificación de los *frameworks* por su ámbito [Fayad y Schmidt, 1997], [Fayad et al., 1998]:

- *Frameworks de infraestructura de sistema:* Su uso primario es el de simplificar el desarrollo de una infraestructura de sistema portable y eficiente, incluyendo sistema operativo, *frameworks* de comunicaciones y *frameworks* para interfaces de

usuario. Son utilizados internamente por las organizaciones, no vendiéndose a los clientes directamente.

- *Frameworks de soporte o middleware de integración:* Su utilidad es integrar aplicaciones y componentes distribuidos. Potencian la facilidad del software para ser dividido en módulos, reutilizado y fácilmente extendido. Ejemplos de este tipo de *frameworks* incluyen *middleware* orientado a mensajes y bases de datos transaccionales.
- *Frameworks de dominio o frameworks de aplicaciones de empresa:* Se utilizan para soportar el desarrollo de aplicaciones y productos finales, representando la base de las actividades del negocio de la empresa. Representan diferentes tipos de aplicación en dominios de aplicación amplios tales como las telecomunicaciones, la aviónica, educación o finanzas. Estos *frameworks* presentan un retorno sustancial de la inversión dado que soportan directamente el desarrollo de aplicaciones y productos finales.

Otra clasificación es la que considera las técnicas utilizadas para extender un *framework*. Desde esta perspectiva, los *frameworks* se clasifican en tres categorías [Yassin y Fayad, 1999]:

- *Frameworks de caja blanca o frameworks dirigidos por la arquitectura:* Profundamente ligados a las características de la orientación a objetos, tales como la herencia o la ligadura dinámica. El *framework* se extiende por herencia de sus clases base o redefiniendo sus métodos. Estos *frameworks* definen interfaces para componentes que pueden integrarse mediante composición de objetos. Sin embargo, la dificultad del uso de este tipo de *frameworks* reside en el hecho de que requieren un profundo conocimiento y comprensión de las clases a extender. Otro punto débil, propio de la herencia en general, es la dependencia entre métodos, redefinir una operación puede requerir redefinir otra, y así sucesivamente.
- *Frameworks de caja negra o frameworks dirigidos por datos:* Se estructuran utilizando composición de objetos y delegación en lugar de herencia. Enfatizan las relaciones dinámicas entre los objetos en lugar de las relaciones estáticas entre las clases. Se puede añadir nueva funcionalidad al *framework* componiendo objetos existentes de diferentes formas, para reflejar el comportamiento de la aplicación. El usuario del *framework* no tiene que conocer con profundidad los detalles de éste, sino sólo conocer cómo usar y combinar los objetos existentes. En general son más fáciles de utilizar que los *frameworks* de caja blanca. Por el contrario, los *frameworks* de caja negra son más difíciles de construir porque sus interfaces y puntos de anclaje deben anticiparse a un conjunto muy amplio de posibilidades. Debido a su flexibilidad predefinida, éstos son más rígidos dentro del dominio al que dan soporte. Un excesivo uso de la composición de objetos puede provocar diseños difíciles de comprender.
- *Frameworks de caja gris:* Es una mezcla de los dos tipos anteriores, buscando evitar sus desventajas. Permiten la extensión mediante la herencia y la ligadura dinámica, y además la propiedad de ocultar información innecesaria a los usuarios del *framework*.

5.4. Componentes de un framework

Conceptualmente, una gran parte de los autores consideran que un *framework* tiene dos componentes principales [Demeyer et al., 1997]:

- *Los contratos del framework:* La funcionalidad común dentro de un dominio específico es capturada en los contratos del *framework*. Se encargan de formalizar

exactamente qué partes del *framework* van a ser reutilizadas. Ellos imponen una estructura común para todas las aplicaciones que utilizan el mismo *framework*. La implementación y la funcionalidad de los contratos permanecen normalmente ocultas al usuario del *framework*. Sin embargo, dado que los contratos forman el esqueleto de todas las aplicaciones, tienen una influencia directa en el rendimiento y en la corrección de la aplicación. Esta parte común fija que no pretende ser cambiada recibe también el nombre de puntos congelados o *frozen spots* [Johnson y Foote, 1988], [Pree, 1995], [Schmid, 2000].

- *Puntos calientes (hot spots o hooks):* Un punto caliente es un aspecto variable dentro de un dominio de aplicación [Pree, 1994], [Pree, 1995]. Estos lugares indican dónde puede extenderse el *framework* por los usuarios del mismo. Las aplicaciones instanciadas a partir del *framework* diferirán entre sí por la forma de implementar estos puntos calientes. Éstos permiten que un usuario del *framework* inserte clases o subsistemas específicos de la aplicación a construir. En un *framework* de caja negra se seleccionarán de un conjunto de clases predefinidas aportadas por el framework, mientras que en un *framework* de caja blanca se extenderán por herencia sus clases abstractas. Hacen que un *framework* sea flexible.

Por otro lado en [Bosch, 2000a] se establece que la estructura física de un *framework* está compuesta por el diseño del núcleo del *framework* y por los incrementos internos del *framework*.

El diseño del núcleo del *framework* describe la arquitectura software de las clases de las aplicaciones en el dominio del problema e incluye las clases del dominio, tanto abstractas como concretas. Las clases concretas deben permanecer ocultas al usuario del *framework*. Los puntos calientes se soportan por clases abstractas, que pueden ser tanto visibles como invisibles para el usuario del *framework*.

Por su parte, los incrementos internos del *framework* representan clases opcionales que acompañan al *framework* para hacerlo más usable. Son clases que capturan implementaciones comunes al núcleo del *framework*. Existen dos categorías:

- *Subclases:* Representan implementaciones comunes de los conceptos capturados por las superclases.
- *Colecciones de subclases:* Representan las especificaciones para una instanciación completa del *framework* en un contexto determinado.

Opcionalmente, otro componente muy interesante con el que pueden contar los *frameworks* son los *cookbooks* o "libros de recetas" [Pree, 1994]. Son documentos que contienen instrucciones que describen las formas típicas de utilizar el *framework*. Siguiendo con la metáfora, las instrucciones individuales se denominan *recipes*, esto es "recetas". Describen informalmente la forma de utilizar el *framework* para resolver problemas específicos, quedando normalmente fuera los detalles de diseño e implementación propios del *framework*.

5.5. Desarrollo de frameworks

Los *frameworks* deben ser desarrollados desde cero. No se considera adecuado que una aplicación pueda transformarse en un *framework* de una manera directa. Los *frameworks*, como cualquier otro software que sea reutilizable, tiene que ser diseñado para ser reutilizable desde su génesis [Froehlich et al., 1997].

Sin embargo, el desarrollo de un *framework* es una tarea difícil. Como se expresa en [Gamma et al., 1995]: "*si las aplicaciones son difíciles de diseñar, y los toolkits son más difíciles, entonces los frameworks son los más difíciles de diseñar*". Parte del problema se

localiza en el conflicto inherente entre la reutilización y la construcción del software. Empaquetar *assets* que puedan ser reutilizados en diferentes contextos y diseñar arquitecturas software que sean fáciles de adaptar a los requisitos concretos de las aplicaciones finales son situaciones contradictorias. Otra dificultad radica en la comprensión del dominio del problema y en encontrar las abstracciones apropiadas [Johnson y Foote, 1991].

La mayoría de los autores coinciden en señalar que la construcción de un *framework* es esencialmente un proceso iterativo en el que el *framework* es definido, probado y refinado un número indefinido de veces [Johnson y Foote, 1991], [Booch, 1994], [Froehlich et al., 1997], [Bosch et al., 1997]. Los *frameworks* requieren, más que cualquier otro tipo de software, el cambio y la refactorización [Opdyke, 1992].

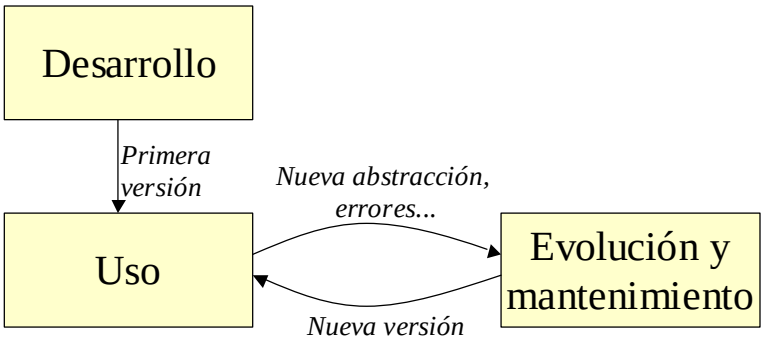


Figura 16. Proceso de desarrollo de un *framework* [Bosch et al., 1997]

En concreto, diferentes autores expresan diferentes fases en el proceso de desarrollo de un *framework*, así, por ejemplo, [Froehlich et al., 1997] distingue las fases de análisis, diseño e implementación, prueba y refinamiento; [Bosch, 2000a] habla de las actividades de análisis de dominio y variabilidad, diseño arquitectónico, diseño del *framework*, implementación del *framework*, prueba del *framework* y generación de las pruebas de las instancias. Como resumen del proceso de desarrollo de un *framework* se presenta en la Figura 16 el ciclo de fases descrito en [Bosch et al., 1997].

Para terminar este apartado se van a presentar dos características diferenciadoras en el diseño de un *framework*: los puntos de variabilidad o puntos calientes y el control entre el *framework* y las aplicaciones instancias del *framework*.

5.5.1 Puntos calientes

Los puntos calientes ofrecen el mecanismo adecuado para extender un *framework* y por tanto su diseño tiene un enorme impacto en la usabilidad del *framework* en general, y especialmente en su flexibilidad y variabilidad.

La variabilidad requerida por un punto caliente puede clasificarse según las siguientes características [Schmid, 1997]:

- La responsabilidad común (R) que generalizan las diferentes alternativas.
- Las diferentes alternativas que realizan (R_i).
- El tipo de variabilidad requerida. Puede ser considerada en alternativas con la misma interfaz y diferente implementación, o alternativas con servicio uniforme y diferente estructura...
- La multiplicidad da el número y la estructura de las alternativas que se encierran en un punto caliente.
- El tipo de ligadura representa el momento temporal en el que la alternativa es seleccionada: tiempo de aplicación o tiempo de ejecución.

Estructuralmente un punto caliente se compone de una clase base abstracta, que define la interfaz común con las responsabilidades; de diversas clases concretas derivadas, que implementan alternativas de aplicación específicas; así como de posibles clases adicionales y de sus relaciones.

En el caso de un *framework* de caja blanca, el usuario del *framework* tiene que implementar las clases concretas derivadas de la clase abstracta que define al punto caliente (Figura 17 b). Por otra parte, en el caso de un *framework* de caja negra, éste incluye todas las clases concretas, siendo el usuario del *framework* el responsable de elegir las más apropiadas y combinarlas para obtener la funcionalidad requerida en la aplicación final (Figura 17 a).

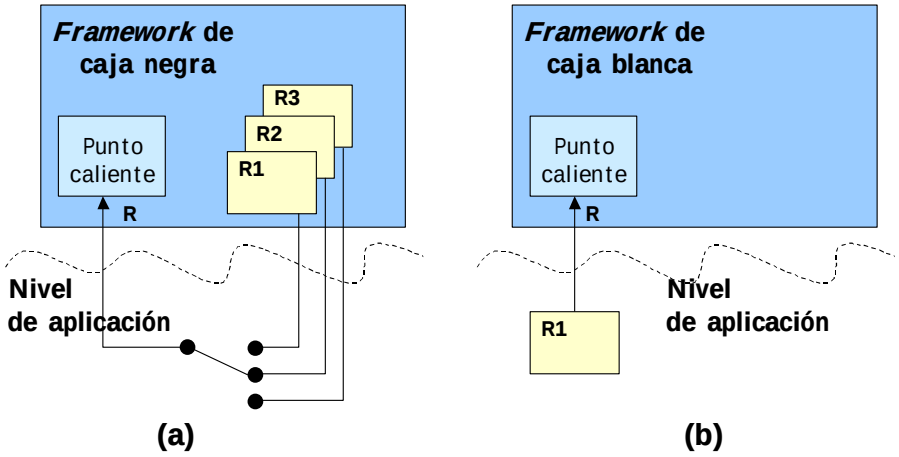


Figura 17. Mecanismos en un punto caliente

El punto caliente normalmente contiene una referencia polimórfica a la clase base. El usuario del *framework* personaliza el punto caliente enlazando dichas referencia con un objeto instancia de una clase derivada de la clase base. Este objeto puede estar incluido en un *framework* de caja negra o tener que ser construido por el usuario del *framework* extendiendo la clase base. Los métodos de la clase base son especializados en la clase derivada, de forma que cada llamada a éstos será dinámicamente enlazada, a través de esta referencia, a la implementación aportada por la subclase. De esta forma un subsistema de puntos calientes introduce una variabilidad que es normalmente transparente al resto de los integrantes del *framework*.

En el caso de los *frameworks* de caja negra la forma de implementar la variabilidad suele venir de la mano de la utilización de patrones de comportamiento como los definidos en [Gamma et al., 1995], tales como el patrón Orden (*Command Pattern*) o el patrón Estrategia (*Strategy Pattern*).

5.5.2 Control entre el *framework* y las aplicaciones instancias del *framework*

Una de las características más distintivas de un *framework* es su habilidad para hacer un uso intensivo de la ligadura tardía o dinámica. En las bibliotecas tradicionales, ya fueran orientadas a objetos o procedimentales, el código de la aplicación invoca a las rutinas incluidas en la biblioteca. En los *frameworks* orientados a objetos la situación se invierte y en una gran parte de las ocasiones es el código del *framework* el que tiene el hilo de control y llama al código de la aplicación cuando lo considera apropiado. Esta inversión del control es frecuentemente referenciada como el *Principio de Hollywood*, esto es, "*No nos llame, ya lo llamaremos nosotros*" [Johnson y Foote, 1988]. Esta situación se ilustra gráficamente en la Figura 18.

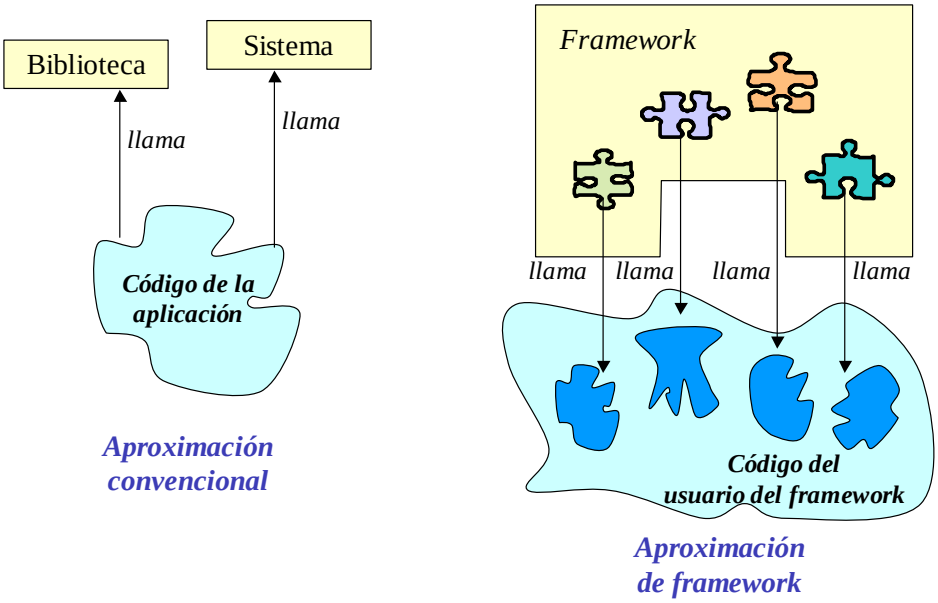


Figura 18. Inversión de control

En [Sparks et al., 1996] se hace distinción entre *frameworks* que llaman y *frameworks* que son llamados. Los primeros son entidades activas en una aplicación, controlando e invocando a otras partes. Por su parte, los *frameworks* llamados son entidades pasivas que pueden ser invocadas por otras partes de la aplicación, a semejanza de las bibliotecas de clases.

5.6. Modelos de componentes de frameworks

Hay varios factores que influyen en la óptima aproximación de modelar un *framework* como un componente dentro de una línea de productos. Uno de los factores es la cantidad de comportamiento específico del producto que se requiere del componente. Por otro lado, un factor que influye sobremanera en la organización del *framework* es su número de puntos de variación independientes. Un *framework* puede tener varios puntos calientes, pero la elección de una alternativa para uno de ellos puede restringir las elecciones de otros puntos calientes. En estos casos deben formarse conjuntos de alternativas.

En [Bosch, 2000a] se han identificado cuatro modelos de componentes de *frameworks* que pueden utilizarse en líneas de productos: *modelo de extensión de un producto específico*, *modelo de extensión de un estándar específico*, *modelo de extensión de grano fino* y *modelo basado en generadores*.

5.6.1 Modelo de extensión de un producto específico

La aproximación tradicional es utilizar los *frameworks* extendiéndolos para cada instanciación que se hace necesaria. En el caso de una línea de productos, esto tiene lugar en cada uno de los productos que lo incluyen como componente. Como el *framework* sólo cubre el comportamiento que es común a todos los productos en la línea de productos, cada producto añade una extensión propia del producto al *framework*.

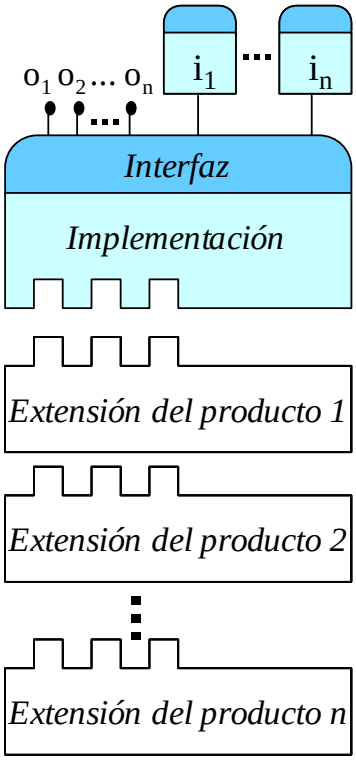


Figura 19. Modelo de extensión específico del producto [Bosch, 2000a]

En la Figura 19 este modelo se presenta gráficamente. El *framework* exporta una interfaz compuesta de un conjunto de operaciones (o_1, o_2, ..., o_n) y de un conjunto de tipos de interfaces (i_1, i_2, ..., i_n). Idealmente, la interfaz del *framework* no se ve afectada por la extensión de un producto específico. Sin embargo, la extensión o el cambio de la interfaz no se permite en todos los casos.

La principal ventaja de este modelo es su simplicidad. Dado que se desarrolla y se mantiene una única extensión por producto, se tiene una organización del software relativamente simple.

Las desventajas de este modelo son la falta de reutilización entre productos concretos y de flexibilidad. Con frecuencia los productos concretos comparten una parte significativa de sus requisitos funcionales, pero el modelo no permite la reutilización de las partes comunes entre ellos. Finalmente, el modelo es inflexible de forma que los cambios del *framework* afectan a todas las instancias del mismo.

5.6.2 Modelo de extensión de un estándar específico

En lugar de implementar una extensión del *framework* para cada producto de la línea de productos, se hace para cada elemento estándar del mismo, como por ejemplo para un sistema de ficheros o para un protocolo de comunicaciones. Cada producto generalmente incorpora una o más implementaciones del *framework*, como variantes del producto o como partes configurables del producto. Otra diferencia con respecto al modelo anterior es que la parte común del *framework* sólo define la interfaz del *framework* y nada, o muy poco, comportamiento común para las implementaciones del mismo. Este modelo se ilustra gráficamente en la Figura 20.

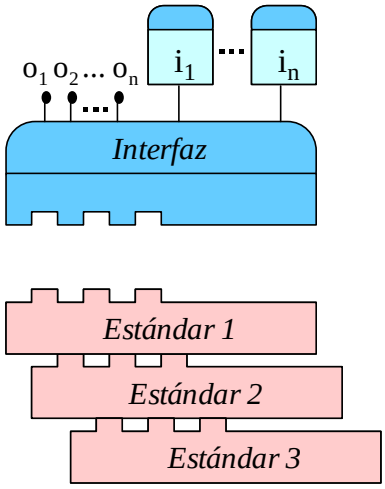


Figura 20. Modelo de extensión de un estándar específico [Bosch, 2000a]

La principal ventaja de este modelo es que ofrece una interfaz uniforme a otros componentes de la arquitectura. Por ejemplo, en una arquitectura consistente en cinco componentes completamente conectados, con tres implementaciones posibles de cada uno, se requeriría que cada implementación fuera capaz de comunicarse con las tres versiones de cada uno de los cuatro componentes restantes, esto es doce alternativas. Si se define una interfaz uniforme cada componente reduce el número de alternativas requeridas a una.

Otra ventaja de este modelo es su simplicidad. A la larga como cada implementación del *framework* se ajusta a la interfaz definida por el *framework*, éstas pueden evolucionar independientemente sin dependencias con respecto a otras implementaciones del *framework*.

Este modelo también presenta algunas desventajas. En primer lugar cabe citar la falta de reutilización entre implementaciones del *framework*, dado que la parte común del componente sólo define la interfaz, no explota el potencial de los *frameworks* orientados a objetos a este respecto. Como segundo aspecto está la falta de extensiones específicas de los productos. Por último, se tiene un descenso de la mantenibilidad, ya que este modelo no es adecuado para incorporar cambios por los componentes clientes que requieran cambios en la interfaz del componente. Cada cambio en la interfaz requiere cambios en cada implementación del *framework*. Además, como la funcionalidad compartida entre las implementaciones del *framework* está presente en cada implementación, el esfuerzo de mantenimiento se multiplica.

El modelo que se ha presentado en este subapartado es una versión extrema del modelo donde sólo se comparte la interfaz. Es posible mover alguna funcionalidad común a la parte de la interfaz del *framework*.

5.6.3 Modelo de extensión de grano fino

Los modelos presentados con anterioridad intentan extender el *framework* con una única extensión que cubra todos los puntos de variación del *framework*. Estos modelos pagan su simplicidad aparente con una falta de flexibilidad y de reusabilidad entre las extensiones del *framework*. El modelo de extensión de grano fino toma una posición radicalmente opuesta, su objetivo es ofrecer un conjunto de pequeños módulos de extensión, cada uno de los cuales cubre uno o unos pocos puntos de variación, pudiéndose ser ellos mismos configurables.

El *framework* común consiste de una interfaz y de una implementación común a todas las instancias. Para cada punto caliente, hay un conjunto de extensiones genéricas y éstas pueden ser configuradas con extensiones específicas de los productos. Este modelo se expresa gráficamente en la Figura 21.

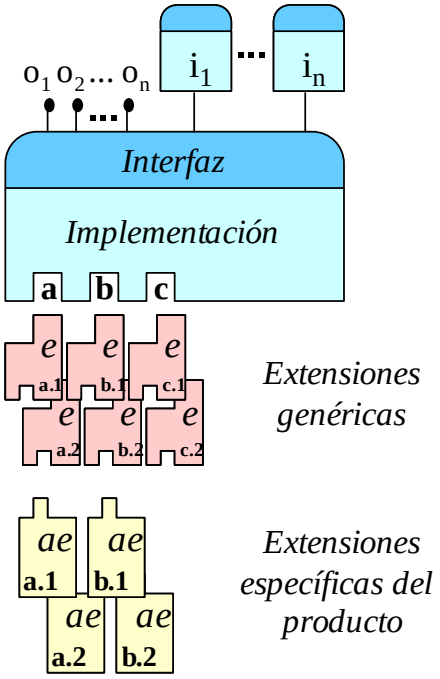


Figura 21. Modelo de extensión de grano fino [Bosch, 2000a]

Este tipo de modelo indica un *framework* normalmente más maduro que los presentados en los casos anteriores, porque requiere de una comprensión más detallada de los puntos calientes y de una definición ortogonal de estos puntos. Ejemplos típicos de este modelo se pueden encontrar en los *frameworks* comerciales para la construcción de interfaces gráficas de usuario, tales como MFC (*Microsoft Foundation Classes*) o *Smalltalk-80 VisualWorks*. Estos *frameworks* normalmente consisten de un motor y de un conjunto de componentes de caja negra que pueden ser configurados con pequeños componentes específicos de la aplicación.

Las principales ventajas de este modelo se encuentran en la gran flexibilidad de configuración y en la alta reusabilidad de sus extensiones. Comparado con los modelos anteriores, la flexibilidad es considerablemente mejor debido a los componentes de extensión de grano fino, que generalmente son clases individuales, y a la independencia entre los componentes de extensión. El usuario del *framework* es libre para componer conjuntos arbitrarios de componentes de extensión, aunque no todas las combinaciones pueden ser semánticamente significativas o correctas. La reusabilidad de los componentes de extensión es mucho mayor gracias a su relativa independencia y atomicidad. Sin embargo, un peligro puede venir de la naturaleza inherente de grano fino de estos componentes de extensión; la reutilización de unos componentes demasiado pequeños no es efectiva en cuanto al coste, ya que entender su funcionalidad y su lugar en el *framework* puede tomar más esfuerzo que crear un componente de extensión desde cero.

La principal desventaja de este modelo es su alta complejidad. Mientras que los modelos anteriormente presentados eran relativamente sencillos de usar, este modelo es mucho más complejo de utilizar, dependiendo del número de puntos calientes, el número de componentes de extensión y el número y la complejidad de la relación entre ellos. Una segunda desventaja es que las extensiones son de grano fino, existiendo relaciones entre las extensiones para diferentes puntos de variación.

5.6.4 Modelo basado en generadores

Este modelo es considerablemente diferente a los tres anteriores. Es una extensión del modelo de extensión de grano fino. La utilización de un soporte basado en herramientas puede ser

considerada como una opción a las desventajas presentadas anteriormente. Aunque hay diferentes aproximaciones, por ejemplo [Roberts y Jonson, 1996], todas tienen en común basarse en algún tipo de generador. Las dos principales tendencias son una herramienta de configuración gráfica en la que los componentes se configuran con extensiones de componentes disponibles, o la utilización de lenguajes específicos del dominio que se utilizan para especificar una configuración y generar un componente congruente con ella.

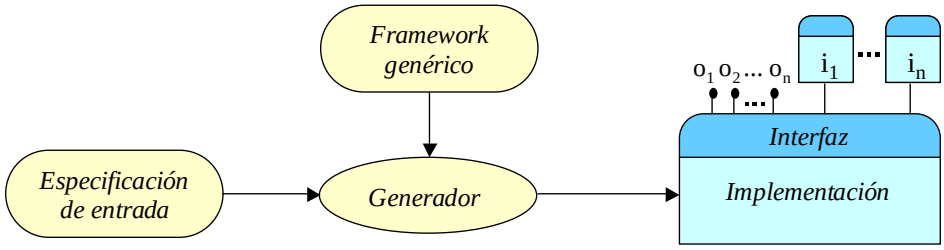


Figura 22. Modelo basado en generadores [Bosch, 2000a]

Las ventajas de este modelo son una combinación de todas las que presentan los modelos anteriores, esto es, una alta reusabilidad y flexibilidad, así como una integridad lógica de las extensiones. La reusabilidad y la flexibilidad se logran al utilizar las mismas extensiones de grano fino que en el modelo anterior, pero con la ventaja añadida de que las herramientas utilizadas proveen al usuario del *framework* de un chequeo de la semántica. Un generador simplifica el uso de un *framework* con extensiones de grano fino.

Como desventajas cabe citar el descenso de la capacidad de evolución y aumenta la complejidad de las extensiones de los productos específicos. La evolución del *framework* es más cara porque unido a cada cambio no trivial del *framework* tiene que cambiarse la herramienta. Las extensiones específicas de los productos son difíciles de incorporar. Además, la herramienta de configuración generalmente no permite instanciaciones radicalmente diferentes a las inicialmente contempladas por el diseñador de la herramienta.

6. Mecanos

Un mecano es un elemento software reutilizable de grano grueso, compuesto por un conjunto de elementos reutilizables de grano fino, *assets*, cada uno de los cuales está clasificado en uno de los tres niveles de abstracción soportados en el modelo: requisitos, diseño e implementación.

Los *assets* componentes de un mecano están relacionados entre sí por un conjunto de relaciones estructurales definidas en el Modelo de Mecano [García 2000], donde se distinguen dos categorías de relaciones estructurales: ***relaciones intranivel*** – aquéllas que relacionan *assets* clasificados en un mismo nivel de abstracción, estando soportados los siguientes tipos de relaciones intranivel: agregación, composición, uso, extensión y asociación – y ***relaciones internivel*** – aquéllas que relacionan *assets* clasificados en diferentes niveles de abstracción, estando soportado sólo un tipo de relaciones internivel, la reificación.

Además, existe una restricción de obligado cumplimiento para que un elemento software reutilizable de grano grueso sea considerado como un mecano, y es que debe existir al menos una relación internivel en su estructura, lo que implica que al menos dos niveles de abstracción están presentes en dicho elemento reutilizable.

Opcionalmente, un mecano puede incluir uno o varios descriptores funcionales. En el Modelo de Mecano, un descriptor funcional es un conjunto de enlaces a aquellos *assets* de descripción de requisitos funcionales que el desarrollador para reutilización ha querido destacar [García, 2000], de forma que cada uno de estos requisitos es un *asset* componente del mecano. Un *asset* perteneciente a un descriptor funcional puede relacionarse con otros mediante las

relaciones intranivel soportadas. A su vez un descriptor funcional se asocia a otros *assets* del nivel de análisis y por reificación a assets de los niveles de diseño e implementación.

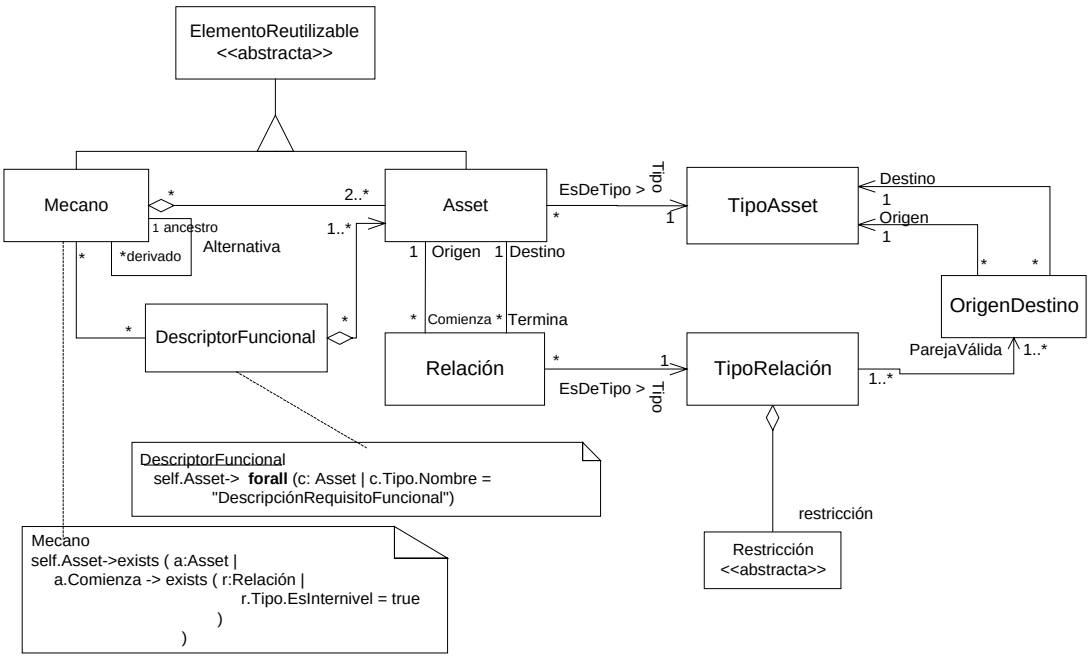


Figura 23. Modelo de Mecano

El núcleo central de la estructura de los mecanos como elementos reutilizables de grano grueso se muestra en la Figura 23, utilizando UML 1.x [OMG, 2001] como lenguaje de modelado.

El repaso que aquí se ha realizado del Modelo de Mecano se ha centrado en su perspectiva técnica, presentada desde un punto de vista semi-formal[4], aunque dicho modelo incluye además un submodelo de proceso y submodelo de cualificación [García et al., 1998], [García, 2000].

7. Mecanos como soporte a las líneas de productos

Como se desprende de la Figura 6, la perspectiva técnica que ofrece una línea de productos desde un punto de vista de grano fino, una línea de productos no es más que un conjunto de *assets* interrelacionados, donde claramente se diferencian los tres niveles de abstracción presentes en el Modelo de Mecano: *requisitos*, donde se expresan el modelo de negocio, los requisitos de la línea de productos y su grafo de variabilidad; *diseño*, donde se recoge la arquitectura de la línea de productos; e *implementación*, donde aparecen los componentes que satisfacen las restricciones impuestas por la arquitectura [García et al., 2001].

Con esta aproximación cualquier producto perteneciente a una línea de productos puede convertirse en un elemento reutilizable de grano grueso modelado como un mecano y almacenarse en un repositorio adecuado al tratamiento de mecanos.

A la hora de llevar a la práctica esta aproximación, el paso inicial pasa por representar la definición de la línea de productos, esto es, una línea de productos básica formada por una especificación y la arquitectura base de la línea de productos, mediante un mecano.

Partiendo de esta línea base, se empezará el desarrollo de nuevos productos que nutrirán la línea de productos (como ya se indicó anteriormente, los productos de la línea de productos son

[4] Existe una definición formal del mismo en [García, 2000].

activos reutilizables de la misma), incorporando estos productos en el repositorio de reutilización en forma de mecano. Se inicia así el ciclo de reutilización formado por la parte de ingeniería de dominio y por la parte de ingeniería de aplicación.

En la Figura 24 se presentan, de forma esquemática, las relaciones entre los conceptos de línea de productos y mecanos, donde un mecano representa tanto a la línea básica como a cualquier producto de la línea, gracias a su definición como elemento reutilizable de grano grueso. Además, y por añadidura, esta característica incide en el hecho de contar con un proceso de introducción y recuperación de mecanos/productos en y desde el repositorio de reutilización basado en la información de clasificación que ofrece tanto el dominio como la línea de productos.

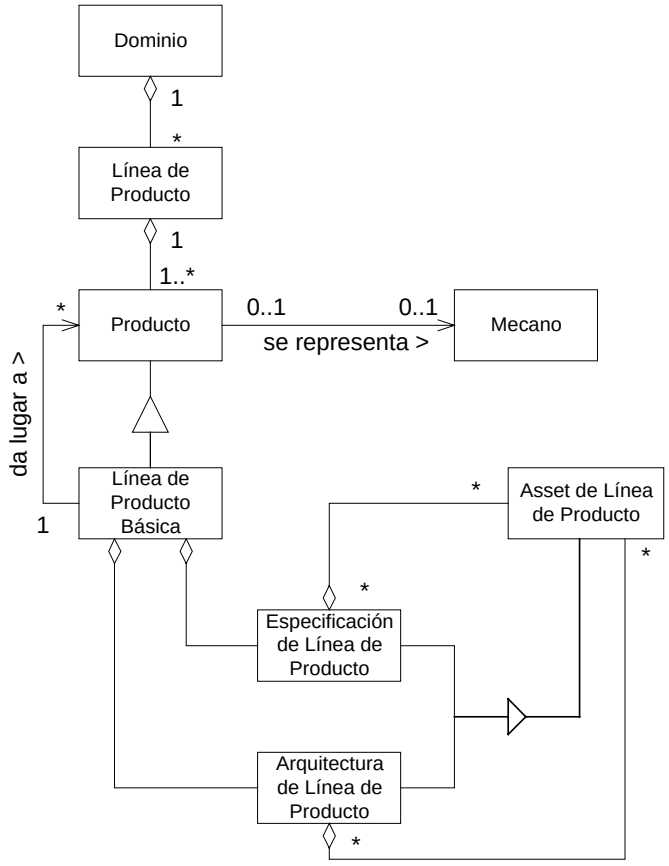


Figura 24. Dominios, líneas de productos y mecanos [García, 2000]

Otro hecho muy significativo, que potencia en gran medida la relación del Modelo de Mecano con la ingeniería de dominio en general y con las líneas de productos en particular, es la presencia en los mecanos de descriptores funcionales que van a permitir, gracias a las relaciones intranivel, modelar los grafos de características (*feature graphs*) propios de los métodos de ingeniería de dominio [Kang et al., 1998] y que representan propiedades de las características como su variabilidad o su exclusión mutua.

8. Conclusiones

En este trabajo se ha hecho un recorrido por algunos de los tipos de elementos reutilizables que han contribuido de alguna manera al aumento de la granularidad y el nivel de abstracción en la unidad de reutilización.

En todas ellas se ha buscado un enfoque de ingeniería de dominio o de desarrollo para reutilización. Así mismo se ha intentado potenciar la relación de todas ellas con el concepto de línea de productos, como catalizador de lo que podía denominarse un proceso de reutilización organizado y maduro.

Cabe destacar de una manera especial el soporte natural que ofrece el Modelo de Mecano a las líneas de productos, gracias a su alcance de proyecto y a su definición basándose en una serie de relaciones estructurales entre sus componentes: elementos reutilizables de menor granularidad.

9. Referencias

[Andert, 1994] Andert, G. (1994). "*Object Frameworks in the Taligent OS*". Proceedings of Compcon 94. Los Alamitos, CA. IEEE CS Press.

[Arango y Prieto-Díaz, 1991] Arango, G. y Prieto-Díaz, R. (1991). "*Domain Analysis Concepts and Research Directions*". *Domain Analysis and Software Systems Modeling*. R. Prieto-Díaz y G. Arango editores. Páginas 9-32. IEEE Computer Society Press.

[Bachman et al., 2000] Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R. y Wallnau, K. (2000). "*Volume II: Technical Concepts of Component-Based Software Engineering*". Technical Report CMU/SEI-2000-TR-008 (ESC-TR-2000-007), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Bass et al., 1997] Bass, L., Clements, P., Cohen, S., Northrop, L. M. y Withey, J. (1997). "*Product Line Practice Workshop Report*". Technical Report CMU/SEI-97-TR-003 (ESC-TR-97-003), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Bass et al., 1998a] Bass, L., Chastek, G., Clements, P., Northrop, L. M., Smith, D. y Withey, J. (1998) "*Second Product Line Practice Workshop Report*". Technical Report CMU/SEI-98-TR-015 (ESC-TR-98-015), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Bass et al., 1998b] Bass, L., Clements, P. y Kazman, R. (1998). "*Software Architecture in Practice*". Addison-Wesley Longman.

[Bass et al., 1999] Bass, L., Campbell, G., Clements, P., Northrop, L. M. y Smith, D. (1999). "*Third Product Line Practice Workshop Report*". Technical Report CMU/SEI-99-TR-003 (ESC-TR-99-003), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Bass et al., 2000] Bass, L., Clements, P., Donohoe, P., McGregor, J. y Northrop, L. (2000). "*Fourth Product Line Practice Workshop Report*". Technical Report CMU/SEI-2000-TR-002 (ESC-TR-2000-002), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Beck y Cunningham, 1989] Beck, K. y Cunningham, W. (1989). "*A Laboratory for Teaching Object-Oriented Thinking*". Proceedings of the 1989 OOPSLA - Conference proceedings on Object-Oriented Programming Systems, Languages and Applications (October 2 - 6, 1989, New Orleans, LA USA); Reprinted in Sigplan Notices, 24(10):1-6.

[Booch, 1994] Booch, G. (1994). "*Designing an Applications Framework*". Dr. Dobb's Journal, 19(2):24-32.

[Bosch et al., 1997] Bosch, J., Molin, P., Mattson, M. y Bengtson, P. (1997). "*Object-Oriented Frameworks – Problems & Experiences*". Proceedings of the 23rd International

Conference in Technology of Object-Oriented Languages and Systems, TOOLS'97 USA, (Santa Barbara, California, July 28 – August 1, 1997). Pages 203-214.

[Bosch, 2000a] Bosch, J. (2000). "*Design & Use of Software Architectures. Adopting and Evolving a Product-Line Approach*". Addison-Wesley.

[Bosch, 2000b] Bosch, J. (2000). "*Product Line Architectures*". ObjectiveView, (4):13-18. http://www.ratio.co.uk

[Bray et al., 2000] Bray, T., Paoli, J. y Sperberg-McQueen, C. M. (editors). (2000). "*Extensible Markup Language (XML) 1.0 (Second Edition)*". World Wide Web Consortium Recommendation 6 October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[Brown, 1999] Brown, A. (1999). "*Building Systems from Pieces with Component-Based Software Engineering*". Whitepaper. Sterling Software. July, 1999.

[Clements, 1997] Clements, P. (1997). "*Report of the Reuse and Product Lines Working Group of WISR8*". Special Report CMU/SEI-97-SR-010, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Clements et al., 1998] Clements, P., Northrop, L. M., Bachmann, F., Bass, L., Bergey, J., Chastek, G., Cohen, S., Donohoe, P., Jones, L., Krut, R., Little, R., Smith, D., Tilley, S., Weiderman, N., Withey, J. y Woods, S. (1998). "*A Framework for Software Product Line Practice – Version 1.0*". Product Line Systems Program, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Cohen et al., 1995] Cohen, S. G., Friedman, S., Martin, L., Solderitsch, N. y Webster, R. (1995). "*Product Line Identification for ESC-Hanscom*". Special Report CMU/SEI-95-SR-024, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[Cohen y Northrop, 1998] Cohen, S. G. y Northrop, L. M. (1998). "*Object-Oriented Technology and Domain Analysis*". Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 86-93. 2 - 5 June, 1998, Victoria, B.C., Canada. IEEE-CS.

[Cotter y Potel, 1995] Cotter, S. y Potel, M. (1995). "*Inside Taligent Technology*". Addison-Wesley.

[Crespo, 2000] Crespo González-Carvajal, Yania. (2000). "*Incremento del Potencial de Reutilización del Software mediante una Refactorización para Parametrizar*". Tesis Doctoral. Universidad de Valladolid.

[Cheesman, 1998] Cheesman, J. (1998). "*What Is a Component*". http://www.cbdedge.com/cbdweb/technology/whatComp.html.

[Demeyer et al., 1997] Demeyer, S., Meijler, T. D., Nierstrasz, O. y Steyaert, P. (1997). "*Design Guidelines for 'Tailorable' Frameworks*". Communications of the ACM, 40(10):60-64.

[Doscher y Hodges, 2000] Doscher, D. y Hodges, R. (2000). "*SEMATECH CIM Framework*". *Domain-Specific Application Frameworks. Frameworks Experience by Industry*, Fayad, E. M. y Johnson, R. E. (editores). Capítulo 2, páginas 7-19. Wiley & Sons.

[D'Souza y Wills, 1999] D'Souza, D. F. y Wills, A. C. (1999). "*Objects, Components and Frameworks with UML. The Catalysis Approach*". Object Technology Series. Addison-Wesley.

[Fayad et al., 1998] Fayad, E. M., Schmidt, D. C. y Johnson, R. E. (1998). "*Applications Frameworks*". *Building Application Frameworks. Object-Oriented Foundations of Framework design*, Fayad, E. M., Schmidt, D. C. y Johnson, R. E. (editores). Capítulo 1, páginas 3-27. Wiley & Sons.

[Fayad et al., 1999] Fayad, E. M., Schmidt, D. C. y Johnson, R. E. eds. (1999). "*Implementing Applications Frameworks: Object-Oriented Frameworks at Work*". Wiley & Sons.

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

[Froehlich et al., 1997] Froehlich, G., Hoover, H. J., Liu, L. y Sorenson, P. (1997). "*Designing Object-Oriented Frameworks*". Technical Report. University of Alberta.

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

[García, 2000] García Peñalvo, F. J. (2000). "*Modelo de Reutilización Soportado por Estructuras Complejas de Reutilización Denominadas Mecanos*". Tesis Doctoral. Facultad de Ciencias, Universidad de Salamanca. Enero, 2000.

[García et al., 1998] García, F. J., Marqués, J. y Maudes, J. (1998). "*Mecanos as Basis of a Compositional/Generative Mixed Reuse Model*". Proceedings of the 2nd ed. of the European Reuse Workshop. Madrid - Spain, Vol. 2, 17-20.

[García et al., 2001] García, F. J., Laguna, M. Á., Marqués, J. M., Moreno, Mª N. y Hernández, J. A. (2001). "*Mecanos como Soporte a las Líneas de Productos*". Actas de las I Jornadas de Trabajo DOLMEN (Sevilla, 12 y 13 de junio de 2001). Páginas 93-102.

[García-Molina et al., 2000] García-Molina, J., Hernández, J., Sánchez, F., Sevilla, D. y Vallecillo, A. (2000). "*Componentes Software: Definiciones y Propiedades*". Actas del Primer Taller de Trabajo en Ingeniería del Software basada en Componentes Distribuidos - IScDIS'00, desarrollado dentro de las V Jornadas de Ingeniería del Software y Bases de Datos, JISBD'2000 (Valladolid, 8-10 de Noviembre de 2000). J. García-Molina, J. Hernández, F. Sánchez, D. Sevilla y A. Vallecillo (Editores). Informe Técnico nº TR-12/2000. Dpto. de Informática. Universidad de Extremadura. (http://webepcc.unex.es/juan/iscdis00/). Páginas 121-129. Diciembre, 2000.

[Goldberg y Robson, 1989] Goldberg, A., y Robson, D. (1989). "*Smalltalk-80 – The Language*". Addison-Wesley.

[Griss, 1993] Griss, M. L. (1993). "*Software Reuse: From Library to Factory*". IBM System Journal, 32(4):1-23. November.

[Griss, 1996a] Griss, M. L. (1996). "*Domain Engineering and Variability in the Reuse-Driven Software Engineering Business*". Object Magazine, 6(7).

[Griss, 1996b] Griss, M. L. (1996). "*Systematic Software Reuse: Architecture, Process and Organization are Crucial*". Fusion Newsletter. http://hpl.hp.com/reuse/papers/fusion1.htm.

[Griss, 2000] Griss, M. L. (2000). "*Implementing Product-Line Features by Composing Component Aspects*". Proceedings of First International Software Product Line Conference. Denver, CO (USA). August, 2000.

[Griss et al., 1998] Griss, M. L., Favaro, J. y d'Alessandro, M. (1998). "*Integrating Feature Modeling with RSEB*". Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 76-85. 2 - 5 June, 1998, Victoria, B.C., Canada. IEEE-CS.

[Griss y Wentzel, 1994] Griss, M. L. y Wentzel, K. D. (1994). "*Hybrid Domain-Specific Kits for a Flexible Software Factory*". Proceedings of SAC'94. Reuse and Reengineering Track, páginas 47-52, New York. ACM.

[Griss y Wentzel, 1995] Griss, M. L. y Wentzel, K. D. (1995). "*Hybrid Domain-Specific Kits*". The Journal of Systems and Software, 30(3):213-230. Special Issue on Software Reusability.

[Hopkins, 2000] Hopkins J. (2000). "*Component Primer*". Communications of the ACM, 43(10):27-30.

[Hüni et al., 1995] Hüni, H., Johnson, R. y Engel, R. (1995). "*A Framework for Network Protocol Software*". Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications – OOPSLA'95. (October 15 - 19, 1995, Austin, TX USA). Páginas 358-369.

[Jacobson et al., 1992] Jacobson, I., Christerson, M., Jonsson, P. y Övergaad, G. (1992). "*Object Oriented Software Engineering: A Use Case Driven Approach*". Addison-Wesley.

[Jacobson et al., 1994] Jacobson, I., Ericsson, M. y Jacobson, A.. (1994). "*The Object Advantage - Business Process Reengineering with Object Technology*", Addison-Wesley, Menlo Park, CA.

[Jacobson et al., 1997] Jacobson I., Griss M. y Jonsson P. (1997). "*Software Reuse. Architecture, Process and Organization for Business Success*". ACM Press. Addison Wesley Longman.

[Jezequel y Pacherie, 2000] Jezequel, J. y Pacherie, J. (2000). "*EPEE: A Framework for Supercomputing*". *Domain-Specific Application Frameworks. Frameworks Experience by Industry*, Fayad, E. M. y Johnson, R. E. (editores). Capítulo 14, páginas 251-279. Wiley & Sons.

[Johnson, 1992] Johnson, R. (1992). "*Documenting Frameworks with Patterns*". Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications – OOPSLA'92. (October 18 - 22, 1992, Vancuver, Canada). Páginas 63-76.

[Johnson y Foote, 1988] Johnson, R. y Foote, B. (1988). "*Design Reusable Classes*". Journal of Object-Oriented Programming (JOOP), 1(2):22-35.

[Johnson y Foote, 1991] Johnson, R. y Foote, B. (1991). "*Reusing Object-Oriented Design*". Technical Report UIUCDCS 91-1696, University of Illinois.

[Johnson y Opdyke, 1993] Johnson, R. y Opdyke, W. F. (1993). "*Refactoring and Aggregation*". Proceedings of ISOTAS'93: International Symposium on Object Technologies for Advanced Software.

[Kang, 1998] Kang, K. C. (1998). "*Feature-Oriented Development of Applications for a Domain*". Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 354-355. 2 - 5 June, 1998, Victoria, B.C., Canada. IEEE-CS.

[Kang et al., 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. y Peterson, A. S. (1990). "*Feature-Oriented Domain Analysis (FODA). Feasibility Study*". Technical Report CMU/SEI-90-TR21 (ESD-90-TR-222), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.

[Kang et al., 1998a] Kang, K. C., Kim, S., Lee, J. y Kim, K. (1998). "*FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*". Annals of Software Engineering, 5:143-168.

[Kang et al., 1998b] Kang, K. C., Kim, S., Lee, J., Kim, K. y Shim, E. (1998). "*Feature-Oriented Software Engineering for the Electronic Bulletin Board System (EBBS) Domain*". Proceedings of the Third World Conference on Integrated Design and Process Technology (IDPT'98).

[Klingler y Solderitsch, 1996] Klingler, C. D. y Solderitsch, J. (1996). "*DAGAR: A Process for Domain Architecture Definition and Asset Implementation*". Proceedings of the Annual International Conference on ADA (TriAda'96), páginas 231-245, December 3-7, 1996, Philadelphia, PA (USA). ACM, ACM Press.

[Knauber y Succi, 2001] Knauber, P. y Succi, G. "*Perspectives on Software Product Lines*". ACM Software Engineering Notes, 26(2):29-33. March, 2001.

[Lee et al., 2000] Lee, K., Kang, K. C., Chae, W. y Choi, B. W. (2000). "*Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse*". Software: Practice and Experience, 30(9):1025-1046.

[Linton et al., 1989] Linton, M. A., Vlissides, J. M. y Calder, P. R. (1989). "*Composing User Interfaces with Interviews*". IEEE Computer, 22(2).

[McIlroy, 1976] McIlroy, M. D. (1976). "*Mass-Produced Software Components*". Software Engineering Concepts and Techniques; 1968 NATO Conference on Software Engineering. J. M. Buxton, P. Naur y B. Randell editores. Páginas 88-98. Van Nostrand Reinhold.

[McInnis, 2000] McInnis, K. (2000). "*Component-Based Development. The Concepts, Technology and Methodology*". Castek Software Factory Inc. http://www.cbd-hq.com.

[Microsoft, 1995] Microsoft. (1995). "*The Component Object Model Specification*". Version 0.9. October, 1995.

[Microsoft, 1997] Microsoft. (1997). "*Microsoft Repository Documentation*". http://www.microsoft.com/repository.

[Microsoft, 2001] Microsoft. (2001). "*.NET Framework Developer's Guide*". http://msdn.microsoft.com/.

[Molin y Ohlsson, 1998] Molin, P. y Ohlsson, L. (1998). "*Points & Deviations – A Pattern Language for Fire Alarm Systems*". *Patterns Languages of Program Design 3*. R. Martin, D. Riehle y F. Buschmann editores. Capítulo 24, páginas 431-445. Addison-Wesley.

[Neighbors, 1984] Neighbors, J. M. (1984). "*The Draco Approach to Constructing Software from Reusable Components*". IEEE Transactions on Software Engineering, SE-10(5):564-574.

[Oberndorf, 1998] Oberndorf, P. (1998). "*COTS and Open Systems*". SEI Monographs on the Use of Commercial Software in Government Systems. Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA).

[OMG, 1999] OMG. (1999). "*CORBA Components - Volume I*". Object Management Group Inc. http://www.omg.org/cgi-bin/doc?orbos/99-07-01.

[OMG, 2001] OMG. (2001). "*OMG Unified Modeling Language Specification. Version 1.4*". Object Management Group Inc. http://www.celigent.com/omg/umlrtf/artifacts.htm. [Última vez visitado, 01/09/2001]. September, 2001.

[Opdyke, 1992] Opdyke, W. F. (1992). "*Refactoring Object-Oriented Frameworks*". PhD thesis, University of Illinois.

[Opdyke y Johnson, 1993] Opdyke, W. F. y Johnson, R. (1993). "*Creating Abstract SuperClasses by Refactoring*". Proceedings of CSC'93: The ACM 1993 Computer Science Conference. February, 1993.

[Poulin, 1997] Poulin, J. S. (1997). "*Measuring Software Reuse*". Addison-Wesley.

[Pree, 1994] Pree, W. (1994). "*Meta Patterns – A Means for Capturing the Essential of Reusable Object-Oriented Design*". Proceedings of the 8th European Conference on Object-Oriented Programming. Bologna, Italy.

[Pree, 1995] Pree, W. (1995). "*Design Patterns for Object-Oriented Software Development*". Addison-Wesley.

[Roberts y Johnson, 1996] Roberts, D. y Johnson, R. (1996). "*Evolving Frameworks. A Pattern Language for Developing Object-Oriented Frameworks*". Proceedings of the Third Annual Conference on the Patterns Languages of Programs (PLoP'96).

[Russo, 1990] Russo, V. F. (1990). "*An Object-Oriented Operating System*". PhD thesis, University of Illinois.

[Schmid, 1997] Schmid, H. A. (1997). "*Systematic Framework Design by Generalization*". Communications of the ACM, 40(10):48-51.

[Schmid, 2000] Schmid, H. A. (2000). "*OSEFA: Framework for Manufacturing*". *Domain-Specific Application Frameworks. Frameworks Experience by Industry*, Fayad, E. M. y Johnson, R. E. (editores). Capítulo 4, páginas 43-65. Wiley & Sons.

[Schmucker, 1986] Schmucker, K. J. (1986). "*Object-Oriented Programming for the Macintosh*". Hayden Book Co.

[SEI, 2001] Software Engineering Institute. (2001). "*A Framework for Software Product Line – Version 3.0*". Product Line Systems Program, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA). http://www.sei.cmu.edu/plp/framework.html.

[SEMATECH, 1998] SEMATECH. (1998). "*Computer Integrated Manufacturing (CIM) Framework Specification. Version 2.0*". Technology Transfer # 93061697J-ENG. International SEMATECH, Inc. http://www.sematech.org. January 31, 1998.

[Sevilla, 2000] Sevilla, D. (2000). "*CORBA & Components*". Actas del Primer Taller de Trabajo en Ingeniería del Software basada en Componentes Distribuidos - IScDIS'00, desarrollado dentro de las V Jornadas de Ingeniería del Software y Bases de Datos, JISBD'2000 (Valladolid, 8-10 de Noviembre de 2000). J. García-Molina, J. Hernández, F. Sánchez, D. Sevilla y A. Vallecillo (Editores). Informe Técnico nº TR-12/2000. Dpto. de Informática. Universidad de Extremadura. (http://webepcc.unex.es/juan/iscdis00/). Páginas 3-43. Diciembre, 2000.

[Simos, 1995] Simos, M. (1995). "*Organization Domain Modeling (ODM): Formalizating the Core Domain Modeling Life Cycle*". Symposium on Software Reusability. April, 1995.

[Simos et al., 1996] Simos, M., Creps, D., Klingler, C., Levine, L. y Allemang, D. (1996). "*Organization Domain Modeling (ODM) Guidebook – Version 2.0*". Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, 9255 Wellington Road Manassas, VA 22110-4121.

[Sonnemann, 1995] Sonnemann, R. (1995). "*Exploratory Study of Software Reuse Factors*". PhD thesis, George Mason University, Fairfax, Virginia, Spring.

[Sparks et al., 1996] Sparks, S., Benner, K. y Faris, C. (1996). "*Managing Object-Oriented Framework Reuse*". IEEE Computer, 29(9):52-61.

[Sparling, 2000] Sparling, M. (2000). "*Lessons Learned through Six Years of Component-Based Development*". Communications of the ACM, 43(10):47-53. October, 2000.

[Sun, 2000] Sun Microsystems. (2000). "Enterprise JavaBeans™ Specification, Version 2.0". http://java.sun.com. October, 2000.

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

[Szyperski, 2000] Szyperski, C. (2000). "*Components versus Objects*". ObjectiveView, (5):8-16. http://www.ratio.co.uk.

[Szyperski y Pfister, 1997] Szyperski, C. y Pfister, C. (1997). "*Component-Oriented Programming*". M. Muhlhauser, editor, *Special Issues in Object-Oriented Programming - ECOOP96 Workshop Reader*. Dpunkt Verlag, Heidelberg.

[Taligent, 1993] Taligent Inc. (1993). "*Leveraging Object-Oriented Frameworks*".Taligent White Paper.

[Taligent, 1994] Taligent Inc. (1994). "*Building Object-Oriented Frameworks*". Taligent White Paper.

[Thai y Oram, 1999] Thai, T. L. y Oram, A. (1999). "*Learning DCOM*". O'Reilly & Associates.

[Vici y Argentieri, 1998] Vici, A. D. y Argentieri, N. (1998). "*FODAcom: An Experience with Domain Analysis in Italian Telecom Industry*". Proceedings of the Fifth International Conference on Software Reuse, ICSR-5, páginas 166-175. 2 - 5 June, 1998, Victoria, B.C., Canada. IEEE-CS.

[Weck, 1997] Weck, W. (1997). "*Independently Extensible Component Frameworks*". M. Muhlhauser, editor, *Special Issues in Object-Oriented Programming - ECOOP96 Workshop Reader*. Dpunkt Verlag, Heidelberg.

[Weck et al., 1997] Weck, W., Bosch, J. y Szyperski, C. (1997). "*Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*". TUCS General Publication Nº5, September 1997 Turku: Turku Center for Computer Science.

[Weinand et al., 1989] Weinand, A., Gamme, E. y Marty, R. (1989). "*Design and Implementation of ET++, a Seamless Object-Oriented Application Framework*". Structured Programming, 10(2).

[Wirfs-Brock y Johnson, 1990] Wirfs-Brock, R. J. y Johnson, R. E. (1990). "*Surveying Current Research in Object-Oriented Design*". Communications of the ACM, 33(9):105-124.

[Yassin y Fayad, 1999] Yassin, A. y Fayad, M. E. (1999). "*Application Frameworks: A Survey*". *Domain-Specific Application Frameworks. Frameworks Experience by Industry*, Fayad, E. M. y Johnson, R. E. (editores). Capítulo 29, páginas 615-632. Wiley & Sons.